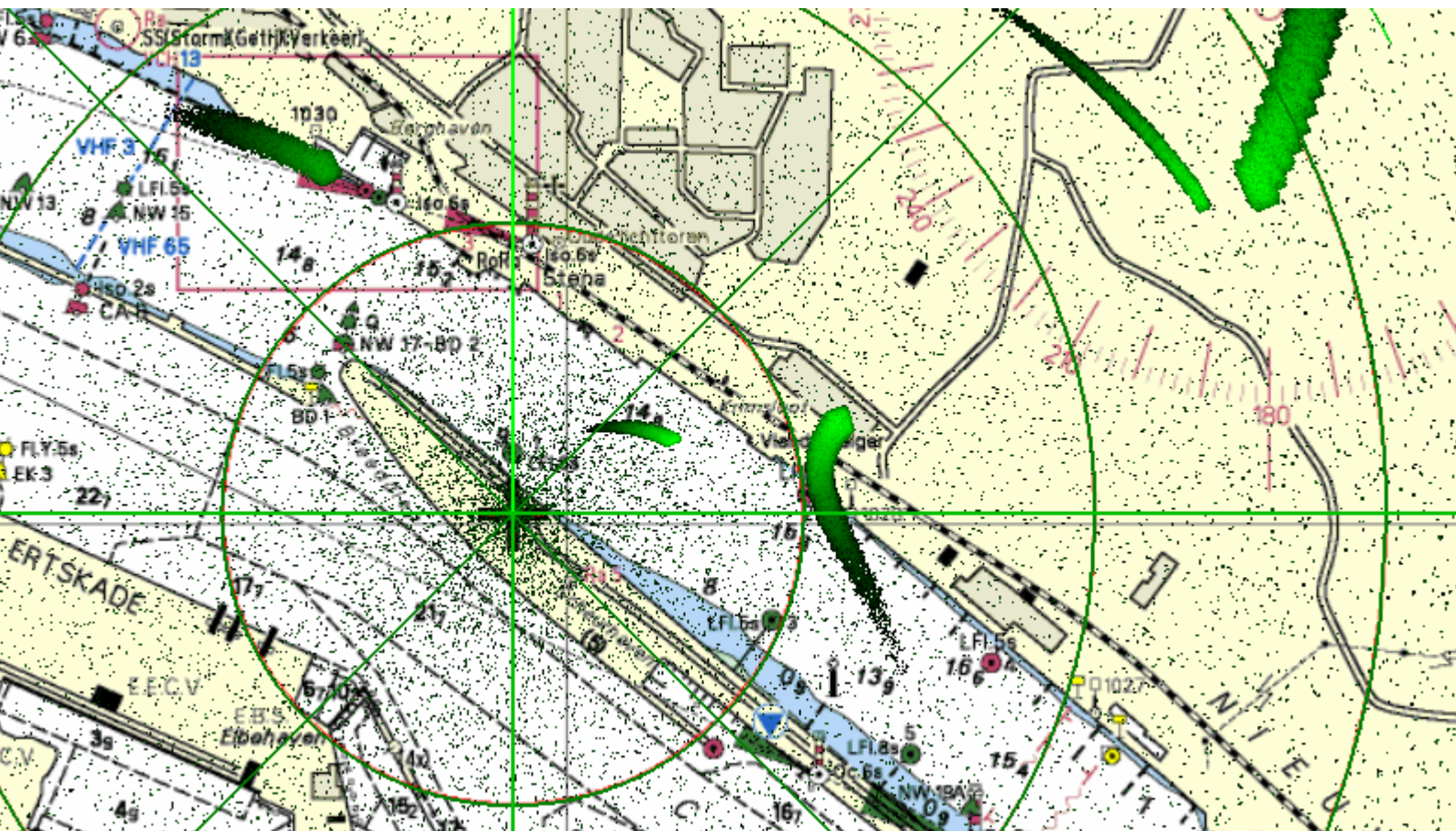


Realtime Softwarematige Radarscan Conversie Met GPU Programmeren



Student	Riccardo Sirchia
Nummer	0011177
Datum	17-11-2006
Commissieleden Universiteit Twente	Dr. J. Zwiers Dr. M. Poel Ing. H. Hondorp
Commissieleden Thales Nederland BV	Ir. R. Boers Ing. S. van Os

Samenvatting

De programmeerbaarheid van de huidige generatie GPUs biedt de mogelijkheid om zelf algoritmen hierop uit te voeren. Hiermee zijn filtermethoden ontwikkeld, die het verlies van radarsamples bij radarscan conversie tegen gaan. Daarnaast is de functionaliteit van de op dit moment gebruikte radarscan convertor (RSC-TB) geëvenaard en op enkele punten zelfs overtroffen met een volledig op de GPU uitgevoerde radarscan convertor (RSC-GPU). Deze biedt tevens meer mogelijkheden voor toekomstige uitbreiding en verdere ontwikkeling dan de RSC-TB. Het prototype functioneert in een digitale distributie architectuur, equivalent aan die van het radarsysteem Flycatcher Mk2, waarvoor in dit onderzoek ook opname- en replayfunctionaliteit ontwikkeld is. Deze architectuur is sterk vergelijkbaar met de nog te ontwikkelen architectuur, waarin de RSC-GPU daadwerkelijk gebruikt zal worden. Met het prototype is waardevolle risicoreductie gedaan, zowel van het gebruik van een digitale distributie architectuur als van het gebruik van de GPU voor alle aspecten van radarscan conversie.

Voorwoord

Dit verslag en mij Literatuurstudie naar GPU programmeren zijn de resultaten van mijn afstudeeropdracht waarmee ik mijn studie Informatica aan de Universiteit Twente afrond. Met plezier heb ik deze opdracht uitgevoerd bij de afdeling Above Water Systems van Thales Nederland B.V. te Hengelo.

De Literatuurstudie is geschreven voor alle medestudenten of collega's die meer inzicht willen krijgen in de zeer snel ontwikkelende wereld van GPU programmeren en hoe deze technologie efficiënt kan worden toegepast. Dit verslag is met name interessant voor alle collega's of toekomstige stagairs bij Thales, die verder willen werken met de ontwikkeling of uitbreiding van een op GPU programmeren gebaseerd radarscan conversie systeem.

Graag wil ik in dit voorwoord de gelegenheid benutten om een aantal mensen te bedanken die in het bijzonder een bijdrage hebben geleverd aan mijn afstudeeropdracht, zowel in het bereikte resultaat als in de belevenis van de afgelopen periode. Allereerst Ronald Boers en Stefan van Os, mijn begeleiders van Thales, voor alle tijd die zij in mijn begeleiding hebben gestoken. Job Zwiers, Mannes Poel en Henri Hondorp, vooral voor hun bruikbare commentaar en suggesties om de verslagen tot een goed einde te brengen. Alle medestudenten binnen Thales, voor hun nuttige tips en gezellige gesprekken vóór, tijdens en na de lunch. De medewerkers van het bedrijfsrestaurant voor het bereiden van deze vaak heerlijke lunch. Alle collega's binnen Thales voor hun interesse en bijdrage in mijn opdracht. En natuurlijk mijn ouders en mijn vriendin voor hun steun die ze gedurende mijn studie en afstudeerperiode gegeven hebben.

Riccardo Sirchia

Hengelo, November 2006.

Inhoudsopgave

Samenvatting	2
Voorwoord	3
Inhoudsopgave	4
Lijst van Figuren	6
Lijst van Code fragmenten	8
1 Inleiding	9
1.1 Doelstellingen.....	9
1.2 Begrenzingsen	10
1.3 Het verslag	11
2 Introductie radar en radarvideo	12
3 Radarscan conversie	14
3.1 Generatie uit polair georiënteerde texture coördinaten	14
3.2 Generatie uit cartesisch georiënteerde coördinaten	16
4 Beeldverbetering van PPI weergave	19
4.1 Analyse van sampling probleem	19
4.2 Bepaling van samen te voegen samples	20
4.2.1 Pixel-sample verhouding over het azimut	20
4.2.2 Pixel-sample verhouding over de afstand.....	30
4.3 Sample combinatie algoritmen	33
4.3.1 Gemiddelde waarde.....	33
4.3.2 Gemiddelde met threshold	33
4.3.3 Maximum waarde.....	34
4.4 Kanttekeningen bij de gedane texture sampling	34
4.5 Ondervonden problemen met arctangens	35
4.5.1 Analyse van het arctangens probleem	35
4.5.2 Implementatie van de geboden oplossing	38
4.5.3 Probleem door NVIDIA opgelost	39
5 Op dit moment gebruikte RSC en architecturen.....	40
5.1 Functieoverzicht RSC-TB	40
5.1.1 RVG Hardware	42
5.1.2 RSC-TB Software	46
5.2 Analoge radarvideo distributie.....	47
5.3 Digitale radarvideo distributie	48
6 Opzet radarvideo distributie architectuur.....	52
6.1 Rolverdeling consument – producent	53
6.2 Gebruik van GPU programmeren	54
7 Radarscan conversie prototype	55
7.1 De eisen	55
7.2 De componenten.....	56
7.2.1 Centrale radardisplay.....	56
7.2.2 Radarvideo.....	64
7.2.3 Scheepstoestand.....	67
7.2.4 Panorama	68

8	Integratie van gelogd radarvideo	76
8.1	Analyse mogelijkheden	76
8.2	Radarvideo recorder	77
8.3	Radarvideo replay	79
9	Analyse van het prototype	81
9.1	CPU belasting	81
9.2	GPU belasting	84
9.3	GPU Geheugengebruik	85
9.4	Platform onafhankelijkheid	88
10	Integratie in TAC-Server	89
10.1	Integratie van digitaal radarvideo gedistribueerd met SR(D)V	89
10.2	Integratie van de op GPU gebaseerde RSC functionaliteit	89
11	Conclusies	90
12	Aanbevelingen	91
	Referenties	92
	Begrippenlijst	93
	Bijlage A Screenshot van het prototype	99

Lijst van Figuren

Figuur 1	Schematische weergave van een radarpuls	12
Figuur 2	Enkele mogelijke visualisatievormen van radarvideo.....	13
Figuur 3	Indeling radarvideo als polair coördinatenstelsel.....	14
Figuur 4	PPI geometrie en texture coördinaten in B-Scope met 8 sectoren	15
Figuur 5	Generatie van PPI uit één enkele quad.....	17
Figuur 6	Polaire coördinaten uit cartesisch stelsel met correcte oriëntatie van θ	18
Figuur 7	Sampling probleem in radarscan conversie rondom het middelpunt van de PPI.....	19
Figuur 8	Afbeelding van radarvideo op beeldscherm pixels	20
Figuur 9	Visualisatie van PPI en genererende geometrie	21
Figuur 10	Resultaten van de verschillende azimuth filter algoritmen	22
Figuur 11	Bepalen van correctiefactor afhankelijk van de hoek θ	24
Figuur 12	Goniometrische bepaling van correctiefactor uit polaire coördinaten	24
Figuur 13	Grafieken van correctiefactoren over het domein $[-\pi, \pi]$	25
Figuur 14	Berekening van correcte lijnstuk als correctiefactor	27
Figuur 15	Schematische weergave van sampling bij 6 samples per pixel.....	30
Figuur 16	Resultaten van verschillende afstand normalisatie algoritmen	31
Figuur 17	Schematische weergave van sampling bij 6 samples per pixel.....	33
Figuur 18	GL_REPEAT interpretatie van texture coördinaten buiten het bereik $[0..1]$	34
Figuur 19	Afwijking van de met $\text{atan2}(y,x)$ berekende coördinaten in rood.....	35
Figuur 20	Versil tussen double en single precisie berekeningen in pixels	36
Figuur 21	Versil tussen standaard en alternatieve arctangens vermenigvuldigd met 98	38
Figuur 22	Enkele voorbeelden van radars	40
Figuur 23	Functioneel context diagram van de RSC-TB [2].....	41
Figuur 24	Functionele data flow van de RSC-TB [2].....	41
Figuur 25	RVG Hardware [2]	42
Figuur 26	Radarsectoren, –sweeps en –samples in een complete radarscan	44
Figuur 27	Radarsweep integratie (a) en radarsweep duplicatie (b)	44
Figuur 28	Foutief afterglow effect weergegeven in rood	46
Figuur 29	Analoge radarvideo distributie architectuur [6]	47
Figuur 30	Flycatcher Mk2 opstelling.....	48
Figuur 31	Digitale radarvideo distributie architectuur van de Flycatcher Mk2.....	49
Figuur 32	Data lay-out van een RSP bericht	50
Figuur 33	Data lay-out van een RSD bericht.....	51
Figuur 34	Opzet nieuwe digitale radarvideo distributie architectuur	52
Figuur 35	DFD van het prototype	56
Figuur 36	Klassendiagram <i>RadarDisplayWidget</i> en aangrenzende componenten	57
Figuur 37	Klassendiagram radarvideo producent – consument interface.....	65
Figuur 38	Radarvideo producent – consument collaboration diagram.....	66
Figuur 39	Klassendiagram scheepstoestand producent – consument interface	67
Figuur 40	Voorbeeld panoramisch TV video gegenereerd door Gatekeeper	68
Figuur 41	Plat panorama	69
Figuur 42	Rechtopstaand panorama.....	70
Figuur 43	Orthogonaal geprojecteerd panorama	71
Figuur 44	Op horizon geknikt panorama, overzicht	71
Figuur 45	Op horizon geknikt panorama, vanuit centrum weergegeven.....	72
Figuur 46	Panorama geprojecteerd langs randen van het werkvlak	73
Figuur 47	Klassendiagram panorama producent – consument interface	74
Figuur 48	Klassendiagram radarvideo recorder.....	77

Figuur 49	Screenshot radarvideo recorder	78
Figuur 50	Klassendiagram radarvideo replayer	79
Figuur 51	Screenshot radarvideo replayer	80
Figuur 52	Testsysteem specificaties	81
Figuur 53	CPU meetresultaten systeem A	82
Figuur 54	CPU meetresultaten systeem B	82
Figuur 55	Overzicht van CPU belasting per verwerkte hoeveelheid MB/s.....	82
Figuur 56	Enkele performance analyse scenario's op systeem A.....	83
Figuur 57	GPU belasting van systeem A	84
Figuur 58	GPU belasting van systeem B	85
Figuur 59	Theoretisch worst-case volledig GPU geheugengebruik prototype.....	86
Figuur 60	Worst-case toegevoegd GPU geheugengebruik door RSC-GPU t.o.v RSC-TB..	87
Figuur 61	Applicatie met GUI elementen.....	99

Lijst van Code fragmenten

Code fragment 1	Cg code voor filtering met een rond filterbereik.....	23
Code fragment 2	Cg code voor filtering met een vierkant filterbereik	26
Code fragment 3	Cg code voor filtering met een bloemvormig filterbereik.....	28
Code fragment 4	Cg code voor azimuth filtering, en bepaling van maximum	29
Code fragment 5	C++ code voor het bepalen van filter gebied voor afstand filtering.....	32
Code fragment 6	Cg code voor afstand filtering, en bepaling van maximum	32
Code fragment 7	Matlab code voor single/double precisie berekening.....	37
Code fragment 8	myAtan en myAtan2 definities	38
Code fragment 9	Cg code voor normale afterglow	58
Code fragment 10	Cg code voor afterglow met threshold	59
Code fragment 11	Cg code voor intensiteit onafhankelijke afterglow met threshold.....	59
Code fragment 12	Cg code voor weergave van radarvideo in groene kleur	62
Code fragment 13	Cg code voor weergave van radarvideo met color lookup texture.....	63
Code fragment 14	Cg code voor weergave van radarvideo met color lookup en ripple.....	63
Code fragment 15	Cg code voor weergave van radarvideo met offset – gain filter	64

1 Inleiding

Dit onderzoek is gestart als testcase voor mijn afstudeeropdracht waarin het programmeren van de hoofdprocessor op een grafische kaart, de Graphical Processing Unit of kortweg *GPU*¹, centraal staat. In mijn literatuurstudie over GPU programmeren [1] wordt in detail ingegaan op allerlei aspecten die hierin een rol spelen. Naast de evolutie en de huidige stand van zaken van de architectuur, programmeertalen en beschikbare ontwikkelapplicaties voor het programmeren van de GPU, wordt ook ingegaan op technieken die het mogelijk maken om de GPU voor generieke, niet 3D grafische doeleinden te gebruiken. Daarnaast zijn in het onderzoek ook interessante performance aspecten ontdekt. De grote verschillen in effectieve bandbreedte bij het op verschillende wijzen versturen van data tussen CPU en GPU zijn hier een voorbeeld van. In de literatuurstudie worden dan ook verscheidene aanbevelingen gedaan over hoe efficiënt gebruik gemaakt kan worden van de mogelijkheden van een GPU.

De opgedane kennis over GPU programmeren is in de praktijk gebracht met het in dit verslag beschreven onderzoek naar de mogelijkheden van de toepassing GPU programmeren voor *radarscan conversie*. Met radarscan conversie wordt het proces bedoeld dat de ruwe radar informatie, afkomstig van de radars zelf, omzet naar de welbekende ronde radar visualisatievorm. In deze vorm corresponderen de locaties van radarecho's met hun werkelijke posities in de wereld en kunnen dus rechtstreeks op een landkaart afgebeeld worden. Deze projectievorm wordt de Plan Position Indication of *PPI* genoemd.

In de loop der tijd zijn ook andere bewerkingen in het takenpakket van een Radar Scan Converter (*RSC*) opgenomen. Voorbeelden hiervan zijn het bijhouden van de geschiedenis van de radarecho's en het gebruik van kleurtabellen teneinde het radarvideo optimaal weer te geven. De op dit moment gebruikte RSC, de Radar Scan Converter – Texture Based of *RSC-TB*, doet het grootste gedeelte van deze bewerkingen hardwarematig op een speciaal voor dit doel ontworpen PCI insteekkaart genaamde de Radar Video Grabber of *RVG*. Deze kaart vormt tevens de interface tussen de console en het analoge distributienetwerk dat op dit moment voor de radarvideo gebruikt wordt.

1.1 Doelstellingen

Met het gebruik van de RSC-TB is geconstateerd dat er belangrijke radar informatie verloren kan gaan rond het middelpunt van een PPI. Dit uit zich in het wegvallen van radarsamples op het scherm. Daarnaast is er interesse ontstaan voor het gebruik van een digitale radarvideo distributie architectuur. Deze punten, samen met de wens om de mogelijkheden van GPU technologie te gebruiken voor productinnovatie, hebben tot de volgende doelstellingen geleid:

- ❑ Ontwikkeling van een filter dat dataverlies rond het middelpunt van een PPI tegen gaat en implementatie hiervan op de GPU.
- ❑ Doen van een onderzoek naar de haalbaarheid van radarscan conversie dat volledig op de GPU wordt uitgevoerd.
- ❑ Implementeren van een prototype GPU radarscan convertor, gebruik makende van een digitale radarvideo distributie.
- ❑ Integratie van gelogd radarvideo in het prototype
- ❑ Implementeren van sensorfusie van panoramisch TV en radarvideo teneinde nieuwe mogelijkheden voor productinnovatie te illustreren.

¹ Alle cursief gedrukte termen en afkortingen zijn in de begrippenlijst aan het eind van dit verslag op te zoeken.

De eerste doelstelling voor dit onderzoek is het ontwikkelen van een algoritme dat het beschreven probleem van dataverlies kan oplossen. Hierbij dient rekening gehouden te worden met het programmeermodel van de GPU, zodat het algoritme op efficiënte wijze hierop geïmplementeerd kan worden. De in Hoofdstuk 4 beschreven filtermethoden zijn geïmplementeerd in een prototype, dat zelf radarvideo patronen genereert.

Om de haalbaarheid van een volledig op de GPU gebaseerde radarscan conversie te bepalen, moet in eerste instantie een overzicht verkregen worden van alle taken die door het huidige systeem, de RSC-TB uitgevoerd worden. Na de inventarisatie hiervan is in Hoofdstuk 6 beargumenteerd welke taken slechts éénmaal per radarbron gedaan kunnen worden en welke taken elke operator individueel moet uitvoeren. Hierbij is ook gekeken welke taken, zowel aan de produceren als de consumerende kant, kunnen worden uitgevoerd met behulp van GPU programmeren.

Na deze analyse is het prototype aangepast en uitgebreid tot het systeem dat in Hoofdstuk 7 wordt beschreven. Hierbij is tevens een scheiding gemaakt tussen het radarvideo producerende en consumerende deel, waartussen communicatie zal plaatsvinden middels een ethernet verbinding. Met de werking van het prototype wordt tevens de mogelijkheid van radarvideo over een digitale distributie architectuur bewezen.

Zoals eerder vermeldt wordt er door het prototype zelf radarvideo gegenereerd, dat in een later stadium via ethernet gedistribueerd wordt. Voor het demonstreren van de sampling problemen kan dit video bestaan uit vaste patronen. Voor het demonstreren van de werking van de radarscan conversie is gedacht aan een ruisbeeld met daarop simpele objecten die volgens een bepaald patroon bewegen. Om de geboden functies van het prototype goed te kunnen testen en demonstreren, heeft het gebruik van gelogd radarvideo een grote meerwaarde. Na een analyse van de mogelijkheden hiervoor, wordt in Hoofdstuk 8 beschreven hoe dit uiteindelijk in de praktijk is gebracht.

Tenslotte is het prototype radarscan conversie systeem gebruikt voor een experiment met sensorfusie. Hierbij is gebruik gemaakt van 360 graden panoramisch TV video, gegenereerd door het Gatekeeper project. Het doel van dit experiment is het aantonen dat dit met GPU programmeren mogelijk is en het in gang zetten van gedachtestromen over de mogelijke visualisatievormen hiervan. De in dit onderzoek ontwikkelde visualisatievormen worden beschreven in Hoofdstuk 7.2.4.

1.2 Begrenzungen

Zowel de producent als de consument van radarvideo worden ontwikkeld en geïmplementeerd in C++ voor het Linux platform. Platform onafhankelijkheid van de implementatie is hierbij een pré, geen vereiste. Om hierin tegemoet te komen wordt gebruik gemaakt van de Qt Toolkit.

Met het oog hierop dient OpenGL als basis voor de grafische bewerkingen. Gezien het gekozen Linux platform valt de keuze voor het type videokaart op een kaart van NVIDIA. Dit vanwege de hogere kwaliteit van de beschikbare drivers voor dit platform, die in belangrijke mate verantwoordelijk zijn voor de behaalde prestaties.

De gemaakte keuze voor een NVIDIA videok kaart geeft de kans om betere prestaties en meer mogelijkheden te verkrijgen door gebruik te maken van de programmeertaal Cg [1]. Dit gegeven en de reeds opgedane ervaring met deze programmeertaal, verantwoordt de keuze om alle implementaties van de gemaakte algoritmen in deze taal te maken.

Het systeem dat voor ontwikkeling van het prototype gebruikt is, bestaat uit de volgende configuratie.

- ❑ Dell Precision 360
- ❑ Intel Pentium 4 CPU op 2.60 GHz
- ❑ 1024 MB intern geheugen
- ❑ GeForce 6600 GT op AGP bus met 128 MB geheugen
- ❑ Linux SUSE 9.3
- ❑ NVIDIA closed source driver versie 7676, later vervangen door versie 8174 en 8756
- ❑ Cg Toolkit versie 1.4, later vervangen door versie 1.5 beta1

Door de keuze van een dergelijk systeem is meteen aangetoond dat het prototype ook prima functioneert op een middenklasse systeem. Er zijn dus geen extreme high-end componenten noodzakelijk.

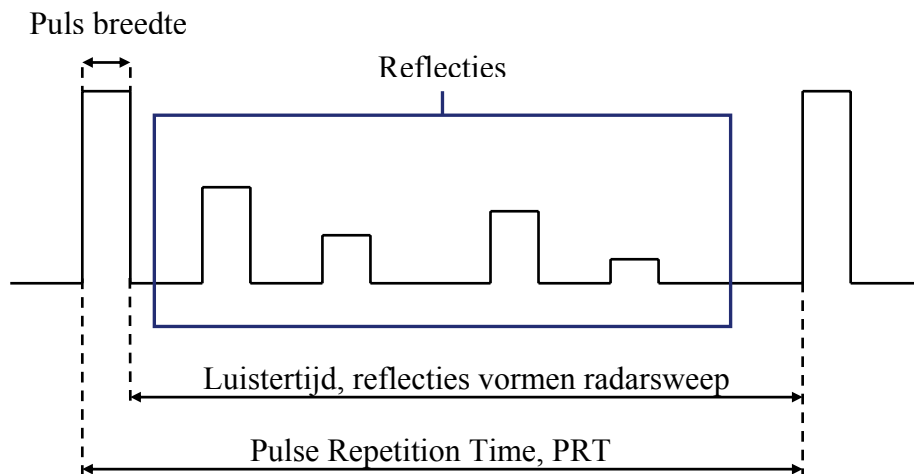
1.3 Het verslag

In het verslag zal als eerste enige uitleg gegeven worden over radarscan conversie. Hierbij komt aan bod hoe deze wordt toegepast in het huidige systeem, de RSC-TB, en hoe dit efficiënt met GPU programmeren kan worden gerealiseerd. Vervolgens wordt het probleem van dataverlies in de weergave van radarvideo in een PPI beschreven en worden enkele filtermethoden hiervoor uitgewerkt. Hierna worden de belangrijkste mogelijkheden en eigenschappen van het op dit moment gebruikte radarscan conversie systeem en bijbehorende distributie architectuur behandeld. Aan de hand hiervan wordt de opzet voor de nieuwe distributie architectuur gepresenteerd en worden de mogelijkheden en beperkingen van het gemaakte prototype beschreven. Hierbij komen ook de in dit onderzoek ontwikkelde recorder en replay van radarvideo aan bod. Tenslotte wordt er nog een korte analyse gegeven van de verwachte impact die de integratie van de voorgestelde technieken in het op dit moment gebruikte Combat Management Systeem (CMS) zal hebben.

2 Introductie radar en radarvideo

Een radar wordt gebruikt voor het detecteren van een object en zijn snelheid op een (grote) afstand. Deze detectie wordt bereikt door het meten van echo's en de *Doppler verschuivingen* in de frequenties hiervan. Al draaiende zendt de radar radiogolven van bepaalde (al dan niet variërende) frequenties uit. Wanneer deze radiogolven op een object stuiten, worden deze (deels) hierdoor teruggekaatst. Door deze weerkaatsing weer op te vangen kan, aan de hand van de tijdsduur tussen het verzenden en ontvangen van de golf en de snelheid van de radiogolven, bepaald worden hoe ver het object zich van de radar bevindt. Daarmee is de maximale afstand die de radar kan meten afhankelijk van de *luistertijd* van de radar.

Daarnaast wordt gekeken naar het verschil in frequentie tussen de ontvangen en verzonden golf. Wanneer de zender en ontvanger naar elkaar toe bewegen, wordt de golf als het ware ingedrukt, wat een verhoging in de frequentie van deze golf tot gevolg heeft. Dit effect wordt Doppler verschuiving genoemd en de mate van deze verandering in golflengte kan gebruikt worden om de relatieve snelheid tussen ontvanger en verzender te bepalen.

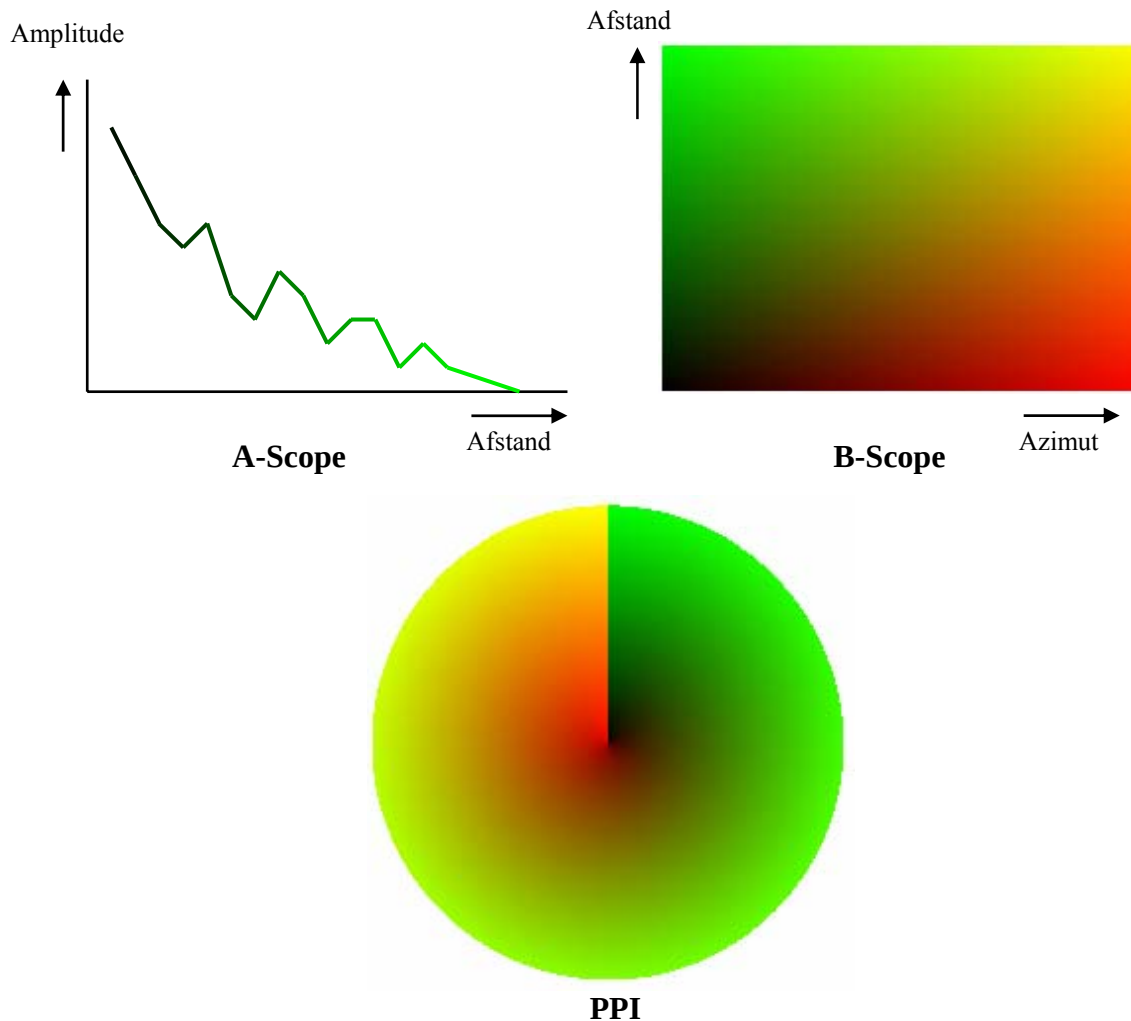


Figuur 1 Schematische weergave van een radarpuls

In Figuur 1 zijn schematisch de onderdelen van de radarpulsen weergegeven. Dit proces herhaalt zich gedurende de rotatie van de radar waardoor steeds een puls verwerkt wordt met een andere *azimut*, een hoek ten opzichte van het geografische noorden. De pulsen worden uitgezonden met een bepaalde herhalingsfrequentie genaamd de Pulse Repetition Frequency (*PRF*). Deze is gelijk aan $1/PRT$. De reflecties die het resultaat zijn van zo'n puls worden *radarsweeps* genoemd. Zo'n radarsweep bevat dus alle informatie met een bepaalde azimut.

Van een enkele radar kan meer dan één radarvideo komen. De belangrijkste zijn Lineair (*LIN*), Moving Target Indication (*MTI*), en Secundair (*SEC*) radarvideo, waarbij LIN het ruwe radarvideo bevat, MTI uitsluitend de bewegende objecten en SEC radarinformatie zoals Identification Friend or Foe (*IFF*) bevat.

Verder kan de radarvideo op verschillende manieren gevisualiseerd worden. In Figuur 2 zijn drie voorbeelden hiervan gegeven. De gebruikte kleuren zijn niet representatief voor de werkelijkheid, maar dienen als ondersteuning om de relaties tussen de verschillende visualisatievormen te verduidelijken.



Figuur 2 Enkele mogelijke visualisatievormen van radarvideo

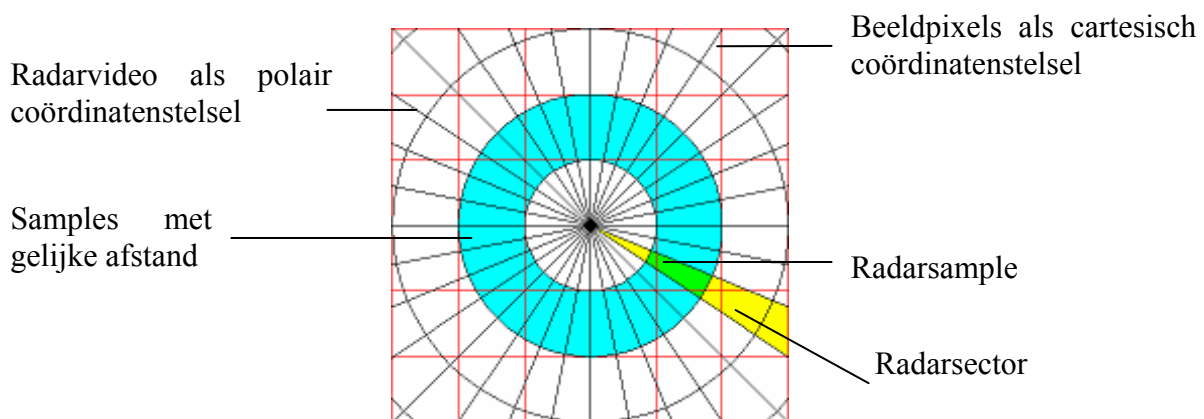
Allereerst kunnen losstaande sweeps of sectoren of delen hiervan in *A-Scope* weergegeven worden. Hierin wordt de amplitude van de *radarsamples* tegen de afstand afgezet. Objecten gedetecteerd in de radar zijn dan zichtbaar als pieken in deze lijn.

Verder is het mogelijk om een complete radarscan of een deel hiervan in *B-Scope* weer te geven. Hierin wordt de afstand van de *radarsamples* tegen het azimut hiervan afgezet. In werkelijkheid wordt door (de intensiteit van) een kleur de afwezigheid dan wel de amplitude van de *radarsample* weergegeven. Objecten zijn hierin dus zichtbaar als punten of vlekken die door verschil in kleur of intensiteit in contrast staan met de omgeving.

Tenslotte wordt hier nog de *PPI* behandeld. Dit is de visualisatievorm die het meest bekend en gebruikt is. Hierin is de radarvideo afgebeeld alsof het op een kompas geplaatst is. Het centrum van de cirkel stelt de locatie van de radar voor, met het geografische noorden naar boven gericht. De afstand tot het middelpunt stelt de afstand tot de radar voor. Ook hier wordt de amplitude van de *radarsample* weergegeven met (de intensiteit van) een kleur.

3 Radarscan conversie

Radarvideo wordt door radars per sweep gegenereerd en gedistribueerd. Door deze sweeps naast elkaar te plaatsen, wordt radarvideo in B-Scope verkregen. Met radarscan conversie wordt het proces aangeduid, dat deze radarvideo in B-Scope converteert naar een PPI beeld. De oriëntatie en indexering van radarsamples in een PPI komt overeen met een polair coördinatenstelsel met coördinaten (r, θ) . Deze analogie wordt verduidelijkt aan de hand van Figuur 3. In een PPI bevinden alle samples van een radarvideo met gelijke afstand tot het schip zich in een ring om het middelpunt. De straal van deze ring komt overeen met de r in polaire coördinaten. Verder bestaan radarsectoren uit alle radarsamples met een gelijke azimuth waarde en hebben dus altijd dezelfde hoek tot het middelpunt, het schip. Deze hoek is op te vatten als de θ van een polaire coördinaat. Wanneer een radarscan uit 4096 sectoren bestaat, kan deze hoek bepaald worden met een nauwkeurigheid van $360/4096$ graden of $2\pi/4096$ radialen. Met behulp van deze gegevens kan een willekeurige radarsample met polaire coördinaten (r, θ) aangeduid worden.

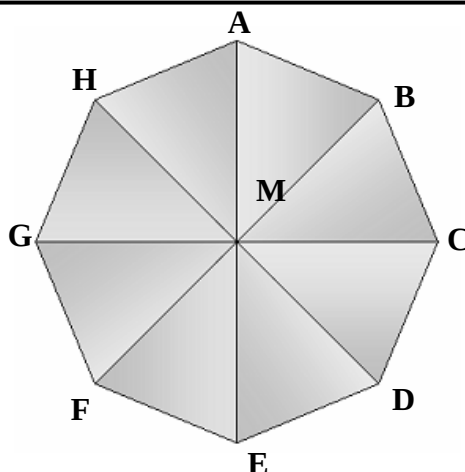


Figuur 3 Indeling radarvideo als polair coördinatenstelsel

Aangezien de pixels van een beeldscherm volgens een cartesisch coördinatenstelsel gerangschikt en geïndexeerd worden, komt de radarscan conversie in feite neer op het converteren tussen polaire en cartesische coördinaten. De vorm van de invoer voor dit proces ligt in dit geval al vast. Dit is de radarvideo die in B-Scope formaat aangeleverd wordt. Doordat de in dit verslag gepresenteerde radarscan convertor evenals de RSC-TB de radarscan conversie met behulp van OpenGL functionaliteit uitvoeren, wordt deze B-Scope als texture gehanteerd. Om deze rechthoekige texture met de radarvideo in B-Scope op de gewenste wijze op het scherm te krijgen, moet er dan ook een correcte combinatie van geometrie en texture coördinaten gebruikt worden. Hierbij moeten die coördinaten al dan niet door middel van een pixel shader verder verwerkt worden. Hiervoor zijn twee methoden bedacht die in de volgende sub-hoofdstukken aan bod komen.

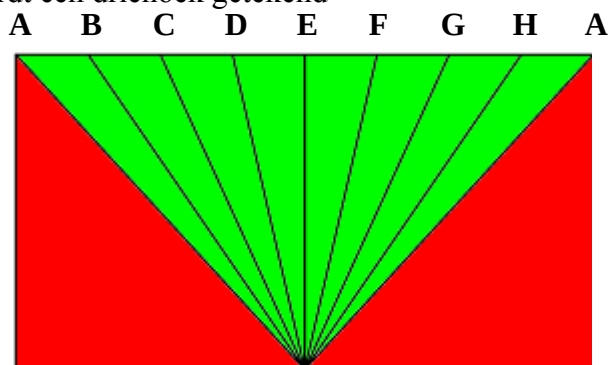
3.1 Generatie uit polair georiënteerde texture coördinaten

Deze eerste methode wordt ook toegepast door de RSC-TB en bestaat uit het tekenen van geometrie in de vorm van de te genereren PPI. Dit kan bijvoorbeeld door een *triangle-fan* te tekenen of door een *quad-strip* of losse *triangles* in een cirkelvormig patroon te genereren. Belangrijk hierbij is dat zoveel mogelijk texture coördinaten tussen $(0,0)$ en $(1,1)$ meegegeven dan wel gegenereerd kunnen worden door interpolatie tussen de *vertices* om zoveel mogelijk radarvideo te indexeren.



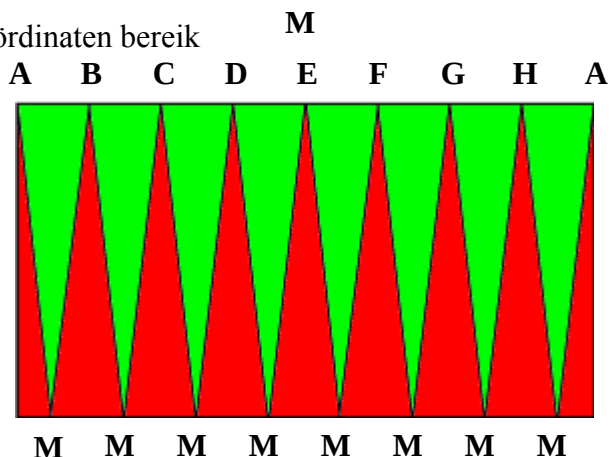
a) Geometrie met vertices, per radarsector wordt één driehoek getekend

Vertex	Texture coördinaat
M	$(\frac{1}{2}, 0)$
A	$(0, 1)$
B	$(\frac{1}{8}, 1)$
C	$(\frac{1}{4}, 1)$
D	$(\frac{3}{8}, 1)$
...	...



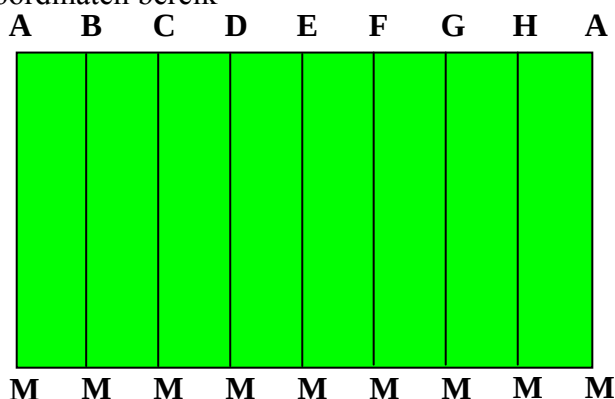
b) Triangle-fan vertex generatie en texture coördinaten bereik

Vertex	Texture coördinaat
M	$(\frac{1}{16}, 0)$
A	$(0, 1)$
B	$(\frac{1}{8}, 1)$
M	$(\frac{3}{16}, 0)$
B	$(\frac{1}{8}, 1)$
C	$(\frac{1}{4}, 1)$
...	...



c) Losse triangle vertex generatie en texture coördinaten bereik

Vertex	Texture coördinaat
A	$(0, 1)$
M	$(0, 0)$
B	$(\frac{1}{8}, 1)$
M	$(\frac{1}{8}, 0)$
C	$(\frac{1}{4}, 1)$
M	$(\frac{1}{4}, 0)$
...	...



d) Quad-strip vertex generatie en texture coördinaten bereik

Figuur 4 PPI geometrie en texture coördinaten in B-Scope met 8 sectoren

Zoals gedemonstreerd wordt in Figuur 4b en c wordt, wanneer gebruik wordt gemaakt van op triangle of triangle-fan gebaseerde geometrie, slechts de helft van alle radarvideo geïndexeerd. Bij indexing zoals in de triangle-fan wordt het effect van dit probleem nog groter gemaakt doordat er een non-uniforme verdeling van de radarvideo op de geometrie gemaakt wordt. Hierdoor heeft data aan de randen van de B-Scope absoluut geen kans om opgenomen te worden in de uiteindelijke afbeelding. Op basis hiervan kan geconcludeerd worden dat de in Figuur 4d voorgestelde geometrie op basis van een quad-strip de beste is.

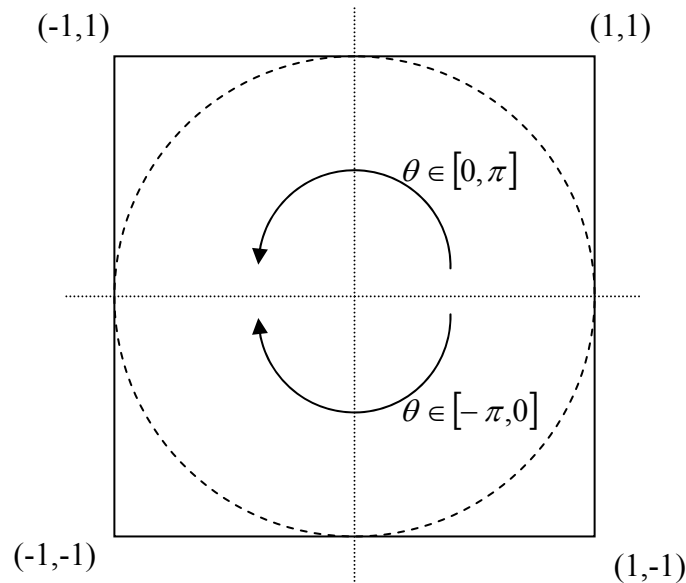
Door deze combinatie van geometrie en texture coördinaten kan rechtstreeks een PPI op het scherm gegenereerd worden. Voor elke pixel in de PPI wordt door het rastering proces een texture coördinaat geïnterpoleerd tussen de opgegeven vertices [1]. Vervolgens wordt de pixel uit de B-Scope, die zich het dichtst bij deze coördinaat bevindt, gekozen en op het scherm afgebeeld. Op deze manier en zonder verdere filtering wordt op dit moment het radarvideo ook weergegeven door de RSC-TB.

Afgezien van het gemis van indexing van alle radar data heeft deze methode voor het genereren van een PPI op het beeldscherm nog een nadeel. Dit komt doordat de complexiteit van de geometrie toeneemt naarmate de hoeveelheid radarsectoren per scan groter wordt. Voor elke radarsector moet er namelijk een extra quad en daarmee twee vertices met texture coördinaten gegenereerd en door de grafische kaart verwerkt worden.

Door de reken capaciteit van de grafische kaart en de verkregen hoeveelheid vertices zou men verwachten dat dit vrij weinig effect heeft maar toch heeft dit een drastische daling in de prestaties tot gevolg. Dit valt te wijten aan het feit dat de rasterizer een zeer grote hoeveelheid tests moet doen om te bepalen welk deel van de geometrie als fragment doorgegeven wordt wanneer er meer dan één polygoon dezelfde pixel beslaat [1]. Analooq aan Figuur 3 op bladzijde 14 beslaan er vooral in het midden van de PPI veel meer polygonen dezelfde pixel. Dit neemt alleen maar toe naarmate er meer radarsectoren en daarmee meer geometrie getekend wordt. Hierdoor is deze methode dan ook minder geschikt om als basis te dienen voor verdere filtering en is er verder onderzoek gedaan naar alternatieve mogelijkheden.

3.2 Generatie uit cartesisch georiënteerde coördinaten

Een andere methode is het genereren van de polair georiënteerde PPI uit cartesische texture coördinaten. De genererende geometrie is in dit geval simpelweg één quad, ongeacht de dimensies of resolutie van de radarvideo. Deze quad bevat op zijn vier vertices de texture coördinaten zoals weergegeven in Figuur 5 waarmee in feite een klein cartesisch georiënteerd coördinatenstelsel gegenereerd wordt.



Figuur 5 Generatie van PPI uit één enkele quad

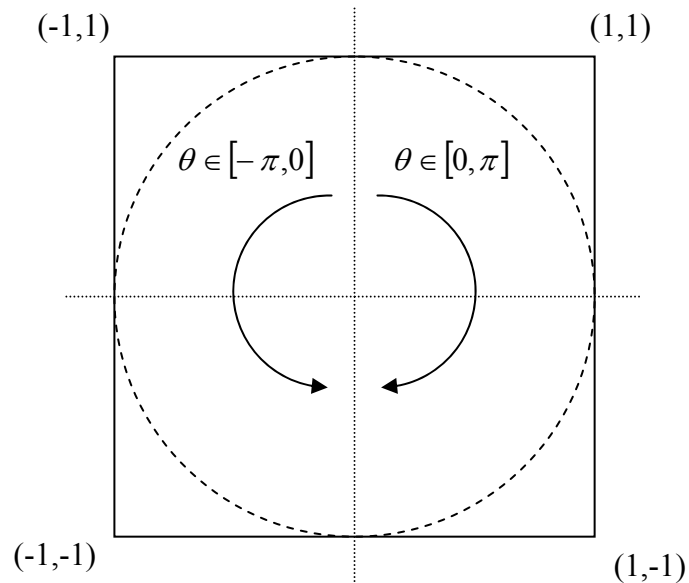
Om de juiste coördinaten binnen de B-Scope te vinden moet er dus een conversie plaatsvinden van cartesische naar polaire coördinaten. Voor deze conversie hanteert men doorgaans de volgende formules:

$$(x, y) \rightarrow (r, \theta) \begin{cases} r = \sqrt{x^2 + y^2} \\ \theta = \arctan\left(\frac{y}{x}\right) \end{cases}$$

Hierbij vertaalt de r zich naar de afstand en de θ naar het azimut binnen de B-Scope. Bij generatie door een quad met hoekpunten $(-1,-1)$ linksonder en $(1,1)$ rechtsboven vormt de coördinaat voor de afstand weinig problemen. Deze ligt uitsluitend buiten het bereik van $[0,1]$ wanneer een coördinaat gekozen wordt die zich buiten de PPI bevindt. Er kan dus eenvoudig gekozen worden om geen output voor een fragment te genereren wanneer $r > 1$. Voor alle andere gevallen is de radius r al genormaliseerd over het bereik $[0,1]$ en kan dus rechtstreeks als element van de coördinaat van de pixel in de B-Scope gebruikt worden.

In het geval van θ is dit proces wat ingewikkelder. De functie $\text{atan}(x)$ in Cg geeft geen informatie over in welk kwadrant de hoek zich bevindt. Hier zou, wanneer deze functie gebruikt wordt, dus zelf op gecontroleerd moeten worden. Gelukkig heeft Cg ook een implementatie van $\text{atan2}(y,x)$; deze functie berekent de formule $\arctan(y/x)$ en corrigeert deze waarde naar het kwadrant waarin het punt (x,y) zich bevindt. Hierdoor wordt voor alle coördinaten (x,y) op correcte wijze de hoek ten opzichte van de x-as weergegeven.

Zoals in Figuur 5 te zien is, is de hoek gedefinieerd in het bereik $[0, \pi]$ wanneer y positief, en in het bereik $[-\pi, 0]$ wanneer y negatief is. Nu is het alleen nog zaak deze hoek te normaliseren over het bereik $[0,1]$. Hierbij moet rekening gehouden worden met het feit dat de eerste sector in B-Scope overeen komt met het noorden waar 0 in de oriëntatie van Figuur 5 overeenkomt met het oosten. Tevens moet het azimut met de klok mee toenemen, waar de polaire coördinaten juist afnemen. Deze problemen zijn echter beide op te lossen door in plaats van $\arctan(y/x)$, de formule $\arctan(x/y)$ te hanteren. Dit vertaalt dus naar de functieaanroep $\text{atan2}(x,y)$ wat resulteert in de oriëntatie zoals gegeven in Figuur 6.



Figuur 6 Polaire coördinaten uit cartesisch stelsel met correcte oriëntatie van θ

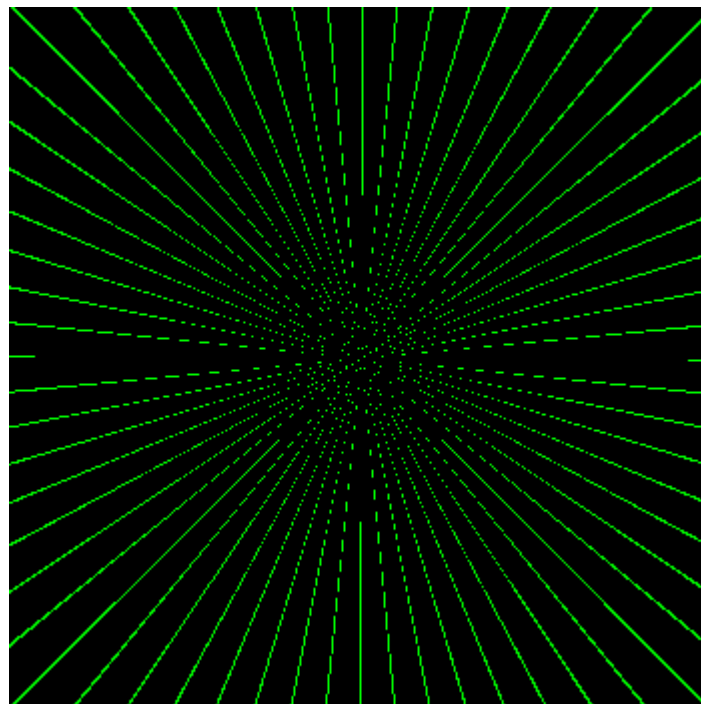
De op deze manier berekende θ kan nu gedeeld worden door 2π wat resulteert in coördinaten binnen het bereik $[-0.5, 0.5]$. Feitelijk zijn de coördinaten in deze vorm nog niet geheel genormaliseerd maar hebben ze al wel de juiste oriëntatie. Normalisering naar het bereik $[0, 1]$ is nu makkelijk mogelijk door 1 bij het berekende azimuth op te tellen wanneer deze kleiner dan 0 is. Dit is echter niet noodzakelijk wanneer gebruik wordt gemaakt van texture sampling met GL_REPEAT. In Hoofdstuk 4.4 nader worden toegelicht waarom dit zo is.

4 Beeldverbetering van PPI weergave

In dit hoofdstuk wordt verder ingegaan op het probleem van dataverlies in de PPI weergave van radarvideo. Er wordt een analyse naar de oorzaak van het probleem beschreven, waarna enkele ontwikkelde filtermethoden behandeld worden, die dit probleem tegengaan. Verder wordt naast deze filtering over het azimut ook filtering over de afstand besproken. Bij het behandelen van deze punten wordt uitsluitend beschreven hoe de radarsamples worden bepaald die samenvallen op een af te beelden pixel. De methoden om deze pixels samen te nemen, wat voor filtering in beide richtingen relevant is, worden in Hoofdstuk 4.3 beschreven. Tenslotte zal uitgelegd worden hoe er voor deze filter methoden met de texture sampler wordt omgegaan, en zullen er nog enkele ondervonden problemen en oplossingen daarvoor beschreven worden.

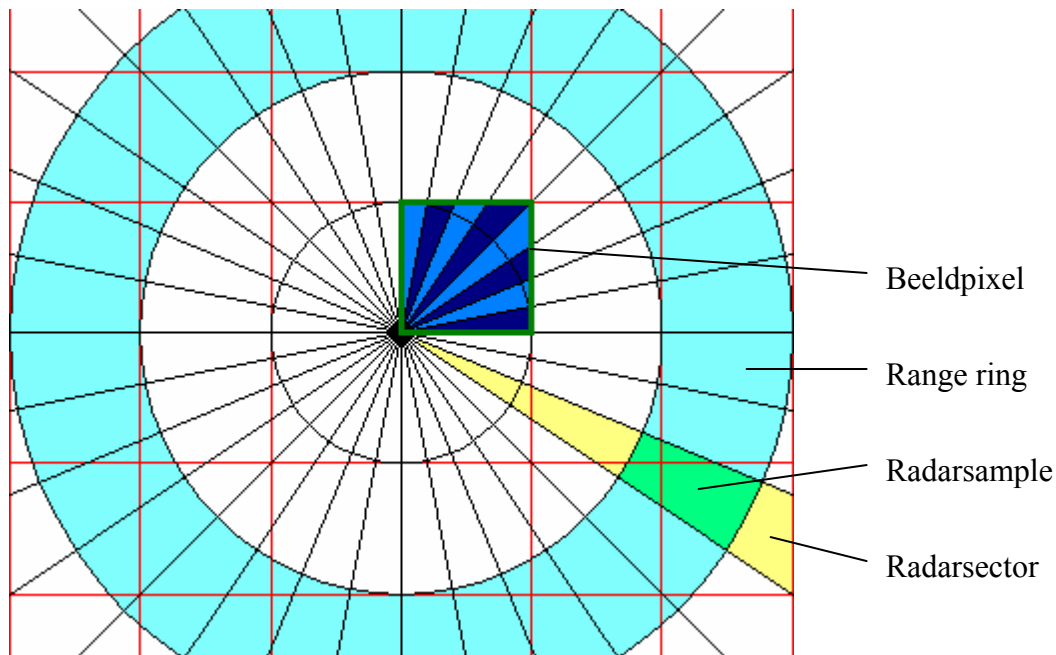
4.1 Analyse van sampling probleem

Zoals eerder vermeld, is het probleem naar voren gekomen in simulatie proeven met het huidige systeem, de *RSC-TB*. Door hiermee een spakenpatroon te genereren, waarbij er van elke 8 sectoren één is die de volle intensiteit heeft, wordt op eenvoudige wijze zichtbaar gemaakt dat er radarinformatie verloren gaat. In de ideale situatie zouden alle spaken tot het middelpunt door moeten lopen. Alle sectoren, of deze nu een maximale of minimale intensiteit heeft, zijn immers afkomstig van de radar welke zich in het middenpunt bevindt. Daarbij zou een sector met een hoge intensiteit altijd voorrang verkrijgen boven een sector met een lagere intensiteit. De eerstgenoemde bevat immers potentieel interessante informatie over gedetecteerde objecten. In Figuur 7 is echter te zien dat in deze simulatie de spaken rondom het midden duidelijk onderbrekingen vertonen en in enkele gevallen duidelijk niet geheel tot het midden doorlopen. Deze simulatie toont een radarscan met in totaal 1024 sectoren, waarvan, zoals beschreven, elke achtste sector de maximale intensiteit bevat.



Figuur 7 Sampling probleem in radarscan conversie rondom het middelpunt van de PPI

Ten grondslag aan dit dataverlies ligt het feit dat er rond het middelpunt meer samples samenvallen op één beeldpixel. Dit is het beste uit te leggen aan de hand van Figuur 8. Hier is duidelijk te zien dat de groen omlijnde pixel rechts boven het middelpunt alle radarsamples van 90 graden vlak bij de radar bevatten, dus $\frac{1}{4}$ van de totale hoeveelheid sectoren in een scan. In het voorbeeld komt dit neer op samples uit 8 radarsectoren, maar in realistische situaties worden resoluties tot 4096 sectoren per scan gebruikt, wat resulteert in het samenvallen van 1024 samples binnen één pixel.



Figuur 8 Afbeelding van radarvideo op beeldscherm pixels

Het wegvallen van samples met hogere intensiteit is te wijten aan het feit dat er per pixel slechts één sample de uiteindelijke waarde zal bepalen. Deze wordt gekozen door te kijken welke van deze samples het dichtst bij het middelpunt van de pixel ligt. Hierdoor is statistisch gezien de kans groot dat er op een pixel niets afgebeeld wordt wanneer het grootste deel van alle samples op de pixel, in het voorbeeld $\frac{7}{8}$ deel, geen of hele zwakke echo's bevat.

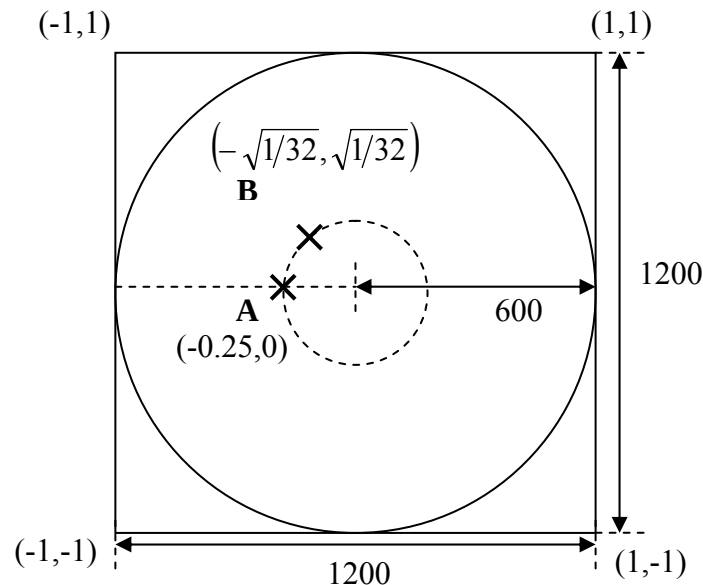
4.2 Bepaling van samen te voegen samples

Om het probleem tegen te gaan moet dus niet alleen de sample in het midden, maar ook alle andere samples welke op die pixel vallen, gebruikt worden om de pixel waarde te bepalen. Het belangrijkste onderdeel van de algoritmen is dan ook te bepalen hoeveel en welke samples uit de radarvideo samengevoegd moeten worden per pixel. Hoe deze samples uiteindelijk samengevoegd kunnen worden en de voor- en nadelen van deze methoden worden uitgelegd in Hoofdstuk 4.3.

4.2.1 Pixel-sample verhouding over het azimut

De eerste en voor dit onderzoek belangrijkste richting van het filteren, is het filteren van samples langs de azimut richting. Dit probleem is uitgelegd in Hoofdstuk 4.1 en komt voort uit het feit dat de omtrek van een cirkel afneemt naarmate de straal kleiner wordt, terwijl de hoeveelheid radarinformatie over het azimut onveranderlijk is met de afstand van het middelpunt.

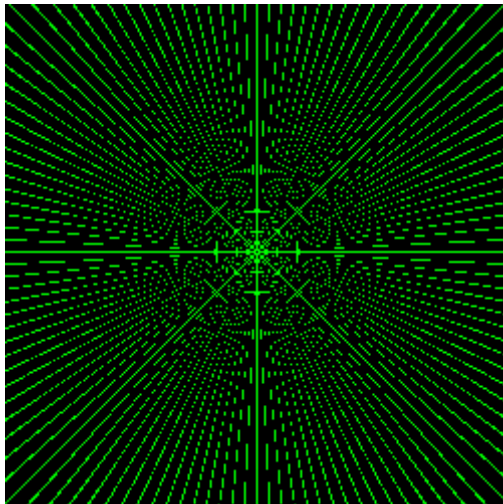
Zoals in Hoofdstuk 3 is uitgelegd kan er vanuit elk willekeurige beeldpunt bepaald worden welke radarsample daarbij het dichtste in de buurt komt. Het probleem is echter dat er radarsamples verloren kunnen gaan wanneer het beeld met uitsluitend deze dichtstbijzijnde sample gebruik wordt voor de projectie. Om dit probleem en de geboden oplossingen uit te leggen, wordt uitgegaan van een voorbeeldsituatie waarbij radarvideo met daarin 2048 sectoren en 1024 range samples per sector wordt afgebeeld op een PPI met een radius van 600 beeldpunten (langs de x-as). Het PPI beeld wordt gegenereerd door een *quad* te tekenen van 1200x1200 pixels met daarin texture coördinaten van, (-1,-1) linksonder naar (1,1) rechtsboven, zie Figuur 9.



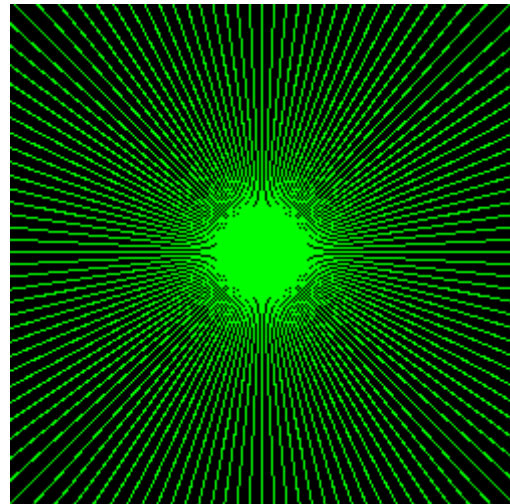
Figuur 9 Visualisatie van PPI en genererende geometrie

In hoofdstuk 4.2.1.1 t/m 4.2.1.3 worden de drie ontwikkelde algoritmen behandeld, waarmee bepaald wordt hoeveel samples er per pixel samengenomen moeten worden om dit probleem tegen te gaan. De verschillende algoritmen zijn ontstaan doordat er met de resultaten van eerdere algoritmen steeds nieuwere inzichten verkregen zijn in de factoren, die bij dit probleem een rol spelen. In Hoofdstuk 4.2.1.4 wordt uitgelegd welke procedure gevolgd wordt om de bepaalde hoeveelheid samples samen te voegen. De verschillende wijzen waarop de individuele samples bij kunnen dragen aan de resulterende intensiteit van de pixel worden in Hoofdstuk 4.3 behandeld.

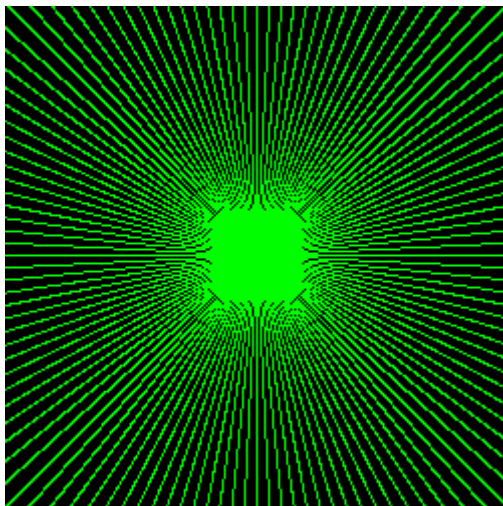
De behaalde resultaten per algoritme worden weergegeven in Figuur 10, met de ongefilterde weergave ter referentie. In deze figuur worden radarbeelden weergegeven, waarbij één op de acht sweeps een hoge intensiteit heeft. Wanneer dit beeld correct weergegeven wordt, zal een spakenpatroon zichtbaar zijn zonder duidelijke onderbrekingen in de individuele spaken.



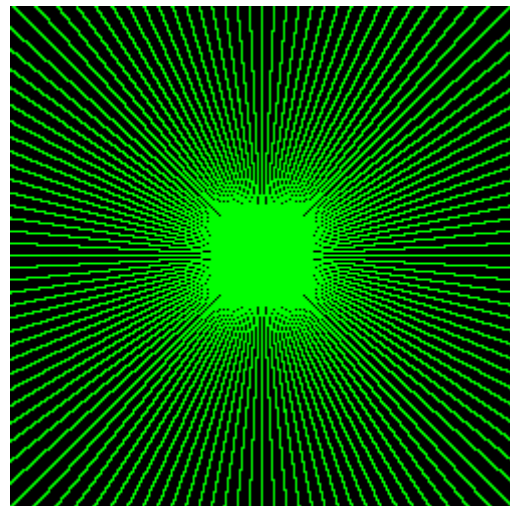
a) Ongefilterde weergave



b) Gefilterd met een rond filterbereik



c) Gefilterd met een vierkant filterbereik



d) Gefilterd met een bloem-vormig filterbereik

Figuur 10 Resultaten van de verschillende azimuth filter algoritmen

4.2.1.1 Rond Filterbereik

Als eerste poging om dit probleem op te lossen werd gekeken naar de verhouding tussen de samples over het azimuth en de hoeveelheid pixels op de omtrek van de cirkel die door het gegeven punt gaat. Deze laatste kan voor bijvoorbeeld punt A in Figuur 9 bepaald worden op $\sqrt{(-0.25)^2 + 0^2} \cdot 600 \cdot 2\pi = 300\pi \approx 942$ pixels. Dit betekent dat er op punt A in dit geval $2048/(300\pi) = 2.17$ samples samengevoegd moeten worden. Wanneer de radius van de ring in PPI groter of gelijk is aan $2048/(2\pi) \approx 326$ pixels wat neerkomt op een fragment met texture coördinaten waarvoor geldt dat $x^2 + y^2 \geq (326/600)^2$, betekent dit dat er geen samples samengevoegd hoeven te worden en dat er dus volstaan kan worden met het weergeven van de dichtstbijzijnde radarsample. Wanneer dit algoritme voor elke pixel in de PPI wordt toegepast levert dit een afbeelding op zoals te zien in Figuur 10b. De Cg code die bij deze procedure hoort is weergegeven in Code fragment 1.

```
float4 radarRoundFP(float2 coords : TEXCOORD0,
                    sampler2D texture,
                    uniform float azimuthResolution,
                    uniform float radarRadius,
                    uniform float unfilteredOffset) : COLOR
{
    //polarCoords will hold the coordinates
    //of the best matching sample in the texture
    float2 polarCoords;

    //Determine range component for the current fragment location
    polarCoords.x = length(coords);

    //determine the theta of the angle
    //by calculating atan2(x,y), rotation will be given range [PI, -PI]
    //with theta = 0 at north direction and increasing clockwise
    float theta = myAtan2(coords.x, coords.y);

    //scale theta to range [-0.5,0.5] with following relation:
    //    [0    0.25  0.5/-0.5  -0.25]
    //    [N     E      S       W  ]
    //Due to GL_REPEAT principle this results in normalized lookup
    //Then shift it with one half sweep width
    //to have sweep 0 centered on north direction
    polarCoords.y = theta/TWO_PI + 0.5/azimuthResolution;

    //factor used to determine if a fragment is inside the PPI area
    //used to prevent unnecessary calculations
    //and to factor output to be black when outside the PPI
    float withinRadarRange = (polarCoords.x > 1 ? 0.0 : 1.0);

    //determine how many pixels will be covered by this fragment
    //determined by circumference of the circle with radius
    //as in polar coords, the width of the PPI, azimuth resolution
    //and the correction factor
    //factored with 'withinRadarRange' to prevent
    //unnecessary calculations
    float samplesCovered = withinRadarRange*azimuthResolution/
        (TWO_PI*polarCoords.x*radarRadius);

    /* .... */
}
```

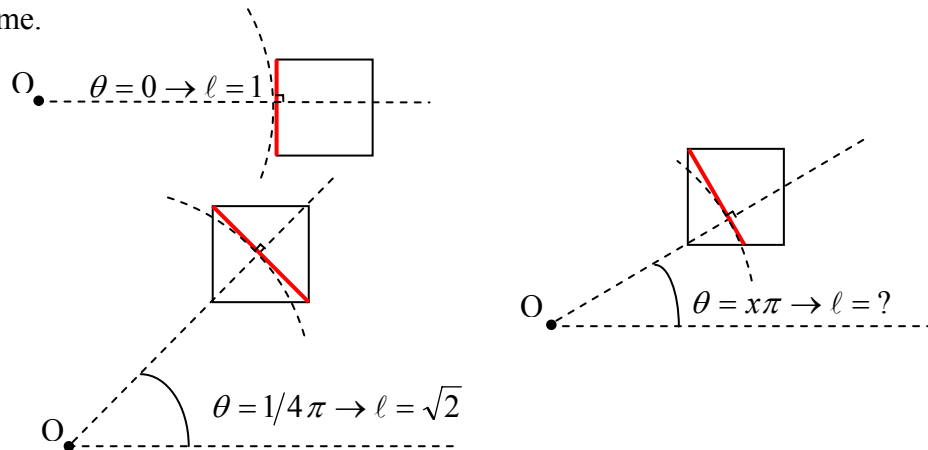
Code fragment 1 Cg code voor filtering met een rond filterbereik

Zoals te zien in Figuur 10b geeft dit algoritme al een sterke verbetering ten opzichte van het ongefilterde beeld. Het beeld lijkt hiermee goed gefilterd langs de horizontale en verticale assen van het cartesische stelsel, maar vertoont nog wat artefacten rond de diagonalen. Dit is zichtbaar in de vorm van enkele Moiré effecten.

4.2.1.2 Vierkant filterbereik

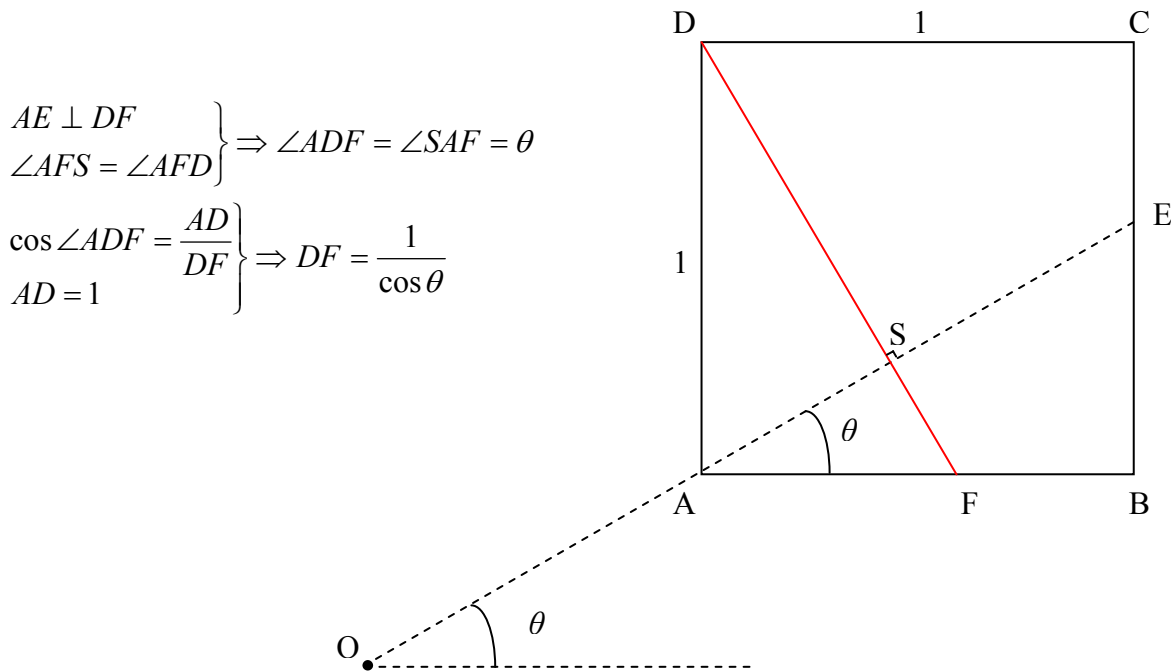
Nadere beschouwing van het algoritme met het ronde filterbereik toont ook aan dat er ook nog steeds informatie verloren kan gaan met deze manier van filteren. Dit valt te wijten aan het feit dat een beeld in het grafische geheugen wordt gerepresenteerd met vierkante pixels, en een diagonaal van een vierkante pixel een factor $\sqrt{2}$ groter is dan de zijden hiervan. Hierdoor bestrijkt een pixel die zich exact op een diagonaal bevindt, zoals bijvoorbeeld punt B in Figuur 9, $\sqrt{2}$ keer meer radarsectoren. Een probleem is alleen het bepalen van deze factor voor andere hoeken dan 0 of $\frac{1}{4}\pi$.

Hieronder wordt de eerste poging om deze correctiefactor te bepalen uitgewerkt. In Hoofdstuk 4.2.1.3 wordt aangetoond waarom deze niet geheel correct is, gevolgd door het uiteindelijke, correcte algoritme.



Figuur 11 Bepalen van correctiefactor afhankelijk van de hoek θ

Zoals Figuur 11 is de correctiefactor te geven door de lengte van de rode lijn te bepalen. Deze lijn is gedefinieerd als het maximale lijnstuk binnen de vierkante pixel dat loodrecht op lijn naar het middelpunt van de PPI staat. Aan de hand van Figuur 12 kan worden aangetoond dat de lengte van deze lijn te berekenen valt met de formule $1/\cos \theta$.



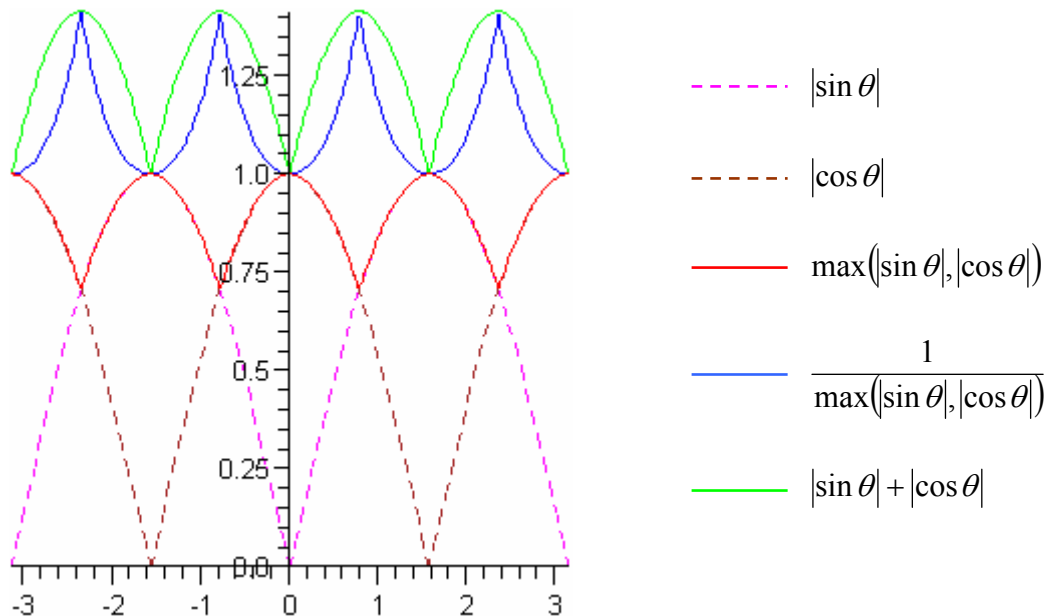
Figuur 12 Goniometrische bepaling van correctiefactor uit polaire coördinaten

Doordat de hoek θ als onderdeel van de polaire coördinaten na bepaling met de atan2 functie een waarde kan hebben in het domein van $[-\pi, \pi]$, kan de functie $1/\cos \theta$ echter niet als universele oplossing aangedragen worden, deze geldt uitsluitend voor $\theta \in [-1/4\pi, 1/4\pi]$. Voor alle waarden uit het domein $[-\pi, \pi]$ kunnen de volgende functies gebruikt worden:

$$\begin{aligned} \frac{1}{\cos(\theta + \pi)} & , \theta \in [-\pi, -\frac{3}{4}\pi] \\ \frac{1}{\cos(\theta + \frac{1}{2}\pi)} & , \theta \in [-\frac{3}{4}\pi, -\frac{1}{4}\pi] \\ \frac{1}{\cos(\theta)} & , \theta \in [-\frac{1}{4}\pi, \frac{1}{4}\pi] \\ \frac{1}{\cos(\theta - \frac{1}{2}\pi)} & , \theta \in [\frac{1}{4}\pi, \frac{3}{4}\pi] \\ \frac{1}{\cos(\theta - \pi)} & , \theta \in [\frac{3}{4}\pi, \pi] \end{aligned}$$

Deze functies worden in Figuur 13 weergegeven met de blauwe grafiek. In deze figuur zijn de inverse van deze grafiek in rood en de grafieken van $|\sin \theta|$ en $|\cos \theta|$ respectievelijk in magenta en bruine onderbroken lijnen weergegeven. Zoals te zien is kan de inverse van de functies in de samenstelling ook vervangen worden door de procedure: $c = \max(|\sin \theta|, |\cos \theta|)$.

Zoals vermeldt in de Literatuurstudie over GPU Programmeren [1] zijn *if-statements* relatief slecht uit te voeren op een GPU, daardoor verdient deze procedure grote voorkeur boven de eerder genoemde samenstelling van functies. De uiteindelijke correctiefactor kan dus voor alle waarden in het domein $[-\pi, \pi]$ bepaald worden door $\frac{1}{\max(|\sin \theta|, |\cos \theta|)}$. Met deze factor wordt een resultaat bereikt zoals te zien in Figuur 10c.



Figuur 13 Grafieken van correctiefactoren over het domein $[-\pi, \pi]$

De hiervoor genoemde procedure gaat uit van de hoek die gemaakt wordt met de x-as terwijl in het Cg programma reeds de hoek met de y-as berekend is, zie Hoofdstuk 3.2. Deze twee hoeken staan als volgt met elkaar in verband:

$$\theta_y = \frac{1}{2}\pi - \theta_x + 2k\pi \quad k \in \mathbb{N}$$

Een extra berekening voor de hoek met de x-as door het uitvoeren van $\arctan(y/x)$ kan voorkomen worden door in te zien dat de functie die door deze procedure gedefinieerd wordt, symmetrisch is op $x = k/2\pi$ $k \in \mathbb{N}$ en dus geldt $\max(|\sin \theta_x|, |\cos \theta_x|) = \max(|\sin \theta_y|, |\cos \theta_y|)$. Er kan dus voor verdere berekeningen zonder problemen de hoek met de y-as gebruikt worden. De volledige resulterende procedure staat weergegeven in Code fragment 2.

```
float4 radarSquareFP(float2 coords : TEXCOORD0,
                    sampler2D texture,
                    uniform float azimuthResolution,
                    uniform float radarRadius,
                    uniform float rotateOffset,
                    uniform float unfilteredOffset) : COLOR
{
    //polarCoords will hold the coordinates
    //of the best matching sample in the texture
    float2 polarCoords;

    //Determine range component for the current fragment location
    polarCoords.x = length(coords);

    //determine the theta of the angle
    //by calculating atan2(x,y), rotation will be given range [PI, -PI]
    //with theta = 0 at north direction and increasing clockwise
    float theta = myAtan2(coords.x, coords.y);

    //scale theta to range [-0.5,0.5] with following relation:
    //    [0    0.25  0.5/-0.5   -0.25]
    //    [N      E      S        W ]
    //Due to GL_REPEAT principle this results in normalized lookup
    //Then shift it with one half sweep width
    //to have sweep 0 centered on north direction
    polarCoords.y = theta/TWO_PI + 0.5/azimuthResolution;

    //determine the correction for difference in radial resolution
    //of a circle in a rectangular grid
    //the compensation for a possible rotation will need to be done
    //because of a non-uniform correction factor
    float2 mysincos;
    sincos(theta-rotateOffset, mysincos.x, mysincos.y);
    mysincos = abs(mysincos);

    float correction = max(mysincos.x, mysincos.y);

    //factor used to determine if a fragment is inside the PPI area
    //used to prevent unnecessary calculations
    //and to factor output to be black when outside the PPI
    float withinRadarRange = (polarCoords.x > 1 ? 0.0 : 1.0);

    //determine how many pixels will be covered by this fragment
    //determined by circumference of the circle with radius
    //as in polar coords, the width of the PPI, azimuth resolution
    //and the correction factor
    //factored with 'withinRadarRange' to prevent unnecessary
    //calculations
    float samplesCovered = withinRadarRange*azimuthResolution/
        (TWO_PI*polarCoords.x*radarRadius*correction);

    /* .... */
}
```

Code fragment 2 Cg code voor filtering met een vierkant filterbereik

In Figuur 10c is te zien dat deze manier van filteren een nog beter resultaat oplevert dan het ronde filterbereik. Aangetoond is dat met deze formule de horizontale, verticale en alle 45° diagonale lijnen zó gefilterd worden, dat hierbij geen informatie meer verloren gaat. Het verloop tussen deze lijnen vertoont echter toch nog enkele Moiré effecten.

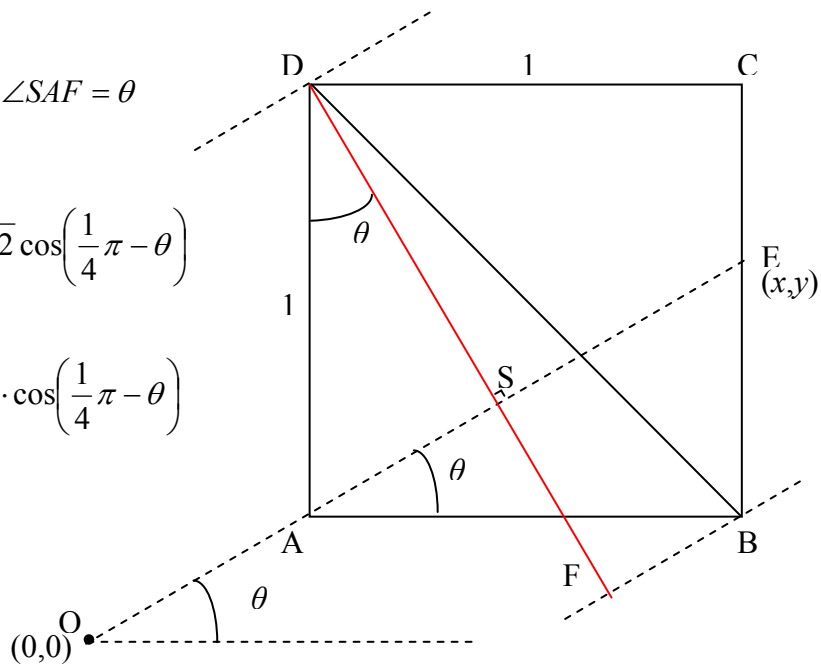
4.2.1.3 Bloemvormig filterbereik

In Figuur 12 is in feite geprobeerd te bepalen wat de grootte is van het aanzicht van een geroteerde vierkant (de rode lijn). Deze grootte ligt echter uitsluitend in geval van een recht zijaanzicht $\theta \in \{0, 1/2\pi, 3/2\pi, 2\pi\}$ of in geval van een diagonaal aanzicht, $\theta \in \{1/4\pi, 3/4\pi, 5/4\pi, 7/4\pi\}$ precies binnen het vierkant. De correcte bepaling voor alle andere hoeken kan gegeven worden door de twee lijnen parallel aan de kijkhoek zo te kiezen, dat het volledige vierkant zich precies binnen deze twee lijnen bevindt. De gezochte waarde is dan gelijk aan de afstand tussen deze twee lijnen. In Figuur 14 is aangetoond hoe deze afstand bepaald kan worden.

$$\left. \begin{aligned} AE \perp DF \\ \angle AFS = \angle AFD \end{aligned} \right\} \Rightarrow \angle ADF = \angle SAF = \theta$$

$$\left. \begin{aligned} \cos \angle FDB = \frac{DF}{DB} \\ DB = \sqrt{2} \\ \angle FDB = \frac{1}{4}\pi - \theta \end{aligned} \right\} \Rightarrow DF = \sqrt{2} \cos\left(\frac{1}{4}\pi - \theta\right)$$

$$\begin{aligned} \sqrt{2} \cos\left(\frac{1}{4}\pi - \theta\right) &= 2 \cos\left(\frac{1}{4}\pi\right) \cdot \cos\left(\frac{1}{4}\pi - \theta\right) \\ &= \cos(\theta) + \cos\left(\frac{1}{2}\pi - \theta\right) \\ &= \cos(\theta) + \cos\left(\theta - \frac{1}{2}\pi\right) \\ &= \cos(\theta) + \sin(\theta) \end{aligned}$$



Figuur 14 Berekening van correcte lijnstuk als correctiefactor

Ook hier geldt weer het probleem dat de gegeven formule uitsluitend geldig is voor een beperkt domein, in dit geval $\theta \in [0, 1/4\pi]$. Gelukkig kan er net als bij het hiervoor behandelde algoritme de algemeen geldende procedure bepaald worden door in te zien dat zowel de sinus als de cosinus een positieve waarde moet opleveren. Immers, het gaat hier om de lengte van een lijnstuk en deze kan nooit negatief zijn. Hiermee kan de procedure bepaald worden op $|\sin \theta| + |\cos \theta|$, in Figuur 13 is de functie die door deze procedure beschreven wordt met een groene lijn weergegeven. Wanneer deze correctiefactor wordt gebruikt voor de filtering over het azimut, wordt elke sample in het radarvideo precies één keer gebruikt. Het resultaat dat hierbij hoort is te zien in Figuur 10d. De Cg code die bij dit algoritme is weergegeven in Code fragment 3.

```
float4 radarCloverFP(float2 coords : TEXCOORD0,
                    sampler2D texture,
                    uniform float azimuthResolution,
                    uniform float radarRadius,
                    uniform float unfilteredOffset,
                    uniform float rotateOffset) : COLOR
{
    //polarCoords will hold the coordinates
    //of the best matching sample in the texture
    float2 polarCoords;

    //Determine range component for the current fragment location
    polarCoords.x = length(coords);

    //determine the theta of the angle
    //by calculating atan2(x,y), rotation will be given range [PI, -PI]
    //with theta = 0 at north direction and increasing clockwise
    float theta = myAtan2(coords.x, coords.y);

    //scale theta to range [-0.5,0.5] with following relation:
    //    [0    0.25  0.5/-0.5    -0.25]
    //    [N      E      S      W  ]
    //Due to GL_REPEAT principle this results in normalized lookup
    //Then shift it with one half sweep width
    //to have sweep 0 centered on north direction
    polarCoords.y = theta/TWO_PI + 0.5/azimuthResolution;

    //determine the correction for difference in radial resolution
    //of a circle in a rectangular grid
    //the compensation for a possible rotation will need to be done
    //because of a non-uniform correction factor
    float2 mysincos;
    sincos(theta-rotateOffset, mysincos.x, mysincos.y);
    mysincos = abs(mysincos);

    float correction = mysincos.x + mysincos.y;

    //factor used to determine if a fragment is inside the PPI area
    //used to prevent unnecessary calculations
    //and to factor output to be black when outside the PPI
    float withinRadarRange = (polarCoords.x > 1 ? 0.0 : 1.0);

    //determine how many pixels will be covered by this fragment
    //determined by circumference of the circle with radius
    //as in polar coords, the width of the PPI, azimuth resolution
    //and the correction factor
    //factor 'withinRadarRange' prevents unnecessary calculations
    float samplesCovered = withinRadarRange*azimuthResolution/
        (TWO_PI*polarCoords.x*radarRadius/correction);

    /* ... */
}
```

Code fragment 3 Cg code voor filtering met een bloemvormig filterbereik

4.2.1.4 Bepaling van concrete samples uit pixel-sample verhouding

Met behulp van de hierboven genoemde algoritmen is bepaald hoeveel samples er samengevoegd moeten worden voor de te tekenen pixel. Hiermee is echter nog niet gegeven welke samples uit de radarvideo texture dit moeten zijn.

Dit wordt bepaald door de dichtstbijzijnde radarsample, aangeduid door de berekende *polarCoords*, en uit de genoemde samples per pixel verhouding. De code die hiervoor verantwoordelijk is, staat vermeld in Code fragment 4. Deze kan worden toegevoegd aan de eerder gegeven code fragmenten voor bepaling van de hoeveelheid samples.

```
//filtering needs to take place if more than one pixel is covered
if (samplesCovered > 1)
{
    //determine the radius of horizontal convolution
    //all pixels are valued equally and max value will be used
    //fract will cover 'partial' pixels
    //that may exist on edges of the convolution
    float fract = (samplesCovered-1)*0.5;

    int radius = floor(fract);
    fract -= radius;

    float mymax = 0.0;

    float2 mycoords = polarCoords;

    //loop through the 'full-pixels-part' of the convolution
    //radius of 0 will only result in the lookup of center pixel
    for (int i = -radius; i <= radius; i++)
    {
        mycoords.y = polarCoords.y +
            ((float)i / azimuthResolution);
        mymax = max(tex2D(texture, mycoords).r, mymax);
    }

    //process the edges of the convolution
    //fraction is added to/subtracted of coordinate
    //nearest filtering will make sure correct sample is taken
    mycoords.y = polarCoords.y +
        ((float)(-radius-fract) /azimuthResolution);
    mymax = max(tex2D(texture, mycoords).r, mymax);

    mycoords.y = polarCoords.y +
        ((float)(radius+fract) /azimuthResolution);
    mymax = max(tex2D(texture, mycoords).r, mymax);

    return mymax;
}
else //otherwise just return the closest matching pixel
{
    //addition of unfilteredOffset only used
    //to visualize the area where samples are filtered
    float result = (tex2D(texture,polarCoords).r+unfilteredOffset)
        * withinRadarRange;

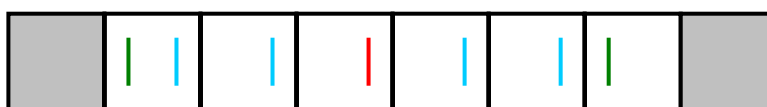
    return result;
}
}
```

Code fragment 4 Cg code voor azimuth filtering, en bepaling van maximum

In eerste instantie hoeft er uiteraard alleen verder gekeken te worden wanneer de hoeveelheid beslagen samples groter is dan 1. Indien dit niet het geval is, wordt de dichtstbijzijnde radarsample als resultaat opgeleverd. De keuze hiervoor, boven lineaire texture sampling, wordt gemotiveerd in Hoofdstuk 4.4.

Wanneer meer dan één sample gebruikt moet worden, wordt de hoeveelheid bestreken samples (*samplesCovered*) opgedeeld zoals in Figuur 15 schematisch is weergegeven. Dit voorbeeld gaat uit van 6 samples per pixel. Van dit getal wordt 1 afgetrokken, dit is de sample in het centrum. De 5 samples die dan nog over zijn, moeten in gelijke mate van de linker en de rechterzijde van deze eerste sample genomen worden. Deze wordt daarom door 2 gedeeld, waardoor links en rechts nog 2.5 samples bestreken moeten worden. Van dit getal wordt de hoeveelheid buiten de fractie, in dit geval 2, als onder en bovengrens voor een for-loop opgegeven. Hierdoor worden alle samples die op $[-2, -1, 0, 1, 2]$ samples afstand van het midden meegenomen. Dit resulteert in de rode en de vier blauwe samples in Figuur 15. Tenslotte worden de twee samples genomen die zich op $[-2.5, 2.5]$ samples afstand van het midden bevinden. Zoals in Figuur 15 te zien is, hoeft dit niet altijd het nemen van een nieuwe sample te betekenen. Hier is de linker sample bijvoorbeeld twee keer genomen.

SamplesCovered = 6
Radius = 2
Fraction = 0.5



Figuur 15 Schematische weergave van sampling bij 6 samples per pixel

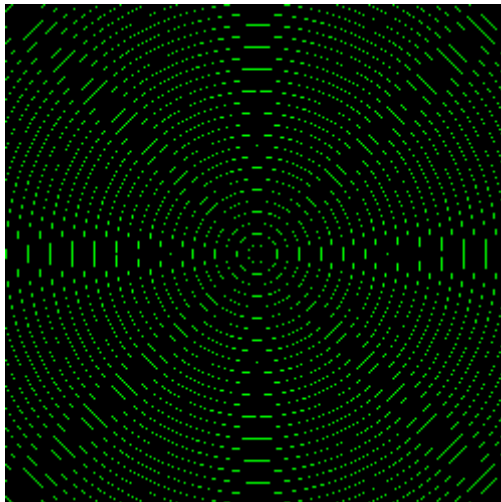
Een opmerking hierbij is wel dat er mogelijk ongewenste effecten kunnen ontstaan rond het punt waar nieuw radarvideo beschikbaar is gekomen. Wanneer de radar precies door het noorden gaat, is alle data links van die locatie het meest recent. De data rechts hiervan is echter al bijna een gehele radarscan verouderd. Wanneer de data altijd op de hierboven genoemde wijze wordt verwerkt, wordt er dus rond de genoemde grens zowel oude als meer recente data gebruikt.

Dit is een aandachtspunt waar verder geen rekening mee is gehouden in het onderzoek. Verder onderzoek moet uitwijzen of dit werkelijk een probleem is. In dat geval kan bijvoorbeeld besloten worden om bij deze samplebepaling uitsluitend samples mee te nemen die hooguit een halve omwentelingstijd oud zijn.

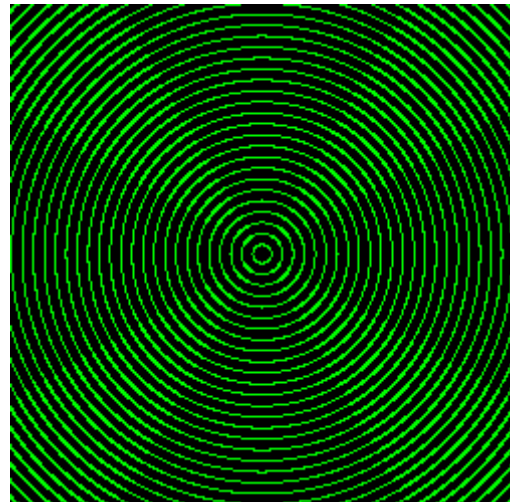
4.2.2 Pixel-sample verhouding over de afstand

Naast de filtering over het azimut speelt ook filtering over de afstand een belangrijke rol bij mogelijk verlies van radarsamples in de weergave. Dit probleem treedt op wanneer het aantal samples over de afstand groter is dan de radius van de PPI in pixels. Om dit probleem te visualiseren, wordt wederom gebruik gemaakt van radarvideo met daarin 2048 sectoren en 1024 range samples per sector, afgebeeld op een PPI met een radius van 350 beeldpunten.

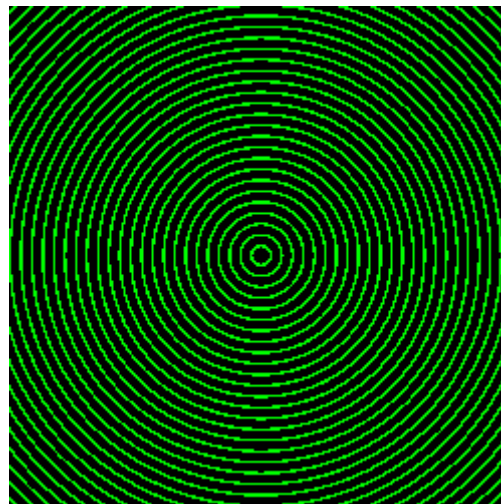
Daar de oorzaak van dit probleem nauw verwant is met het probleem van dataverlies over het azimut, is in eerste instantie ook uitgegaan van de in Hoofdstuk 4.2.1.3 genoemde oplossing, die voor het probleem over het azimut de beste resultaten leverde. Over de afstand komt de procedure neer op het bepalen van de hoeveelheid samen te voegen samples door de totale hoeveelheid samples per sector te delen door de radius van de PPI in pixels. Deze waarde wordt vervolgens vermenigvuldigd met de in Hoofdstuk 4.2.1.3 bepaalde hoeksafhankelijke correctiefactor.



a) Ongefilterde weergave



b) Ongewenste filtering op basis
van azimuth algoritme



c) Filtering met het aangepaste
filter algoritme

Figuur 16 Resultaten van verschillende afstand normalisatie algoritmen

Deze methode kan echter in sommige gevallen nog ongewenste effecten opleveren zoals te zien is in Figuur 16b. Hierbij is zeer duidelijk te zien dat de hoeveelheid samen te voegen samples (afgerond naar boven) variabel is over het azimuth. In dit geval zijn duidelijk ‘banen’ te herkennen, die in meer dan wel mindere mate gefilterd zijn. Aangezien deze lokale afwijkingen in intensiteit de operator kunnen storen/afleiden, is daarom toch gekozen voor een uniforme filtering. Deze filtering past ongeacht de hoek waarop een sample zich bevindt altijd dezelfde mate van filtering toe. Deze is gelijk gesteld aan de worstcase van het mathematisch correcte filter, omdat het minder erg is samples te dupliceren dan ze verloren te laten gaan. De hoeveelheid samples per pixel worden dus vermenigvuldigd met een correctiefactor van $\sqrt{2}$. Het resultaat hiervan is te zien in Figuur 16c. De totale hoeveelheid samengevoegde samples is hiermee gelijk aan $\frac{s}{r}\sqrt{2}$, waarbij s de hoeveelheid samples per sector is, en r de radius van de PPI.

4.2.2.1 Bepaling van concrete samples uit pixel-sample verhouding

Bij filtering over de afstand zijn de hoeveelheid samples per pixel onafhankelijk van zowel de afstand als de azimuth. Deze eerste komt doordat de verdeling van radarsamples over de radius van de PPI uniform moet zijn. De tweede is voortgekomen uit de keuze van het algoritme om de vermelde ongewenste afwijkingen in intensiteit te vermijden.

Deze onveranderlijkheid per pixel heeft als voordeel dat het gebruikte fragment programma de bepaling van concrete samples niet per pixel hoeft uit te voeren. In plaats daarvan kan deze berekening hoger in pipeline worden gelegd, namelijk bij de CPU. Deze kan dan de berekende parameters als uniformen meegeven aan het fragment programma. Hiermee kan voorkomen worden dat dezelfde berekening duizenden malen per scan worden uitgevoerd. In plaats daarvan hoeft het feitelijk slechts éénmaal berekend te worden, namelijk bij het selecteren van de radar en de grootte van het scherm waarin het afgebeeld wordt.

```
double factor = sqrt(2.0)*
               _sac/(parent->getRadarRadius()*parent->getZoom());
factor = (factor <= 1.0 ? 0.0 : (factor-1.0)*0.5);

/* ... */
cgGLSetParameter4f(_normalizationFactors,    // uniform parameter name
                   _sac,                      // #samples per sector
                   256.0/((double)_amr+1.0),  // normalization factor
                   -(float)ceil(factor),       // Lower border filter area
                   (float)floor(factor));      // Upper border filter area
```

Code fragment 5 C++ code voor het bepalen van filter gebied voor afstand filtering

```
/* normalizationFactors components:
   normalizationFactors.x = SAC
   normalizationFactors.y = AMRFactor
   normalizationFactors.z = lower border filter area
   normalizationFactors.w = upper border filter area
*/
float4 rangeNormalizePeakFP(float2 coords : TEXCOORD0,
                           sampler2D texture,
                           uniform float4 normalizationFactors) : COLOR
{
    float mymax = 0.0;

    float2 mycoords = coords;

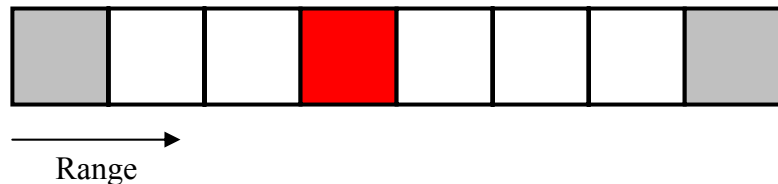
    //process relative neighboring samples convolution
    for (float i = normalizationFactors.z;
         i <= normalizationFactors.w; i++)
    {
        mycoords.x = coords.x + (i / normalizationFactors.x);
        mymax = max(tex2D(texture, mycoords).r, mymax);
    }

    return mymax*normalizationFactors.y;
}
```

Code fragment 6 Cg code voor afstand filtering, en bepaling van maximum

De gebruikte C++ code hiervoor is te zien in Code fragment 5. De verdere verwerking, gedaan door het Cg fragment programma, is gegeven in Code fragment 6. In eerste instantie wordt de sample per pixel verhouding berekend volgens de in Hoofdstuk 4.2.2 genoemde formule. Indien deze factor kleiner of gelijk is aan 1, is het resultaat voor boven en ondergrens van het filterbereik gelijk aan 0. Dit resulteert in het gebruik van uitsluitend de dichtstbijzijnde sample. In alle andere gevallen wordt de helft van deze factor min 1 (de middelste sample) gebruikt om deze te bepalen. De fractie die dan overblijft bepaalt, afgerond naar boven, de hoeveelheid samples op grotere afstand van het schip. Afgerond naar beneden geeft dit de hoeveelheid samples op kleinere afstand van het schip. Deze waarden worden als delta-indices gebruikt in een for-loop in het Cg programma. In dit voorbeeld wordt het maximum van alle samples bepaald. De volgorde is gekozen omdat de textuur coördinaten altijd het midden van een pixel, oftewel $x.5$ aangeven. Sampling naar pixel $(x.5 + 0.5)$ levert pixel $(x + 1)$ op, terwijl sampling naar pixel $(x.5 - 0.5)$, net als sampling naar pixel $(x.5)$, pixel (x) oplevert.

SamplesCovered = 6
Fraction = 2.5
Lower Border = 2
Upper Border = 3



Figuur 17 Schematische weergave van sampling bij 6 samples per pixel

4.3 Sample combinatie algoritmen

Nadat alle exacte samples zijn bepaald, kan de uiteindelijke waarde voor de pixel op het scherm berekend worden. De gekozen samples kunnen echter op verschillende wijzen worden samengevoegd. In de volgende paragrafen worden de meest gekozen methoden en hun voor- en nadelen beschreven.

4.3.1 Gemiddelde waarde

Een voor de hand liggende methode is het bepalen van de gemiddelde waarde van alle samples. Dit heeft echter de vervelende eigenschap dat, in geval meer dan één sample beschouwd wordt, vrijwel nooit een exact gemeten waarde op het scherm afgebeeld wordt. In het extreemste geval wordt een sample met maximale intensiteit afgevlakt en verdwijnt het in de ruis wanneer het omgeven is door samples met een lage intensiteit.

4.3.2 Gemiddelde met threshold

Het bepalen van een gemiddelde met threshold is te zien als een variatie op het hierboven beschreven gemiddelde. Hierbij worden uitsluitend samples in het gemiddelde opgenomen, die een hogere intensiteit hebben dan de opgegeven threshold. Wanneer de threshold hoog genoeg gekozen wordt, vermindert het probleem van afvlakking van hoge intensiteit samples.

Het bepalen van deze threshold maakt deze methode echter moeilijk generiek toe te passen. Wanneer de threshold te laag is, neemt het probleem van de standaard middeling methode toe. Naarmate deze hoger gekozen wordt, komt het resultaat steeds dichterbij het resultaat van de maximum methode. Daarnaast moet er tevens nagedacht worden over wat het resultaat moet zijn wanneer alle samples zich onder de threshold bevinden.

4.3.3 Maximum waarde

Deze laatst genoemde methode bepaalt van alle samples het maximum. Deze methode heeft als voordeel dat op deze wijze geen enkele sample met hoge intensiteit verloren kan gaan. Het nadeel hierbij is dat rond het centrum van de PPI, waar veel samples samengevoegd worden, de kans zeer hoog is dat het beeld dicht smeert. Dit doordat er slechts één sample een (afwijkend) hoge intensiteit hoeft te hebben om een pixel een hoge intensiteit te geven.

Deze methode geniet ondanks dit nadeel toch de grootste voorkeur. Het probleem dat zich rond het midden voordoet heeft in veel gevallen bovendien weinig invloed. Immers, hier zit doorgaans toch al veel *clutter* in het radarvideo.

4.4 Kanttekeningen bij de gedane texture sampling

Wanneer de coördinaten voor de te nemen sample volgens één van de in dit hoofdstuk behandelde methoden bepaald zijn, kunnen deze aan de texture sampler aangeboden worden om als index binnen de afbeelding gebruikt te worden. Op basis hiervan wordt een sample genomen uit het texture geheugen. Het nemen van deze sample kan echter nog op verschillende wijzen beïnvloed worden. Hieronder worden een aantal van deze methoden behandeld, die relevant zijn voor de gedane radarscan conversie.

Als eerste heeft de gebruiker invloed op de wijze waarop er wordt omgegaan met texture coördinaten, die niet exact het midden van een sample aanwijzen. In dit geval kan de texture sampler een filtering toepassen. Standaard wordt hiervoor een lineaire filtering gebruikt. Voor de hier besproken algoritmen is dit echter ongewenst. Hierbij worden immers de exacte waarden van de radarsamples aangetast, wat verlies van informatie of weergave van valse informatie tot gevolg kan hebben. Om dit te voorkomen wordt bij alle gedefinieerde textures interpolatie volgens het *GL_NEAREST* principe toegepast. Hierdoor wordt altijd de dichtstbijzijnde, exacte radarsample opgeleverd. Hiermee kan de operator, met behulp van het gekozen fragment programma, zelf de keuze maken voor de toegepaste vorm van filtering. Zie voor meer details hierover Hoofdstuk 4.3.

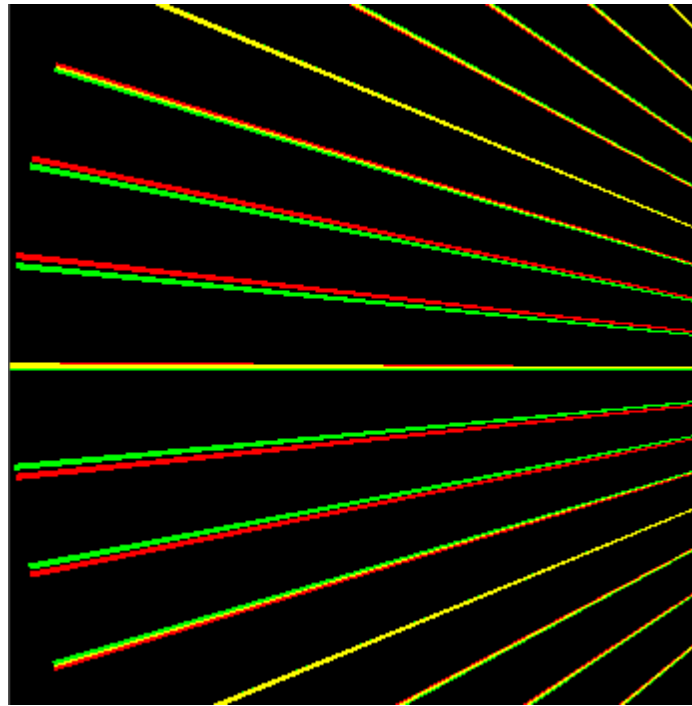
Daarnaast kan ingesteld worden hoe de sampler moet omgaan met texture coördinaten buiten het bereik [0..1]. In Code fragment 1 is te zien dat de coördinaat van de azimuth met behulp van de *atan2* functie bepaald wordt in het bereik [-0.5..0.5], met 0 in het noorden. Gezien de azimuth cyclisch is, kan voor indexering van de texture in deze richting *GL_REPEAT* gebruikt worden. Hierdoor is het onnodig deze coördinaten handmatig te normaliseren. Zoals weergegeven in Figuur 18 kunnen de coördinaten zonder aanpassing gebruikt worden voor indexering van de B-Scope texture. De verdeling van de coördinaten [0..1] verloopt dan kloksgewijs met 0 in het noorden. De afstand richting heeft niet een dergelijke cyclische eigenschap. Hiervoor wordt dan ook *GL_CLAMP* indexering gebruikt.

Reële texture coördinaat	-1	-0.75	-0.5	-0.25	0	0.25	0.5	0.75	1
GL_REPEAT interpretatie	1 / 0	0.25	0.5	0.75	1 / 0	0.25	0.5	0.75	1 / 0

Figuur 18 GL_REPEAT interpretatie van texture coördinaten buiten het bereik [0..1]

4.5 Ondervonden problemen met arctangens

Ondanks het feit dat de gebruikte arctangens methode wiskundig gezien correct is, zijn er toch afwijkingen geconstateerd wanneer deze methode op de GPU uitgevoerd wordt. Dit is het duidelijkst te zien rondom de horizontale en verticale assen van de PPI. In Figuur 19 is dit probleem weergegeven. Hierin zijn de correcte resultaten in groen en de incorrecte, met standaard $\text{atan2}(y,x)$ functie berekende resultaten, in rood weergegeven. Waar deze twee overlappen is het resultaat geel.



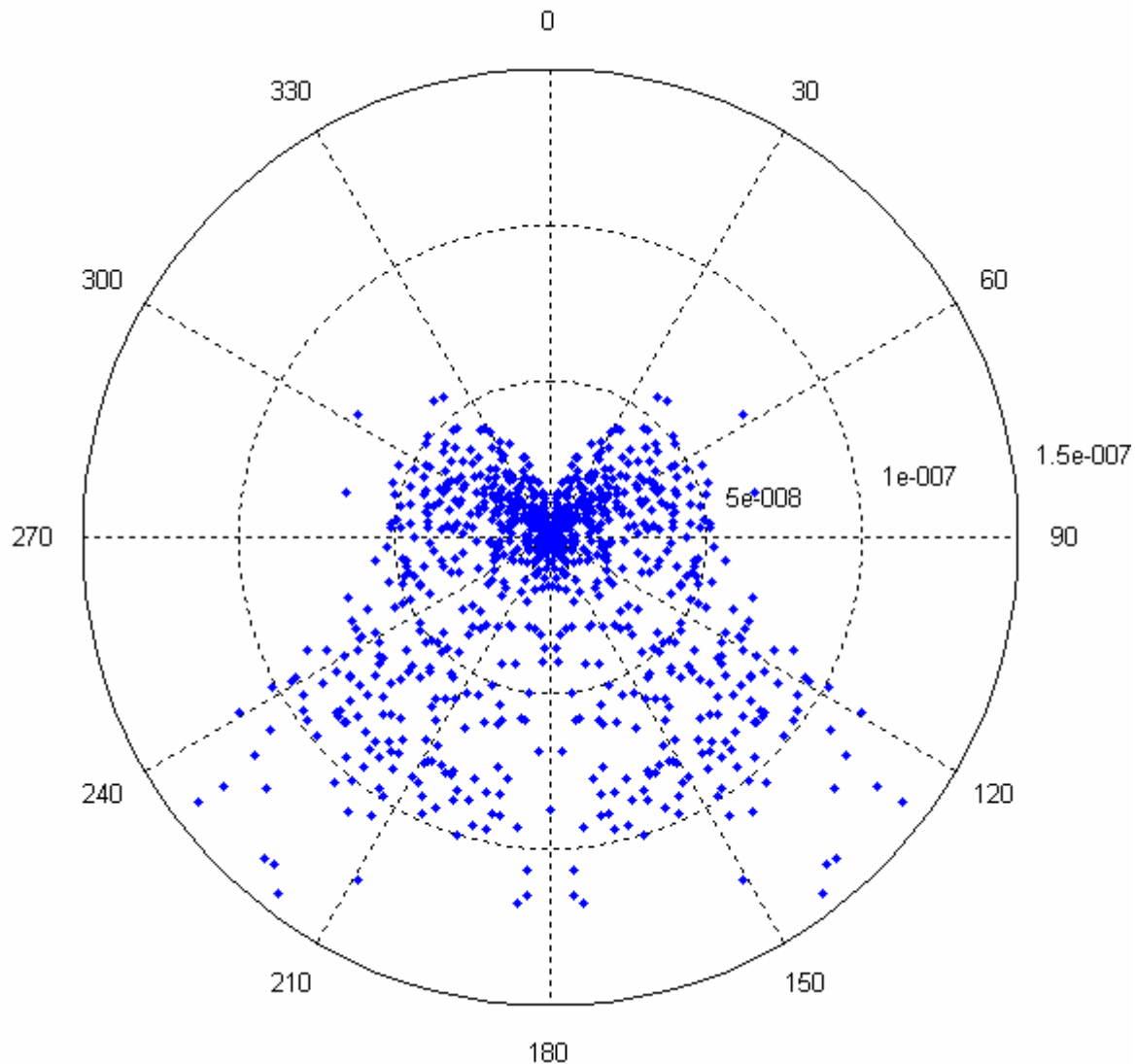
Figuur 19 Afwijking van de met $\text{atan2}(y,x)$ berekende coördinaten in rood

Aangezien door deze fout de radar echo's misplaatst kunnen worden en daardoor tot foute conclusies bij de operator kan leiden is het belangrijk de oorzaak van deze afwijking te constateren en hiervoor een oplossing te bedenken.

4.5.1 Analyse van het arctangens probleem

Als eerste is onderzocht of de 'beperkte' precisie van de intern gebruikte 32 bits float (gelijk aan de door IEEE gedefinieerde standaard 754) een mogelijke oorzaak hiervoor zou kunnen zijn. Deze gedachte is ontstaan doordat de afwijking het sterkst zichtbaar is rond de horizontale en verticale as. Gezien hier de x en of y component naar 0 nadert en de relatieve afwijking groter is bij dergelijke waardes.

Deze hypothese is onderzocht met behulp van Matlab. Hierin is het verschil tussen de berekening in single en double precisie uitgezet tegen het azimuth. Doordat de afwijking in pixels groter wordt naarmate de radius van de radar toeneemt, is in eerste instantie gekeken naar een PPI met een radius van 600 pixels omdat de maximale resolutie van de schermen gebruikt in de consoles 1600x1200 is. Een PPI met radius van 600 pixels kan hierop dus nog volledig worden weergegeven. Voor grotere PPI weergaven kan deze afwijking lineair geschaald worden.



Figuur 20 Verschil tussen double en single precisie berekeningen in pixels

In Figuur 20 is het resultaat van deze meting te zien. De Matlab code die deze figuur heeft voortgebracht is te zien in Code fragment 7. Uit de figuur is af te lezen dat de maximale afwijking veroorzaakt door de single precisie nog geen $1.5 \cdot 10^{-7}$ pixels bedraagt. Hieruit volgt dat de afwijking in pixels aan de rand van een PPI met een straal van 600 pixels minder is dan aan $9 \cdot 10^{-5}$ pixels. Gezien de werkelijke afwijking in een PPI met een straal van 600 pixels toch wel op 6 pixels te meten valt kan dit dus niet de (enige) oorzaak van het probleem zijn.

Hierna is de hypothese gesteld dat de NVIDIA implementatie van *atan2* dan wel de hierin gebruikte *atan* functie voor deze toepassing niet nauwkeurig genoeg is. Om deze hypothese te toetsen is er gezocht naar alternatieven waarbij deze twee functies handmatig en op een correcte manier geïmplementeerd worden.

```
x=[]; y=[]; %initialize empty arrays

for row = -1:0.05:1 %generate cartesian coordinates
    for col = -1:0.05:1
        x = [x row];
        y = [y col];
    end
end

%make polar plot of difference between
%double and single calculation
polar(atan2(y,x),abs(atan2(y,x)-single(atan2(single(y),single(x)))),'.');

camorbit(90,180); %rotate image to match PPI orientation
```

Code fragment 7 Matlab code voor single/double precisie berekening

Hiervoor is begonnen met een alternatief voor de *atan2* functie. Deze functie bepaalt op correcte wijze het kwadrant waarin de (x,y) coördinaten zich bevinden en maakt gebruik van de *atan* functie van NVIDIA. Wanneer deze implementatie gebruikt wordt, is er echter geen verbetering in het resultaat te constateren. Er kan dus geconstateerd worden dat de *atan2* functie wel op correcte wijze het kwadrant bepaald.

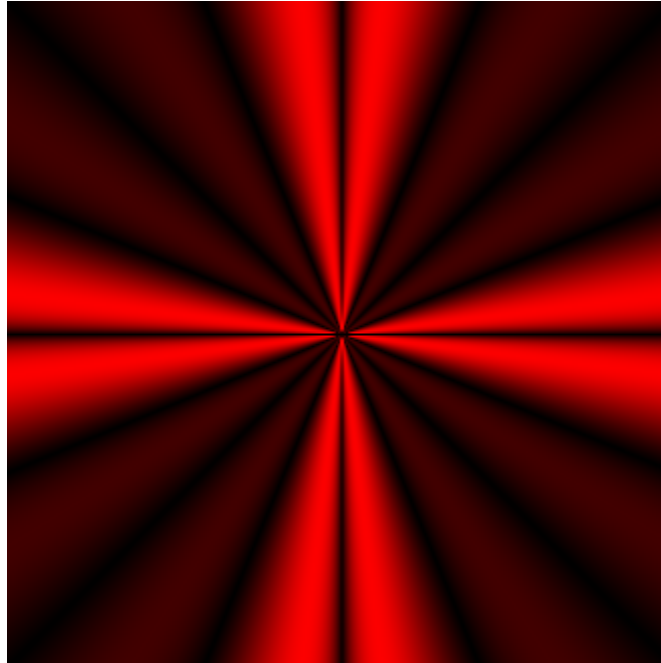
Interessantere resultaten zijn echter behaald bij de alternatieve implementatie van de arctangens functie. Hiervoor is gebruik gemaakt van de volgende goniometrische identiteit:

$$\arctan(x) = \arcsin\left(\frac{x}{\sqrt{1+x^2}}\right)$$

Wanneer in de eerder gemaakte implementatie van *atan2* niet de *atan* van NVIDIA aangeroepen wordt maar deze functie gebaseerd op *arcsin*, is het resultaat wel correct.

Om het absolute verschil tussen deze twee implementaties te bepalen is dit absolute verschil als uitvoer van het fragment programma genomen. Gezien de GPU niet in staat is om resultaten van berekeningen in een fragment programma in getalvorm weer te geven, moet dit resultaat omgezet worden naar een kleurwaarde. Deze moet daarmee ook genormaliseerd zijn naar het bereik [0,1]. In dit geval is zelfs het maximale verschil echter een zeer kleine waarde. Deze wordt hierdoor dan ook gerepresenteerd door een nauwelijks van zwart te onderscheiden kleur. Om dit verschil toch te kunnen bepalen, wordt gezocht naar een factor *x* die dit maximale verschil ongeveer gelijk aan 1 maakt. Wanneer deze factor gevonden is, wordt het werkelijke verschil bepaald door 1/*x*. Het bepalen van deze factor kan vergemakkelijkt worden door de uitvoerkleur uitsluitend wit te laten zijn wanneer de afwijking vermenigvuldigd met *x* groter of gelijk aan 1 is en zwart in alle andere gevallen door in de uitvoer gebruik te maken van een conditionele expressie.

Proefondervindelijk is bepaald dat bij factor 98 nog net geen maximale kleurintensiteit bereikt wordt waar er bij factor 99 al meerdere lijnen zijn met de maximale intensiteit. Dit betekent dat het maximale verschil tussen de correcte en incorrecte implementatie tussen 1/98 en 1/99 ligt. In Figuur 21 is het verloop van deze fout vermenigvuldigd met 98 zichtbaar gemaakt in een polair coördinatenstelsel. De afwijking in pixels kan met deze factor gegeven worden door $\tan(1/98.5) \cdot r$ met *r* gelijk aan de straal van de PPI. Hiermee kan bepaald worden dat het verschil in een PPI met een radius van 600 pixels gelijk is aan $\tan(1/98.5) \cdot 600 \approx 6$ pixels. Dit komt inderdaad overeen met het verschil dat op het scherm te meten is.



Figuur 21 Verschil tussen standaard en alternatieve arctangens vermenigvuldigd met 98

De oorzaak van deze afwijking kan gezocht worden in het feit dat goniometrische functies niet exact bepaald (kunnen) worden in de hardware maar in plaats hiervan benaderd worden door bijvoorbeeld een n^{e} orde Taylor polynoom. Schijnbaar wordt er voor de arctangens een lagere orde van deze polynoom gebruikt voor de benadering dan voor de arcsinus. Anderzijds is het mogelijk dat voor beide functies dezelfde orde Taylor polynoom is gebruikt en dat het verschil veroorzaakt wordt doordat dat de arctangens functie een hogere orde vereist voor dezelfde mate van nauwkeurigheid van de benadering. Dit vanwege het feit dat de arctangens een asymptotisch verloop heeft.

4.5.2 Implementatie van de geboden oplossing

Zoals in de vorige paragraaf is uitgelegd is de *atan* functie de echte oorzaak van het probleem. Om hiervoor een alternatieve implementatie in gebruik te nemen moet echter ook de *atan2* functie opnieuw geïmplementeerd worden. Deze roept immers normaliter NVIDIA's eigen *atan* functie aan. Om de implementatie van deze twee functies in gebruik zo efficiënt mogelijk te maken, is ervoor gekozen deze te implementeren als C functie-macro's. De gemaakte implementatie hiervoor is te zien in Code fragment 8.

```
#define myAtan(x) asin(x/(sqrt(1+ x * x )))

#define myAtan2(y,x) (x >= y ? \
    (x >= -y ? \
        myAtan( y / x ) : \
        myAtan(-x / y) - HALF_PI) : \
    (x >= -y ? \
        myAtan(-x / y) + HALF_PI : \
        (y > 0.0 ? \
            myAtan( y / x ) + PI : \
            myAtan( y / x ) - PI)))
```

Code fragment 8 myAtan en myAtan2 definities

De eigen arctangens functie roept simpelweg de bestaande arcsinus functie aan met een aangepast argument. Het geboden alternatief voor de *atan2* functie is echter iets ingewikkelder aangezien hiervoor naast het resultaat van de arctangens, ook een correctie bepaald moet worden afhankelijk van het kwadrant waarin de (x,y)-coördinaten zich bevinden. Daarnaast moet het resultaat een correcte waarde krijgen in geval één van de twee coördinaten 0 is. In de bovenstaande implementatie worden alle gevallen correct afgehandeld met een minimale hoeveelheid condities.

Deze condities zijn om verschillende redenen geïmplementeerd als een conditionele expressie. Ten eerste wordt het hierdoor mogelijk om het resultaat van de functie als onderdeel van een grotere expressie te gebruiken. Dit zou niet mogelijk zijn wanneer de functie als een aaneenschakeling van if-else statements geïmplementeerd is.

Verder wordt deze implementatie efficiënter uitgevoerd op de GPU, dan combinaties van if-else statements. Op het eerste gezicht lijkt dit wat moeilijk te geloven gezien alle mogelijke ‘takken’ van een conditionele expressie geëvalueerd worden. Op basis van de condities wordt er hieruit slechts één gekozen als resultaat, terwijl voor het alternatief uitsluitend condities worden geëvalueerd totdat de correcte tak gevonden is, waarna het bijbehorende statement wordt uitgevoerd. Toch is deze optie in de praktijk langzamer dan de gekozen implementatie.

De verklaring hiervoor is te geven door na te gaan dat de fragment processors in de GPU het *SIMD* dataverwerkingsmodel hanteren [1]. Dit betekent dat dezelfde operatie uitgevoerd wordt op (een sub-set van) de complete dataset. Wanneer zich hierin zowel elementen bevinden die voldoen aan conditie C_A als elementen die voldoen aan conditie C_B , dan worden de bijbehorende statements S_A en S_B op alle elementen in die dataset uitgevoerd. Wanneer de lokaliteit van deze condities sterk verspreid is, wordt er ten opzichte van de conditionele expressie geen enkele snelheidswinst behaald. Sterker nog, door de relatief grote overhead die deze if-else statements kosten in de huidige generaties GPUs, wordt de geleverde prestaties enkel minder [1]. Uit metingen met beide methoden is dan ook gebleken dat de implementatie als conditionele expressie de beste prestaties levert.

De in Code fragment 8 gegeven implementatie lijkt dus de meest efficiënte methode om de *atan2* en bijbehorende *atan* functie op meer nauwkeurige wijze de GPU te bepalen. Deze methode is natuurlijk nog altijd minder efficiënt dan de functie zoals deze in de GPU als instructie is geïmplementeerd. De impact die deze operatie heeft op de prestaties van het complete systeem is nauwelijks te meten maar wordt geschat op een vermindering van ongeveer 1% op het in Hoofdstuk 1.2 vermeldde testsysteem. Dit is dan ook een zeer acceptabele inruil voor de verbeterde weergave.

4.5.3 Probleem door NVIDIA opgelost

Enige tijd na het onderzoek naar dit probleem en de implementatie van de ontwikkelde oplossing, bracht NVIDIA de Cg Toolkit 1.5 beta uit. Tests met deze Toolkit in combinatie met de nieuwste driver hebben uitgewezen dat het geconstateerde probleem met de *atan2* functie hierin opgelost is. Het is dus niet meer nodig de geboden oplossing te hanteren. In plaats daarvan kan de reguliere *atan2* functie gebruikt worden.

Hoe dit probleem precies is opgelost is helaas niet duidelijk gemaakt. Het is echter een feit dat met een wijziging van driver en Cg Toolkit geen hardwarematige bugs of onnauwkeurigheden opgelost kunnen worden. Dit wijst erop dat de door NVIDIA geboden oplossing een soortgelijk karakter heeft aan de in Hoofdstuk 4.5.2 geboden oplossing.

5 Op dit moment gebruikte RSC en architecturen

Er zijn vele verschillende radars gemaakt, elk met eigen voor en nadelen. Zo zijn er bijvoorbeeld radars die geschikt zijn voor detectie op korte afstand voor bijvoorbeeld navigatie of kust surveillance (bv. SCOUT Mk2), radars voor de middenlange afstanden (bv. VARIANT) of voor de lange afstanden (bv. SMART-L).

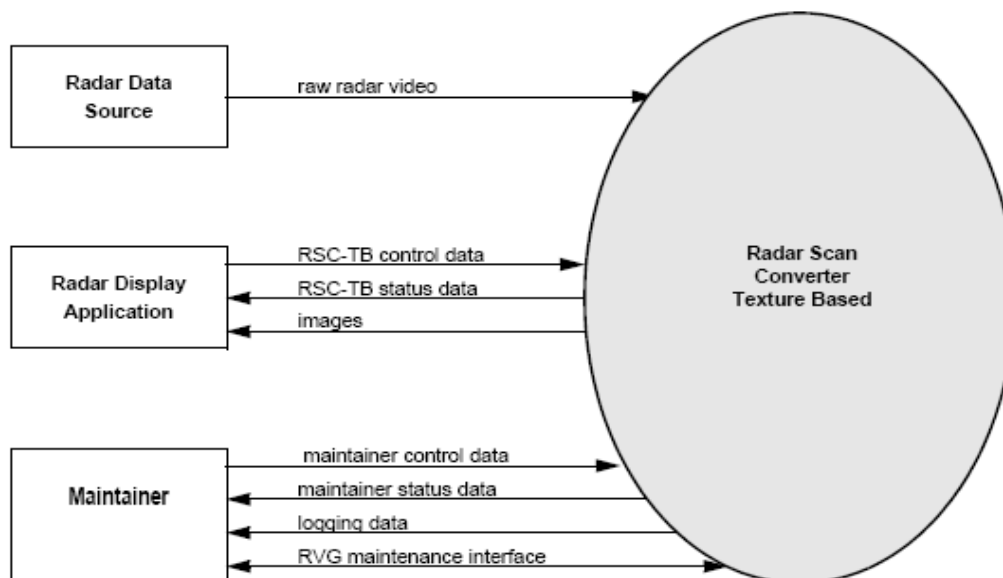


Figuur 22 Enkele voorbeelden van radars

Doordat de huidige RSC in staat moet zijn radarvideo van (vrijwel) alle momenteel gebruikte radars te verwerken is deze ontworpen met een breed scala aan instelmogelijkheden. Ook is het mogelijk om radars via verschillende distributiekkanalen aan te sluiten. Vroegere radars waren allemaal analoog en hebben daarmee ook een analoge distributie van de radarvideo. Nieuwe radars worden echter ontwikkeld voor productie van digitaal radarvideo. In dit hoofdstuk wordt een overzicht gegeven van de functies en mogelijkheden van de RSC-TB. Daarnaast worden de twee distributie architecturen nader uitgelegd.

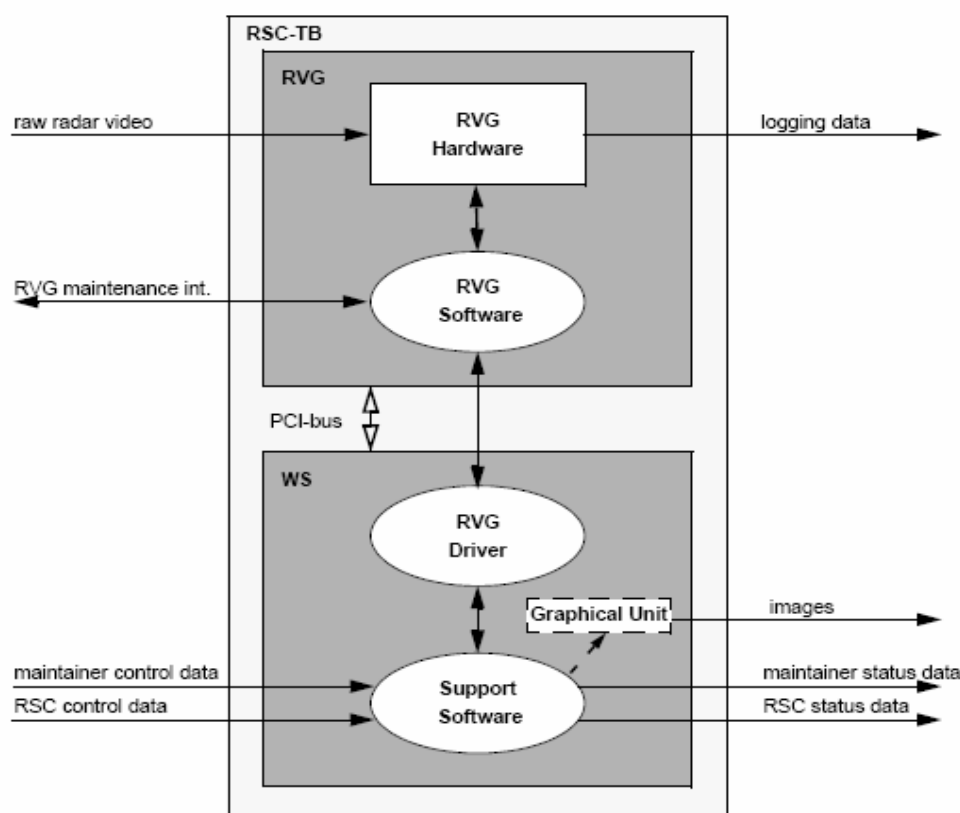
5.1 Functieoverzicht RSC-TB

De *RSC-TB* is een Texture Based Radar Scan Convertor, een systeem dat uitgaande van polair georiënteerde (B-Scope) ruwe radarvideo een cartesisch georiënteerde weergave kan geven (PPI). De functionele context van de RSC-TB is te zien in Figuur 23.



Figuur 23 Functioneel context diagram van de RSC-TB [2]

Het systeem bestaat in feite uit een hardware deel in de vorm van een PCI insteekkaart (*RVG*) en een stuk software dat op een *COTS* werkstation (*WS*) draait. Een globaal overzicht van de functionele data flow in dit systeem is zichtbaar in Figuur 24.



Figuur 24 Functionele data flow van de RSC-TB [2]

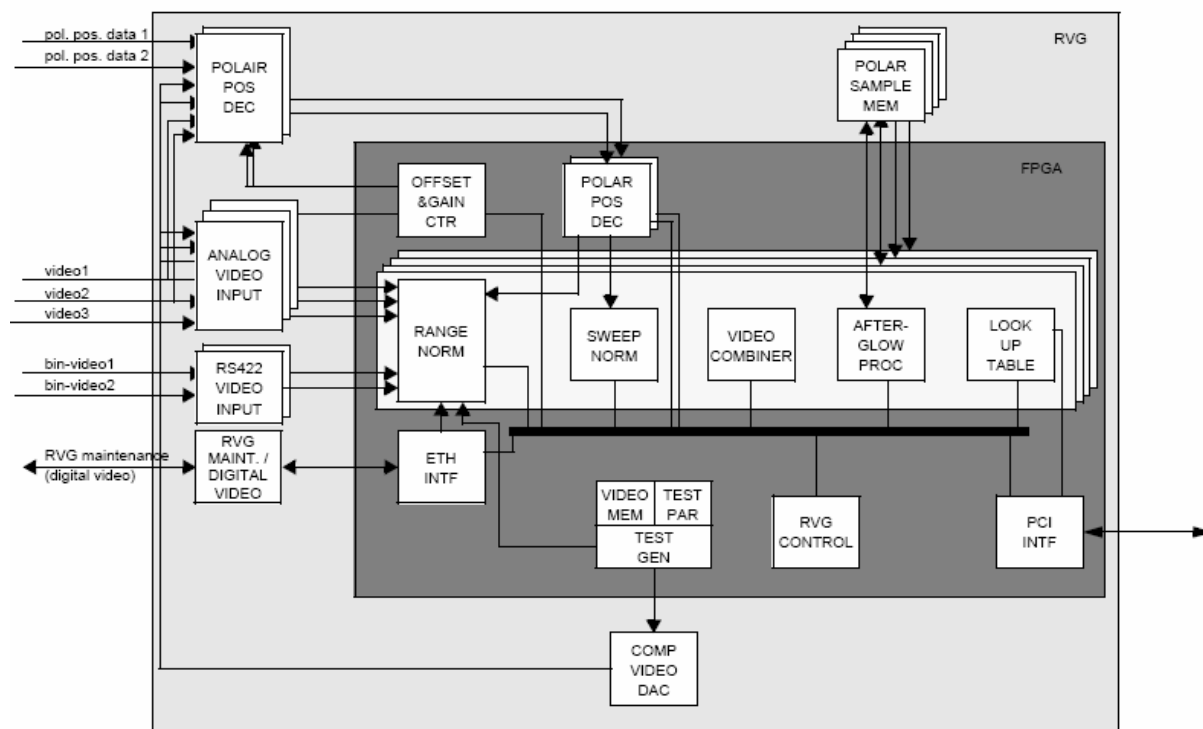
In de volgende sub-hoofdstukken worden de belangrijkste taken en restricties van zowel de hardware als de software kant van de RSC-TB behandeld. Hiermee wordt een indicatie gegeven van wat een nieuw te ontwikkelen RSC minimaal zou moeten kunnen.

5.1.1 RVG Hardware

De RVG is zoals eerder gezegd een PCI insteekkaart en is het aansluitpunt voor de radarvideo bronnen. Hiervoor heeft de RVG een analoge interface. Meer informatie over deze interface en bijbehorende distributie architectuur is te vinden in Hoofdstuk 5.2. In de huidige configuratie is het dus noodzakelijk dat elke console voorzien is van een RVG insteekkaart.

Gezien er meerdere radarvideo bronnen tegelijk aangesloten kunnen worden, is het belangrijk ook meerdere visualisaties hiervan tegelijk te kunnen verwerken. De RVG is in staat om gelijktijdig vier visualisaties van de radarvideo te verwerken. Deze hoeven onderling niet in bron te verschillen. Er kunnen ook meerdere visualisaties van dezelfde bron weergegeven worden met een andere zoomfactor of een andere locatie als middelpunt door *off-centering*.

In Figuur 25 is de RVG schematisch weergegeven. Hierop is te zien dat de daadwerkelijke processing functionaliteit van de RVG gebeurt in een *FPGA*. De exacte functies en beperkingen hiervan komen in de volgende paragrafen aan bod. De volgorde waarin deze genoemd worden is niet noodzakelijkerwijs de volgorde waarin ze daadwerkelijk in de praktijk uitgevoerd worden.



Figuur 25 RVG Hardware [2]

5.1.1.1 Afstand Normalisatie

Elke radar heeft een bepaalde maximum afstand waarover radarvideo gegenereerd wordt. Het is echter vaak het geval dat een operator niet naar het complete radarbeeld kijkt, maar slechts

naar een klein gedeelte daarvan. Zo kan deze er bijvoorbeeld voor kiezen om van een radar die normaal gesproken 100 km ver kan kijken, slechts geïnteresseerd te zijn in wat er binnen 50 km rondom het schip gebeurt. Wanneer deze instelling gemaakt wordt kan het beeld dat van deze informatie gemaakt wordt een dubbel zo hoge resolutie bevatten of kunnen berekeningen beperkt worden doordat slechts de helft van de samples in de radarvideo verwerkt hoeft te worden.

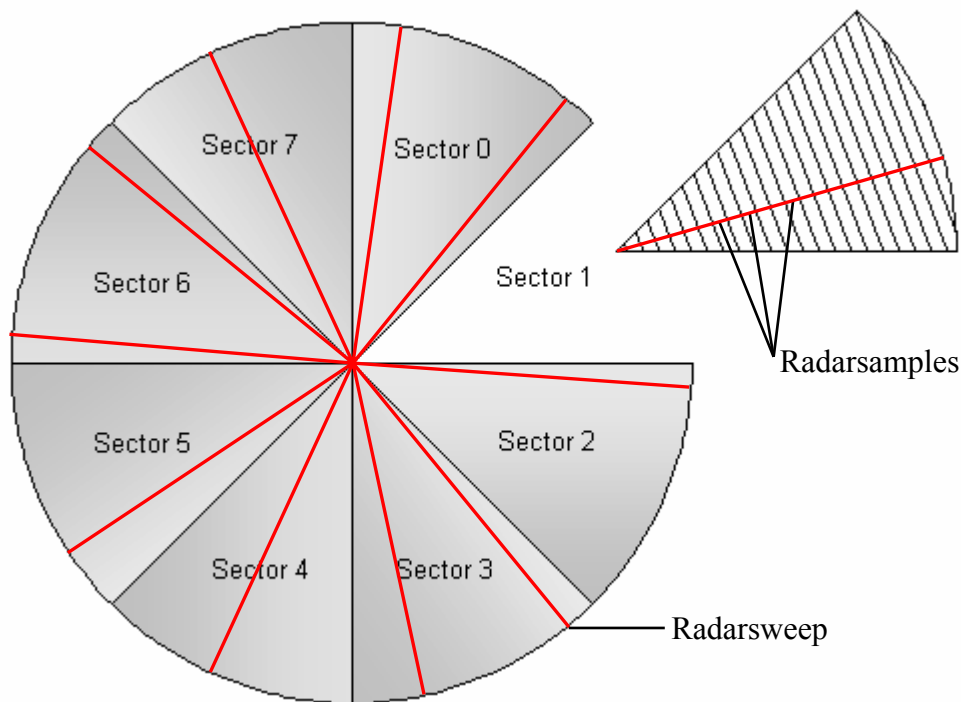
Andersom is het ook mogelijk dat bijvoorbeeld de radarvideo van een 50 km radar samen met de radarvideo van een 400 km radar afgebeeld moet worden. Hiervoor moeten de datasamples van de eerste radar dus ook tot 400 km genormaliseerd worden voor een correcte afstandverhouding tussen de beide radars. In dit geval moeten datasamples samengevoegd worden om tot de gevraagde resolutie te komen. Voor dit samenvoegen van datasamples kan gekozen worden tussen een piek- of een middelingalgoritme. De totale afstand waarop genormaliseerd kan worden is in de RVG beperkt van 0.25 tot 1000 km.

Ook off-centering speelt een rol in deze afstand normalisatie. Hierbij kan gekozen worden om het middelpunt van het scherm af te laten wijken van het middelpunt van de PPI. Op deze manier kan dus ingezoomd worden op een gebied op enige afstand van het schip om daar bijvoorbeeld een mogelijk doel goed in de gaten te houden. Dit off-centering kan gedaan worden in elke richting, met een maximum van 1000 km minus de gekozen totale afstand.

Verder is gegeven dat radarvideo van één sector uit maximaal 2048 samples in de afstand richting bestaat.

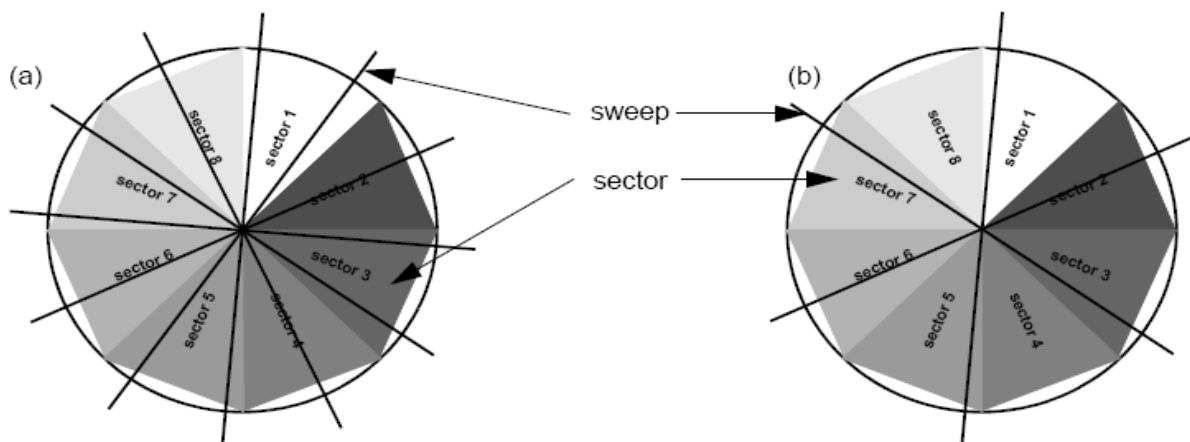
5.1.1.2 Sweep normalisatie

Doordat radarsweeps onder andere door externe invloeden zoals wind niet elke rotatie in exact dezelfde richting staan, en er voor processing en afbeelding toch behoefte is aan een constante verdeling van de radarinformatie, worden hiervoor radarsectoren gehanteerd. Deze radarsectoren worden van inhoud voorzien door de radarsweeps die hierbinnen vallen. De 360 graden van een volledige omwenteling worden opgedeeld in sectoren van gelijke grootte. In de RVG kan gekozen worden voor een sector resolutie van 256, 512, 2048 en 4096 sectoren per scan, waarbij elke sector dus een hoek beslaat van respectievelijk 256/360, 512/360, 2048/360 of 4096/360 graden. In Figuur 26 is weergegeven hoe radarsectoren, -sweeps en -samples in verhouding staan tot elkaar binnen een complete radarscan.



Figuur 26 Radarsectoren, –sweeps en –samples in een complete radarscan

Zoals ook te zien is in Figuur 27 kan het voor komen dat bepaalde sectoren twee of meer sweeps bevatten (a) en bepaalde sectoren zelfs helemaal geen sweeps (b). In het eerste geval kan gekozen worden tussen een piek en middeling algoritme voor het samen voegen van de datasamples in de sweeps. Door de RVG wordt integratie van een maximum van 8 sweeps binnen een sector ondersteund. Wanneer de hoeveelheid sweeps binnen een sector dit aantal overschrijdt, worden verdere sweeps binnen die sector genegeerd. In het tweede geval wordt de radarvideo van de voorgaande sector als output gegeven. Op deze manier worden maximaal 8 sectoren gekopieerd. Wanneer deze situatie zich meer dan 8 maal heeft voorgedaan, worden in plaats daarvan null-sectoren gegenereerd. Wanneer er meer dan 64 null-sectoren gegenereerd zijn, wordt een radar video error gerapporteerd.



Figuur 27 Radarsweep integratie (a) en radarsweep duplicatie (b)

5.1.1.3 Video Combinatie

Elk video verwerkingskanaal waarop een radarvideo bron aangesloten kan worden, is in staat om maximaal drie videostreams samen te voegen en te verwerken, bijvoorbeeld LIN, MTI en SEC video. De binnengekomen videostreams worden vergeleken met vastgestelde drempel waarden om te bepalen of ze voor weergave in aanmerking komen. Wanneer geen of meerdere videostreams hiervoor in aanmerking komen, wordt op basis van een set prioriteitregels een keuze gemaakt.

Naast deze keuze worden van de 8 bits binnengekomen radarvideo de 6 meest significante bits geplaatst op de 6 minst significante bits van de uitvoer. De 2 meest significante bits worden gezet op een code, die de bron van de radarvideo representeert.

5.1.1.4 Afterglow Proces

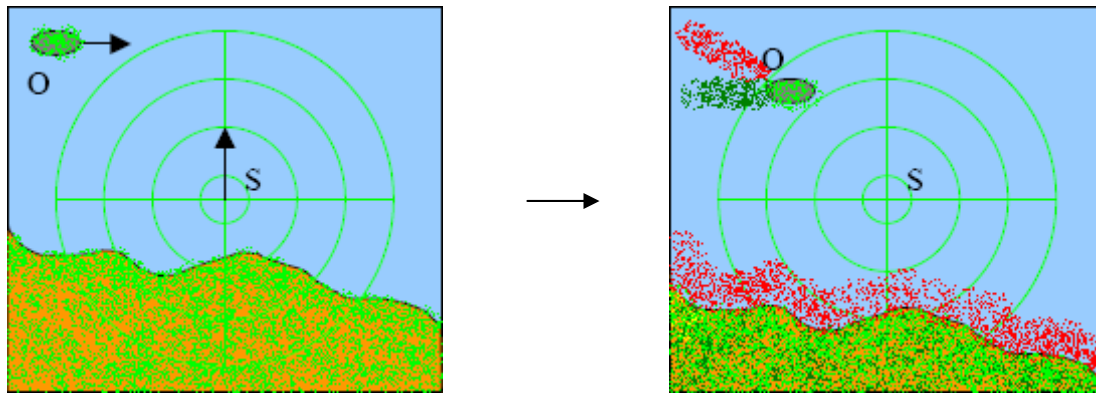
In dit proces wordt de radarvideo van eerdere radarscans gebruikt en samengevoegd met de nieuwe radarvideo om zo de geschiedenis van voorgaande scans bij te houden. De resulterende waarde van de datasample wordt bepaald door een algoritme dat de amplitude van de betreffende sample in het huidige radarvideo beeld met de corresponderende waarde in de geschiedenis vergelijkt.

De operator kan voor de duur waarover de geschiedenis bijgehouden wordt vrij kiezen tussen een waarde van 2 tot 32 radar omwentelingen of oneindig. De intensiteit van de samples in de geschiedenis vermindert lineair met een factor *maximale intensiteit / hoeveelheid scans* per scan. Uiteraard is het ook mogelijk het bijhouden van de geschiedenis geheel uit te schakelen. Verder is het mogelijk te kiezen voor amplitude afhankelijke en onafhankelijke afterglow.

In het eerste geval wordt de intensiteit van de sample in de eerste geschiedenisframe gelijk gesteld aan de intensiteit die de sample oorspronkelijk bevatte. Wanneer dit bijvoorbeeld de helft van de maximale amplitude bedraagt, blijft de afterglow ook de helft van de gekozen omwentelingen zichtbaar. Dit valt te wijten aan de eerder vermelde lineaire afname van de intensiteit.

De tweede methode kent altijd de maximale intensiteit toe aan de eerste geschiedenisframe. Dit gebeurt ongeacht de oorspronkelijke intensiteit van de radarsample. Hierdoor blijven alle samples de maximale hoeveelheid gekozen radaromwentelingen zichtbaar. Hierbij wordt echter wel gekeken of de oorspronkelijke intensiteit hoger is dan de ingestelde threshold. Wanneer dit niet het geval is, wordt de sample niet toegevoegd aan de geschiedenisframe. Dit wordt gedaan om te voorkomen dat het beeld dicht gesmeerd wordt door ook ruis met maximale intensiteit in de geschiedenisframe op te slaan.

De afterglow wordt bijgehouden in de B-Scope van de radarvideo, die als input aangeboden wordt aan de software om de PPI te genereren. Dit heeft echter ongewenste bijeffecten wanneer de PPI in *Relative Motion* wordt weergegeven. Wanneer het beeld ver is ingezoomd, het schip snel vaart of een erg lange/oneindige afterglow gekozen is, beweegt de afterglow duidelijk met het schip mee. Hierdoor ontstaat zelfs een smeereffect bij stilstaande objecten zoals eilanden en is de locatie van eerdere scans niet meer juist gepositioneerd. Dit is zichtbaar in Figuur 28. Het foutieve effect zoals het op dit moment bewerkstelligd wordt is in rood weergegeven, de correcte afterglow is in donker groen afgebeeld ter referentie. In Hoofdstuk 7.2.1.4 worden enkele mogelijkheden beschreven, waarmee dit probleem te behandelen is.



Figuur 28 Foutief afterglow effect weergegeven in rood

5.1.1.5 Pixel data conversie

In deze conversiestap wordt de 8 bits sampledata met gecombineerd video, genormaliseerde sweeps en eventueel afterglow, naar een *RGBA* texture omgezet. Hierbij wordt gebruik gemaakt van een op runtime te kiezen kleurentabel. Door gebruik te maken van verschillende kleurentabellen voor verschillende videostreams, kan er visueel onderscheid gemaakt worden tussen radarvideo afkomstig van deze verschillende videostreams.

De polaire radarvideo wordt uiteindelijk via de PCI bus in een blok geheugen weggeschreven waarin het radarvideo B-Scope georiënteerd is opgeslagen. Hiervoor worden per radarsample 4 waarden gebruikt. Deze geven de intensiteit van rood, groen, blauw en de mate van transparantie weer, wanneer de radarvideo op het scherm afgebeeld moet worden. Zoals eerder vermeld, ondersteunt de RVG maximaal 4 gelijktijdige visualisaties. De maximale texture configuratie voor deze 4 visualisaties in de meest moderne toepassing van de RVG is één maal 1024 x 2048, één maal 1024 x 1024, en twee maal 512 x 512. Deze waarden worden echter niet door de RVG zelf beperkt, maar door de gebruikte grafische kaart in de console.

5.1.2 RSC-TB Software

Nadat de RVG de radarvideo zover heeft verwerkt tot een B-Scope georiënteerde weergave van het radarvideo, is het de taak van de RSC-TB software om dit verder te verwerken tot een afbeelding op het scherm. Het geheugenblok kan zoals gezegd regelrecht in een *RGBA* texture ingelezen worden. Vervolgens wordt door middel van standaard texture mapping op een PPI-vormige geometrie, zoals uitgewerkt in Hoofdstuk 3.1, de weergave op het scherm gerealiseerd. De maximale resolutie van deze afbeelding is gesteld op 1600 x 1200 waarbij een verversingssnelheid tussen 1 Hz en 30 Hz te kiezen is door de operator. De applicatie waarin de radarvideo wordt afgebeeld hanteert een verversingssnelheid van 20 Hz.

De software biedt daarnaast de mogelijkheid de inhoud van de radarvideo afbeelding te wissen, wat automatisch gebeurt wanneer:

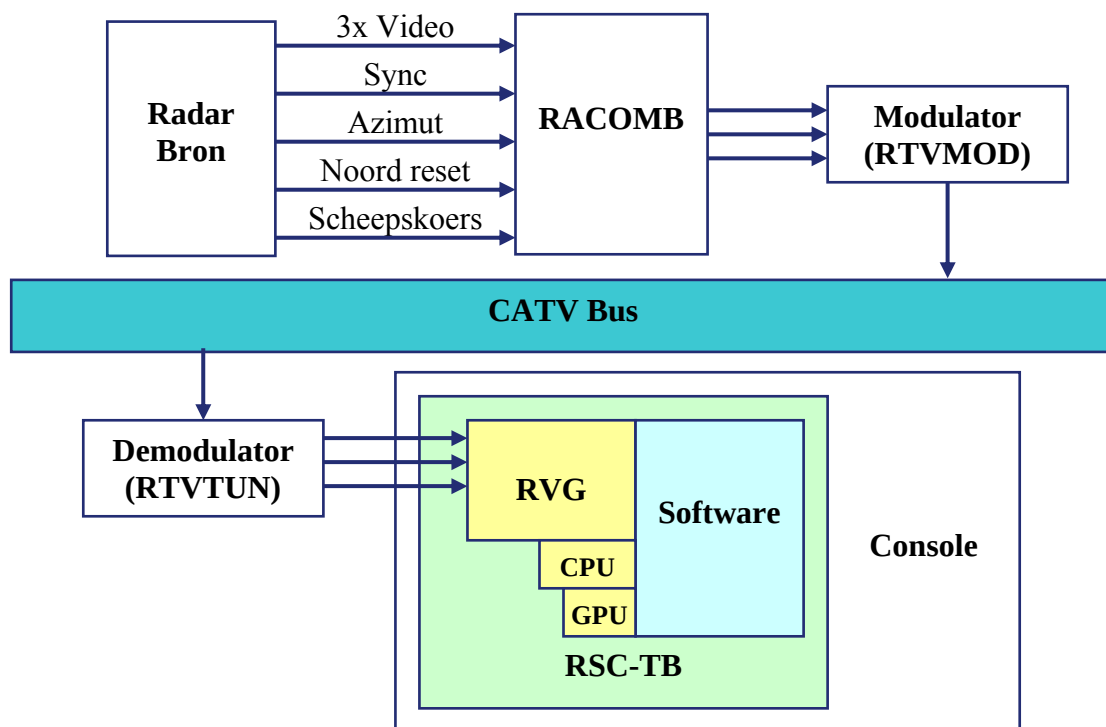
- ☐ De afstand schaal veranderd wordt.
- ☐ Het venster met de afbeelding van groter of kleiner gemaakt wordt.
- ☐ Een andere radar geselecteerd wordt.

Verder wordt de mogelijkheid geboden om de huidige afbeelding te bevriezen. Nieuwe radarvideo wordt dan niet meer op het scherm afgebeeld. Hiermee wordt voorkomen dat het oude gegevens overschrijft. Ook is een optie voor update van één enkele radarscan, na één complete omwenteling van de radar wordt de afbeelding automatisch bevroren.

5.2 Analoge radarvideo distributie

Analoge radarvideo distributie is op dit moment nog de meest gebruikte distributie architectuur doordat alle behalve de meest recente radars hun video in analoge vorm generen. Deze wordt in de meeste gevallen over een CATV bus door het schip verstuurd naar de consoles waar het op een scherm geprojecteerd kan worden. Alle onderdelen die op dit moment een rol spelen in de distributie van radarvideo zijn schematisch weergegeven in Figuur 29. Hierbij zijn de hardware componenten waarvan gebruik wordt gemaakt in de RSC-TB (weergegeven met een gele kleur) semi-proportioneel naar hun gebruik afgebeeld.

De radarbron is hier uiteraard de radar die de radarvideo genereert, bijvoorbeeld een LW08. Naast radarvideo worden ook andere gegevens geproduceerd, zoals een synchronisatie puls waarmee het 0-km punt ten opzichte van de radar in de radarvideo wordt gemarkeerd. Verder wordt de hoek ten opzichte van het noorden waarop de radar zich op het moment van de radarsweep zich bevindt in een azimuth puls afgegeven. De RSC-TB kan deze pulsen met een resolutie van 360/4096 graden verwerken. De Noord reset puls wordt afgegeven wanneer de radar precies in lijn staat met het geografische noorden. Hiermee wordt tevens de azimuth puls op 0 gezet. Tenslotte wordt er een scheepskoers puls afgegeven, wanneer de radar precies in lijn staat met de boeg van het schip. Al deze informatie wordt in de RACOMB samengevoegd tot zogenaamd *composiet radarvideo*. De RACOMB kan dit proces voor drie radarbronnen tegelijk doen en daarmee ook drie radarvideostreams tegelijk genereren



Figuur 29 Analoge radarvideo distributie architectuur [6]

De bandbreedte van een radarvideo signaal kan oplopen tot een maximum van 20 MHz, de exacte grootte hiervan hangt af van de rotatiesnelheid, het aantal sweeps per rotatie en de resolutie van de radar. Door een modulator wordt dit signaal gereduceerd tot 5 MHz om te voldoen aan de TV video norm en wordt het op een CATV bus gedistribueerd. Deze reductie in bandbreedte lijkt erg groot, uit de praktijk is gebleken dat het effect hiervan erg mee te valt.

Aan de console kant ontvangt een demodulator genaamd de RTVTUN de videostream en wordt deze beschikbaar gesteld aan de RVG. Hier zijn weer de drie radarvideo streams te ontvangen, die door de RACOMB gegenereerd zijn. Dit radarvideo wordt volgens het eerder beschreven proces door de RSC-TB verwerkt tot een uiteindelijke weergave op de console.

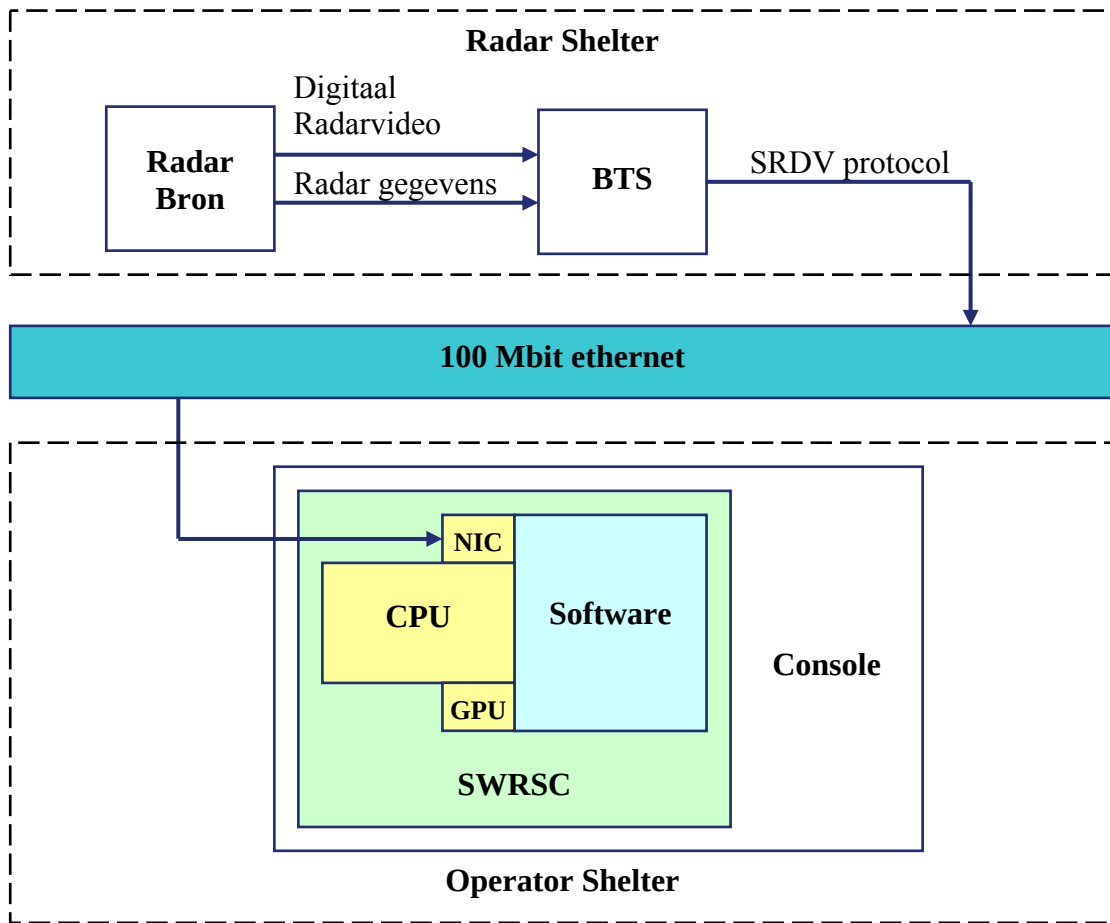
5.3 Digitale radarvideo distributie

Dit onderzoek beoogt een digitale radarvideo distributie architectuur. Daarom wordt hier nog kort de tot op dit moment enige geproduceerde digitale architectuur, die van de Flycatcher Mk2, besproken. Dit systeem bestaat, zoals te zien is in Figuur 30, uit twee shelters. In één van deze shelters is de radar gemonteerd. De andere shelter biedt ruimte aan twee consoles en operators die deze moeten bedienen. De shelters zijn met elkaar verbonden met een 100 Mbit ethernet connectie. De schematische voorstelling van deze distributie architectuur is gegeven in Figuur 31.



Figuur 30 Flycatcher Mk2 opstelling

De Flycatcher gebruikt altijd hetzelfde type radar. Dit is degene die op de radarshelter is gemonteerd. Hierdoor kan de interface tussen radar en radarscan conversie systeem geheel toegespitst worden op deze specifieke toepassing. De radarbron geeft genormaliseerd, digitaal video af. Het is dus niet meer nodig hierop sweep normalisatie toe te passen zoals vermeld in Hoofdstuk 5.1.1.2.



Figuur 31 Digitale radarvideo distributie architectuur van de Flycatcher Mk2

De radarvideo wordt vervolgens, samen met verdere data van de radar zoals gegevens over de huidige azimuth, doorgestuurd wordt naar een *BTS*. Deze zorgt ervoor dat alle radarvideo en verdere informatie verstuurd wordt via ethernet. Hiervoor wordt het Surveillance Radar Display Video (*SRDV*) protocol gehanteerd. In dit protocol zijn twee bericht types gedefinieerd, een *RSP* en een *RSD* bericht.

Het *RSP* bericht wordt eenmaal per radarscan verstuurd. Dit gebeurt na de verwerken van laatste sector van de vorige scan en vóór het verwerken van de eerste sector van de nieuwe scan. Hierin worden alle karakteristieken van het radarvideo gespecificeerd. Het bericht is 28 bytes groot. De exacte data lay-out van dit bericht is te zien in Figuur 32.

Parameter	Abbr.	bit 7 MSB	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0 LSB	Byte count
Message ID (0 .. 255)	MID	01 Hex								1
Source ID (0 .. 255)	SID	binary data								2
Reserved	-	0								3
Reserved	-	0								4
Radar Scan Time in ms	RST	binary data (* 2 ²⁴)								5
		binary data (* 2 ¹⁶)								6
		binary data (* 2 ⁸)								7
		binary data (* 2 ⁰)								8
Scan Sequence Number	SSN	binary data (* 2 ²⁴)								9
		binary data (* 2 ¹⁶)								10
		binary data (* 2 ⁸)								11
		binary data (* 2 ⁰)								12
Sample range in mm (0 ..16,777,215)	SRG	binary data (* 2 ²⁴)								13
		binary data (* 2 ¹⁶)								14
		binary data (* 2 ⁸)								15
		binary data (* 2 ⁰)								16
Azimuth resolution in 2 ^{AZR} sectors per 360 ⁰ (AZR = 0 .. 14, resolution = 1 .. 16k)	AZR	n in binary data								17
Channels per sample indi- cator (0 .. 13)	CPS	binary data								18
Amplitude resolution (2 .. 256 levels/sample) (for channel 0)	AMR	resolution-1 binary data								19
Compression Algorithm: 0 = none 1.. 255 = reserved	CAL	binary data								20
Maximum range in m (0 ..16,777,215)	MRG									21
		binary data (* 2 ¹⁶)								22
		binary data (* 2 ⁸)								23
		binary data (* 2 ⁰)								24
Azimuth reference: 0 = north 1 = heading	ARF	binary data								25
Reserved	-	0								26
Reserved	-	0								27
Reserved	-	0								28

Figuur 32 Data lay-out van een RSP bericht

Het RSD bericht wordt eenmaal per radarsector verstuurd. Naast de radarsweep zelf bevat het een minimale hoeveelheid aan info over die sweep. De overhead die door dit bericht aan de radarsweep wordt toegevoegd is 20 bytes. De exacte data lay-out van dit bericht is te zien in Figuur 33.

Parameter	Abbr.	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Byte count
Message ID (0 .. 255)	MID	02 Hex								1
Source ID (0 .. 255)	SID	binary data								2
Sector Type 0 = standard 1 = blank sector 2 .. 255 = reserved	STY	binary data								3
Reserved	-	0								4
Scan Sequence Number	SSN	binary data (* 2 ²⁴)								5
		binary data (* 2 ¹⁶)								6
		binary data (* 2 ⁸)								7
		binary data (* 2 ⁰)								8
Sector number (0 ..16k-1)	SEN	binary data (* 2 ²⁴)								9
		binary data (* 2 ¹⁶)								10
		binary data (* 2 ⁸)								11
		binary data (* 2 ⁰)								12
Reserved	-	0								13
Reserved	-	0								14
Sample Count (0 .. 4k), always a multiple of 4	SAC	binary data (* 2 ⁸)								15
		binary data (* 2 ⁰)								16
Sample 1	SD	binary data (sample 1)								17
Sample n		binary data (sample n)								17+n
Sample n+1		binary data (sample n+1)								17+n+1
Sample SAC		binary data (sample SAC)								17+SAC-1
End of Sector	EOS	hex'55'								18+SAC-1
		hex'55'								19+SAC-1
		hex'55'								20+SAC-1
		hex'55'								21+SAC-1

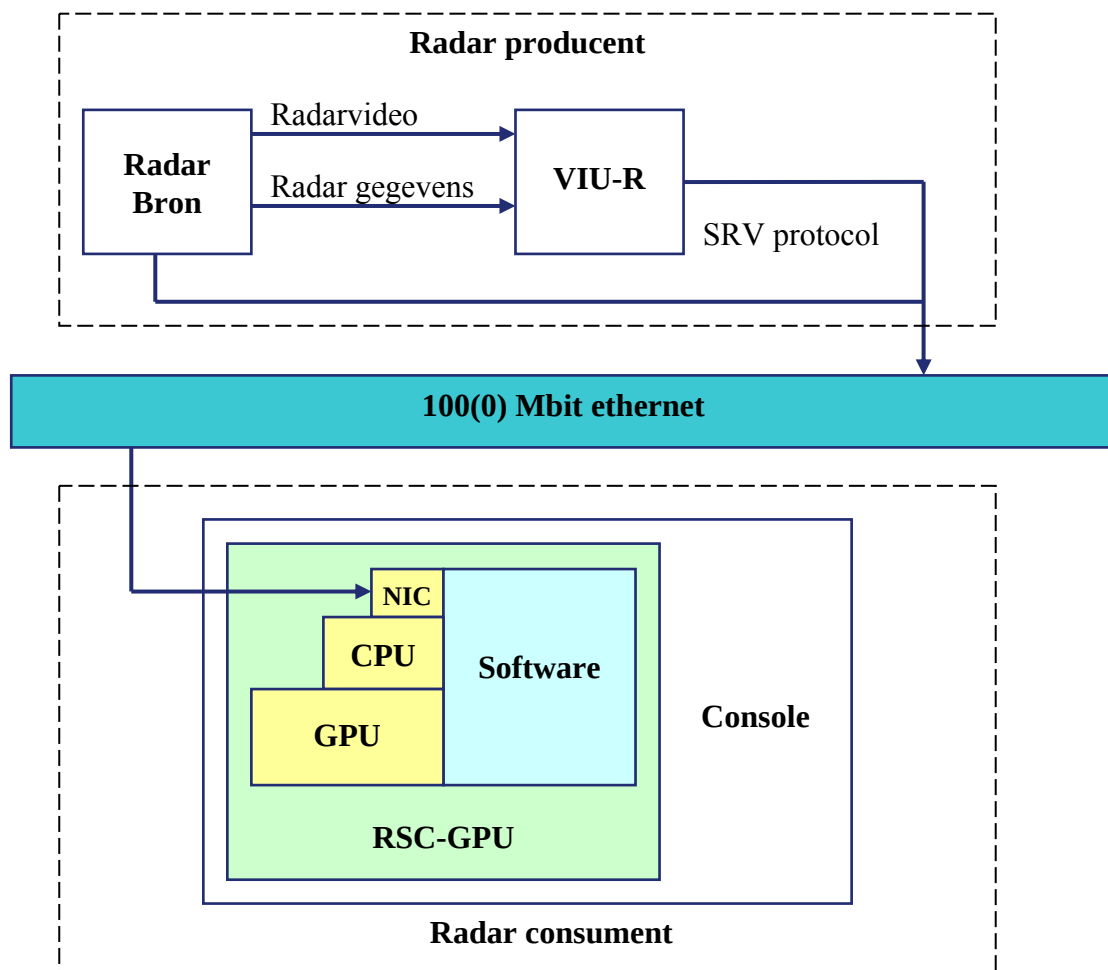
Figuur 33 Data lay-out van een RSD bericht

Beide berichten worden via UDP over ethernet gebroadcast en komen in de operator shelter via de NIC binnen in een console. Het ontvangen en verwerken van de radarvideo gebeurt in de Flycatcher in de Software RSC (SWRSC). Deze maakt net als de RSC-TB uitsluitend voor de texture mapping gebruik van de GPU. Alle overige werkzaamheden voor de radarscan conversie worden door de software op de CPU uitgevoerd.

6 Opzet radarvideo distributie architectuur

Het ligt niet in de doelstelling van dit onderzoek om alle aspecten van een digitale radarvideo distributie tot in detail uit te zoeken. Het is echter wel belangrijk om hier toch een globaal idee over te vormen. Omdat een overzicht verkregen moet worden van alle operaties die gedaan moeten worden en de verdeling hiervan over de verschillende onderdelen in de architectuur.

De architectuur die hiervoor bedacht is, is schematisch weergegeven in Figuur 34. Het valt direct op dat deze sterk lijkt op de distributie architectuur van de Flycatcher Mk2. Een belangrijk verschil is echter dat de architectuur geschikt moet zijn voor meerdere typen radars. Dit hoeven zelfs geen digitale radars te zijn.



Figuur 34 Opzet nieuwe digitale radarvideo distributie architectuur

Het is voor de rest van het proces wel belangrijk dat de radardata eenduidig aangeleverd wordt. Deze vorm van de output zal vergelijkbaar zijn met een meer generiek toepasbare variant van het SRDV protocol. In het verdere verslag wordt hiernaar met het Streaming Radar Video of *SRV protocol* verwezen. Digitale radars zijn in staat zelf dit protocol te genereren. Als het een analoge radar betreft, zal de VIU-R hier zorg voor dragen. Deze bestaat op dit moment nog niet maar is al wel in ontwikkeling.

Naast de omzetting naar het SRV protocol moeten de gegenereerde berichten nog verstuurd worden naar alle consumenten. Hiervoor zal geen broadcast van UDP pakketten gebruikt worden, maar multi-cast. Hiermee kan een radarconsument zich aanmelden voor één radarbron, zonder dat hij ook automatisch de data van eventuele andere radarbronnen te verwerken krijgt.

Een ander belangrijk verschil met de architectuur zit in het hardwaregebruik binnen de RSC, in Figuur 34 gesymboliseerd door de verhoudingen tussen de gele vlakken. Zo maakt de RSC-GPU veel intensiever gebruik van de GPU dan de SWRSC. Hierdoor kan de CPU worden ontlast. Het CPU gebruik is echter toch iets hoger dan in het geval van de RSC-TB. Dit komt doordat de SRV pakketten, die de NIC ontvangen heeft, door de CPU moeten worden klaargezet voor de verwerking op de GPU. Hier staat echter tegenover dat niet elke radarconsument over speciale hardware zoals de RVG hoeft te beschikken. Daarbij is de hierdoor gegenereerde belasting erg laag. Deze neemt hooguit enkele procenten in beslag.

Voor de architectuur is gedurende het onderzoek een worstcase scenario aangehouden van een radar met 4096 sectoren (AZR), 2048 samples per sector (SAC) en een omwentelingstijd van 2 seconden (RST). Uitgaande van het SRDV protocol en verzending als UDP datagrammen kan hiermee de totale netwerkbelasting als volgt berekend worden:

$$e = 26, ip = 20, udp = 8$$

$$RSD = 20, RSP = 28$$

$$AZR = 4096, SAC = 2048, RST = 2$$

$$bandwidth = \frac{(e + ip + udp + RSD + SAC) \cdot AZR + (e + ip + udp + RSP)}{RST} = 4.14 MB/s$$

In deze formule staan de bovenste drie elementen, e , ip en udp voor het aantal bytes overhead dat respectievelijk een ethernet, IP, en UDP pakket met zich mee brengen. RSD en RSP geven de hoeveelheid bytes aan overhead van de protocol berichten weer. De hier berekende 4.14 MB/s komt hiermee overeen met de hoeveelheid data die de CPU in worst-case te verwerken krijgt van één radarvideo stream. Met een maximum van drie radarvideo streams, dezelfde hoeveelheid als het maximum van de RVG, komt het totaal op 12.43 MB/s.

6.1 Rolverdeling consument – producent

In de vorige paragraaf is uitgelegd dat de producerende kant wordt opgemaakt door de radarbron zelf en, wanneer het een analoge radar betreft, de VIU-R. De genoemde operaties die aan de producerende kant uitgevoerd moeten worden zijn:

- ❑ Digitaliseren van analoog radarvideo
- ❑ Genereren van SRV berichten op basis van radarvideo en verdere radareigenschappen
- ❑ Versturen van de SRV berichten door multi-cast

Om te bepalen welke operaties nog meer aan de kant van de producent uitgevoerd kunnen worden, moet gekeken worden naar mogelijke beperkingen die deze opleggen aan de consumenten van de radarvideo. Uitgaande van de situatie zoals bij de RSC-TB kan gekeken worden naar alle functies die door de RVG uitgevoerd worden. Deze zijn uitgelegd in Hoofdstuk 5.1.1. Per functie kan vervolgens bepaald worden of het de keuzevrijheid van de operator significant beperkt, wanneer deze al eerder in de keten zijn uitgevoerd.

Voor de afstand normalisatie, beschreven in Hoofdstuk 5.1.1.1, is kennis vereist van de op dat moment ingestelde kijkafstand. Gezien verschillende operators doorgaans verschillende kijkafstanden hanteren, is het dus niet mogelijk dit al eerder door de producent te laten doen.

Sweep normalisatie kan echter al wel door de producent uitgevoerd worden. Hiervoor wordt immers altijd hetzelfde algoritme gehanteerd, ongeacht de instellingen van de individuele operator. Dit proces leent zich dan ook uitstekend om uitgevoerd te worden door de VIU-R. Dit is ook in overeenstemming met de eigenschappen van het SRDV en daarmee ook hoogstwaarschijnlijk SRV. De radarvideo die hiermee verstuurd wordt is ook sweep genormaliseerd.

Alle overige taken zijn, net als afstand normalisatie, te sterk afhankelijk van de instellingen van de operator. Om de diversiteit in inzetbaarheid van de operators te behouden, moeten deze taken per operator instelbaar blijven. Deze vallen dus allemaal onder de taken van de radarconsument.

6.2 Gebruik van GPU programmeren

Vervolgens kan de vraag gesteld worden of het zinvol is GPU programmeren toe te passen voor het uitvoeren van de taken van de producent en consument. Op deze vraag kan voor de consument zonder twijfel een bevestigend antwoord gegeven worden. Een deel van deze werkzaamheden, het doen van de werkelijke radarscan conversie door texture mapping, wordt ook in de RSC-TB en SWRSC al door de GPU uitgevoerd. Het gebruik van de in dit verslag beschreven filteralgoritmen, en verdere mogelijkheden beschreven in Hoofdstuk 7, geven hier alleen nog maar meer redenen toe.

Voor de producent van radarvideo zal het gebruik van GPU programmeren echter naar verwachting minder zinvol zijn. De GPU werkt uitsluitend met digitale data, dus digitalisering van analoog video moet al eerder uitgevoerd zijn. Daarbij is het genereren van datapakketten niet een operatie waarin de GPU excelleert. Ook moet de data voor elke verwerking naar de GPU worden gedownload en daarna weer worden teruggelezen. Vooral dit laatste kan vervelende gevolgen hebben voor de totale doorvoersnelheid van het systeem [1].

7 Radarscan conversie prototype

Aanvankelijk is voor de uitvoering van dit onderzoek een simpel framework gemaakt. Deze diende uitsluitend als ontwikkel-, test- en demonstratieomgeving voor de eerste doelstelling, het verbeteren van de weergave door het samplingprobleem op te lossen. Nadat de nieuwe doelstellingen geformuleerd waren, voldeed het initiële prototype niet meer. In dit hoofdstuk worden de hiervoor gestelde eisen behandeld. Daarnaast wordt het ontwerp en de belangrijkste functies van de verschillende componenten behandeld.

7.1 De eisen

Voor de eisen van de geïmplementeerde RSC is voornamelijk gekeken naar de mogelijkheden van de RSC-TB. Het doel is immers een RSC te ontwikkelen die gelijkwaardige of betere mogelijkheden biedt. Hierbij zijn de volgende eisen opgesteld.

Uitgaande van een rechthoekige texture met daarin polair georiënteerde, sweep genormaliseerde radarvideo moet een PPI beeld gegenereerd kunnen worden. Het programma moet om kunnen gaan met textures tot 4096 x 2048. Hierbij moet gebruik worden gemaakt van het ontwikkelde filteralgoritme. De distributie van de radarvideo van producent naar consument moet via ethernet verlopen.

Het centrum van deze PPI afbeelding moet op het scherm verplaatsbaar zijn om zo off-centering te kunnen demonstreren. Daarbij moet er in- en uitgezoomd kunnen worden op de radarvideo.

Om inzicht te krijgen in het verband tussen de locatie van het schip, de radarreflecties en de wereld moeten de reflecties geprojecteerd worden op een achtergrond waarop een (zee)kaart is afgebeeld. Voor lokalisatie van het schip wordt op deze kaart een kruis geprojecteerd op de plaats waar het schip zich bevindt met daar omheen een aantal afstandsringen.

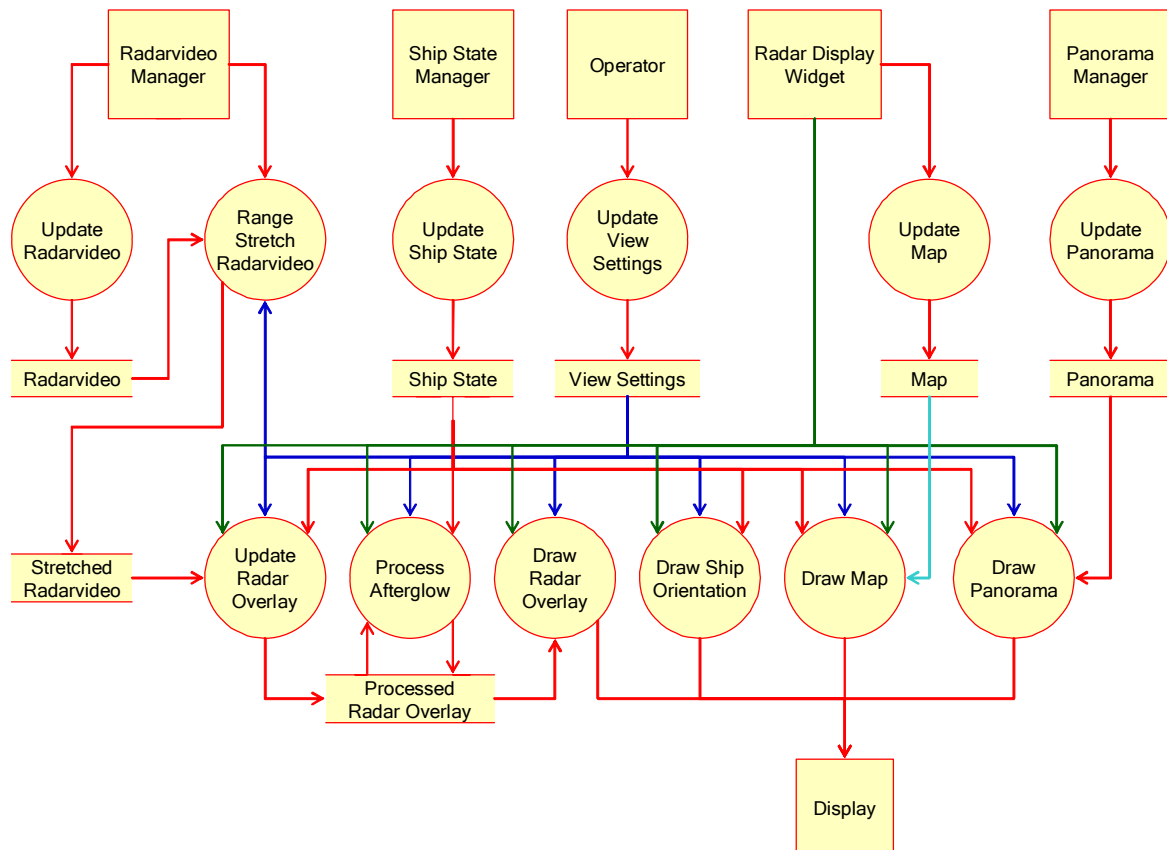
De applicatie moet in staat zijn om de radarvideo op verschillende manieren weer te geven. Zo moet het mogelijk zijn om kleuring op basis van een kleurentabel uit te voeren.

De geschiedenis van voorgaande tracks moet bijgehouden kunnen worden. Deze geschiedenis moet een instelbare hoeveelheid radarscans zichtbaar blijven en wordt met een continu afnemende intensiteit afgebeeld. Wanneer er zowel nieuwe radarvideo als geschiedenis data beschikbaar is voor één pixel, wordt degene met de hoogste intensiteit gekozen voor afbeelding.

Het gehele proces moet zo snel mogelijk uitgevoerd kunnen worden, liefst met snelheden boven 30 Hz. Tevens dient de CPU belasting geminimaliseerd te worden. Dit mag zeker niet boven de 80% van de systeemcapaciteit uitkomen.

7.2 De componenten

Om de hiervoor beschreven eisen te implementeren en de ruimte te bieden voor verdere ontwikkelingen, is een modulaair ontwerp nagestreefd. In Figuur 35 is een dataflow diagram van de meest relevante delen van het systeem te zien. Hierbij zijn de entiteiten bovenaan, met uitzondering van de *Operator*, de belangrijkste componenten. Deze zijn elk verantwoordelijk voor het genereren van een deel van alle informatie die nodig is voor radarscan conversie. Deze componenten worden in de volgende sub-hoofdstukken verder besproken. Hierbij wordt de nadruk gelegd op het ontwerp, de mogelijkheden en beperkingen van de componenten.

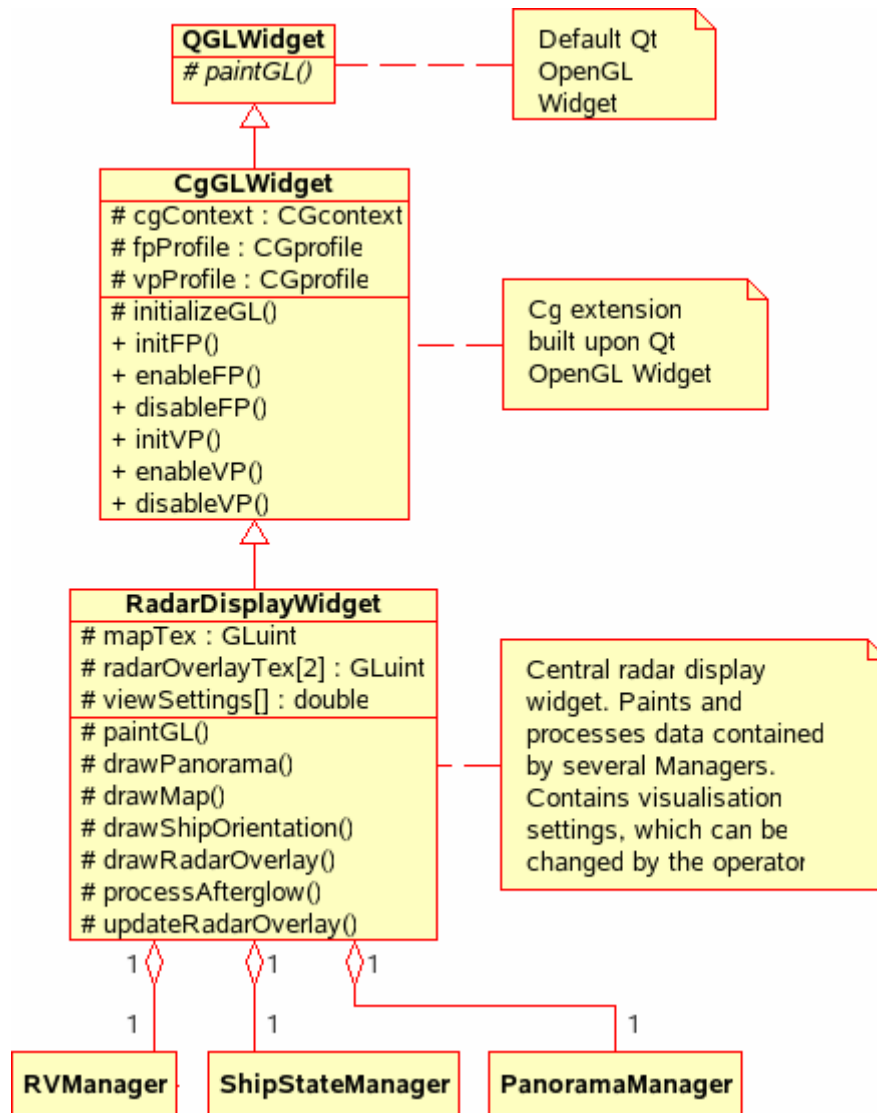


Figuur 35 DFD van het prototype

Het prototype is geïmplementeerd met behulp van de C++ Toolkit Qt. Deze biedt een breed scala aan componenten voor onder andere GUI elementen en maakt communicatie tussen objecten zeer flexibel door het signals – slots mechanisme. De toolkit is bovendien platform onafhankelijk en biedt daardoor de mogelijkheid, om eenvoudig een gemaakte applicatie te porten naar bijvoorbeeld Windows wanneer dit gewenst is.

7.2.1 Centrale radardisplay

Uit Figuur 35 valt op te maken dat de *RadarDisplayWidget* een centrale rol speelt in de geboden functionaliteit van het systeem. In Figuur 36 is het klassendiagram hiervan te zien. Voor gebruik van OpenGL in een Qt omgeving is de *QGLWidget* beschikbaar. Deze is verder uitgebreid met ondersteunende functies voor het eenvoudig initialiseren en gebruik van Cg programma's in de OpenGL context. Deze *CgGLWidget* ligt ten grondslag aan de *RadarDisplayWidget*, waar alle data verwerkt en samengevoegd wordt met behulp van OpenGL en Cg programma's.



Figuur 36 Klassendiagram *RadarDisplayWidget* en aangrenzende componenten

Het belangrijkste deel van de te verwerken data wordt gegenereerd of ontvangen en opgeslagen in de betreffende managers. Enkele van deze managers hebben zelf een OpenGL context nodig, bijvoorbeeld om de informatie in een texture op te slaan. De *RadarDisplayWidget* bij elke update deze managers hiervoor in de gelegenheid stellen door een functie aanroep te doen terwijl de context actief is. Deze constructie wordt gebruikt omdat het over het algemeen sterk afgeraden wordt om een OpenGL context vanuit meerdere threads te benaderen. Zo is op eenvoudige wijze de toegang tot de OpenGL context gesynchroniseerd. De managers zelf en, waar van toepassing, hun gebruik van OpenGL komen in de volgende sub-hoofdstukken aan bod.

De *RadarDisplayWidget* zelf bevat verscheidene visualisatie instellingen, zoals translatie coördinaten voor off-centering, zoomfactoren en rotaties. Verder bevat het nog een map in de vorm van een texture. Deze kan op de achtergrond weergegeven worden om de radarvideo en scheepslocatie in een geografische context te kunnen plaatsen.

7.2.1.1 Radarscan conversie

Voor het uitvoeren van de radarscan conversie zelf bevat het nog een tweetal textures ter grootte van het de display. Bij het uitvoeren van de radarscan conversie met de algoritmen beschreven in Hoofdstuk 4 wordt het resultaat direct in een dergelijke texture getekend. Indien zich geen wijzigingen hebben voorgedaan in de visualisatie instellingen, kan hierbij voldaan worden met het tekenen van uitsluitend de nieuw ontvangen radarsectoren. Op deze wijze kan de hoeveelheid bewerkingen geminimaliseerd worden. Deze nieuwe informatie wordt dan met behulp van OpenGL functionaliteit bij de oude inhoud ingemengd. Op deze wijze wordt een afbeelding, die verder aangeduid wordt met *radar overlay*, van PPI-georiënteerd radarvideo gegenereerd dat in overeenstemming is met de visualisatie instellingen. Deze kan zonder verdere transformaties direct op het scherm afgebeeld worden.

7.2.1.2 Afterglow

De reden dat er twee van deze textures nodig zijn, is dat deze ook meteen gebruikt worden voor het bijhouden van afterglow. Hiervoor wordt in de applicatie gebruik gemaakt van textures met drie kleurkanalen. In het rode kanaal bevindt zich de huidige afterglow waarde, in het groene kanaal de ongewijzigde radarvideo. Nieuw radarvideo dat voor afterglow nog onverwerkt is, wordt ook tijdelijk in het blauwe kanaal opgeslagen. Het groene en blauwe kanaal wordt bij het uitvoeren van de radarscan conversie gevuld met de radarvideo. Het blauwe kanaal wordt bij de eerstvolgende verwerking van afterglow leeg gemaakt. Dit extra kanaal dient ervoor om eenvoudige verwerking van radarvideo tot nieuwe afterglow te garanderen. De geïmplementeerde algoritmen, die voor verwerking van afterglow zorgen, zijn zichtbaar in Code fragment 9 tot en met Code fragment 11. De daarin vermelde *fadeFactor* wordt bepaald door de omwentelingssnelheid van de radar, de ingestelde zichtbaarheidsduur en de verversingssnelheid van het beeld op het scherm.

Vanwege het feit dat de GPU voldoende rekenkracht heeft, vindt het verwerken van afterglow altijd plaats. Dit zelfs wanneer de operator op dat moment niet geïnteresseerd is in deze informatie. Wanneer dit wel het geval wordt, kan deze dan ook direct worden weergegeven, inclusief de informatie van enige tijd geleden. Dit in tegenstelling tot het huidige systeem, waarbij de operator eerst enkele scans moet wachten voordat een kleine geschiedenis is opgebouwd, waaruit het gedrag van een object opgemaakt kan worden.

```
float4 historyFaderFP(float2 coords : TEXCOORD0,
                    sampler2D texture,
                    uniform float fadeFactor) : COLOR
{
    //retrieve current radarvideo and afterglow intensity
    float4 value = tex2D(texture, coords);

    //afterglow intensity is equal to greater of previous afterglow
    //and unprocessed radarsample intensity
    //current fadeFactor is subtracted to make afterglow fade over time
    value.r = (value.b > value.r ? value.b : value.r) - fadeFactor;

    //radarsample intensity has been processed for afterglow
    //unmark it as such
    value.b = 0;

    return value;
}
```

Code fragment 9 Cg code voor normale afterglow

```
float4 historyThresholdFaderFP(float2 coords : TEXCOORD0,
                               sampler2D texture,
                               uniform float fadeFactor,
                               uniform float threshold) : COLOR
{
    //retrieve current radarvideo and afterglow intensity
    float4 value = tex2D(texture, coords);

    //afterglow intensity is set to unprocessed radarsample intensity
    //if it is higher than set threshold
    //and higher than the current afterglow value
    //otherwise the current afterglow value is maintained
    //current fadeFactor is subtracted to make afterglow fade over time
    value.r = (value.b > threshold && value.b > value.r ?
               value.b : value.r) - fadeFactor;

    //radarsample intensity has been processed for afterglow
    //unmark it as such
    value.b = 0;

    return value;
}
```

Code fragment 10 Cg code voor afterglow met threshold

```
float4 historyThresholdIIFaderFP(float2 coords : TEXCOORD0,
                                  sampler2D texture,
                                  uniform float fadeFactor,
                                  uniform float threshold) : COLOR
{
    //retrieve current radarvideo and afterglow intensity
    float4 value = tex2D(texture, coords);

    //afterglow intensity is set to max intensity (1.0)
    //if unprocessed radarsample intensity is higher than set threshold
    //and higher than the current afterglow value
    //otherwise the current afterglow value is maintained
    //current fadeFactor is subtracted to make afterglow fade over time
    value.r = (value.b > threshold && value.b > value.r ? 1.0 : value.r)
              - fadeFactor;

    //radarsample intensity has been processed for afterglow
    //unmark it as such
    value.b = 0;

    return value;
}
```

Code fragment 11 Cg code voor intensiteit onafhankelijke afterglow met threshold

Belangrijk is hierbij op te merken dat de afterglow pas wordt verwerkt in de radar overlay en dus nadat de radarscan conversie is gedaan. Dit in tegenstelling tot de RSC-TB, waar de afterglow al in het B-Scope georiënteerde radarvideo verwerkt wordt. Hierdoor is het mogelijk om het in Hoofdstuk 5.1.1.4 beschreven probleem van incorrect geplaatste afterglow te vermijden. Hoe dit gedaan wordt en welke complicaties dit met zich meebrengt, wordt verder beschreven in Hoofdstuk 7.2.1.4.

7.2.1.3 Beeldopbouw PPI

Nadat de afterglow en de radarscan conversie verwerkt is in een nieuwe radar overlay, zijn alle componenten beschikbaar om het beeld verder op te bouwen. Hiertoe worden de verschillende componenten van vóór naar achter in verschillende diepte lagen getekend. Hierbij wordt gebruik gemaakt van *Z-Culling* [1] om zo min mogelijk pixels dubbel te tekenen. Tevens bestaat de mogelijkheid om per laag te kiezen of deze wel of niet weergegeven moet worden.

Als eerste wordt een simpele benadering van *tracks* weergegeven. In een echt systeem zijn dit iconen die de locaties van verschillende met sensoren gedetecteerde objecten weergegeven. In dit prototype zijn de locaties hiervan ongecorrleerd met de weergegeven radarvideo. Hierin zijn het simpele geometrische figuren die volgens een gegeven patroon over het scherm bewegen. Het doel van de weergave deze *tracks* is dan ook uitsluitend om aan te tonen dat dit eenvoudig te integreren is in het gegeven prototype.

Vervolgens wordt de scheepsoriëntatie weergegeven. Dit gebeurt door een viertal concentrische cirkels, gelijk verspreid over het totale bereik van de radar. Daarbij wordt met kruisende horizontale, verticale en diagonale lijnen de exacte locatie van het schip aangeduid. De lijn in de richting van de koers van het schip wordt hierbij met een hogere intensiteit afgebeeld. Al deze geometrie bij elkaar is opgeslagen in een display list voor het efficiënt renderen hiervan.

Na de scheepsoriëntatie wordt het panoramabeeld weergegeven. Hiervan worden de verschillende weergave mogelijkheden in meer detail beschreven in Hoofdstuk 7.2.4. Hierna wordt de textuur van de map op het scherm getekend.

Tenslotte wordt de gegenereerde radar overlay op het scherm getekend. Het resultaat hiervan kan echter nog op verscheidene wijzen beïnvloed worden. Dit wordt in meer detail behandeld in Hoofdstuk 7.2.1.6.

Het is bij al deze onderdelen van belang dat ze correct in verhouding tot elkaar op het scherm geprojecteerd worden. Hiervoor kunnen doorgaans twee projectievormen worden gehanteerd, namelijk *true motion* en *relative motion*. In het eerste geval staat een gekozen punt in de wereld centraal. Wanneer radarvideo geprojecteerd wordt op de kaart, blijft deze kaart stil staan en beweegt de radarvideo met de locatie van de radar als middelpunt over deze kaart. Bij relative motion staat de locatie van de radar centraal. Wanneer radarvideo geprojecteerd wordt op een kaart, lijkt deze kaart dus onder de radarvideo door te bewegen.

De correcte transformatie van alle onderdelen wordt gerealiseerd door gebruik te maken van de transformatie matrices van OpenGL. Deze worden, afhankelijk van alle beeldinstellingen, de scheepslocatie en oriëntatie en de gekozen projectievorm, voor elk van de genoemde onderdelen individueel ingesteld.

7.2.1.4 Probleem met afterglow in combinatie met relative motion projectievorm

Wanneer gekozen wordt voor de relative motion projectievorm moet er echter rekening gehouden worden met het in Hoofdstuk 5.1.1.4 genoemde probleem, waarbij afterglow met het schip mee beweegt. Dit is theoretisch tegen te gaan door de inhoud van de radar overlay continu te transleren in tegenovergestelde richting van de scheepsverplaatsing. Hierbij treedt er echter een ander probleem op, namelijk: hoe gaat het programma om met translaties kleiner dan één pixel.

Dit vormt een probleem doordat voor alle texture sample operaties *GL_NEAREST* sampling gedaan wordt zoals vermeld in Hoofdstuk 4.4. Hierdoor zal een translatie kleiner dan één pixel altijd naar boven, dan wel naar beneden afgerond worden. Na enkele van dergelijke translaties is het dus zeer waarschijnlijk dat de afterglow in meer of mindere mate afwijkt van de locatie op de map waarop ze aanvankelijk gepositioneerd waren.

De eerste mogelijkheid om dit probleem tegen te gaan, is door uitsluitend een translatie van gehele aantallen pixels uit te voeren. Hiertoe wordt continu een totale translatie bijgehouden, en wordt deze verminderd, wanneer translatie van een geheel aantal pixels uitgevoerd kan worden. Op deze wijze kan in ieder geval gegarandeerd worden, dat de positie van afterglow correct met de onderliggende map mee verplaatst. Een vervelend effect bijkomstig aan deze methode is echter dat het beeld lijkt te flikkeren. Dit wordt veroorzaakt doordat het menselijk oog erg gevoelig is voor veranderingen in een beeld. Dit flikkerend effect wordt dan ook sterker ervaren wanneer het beeld veel contrast bevat. Doordat alle pixels bij een dergelijke translatie één of meer pixels opschuiven, zal een groot gedeelte van de pixels over het gehele beeld een relatief grote verandering in de intensiteit ondergaan.

Een andere mogelijkheid is het gebruik van lineaire filtering bij de verplaatsing. Dit wordt bereikt door gebruik te maken van *GL_LINEAR* sampling. Dit heeft als voordeel ten opzichte van de hiervoor genoemde oplossing, dat de verplaatsing van de data in de radar overlays geheel vloeiend verloopt. Een ander probleem dat zich hierbij echter voordoet is de mogelijkheid tot verlies van data. Ter illustratie wordt uitgegaan van herhaaldelijke translaties van een halve pixel. Wanneer een pixel met maximale intensiteit omgeven is door pixels met een intensiteit van nul, zullen na de eerste translatie twee pixels ontstaan zijn met de helft van de maximum intensiteit. Verdere herhaling van translatie zorgt ervoor dat de oorspronkelijke pixel steeds verder wordt uitgesmeerd over de naburige pixels.

Tenslotte bestaat de mogelijkheid om de voorgaande scans geheel te bewaren en bij elke afbeelding het geheel opnieuw te tekenen met lineaire filtering. In daarop volgende afbeeldingen kan steeds teruggevallen worden op de oorspronkelijke scans en zal er dus geen informatie verloren gaan vanwege herhaaldelijk gebruik van de lineaire filtering. Een groot nadeel van deze methode is echter het extreem hoge resourcegebruik. In plaats van slechts twee radar overlays voor het gehele proces moet hierbij één overlay per gewenste hoeveelheid geschiedenis scans bewaard worden. Dit geeft zeker grote problemen wanneer de oneindige afterglow optie gebruikt wordt. Bovendien moet elke overlay ook steeds apart getekend worden waardoor niet alleen het geheugengebruik, maar ook de gebruikte GPU tijd drastisch zal toenemen.

Uiteraard kan ook overwogen worden om het probleem geheel te laten liggen en de werkwijze van de RSC-TB en diens incorrectheid op dit gebied te aanvaarden. Hiertoe hoeft er in het geheel geen translatie plaats te vinden op basis van veranderingen in de scheepspositie. Ook is het dan mogelijk om de verwerking van afterglow weer in B-Scope georiënteerd radarvideo te laten plaatsvinden.

7.2.1.5 Beeldopbouw B-Scope

Naast de weergave als PPI biedt het prototype ook nog de mogelijkheid om het radarvideo in B-Scope weer te geven. Hierbij is het in het prototype uitsluitend mogelijk om, naast het radarvideo, het panoramabeeld weer te geven. In deze situatie is het panorama uitsluitend horizontaal af te beelden, in lijn met het beschikbare radarvideo. Dit radarvideo kan echter nog wel afgebeeld worden met de verschillende, in het volgende sub-hoofdstuk beschreven methoden.

7.2.1.6 Radar overlay weergave mogelijkheden

Bij display van het radarvideo is het uiteraard het belangrijkste dat alles waar de operator op dat moment in geïnteresseerd is, zo duidelijk mogelijk zichtbaar is. Om dit doel te bereiken zijn er verschillende methoden ontwikkeld om de radar overlay weer te geven.

In de eerste plaats zijn er de verschillende mogelijkheden om het radarvideo afhankelijk van de intensiteit een kleur te geven. De eerste implementatie hiervan is het gebruik van de algemeen als standaard aanvaarde groene kleur. De intensiteit van deze kleur is recht evenredig met de intensiteit van het maximum van de huidige radarsample of de afterglow op die positie. De Cg code hiervoor staat in Code fragment 12.

```
float4 allGreenFP(float2 coords : TEXCOORD0,
                 sampler2D texture,
                 uniform float afterglowFactor,
                 uniform float4 alphaFactors) : COLOR
{
    //result.r = afterglow intensity
    //result.g = current radarsample intensity
    float4 result = tex2D(texture, coords);

    //place max of radarsample and afterglow (if enabled) in green
    result.g = (result.r*afterglowFactor > result.g ?
               result.r*afterglowFactor : result.g);
    //clear red and blue values
    result.rb = 0.0;
    //adjust alpha according to alpha factors and sample intensity
    result.a = (result.g > 0 ?
               alphaFactors.x+alphaFactors.y*pow(alphaFactors.z , 1.0-result.g) : 0);

    return result;
}
```

Code fragment 12 Cg code voor weergave van radarvideo in groene kleur

Een andere, zeer veelzijdige methode, is het gebruik van een color lookup table. De intensiteit van de radar overlay wordt hierbij gebruikt als index in een texture van 256 x 1 pixels. Doordat kleurinhoud van deze texture vrij gekozen kan worden, kan elk willekeurige verband met de intensiteit bereikt worden. Mogelijkheden zijn bijvoorbeeld een texture met een logaritmisch kleurverloop, of textures met meerdere verschillende kleuren om op interessante intensiteit regionen extra contrast te verkrijgen. De implementatie van deze wijze van kleuring is gegeven in Code fragment 13.

Gezien de intensiteit van de meest recente radarsample en de waarde van afterglow op die locatie beiden bekend zijn in de radar overlay, kan deze informatie ook gebruikt worden om visueel onderscheid tussen de beide te maken. De hardwarematige oplossing die enkele generaties vóór de RSC-TB gebruikt werden, de RSC en de RSC-2V, waren in staat om over het spoor van de afterglow de intensiteit te laten pulseren. Hierdoor werd de indruk van beweging vergregen van de oudst gedetecteerde naar de nieuwste locatie van de reflecties. Dit effect wordt *ripple* genoemd en had als doel om bewegende objecten ook in gebieden met veel clutter van bijvoorbeeld wolken goed zichtbaar te houden. Deze mogelijkheid is vanwege technische beperkingen verloren gegaan in de RSC-TB. Om aan te tonen dat deze techniek wel weer toegepast kan worden in de RSC-GPU, is hiervan een implementatie gemaakt zoals beschreven in Code fragment 14. Hierbij wordt over de afterglow de index voor de color lookup texture continu gevarieerd en op deze wijze het ripple effect veroorzaakt.

```
float4 lookupFP(float2 coords : TEXCOORD0,
               sampler2D texture,
               sampler2D lookup,
               uniform float afterglowFactor,
               uniform float4 alphaFactors) : COLOR
{
    //intensity.r = afterglow intensity
    //intensity.g = current radarsample intensity
    float4 intensity = tex2D(texture, coords);

    //determine max of radarsample and afterglow (if enabled)
    intensity.b = max(intensity.r*afterglowFactor, intensity.g);

    //lookup color with derrived intensity
    float4 result = tex2D(lookup, float2(intensity.b, 0));

    //adjust alpha according to alpha factors and sample intensity
    result.a = (intensity.b > 0 ? alphaFactors.x + alphaFactors.y *
        pow(alphaFactors.z , 1.0-intensity.b) : 0);

    return result;
}
```

Code fragment 13 Cg code voor weergave van radarvideo met color lookup texture

```
float4 lookupRippleFP(float2 coords : TEXCOORD0,
                     sampler2D texture,
                     sampler2D lookup,
                     uniform float displacement,
                     uniform float afterglowFactor,
                     uniform float4 alphaFactors) : COLOR
{
    //intensity.r = afterglow intensity
    //intensity.g = current radarsample intensity
    float4 intensity = tex2D(texture, coords);

    float factoredAfterglow = afterglowFactor*intensity.r;

    //calculate rippled afterglow intensity and place in alpha channel
    intensity.a = (factoredAfterglow < displacement ?
        1 + factoredAfterglow - displacement :
        factoredAfterglow - displacement);

    //determine max of radarsample and afterglow (if enabled)
    intensity.b = (factoredAfterglow > intensity.g ?
        max(intensity.a, intensity.g) :
        intensity.g);

    //lookup color with derived intensity
    float4 result = tex2D(lookup, float2(intensity.b, 0));

    //adjust alpha according to alpha factors and sample intensity
    intensity.b = max(factoredAfterglow, intensity.g);
    result.a = (intensity.b > 0 ? alphaFactors.x + alphaFactors.y *
        pow(alphaFactors.z , 1.0-intensity.b) : 0);

    return result;
}
```

Code fragment 14 Cg code voor weergave van radarvideo met color lookup en ripple

Behalve het gebruik van verschillende kleuren, kan ook de zichtbaarheid van interessante delen in radarvideo vergroot worden door de delen met lage intensiteit, zoals ruis, weg te filteren. Hiertoe zijn verschillende filters geïmplementeerd. Een voorbeeld hiervan is de offset – auto-gain filtering, te zien in Code fragment 15. Hierbij wordt een instelbaar deel van de maximum intensiteit van de samples in de radar overlay afgetrokken. Daarna wordt de overgebleven intensiteit opnieuw geschaald zodat weer het volledige dynamische intensiteit bereik benut wordt.

```
float4 allGreenOffsetGainFP(float2 coords : TEXCOORD0,
                           sampler2D texture,
                           uniform float afterglowFactor,
                           uniform float4 alphaFactors,
                           uniform float4 ogFactors) : COLOR
{
    //result.r = afterglow intensity
    //result.g = current radarsample intensity
    float4 result = tex2D(texture, coords);

    //place max of radarsample and afterglow (if enabled) in green
    result.g = (result.r*afterglowFactor > result.g ?
               result.r*afterglowFactor : result.g);

    //clear red and blue values
    result.rb = 0.0;

    //process offset and gain factors
    result.g = (result.g-ogFactors.a)*ogFactors.r;

    //adjust alpha according to alpha factors and sample intensity
    result.a = (result.g > 0 ? alphaFactors.x + alphaFactors.y *
                          pow(alphaFactors.z , 1.0-result.g) : 0);

    return result;
}
```

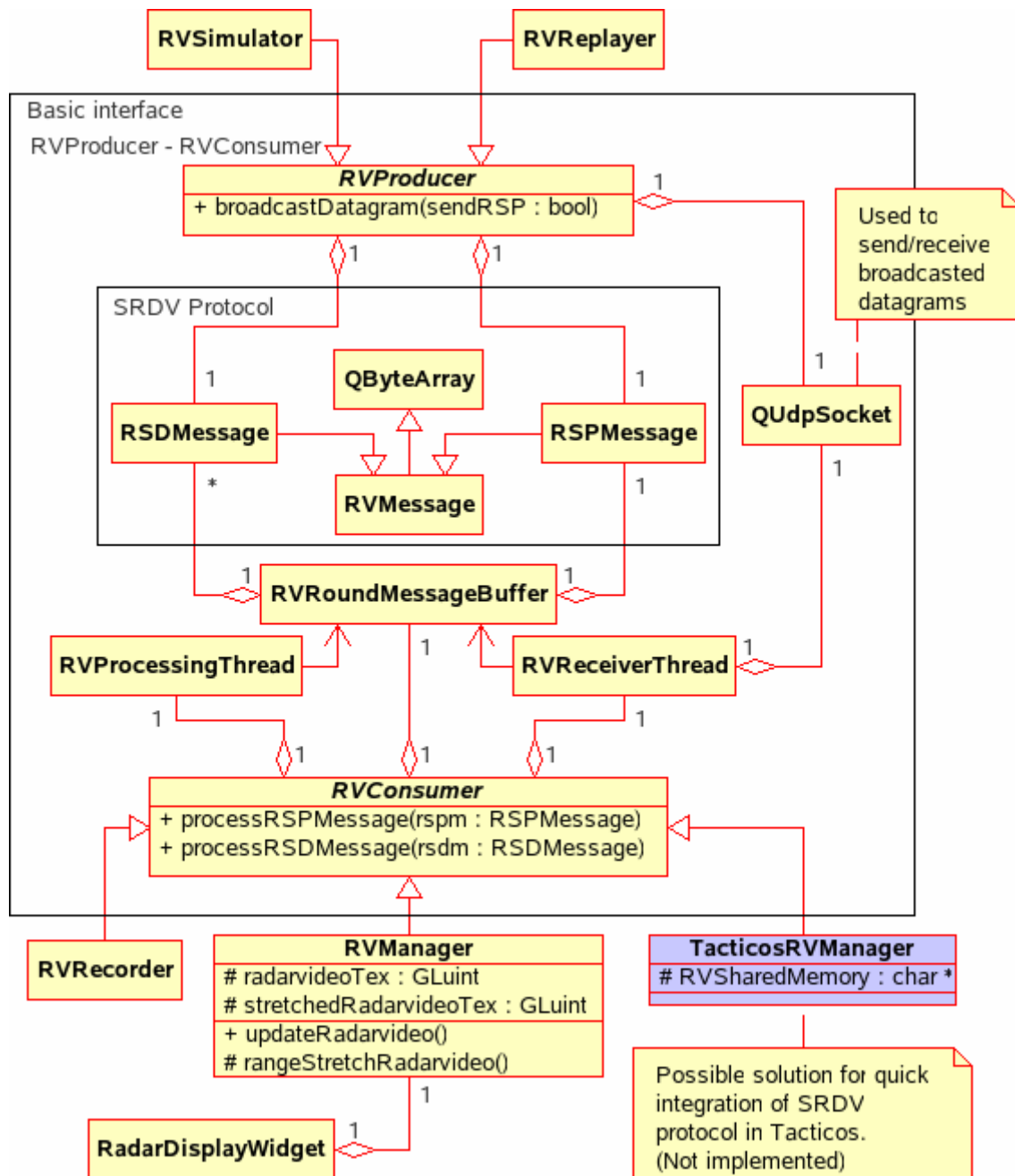
Code fragment 15 Cg code voor weergave van radarvideo met offset – gain filter

Behalve deze wijze van filtering, waarbij radarsamples geheel worden weg gefilterd, bestaat ook de mogelijkheid om de zichtbaarheid van de samples te verminderen. Op deze wijze kan bijvoorbeeld de onderliggende map meer zichtbaar gemaakt worden. Dit is geïmplementeerd door gebruik te maken van alpha-blending, de wijze waarop transparantie in OpenGL is geïmplementeerd. In alle hiervoor vermelde code fragmenten is ook te zien hoe de alpha waarde van de output afhankelijk van de intensiteit wordt ingesteld. Hierdoor zullen interessante delen in de radarvideo, de delen met hoge intensiteit, minder snel verdwijnen in de achtergrond.

7.2.2 Radarvideo

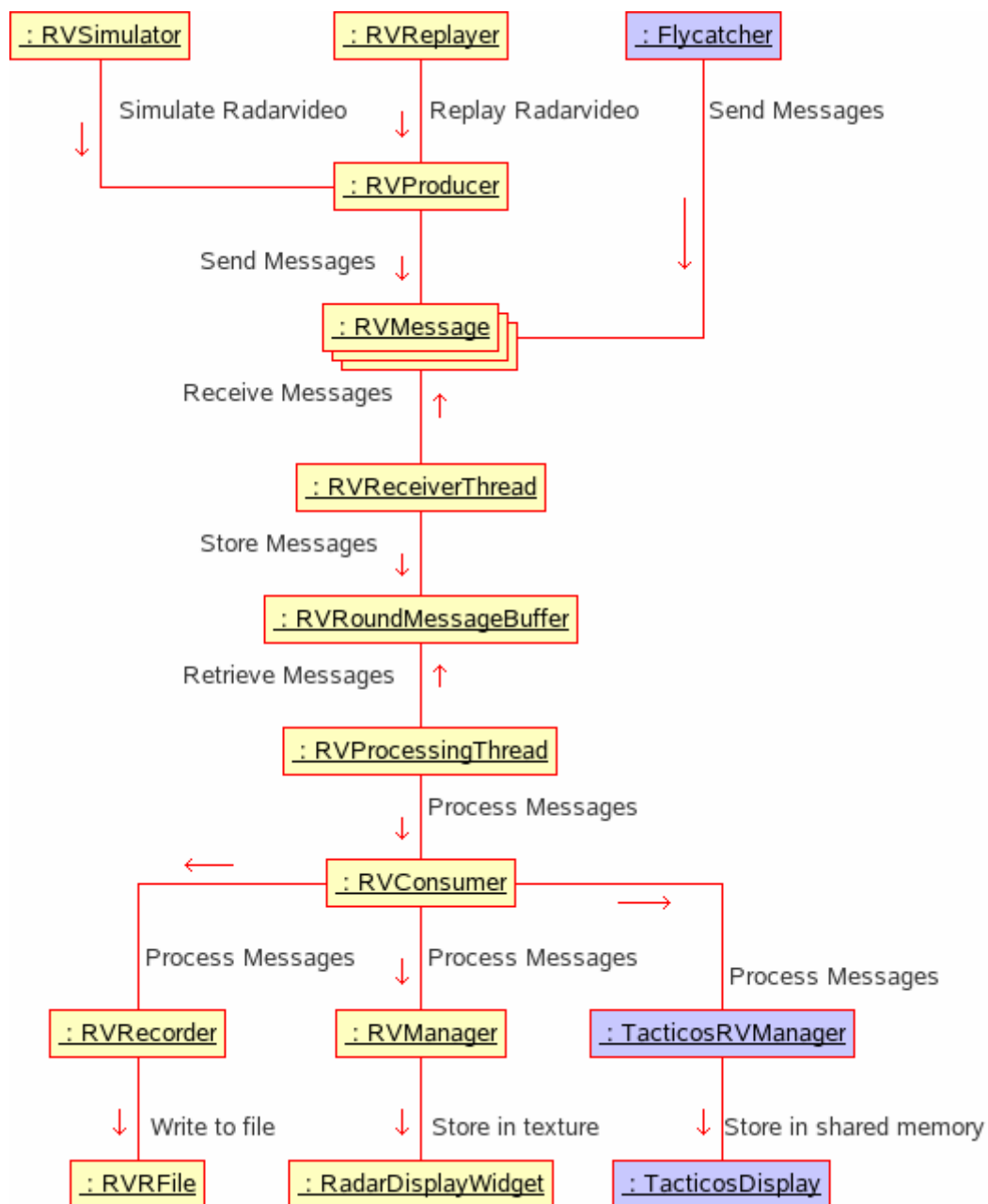
In de vroege ontwikkelingsfasen van het prototype, werden testpatronen als radarvideo binnen de applicatie zelf gegenereerd. Hiervoor was dan ook geen enkele vorm van distributie nodig. Omdat de focus van dit onderzoek gaandeweg kwam te liggen op een volledig radarscan conversie systeem met een digitale distributie architectuur, ging dit wel een rol spelen. Zoals uitgelegd in Hoofdstuk 6, zal deze distributie plaats vinden middels een ethernet verbinding. Vanwege redenen welke genoemd worden in Hoofdstuk 8.1, is voor het protocol over deze ethernet verbinding gebruik gemaakt van het SRDV protocol.

Het klassendiagram van de interface tussen de radarvideo producent en consument is te zien in Figuur 37. De berichten van het protocol zijn geïmplementeerd volgens de specificaties gegeven Figuur 32 en Figuur 33 op pagina 50 en 51. Deze worden via UDP broadcast verstuurd door de producent. De informatie in deze berichten is uiteraard afhankelijk van het type producent, welke allen overerven van de algemene *RVProducer* klasse. De *RVSimulator* vult deze bijvoorbeeld met zelf gegenereerde testpatronen.



Figuur 37 Klassendiagram radarvideo producent – consument interface

De verzonden berichten van het SRDV protocol worden in de *RVConsumer* door een tweetal threads afgehandeld. De *RVReceiverThread*, welke in het bezit is van de UDP socket, slaat alle ontvangen berichten op in de *RVRoundMessageBuffer*. De *RVProcessingThread* haalt deze berichten hieruit en laat ze verwerken door de *RVConsumer*. Middels een semafoor wordt de toegang tot dit buffer gesynchroniseerd en wordt de *RVProcessingThread* van de aanwezigheid van nieuwe berichten op de hoogte gesteld. De *RVRoundMessageBuffer* is dusdanig geconstrueerd dat de lezer ingehaald kan worden door de schrijver. Hierdoor komt de verwerking van nieuw radarvideo data nooit in het geding wanneer de *RVProcessingThread* om welke reden dan ook een achterstand heeft opgelopen met het verwerken van radarvideo. Hierbij wordt altijd de oudste radarvideo data overschreven. In Figuur 38 is het collaboration diagram van dit proces weergegeven.



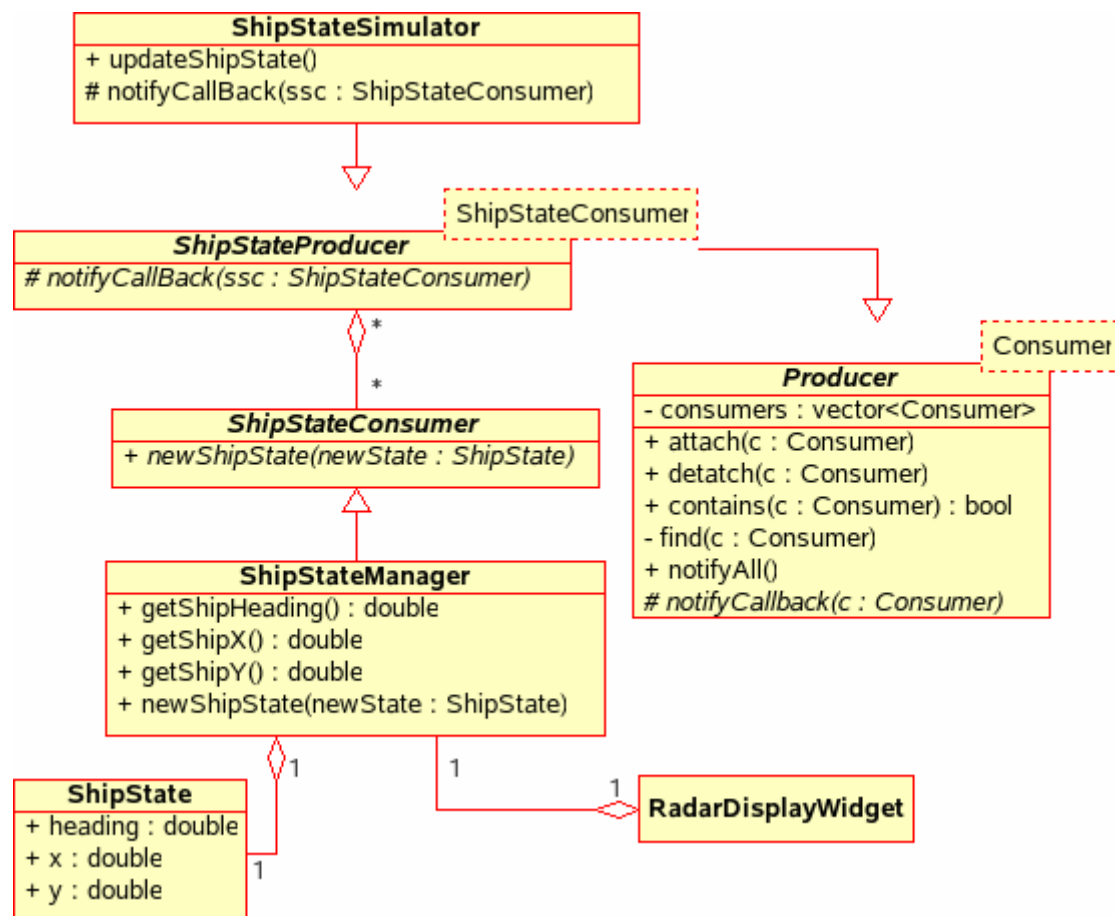
Figuur 38 Radarvideo producent – consument collaboration diagram

In dit prototype is de *RVManager* als radarvideo consument verantwoordelijk voor het ontvangen en verwerken van de radarvideo. Dit verwerken bestaat in de eerste plaats uit het samenvoegen van de radarsweeps in de verschillende RSD berichten, tot een B-Scope georiënteerd totaalbeeld van de radarvideo. Dit wordt in de eerste instantie in CPU geheugen gedaan. Wanneer de *RadarDisplayWidget*, met actieve OpenGL context, de *updateRadarVideo* functie aanroept, kan de radarvideo naar een texture gekopieerd worden.

Nadat de nieuwe radarvideo in een texture gedownload is, kan ook de afstand normalisatie door de GPU uitgevoerd worden. Dit gebeurt door het uitvoeren van de algoritmen zoals beschreven in Hoofdstuk 4.2.2. Deze genormaliseerde data wordt in een aparte texture weggeschreven. Hiermee blijft alle oorspronkelijke radarvideo data nog beschikbaar. Dit is immers nodig voor het opnieuw uitvoeren van afstand normalisatie over de gehele radarvideo. Dit kan nodig zijn wanneer de visualisatie instellingen veranderd worden.

7.2.3 Scheepstoestand

De scheepstoestand, bestaande uit een positie en een koers, is belangrijk voor het correct positioneren van alle onderdelen in het display. Om deze informatie beschikbaar te stellen aan de *RadarDisplayWidget*, is een modulaair systeem ontwikkeld dat bestaat uit een producent en consument van deze informatie. De keuze hiervoor is gemaakt om, wanneer gewenst, het makken van verschillende producenten makkelijk mogelijk te maken. Deze kunnen bijvoorbeeld de informatie afkomstig van verschillende bronnen doorgeven. Het klassendiagram van de hiervoor gebruikte architectuur is weergegeven in Figuur 39.



Figuur 39 Klassendiagram scheepstoestand producent – consument interface

De abstractie van een Producer – Consumer model gebaseerd op het Bridge Design Pattern [8] staat centraal in dit ontwerp. Hiermee wordt een interface beschreven waarbij beide componenten onafhankelijk van elkaar veranderd kunnen worden. Hierdoor kunnen op eenvoudige wijze verschillende producenten dan wel consumenten van de scheepstoestand geïmplementeerd worden.

De C++ template *Producer* ligt hieraan ten grondslag. Deze biedt zelf alle mogelijkheden voor administratie van een bepaald type consument. Hierbij valt te denken aan het toevoegen en verwijderen van consumenten of de geregistreerde consumenten van nieuwe informatie voorzien. Concrete implementaties van deze template leggen vervolgens vast, welk type consument er bediend wordt, en hoe deze van nieuwe informatie moeten worden voorzien. Met deze template als generalisatie kan het Bridge Design Pattern eenvoudig worden toegepast op producenten en consumenten van verschillende data-types.

Voor de scheepstoestand is in het prototype slechts één type producent gemaakt. Deze *ShipStateSimulator* genereert zelf een scheepspositie volgens een vast patroon. De koers wordt dynamisch uit het verschil tussen een oude en een nieuwe positie bepaald. De *ShipStateManager* update hiermee zijn intern gehouden scheepstoestand en stelt deze informatie beschikbaar aan de *RadarDisplayWidget*.

7.2.4 Panorama

Gedurende het onderzoek groeide het besef dat de GPU, naast de nu bestaande toepassingen, ook in staat is nieuwe, revolutionaire ontwikkelingen te realiseren. Om hier wat mee te experimenteren is bedacht om de weergave van radarvideo te fuseren met panoramisch TV video. Dit lijkt een geschikte vorm van sensorfusie, gezien beide informatie over 360 graden bevat. De operator zou dan mogelijkwerwijs objecten, die op korte afstand gedetecteerd zijn door de radar, aan de hand van de TV video kunnen identificeren. Dit kan handig zijn bij bijvoorbeeld het gebruik van navigatie radars. In dergelijke gevallen zou automatisch de aandacht op locaties in het panorama gevestigd kunnen worden. In ieder geval moet het de mogelijkheid bieden om informatie van beide sensoren in één oogopslag te kunnen overzien.

De genoemde panoramische TV video wordt geproduceerd door Gatekeeper, een project dat op dit moment nog volop in ontwikkeling is. Het bestaat onder meer uit een opstelling van 9 camera's die gepositioneerd zijn in een krans, en software die de beeldinformatie van deze camera's verwerkt. Het samengevoegde panorama, welke in totaal 5490 x 480 pixels beslaat, wordt gedistribueerd via SPLICE. Dit is een data distributie service gekarakteriseerd door autonome applicaties, waarbij de functies van, en de interactie tussen de applicaties gescheiden zijn. Het kan opgevat worden als een database waar applicaties als producent data in kunnen publiceren. Andere applicaties kunnen zich op deze data abonneren om deze bijvoorbeeld weer te geven, en eventueel het resultaat van verdere verwerkingen van de data weer voor geïnteresseerden te publiceren [7]. Een voorbeeld van een panorama dat door dit systeem geproduceerd wordt, is te zien in Figuur 40.

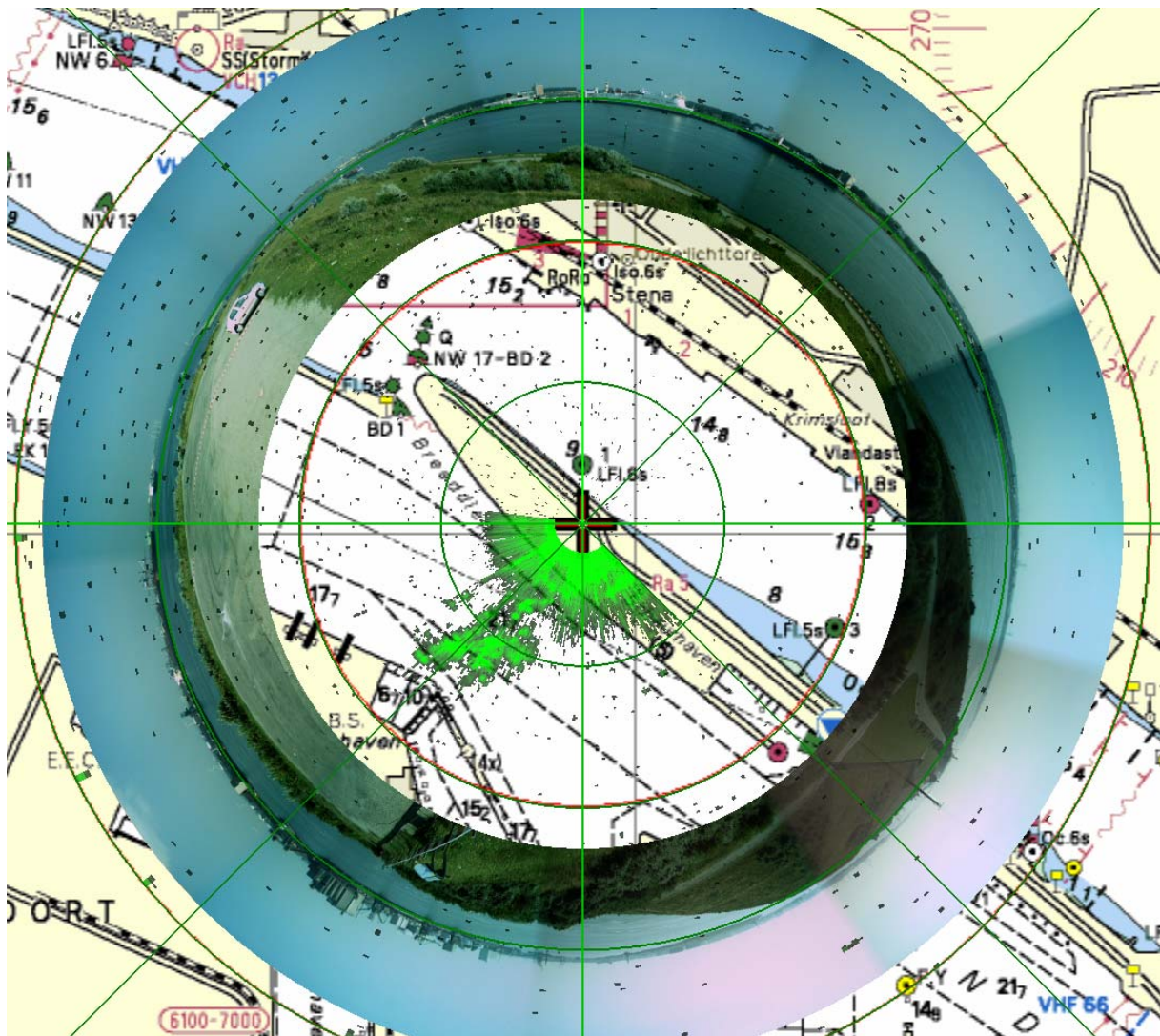


Figuur 40 Voorbeeld panoramisch TV video gegenereerd door Gatekeeper

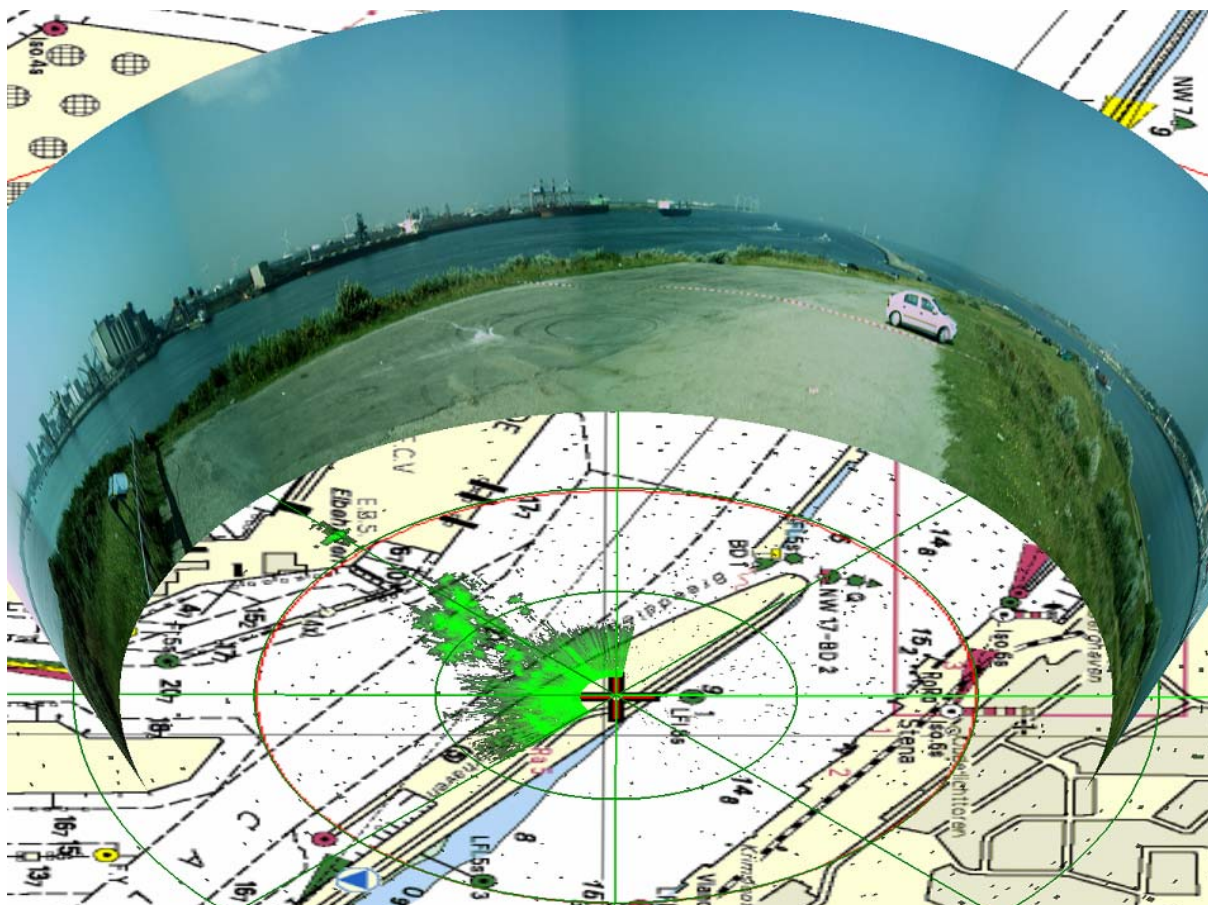
7.2.4.1 Visualisatievormen

Gaandeweg zijn er verscheidene visualisatievormen tot stand gekomen. De ideeën hiervoor zijn gevormd door enkele brainstormsessies, waarbij ook steeds inspiratie werd opgedaan uit reeds gemaakte implementaties. In de volgende paragrafen worden alle visualisatievormen kort besproken. Hierbij moet wel opgemerkt worden dat het panorama, getoond in de afbeeldingen niet werkelijk correleert met de radarvideo. Het komt echter wel in locatie en oriëntatie overeen met de map.

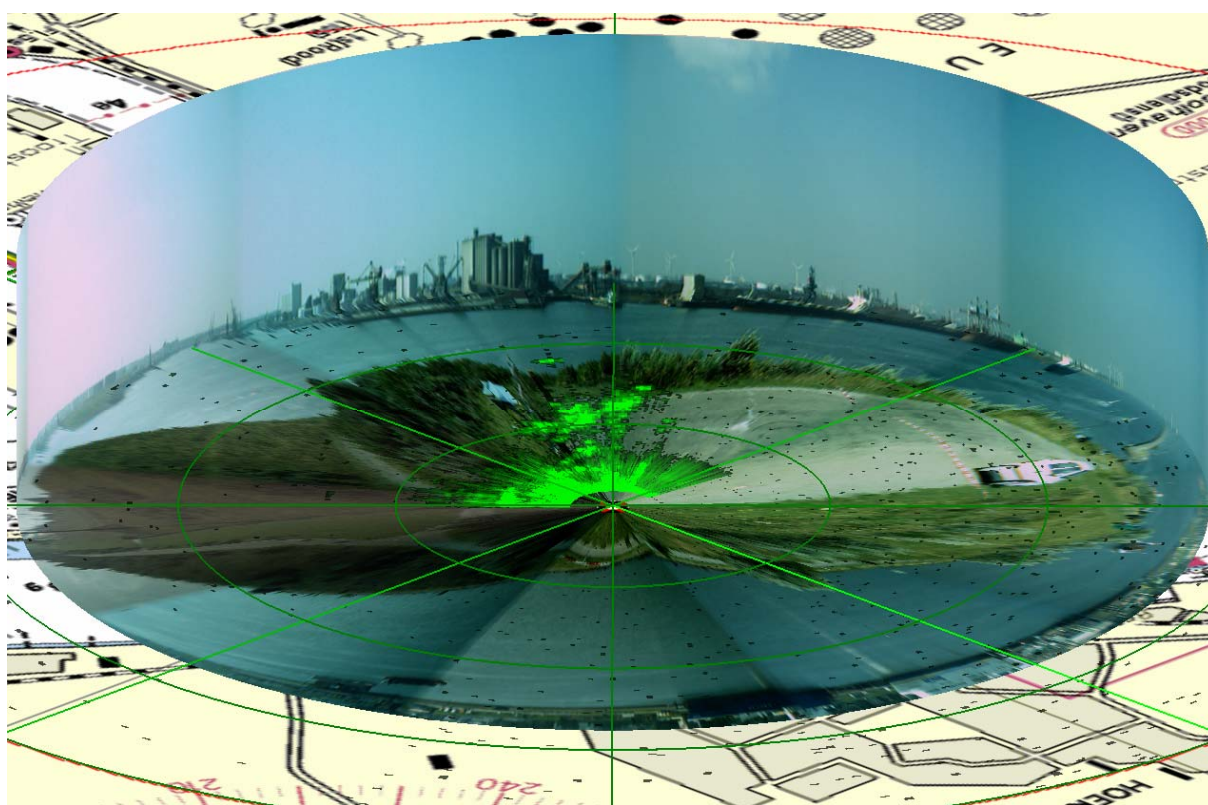
In de eerste plaats is een platte projectie ontwikkeld. Hierbij wordt het panorama als een ring om het middelpunt van de PPI afgebeeld. Dit is te zien in Figuur 41. De grootte van de ring is onafhankelijk van het radarvideo te schalen. Dit maakt het eenvoudig mogelijk om het radarvideo in verband te leggen met het panorama. Hiertoe kan een lijn getrokken worden vanuit het middelpunt van de PPI. De beeldinformatie in het panorama op die lijn komt overeen met de informatie in de radarvideo op diezelfde lijn.



Figuur 41 Plat panorama

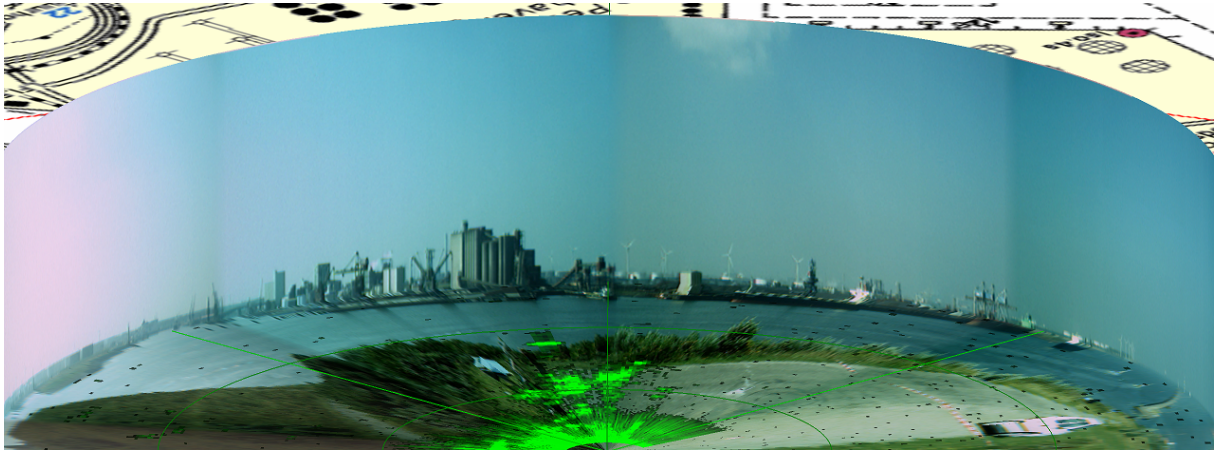


Figuur 43 Orthogonaal geprojecteerd panorama



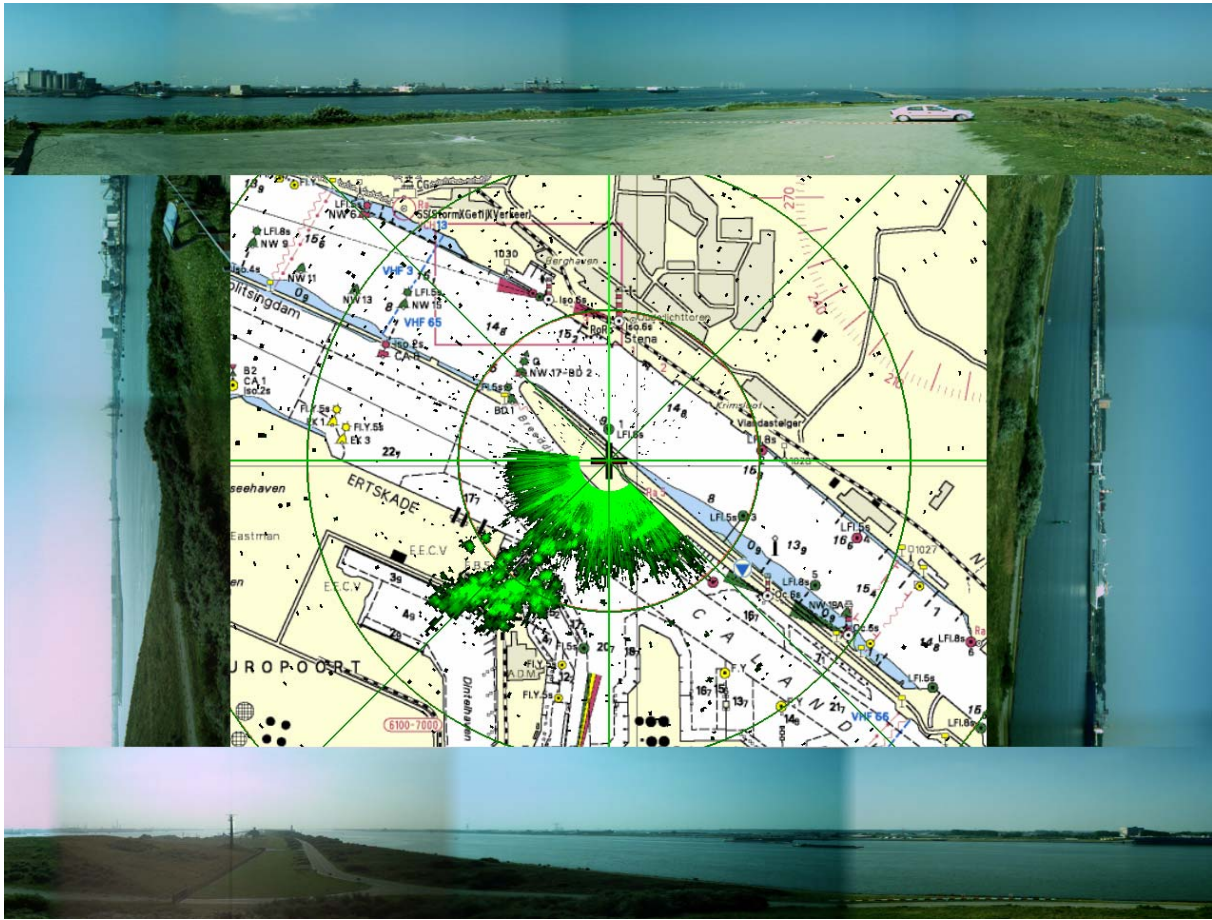
Figuur 44 Op horizon geknikt panorama, overzicht

Bij deze visualisatievorm wordt met name een verrassend inlevingsgevoel gecreëerd, wanneer het middelpunt van de PPI midden onder in het scherm wordt afgebeeld onder een bepaalde hoek. Wanneer het beeld in deze situatie geroteerd wordt, komt dit gevoelsmatig sterk overeen met het rondkijken op de betreffende locatie. Het is echter voornamelijk aan de zijkanten nog wel zichtbaar dat dit slechts een illusie is. Dit gezien het vlak onder de horizon geheel plat wordt weergegeven, terwijl deze wel degelijk ook elevatie van objecten weergeeft. In de figuren is dit duidelijk te zien bij de auto aan de rechterkant van het beeld.



Figuur 45 Op horizon geknikt panorama, vanuit centrum weergegeven

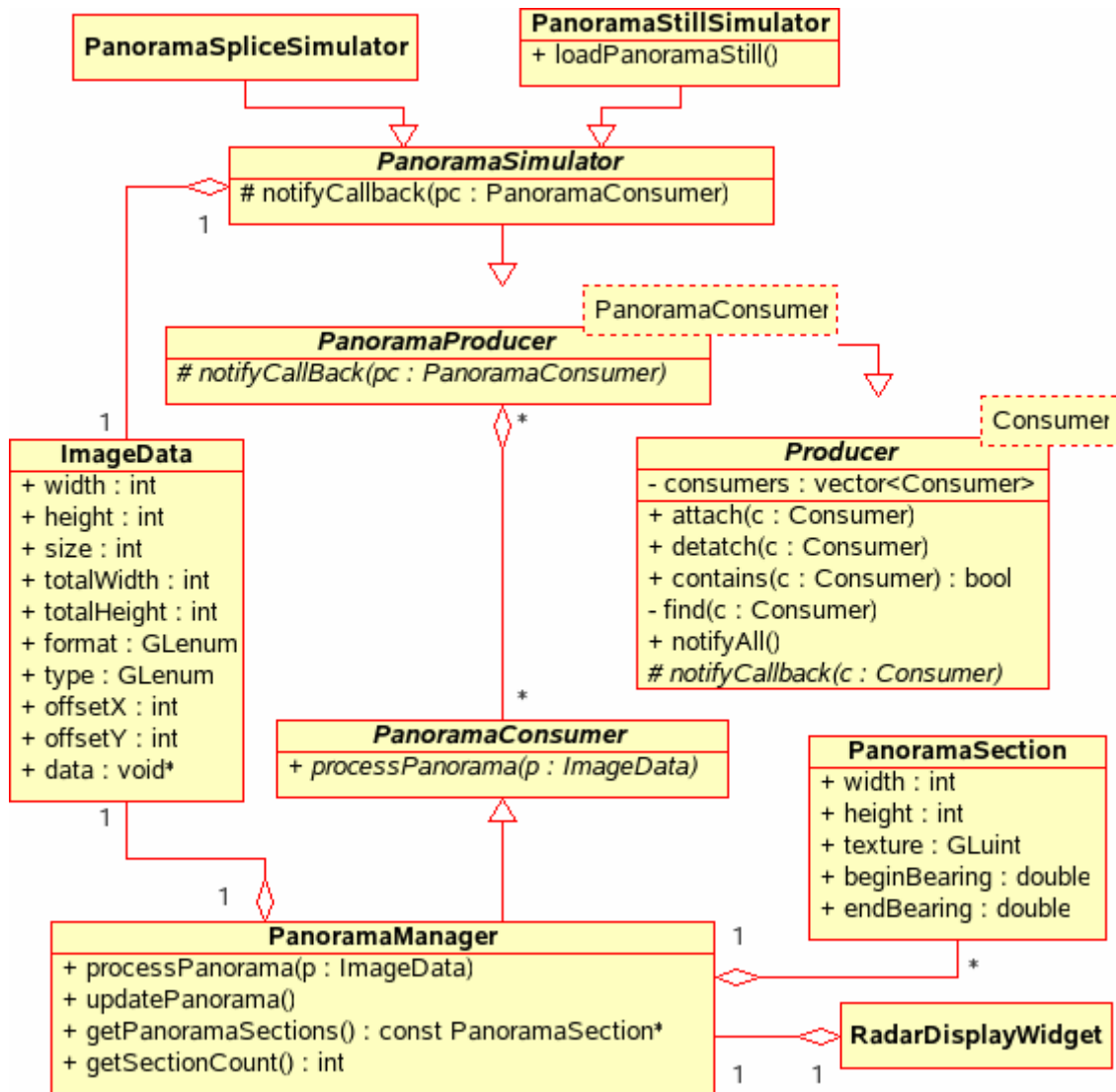
Tenslotte is er nog een visualisatievorm bedacht, waarbij geen rekening is gehouden met het zo realistisch mogelijk weergeven van het panorama. In plaats daarvan ligt de nadruk op het continu zichtbaar zijn van alle informatie in het panorama, naast een onvervormde weergave van de radarvideo. In Figuur 46 is de bedachte oplossing te zien, waarbij het panorama langs de randen van het beeldscherm is geprojecteerd. Hierbij is het beeld langs de onderzijde van het scherm, dat voor de operator achter hem ligt in de huidige kijkrichting, in spiegelbeeld weergegeven. Dit komt de intuïtiviteit van het beeld, zeker wanneer het geroteerd wordt, zeer ten goede. Deze visualisatievorm heeft ten opzichte van de eerste als voordeel, dat het panorama minder prominent in beeld is, maar meer een ondersteunende rol heeft.



Figuur 46 Panorama geprojecteerd langs randen van het werkvlak

7.2.4.2 Architectuur van de implementatie

De implementatie van het panorama component is net zo opgezet als die van de scheepstoestand. In Figuur 47 is een klassendiagram hiervan te zien. Hierin is te zien dat ook voor het panorama de gegeneraliseerde *Producer* klasse de basis legt voor de flexibele koppeling tussen de producent en consument volgens het Bridge Design Pattern.



Figuur 47 Klassendiagram panorama producent – consument interface

Voor de informatieoverdracht tussen de producent en de consument is de klasse *ImageData* gemaakt. Deze bevat alle benodigde informatie om een panoramasegment te plaatsen in het complete panoramabeeld, en om deze middels *glTex(Sub)Image* in het grafische geheugen als texture in te laden. Hiertoe dient de functie *updatePanorama* door de *RadarDisplayWidget* aangeroepen te worden met een actieve OpenGL context. Belangrijk is het om hierbij rekening te houden met de op de gebruikte hardware geldende maximum texture dimensie van 4096 pixels. Dit betekent dat het gebruikte panorama opgedeeld moet worden in twee textures. Wanneer meer, of hogere resolutie camera's gebruikt gaan worden, kunnen hiervoor zelfs nog meer textures noodzakelijk zijn.

Om hierop in te kunnen spelen is een flexibel mechanisme bedacht om het panorama op te delen in meerdere textures, en de informatie over de welke sectie van het totale panorama de betreffende texture bevat. Dit wordt opgeslagen in een aantal objecten van de *PanoramaSection* struct. Deze bevatten de texture ID, de hoogte en breedte in pixels en de hoeken van het begin en einde van de texture in graden. Met deze informatie kan de *RadarDisplayWidget* alle hiervoor genoemde visualisatievormen realiseren.

Hiertoe wordt, afhankelijk van de gekozen visualisatievorm, één of meerdere stukken geometrie met texture coördinaten gegenereerd. Deze worden met de bijbehorende textures van de *PanoramaSections* getekend. Vanwege de zeer grote hoeveelheid data, heeft dit echter wel een vrij grote impact op de performance. Meer hierover in Hoofdstuk 9.1.

Voor het panorama is aanvankelijk geen koppeling met het eerder genoemde SPLICE gemaakt. In plaats hiervan is gebruik gemaakt van één screenshot. De bijbehorende panoramasimulator behoeft daartoe enkel éénmaal de betreffende afbeelding in te lezen. Deze kan vervolgens herhaaldelijk naar de *PanoramaManager* gestuurd worden. Omdat het werkelijke systeem momenteel het panorama met maximaal 6 Hz kan verwerken, wordt deze update ook met dezelfde frequentie uitgevoerd. Hiermee wordt een gelijkwaardige belasting van het systeem gegenereerd.

Tegen het einde van het onderzoek is, in samenwerking met enkele ontwikkelaars van Gatekeeper, een nieuwe panoramasimulator geïmplementeerd met een SPLICE interface. De exacte werking van deze interface valt buiten de scope van dit onderzoek. Voldoende is het te vermelden dat deze via SPLICE met de genoemde 6 Hz updates binnen krijgt van het panorama video. Deze wordt vervolgens via de genoemde producent – consument interface verstuurd naar de *PanoramaManager*. Hiermee is in het prototype werkelijk panoramisch video af te beelden met de genoemde visualisatievormen.

8 Integratie van gelogd radarvideo

Zoals eerder is vermeld, was het aanvankelijk uitsluitend mogelijk om zelf gegenereerd radarvideo af te beelden. Hiervoor zijn verscheidene testpatronen beschikbaar, alsmede een kunstmatige radar waarbij enkele objecten volgens bepaalde patronen over een achtergrond van ruis bewegen. Voor meer realisme en om de geboden functies beter te kunnen testen, is het wenselijk om in plaats hiervan echt, gelogd radarvideo af te kunnen beelden. In dit hoofdstuk worden de hiervoor beschikbare mogelijkheden geanalyseerd en de implementatie van de uiteindelijk geïmplementeerde oplossing beschreven.

8.1 Analyse mogelijkheden

De eerste optie die voor deze integratie beschouwd is, zijn de opnames afkomstig van de LW08. Dit is een radar met een analoge distributiearchitectuur, welke is uitgelegd in Hoofdstuk 5.2. De opnames zijn gemaakt in een formaat dat door Thales zelf gedefinieerd is en is beschreven in [5]. Met behulp van deze specificaties moet het mogelijk zijn hiervoor een producent te maken, die deze data kan doorgeven aan het prototype.

Een andere optie is om zelf een opnamen te maken van testsessies van de Flycatcher Mk2, een recentelijk ontwikkeld mobiel radarsysteem, die al van een geheel digitale radarvideo distributie architectuur is voorzien. Deze architectuur wordt uitgelegd in Hoofdstuk 5.3. Een soortgelijke digitale architectuur is, zoals beschreven in Hoofdstuk 6, ook beoogd als werkomgeving voor de ontwikkelde RSC en heeft daardoor een grotere meerwaarde.

Doorslaggevend in de beslissing hierover is de aanzet voor implementatie van de in Hoofdstuk 6 vermelde distributie architectuur geweest. Hierbij is de overstap gemaakt van in het prototype zelf gegenereerd radarvideo naar via UDP gedistribueerd radarvideo. Om dit mogelijk te maken, moet er in ieder geval een protocol gedefinieerd worden waarin het formaat van de radarvideo zelf en de eigenschappen hiervan beschreven worden. Door hiervoor het in Hoofdstuk 5.3 beschreven SRDV protocol te hanteren, kunnen twee vliegen in één klap geslagen worden.

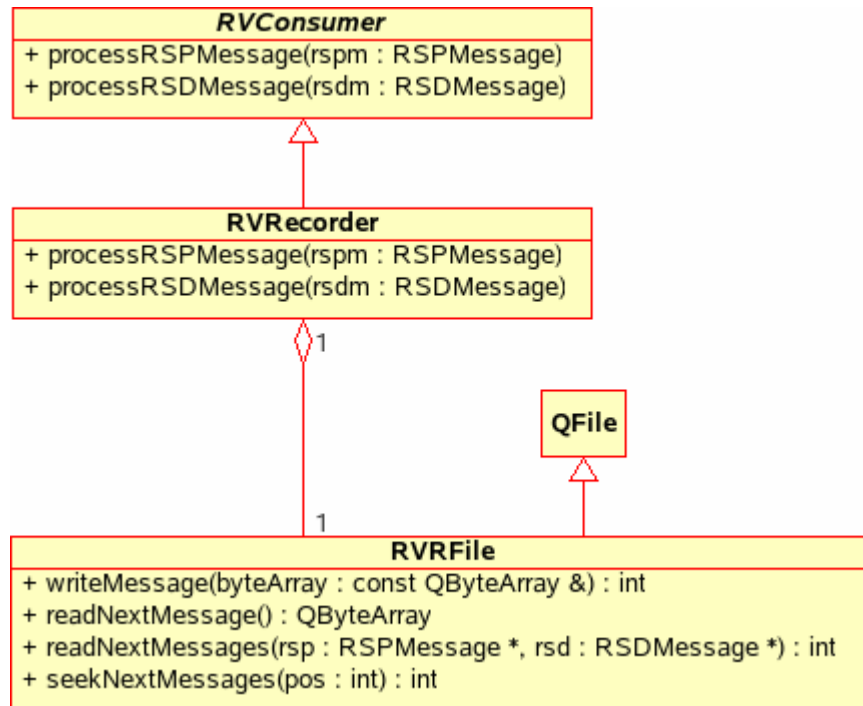
Ten eerste kan hiermee een onderzoek naar alle benodigde eigenschappen van een radar voor het afbeelden van radarvideo hiervan uitgespaard worden. Dit protocol wordt immers al succesvol gebruikt in de Flycatcher Mk2.

Ten tweede zal het prototype, bij correcte implementatie van het protocol, direct compatible zijn met het radarvideo van de Flycatcher Mk2. Dit gegeven en het feit dat deze het radarvideo als broadcast verstuurd, maakt het theoretisch erg eenvoudig om het radarvideo op te vangen en in het prototype weer te geven.

Zoals beschreven in Hoofdstuk 7.2.2 is dan ook gekozen om dit protocol te implementeren in het prototype. De modulaire opzet van radarvideo consumenten en producenten maken het daarnaast mogelijk om op eenvoudige wijze een recorder als consument en een replayer als producent te implementeren.

8.2 Radarvideo recorder

De eerste stap die genomen moet worden voor het integreren van radarvideo opnames, is het produceren van de opnames zelf. Hiervoor moet dus een recorder van radarvideo geschreven worden. Deze *RVRecorder* is, zoals te zien in Figuur 48, geïmplementeerd als sub-klasse van een generieke *RVConsumer*. Hiertoe wordt het verwerken van de berichten van het SRDV protocol aangepast door deze naar harde schijf weg te schrijven.



Figuur 48 Klassendiagram radarvideo recorder

In dit proces speelt de *RVRFile* een grote rol. Deze sub-klasse van *QFile*, de standaard klasse voor bestandstoegang in Qt, is in staat om de SRDV berichten eenduidig op te slaan en deze ook weer uit te lezen. Hiertoe wordt eerst de grootte van het weg te schrijven bericht in een 4 bytes grote integer de file gezet als micro-header. Vervolgens wordt het RSP of RSD bericht in zijn geheel weggeschreven. Hierbij wordt altijd gezorgd dat het eerste bericht dat werkelijk weggeschreven wordt een RSP bericht is. Dit bericht, dat éénmaal per radarscan verstuurd wordt, bevat alle informatie over de radar die noodzakelijk is om de werkelijke radarvideo in de RSD berichten af te kunnen beelden. RSD berichten die vóór een RSP bericht door de consument ontvangen worden, kunnen daardoor ook niet nuttig verwerkt worden. Het is dus onzinnig om deze berichten toch op te slaan in het logbestand.

In Figuur 49 is een screenshot te zien van de radarvideo recorder zoals deze uiteindelijk is geïmplementeerd. Zodra de opname gestart wordt, zullen alle SRDV berichten, ontvangen op de geselecteerde poort en afkomstig van de aangegeven bron, weggeschreven worden in het aangewezen logbestand. Het is hierbij ook mogelijk om automatisch bestandsnamen te laten genereren. Opnames krijgen dan een bestandsnaam voorzien van een tijdstempel van het moment dat de opname gestart is. Hiervoor wordt het volgende formaat gehanteerd: `recording_DD-MM-JJ_hh.mm(_n).rvr`. In de naam is dus de huidige datum en tijd in uren en minuten opgenomen. Indien binnen één minuut twee of meer bestandsnamen gegenereerd worden, wordt een extra uniek `_n` achtervoegsel gegenereerd, waarbij `n` oploopt vanaf 0.

Door deze automatische bestandsnaam generatie kunnen eenvoudig opnames gestart en herstart worden, zonder reeds gemaakte logbestanden te overschrijven of continu handmatig nieuwe bestandsnamen in te voeren. Dit is met name praktisch, aangezien radars veelal in hectische situaties gebruikt worden. Hierbij wil een operator dan ook zo min mogelijk tijd en moeite steken in het maken van een logbestand.

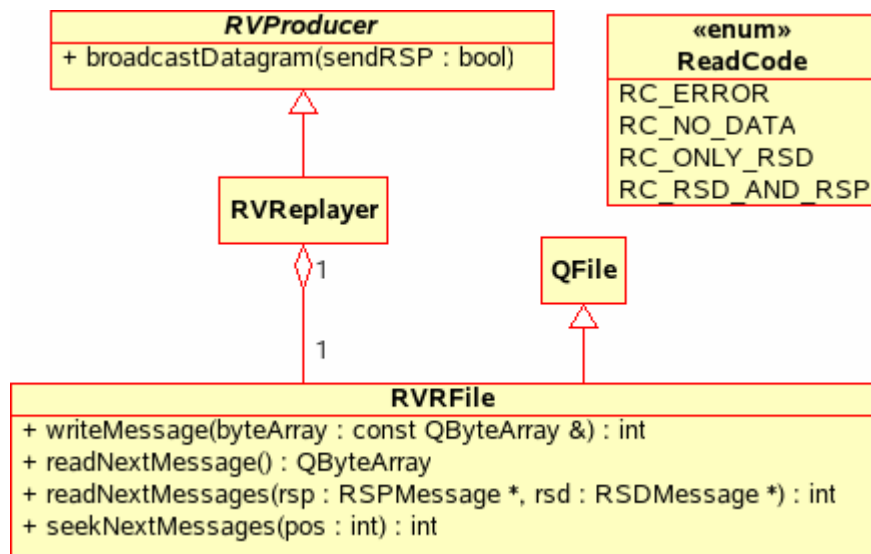
Port	41472
Source ID	1
Accept All Source IDs	☐
Radar Scan Time	1000
Azimuth Resolution	2048
Sample Count	0
Channels per Sample	0
Amplitude Resolution	0
Radar Range	18000
Azimuth Reference	0
Compression Algorithm	0
Scan Sequence Nr.	91913
Sectors Processed Last Scan	2047
Sectors Received Last Scan	2048
Auto-generate Filenames	☐
Save Location	Change Save Location
	./recording.rvr
Total Data	22.561 MB
Data per Second	2.03691 MB/s
Recording Duration	00:00:12
Stop Recording Quit	
Recording...	

Figuur 49 Screenshot radarvideo recorder

De radarvideo recorder heeft, mede dankzij de lage bandbreedte eisen van de digitale distributiearchitectuur, erg lage systeemeisen om realtime radarvideo op te nemen. Vandaar dat zelfs een zeer eenvoudige laptop met een netwerkaansluiting kan dienen als radarvideo logger. Dit systeem is reeds in de praktijk gebruikt bij mortierproeven met de Flycatcher Mk2.

8.3 Radarvideo replayer

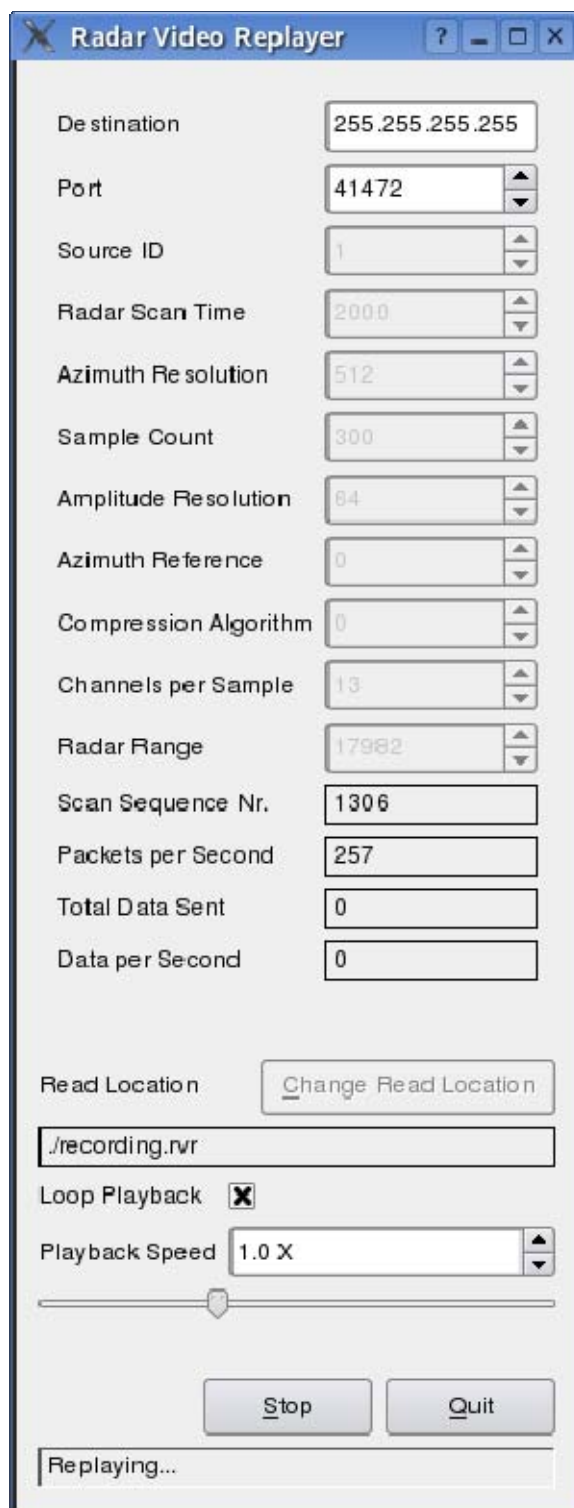
Om de opnames van radarvideo te kunnen gebruiken is een afspel mechanisme ontwikkeld. Deze *RVReplayer* is hiervoor als sub-klasse van de generieke *RVProducer* geïmplementeerd. Het maakt gebruik van dezelfde *RVRFile* klasse welke ook de interface met het logbestand verzorgt voor de recorder. Door de wijze waarop de informatie hierin is weggeschreven, is het zeer eenvoudig om hier de SRDV berichten weer uit te lezen. Begonnen wordt met het uitlezen van 4 bytes. Deze geven, geïnterpreteerd als integer getal, de grootte van het komende bericht aan. Op basis hiervan kan voldoende ruimte gereserveerd worden en kan de informatie uit het logbestand gelezen worden. Dit proces kan voor elk volgende bericht herhaald worden. Alle uitgelezen SRDV berichten worden vervolgens middels de door *RVProducer* beschikbaar gestelde functionaliteit als UDP datagrammen verstuurd.



Figuur 50 Klassendiagram radarvideo replayer

De replayer biedt, naast het standaard afspelen van de logbestanden, de mogelijkheid om het radarvideo versneld of vertraagd af te spelen. Ook is het mogelijk om een willekeurige locatie in de tijdbalk te selecteren om af te spelen. Tenslotte biedt het programma nog de mogelijkheid om het logbestand opnieuw af te spelen wanneer het einde ervan bereikt is. Een screenshot van de applicatie is te zien in Figuur 51.

Tenslotte is het interessant te vermelden dat de recorder ook gebruikt kan worden om data, afgespeeld door een replayer, weer op te nemen. Dit kan handig gebruikt worden om interessante delen uit één grote opname te knippen.



Figuur 51 Screenshot radarvideo replayer

9 Analyse van het prototype

In dit hoofdstuk worden de op verschillende aspecten behaalde resultaten van het prototype beschreven. Hierin komt de CPU belasting en het grafisch geheugengebruik aan bod. Verder wordt gekeken in hoeverre aan de wens van platform onafhankelijkheid is voldaan.

9.1 CPU belasting

Één van de belangrijkste graadmeters voor de performance van het systeem is de mate van CPU belasting. Het minimaliseren hiervan is immers een eis voor het gebruik van de RSC-GPU. Dit is belangrijk omdat er op de console waarop het programma uitgevoerd wordt, ook nog vele andere applicaties actief zijn.

Om de CPU belasting te bepalen zijn er op twee systemen benchmark tests gedaan. Hiervan zijn de specificaties te zien in Figuur 52. Systeem A is hierin de PC waarop het prototype is ontwikkeld. Systeem B heeft een configuratie welke meer lijkt op het systeem dat werkelijk in de praktijk gebruikt zal worden. De resultaten behaald op dit systeem geven dan ook een waardevoller beeld van de performance. Systeem B zal naar verwachting in de uiteindelijke configuratie nog een krachtigere grafische kaart krijgen. Het is dus zeker dat met de gemeten waarden een ondergrens voor de performance bepaald wordt.

	Systeem A	Systeem B
CPU	Pentium 4, 2.60 GHz, Hyperthreading	2x AMD Opteron 250, 2.40 GHz
GPU	NVIDIA GeForce 6600 GT	NVIDIA Quadro FX 560
Monitor(s)	Dual head 17"	Single head 20"
Bus	AGP	PCI-Express
Geheugen	1024 MB	2048 MB per CPU, 4096 totaal
Netwerkcontroller	Intel 82540EM 1000 Mbit (onboard)	Intel 82546GB 1000 Mbit (PCI)
OS	Suse 9.3	Redhat Enterprise Linux 4
Kernel	2.6.11.4-20a-smp	2.6.16.11-10-smp
Schermresolutie	1280 x 1024	1600 x 1200

Figuur 52 Teststelsysteem specificaties

Alle tests zijn, tenzij anders aangegeven, uitgevoerd met een radardisplay ter grootte van de maximale schermresolutie en met een updatelimit van 60 frames per seconde. Hierbij wordt afterglow bijgehouden en de beste filtermethode toegepast. Ook wordt het radarvideo geprojecteerd op een onderliggende kaart en wordt de scheepsoriëntatie weergegeven met enkele afstandsringen.

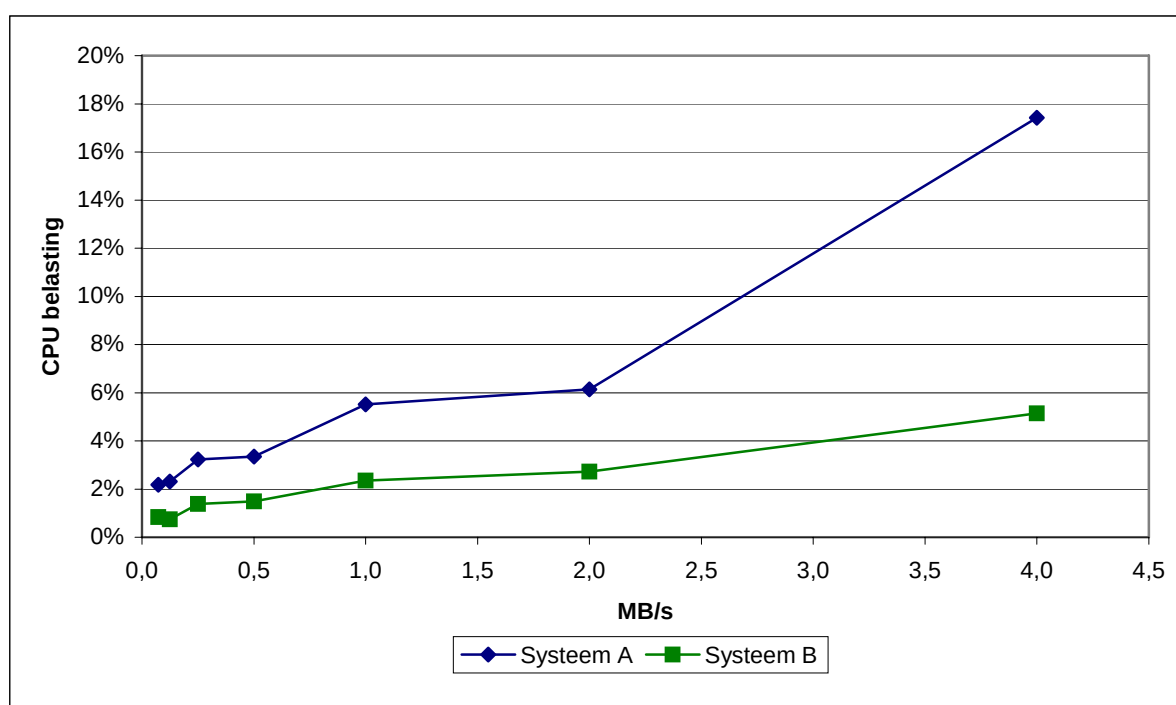
De processorbelasting is gemeten met het Linux programma *time*. Deze toepassing meet van een gegeven proces de totale tijd dat het bestaat en de tijdsduur die het in user en kernel mode actief door de CPU verwerkt wordt. Het percentage van de CPU belasting door dit proces kan dus berekend worden door de laatste twee bij elkaar opgeteld, te delen door de totale looptijd van het proces. De tests zijn allemaal over een totale tijdsduur van 10 minuten uitgevoerd. Hiermee wordt de extra hoge CPU belasting ten tijde van de initialisatie grotendeels weggemiddeld. De meetresultaten zijn te zien in Figuur 53 tot en met Figuur 55.

Azimut	Range	MB/s	User (s)	Kernel (s)	CPU
512	300	0,073	7,86	5,2	2,18%
512	512	0,125	8,13	5,75	2,31%
1024	512	0,250	10,58	8,79	3,23%
1024	1024	0,500	11,05	9,02	3,35%
2048	1024	1,000	15,85	17,26	5,52%
2048	2048	2,000	18,93	17,92	6,14%
4096	2048	4,000	70,78	33,76	17,42%

Figuur 53 CPU meetresultaten systeem A

Azimut	Range	MB/s	User (s)	Kernel (s)	CPU ¹
512	300	0,073	2,21	2,8	0,84%
512	512	0,125	2,52	1,91	0,74%
1024	512	0,250	2,96	5,37	1,39%
1024	1024	0,500	3,19	5,76	1,49%
2048	1024	1,000	4,27	9,82	2,35%
2048	2048	2,000	4,96	11,35	2,72%
4096	2048	4,000	8,67	22,22	5,15%

Figuur 54 CPU meetresultaten systeem B



Figuur 55 Overzicht van CPU belasting per verwerkte hoeveelheid MB/s

Om beter inzicht te krijgen in deze processorbelasting zijn nog een aantal tests gedaan waarbij bepaalde delen van de distributie architectuur en radarscan conversie omzeild, dan wel minder frequent uitgevoerd worden. Deze metingen zijn verricht op systeem A, gezien deze de hoogste belasting vertoont. Hierdoor is een bepaald relatief verschil met de referentie situatie als een groter absolute verschil te meten waardoor het eenvoudiger te constateren is.

¹ Dit is de belasting van één CPU. Het totale systeem kan dus een maximale belasting aan van 200%

Scenario	User (s)	Kernel (s)	CPU belasting	Relatief verschil CPU belasting
Referentie	70,78	33,76	17,42%	0%
Lokale radarvideo producent	56,74	4,42	10,19%	-41,50%
Geen <i>glTexSubImage2D</i>	24,97	32,71	9,61%	-44,82%
Lokaal en geen <i>glTexSubImage2D</i>	9	5,52	2,42%	-86,11%
Geen filtering	67,35	33	16,73%	-4,01%
FBO optimalisatie	67,2	32,72	16,65%	-4,42%
30 Hz update	49,88	32,43	13,72%	-21,26%
Panorama, 6Hz	163,17	207,09	61,71%	+254,25%
Panorama, geen transfer	70,43	33,01	17,86%	-1,04%

Figuur 56 Enkele performance analyse scenario's op systeem A

Als eerste is getest wat de invloed is van het gebruik van een lokale radarbron, ten opzichte van een radarbron in een netwerk. Hierdoor wordt de dataverwerking door de netwerkkaart vermeden, doordat de data reeds lokaal op het systeem aanwezig is. In Figuur 56 is hiermee niet alleen een afname in de user tijd te zien, maar ook een drastische afname in kernel tijd. Gekeken naar de afname van de kernel tijd bij alle andere scenario's kan geconcludeerd worden dat de dataverwerking van de netwerkkaart verreweg het grootste gedeelte van deze kernel tijd voor zijn rekening neemt. Teststelsel B heeft dus een deel van de verminderde CPU belasting ten opzichte van systeem A aan de meer geavanceerde netwerkcontroller te danken.

Ook is gekeken wat de het effect op de CPU belasting is wanneer het datatransfer door *glTexSubImage2D* achterwege gelaten wordt. Hierbij wordt nog wel dezelfde textuurgrootte gehanteerd. Er verandert dus niets aan de overige bewerkingen. Zo kan uitsluitend de invloed van de datatransfer naar de GPU gemeten worden. Uit de resultaten blijkt dat dit een groot deel uitmaakt van de gebruikte user tijd. In de metingen met systemen A en B is ook te zien dat de grootste verschillen zitten in deze user tijd. Hiermee kan geconcludeerd worden dat de datatransfer in systeem B sneller verloopt dan in systeem A. Dit is mogelijk gezien systeem B gebruik maakt van een PCI-Express bus in plaats van de AGP bus van systeem A. Daarbij is het geheugen van de Quadro met 600 MHz hoger geklokt dan de 500 MHz van de GeForce, wat een grotere geheugenbandbreedte tot gevolg heeft.

Als extra test is tegelijkertijd een lokale radarbron gebruikt en het *glTexSubImage2D* commando achterwege gelaten. Hieruit wordt duidelijk dat de verwerking van radarvideo door deze twee componenten verantwoordelijk zijn voor verreweg het grootste deel van de CPU belasting. Wanneer alle filtering van het radarvideo zoals beschreven in Hoofdstuk 4 wordt uitgeschakeld, heeft dit dan ook erg weinig invloed op de totale CPU belasting.

Een verbetering van dezelfde orde van grootte kan verkregen worden door een kleine optimalisatie in het gebruik van OpenGL. De verschillende operaties in de radardisplay, zoals afterglow en radarscan conversie, werden als losstaande onderdelen beschouwd. Hierdoor moet vóór elk onderdeel de huidige render-state goed gezet worden. Een onderdeel hiervan is het kiezen van het *Framebuffer Object (FBO)* dat gebruikt wordt om de resultaten van tekenoperaties in weg te schrijven. Deze wordt na het uitvoeren van deze operaties weer uitgeschakeld. Gezien de verwerking van afterglow en de radarscan conversie beiden dezelfde FBO instellingen gebruiken, kan het uitschakelen en opnieuw inschakelen tussen deze twee operaties achterwege gelaten worden. Deze exercitie toont aan dat er nog zeker wat winst te behalen valt door optimalisatie van het gehele proces.

Ook kan opgemerkt worden dat er ook nog een redelijke reductie in CPU belasting behaald kan worden, wanneer de display niet met 60 maar met 30 Hz getekend wordt. De belangrijkste oorzaak hiervoor is echter dat met deze vermindering in updates, de hoeveelheid aanroepen van *glTexSubImage2D* ook gehalveerd wordt. In plaats daarvan wordt er per aanroep twee maal zoveel data in één keer verstuurd. Dit laat zien dat er nog belangrijke prestatiewinst te behalen valt door deze datatransfer te optimaliseren. Hiervoor kan gekeken worden naar het gebruik van *Pixelbuffer Objects* of *PBO's* en de hiermee toe te passen technieken zoals beschreven in de Literatuurstudie GPU programmeren [1].

Tenslotte valt op dat het afbeelden van het panorama een zeer intensief proces is. Dit is echter uitsluitend merkbaar wanneer het panorama beeld met 6 Hz in zijn geheel naar de GPU gekopieerd wordt. Wanneer dit beeld slechts uitsluitend éénmaal naar de GPU gestuurd wordt tijdens de initialisatie, is het gemeten verschil te wijten aan onnauwkeurigheid in de meting. Gezien ook hier het CPU gebruik door de datatransfer veroorzaakt wordt, kan deze ook geoptimaliseerd worden door gebruik te maken van *PBO's*.

9.2 GPU belasting

De exacte GPU belasting is helaas op het Linux platform niet zo eenvoudig te meten als de CPU belasting. Om toch een indicatie van deze waarde te krijgen, is gekeken naar de maximaal haalbare hoeveelheid updates per seconde (Fps), wanneer deze op geen enkele wijze behalve door de videokaart gelimiteerd worden. De GPU belasting voor een 60 Hz display, kan dan benaderd worden door $60/Fps$.

Alle gedane metingen zijn uitgevoerd met de worst-case radar van 4096x2048 samples en met filtering over de afstand en azimut richting, tenzij anders aangegeven. Voor het panorama zijn de weergave mogelijkheden gekozen, zoals die te zien zijn in Figuur 41 en Figuur 46. Er is uitsluitend gekeken naar deze twee visualisatievormen omdat deze geheel zichtbaar zijn, zonder dat hiervoor het beeld gekanteld moet worden. Dit zal immers ook effect hebben op de performance van alle overige onderdelen, waardoor vergelijking met de overige prestaties niet meer correct is. De resultaten van de metingen zijn te zien in Figuur 57 en Figuur 58.

Resolutie	Map	Oriëntatie	Afterglow	Filtering	Panorama	Fps	GPU belasting
1280 x 1024	✓	✓	✓	✓	✗	158	37,97%
1280 x 1024	✗	✓	✓	✓	✗	196	30,61%
1280 x 1024	✓	✗	✓	✓	✗	162	37,04%
1280 x 1024	✓	✓	✗	✓	✗	161	37,27%
1280 x 1024	✓	✓	✓	✗	✗	181	33,15%
1280 x 1024	✓	✓	✓	✓	Plat	68	88,24%
1280 x 1024	✓	✓	✓	✓	Rand	106	65,60%

Figuur 57 GPU belasting van systeem A

Resolutie	Map	Oriëntatie	Afterglow	Filtering	Panorama	Fps	GPU belasting
1280 x 1024	✓	✓	✓	✓	✗	150	40,00%
1280 x 1024	✗	✓	✓	✓	✗	180	33,33%
1280 x 1024	✓	✗	✓	✓	✗	152	39,47%
1280 x 1024	✓	✓	✗	✓	✗	166	36,14%
1280 x 1024	✓	✓	✓	✗	✗	182	32,97%
1280 x 1024	✓	✓	✓	✓	Plat	59	101,69%
1280 x 1024	✓	✓	✓	✓	Rand	88	68,18%
1600 x 1200	✓	✓	✓	✓	✗	105	57,14%
1600 x 1200	✗	✓	✓	✓	✗	151	39,74%
1600 x 1200	✓	✗	✓	✓	✗	106	56,60%
1600 x 1200	✓	✓	✗	✓	✗	108	55,56%
1600 x 1200	✓	✓	✓	✗	✗	123	48,78%
1600 x 1200	✓	✓	✓	✓	Plat	50	120%
1600 x 1200	✓	✓	✓	✓	Rand	63	95,24%

Figuur 58 GPU belasting van systeem B

Uit de metingen valt op dat weergave van het panorama verreweg de meest kostbare operatie is. In de eerste plaats speelt de grootte van de panorama texture hierin een grote rol. Wanneer hiervoor een kleine texture gebruikt wordt, is een flinke toename van de updates per seconde te constateren. Opvallend is wel het grote verschil tussen de platte visualisatie en de visualisatie langs de rand van het scherm. Deze wordt hoofdzakelijk veroorzaakt doordat het panorama langs de rand vóór alle andere objecten gepositioneerd wordt. Dit, in combinatie met Z-culling, zorgt er voor dat er door alle andere tekenoperaties minder werk verricht hoeft te worden. Daarnaast is de geometrie, gebruikt voor de randprojectie, vele malen minder complex dan die van de platte projectie.

Naast het panorama is ook de weergave van de map een vrij kostbare operaties. Ook hier is gebleken dat de texture unit de grootste bottleneck vormt. Het gebruik van een kleinere texture hiervoor laat ook een flinke toename van de updates per seconde zien.

9.3 GPU Geheugengebruik

Naast de processorkracht is ook het grafische geheugen een beperkte resource waar zorgvuldig mee omgesprongen dient te worden. Het is helaas niet mogelijk om het exacte geheugengebruik te bepalen. In Figuur 59 wordt voor een benadering hiervan een optelling gedaan van het worst-case geheugengebruik van alle relevante componenten van het prototype.

	Breedte	Hoogte	Bpp ¹	Aantal	Totaal (MB)
RSC					
Ruw Radarvideo	4096	2048	1	1 ²	8
Afstand genormaliseerd radarvideo	4096	2048	4 ³	1 ⁴	32
Radar overlay	1600	1200	4 ⁵	2 ⁴	14,648
Subtotaal					54,648
Optioneel					
Kleurentabel	255	1	4 ⁵	3 ⁶	0,003
Map	1600	1200	4 ⁵	1	7,324
Panorama	5490	480	4 ⁵	1	10,052
Subtotaal					17,379
Framebuffer					
Kleurbuffer	1600	1200	4	2 ⁷	14,648
Z-Buffer	1600	1200	3	1	5,493
Stencil buffer	1600	1200	1	1	1,831
Subtotaal					21,972
Totaal					94,001

Figuur 59 Theoretisch worst-case volledig GPU geheugengebruik prototype

Zoals vermeld zijn dit de absolute worst-case waarden. Wanneer bijvoorbeeld uitgegaan wordt van de resolutie van de Flycatcher Mk2 kan er van dit totaal al 39,268 MB afgetrokken worden. Ook is het panorama op dit moment nog slechts een punt van onderzoek. Het zal nog enige tijd duren voordat deze waarde werkelijk relevant wordt.

Verder moet er rekening gehouden worden met het feit dat ook de huidige oplossing beslag legt op een redelijke hoeveelheid geheugen. Zo wordt het radarvideo ook al in een texture opgeslagen door de RSC-TB. Daarbij wordt er ook in het huidige systeem al een map getoond, deze kosten worden dus ook niet extra toegevoegd door gebruik van de RSC-GPU. Tenslotte zijn de laatste drie waarden voor het framebuffer ook altijd nodig wanneer er van een OpenGL display gebruik wordt gemaakt. Ter illustratie geeft Figuur 60 een overzicht van het toegevoegde geheugengebruik ten opzichte van het gebruik van de RSC-TB.

Hierin is te zien dat de maximale configuratie van radarvideo textures, ondersteund door de RSC-TB, gebruikt wordt. Verder is uitgegaan van een display configuratie met daarin één schermvullend radarscherm, daaroverheen een drietal kleinere schermen met bijvoorbeeld een overzicht en enkele detail afbeeldingen. Hierbij zal dus toch nog een groot gedeelte van het schermvullende deel of de kleinere radarvideo weergaven voor niets verwerkt worden door de overlap van deze delen. Het hiervoor gebruikte geheugen kan dus feitelijk weggelaten worden wanneer puur gekeken wordt naar zinvol geheugengebruik.

¹ Bytes per pixel

² Per radarbron

³ Feitelijk slechts 1 kleurkanaal nodig. Enkel kanaal texture helaas niet schrijfbaar met Framebuffer Object

⁴ Per scherm met radarvideo visualisatie

⁵ Feitelijk slechts 3 kleurkanalen nodig. Effectief geheugengebruik lijkt echter gebaseerd op 4 kanalen.

⁶ Willekeurige hoeveelheid te kiezen. Kan gedeeld worden door alle visualisaties

⁷ Twee buffers vanwege double buffering

	Breedte	Hoogte	Bpp	Aantal	Totaal (MB)
RSC					
Ruw Radarvideo	1024	2048	1	1	2
Afstand genormaliseerd radarvideo	1024	2048	4	1	8
Radarvideo RSC-TB	1024	2048	4	1	-8
Ruw Radarvideo	1024	1024	1	1	1
Afstand genormaliseerd radarvideo	1024	1024	4	1	4
Radarvideo RSC-TB	1024	1024	4	1	-4
Ruw Radarvideo	512	512	1	2	0,5
Afstand genormaliseerd radarvideo	512	512	4	2	2
Radarvideo RSC-TB	512	512	4	2	-2
Subtotaal					3,5
Radar overlay	1600	1200	4	2	14,648
Radar overlay	512	512	4	6	6
Subtotaal					20,648
Totaal					24,148

Figuur 60 Worst-case toegevoegd GPU geheugengebruik door RSC-GPU t.o.v RSC-TB

De grafische kaarten in de (hoge) middenklasse, welke gebruikt gaan worden voor toepassing van de RSC-GPU, worden tegenwoordig niet met minder dan 256 MB geproduceerd. Er wordt bijvoorbeeld gedacht aan de NVIDIA Quadro FX 1500. De verwachting is dan ook dat dit geheugengebruik niet een bottleneck zal vormen. Mocht dit echter wel het geval zijn, bestaat er nog wel enige ruimte in het optimaliseren van dit geheugengebruik.

Als eerste kan er gekeken worden naar een methode om toch de texture met over de afstand genormaliseerd radarvideo weg te schrijven in een texture met één kanaal. Hierdoor zal slechts een kwart van het geheugengebruik overblijven. Dit zal, zeker in het geval uitgegaan wordt van een worst-case scenario, een grote afname in het absolute geheugengebruik betekenen. Ook hoeft deze texture niet de dimensies van het oorspronkelijke radarvideo aan te houden. Wanneer de informatie in de texture wordt samengenomen, kunnen de texture dimensies uiteraard ook proportioneel mee krimpen.

Ook kan er nog winst behaald worden in het gebruik van de radar overlays. Zoals in Hoofdstuk 7.2.1.2 is beschreven, worden er op dit moment 3 kanalen van deze texture gebruikt. Door het interne gebruik van texturegeheugen in de GPU, wordt hiervoor echter effectief geheugen voor een 4 kanaal texture in gebruik genomen. Daarbij zijn er theoretisch slechts 2 kanalen noodzakelijk, namelijk die van de huidige radarvideo intensiteit en de intensiteit van de afterglow. Wanneer er daadwerkelijk slechts 2 kanalen gebruikt worden, zal het geheugengebruik voor de beide radar overlays dan ook halveren.

9.4 Platform onafhankelijkheid

Zoals vermeldt in Hoofdstuk 1.2 is Linux het beoogde platform. Platform onafhankelijkheid was daarmee ook niet direct een eis. Om hier toch in tegemoet te komen is er voor het gehele prototype zo veel mogelijk gebruik gemaakt van de Qt toolkit. Hiermee zijn alle onderdelen dan ook erg gemakkelijk te porten naar andere platformen, zoals Windows, waarop deze toolkit geïmplementeerd is. Hiervoor hoeft de code uitsluitend opnieuw gecompileerd te worden.

Voor correcte implementatie van het prototype moest helaas één uitzondering gemaakt worden op het gebruik van de Qt toolkit, namelijk voor hoogfrequente, nauwkeurige timing. De Qt toolkit biedt een kleinst mogelijke granulariteit van 1 ms. Dit bleek echter onvoldoende voor bepaalde toepassingen in het prototype. Wanneer bijvoorbeeld een radar gesimuleerd wordt met 4096 sectoren en een omwentelingstijd van 2 seconden, moet er immers elke $2000/4096 = 0.488$ ms een sector worden verstuurd. Hiervoor zijn functies zoals *usleep* en *gettimeofday*, welke standaard in Linux beschikbaar zijn, gebruikt om de nodige grotere nauwkeurigheid te behalen. Bij het porten naar een ander platform moet het gebruik van deze functies vervangen worden door de voor dat platform beschikbare equivalente functies.

De radarvideo recorder maakt echter geen gebruik van deze functionaliteit. Deze is dan ook succesvol geport naar Windows. De Windows versie is zelfs gebruikt om de radarvideo opnames bij de mortierproeven te maken.

10 Integratie in TAC-Server

Om de in dit verslag beschreven RSC als product te kunnen gebruiken moet het geïntegreerd worden in Tactics, het door Thales gebruikte Combat Management Systeem (CMS). Het grootste gedeelte hiervan zal plaats vinden in de TAC-Server, een component hiervan waar op dit moment ook de door de RSC-TB gegenereerde radarvideo texture wordt afgebeeld. Gedurende het onderzoek was de Linux versie hiervan echter nog niet beschikbaar. Deze werd op dat moment nog geport van het Solaris platform. Wel is er kort gekeken naar de verwachte impact die de integratie van de in dit verslag vermelde technieken zal hebben. De verschillende aspecten die in deze analyse naar voren zijn gekomen, worden in dit hoofdstuk beschreven.

10.1 Integratie van digitaal radarvideo gedistribueerd met SR(D)V

Het software gedeelte van de RSC-TB en de TAC-Server communiceren met elkaar via shared memory. In dit stuk geheugen wordt door de RSC-TB het radarvideo verwerkt volgens de instellingen van de operator. De TAC-Server slaat de informatie in dit stuk geheugen op in een texture die het binnen zijn OpenGL context kan gebruiken en op het scherm af kan beelden.

De integratie digitaal radarvideo is dus het eenvoudigste te integreren door ook gebruik te maken van dit shared memory mechanisme. De ontvanger van het radarvideo kan dit dan ook op vrijwel gelijke wijze verwerken als de *RVManager*, zoals deze gebruikt is in het prototype. In Figuur 37 op bladzijde 65 is te zien hoe dit gerealiseerd kan worden door een sub-klasse te maken van de *RVConsumer*.

Het regelrecht kopiëren van de radarvideo in shared memory, is de enige taak die hierbij gedaan hoeft te worden. Verdere verwerking van de radarvideo gebeurt in de RSC-GPU immers niet meer op de CPU maar wordt in de GPU uitgevoerd.

10.2 Integratie van de op GPU gebaseerde RSC functionaliteit

De tweede stap die genomen moet worden is integratie van de op de GPU gebaseerde functionaliteit. Hiertoe zal in de eerste plaats de mogelijkheid van het gebruik van GPU programma's gecreëerd moeten worden. Wanneer gebruik wordt gemaakt van de Cg toolkit, zoals dit ook gedaan is in het prototype, zullen hiervoor enkele onderdelen geïnitieerd moeten worden. Hiervoor moet een grafische context beschikbaar zijn en kunnen daarom het beste worden uitgevoerd vlak na de initialisatie van de OpenGL context zelf. Het is echter ook mogelijk om de Cg programma's zonder Cg toolkit te gebruiken [1], of de programma's om te schrijven naar GLSL. In beide gevallen zal uitsluitend van OpenGL functionaliteit gebruik worden gemaakt. Hiervoor moeten echter nog steeds verscheidene componenten, dit maal van OpenGL, geïnitieerd worden om GPU programma's te kunnen gebruiken.

Voor het toepassen van alle functionaliteit dient de renderloop van de TAC-Server aangepast te worden. Oude code, welke oorspronkelijk uitsluitend de radarvideo in een texture laad en deze middels standaard texture mapping op het scherm afbeeldt, moet vervangen worden door de gewenste onderdelen van de renderloop gebruik in de *RadarDisplayWidget*.

11 Conclusies

In Hoofdstuk 1.1 zijn de volgende doelstellingen voor het onderzoek geformuleerd:

- ❑ Ontwikkeling van een filter dat dataverlies rond het middelpunt van een PPI tegen gaat en implementatie hiervan op de GPU.
- ❑ Doen van een onderzoek naar de haalbaarheid van radarscan conversie dat volledig op de GPU wordt uitgevoerd.
- ❑ Implementeren van een prototype GPU radarscan convertor, gebruik makende van een digitale radarvideo distributie.
- ❑ Integratie van gelogd radarvideo in het prototype
- ❑ Implementeren van sensorfusie van panoramisch TV en radarvideo teneinde nieuwe mogelijkheden voor productinnovatie te illustreren.

Uit het onderzoek is gebleken dat de programmeerbaarheid van de GPU ingezet kan worden om de beeldkwaliteit bij radarscan conversie te verbeteren. Van de hiervoor ontwikkelde bloemvormige filtermethode is in Hoofdstuk 4.2.1.3 bewezen dat er geen radarsamples meer verloren kunnen gaan, terwijl een minimale hoeveelheid radarvideo gedupliceerd wordt weergegeven.

Verder kan geconcludeerd worden dat er een aanzienlijke risicoreductie is gedaan voor het gebruik van de GPU als vervanging voor de huidige hardwarematige oplossing voor radarscan conversie, de RVG. Met het gemaakte prototype, beschreven in Hoofdstuk 7, is aangetoond dat met GPU programmeren de functionaliteit, geboden door de RVG, geëvenaard en zelfs overtroffen kan worden. Uit Hoofdstuk 9.1 kan geconcludeerd worden dat dit mogelijk is zonder dat hierbij de CPU in hoge mate wordt belast. Daarbij biedt het gebruik van de GPU alle voordelen van softwarematige radarscan conversie. Voorbeelden hiervan zijn de reductie in de productiekosten van een console en de gelegenheid om eenvoudig de functionaliteit ervan aan te passen. Deze aanpasbaarheid in combinatie met het zeer hoge tempo waarmee GPUs ontwikkeld en verbeterd worden, biedt tevens ongekende mogelijkheden voor continue productinnovaties. Dit is aangetoond met de gegeven demonstratie van sensorfusie met panoramisch TV video, wat beschreven is in Hoofdstuk 7.2.4.

Tevens is uit het onderzoek gebleken dat het gebruik van een digitale distributie architectuur verscheidene voordelen met zich meebrengt. Naast de reeds bekende voordelen zoals het kostenefficiënt aan te leggen distributienetwerk, is aangetoond dat ook het loggen en afspelen van digitaal radarvideo zeer eenvoudig is te realiseren. Deze conclusie kan getrokken worden uit het feit dat radarvideo, afkomstig van de digitale distributiearchitectuur van de Flycatcher Mk2, succesvol gelogd is met behulp van een notebook, zoals vermeld in Hoofdstuk 8.2. Hiervoor is bij de analoge distributie architectuur onhandzame en vooral kostbare apparatuur nodig.

Tenslotte kan op basis van de korte analyse beschreven in Hoofdstuk 10 geconcludeerd worden, dat integratie van de gepresenteerde RSC-GPU functionaliteit in de TAC-Server geen problemen met zich mee lijkt te brengen. Deze functionaliteit is bovendien erg modulair, waardoor gekozen kan worden voor een gefaseerd integratieproces om de risico's minimaal te houden.

12 Aanbevelingen

Op basis van het onderzoek en de daaruit voortgekomen conclusies is het gebruik van de GPU voor radarscan conversie zonder meer aan te bevelen. In de eerste plaats zal hiervoor de in Hoofdstuk 10 gegeven integratieanalyse in meer detail moeten worden uitgevoerd. De resultaten en bevindingen kunnen direct in de praktijk gebracht worden door het doen van de daadwerkelijke integratie van de radarscan conversie functionaliteit in de TAC-Server. Op dit moment is er niets meer dat uitvoering hiervan in de weg staat. Het is dan ook aanbevelingswaardig deze integratie op korte termijn uit te voeren, zodat onvoorziene problemen snel aan het licht komen en opgelost kunnen worden.

Koppeling van de TAC-Server aan de digitale distributie architectuur kan indien gewenst zelfs parallel aan de RSC-GPU integratie uitgevoerd worden. Dit doordat beide onderdelen feitelijk los van elkaar staan. De herkomst van radarvideo zal immers geen invloed hebben op de verwerking van de RSC-GPU. Evenzo maakt het voor het proces, dat radarvideo ontvangt en beschikbaar stelt voor de RSC, niet uit wat er vervolgens met de data gedaan wordt. Wel moet er rekening gehouden worden met het feit dat belangrijke onderdelen van de digitale distributie, met name het te gebruiken SRV protocol, nog ontwikkeld moeten worden. Om deze reden wordt aanbevolen de integratie initieel uit te voeren met het in het prototype gebruikte SRDV protocol. Wanneer de specificatie van het SRV protocol gereed is, kan vrij eenvoudig het protocolspecifieke deel van de implementatie vervangen worden. Deze aanpak heeft als voordeel dat de TAC-Server tevens een SRDV interface heeft gekregen. Dit kan van pas komen bij bijvoorbeeld een upgrade van de RSC functionaliteit van de Flycatcher.

Nadat de integratie van de RSC-GPU voltooid en daarmee weer een risico gereduceerd is, wordt aanbevolen nader onderzoek te doen naar de in het verslag genoemde aandachtspunten. Zo moet bepaald worden of het gebruik van radarsamples van de vorige radarscan, zoals beschreven in Hoofdstuk 4.2.1.4, een probleem vormt. Ook moet een keuze gemaakt worden tussen de in Hoofdstuk 7.2.1.4 voorgestelde methoden om met het probleem van de combinatie relative motion met afterglow om te gaan. Verder bestaat de mogelijkheid dat het geheugengebruik op de grafische kaart te groot is. Hiervoor kunnen de suggesties gedaan in Hoofdstuk 9.3 verder uitgewerkt en in de praktijk gebracht worden.

Tenslotte wordt het sterk aanbevolen om te blijven investeren in het bedenken van nieuwe concepten voor visualisatie van sensordata. In dit onderzoek is in het kader hiervan een voorbeeld gegeven van hoe sensorfusie van radarvideo met panoramisch TV video eruit zou kunnen zien. Hiermee is al bewezen dat de rekenkracht van de hedendaagse GPUs al in staat is om dergelijke nieuwe visualisatievormen te kunnen weergeven. Daarnaast biedt de zeer snelle ontwikkeling van GPUs goede vooruitzichten voor realisatie van toekomstige revolutionaire ideeën.

Referenties

- [1] **Literatuurstudie - GPU Programmeren**, *Riccardo Sirchia*, Universiteit Twente in opdracht van Thales Nederland B.V., 17-11-2006
- [2] **PFS: RSC-TB**, 12NC: 9501 027 76603, Thales Nederland B.V., 27-01-2005
- [3] **IRS: Radar Video**, 12NC: 9501 017 45000, Thales Nederland B.V., 24-02-1992
- [4] **IRS: Surveillance Radar Display Video on the SDB**, 12NC: 9501 025 80101, Thales Nederland B.V., 28-11-2003
- [5] **IRS: Video Data Format**, 12NC: 9501 303 36801, Thales Nederland B.V., 03-12-2004
- [6] **Onderzoek naar kosteneffectieve realtime video distributie ten behoeve van het Naval Combat System** - *Harm de Boer*, in opdracht van Thales Nederland B.V., 13 Juni 2005
- [7] **Surround Vision – Image processing met OpenGL en Cg**, *Robin van Kaathoven, Arjan Somers*, juni 2005
- [8] **Design Patterns – Elements of reusable Object-Oriented software**, *Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides*, 1995, Addison-Wesley, 26e druk, mei 2003
- [9] **Software Engineering – A practitioner's approach**, *Roger S. Pressman*, McGraw-Hill, 5th edition
- [10] **GPU Gems 2**, *Edited by Matt Phar, Randima Fernando*, 2005, Addison-Wesley, 1e druk, maart 2005
- [11] **Leren Communiceren**, *Michaël Steehouder*, Wolters-Noordhoff, mei 1998

Begrippenlijst

A

Anti-aliasing

Een term om de techniek aan te duiden, die de effecten die kunnen ontstaan wanneer hoge resolutie data wordt omgezet naar een lagere resolutie.

A-Scope

De benaming voor een weergave van een *radarsweep* of *radarsector* waarin de amplitude van het *radarvideo* is afgezet tegen de afstand tot de radar.

Afterglow

Deze term is afkomstig van het gebruik van fosfor beeldschermen. Hiermee wordt het nagloeien van de fosfordeeltjes bedoeld, wat resulteert in het (deels) zichtbaar blijven van de geschiedenis van een *radarsample*. In dit verslag wordt hiermee kunstmatige nabootsing van dit effect bedoeld.

Azimut

Een benaming voor de koers van bijvoorbeeld een vaartuig of de richting waar een radar naar wijst. Deze wordt gegeven in graden te opzichte van het noorden waarbij de graden met de klok mee van 0 tot 360 lopen.

B

B-Scope

De benaming voor een weergave van een *radarscan* waarin de afstand tegen *azimut* is afgezet. De punten in dit coördinatenstelsel hebben als derde dimensie de amplitude van het radarsignaal, bijvoorbeeld weergegeven als intensiteit van de kleur van een punt.

Back-face culling

Deze techniek maakt standaard deel uit van OpenGL en zorgt ervoor dat geometrie, waarvan uitsluitend de achterzijde zichtbaar is, uit het beeld weggelaten wordt.

BTS

Staat voor Bus Tap Server. Een algemene benaming gebruikt binnen Thales om bijvoorbeeld een data conversie systeem mee aan te duiden.

C

CATV

Staat voor Community Antenna Television. Het CATV netwerk heeft als doel TV kanalen te verspreiden.

CMS

Staat voor Combat Management Systeem. Hierin wordt informatie over de huidige (gevecht)situatie weergegeven en het kan een interface bieden om troepen of materieel in te zetten om deze situatie te beïnvloeden.

Clutter

Dit is de benaming voor reflecties van de omgeving in radarvideo. Dit kunnen bijvoorbeeld bomen, gebouwen of golven zijn. Deze reflecties zijn met name merkbaar op relatief korte afstanden van de radar.

Composiet radarvideo

Radarvideo dat naast de radarreflecties ook andere informatie zoals *azimut* en een synchronisatie, noord reset en scheepskoers puls bevat.

Conditionele expressie

Een expressie die er als volgt uit kan zien: (*expressie* ? *expressie* : *expressie*).

COTS

Staat voor Commercial Of The Shelf. Een aanduiding voor alle apparatuur die commercieel te koop wordt aangeboden en niet (zelf) ontwikkeld hoeft te worden.

D

DMA

Staat voor Direct Memory Access. Een techniek welke het mogelijk maakt om randapparatuur, zoals in dit geval de videokaart, data van en naar het geheugen te kopiëren zonder directe sturing van de CPU.

Doppler verschuiving

De verandering in frequentie van een golf die waar te nemen is wanneer de verzender en ontvanger naar elkaar toe of van elkaar af bewegen.

E

Edge

De benaming voor een lijn tussen twee *vertices* in 3D-Graphics.

F

FBO

Zie *Framebuffer object*.

FPGA

Staat voor Field Programmable Gate Array. Dit is een chip waarop de logische configuratie van elektronische componenten programmeerbaar is.

Framebuffer object

Een extensie op *OpenGL*, o.a. te gebruiken om de uitvoer van een tekenoperatie rechtstreeks in een texture te plaatsten.

G

GPU

Graphical Processing Unit, de processor van een *videokaart*, of gebruikt om te verwijzen naar de *videokaart* in zijn geheel.

I

If-Statement

Een statement die er als volgt uit kan zien: **if** (*expressie*) **then** *statement* **else** *statement*

IFF

Staat voor Identification Friend or Foe. Dit is een code die wordt uitgezonden door vliegtuigen, schepen, etc. zodat er kan worden bepaald of een object een vriend of vijand is.

IRS

Staat voor Interface Requirements Specification. Dit is de benaming voor een type document dat binnen Thales Nederland B.V. gehanteerd wordt.

L

LIN

Is een afkorting voor LINear, hiermee wordt het gewone *radarvideo* bedoelt.

Luistertijd

De tijd die een radar wacht op echo's vlak na het uitzenden van een radarpuls.

M

MTI

Staat voor Moving Target Indication, dit is een vorm van *radarvideo* waarop uitsluitend bewegende objecten zichtbaar zijn.

O

Off-centering

Term die gebruikt wordt voor weergave van een *PPI* waarbij het centrum van het beeld ongelijk is aan het centrum van de cirkel.

P

PBO

Zie *Pixelbuffer Object*.

PFS

Staat voor Product Function Specification. Dit is de benaming voor een type document dat binnen Thales Nederland B.V. gehanteerd wordt.

Pixelbuffer Object

Een extensie op *OpenGL*, o.a. te gebruiken om transfer van pixel data in textures middels *DMA* te laten plaatsvinden.

PPI

Staat voor Plan Position Indicator. Dit is de benaming voor het radarbeeld waarin informatie over 360 graden in een cirkel wordt afgebeeld. Het midden van de cirkel is daarin de locatie van de radar en de afstand tot de radar wordt groter naarmate een object meer naar de rand van de cirkel afgebeeld wordt.

PRF

Staat voor Pulse Repetition Frequency. Dit is de herhaalfrequentie waarmee radiopulsen worden uitgezonden en is gelijk aan $1/PRT$.

PRT

Staat voor Pulse Repetition Time. Dit is de totale tijdsduur van het versturen en wachten op antwoord van één radiopuls.

Q

Quad

Een begrip uit de 3D-Graphics, wordt gebruikt om een vierhoek aan te duiden.

Quad-strip

Een serie aaneengesloten *quads* welke steeds twee *vertices* en daarmee dus een *edge* delen.

R**RADAR**

Staat voor RAdio Detection And Ranging. Een manier van visualisatie van objecten op (grote) afstand door het versturen van radiogolven en het opvangen van de reflecties hiervan.

Radar overlay

De benaming voor de in het prototype gebruikte afbeelding waarin een geconverteerde *radarscan* en bijbehorende *afterglow*

Radarsample

Een gekwantificeerd punt in *radarvideo* met een afstand, amplitude en *azimut*.

Radarsector

De 360 graden om een radar is verdeeld in sectoren van gelijke grootte. *Radarvideo* in *radarsweeps* wordt samengevoegd en waar nodig geïnterpoleerd om de sectoren van informatie te voorzien.

Radarsweep

Radarvideo afkomstig uit één enkele richting. Dit bestaat dus uit alle *radarsamples* met dezelfde *azimut*. Een complete *radarscan* bestaat uit een (grote) hoeveelheid van deze lijnen informatie. Zie ook *Radarsector*.

Radarscan

Alle *radarvideo* over 360 graden.

Radarscan conversie

Het omzetten van polair georiënteerd *radarvideo* naar een cartesisch georiënteerd vorm zoals in een *PPI*.

Radarvideo

De intensiteit van ontvangen weerkaatsingen van (verzonden) radargolven.

Radarvideo Stream

De benaming voor een videokanaal, waaronder bijvoorbeeld *LIN*, *MTI* of *SEC* video, welke van de radar afkomstig zijn.

RGBA

Staat voor Red Green Blue Alpha. Dit zijn de meest gebruikte kleurkanalen in computergraphics. De mix van rood groen en blauw bepalen de uiteindelijke kleur, en het alpha kanaal kan gebruikt worden om de mate van transparantie te bepalen.

RSC

Staat voor Radar Scan Convertor. Dit is een apparaat of programma dat *radarvideo* in de *sectoren* van een *radarscan* omzet naar een *PPI* weergave.

RSC-TB

Staat voor Radar Scan Convertor – Texture Based. Dit is de tot op heden meest recent ontwikkelde *RSC* van Thales Nederland B.V.

RSC-GPU

Staat voor Radar Scan Convertor – Graphical Processing Unit. Dit is de (voorlopige) naam voor de in dit verslag behandelde *RSC*.

RSD

Staat voor Radar Source Data. Dit is een bericht, gebruikt in het *SRDV protocol*, welke eens per *radarsector* verstuurd wordt. Naast de radarsweep zelf bevat het een minimale hoeveelheid aan info over die sweep.

RSP

Staat voor Radar Source Parameter. Dit is een bericht, gebruikt in het *SRDV protocol*, welke eenmaal per *radarscan* verstuurd wordt. Hierin worden de karakteristieken van het radarvideo gespecificeerd.

Relative Motion

Een manier van afbeelden van een *PPI* waarbij de locatie van de radar centraal staat. Wanneer de *PPI* geprojecteerd wordt op een (land/zee)kaart, lijkt deze kaart dus onder de *PPI* door te bewegen.

RVG

Staat voor Radar Video Grabber. Is de huidige hardwarematige oplossing voor het verwerken van ruwe *radarvideo* naar een *PPI* weergave.

S

SEC

Is een afkorting voor SECundair, hiermee wordt *radarvideo* bedoeld dat secundaire informatie zoals *IFF* bevat.

Sweep

Zie *radarsweep*.

Sector

Zie *radarsector*.

SIMD

Staat voor Single Instruction Multiple Data. Dit is een dataverwerkingsmodel waarbij een programma iteratief één enkele instructie uitvoert op elementen van een dataset.

SRDV protocol

SRDV staat voor Surveillance Radar Display Video. Dit protocol wordt gebruikt voor communicatie tussen de verschillende Flycatcher shelters.

SRV protocol

SRV staat voor Streaming Radar Video. Dit is de (voorlopige) naam gegeven aan het van *SRDV* afgeleide protocol voor digitale radarvideo distributie.

Supersampling

De methodiek welke ten grondslag ligt aan alle *anti-aliasing* technieken. Hiermee wordt bedoeld dat er voor de weergave van één pixel meerdere samples genomen worden om het resultaat te bepalen.

T

Track

Een door een sensor (bijvoorbeeld radar) gedetecteerd object dat vaak met een simpel icoon wordt weergegeven op het scherm.

Triangle

De benaming voor driehoeken waarin alle 3D geometrie opgedeeld kan worden.

Triangle-fan

Een manier van doorgeven van *vertices* waarbij driehoeken gevormd worden welke in een waaivorm tegen elkaar aan liggen.

True Motion

Een manier van afbeelden van een *PPI* waarbij een gekozen punt in de wereld centraal staat. Wanneer de *PPI* geprojecteerd wordt op een (land/zee)kaart, blijft deze kaart dus stil staan en beweegt de *PPI* met de locatie van de radar als middelpunt over deze kaart.

V**Vertex**

De benaming van een driedimensionaal punt voor een GPU.

Videostreams

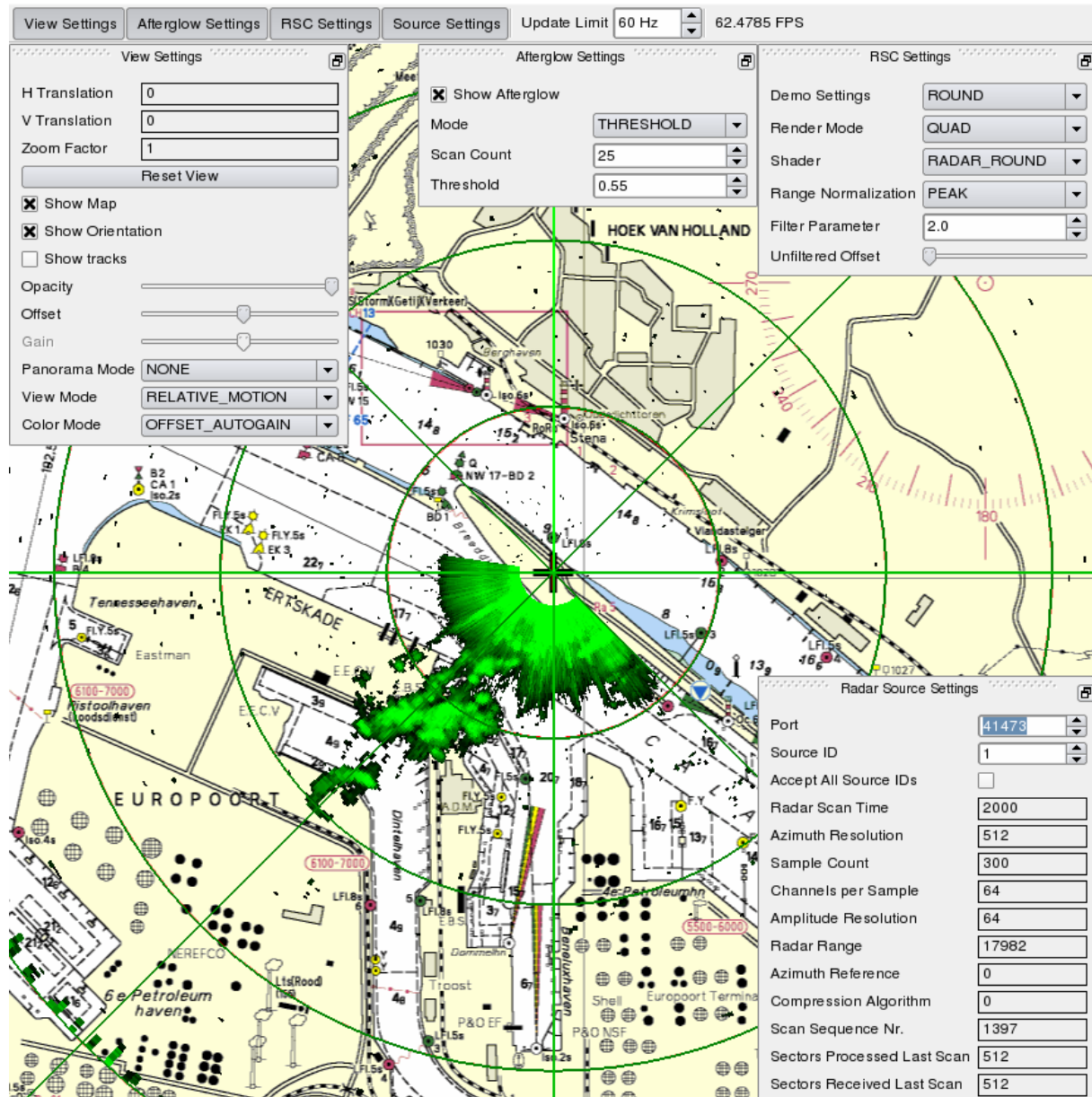
Zie radarvideo streams.

VIU-R

Staat voor Video Interface Unit – Radar. Dit is de huidige benaming voor het stuk hardware, dat momenteel nog in ontwikkeling is, dat de *radarvideo* en overige data van een *radar* omzet naar het *SRV protocol*.

Bijlage A Screenshot van het prototype

Hieronder wordt een screenshots getoond van het prototype dat in dit onderzoek ontwikkeld is. Het radarvideo is afkomstig van opnames, gemaakt met de zelf ontwikkelde radarvideo recorder, van de Flycatcher Mk2. Deze opnames zijn gemaakt op 28-03-2006. De Flycatcher Mk2 was toen aanwezig bij enkele mortierproeven, waarbij de mogelijkheden van de radar werden getest. Hierbij werd helaas niet over de volle 360 graden radarvideo geproduceerd. Vandaar dat er slechts reflecties te zien zijn in een segment van zuidoost tot west.



Figuur 61 Applicatie met GUI elementen

Literatuurstudie

GPU Programmeren



Student	Riccardo Sirchia
Nummer	0011177
Datum	17-11-2006
Commissieleden Universiteit Twente	Dr. J. Zwiers Dr. M. Poel Ing. H. Hondorp
Commissieleden Thales Nederland BV	Ir. R. Boers Ing. S. van Os

Samenvatting

De architectuur van GPUs heeft grote ontwikkelingen doorgemaakt. Onder meer door het zeer parallelle karakter is de verwerkingssnelheid van de GPU vele malen groter dan die van de CPU. Huidige generaties hebben vertex en fragment processors, die programmeerbaar zijn als respectievelijk MIMD en SIMD stream processors. Hierdoor zijn GPUs te gebruiken als zeer snelle, generieke coprocessors. Naast veelgebruikte programmeertalen als Cg en GLSL is de GPU ook te programmeren zonder enige kennis van de architectuur of grafische API met de programmeertaal Sh. Deze kennis is echter wel vereist wanneer optimale performance noodzakelijk is. Momenteel groeit de verwerkingssnelheid van de GPU drie keer sneller dan die van de CPU. Ook ligt een uitbreiding van de programmeerbaarheid al vast in de specificaties van de volgende generatie GPUs. De mogelijkheden van GPUs worden nu al succesvol gebruikt in diverse toepassingen, en dit succes zal met de genoemde ontwikkelingen alleen nog maar toenemen.

Inhoudsopgave

Samenvatting	2
Inhoudsopgave	3
Lijst van Figuren	5
Lijst van Tabellen.....	7
Lijst van Code fragmenten	8
1 Inleiding	9
2 Doelstellingen en begrenzingen	10
2.1 Doelstelling	10
2.2 Onderzoeksvragen	10
2.3 Begrenzingen	10
3 GPU architectuur	11
3.1 Evolutie van de architectuur	11
3.2 Pipeline architectuur	12
3.3 Mogelijkheden en beperkingen	14
3.3.1 De geboden mogelijkheden en voordelen.....	14
3.3.2 De beperkingen en nadelen	16
4 GPU programmeertalen	19
4.1 Categorisatie van GPU programmeertalen.....	19
4.2 Derde generatie programmeertalen (3GL)	20
4.2.1 Cg	20
4.2.2 HLSL	23
4.2.3 GLSL	23
4.3 Vierde generatie programmeertalen (4GL)	25
4.3.1 Sh	25
4.3.2 Brook for GPU	27
4.4 Overwegingen bij programmeertaal keuzes.....	28
4.4.1 Keuze voor een derde generatie taal.....	28
4.4.2 Keuze tussen derde en vierde generatie talen	29
5 GPGPU programmeren	31
5.1 CPU-GPU Analogieën	31
5.1.1 GPU texture = CPU array	31
5.1.2 GPU fragment programma = CPU lus instructies	31
5.1.3 Output naar texture = feedback.....	32
5.1.4 Geometrie rasterisatie = aanroep van computatie.....	32
5.1.5 Texture coördinaten = computatie domein.....	32
5.1.6 Vertex Coördinaten = computatie bereik	32
5.2 Programmeermodel van de GPU.....	32
5.2.1 Stream communicatie.....	34
5.2.2 Implementatie van scatter algoritmen.....	34
5.3 Toepassing in de praktijk	37

6	Ontwikkelapplicaties voor de GPU	38
6.1	NVPerfKit	38
6.1.1	Geïndstrumenteerde NVIDIA driver	39
6.1.2	NVPerfSDK	40
6.1.3	GLExpert	41
6.2	NVShaderPerf	42
6.3	FX Composer	43
6.4	NVPerfHUD	44
6.5	gDEDebugger	47
6.6	GLIntercept	50
6.7	IMDebug	52
6.8	Rendermonkey	53
6.9	Shader Designer	55
6.10	GLTrace	56
6.11	BuGLE	56
7	Performance en Debugging tips	58
7.1	Performance optimalisatie	58
7.1.1	Identificeer de bottleneck	58
7.1.2	Render met een triangle in plaats van quad	60
7.1.3	Render to Texture met Framebuffer Objects	61
7.1.4	Datatransfer van en naar de GPU	62
7.1.5	Efficiënte branching in GPU programma's	67
7.1.6	Profielen, datatypes en data-indeling	68
7.2	Debug methoden	70
7.2.1	Debuggen met afbeeldingen	70
7.2.2	Debuggen met exacte waarden	71
8	Multi-GPU aspecten	74
8.1	GPUs in SLI configuratie	74
8.1.1	AFR	74
8.1.2	SFR	75
8.1.3	SLI anti-aliasing	75
8.2	Onafhankelijk gebruik van meerdere GPUs	76
8.3	De toekomst op het gebied van multi-GPU	77
9	Overige ontwikkelingen	79
9.1	Multi-Core CPU	79
9.2	Cell processor	79
9.3	PhysX processor	80
10	De Toekomst	82
11	Conclusies en aanbevelingen	84
	Referenties	85
	Websites	86
	Begrippenlijst	88
	Bijlage A Omschrijving afstudeeropdracht	96
	Bijlage B Vergelijkingstabel HLSL, GLSL en Cg	97
	Bijlage C Resultaten bandbreedte tests	102

Lijst van Figuren

Figuur 1	Vershil in trends van rekenkracht tussen GPU en CPU	9
Figuur 2	De traditionele grafische pipeline [4].....	11
Figuur 3	De moderne grafische pipeline [4]	11
Figuur 4	Overzicht van het proces dat door de pipeline doorlopen wordt [15].....	12
Figuur 5	De mogelijkheden van de Shader Units in de Fragment Processor Units [4]	13
Figuur 6	De architectuur van de GeForce 6 serie [4].....	14
Figuur 7	Verschillende niveaus van parallelisme in de GPU architectuur	15
Figuur 8	Veel instructies per data element camoufleert geheugen toegangstijd [24].....	16
Figuur 9	De onderdelen van de Cg Runtime API [1]	20
Figuur 10	GLSL Runtime model [8]	24
Figuur 11	De metaprogrammeer stappen van de Sh API [W6]	25
Figuur 12	Het Brook for GPU compilatie- en uitvoeringsmodel	28
Figuur 13	Interface verschillen tussen 3GL (links) en 4GL (rechts)	30
Figuur 14	Beschouwing van GPU pipeline in grafische en generieke context.....	31
Figuur 15	De fragment processor als stream processor voorgesteld	33
Figuur 16	De grafische pipeline als aaneenschakeling van stream processors.....	33
Figuur 17	Scatter versus Gather operaties	34
Figuur 18	Massa – veer systeem in scatter vorm	35
Figuur 19	Massa – veer systeem in gather vorm	35
Figuur 20	Scatter door sorteren op adres	36
Figuur 21	Veelvoorkomende fragmentatie voor GPGPU.....	37
Figuur 22	Beschikbare counters in NVPerfKit 2	39
Figuur 23	NVIDIA performance counters weergegeven in Windows perfmon.....	40
Figuur 24	Debug opties geboden door GLExpert.....	41
Figuur 25	Screenshot van FXComposer	44
Figuur 26	Interactie tussen NVPerfHUD, het systeem en de applicatie	44
Figuur 27	Screenshot Performance Dashboard van NVPerfHUD.....	45
Figuur 28	Screenshot Frame Profiler van NVPerfHUD	46
Figuur 29	Screenshot van gDEDebugger en bijbehorende viewers	48
Figuur 30	HTML log bestand gegenereerd door gDEDebugger	49
Figuur 31	Voorbeeld XML output van GLIntercept.....	51
Figuur 32	Screenshot van de runtime shader editor plugin van GLIntercept.....	52
Figuur 33	Afbeeldinganalyse met IMDebug	53
Figuur 34	Screenshot van Rendermonkey	54
Figuur 35	Screenshot van Shader Designer	55
Figuur 36	Twee mogelijkheden voor het bepalen van een bottleneck [18].....	58
Figuur 37	Bottleneck identificatie flowchart [18]	59
Figuur 38	Rasterisatie van een rechthoekig vlak door één driehoek	60
Figuur 39	Vershil in bandbreedte in rasterizer tussen triangle en quad rendering.....	61
Figuur 40	Transfer bandbreedtes bij verschillende texture formaten en volgordes [19].....	63
Figuur 41	Vertraging van GPU en CPU door glReadPixels.....	65
Figuur 42	Lagere CPU belasting door gebruik van PBO's.....	65
Figuur 43	Asynchrone datatransfer door gebruik van PBO's.....	66
Figuur 44	Optimaal parallel gebruik van CPU en GPU door gebruik van PBO's	66
Figuur 45	Implementeren van statische branches met verschillende delen geometrie [24] .	68
Figuur 46	Efficiënt geheugengebruik kan algoritmen drastisch versnellen.....	69
Figuur 47	Voorbeeld van texture coördinaten als uitvoer	70
Figuur 48	Twee GPUs in AFR configuratie	74

Figuur 49	Twee GPUs in SFR configuratie	75
Figuur 50	Vershil tussen aliased en anti-aliased graphics	76
Figuur 51	GeForce 7950 GX2	77
Figuur 52	Quad SLI opstelling	78
Figuur 53	De rolverdeling binnen een computer met een PhysX processor [W26]	80
Figuur 54	Unsigned bytes download 1024 x 1024	103
Figuur 55	Unsigned bytes readback 1024 x 1024.....	103
Figuur 56	16-bit float download 1024 x 1024	104
Figuur 57	16-bit float readback 1024 x 1024.....	104
Figuur 58	32-bit float download 1024 x 1024	105
Figuur 59	32-bit float readback 1024 x 1024.....	105
Figuur 60	Unsigned bytes download 256 x 256	108
Figuur 61	Unsigned bytes readback 256 x 256.....	108
Figuur 62	16-bit float download 256 x 256	109
Figuur 63	16-bit float readback 256 x 256.....	109
Figuur 64	32-bit float download 256 x 256	110
Figuur 65	32-bit float readback 256 x 256.....	110
Figuur 66	Unsigned bytes download verschil tussen dimensies 256 en 1024.....	113
Figuur 67	Unsigned bytes readback verschil tussen dimensies 256 en 1024	113
Figuur 68	16-bit float download verschil tussen dimensies 256 en 1024.....	114
Figuur 69	16-bit float readback verschil tussen dimensies 256 en 1024	114
Figuur 70	32-bit float download verschil tussen dimensies 256 en 1024.....	115
Figuur 71	32-bit float readback verschil tussen dimensies 256 en 1024	115

Lijst van Tabellen

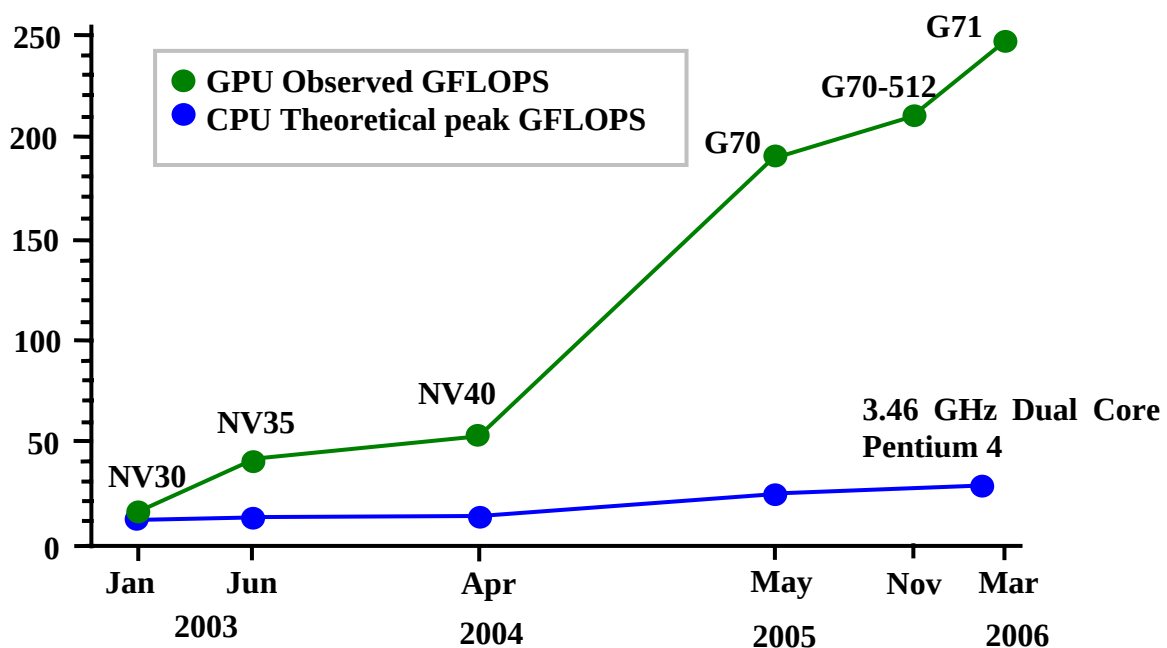
Tabel 1	Vergelijking tussen Shader Model 2.0 en 3.0 [12].....	16
Tabel 2	Overhead kosten van flow-control statements	17
Tabel 3	Cg Vertex Shader Profielen voor OpenGL	22
Tabel 4	Cg Vertex Shader Profielen voor DirectX	22
Tabel 5	Cg Fragment Shader Profielen voor OpenGL.....	22
Tabel 6	Cg Fragment Shader Profielen voor DirectX.....	22
Tabel 7	Overzicht van de door Sh ondersteunde backends.....	26
Tabel 8	Vergelijkingstabel beschikbare hulpapplicaties voor de GPU	38
Tabel 9	Vershil in bandbreedte in rasterizer tussen triangle en quad rendering.....	60
Tabel 10	Transfer bandbreedtes bij verschillende texture formaten en volgordes [19].....	63
Tabel 11	Vergelijking mogelijkheden Shader Model 4.0 met oudere versies [23].....	82
Tabel 12	8-bit unsigned byte download 1024 x 1024	106
Tabel 13	16-bit unsigned byte download 1024 x 1024	106
Tabel 14	16-bit float ARB download 1024 x 1024	106
Tabel 15	32-bit float ARB download 1024 x 1024.....	106
Tabel 16	16-bit float NV download 1024 x 1024	106
Tabel 17	32-bit float NV download 1024 x 1024	106
Tabel 18	8-bit unsigned byte readback 1024 x 1024.....	107
Tabel 19	16-bit unsigned byte readback 1024 x 1024.....	107
Tabel 20	16-bit float ARB readback 1024 x 1024	107
Tabel 21	32-bit float ARB readback 1024 x 1024	107
Tabel 22	16-bit float NV readback 1024 x 1024.....	107
Tabel 23	32-bit float NV readback 1024 x 1024.....	107
Tabel 24	8-bit unsigned byte download 256 x 256	111
Tabel 25	16-bit unsigned byte download 256 x 256	111
Tabel 26	16-bit float ARB download 256 x 256	111
Tabel 27	32-bit float ARB download 256 x 256	111
Tabel 28	16-bit float NV download 256 x 256	111
Tabel 29	32-bit float NV download 256 x 256	111
Tabel 30	8-bit unsigned byte readback 256 x 256.....	112
Tabel 31	16-bit unsigned byte readback 256 x 256.....	112
Tabel 32	16-bit float ARB readback 256 x 256	112
Tabel 33	32-bit float ARB readback 256 x 256	112
Tabel 34	16-bit float NV readback 256 x 256.....	112
Tabel 35	32-bit float NV readback 256 x 256.....	112
Tabel 36	8-bit unsigned byte download verschil tussen dimensies 256 en 1024.....	116
Tabel 37	16-bit unsigned byte download verschil tussen dimensies 256 en 1024.....	116
Tabel 38	16-bit float ARB download verschil tussen dimensies 256 en 1024.....	116
Tabel 39	32-bit float ARB download verschil tussen dimensies 256 en 1024.....	116
Tabel 40	16-bit float NV download verschil tussen dimensies 256 en 1024.....	116
Tabel 41	32-bit float NV download verschil tussen dimensies 256 en 1024.....	116
Tabel 42	8-bit unsigned byte readback verschil tussen dimensies 256 en 1024	117
Tabel 43	16-bit unsigned byte readback verschil tussen dimensies 256 en 1024	117
Tabel 44	16-bit float ARB readback verschil tussen dimensies 256 en 1024.....	117
Tabel 45	32-bit float ARB readback verschil tussen dimensies 256 en 1024.....	117
Tabel 46	16-bit float NV readback verschil tussen dimensies 256 en 1024	117
Tabel 47	32-bit float NV readback verschil tussen dimensies 256 en 1024	117

Lijst van Code fragmenten

Code fragment 1	Pseudo code massa – veer systeem in scatter vorm	35
Code fragment 2	Pseudo code massa – veer systeem in gather vorm.....	35
Code fragment 3	Door NVIDIA aangeraden opt-in mechanisme van NVPerfHUD.....	47
Code fragment 4	Gebruik Framebuffer Object	62
Code fragment 5	Vervang branch code door wiskundige operaties	67
Code fragment 6	Vervang branch code door een conditionele expressie	67
Code fragment 7	Debug macro voor gebruik in shaders.....	72
Code fragment 8	Gebruik van de macro in een shader	72
Code fragment 9	C++ code voor het gebruik van de debug output	73

1 Inleiding

In 1965 heeft Gordon Moore, mede oprichter van Intel, opgemerkt dat de hoeveelheid transistors per vierkante centimeter op een CPU elk jaar verdubbelt. Deze uitspraak staat bekend als 'De Wet van Moore' [W9]. Enkele jaren later bleek dat de vooruitgang toch wat langzamer ging, waardoor de 'Wet' is bijgesteld tot een verdubbeling per 18 maanden. Over de afgelopen jaren hebben fabrikanten van Graphical Processing Units (GPUs¹) fabrikanten echter opgemerkt dat de hoeveelheid transistors per vierkante centimeter van een GPU elke 6 maanden verdubbelt. Deze grotere groei heeft nu al geleid tot een groot absoluut verschil. Zo heeft bijvoorbeeld de NVIDIA GeForce 7800 GTX serie met 302 miljoen [W10] bijna 2 maal de hoeveelheid transistors van een Pentium 4 600 serie met 169 miljoen [W11] aan boord. Dit heeft uiteraard ook invloed op de totale rekenkracht van de processors, waarvan de trend te zien is in Figuur 1. Dit gegeven, in combinatie met de toenemende programmeerbaarheid van de GPU, maakt het dan ook zeer interessant om de GPU te gebruiken als een zeer krachtige coprocessor naast de CPU.



Figuur 1 Verschil in trends van rekenkracht tussen GPU en CPU

Dit verslag geeft een overzicht van de mogelijkheden en beperkingen van de GPU architectuur op dit moment. Ook worden de belangrijkste GPU programmeertalen van dit moment besproken en vergeleken. Verder wordt het GPU programmeer paradigma beschreven en in vergelijking gebracht met programmeer concepten op de CPU. Daarnaast wordt een greep gedaan uit de huidige set beschikbare applicaties die zouden kunnen helpen bij het ontwikkelen van programma's op de GPU. Ook worden er verscheidene tips gegeven voor het behalen van optimale performance en voor het debuggen van GPU applicaties. Tenslotte worden de mogelijkheden van het gebruik van meerdere grafische kaarten in één systeem behandeld en worden enkele andere ontwikkelingen besproken, die een raakvlak hebben met het mogelijke toepassingsgebied van de GPU.

¹ Alle cursief gedrukte termen en afkortingen zijn in de begrippenlijst aan het eind van dit verslag op te zoeken.

2 Doelstellingen en begrenzingen

Deze literatuurstudie is uitgevoerd als een onderdeel van mijn afstudeeropdracht. De oorspronkelijke opdracht omschrijving staat vermeld in Bijlage A. In dit hoofdstuk worden de doelstellingen en onderzoeksvragen beschreven, die aan de hand van deze omschrijving zijn opgesteld. Daarnaast worden de begrenzingen voor dit onderzoek vermeldt.

2.1 Doelstelling

De doelstelling van dit onderzoek is het vergaren van algemene informatie en kennis op het gebied van GPU programmeren. Deze informatie wordt in de vorm van dit verslag samengevoegd en kan als bron van informatie voor de beginnende GPU programmeur dienen. Ook kunnen ontwikkelaars met enige ervaring in GPU programmeren hun kennis verdiepen met verscheidene geboden tips en suggesties voor bijvoorbeeld het optimaliseren van hun programma's.

2.2 Onderzoeksvragen

In dit onderzoek wordt getracht antwoorden te vinden op de volgende vragen:

- ❑ Hoe is de architectuur van grafische kaarten in de loop der tijd geëvolueerd en welke mogelijkheden biedt het nu?
- ❑ Welke programmeertalen zijn er beschikbaar om deze mogelijkheden aan te spreken en wat zijn de belangrijkste verschillen hiertussen?
- ❑ Hoe is de GPU te gebruiken voor algemene, niet noodzakelijkerwijs grafische doeleinden, vaak aangeduid met General Purpose GPU of *GPGPU Programming*?
- ❑ Welke applicaties zijn er beschikbaar om in het programmeren van de GPU te assisteren?
- ❑ Op welke manier kan optimaal gebruik worden gemaakt van de mogelijkheden van de grafische kaart?
- ❑ Is het mogelijk om met meerdere grafische kaarten tegelijk te werken en snelheidswinst te behalen ten opzichte van een enkele kaart wanneer gewerkt wordt met GPGPU toepassingen?
- ❑ Hoe ziet de toekomst van GPU programmeren eruit?

2.3 Begrenzingen

De voorkeur gaat uit naar kennis die toegepast kan worden op meerdere platformen, waaronder ook Linux of Solaris. Hierdoor wordt in mindere mate aandacht besteed aan talen, applicaties en andere zaken die zich uitsluitend richten op één platform. Een voorbeeld hiervan is de GPU programmeertaal *HLSL*, dat afhankelijk is van *DirectX* en dus Windows.

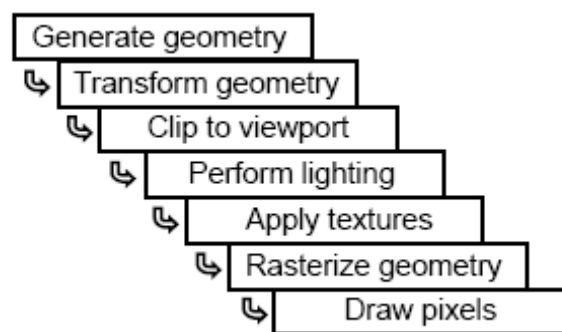
Verder kan er met het oog hierop vanuit worden gegaan dat hardware van *NVIDIA* gebruikt wordt. Dit komt doordat bijvoorbeeld *ATI* voor andere platformen dan Windows, bijvoorbeeld Linux, tot op heden minder goede ondersteuning en drivers levert.

3 GPU architectuur

Bewerkingen in 3D graphics bestaan voor het overgrote deel uit een enorme hoeveelheid herhalingen van dezelfde ‘simpele’ bewerkingen: het tekenen van een grote hoeveelheid onafhankelijke *pixels*. Dit maakte het dan ook erg geschikt om geoptimaliseerd te worden op gespecialiseerde hardware. Deze grafische kaarten zijn in de loop der jaren steeds geavanceerder geworden. In dit hoofdstuk wordt de architectuur en de evolutie ervan nader behandeld.

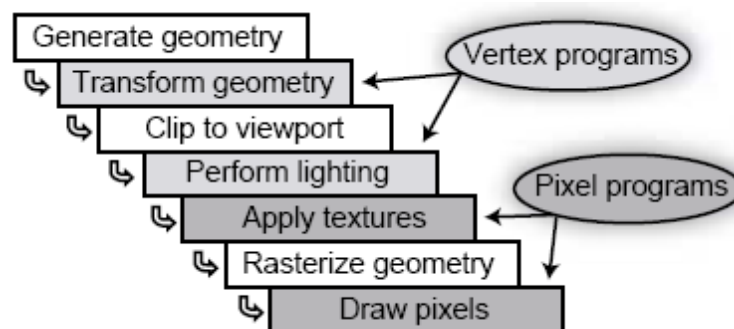
3.1 Evolutie van de architectuur

Traditioneel bestond de *grafische pipeline* uit een serie vaste bewerkingen. De mogelijke bewerkingen voor bijvoorbeeld belichting en transformatie lagen vast in de hardware, en konden door de applicatie aangeroepen worden. Op het resultaat van de bewerking kon door de applicatie enigszins invloed uitgeoefend worden door een aantal parameters mee te geven, zoals transformatie matrices, belichting en materiaal parameters. In dit stadium was de GPU dan ook uitsluitend te gebruiken voor grafische toepassingen en waren de ontwerpers van deze toepassingen nog vrij gelimiteerd in de effecten die ze konden bereiken. Figuur 2 geeft een schematisch overzicht van de stappen die de grafische kaart per *frame* doorliep.



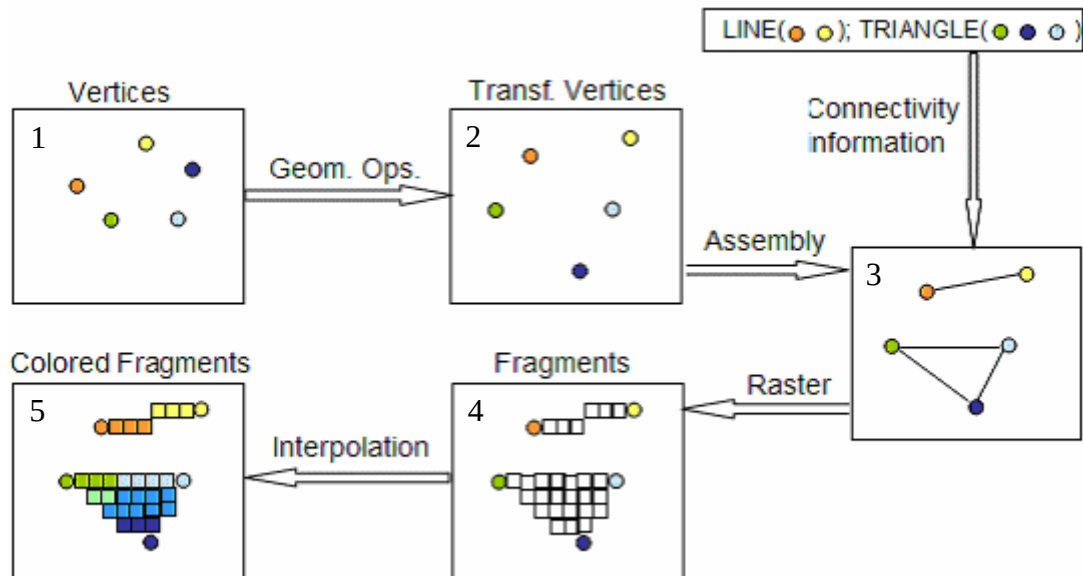
Figuur 2 De traditionele grafische pipeline [4]

De huidige generatie GPUs bieden, naast deze vaste functionaliteit, de mogelijkheid om een eigen transformatie en belichting programma, genaamd een *vertex programma*, of een programma voor het toepassen van *textures* en het daadwerkelijk tekenen van de pixels, genaamd een *pixel programma* of een *fragment programma*, te definiëren en te gebruiken in plaats van de vaste functionaliteit. Een schema van deze modernere grafische pipeline is te zien in Figuur 3.



Figuur 3 De moderne grafische pipeline [4]

Aan de hand van Figuur 4 kan nader uitgelegd worden welke bewerkingen hierdoor mogelijk zijn gemaakt. De vertices en hun kleuren, die in blok 1 te zien zijn, worden door de applicatie zelf opgegeven. Per vertex wordt het gedefinieerde vertex programma aangeroepen. Dit programma zorgt voor de transformatie naar blok 2 en is in staat om bijvoorbeeld de positie van de vertex te transformeren, door bijvoorbeeld van model coördinaten naar wereld coördinaten over te gaan. Verder kunnen zaken zoals belichting op basis van de normaal van de vertex berekend worden.



Figuur 4 Overzicht van het proces dat door de pipeline doorlopen wordt [15]

Nadat het vertex programma de vertices bewerkt heeft, wordt door de vaste functionaliteit van de GPU de door de applicatie meegegeven informatie over de connectiviteit in blok 3 gecombineerd met de getransformeerde vertices. De *raster eenheid* zorgt er vervolgens voor dat de te verwerken fragmenten bepaald worden in blok 4. Deze fragmenten worden vervolgens, voorzien van de geïnterpoleerde kleur, normaal en texture coördinaten, doorgegeven aan het fragment of pixel programma, die als basis dient voor de uiteindelijke kleur die op het scherm zichtbaar zal zijn. Zo kan het fragment voorzien worden van een kleurwaarde uit een texture, of door middel van berekeningen met de normaal, kijk- en belichtingshoek voorzien worden van realistische licht effecten.

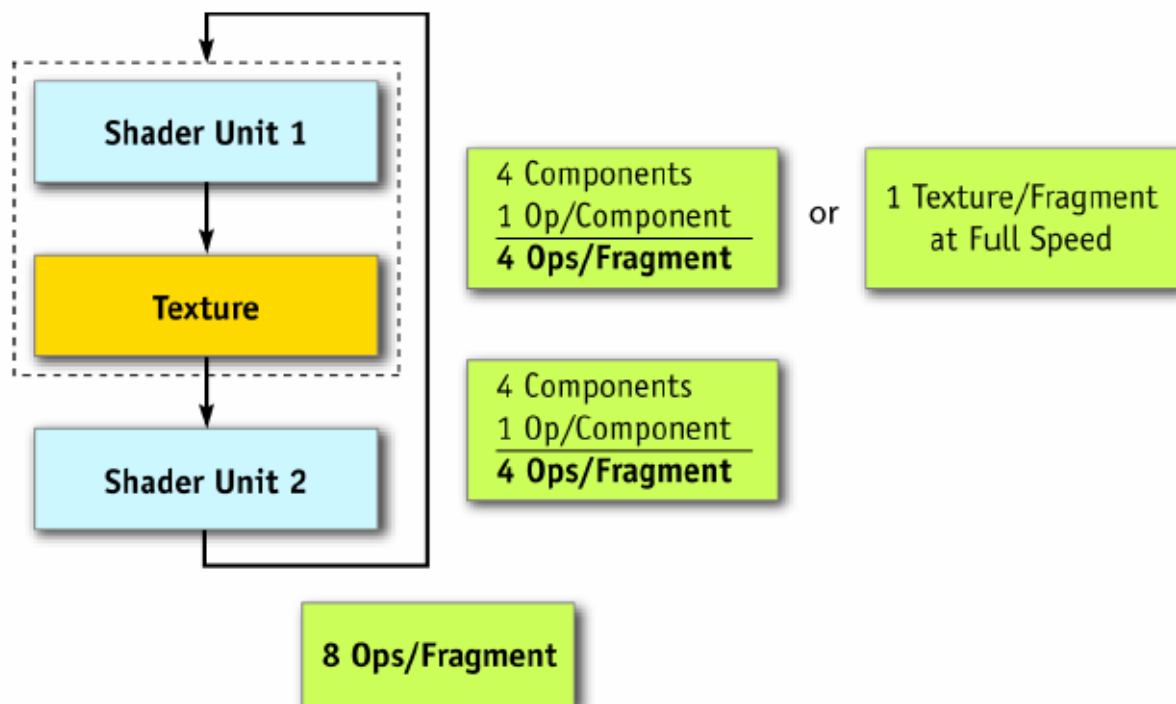
3.2 Pipeline architectuur

Zoals eerder gezegd is de architectuur van de grafische kaart een *pipeline*. Dit betekent dat de verschillende onderdelen, zichtbaar in Figuur 2 en Figuur 3, als een soort ketting aan elkaar liggen. Dit is vergelijkbaar met een assemblagelijns van een fabriek. Het gehele proces van genereren van 3D beelden wordt zo opgedeeld in kleine stappen die sequentieel voltooid worden. Zodra bijvoorbeeld de *fragment processor* klaar is met zijn bewerking op een *fragment*, stuurt deze het resultaat door naar het volgende onderdeel in de pipeline en wordt direct de bewerking van het volgende fragment gestart. Hierdoor kan er toch 1 pixel per *clock cycle* per pipeline geproduceerd worden, ondanks dat de verwerking van een pixel meerdere clock cycles kan duren [13].

Er ontstaat hiermee echter wel een klein tijdsverschil tussen het aanbieden van de eerste onderdelen en de uitvoer van de eerste onderdelen (inlooptijd). Dit leidt vervolgens tot een tijdsverschil tussen het moment van invoer van de laatste onderdelen en de uitvoer van de laatste onderdelen (uitlooptijd). Dit verschil is, vergeleken met de totale verwerkingstijd, bij grote datasets of schermgroottes echter te verwaarlozen. Zo heeft een beeld met een resolutie van 1280x1024 al 1310720 pixels die per frame verwerkt moeten worden.

Gezien de te verwerken data in een GPU een inherent parallel karakter heeft, is het eenvoudig mogelijk om de hoeveelheid verwerkte data per tijdseenheid te vervieelvoudigen door simpelweg extra pipelines toe te voegen. Zo heeft de GPU op de GeForce 7800 serie al 24 pixel pipelines aan boord en is dus theoretisch in staat om 24 pixels per clock cycle te verwerken.

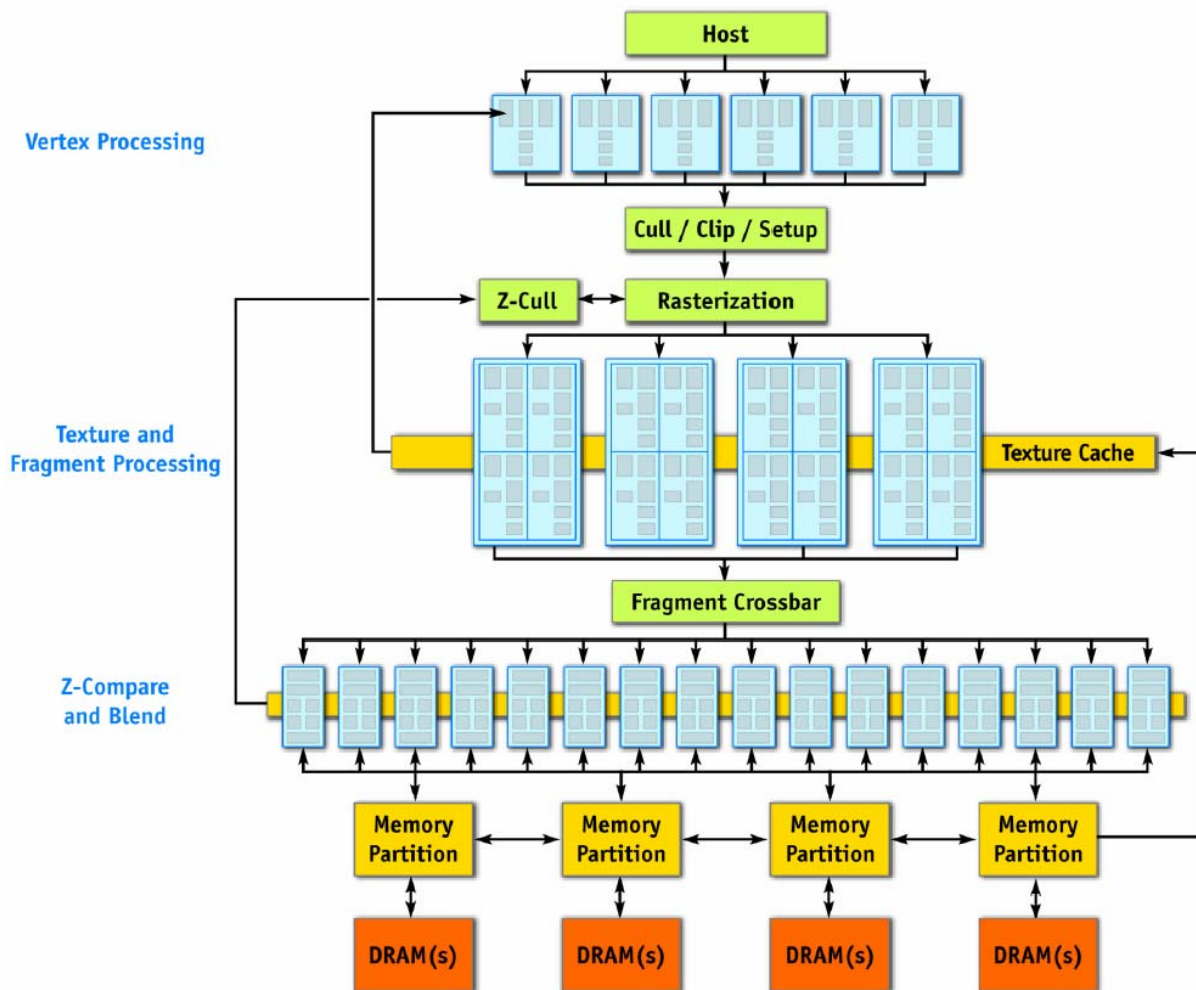
In de GeForce 6 serie bevat elke pixel pipeline één fragment processor unit, die uit twee shader units bestaat. Dit is te zien in Figuur 5. Deze shader units zijn beide in staat om per clock cycle één (wiskundige) operatie op een 4-tuple uit te voeren. De eerste kan in plaats daarvan ook ingezet worden om data uit een texture te lezen. Zo kunnen er per clock cycle maximaal twee vector operaties per pipeline uitgevoerd worden, wat met 24 pipelines neerkomt op $2 \cdot 4 \cdot 24 = 196$ operaties per clock cycle.



Figuur 5 De mogelijkheden van de Shader Units in de Fragment Processor Units [4]

De instructies in de shader units worden op een Single Instruction Multiple Data (SIMD) manier uitgevoerd. Er worden dus instructies in de beide shader units van alle fragment processor units geladen, waarna alle fragmenten door een van de fragment processor units verwerkt worden. Hierna kunnen nieuwe instructies geladen worden en worden de fragments opnieuw door de fragment processor units verwerkt. [4]

In Figuur 6 is een overzicht van de complete architectuur van de GeForce 6 serie GPUs te zien waaruit duidelijk het pipeline karakter van de GPU naar voren komt. De blokken aangegeven met *Vertex Processing* en *Texture and Fragment Processing* zijn hierin de programmeerbare onderdelen van de GPU.



Figuur 6 De architectuur van de GeForce 6 serie [4]

3.3 Mogelijkheden en beperkingen

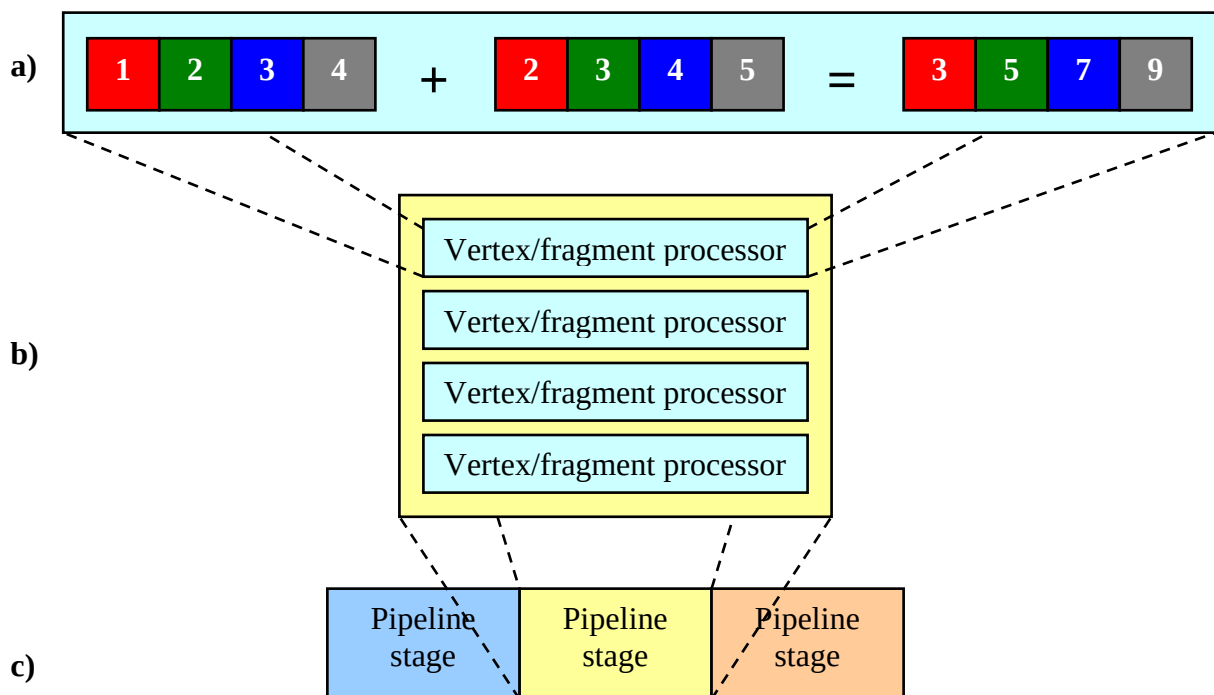
De GPU heeft een architectuur welke sterk afwijkt van die van de CPU. In dit hoofdstuk worden de belangrijkste voor en nadelen, mogelijkheden en beperkingen besproken.

3.3.1 De geboden mogelijkheden en voordelen

Het grootste voordeel dat de GPU heeft ten opzichte van de CPU is een grotere rekenkracht dankzij het sterk parallelle karakter van de architectuur. In de eerste plaats hebben grafische bewerkingen veelal te maken met vectoren van vier elementen, zoals kleuren met een rood, groen, blauw en alfa kanaal of 4D homogene coördinaten. Deze vectoren kunnen dan ook als basis data-objecten beschouwd worden. Rekenkundige instructies zijn hiervoor zo geïmplementeerd, dat ze direct op dergelijke data-objecten uitgevoerd kunnen worden. Twee vectoren kunnen dus met slechts één optel-instructie verwerkt worden, met als resultaat dat de instructie parallel op de verschillende elementen in de vector wordt uitgevoerd. Hierdoor wordt parallelisme op instructie niveau gerealiseerd. Dit is te zien in Figuur 7a.

Daarnaast kunnen er, door de aanwezigheid van meerdere vertex dan wel fragment pipelines, meerdere vertices en fragments tegelijkertijd verwerkt worden. In het eerder genoemde voorbeeld met 24 fragment processors kunnen dan ook 24 data-elementen parallel verwerkt worden. Daardoor ontstaat er ook parallelisme op data niveau. Dit is te zien in Figuur 7b.

Tenslotte wordt er door het pipeline karakter van de gehele architectuur ook nog parallelisme op taak niveau bewerkstelligd. Er kunnen immers tegelijkertijd nieuwe vertices verwerkt worden door de vertex processors terwijl de rasterizer de geometrie, gedefinieerd door eerder verwerkte vertices, opdeelt in fragments. Gelijktijdig kan de fragment processor eerder gegenereerde fragments verwerken.

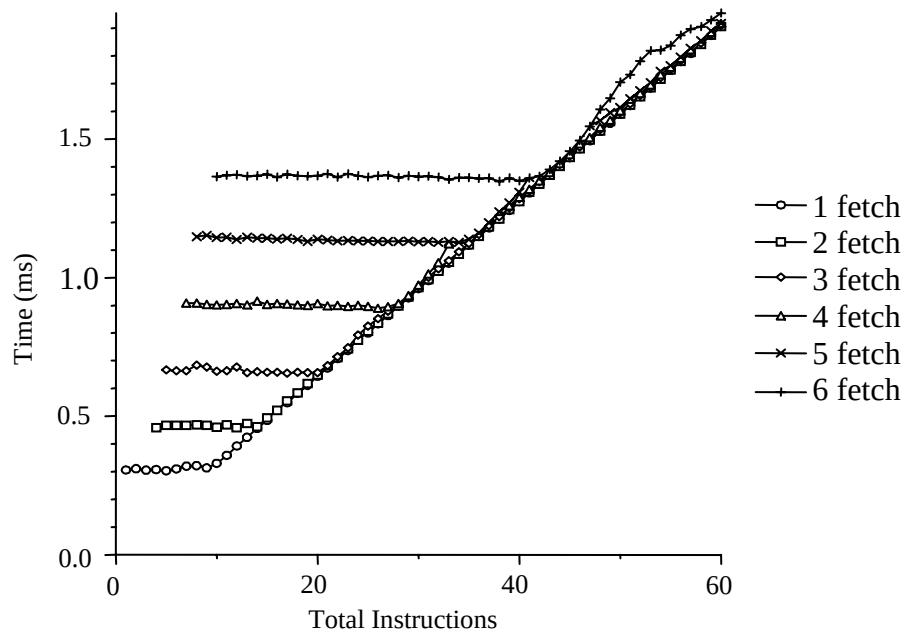


Figuur 7 Verschillende niveaus van parallelisme in de GPU architectuur

Daarnaast is de GPU geoptimaliseerd om veelvoorkomende (geometrische) bewerkingen uit te voeren. Zoals het normaliseren van een vector, het berekenen van de afstand tussen twee vectoren of het berekenen van het inproduct. Zulke berekeningen kunnen op de GPU efficiënt (in bepaalde gevallen zelfs in één clock cycle) uitgevoerd worden.

Een ander groot voordeel dat de GPU heeft boven de CPU is de vele malen grotere bandbreedte naar het geheugen. Zo heeft de GeForce 7800 GTX bijvoorbeeld 38.4 GB/s tot zijn beschikking waar een Pentium CPU met dual channel DDR2 'slechts' 12 GB/s ter beschikking heeft. Deze grote geheugenbandbreedte kan in de praktijk goed gebruikt worden om bijvoorbeeld *convolutie filters* met grote diameters te verwerken. Hiervoor moet er per pixel een grote hoeveelheid data uit het geheugen gelezen worden om de nieuwe waarde van de pixel te kunnen berekenen. Een grote geheugenbandbreedte is daarbij noodzakelijk voor goede performance.

Dit grote verschil wordt veroorzaakt doordat de architectuur van de GPU op bandbreedte geoptimaliseerd is. Dit in tegenstelling tot die van de CPU, waarbij de nadruk ligt op een lage latency. De reden hiervoor is dat een wat hogere latency in de GPU als het ware gecamoufleerd kan worden door parallelisme. Wanneer bijvoorbeeld een berekening op fragment A wacht op data uit het geheugen, kan ondertussen wel een operatie uitgevoerd worden op fragment B. Ook kan het mogelijk zijn dat er voor hetzelfde fragment A nog andere operaties uitgevoerd kunnen worden, die niet afhankelijk zijn van de op te vragen data. Het is voor optimale prestaties uiteraard wel belangrijk dat er een relatief grote hoeveelheid berekeningen per eenheid opgehaalde data plaats vinden om deze datatoegang te kunnen camoufleren.



Figuur 8 Veel instructies per data element camoufleert geheugen toegangstijd [24]

De GPU blinkt dus uit in het uitvoeren van grote hoeveelheden operaties op grote hoeveelheden data. Hierin is het vele malen sneller dan zelfs de snelste CPU van bijvoorbeeld AMD of Intel.

3.3.2 De beperkingen en nadelen

Helaas gelden er naast al deze mogelijkheden nog wel wat beperkingen. Maar waar in het begin nog erg strenge beperkingen golden voor deze programma's, krijgt de programmeur met elke nieuwere generatie GPU steeds meer vrijheid. In DirectX worden deze generaties *Shader Model N* genoemd.

	Shader Model 2.0	Shader Model 3.0
Vertex Shader Model	2.0	3.0
Vertex Shader instructions	256	65535
Dynamic flow support	Partially	Full
Loops & Branches	Full	Full
Pixel Shader Model	2.0	3.0
Pixel Shader instructions	96	65535
Dynamic flow support	Partially	Full
Loops & Branches	Partially	Full

Tabel 1 Vergelijking tussen Shader Model 2.0 en 3.0 [12]

In Tabel 1 is een vergelijking te zien tussen de mogelijkheden van Shader Model 2.0 en 3.0. Hieruit is af te lezen dat de hoeveelheid instructies, die een vertex of fragment programma kan bedragen, sterk is toegenomen. Daarnaast is het nu bij beiden mogelijk om volledig dynamische condities te maken. In Shader Model 2.0 was hier al gedeeltelijke ondersteuning voor, maar de beperking gold dat de testexpressie een op compile-tijd bekende constante moest zijn. Daarnaast ondersteunt het Pixel Shader Model 3.0 het gebruik van *do*-, *for*- en *while*-lussen volledig. In de voorgaande generatie werd dit alleen ondersteund wanneer de lus op compile-tijd volledig uitgeschreven kon worden. Hierbij moet echter wel vermeld worden, dat het gebruik van *branching* wel een bepaalde overhead met zich mee brengt. Een overzicht van deze kosten is te zien in Tabel 2. De programmeur moet dus wel opletten dat de kosten van de controle van de branching niet hoger zijn, dan de daardoor bespaarde operaties.

Een ander belangrijk punt hierin is het verschil tussen het MIMD en SIMD executiemodel van respectievelijk de vertex en de fragment processor. De vertex processor kan verschillende instructies uitvoeren op verschillende data elementen. Elk element kan dus zijn eigen pad volgen, zonder dat dit invloed heeft op de andere elementen. In de fragment processor moet elk element echter dezelfde instructie uitvoeren. Dit betekent dat, wanneer verschillende elementen verschillende paden door de code kiezen, de instructies van beide paden uitgevoerd worden. Dit heeft in geval van weinig coherente data een negatieve invloed op de performance. Zie ook Hoofdstuk 7.1.5.

Instructie	Kosten (cycles)
If / endif	4
If / else / endif	6
Call	2
Return	2
Loop / endloop	4

Tabel 2 Overhead kosten van flow-control statements

Een andere beperking is het feit dat een programma op de GPU niet kan schrijven naar input-bronnen of kan lezen van output-bronnen. Deze beperking is inherent aan het pipeline-karakter. Wanneer dit voor algoritmen nodig is om goed te functioneren ligt de oplossing voorhanden: het programma moet opgedeeld worden in meerdere, sequentieel uit te voeren onderdelen. De output van het eerste deel kan dan worden opgeslagen en als input meegegeven worden aan het tweede deel. Zie hiervoor ook Hoofdstuk 5.1.3.

Met de komst van Shader Model 3.0 zal de programmeur vrijwel nooit meer gelimiteerd zijn in de grootte van het uit te voeren programma. Maar wanneer het programma toch meer dan de ondersteunde hoeveelheid instructies bevat, is het ook hier vaak mogelijk het probleem op te delen in twee of meer kleinere *shaders*. Deze worden dan in serie uitgevoerd, zoals vermeld in Hoofdstuk 5.1.3.

Een ander probleem waar een GPU programmeur tegenop kan lopen is de ‘beperkte’ nauwkeurigheid van de GPU berekeningen. Deze worden in de hardware in maximaal 32 bit floating point getallen conform aan de IEEE 754 standaard (s23e8) berekend. Voor sommige (wetenschappelijke) toepassingen, waarvoor op de CPU gebruik wordt gemaakt van dubbele precisie (s52e11), kan dit gebrek aan nauwkeurigheid een groot gemis zijn. Een oplossing voor dit probleem is al wel geboden [15], maar dit gaat uiteraard wel ten koste van enige performance.

Het laatste en meest onoverkomelijke probleem met programmeren op de GPU is de toepasbaarheid van het GPU programmeermodel op bepaalde problemen/algoritmen. Problemen die bijvoorbeeld erg weinig berekeningen per data-eenheid vergen, of extreem afhankelijk zijn van condities, zullen weinig baat hebben bij portering naar de GPU. Andere problemen die bijvoorbeeld output naar verschillende, data-afhankelijke geheugenlocaties moeten schrijven of naast de output nog veel andere variabelen moeten bijwerken, zijn simpelweg niet of heel moeilijk, te implementeren op de GPU. In Hoofdstuk 5.2.2 worden enkele technieken geboden voor het implementeren van algoritmen van de eerstgenoemde categorie.

4 GPU programmeertalen

Het is nu duidelijk welke programmeerbare functionaliteit de huidige generaties grafische kaarten hebben. Een probleem is nu het aanwenden van deze functionaliteit. In dit hoofdstuk worden de verschillende beschikbare programmeertalen gecategoriseerd en beschreven. Ook worden er richtlijnen geboden voor de keuze tussen de beschikbare programmeertalen.

4.1 Categorisatie van GPU programmeertalen

Om een overzicht van verschillende programmeertalen te geven, worden deze doorgaans op verschillende wijzen gecategoriseerd. Een voorbeeld hiervan is een onderscheid te maken in het programmeer paradigma dat erdoor geïmplementeerd wordt. Voorbeelden hiervan zijn functionele of object-georiënteerde programmeertalen. Een andere aspect waarop gecategoriseerd kan worden, is de mate van abstractie van de onderliggende hardware. Deze wordt doorgaans aangeduid als een generatie van programmeertalen. Ter illustratie zullen de eigenschappen van eerste vier van deze generaties kort uitgelegd worden.

De eerste generatie (*1GL*) is exact de taal die de hardware zelf hanteert. In het geval van een digitale processor bestaat deze taal dan ook uitsluitend uit enen en nullen. Het programmeren in deze taal is erg lastig voor de programmeur.

Om hierin tegemoet te komen zijn tweede generatie programmeertalen (*2GL*) ontwikkeld. Hieronder vallen de assembly achtige talen. Deze talen bevatten een één-op-één relatie tussen de instructies in machinecode en instructienamen in een voor mensen makkelijker te hanteren vorm. Om het programma uit te kunnen voeren, moet een inverse vertaalslag gemaakt worden van de assembly instructies naar machinecode. De leesbaarheid van de programma's is met deze generatie sterk verbeterd. Grote programma's zijn echter nog steeds erg moeilijk te implementeren doordat deze nog steeds instructie voor instructie gegeven moeten worden.

Een grote verbetering hierin is gemaakt met de derde generatie programmeertalen (*3GL*). Hierin hoeft de programmeur zich niet meer te bekommeren om individuele instructies maar kunnen applicaties op een hoger, procedureel niveau gedefinieerd worden. Voorbeelden hiervan zijn C(++) en Java. De hiermee gedefinieerde procedures moeten door een compiler omgezet worden naar één of meerdere instructies in machinecode. Op deze wijze kan een hogere mate van abstractie verkregen worden van het hardware platform en de exacte instructies die het ondersteunt. Er moet echter nog wel precies gedefinieerd worden hoe een gegeven probleem opgelost moet worden.

Vierde generatie programmeertalen (*4GL*) gaan hier nog een stap verder in. In plaats van te beschrijven *hoe* iets moet gebeuren, kan er hiermee volstaan worden met te beschrijven *wat* er moet gebeuren. Deze worden vaak als extra abstractielaag boven op een 3GL gebruikt. In de meeste gevallen zal een compiler de 4GL componenten vertalen naar de onderliggende 3GL, welke vervolgens op gebruikelijke wijze gecompileerd wordt.

In dit verslag worden de definities van deze generaties van programmeertalen ook toegepast op de voor de GPU beschikbare programmeertalen. De hiervoor beschikbare 2GL, waarbij de GPU dus ook op instructieniveau geprogrammeerd kan worden, worden hier niet verder besproken. 3GL hebben een veel hoger niveau van leesbaarheid en daarmee aanpasbaarheid en onderhoudbaarheid hebben. Daarbij gaat dit dankzij volwassen, efficiënte compilers vrijwel nooit meer ten koste van de performance van het programma.

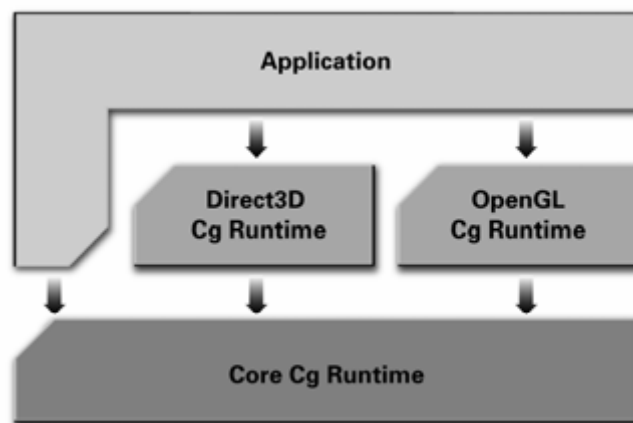
4.2 Derde generatie programmeertalen (3GL)

Als eerste worden de talen Cg, HLSL en GLSL behandeld. Deze programmeertalen zijn op dit moment de grootste en meest gebruikte high level programmeertalen voor GPUs. Deze talen zijn hoofdzakelijk ontworpen voor grafische toepassingen. Het eerste en meest voorname doel van deze talen is dan ook om de programmeur of artiest een programmeerbaar alternatief te geven voor de beperkte en beperkende mogelijkheden die de vaste functionaliteit van oudere grafische kaarten bood. Dit wil echter zeker niet zeggen dat deze programmeertalen niet bruikbaar zijn voor GPGPU. Door gebruik te maken van de CPU-GPU analogieën zoals beschreven in Hoofdstuk 5.1 zijn deze prima, in sommige gevallen zelfs beter te gebruiken voor GPGPU, dan de in Hoofdstuk 4.3 vermelde talen.

In Bijlage B is een overzichtelijke tabel opgenomen die de voornaamste verschillen tussen drie hier besproken programmeertalen weergeeft.

4.2.1 Cg

Cg, oftewel C for graphics was de eerste programmeertaal voor de GPU die uitgebracht werd. Het werd oorspronkelijk door NVIDIA in samenwerking met Microsoft ontwikkeld en is onder de naam Cg eigendom van NVIDIA [W4]. Zoals de naam doet vermoeden is de taal gebaseerd op de programmeertaal C en ziet er dan ook syntactisch vrijwel hetzelfde uit.



Figuur 9 De onderdelen van de Cg Runtime API [1]

De Cg Runtime API bestaat uit 3 onderdelen (Figuur 9)

- ❑ Een basis set functies en structuren dat de volledige functionaliteit van de Runtime bevat
- ❑ Een set OpenGL specifieke functies gebaseerd op de basis set
- ❑ Een set Direct3D specifieke functies gebaseerd op de basis set

De eerste functie set bevat alle basis functionaliteit benodigd om de Cg runtime omgeving te gebruiken. Hieronder valt onder meer het instellen van het te gebruiken profiel en het declareren en initialiseren van Cg programma's. De laatst genoemde twee functie sets zijn bedoeld voor de communicatie tussen Cg programma's en de binnen de applicatie gebruikte API. Hieronder valt bijvoorbeeld het toewijzen van waarden aan de parameters of textures van de shaders.

Het is echter ook mogelijk om Cg shaders toe te passen in een OpenGL applicatie zonder hiervoor gebruik te maken van de Cg Runtime API. In plaats daarvan kan uitsluitend de voor GLSL shaders gedefinieerde API gebruikt worden in combinatie met de GL_EXT_Cg_shader extensie [22]. Dit wordt mogelijk gemaakt doordat de NVIDIA compiler een Unified Compiler Architecture heeft. Dit wil zeggen dat de compiler verschillende front-ends heeft om de verschillende shader programmeertalen te parsen tot een uniform tussenformaat. Nadat de invoer geparsed is, kunnen alle shaders, ongeacht hun initiële programmeertaal, op eenduidige wijze verder gecompileerd worden.

In Cg wordt ook het concept van GPU programma profielen geïntroduceerd. Deze profielen zijn vergelijkbaar met het shader model concept en geven dan ook aan welke GPU functionaliteit beschikbaar is. Wanneer GPUs nieuwe mogelijkheden krijgen, zullen hiervoor nieuwe *OpenGL extensies* en DirectX shader versies gemaakt worden en nieuwe profielen die hiervan gebruik maken. Op deze manier worden de hoeveelheid beschikbare profielen steeds uitgebreid. GPUs blijven hierbij altijd backwards-compatible. Wanneer het nieuwste profiel wordt ondersteund door de GPU, kunnen ook alle oudere profielen gebruikt worden. Deze hebben dan als doel de mogelijkheden van de compiler te 'beperken', zodat het mogelijk is een shader programma te genereren dat ook op oudere hardware kan draaien. Belangrijk is hierbij ook op te merken dat de nieuwere, uitgebreidere profielen doorgaans minder goed presteren dan de oudere profielen. Er vanuit gaande dat deze oudere profielen nog wel de minimaal benodigde functionaliteit bieden om het betreffende shader programma te kunnen uitvoeren. Meer hierover in Hoofdstuk 7.1.6.

In Tabel 3 tot en met Tabel 6 staat een overzicht van de op dit moment beschikbare profielen in Cg en een indicatie van het Shader Model waarmee de mogelijkheden van het profiel het meeste overeen komen. Een videokaart moet de benodigde OpenGL extensies of DirectX versie ondersteunen om shaders voor die profielen te kunnen draaien. De OpenGL profielen *vp*0* en *fp*0* bevatten de (ten tijde van het uitkomen van deze profielen) nieuwste functionaliteit van de NVIDIA GPUs en maken gebruik van NVIDIA specifieke OpenGL extensies, aangeduid met het NV_ voorvoegsel. De profielen *arbvp1* en *arbfp1* maken daarentegen gebruik van door het ARB goedgekeurde extensies, te herkennen aan het ARB_ voorvoegsel, en worden door een groter scala aan GPUs ondersteund. Shaders die hiervoor geschreven zijn, zijn doorgaans ook te gebruiken op bijvoorbeeld GPUs van ATI. Een ander voordeel dat deze profielen hebben is rechtstreekse toegang tot de *OpenGL state* vanuit de shaders.

Cg Profielen	Benodigde OpenGL extensies	Shader Model
Arbvp1	ARB_vertex_program	2.0
vp20	NV_vertex_program1_1	1.1
vp30	NV_vertex_program2	2.0
vp40	NV_vertex_program3	3.0

Tabel 3 Cg Vertex Shader Profielen voor OpenGL

Cg Profielen	Benodigde DirectX en gebruikte shader versie	Shader Model
vs_1_1	DirectX 8.1, DirectX 9 Vertex Shader 1.1	1.1
vs_2_0	DirectX 9 Vertex Shader 2.0	2.0
vs_2_x	DirectX 9c Extended Vertex Shader 2.0	3.0

Tabel 4 Cg Vertex Shader Profielen voor DirectX

Cg Profielen	Benodigde OpenGL extensies	Shader Model
arbfp1	ARB_fragment_program	2.0
fp20	NV_texture_shader + NV_register_combiners	1.1
fp30	NV_fragment_program	2.0
fp40	ARB_fragment_program NV_fragment_program2	3.0

Tabel 5 Cg Fragment Shader Profielen voor OpenGL

Cg Profielen	Benodigde DirectX en gebruikte shader versie	Shader Model
ps_1_1	DirectX 8.1, DirectX 9 Pixel Shader 1.1	1.1
ps_1_2	DirectX 8.1, DirectX 9 Pixel Shader 1.2	1.2
ps_1_3	DirectX 8.1, DirectX 9 Pixel Shader 1.3	1.3
ps_2_0	DirectX 9 Pixel Shader 2.0	2.0
ps_2_x	DirectX 9c Extended Pixel Shader 2.0	3.0

Tabel 6 Cg Fragment Shader Profielen voor DirectX

Shaders geschreven in Cg kunnen op runtime gecompileerd worden voor een op dat moment te kiezen profiel. Ook kunnen ze al van tevoren door een op zichzelf staande compiler verwerkt worden tot de eerder genoemde assembly code. Een nadeel van deze laatste optie is, dat er in enkele gevallen van het programma verschillende gecompileerde versies moeten bestaan. Bijvoorbeeld wanneer op runtime pas besloten kan worden, welk profiel gebruikt wordt. Dit kan bijvoorbeeld noodzakelijk zijn doordat de hardware waarop de shader moet draaien bepaalde functies van de nieuwere profielen niet ondersteunt. Daarnaast kan het nadelig zijn doordat het programma in feite stil staat in de tijd: het kan bijvoorbeeld niet profiteren van nieuwe compiler optimalisaties totdat het opnieuw gecompileerd wordt [1].

Een ander groot voordeel van Cg is dat deze taal als (een van de) eerste ontwikkeld is en dus al een lange geschiedenis van ontwikkeling en uitbreiding achter zich heeft liggen. Hierdoor is een groot scala aan kwalitatief goede documentatie en voorbeelden voorhanden waardoor het eenvoudig is de taal te leren. Verscheidene programmeertalen zijn geschikt gemaakt voor het gebruik van Cg in de applicatie door de runtime library hiervoor beschikbaar te maken. Naast C en C++ is Cg ondersteuning ook beschikbaar voor onder andere Java en Delphi.

4.2.2 HLSL

Zoals eerder vermeld heeft Microsoft, samen met NVIDIA, Cg ontwikkeld. Het resultaat van deze samenwerking gaat onder de naam HLSL (High Level Shading Language) door als onderdeel van Microsoft's DirectX API en ondersteunt hierom uitsluitend een Direct3D omgeving. Door de nauwe samenwerking met NVIDIA in de ontwikkeling, is de syntax (nagenoeg) gelijk aan dat van Cg.

Veel shaders geschreven voor HLSL zijn in Cg doorgaans zonder aanpassingen te gebruiken. Andersom kan dit wat meer problemen opleveren doordat Cg een aantal mogelijkheden ondersteunt, waar dit door HLSL niet gedaan wordt. Zie voor details de vergelijkingstabel in Bijlage B. In Tabel 4 en Tabel 6 zijn de overeenkomstige Shader Models bij de verschillende versies van DirectX te zien.

Het belangrijkste verschil tussen HLSL en de DirectX profielen van Cg, is dat HLSL programma's worden geïnitieerd en gedefinieerd door middel van de DirectX API. Dit is de reden dat moderne spellen voor Windows, die vrijwel altijd gebruik maken van DirectX, in de meeste gevallen gebruik maken van HLSL. De taal is te gebruiken in alle programmeertalen, die de DirectX API ondersteunen.

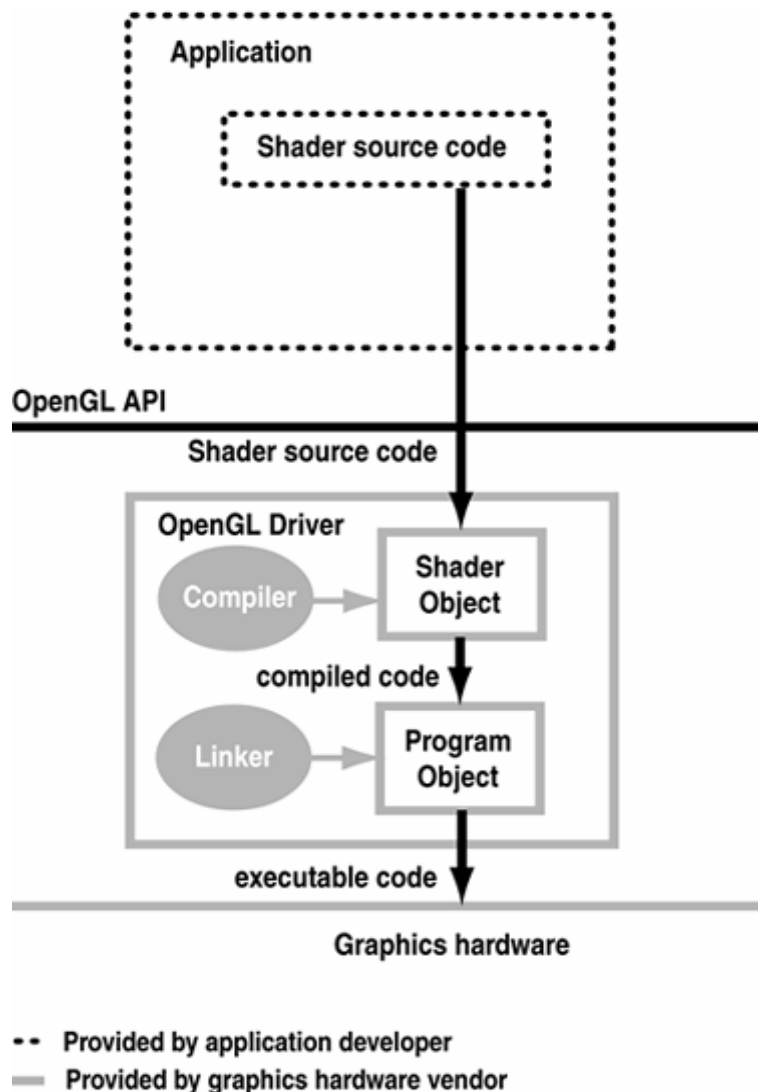
4.2.3 GLSL

GLSL staat voor OpenGL Shading Language en is te zien als de OpenGL tegenhanger van HLSL. Het is voor het eerst als extensie uitgebracht op OpenGL 1.5, maar inmiddels door het OpenGL ARB formeel opgenomen als onderdeel van OpenGL 2.0 [W5].

Een groot voordeel is dat deze programmeertaal zo nauw is geïntegreerd in OpenGL. Zo is bijvoorbeeld de *OpenGL state* direct beschikbaar voor de shaders. Hierdoor kan de communicatie tussen de applicatie en de shader verminderen en dit zorgt voor groter programmeer gemak.

Een ander voordeel van deze taal is dat programma's die hiervoor geschreven zijn op alle GPUs kunnen werken die OpenGL in de drivers ondersteunen. Het is hiermee dus volledig onafhankelijk van de fabrikant van de video kaart, dit in tegenstelling tot Cg, dat voor geavanceerde profielen gebruik maakt van NVIDIA-specifieke OpenGL extensies. In Figuur 10 is te zien hoe de applicatie, de OpenGL API en de driver van de GPU met elkaar samen werken.

GLSL shaders kunnen momenteel uitsluitend pas op runtime gecompileerd worden. Dit compileren gebeurt door de OpenGL driver interface, die met elke nieuwe driver van de fabrikant van de grafische kaart geüpdate kan worden met de nieuwste optimalisaties en de nieuwste mogelijkheden van hun grafische kaart. Op deze manier kan elke fabrikant zorgen voor optimaal gebruik van de mogelijkheden van de grafische kaart, zonder dat hiervoor aanpassingen gemaakt hoeven te worden in de specificatie van GLSL zelf [8].



Figuur 10 GLSL Runtime model [8]

Een nadeel is wel dat revolutionair nieuwe ontwikkelingen, waarvan geen gebruik gemaakt kan worden door enkel de drivers aan te passen, moeilijker toe te passen zijn. Dit omdat de fabrikant van een bepaalde GPU niet direct de GLSL specificatie kan aanpassen. Wanneer dit noodzakelijk is, moet er allereerst een OpenGL extensie gemaakt worden die de nieuwe ontwikkeling ondersteunt. Hiermee zal het voordeel van fabrikant onafhankelijkheid van GLSL programma's, die de betreffende functionaliteit gebruiken, deels verloren gaan. Dit is, zoals te zien in Bijlage B, ook nu al het geval doordat NVIDIA bepaalde functionaliteit ook in GLSL toelaat ondanks het feit dat dit geen onderdeel is van de specificatie van GLSL. Wel bestaat er een optie voor de compiler om de mogelijkheden van de NVIDIA GPUs te beperken, zodat uitsluitend shaders worden uitgevoerd die strikt aan de GLSL specificatie voldoen. In andere gevallen wordt dan een foutmelding gegenereerd.

Een ander nadeel is dat GLSL altijd alle functionaliteit ondersteunt. Het is het dus niet mogelijk om snelheidswinst te behalen door de mogelijkheden van de shader taal te beperken, zoals dit bij Cg wel het geval is.

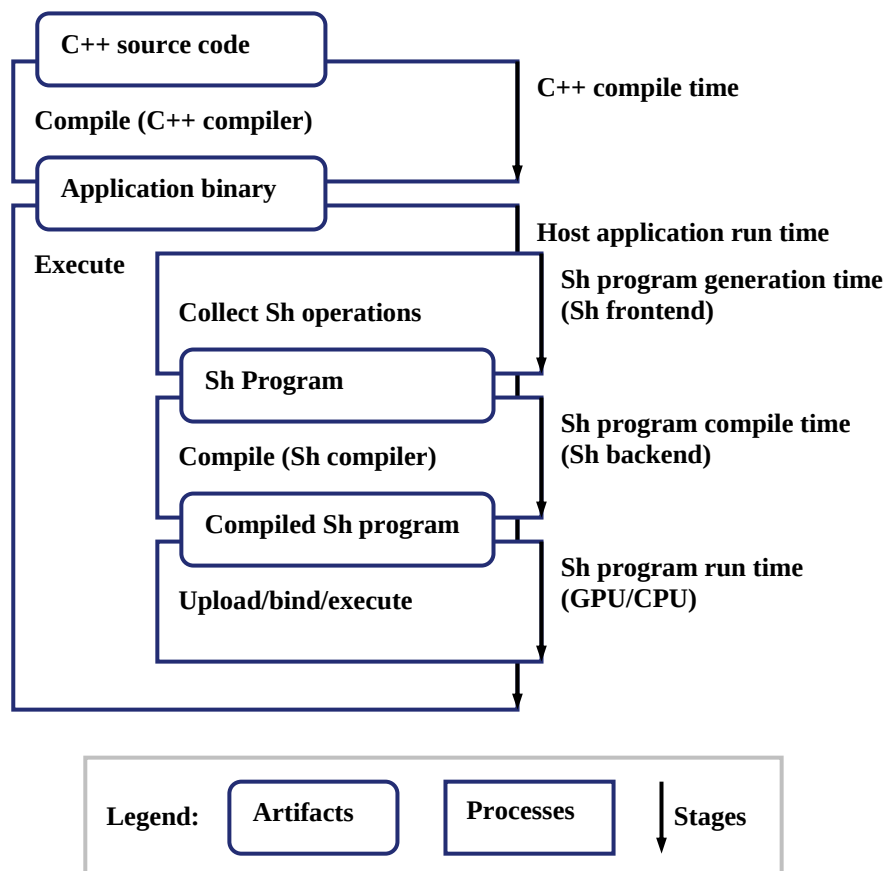
GLSL is te gebruiken in alle programmeertalen die het gebruik van OpenGL 2.0, of OpenGL 1.5 in combinatie met de benodigde extensies, ondersteunen.

4.3 Vierde generatie programmeertalen (4GL)

In dit hoofdstuk worden de programmeertalen Sh en Brook behandeld. Deze twee talen wijken af van de drie eerder genoemde talen door het feit dat deze in meer of mindere mate ontworpen zijn voor GPGPU. In deze programmeertalen kan de programmeur zich beperken tot het opgeven van welke berekeningen uitgevoerd moeten worden. De exacte instructies voor het aanwenden van de specifieke componenten van de GPU en het opzetten van de programmeeromgeving hiervoor worden hierbij door de programmeertaal zelf afgehandeld. Hiermee valt het onder de definitie van een 4GL. Het idee voor deze programmeertalen is vrij nieuw. Dit is dan ook een van de redenen dat deze twee talen minder volwassen zijn dan bijvoorbeeld Cg, HLSL en GLSL.

4.3.1 Sh

Sh is ontworpen op de Universiteit van Waterloo en was oorspronkelijk onderdeel van een systeem genaamd SMASH, wat later opgedeeld is in twee delen: een GPU simulator genaamd Sm en Sh. De laatste is een krachtige metaprogrammeertaal dat het mogelijk maakt om de shader en de host-applicatie beiden in één taal (C++) te definiëren. Met metaprogrammeren wordt het gebruik van een programma om een ander programma te genereren of te veranderen bedoeld. Programma's zelf worden dus gezien als data dat gemanipuleerd kan worden door andere programma's. Besturingssystemen, compilers, assemblers en linkers zijn voorbeelden van programma's die ook dit uitgangspunt hebben. [2].



Figuur 11 De metaprogrammeer stappen van de Sh API [W6]

Sh is als metaprogrammeertaal in staat om de gedefinieerde programma's (shaders) te draaien op verschillende *backends*. Het kiezen van zo'n backend bepaalt het resultaat van de compilatie. In Tabel 7 staat een overzicht van alle op dit moment ondersteunde backends in Sh en in Figuur 11 zijn de metaprogrammeer stappen van de Sh API weergegeven. Het eerste backend komt overeen met de ARB profielen van Cg. De tweede is uiteraard gelijk aan het gebruik van GLSL. De laatste is voor shader compilatie een meer unieke toevoeging. Deze stelt de programmeur namelijk in staat om programma's die geschreven zijn voor de GPU te draaien op een CPU. Dit kan nuttig zijn in gevallen waar geen geschikte GPU aanwezig is om het programma uit te voeren. Belangrijk is het hierbij op te merken dat er op runtime gekeken kan worden welke hardware aanwezig is en afhankelijk daarvan het programma te compileren en uit te voeren op de CPU dan wel op de GPU.

Daarnaast kan het uitkomst bieden bij het debuggen van een applicatie. Wanneer geen geschikte GPU debuggers beschikbaar zijn (zie Hoofdstuk 6) kunnen reguliere CPU debuggers gebruikt worden eventuele problemen met de shader te identificeren.

Sh Backend	Resultaat van compilatie
ARB	ARB_fragment_program en ARB_vertex_program assembly code
GLSL	High-level OpenGL 2.0 GLSL code
GCC	C code dat door de GCC compiler gecompileerd en dynamisch gelinked wordt

Tabel 7 Overzicht van de door Sh ondersteunde backends

Doordat Sh een C++ API is, profiteren in Sh geschreven shaders meteen van alle Object georiënteerde voordelen van C++. Zo kunnen de gemaakte shaders als klassen of zelfs templates gedefinieerd worden en is het dus makkelijk mogelijk hun functionaliteit uit te breiden door gebruik te maken van bijvoorbeeld subklassering [2].

Verder kunnen de host-applicatie en de shader variabelen delen doordat beide onderdelen in dezelfde taal geschreven zijn. Om bijvoorbeeld een texture te maken voor een shader is het voldoende om deze in OpenGL aan te maken, de extra binding tussen deze texture en de shader die ervan gebruik moet maken is bij Sh niet nodig. Ook het uitlezen van resultaten uit het videogeheugen, bijvoorbeeld het resultaat van een bewerking, dat is opgeslagen in een texture, kan simpelweg gedaan worden door de inhoud van deze variabele aan te roepen in een C++ statement. Wanneer deze niet up-to-date is, detecteert Sh dit en haalt het de nieuwste versie uit het videogeheugen om als resultaat van de aanroep op te leveren. [2].

Sh al goed voorbereid op GPGPU doordat deze taal naast het programmeren van shaders ook in staat is om zogenaamde *stream programma's* te maken. Zie Hoofdstuk 5.2 voor details over deze vorm van programmeren. De abstractie van fragment en vertex shaders naar streams heeft als voordeel dat de programmeur helemaal niets meer hoeft te weten over de onderliggende werking van de grafische omgeving zoals OpenGL [2]. Wanneer een programma behalve de berekeningen verder niets met de grafische kaart hoeft te doen, bijvoorbeeld het weergeven van resultaten als 3D graphics, hoeft de programmeur zelfs geen regel OpenGL code te typen. De benodigde instructies worden allemaal door de metaprogrammeer mogelijkheden van Sh geïmplementeerd.

Sh is nog sterk in ontwikkeling, de nieuwste versie is de tweede release candidate voor 0.8, uitgebracht in februari 2006. Het is dus al erg dicht bij de eerste geplande stabiele versie 0.8, die volledig overeen moet komen met de specificatie zoals beschreven in het boek *Metaprogramming GPUs with Sh* [2]. Verder blijkt uit de mailing lijst en de website [W6] dat er nog volop activiteit is in dit project.

4.3.2 Brook for GPU

Brook is een programmeertaal die oorspronkelijk aan de Stanford Universiteit is ontwikkeld voor de stream processor Merrimac. Later is hiervoor een compiler en run-time library ontwikkeld die het mogelijk maakt om programma's (kernels) geschreven in Brook uit te voeren op een GPU. De taal zelf is vanwege de oorsprong dan ook uitsluitend gericht op algemene toepassingen en niet direct voor 3D-grafische doeleinden te gebruiken.

In Figuur 12 is te zien hoe een Brook programma gecompileerd en uitgevoerd kan worden op de GPU. In eerste instantie wordt het programma geschreven in de taal Brook. De programmeur is, in het geval het programma op de GPU uitgevoerd moet worden, wel enigszins beperkt in de mogelijkheden van die taal. Zo kunnen er bijvoorbeeld uitsluitend streams gemaakt worden van gewone floats of vectoren tot en met 4 float elementen. Streams van structs, arrays of andere C types zijn niet mogelijk. Een andere belangrijke beperking is dat een kernel slechts één stream als output kan genereren.

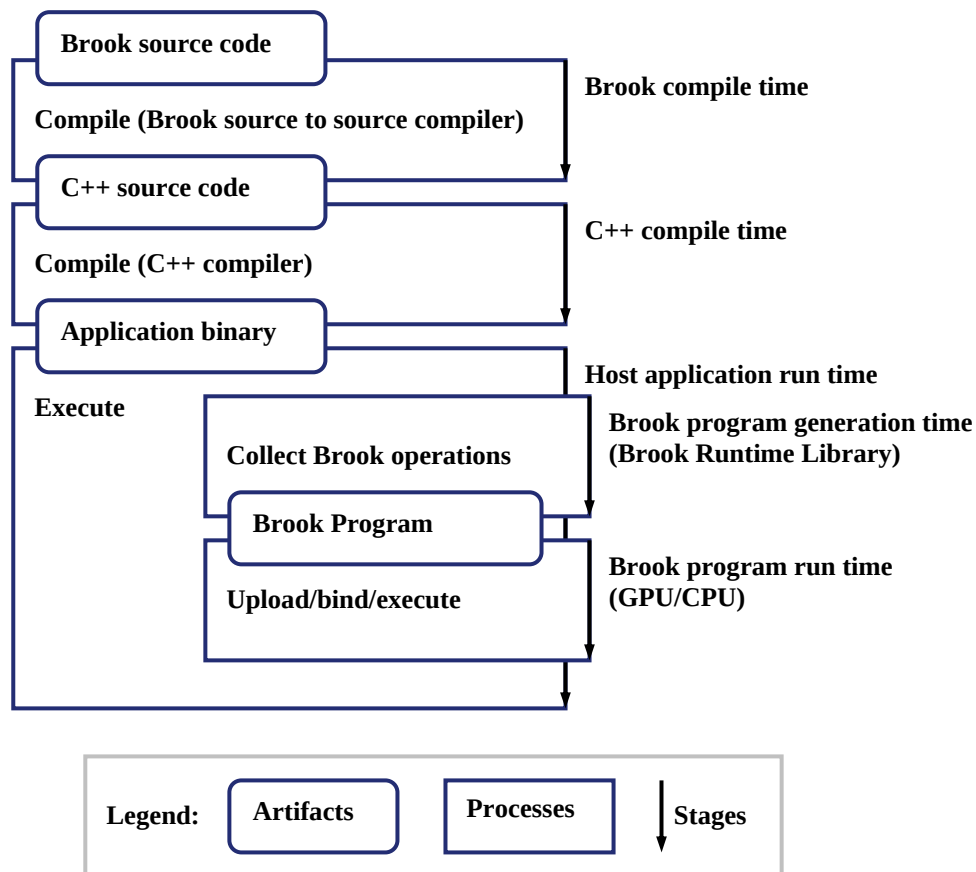
Brook is grotendeels te zien als een uitbreiding op ANSI C. De belangrijkste verschillen hiermee zijn de volgende [W7]:

- ❑ Declaratie en gebruik van streams is mogelijk met de `<>` notatie
- ❑ Toevoeging van keywords *kernel*, *reduce*, *iter*, *indexof*
- ❑ Toevoeging van types *float2*, *float3*, *float4*
- ❑ Kernel functies ondersteunen uitsluitend dat wat ook in Cg/HLSL ondersteund wordt

Wanneer het programma in Brook geschreven is, kan het met behulp van de meegeleverde source to source compiler omgezet worden naar C++ source code. Deze kan, op zijn beurt, volgens de gebruikelijke procedure gecompileerd worden tot een applicatie, die tegen de Brook Run-time library gelinked is. Deze applicatie wordt vervolgens, afhankelijk van het gekozen/beschikbare backend, uitgevoerd op de GPU met OpenGL of DirectX dan wel geheel op de CPU. Een model van de werking van dit systeem is te zien in Figuur 12.

Een belangrijk verschil met Sh is dat een programma geschreven in Brook niet direct in een andere applicatie, geschreven in bijvoorbeeld C++, te integreren is. Hiervoor dient eerst het programma met bijgeleverde compiler omgezet te worden naar C++ code, waarna deze files handmatig moeten worden geïntegreerd in de host applicatie. Dit maakt tevens het gebruik van GPU programma's in een grafische context moeilijk, wat de taal lastiger en minder veelzijdig toepasbaar maakt.

Een ander belangrijk punt om op te merken is het feit dat de laatste officiële release van versie 0.4 gedaan is in oktober 2004. Een kijkje in de CVS repository van het project wijst uit dat de ontwikkeling hiervan in de tussentijd niet geheel stil heeft gestaan, maar wel op een zéér laag pitje staat. De laatste wijziging, die ik na enig zoeken hierin heb gevonden, dateert al enige maanden terug. Ook de website en de daarop vermelde problemen en beperkingen zijn gedurende mijn afstudeeropdracht ongewijzigd gebleven.



Figuur 12 Het Brook for GPU compilatie- en uitvoeringsmodel

4.4 Overwegingen bij programmeertaal keuzes

In de voorgaande paragrafen zijn de belangrijkste kenmerken van de meest gebruikte en volwassen programmeertalen van GPUs behandeld. Hieruit is gebleken dat de wijze van programmeren, de geboden mogelijkheden en beperkingen van deze talen nogal uiteenlopen. In deze paragraaf worden motivaties en richtlijnen geboden om een keuze te kunnen maken tussen deze talen. Hierbij worden de in Hoofdstuk 2.3 vermelde begrenzungen als uitgangspunt genomen. Bijlage B kan met een gedetailleerde vergelijking tussen HLSL, GLSL en Cg als aanvulling op deze paragraaf worden gezien.

4.4.1 Keuze voor een derde generatie taal

In de eerste plaats kan gekeken worden naar welke 3GL het meest geschikt zou kunnen zijn voor een applicatie. Op basis van de eis van platform onafhankelijkheid valt HLSL al direct af vanwege afhankelijkheid van DirectX. GLSL en Cg zijn beide te gebruiken in combinatie met de platformonafhankelijke OpenGL API. GLSL is als onderdeel van OpenGL op elk platform te gebruiken, waar OpenGL ondersteuning aanwezig is. Cg gebruikt daarnaast doorgaans nog een eigen toolkit, die op dit moment beschikbaar is voor Windows, Linux, Mac OS X en Solaris x86. Zoals in Hoofdstuk 4.2.1 is vermeld, is het echter ook mogelijk om het gebruik van de Cg Toolkit achterwege te laten en uitsluitend van de OpenGL API gebruik te maken. Cg kan in tegenstelling tot GLSL echter ook overweg met de DirectX API.

Zoals uit Hoofdstuk 6 naar voren komt, spreken de op dit moment beschikbare ontwikkelapplicaties in het voordeel van GLSL. Bij het debuggen van Cg shaders zijn op runtime uitsluitend de gecompileerde, op assembler lijkende instructies zichtbaar en te bewerken. In het geval GLSL gebruikt wordt, kan wel direct de broncode hiervan bewerkt worden. Ook zijn de beschikbare ontwikkelomgevingen, bijvoorbeeld voor het maken van prototypes van shaders, uitsluitend in staat om met GLSL shaders te werken.

Wat betreft geboden mogelijkheden voor de programmeur kan Cg gezien worden als een superset van GLSL. Dit geldt met name wanneer gebruik wordt gemaakt van een NVIDIA videokaart. NVIDIA ondersteunt zelfs het gebruik van bepaalde Cg features in GLSL, die geen onderdeel uitmaken van de officiële GLSL specificatie. Daarnaast biedt het meer mogelijkheden tot instellingen, die de performance kunnen beïnvloeden. Hierbij valt te denken aan de in Hoofdstuk 4.2.1 vermelde profielen of de instelmogelijkheden voor compiler optimalisaties. Tenslotte bevat de standaard functiebibliotheek van Cg enkele functies, die niet in GLSL voorkomen. Enkele voorbeelden hiervan zijn hyperbolische goniometrische functies (*sinh*, *cosh*, *tanh*), *round*, *sincos*, *refract* en *transpose*.

Een ander punt dat in het voordeel van Cg spreekt is officiële documentatie. Bij de Cg toolkit wordt standaard een zeer uitgebreide User's Manual [1] en Cg Reference Manual [16] geleverd. Daarnaast is op de website van NVIDIA nog een GPU Programming Guide [17] beschikbaar gesteld. Hierin staan allerlei programmeer en optimalisatie tips. Verder is het boek The Cg Tutorial [5] te koop. Voor GLSL is aan officiële documentatie slechts het boek OpenGL Shading Language [8] beschikbaar. Voor verdere documentatie is de programmeur toegewezen op wat er op het internet te vinden is.

Tenslotte blijkt uit de ervaringen opgedaan tijdens mijn afstudeeropdracht dat Cg in de praktijk op bepaalde punten handiger is in het gebruik dan GLSL. Dit uit zich met name in zaken zoals het inladen van shader code. Wanneer de code zich in een apart bestand bevindt, moet deze voor GLSL eerst zelf ingelezen worden. Pas daarna kan de string met de inhoud van die file als shader gecompileerd worden. Hierdoor is het ook een stuk lastiger om meerdere shaders, bijvoorbeeld wanneer deze een soortgelijke rol vervullen met een kleine variatie, uit één bestand te laden. Cg biedt hiervoor een interface in de toolkit. Tevens is het in deze files mogelijk om volledig gebruik te maken van de preprocessor mogelijkheden van een C(++) compiler. Daardoor ondersteunt Cg naast macro's ook het gebruik van includes.

4.4.2 Keuze tussen derde en vierde generatie talen

Naast de genoemde eerste keuze is er ook nog een overweging te maken over het wel of niet gebruik maken van een van de nieuwere generaties programmeertalen, die programmeren op een nog hoger niveau mogelijk maken. In feite spelen hierbij dezelfde voor- en nadelen als bij CPU programmeertalen. Simpel gezegd vertaalt het zich tot een afweging tussen gemak en controle.

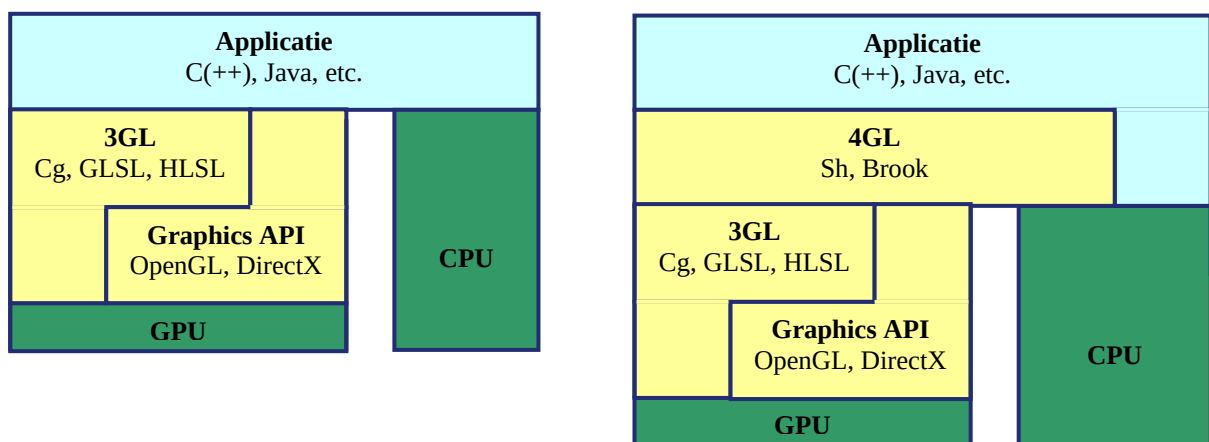
Om wat meer in detail te treden heeft een 4GL als voordeel dat er minder, of in sommige gevallen zelfs géén, kennis van OpenGL of de architectuur van de GPU noodzakelijk is voor het gebruik maken van de geboden mogelijkheden. Dit is met name een voordeel wanneer de beoogde applicatie een puur rekenkundige toepassing betreft, en dus niets met grafische aspecten en weergave daarvan te maken heeft. Is dat wel het geval, dan zou deze kennis ook zonder gebruik van GPU programmeren aanwezig moeten zijn.

Een ander voordeel dat een 4GL biedt is de mogelijkheid om het beschreven algoritme niet alleen op de GPU uit te voeren, maar doorgaans ook zonder problemen op de CPU. De vraag blijft hierbij natuurlijk of het nuttig is. De keuze om een algoritme op de GPU uit te voeren zal in vrijwel alle gevallen gebaseerd zijn op de grote performance verschillen tussen GPU en CPU. Wanneer het toch op de CPU uitgevoerd wordt, is dit voordeel uiteraard verdwenen.

Mogelijke argumenten waarom dit toch nuttig kan zijn is bijvoorbeeld de mogelijkheid om de applicatie uit te voeren op een 'legacy systeem' dat nog niet over de benodigde hardware beschikt. Verder kan deze methode additionele debug methoden bieden gezien het scala aan CPU debuggers enorm veel groter is dan voor de GPU. Belangrijk is het hierbij wel rekening te houden met het feit dat uitsluitend het algoritme zelf te debuggen is. Indien bijvoorbeeld door een bug in de library van de 4GL het algoritme incorrect op de GPU geïmplementeerd wordt, is het mogelijk dat hetzelfde algoritme op de CPU beter/anders uitgevoerd wordt dan op de GPU. In dit geval is de programmeur in de meeste gevallen afhankelijk van support van het ontwikkelteam van de 4GL voor een oplossing.

Naast de voordelen kleven aan het gebruik van 4GL ook enkele nadelen. De belangrijkste hierbij komen voort uit het feit dat de programmeur minder/geen controle heeft over de werkelijke implementatie op de GPU. Hierdoor kan dus alleen gebruik worden gemaakt van bepaalde optimalisaties wanneer de ontwikkelaars van de 4GL hiervoor regels hebben opgesteld in hun 'compiler'. Er kunnen dus naar alle verwachtingen, in ieder geval in de nabije toekomst, vrijwel geen algoritme afhankelijke optimalisaties gedaan worden. Hierbij valt bijvoorbeeld te denken aan het gebruik van gespecialiseerde data representaties, die optimaal passen bij het gegevens benaderingsgedrag van het algoritme, of het gebruik maken van de in Hoofdstuk 7.1.5 vermelde methoden voor efficiënte branching binnen de algoritmen.

Een ander nadeel dat hier deels aan ten grondslag kan liggen is het feit dat het altijd langer zal duren voordat de nieuwste ontwikkelingen op GPU gebied in de 4GL verwerkt zijn. Gezien de talen 'onder water' nog altijd gebruik maken van de grafische API en in veel gevallen ook 3GL, zijn deze in feite een extra laag op het programmeermodel, te zien in Figuur 13. Alle lagen in dit model moeten een dergelijke nieuwe mogelijkheid ondersteunen, wil de programmeur hiervan gebruik kunnen maken. Wederom is er dus een afhankelijkheid van het ontwikkelteam van de 4GL.



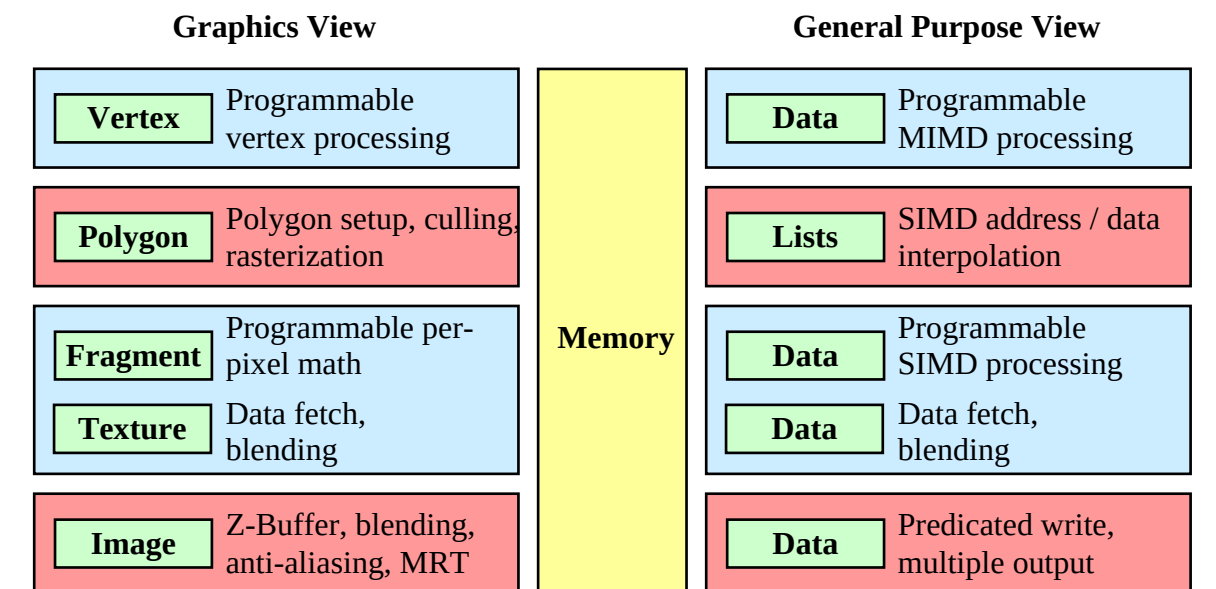
Figuur 13 Interface verschillen tussen 3GL (links) en 4GL (rechts)

5 GPGPU programmeren

GPGPU staat voor General Purpose Graphical Processing Unit en is de term voor het uitvoeren van algemene, niet grafische bewerkingen op een GPU. In Hoofdstuk 3 is uitgelegd hoe de GPU geprogrammeerd kan worden voor verwerking van 3D gegevens. In dit hoofdstuk wordt uitgelegd worden hoe deze programmeerbaarheid voor algemene doeleinden gebruikt kan worden.

5.1 CPU-GPU Analogieën

Om de GPU voor algemene doeleinden te gebruiken moet een denkstap gemaakt worden tussen concepten in de CPU en GPU programmeren. Door middel van een aantal analogieën wordt duidelijk gemaakt hoe verschillende onderdelen van de grafische pipeline in GPGPU context beschouwd kunnen worden. Een overzicht hiervan is te zien in Figuur 14. In de volgende paragrafen zullen enkele concepten verder uitgewerkt worden.



Figuur 14 Beschouwing van GPU pipeline in grafische en generieke context

5.1.1 GPU texture = CPU array

De gegevens voor verwerking van een programma op de CPU bevinden zich doorgaans in een array. Een soortgelijke vorm van dataopslag is op de GPU beschikbaar als een texture. Het uitlezen van een element uit een array komt hiermee overeen met een texture sample operatie.

5.1.2 GPU fragment programma = CPU lus instructies

Een CPU programma gaat doorgaans met een lus alle data-elementen langs en hierop sequentieel de bewerkingen uitvoeren die zich binnen in deze lus bevinden. In een GPU is een fragment programma op te vatten als de uit te voeren instructies in die lus. De texture coördinaten, die per fragment verschillen zoals de iteratie counter van een for-loop, kunnen als index dienen binnen de data-texture om zo alle elementen langs te gaan. Dit proces kan per aanroep van het fragment programma parallel verlopen.

5.1.3 Output naar texture = feedback

Veel algoritmen zijn op te splitsen in een aantal losse stappen, waarbij elke stap afhankelijk is van de resultaten van de stap daarvoor. Om dit te bereiken moet de uitvoer van het eerste fragment programma opgeslagen worden zodat het kan dienen als invoer voor de volgende. Dit is mogelijk door (delen van) het *framebuffer* te kopiëren naar een texture. In nieuwere generaties GPUs is het zelfs mogelijk rechtstreeks naar een texture te tekenen, zie hiervoor Hoofdstuk 7.1.3.

5.1.4 Geometrie rasterisatie = aanroep van computatie

Bij rasterisatie van de geometrie worden één of meer fragments gegenereerd. Elk met hun eigen, geïnterpoleerde texture coördinaten, kleuren en overige data. Doordat op elk van deze fragments het fragment programma wordt uitgevoerd, kan deze fase opgevat worden als het aanroepen van de computatie. In een programma op de CPU is dit dus vergelijkbaar met het begin van elke iteratie van een while- of for-loop.

5.1.5 Texture coördinaten = computatie domein

De fragmenten bevatten na het rasteriseren geïnterpoleerde texture coördinaten. Deze coördinaten kunnen gezien worden als indices in een array, en is vergelijkbaar met de iteratie index van een for-loop. Hiermee wordt bepaald welke data-elementen uit de data-texture verwerkt moeten worden.

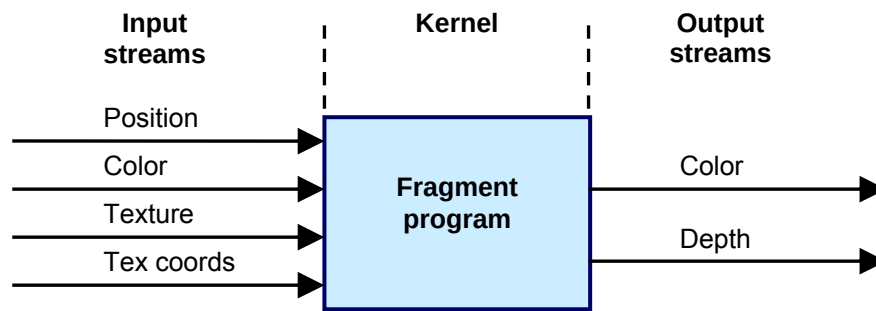
5.1.6 Vertex Coördinaten = computatie bereik

Zoals eerder vermeld worden fragmenten gegenereerd door rasterisatie van de geometrie. Door vier vertices van een *quad* (een vierhoek in 3D Graphics) in beeldcoördinaten te tekenen kan deze direct het bereik van de computatie bepalen.

5.2 Programmeermodel van de GPU

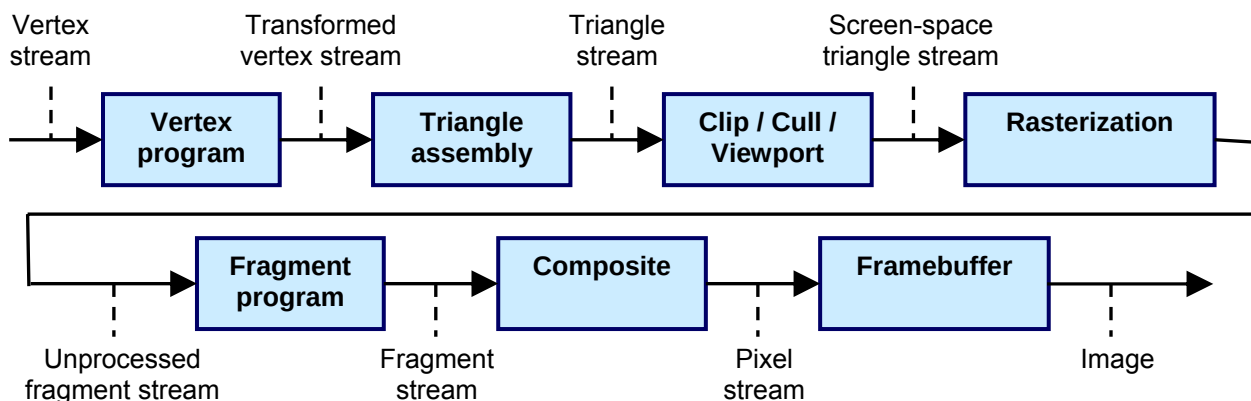
Om de GPU te programmeren voor generieke toepassingen is het belangrijk om te weten hoe het programmeermodel van de GPU eruit ziet. Deze wordt omschreven als het stream programmeermodel [14]. Deze vorm van programmeren bestaat al langer en kan rechtstreeks worden toegepast op gespecialiseerde processors zoals de Merrimac, die ontwikkeld is op Stanford University.

Het idee achter stream programmeren is dat er een programma, genaamd een kernel, gedefinieerd is met één of meerdere input- en outputstreams. Deze streams bestaan uit geordende set van onafhankelijke data elementen van hetzelfde type. Dit programma wordt herhaaldelijk uitgevoerd wordt op de elementen uit deze input stromen. De uitvoer van zo'n kernel is dan ook weer in de vorm van één of meerdere streams. Dat de GPU kan worden voorgesteld als een stream processor, wordt duidelijk in Figuur 15.



Figuur 15 De fragment processor als stream processor voorgesteld

De uitvoer, gegenereerd door zo'n kernel, kan vervolgens ook weer als invoer voor een andere kernel dienen. Op deze wijze is het mogelijk om met het gebruik van meerdere eenvoudige kernels toch complexe eindresultaten te behalen. De gehele grafische pipeline is op te vatten als een keten van dergelijke kernels. Dit is gemodelleerd in Figuur 16.



Figuur 16 De grafische pipeline als aaneenschakeling van stream processors

Hieruit is meteen te zien dat de uitvoerstream niet noodzakelijkerwijs dezelfde hoeveelheid elementen hoeft te bevatten als de inputstream. Zo zullen drie vertices, tezamen slechts één triangle genereren. Één zo'n triangle kan door de rasterizer op zijn beurt weer het genereren van duizenden fragments tot gevolg hebben.

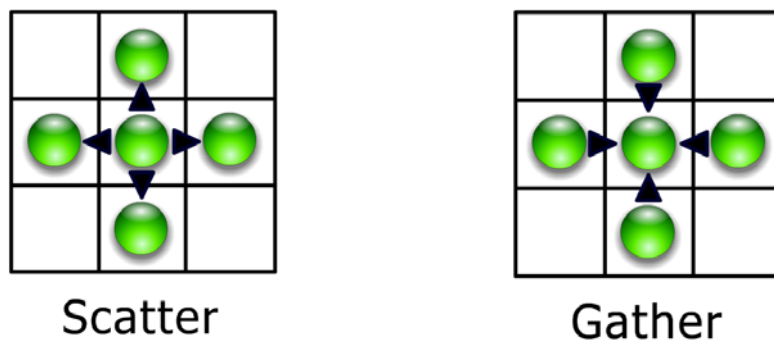
Dit stream programmeer model forceert het blootstellen van parallelisme binnen een applicatie door de onafhankelijkheid van de data-elementen in een stream. Dit maakt het mogelijk om één kernel parallel uit te voeren op alle data-elementen. In de GPU gebeurt dit door de aanwezigheid van meerdere vertex en fragment processors.

In de praktijk maken GPGPU toepassingen het meeste gebruik van de mogelijkheden van de fragment processors. Dit komt doordat een GPU meer van dit type processors heeft dan vertex processors. Een ander belangrijk voordeel is dat fragment processors zich aan het eind van de grafische pipeline bevinden. Hierdoor is de weg van uitvoer naar het geheugen korter. De door de fragment processor gegenereerde uitvoer kan direct naar het geheugen weggeschreven worden. Dit terwijl de uitvoer van vertex processors nog eerst verwerkt moet worden door de rasterizer en de fragment processor.

5.2.1 Stream communicatie

Bij veel algoritmen, uitgevoerd op een stream processor, speelt communicatie een grote rol. Dit is het geval wanneer meerdere data-elementen uit een inputstream nodig zijn voor het bepalen van één element van de uitvoerstream. Enkele voorbeelden hiervan zijn het bepalen van de maximumwaarde van een dataset, of het toepassen van een convolutiefilter op een afbeelding.

Deze algoritmen hebben veelal twee gedaanten: zij vallen onder de categorie scatter dan wel gather algoritmen. Deze termen slaan op de geheugentoegang zoals die in het betreffende algoritme gedaan wordt. De volgende uitleg gaat uit van de verwerking van een element e en relevante (buur)elementen B waarop of waardoor invloed uitgeoefend wordt. In het geval van een scatter algoritme wordt de invloed van e op B verwerkt in de huidige staat van B . Hiervoor moet er geschreven kunnen worden naar één of meerdere semi-willekeurige geheugenadressen. In het geval van een gather algoritme zal bij de verwerking van e de invloed van B verwerkt worden in de nieuwe staat van e zelf. Hiervoor moet er dus gelezen kunnen worden uit semi-willekeurige geheugenadressen. Zie Figuur 17 voor een schematische voorstelling van deze twee methoden.



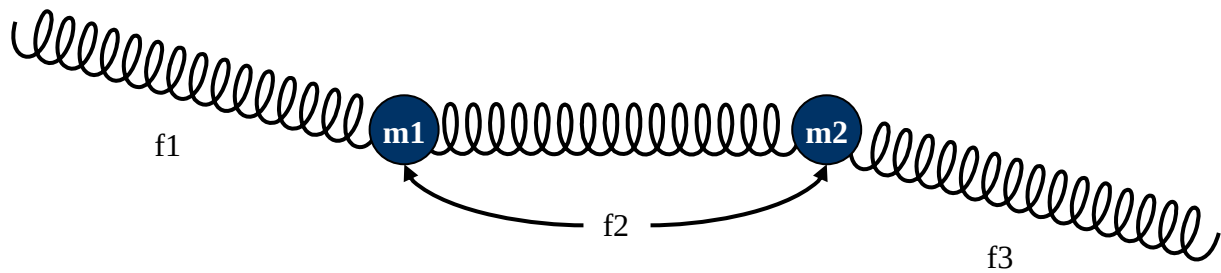
Figuur 17 Scatter versus Gather operaties

5.2.2 Implementatie van scatter algoritmen

Zoals eerder dit hoofdstuk aangegeven worden vrijwel alle GPGPU algoritmen uitgevoerd op de fragment processor. In Hoofdstuk 3.3.2 is uitgelegd, dat het niet direct mogelijk is om met een fragment processor scatter operaties uit te voeren. Immers, bij het uitvoeren van een fragment programma ligt het geheugenadres waarnaar geschreven wordt al vast. Dit is namelijk de locatie van de fragment binnen het gehele uitvoer bereik. Er zijn echter al enkele methoden bedacht voor het implementeren van algoritmen, die oorspronkelijk in scatter vorm ontwikkeld zijn. Enkele van deze methoden worden in de volgende paragrafen behandeld.

5.2.2.1 Omzetten naar een gather algoritme

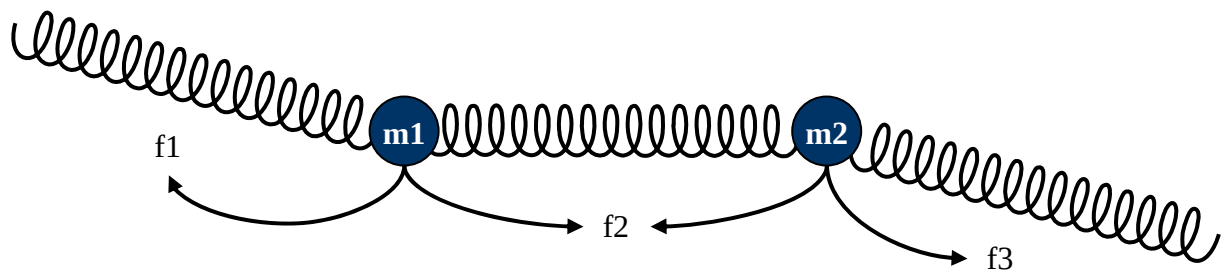
In de eerste plaats is een deel van deze algoritmen om te zetten naar een gather algoritme. Dit is echter alleen mogelijk wanneer de uitvoer adressen altijd gelijk zijn. In het voorbeeld hieronder [24] wordt de staat van de massa's links en rechts van de veer aangepast.



Figuur 18 Massa – veer systeem in scatter vorm

```
for each spring
    f = computed force
    mass_force[left] += f;
    mass_force[right] -= f;
```

Code fragment 1 Pseudo code massa – veer systeem in scatter vorm



Figuur 19 Massa – veer systeem in gather vorm

```
for each spring
    f = computed force
for each mass
    mass_force = f[left] -
                f[right];
```

Code fragment 2 Pseudo code massa – veer systeem in gather vorm

Deze methode is helaas niet in alle gevallen toe te passen. Dit is uitsluitend mogelijk wanneer de gebruikte scatter adressen altijd identiek zijn zoals in het bovenstaande voorbeeld. Een andere situatie waarin dit geldt is in het geval van toepassing van convolutiefilters. Wanneer deze methode toe te passen is, zal deze wel de best mogelijke performance leveren. De complexiteit van het algoritme is in deze gather vorm namelijk gelijk aan de oorspronkelijke scatter vorm.

5.2.2.2 Punt renderen

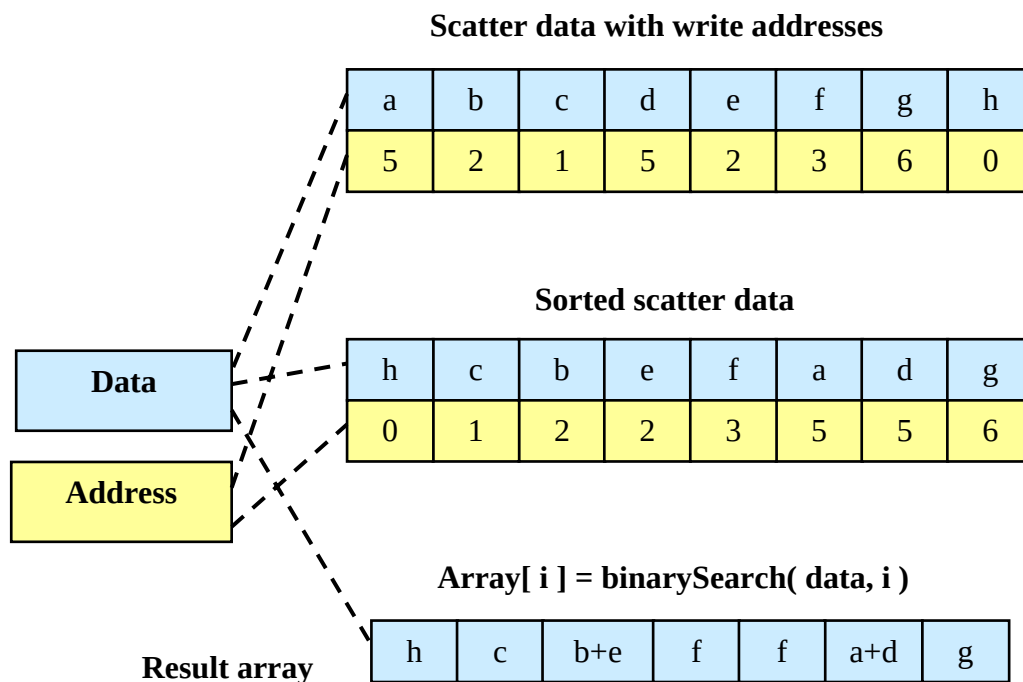
Indien de uitvoer adressen niet vast zijn maar continu kunnen veranderen, bestaat de mogelijkheid om gebruik te maken van de scatter mogelijkheden van de vertex processor. Zoals verteld in Hoofdstuk 3 is de vertex processor in staat om vertices te transformeren. Door van deze mogelijkheid gebruik te maken kan dus de locatie van een vertex binnen de viewport veranderd worden. Anders gezegd kan hiermee het uitvoer adres van het daarna uit te voeren fragment programma op de juiste waarde gezet worden. Op deze wijze kunnen dus, door niet één quad te tekenen maar in plaats daarvan losse vertices, met behulp van de vertex processor toch scatter algoritmen geïmplementeerd worden.

Deze methode heeft echter wel enkele nadelen. Ten eerste wordt in veel gevallen de grafische hardware niet optimaal benut. Wanneer een zeer grote hoeveelheid punten ter verwerking aangeboden wordt, ondergaat de vertex processor een vele malen hogere belasting dan in het geval van reguliere GPGPU toepassingen. Daarbij komt dat de mogelijkheden van de rasterizer, zoals het kunnen interpoleren van waarden zoals texture coördinaten, vrijwel geheel onbenut blijven.

Verder moet er opgelet worden wanneer meerdere vertices dezelfde uitvoer locatie krijgen. In dit geval wordt er, ongeacht de hoeveelheid vertices op die locatie, altijd één fragment gegenereerd. Dit fragment krijgt dan ook de waarden zoals texture coördinaten en dergelijke van één van de vertices mee. Het is dus op deze wijze niet mogelijk om in één *renderpass* de uitvoer samen te stellen uit meerdere bronpunten.

5.2.2.3 Sorteren op adres

De laatste mogelijkheid die hier behandeld wordt, is ook vermeld in GPU Gems 2 [4]. Deze methode geeft als uitvoer niet alleen de berekende uitvoer waarde, maar ook een bijbehorend adres waar deze waarde aan toegevoegd moet worden. Voor verdere verwerking dienen de data-adres tupels gesorteerd te worden op het adres. De voorgestelde methode voor dit sorteren is een bitonic sorteer algoritme [4]. Dit sorteren kan helaas niet (efficiënt) in één enkele renderpass. In het ergste geval kost dit $O(n \log^2(n) + \log(n))$ passes, en kan hierdoor een substantieel deel van het gehele algoritme vormen.



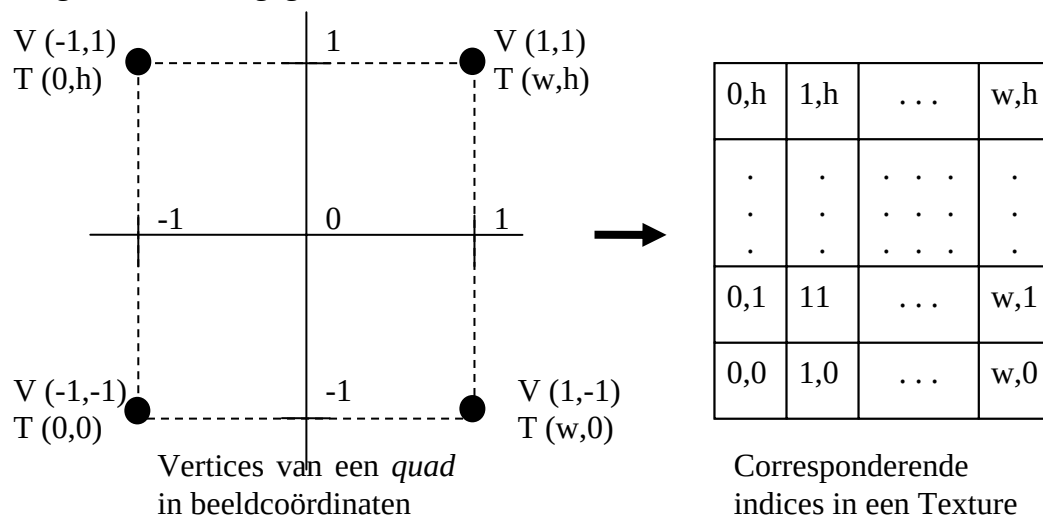
Figuur 20 Scatter door sorteren op adres

Nadat alle waarden op adres gesorteerd zijn, kan de nieuwe output efficiënt op de juiste plaats weggeschreven worden door een binary search toe te passen op de gesorteerde data. Hierbij kan tevens makkelijk rekening gehouden worden met de mogelijkheid dat er meerdere waarden samengevoegd moeten worden voor de uiteindelijke uitvoer.

5.3 Toepassing in de praktijk

Zoals in Hoofdstuk 4 te lezen valt zijn de meeste programmeertalen voor de GPU niet direct geschikt gemaakt voor GPGPU. Echter, door gebruik te maken van de eerder vermelde analogieën kunnen deze toch voor dit doeleinde ingezet worden.

Kort samengevat moeten alle benodigde data elementen in één of meerdere textures en/of constanten gezet worden zodat alle gegevens beschikbaar zijn voor de GPU. Vervolgens moet het algoritme in één of meerdere fragment programma's ingedeeld worden. Hierna kan, wanneer gebruik wordt gemaakt van *Orthografische Projectie* [4] een *quad* getekend worden zoals in Figuur 21 is weergegeven.



Figuur 21 Veelvoorkomende fragmentatie voor GPGPU

Links in Figuur 21 staat de getekende *quad* schematisch weergegeven. De vertices bevatten beeldcoördinaten V en texture coördinaten T . Wanneer gebruik wordt gemaakt van een *viewport* (het gebied in het framebuffer waarnaar getekend moet worden) ter grootte van de dataset die verwerkt moet worden (rechts in Figuur 21), wordt er voor elk van de elementen één fragment gegenereerd met texture coördinaten gelijk aan de indices binnen de texture. In een uit te voeren fragment programma kunnen middels de hiermee verkregen texture coördinaten de inputwaarden van deze en/of omliggende elementen van de invoer verwerkt worden. Op deze wijze kan een één-op-één verwerking van een in- en uitgevoerde dataset gerealiseerd worden.

Na de hierboven genoemde tekenprocedure staat het resultaat van de berekening in het op dat moment actieve framebuffer. Deze kan vervolgens door de CPU uitgelezen worden of gekopieerd worden naar een andere texture, die gebruikt kan worden als input voor het volgende fragment programma. Op deze wijze kan een complex algoritme samengesteld worden uit meerdere simpele stappen. In Hoofdstuk 7.1 staan methoden beschreven om dit zo efficiënt mogelijk te doen.

Voor volledige implementatie details kan het beste gekeken worden naar kleine programmeer voorbeelden op het internet. Een goed voorbeeld hiervan is bijvoorbeeld de 'Hello-world applicatie' op de website van de GPGPU organisatie [W1]. Hiervan is zowel een GLSL als een Cg versie te vinden. Een andere locatie waar concrete implementaties, elk toegespitst op een bepaalde specifieke toepassing, te vinden zijn is bijvoorbeeld de NVIDIA SDK [W12].

6 Ontwikkelapplicaties voor de GPU

Uit de op dit moment beschikbare tools valt duidelijk op te maken dat de games industrie de voornaamste doelmarkt is van de GPU en alle ontwikkelingen hieromtrent. Vrijwel alle games worden ontwikkeld voor Windows, en hiervan gebruikt het grootste gedeelte DirectX als grafische API. Dit is dan ook de oorzaak dat veel tools als eerste, of in veel gevallen zelfs uitsluitend, voor Windows worden ontwikkeld. Bepaalde applicaties hiervoor kunnen echter toch nuttig zijn, ook wanneer de in Hoofdstuk 2.3 genoemde begrenzungen in acht worden genomen. Gezien de grote mate van gelijkheid tussen bijvoorbeeld HLSL en Cg kunnen ontwikkelde HLSL shaders ook vrijwel zonder aanpassingen in een Cg context gebruikt worden. In Tabel 8 is een overzicht te zien van alle applicaties en toolkits, die in dit hoofdstuk behandeld worden

Naam	OS	Taal	IDE	Debug	Profile	Huidige Versie	URL
NVPerfKit	Windows	n.v.t.	Nee	Ja	Ja	2.0	[W20]
NVShaderPerf	Windows	Cg HLSL GLSL	Nee	Nee	Ja	1.8	[W15]
Fx Composer	Windows	HLSL	Ja	Ja	Ja	1.8	[W13]
NVPerfHUD	Windows	HLSL	Ja	Ja	Ja	4	[W14]
gDEDebugger	Windows ¹	n.v.t.	Nee	Ja	Ja	2.5	[W16]
GLIntercept	Windows Linux ²	Cg GLSL	Ja	Ja	Ja	0.5	[W17]
IMDebug	Windows Linux ²	n.v.t.	Nee	Ja	Nee	1.13-beta	[W18]
Rendermonkey	Windows	HLSL GLSL	Ja	Ja	Nee	1.6.91	[W19]
Shader Designer	Windows Linux ³	GLSL	Ja	Nee	Nee	1.5.9.4	[W21]
GLTrace	Windows ⁴ Linux ⁴	n.v.t.	Nee	Ja	Nee	2.3-alpha 2	[W22]
BuGLE	Linux ⁴		Nee	Ja	Nee	0.0.20060528	[W23]

Tabel 8 Vergelijkingstabel beschikbare hulpapplicaties voor de GPU

6.1 NVPerfKit

NVPerfKit is een toolkit ontwikkeld door NVIDIA. Het bestaat momenteel uit een geïntegreerde driver, NVPerfSDK en GLExpert. Deze onderdelen worden in de volgende paragrafen kort behandeld. Alle instellingen van deze onderdelen zijn te doen in de *NVIDIA Developer Control Panel*, die na installatie als optie beschikbaar is gekomen in de standaard *Windows Control Panel*.

¹ De Linux versie is al wel in ontwikkeling

² Source is beschikbaar en volgens readme te porten naar Linux. Echter geen Linux Makefile beschikbaar

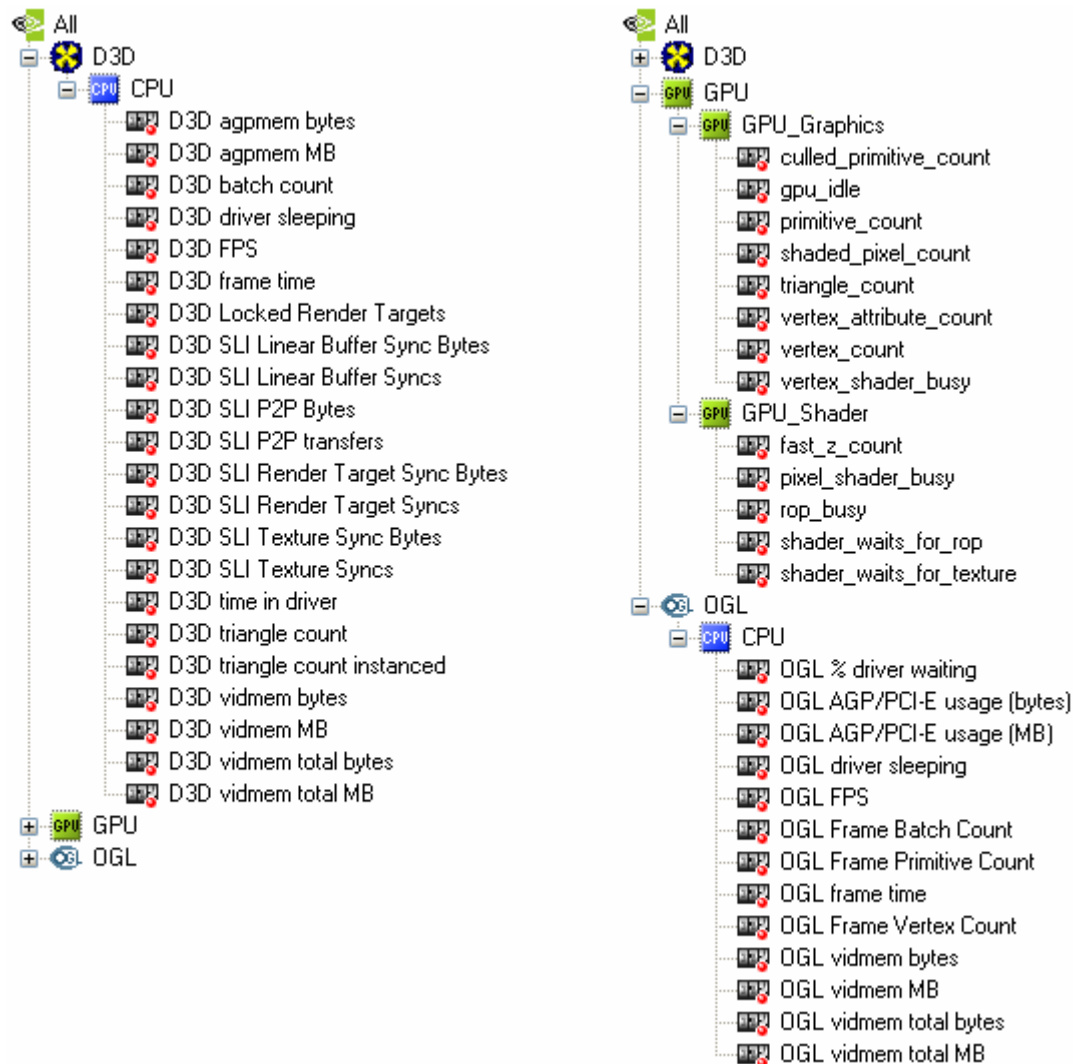
³ Linux port is onvolledig en bevat meer bugs dan de Windows versie. Ontwikkeling hiervan is gestopt

⁴ Source en Makefile/Visual Studio Projectfile beschikbaar

Momenteel is NVPerfKit uitsluitend beschikbaar onder Windows. NVIDIA heeft echter bekend gemaakt dat ontwikkelingen voor Linux in volle gang zijn. Verwacht wordt dat Linux ondersteuning voor in ieder geval een deel van alle features gegeven kan worden in NVPerfKit 2.1.

6.1.1 Geïnstumenteerde NVIDIA driver

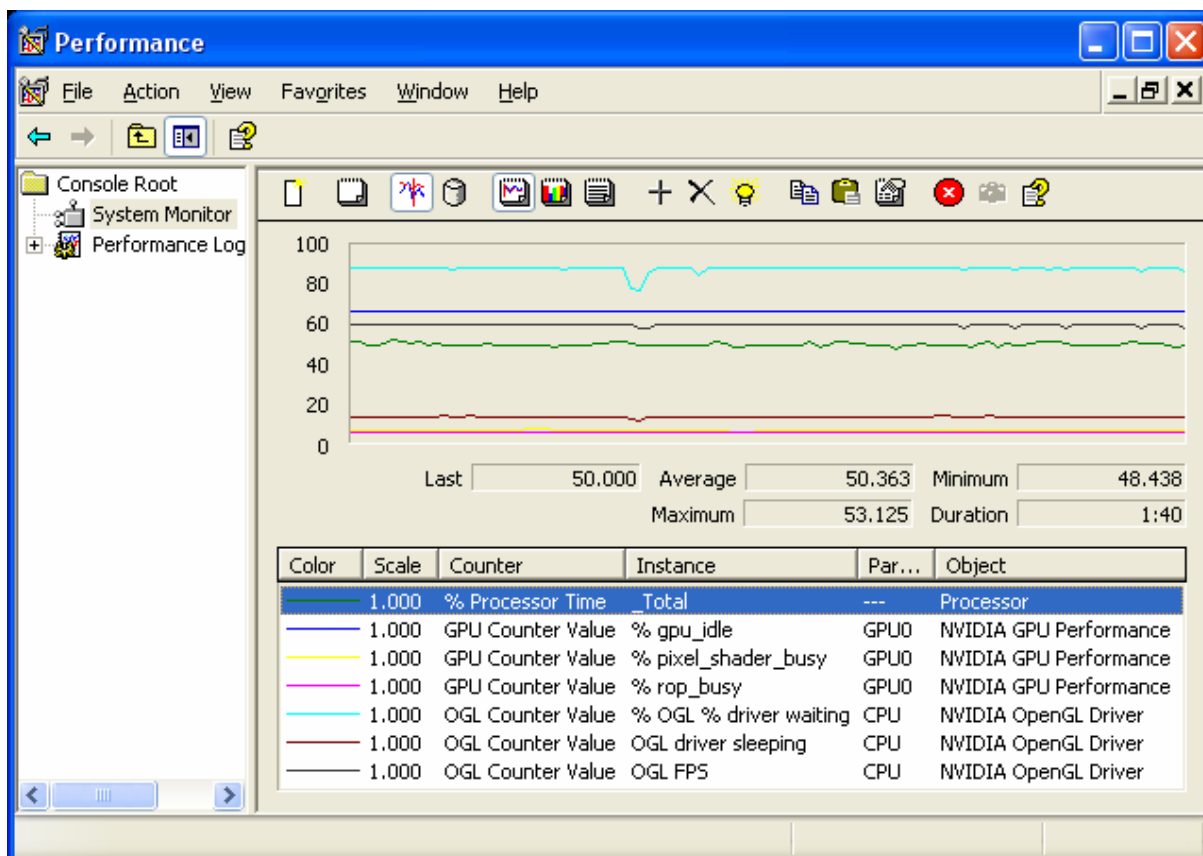
NVIDIA GPUs hebben de mogelijkheid om informatie bij te houden van verschillende stadia in de pipeline. Dit heeft echter wel een kleine impact op de performance van de grafische kaart.



Figuur 22 Beschikbare counters in NVPerfKit 2

Hoe klein deze impact ook is, NVIDIA laat deze bemonstering van de pipeline niet plaats vinden in de reguliere drivers voor de videokaart. Immers, slechts een fractie van de gebruikers van de videokaart is daadwerkelijk geïnteresseerd in deze informatie. Speciaal voor deze groep ontwikkelaars is een speciale geïnstumenteerde driver ontwikkeld waarin deze mogelijkheid geactiveerd kan worden. De counters, die bijgehouden kunnen worden, zijn weergegeven in Figuur 22.

De informatie van de geactiveerde counters zijn vervolgens op verschillende manieren uit te lezen. De gemakkelijkste methode hiervoor is door het gebruik van het standaard Windows programma *perfmon*, dat op elk Windows systeem (Windows NT of later) aanwezig is. Een voorbeeld van hoe deze informatie hierin gevisualiseerd wordt is te zien in Figuur 23. Andere mogelijkheden voor deze visualisatie zijn onder andere Windows PIX, gDEBugger, andere applicaties met een Performance Data Helper (PHD) interface en met behulp van de hieronder beschreven NVPerfSDK.



Figuur 23 NVIDIA performance counters weergegeven in Windows perfmon

6.1.2 NVPerfSDK

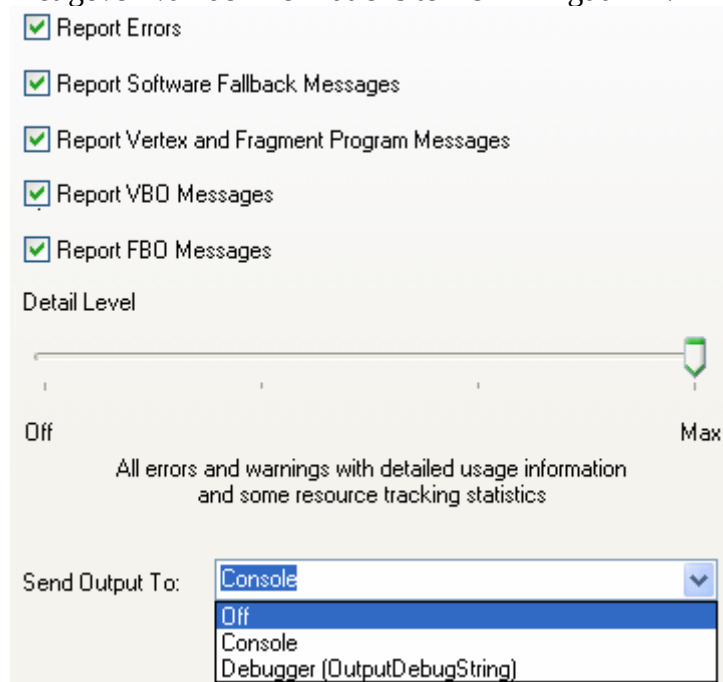
NVPerfSDK biedt de ontwikkelaar in de eerste plaats een API voor de hiervoor genoemde geïntegreerde driver van NVIDIA. Met behulp van deze API kunnen de verschillende counters geactiveerd en gesampled worden. De waarden van deze counters kunnen met dezelfde API uitgelezen worden en door de ontwikkelaar verder in de applicatie of debug output verwerkt worden.

Naast de mogelijkheden voor het werken met counters biedt NVPerfKit ook de mogelijkheid tot zogenaamde 'Simplified Experiments'. Hiermee kunnen 8 locaties in de GPU pipeline worden bemeaten op een 'Speed of Light' en een 'Bottleneck' waarde. Met behulp van de eerstgenoemde kan een percentage worden bepaald van het gebruik van de capaciteit van het betreffende deel. Met de tweede kan het percentage bepaald worden van de tijd dat dit deel de bottleneck van de gehele pipeline vormde.

Ten slotte biedt NVPerfSDK nog de mogelijkheid om alle benodigde operaties voor het uitlezen van al deze counters uit te voeren en eenduidig de naam van de ergste bottleneck te geven. Dit is veruit de gemakkelijkste manier om de ontwikkelaar inzicht te geven in welk deel van de pipeline als eerste voor optimalisatie in aanmerking komt om mogelijke performance problemen aan te pakken.

6.1.3 GLExpert

Dit laatste onderdeel van NVPerfKit is hoofdzakelijk bedoeld voor het debuggen van een OpenGL applicatie. Het is in staat om verscheidene errors, zoals veroorzaakt bij incorrect gebruik van Vertex Buffer Objects (VBO) en Frame Buffer Objects (FBO, zie ook Hoofdstuk 7.1.3), te detecteren en de ontwikkelaar hiervan te waarschuwen. De te rapporteren errors en mogelijkheden van het geven van de informatie is te zien in Figuur 24.



Figuur 24 Debug opties geboden door GLExpert

De door GLExpert gegenereerde foutmeldingen worden voorafgegaan met een identificatie regel met het nummer van de categorie en het bericht. Dit om het parsen van de berichten eenvoudig mogelijk te maken. Vervolgens wordt het basisdeel van het bericht gegeven met daarin generieke informatie over de aard van de melding. Daarna wordt contextspecifieke informatie gegeven. Een voorbeeld van het gehele bericht is hieronder te zien.

```
OGLE: Category: 0x00000010 MessageID: 0x00840000  
The current FBO state (e.g. attachments, texture target) is UNSUPPORTED.  
Reason: COLOR_ATTACHMENT0 attempting to bind to an unsupported texture  
target.
```

6.2 NVShaderPerf

NVShaderPerf is een applicatie, die kwantitatieve performance gegevens kan bepalen uit de broncode van een shader. Een voorbeeld van dergelijke uitvoer is hieronder weergegeven. Als eerste worden de door de compiler gegenereerde instructies getoond. De meest interessante informatie staat echter onderaan. Daar worden de hoeveelheid cycles getoond die in het ergste geval noodzakelijk zijn om alle instructies uit te voeren. Tevens wordt de hoeveelheid gebruikte registers weergegeven. Op basis van deze gegevens wordt tenslotte nog een schatting gemaakt van de doorvoer in pixels per seconde.

```
Compiled using cg.dll.
FP Assembly Code:
!!ARBfp1.0
# cgc version 1.5.0008, build date May 23 2006 10:13:09
# command line args: -q -profile arbfp1 -entry pixelshader -quiet
  -profile fp30
# source file: shader.cg
#vendor NVIDIA Corporation
#version 1.5.0.8
#profile arbfp1
#program alphaCorrectionFP
#semantic alphaCorrectionFP.texture
#var float2 coords : $vin.TEXCOORD0 : TEX0 : 0 : 1
#var sampler2D texture : : texunit 0 : 1 : 1
#var float4 alphaCorrectionFP : $vout.COLOR : COL : -1 : 1
PARAM c[1] = { program.local[0] };
TEMP R0;
TEMP R1;
TEX R0.xyz, fragment.texcoord[0].yxzw, texture[0], 2D;
ADD R0.w, R0.y, -R0.z;
CMP R1.y, -R0.w, R0, R0.z;
ADD R1.x, R0, -R0.z;
CMP R1.x, -R1, R0, R0.z;
ADD R0.w, R0.x, -R0.y;
CMP R0.w, -R0, R1.x, R1.y;
MOV result.color, R0;
END
# 8 instructions, 2 R-regs
----- NV40 -----
Target: GeForce 6800 Ultra (NV40) :: Unified Compiler: v81.95
Cycles: 3.00 :: R Regs Used: 1 :: R Regs Max Index (0 based): 0
Pixel throughput (assuming 1 cycle texture lookup) 2.13 GP/s
```

Gezien de grote diversiteit van GPUs en daarmee hun mogelijkheden, is het resultaat van een programma compilatie mede van het type GPU afhankelijk. NVShaderPerf biedt de mogelijkheid om een selectie te maken uit verschillende architecturen waarop de compilatie en performance metingen worden uitgevoerd. De volledige lijst van op dit moment ondersteunde architecturen zijn hieronder weergegeven. De selectie van te testen architecturen kan met behulp van wildcards gedaan worden, bijvoorbeeld NV4*. Deze worden dan achtereenvolgens getest.

Dat NVShaderPerf uitsluitend voor Windows beschikbaar is, zal weinig bezwaarlijk zijn. De applicatie beschouwt uitsluitend de shader code, die platform onafhankelijk is. Hierdoor kunnen ook shaders gebruikt in applicaties voor bijvoorbeeld Linux getest worden zonder dat de gehele applicatie naar Windows geport hoeft te worden.

```
Chips supported include:  
G70 : GeForce 7800 GT (81.949997)  
G70-GT : GeForce 7800 GTX (81.949997)  
NV44 : GeForce 6200 (81.949997)  
NV43-GT : GeForce 6600 GT (81.949997)  
NV40-12 : GeForce 6800 (81.949997)  
NV40-GT : GeForce 6800 GT (81.949997)  
NV40 : GeForce 6800 Ultra (81.949997)  
NV38 : GeForceFX 5950 Ultra (81.949997)  
NV36 : GeForceFX 5700 Ultra (81.949997)  
NV35 : GeForceFX 5900 Ultra (81.949997)  
NV34 : GeForceFX 5200 Ultra (81.949997)  
NV31 : GeForceFX 5600 Ultra (81.949997)  
NV30 : GeForceFX 5800 Ultra (81.949997)
```

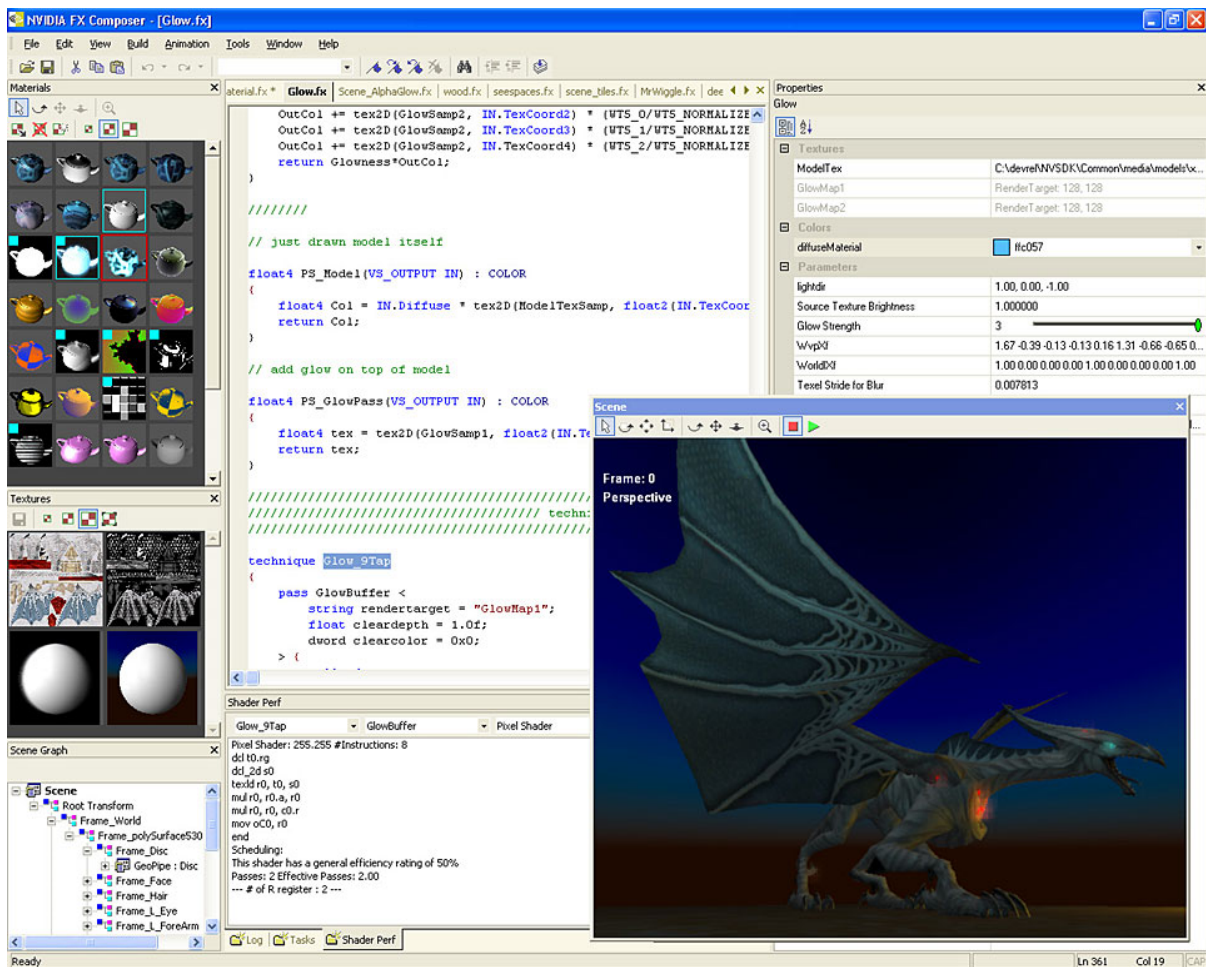
6.3 FX Composer

FX Composer wordt ontwikkeld door NVIDIA ten behoeve van content ontwikkelaars. Zij worden hiermee in staat gesteld om zogenaamde *FX* te ontwikkelen. Deze FX kan men zien als een materiaal of effect dat voor een model gebruikt kan worden. Deze effecten zijn niet equivalent aan een shader, maar bevatten doorgaans wel één of meerdere shaders. In zo'n FX file kunnen dan ook meerdere passes gedefinieerd worden die nodig zijn voor het genereren van het gewenste effect. De FX files zelf zijn gedefinieerd en te gebruiken voor DirectX en gebruiken daardoor HLSL als GPU programmeertaal.

De interface van FX Composer is enigszins vergelijkbaar met die van Visual Studio of soortgelijke IDE's. Een screenshot hiervan is te zien in Figuur 25. Het programma stelt de programmeur in staat om shaders te ontwikkelen en in dezelfde omgeving het resultaat van de shader, toegepast op een gekozen model, direct in te zien. Deze ontwikkelmethode kan zinvol zijn in het geval bijvoorbeeld een beeldverwerkende shader ontwikkeld wordt. Wanneer het echter gaat om een puur GPGPU georiënteerde shader, zal er minder van de functionaliteit van dit programma gebruik gemaakt kunnen worden.

Behalve voor de ontwikkeling van shaders biedt FX Composer ook hulp bij het debuggen hiervan. Methoden zoals genoemd in Hoofdstuk 7.2 worden gemakkelijker gemaakt door het weergeven van textures en tussenresultaten. Dit is met name erg praktisch wanneer gebruik wordt gemaakt van meerdere passes.

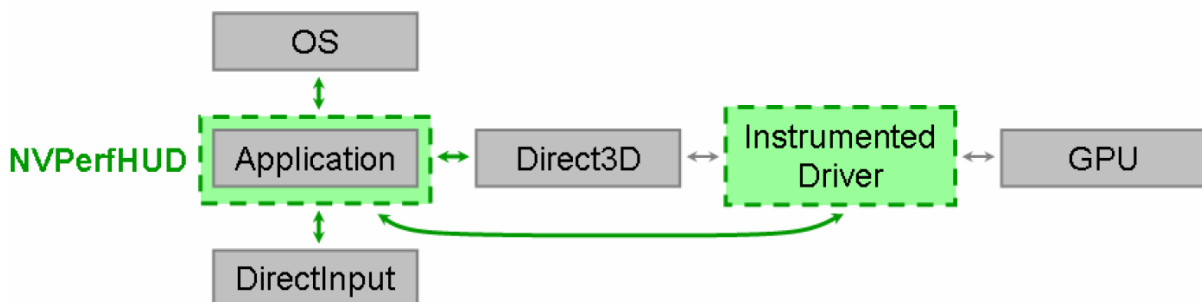
Ten slotte biedt het programma nog enige hulp bij het analyseren van de performance van shaders. Helaas gaat dit niet verder dan het bieden van een grafische interface aan het in Hoofdstuk 6.2 uitgelegde NVShaderPerf.



Figuur 25 Screenshot van FXComposer

6.4 NVPerfHUD

De naam NVPerfHUD is samengesteld uit de afkortingen voor NVIDIA Performance Head-Up Display. NVPerfHUD gedraagt zich dan ook als een soort dashboard bovenop de te analyseren applicatie. In Figuur 26 is te zien hoe NVPerfHUD als een schil om de applicatie met de rest van het systeem communiceert. Hieruit is ook af te lezen dat het uitsluitend beschikbaar is onder Windows en in combinatie met DirectX.



Figuur 26 Interactie tussen NVPerfHUD, het systeem en de applicatie

De applicatie is zeer uitgebreid en duidelijk gericht op gameontwikkelaars. NVPerfHUD kent op dit moment vier verschillende operationele modi, elk ontworpen voor een specifiek doel en met zeer veel mogelijkheden. Belangrijke features hiervan komen in de volgende alinea's aan bod. Zie de gebruikershandleiding [28] voor een volledig overzicht van alle mogelijkheden.

In de eerste instantie start NVPerfHUD op in de *Performance Dashboard* modus. Zoals te zien is in de screenshot hiervan in Figuur 27 worden hierin uitgebreide statistieken vertoond aangaande de performance van de applicatie. Ook is het hierin mogelijk om slimme tests uit te voeren, om snel te bepalen of een bepaalde fase van de grafische pipeline wel of geen bottleneck vormt. Dit zijn onder andere het gebruik van 2x2 textures voor de texture unit en het gebruik van een 1x1 viewport voor de vertex unit.

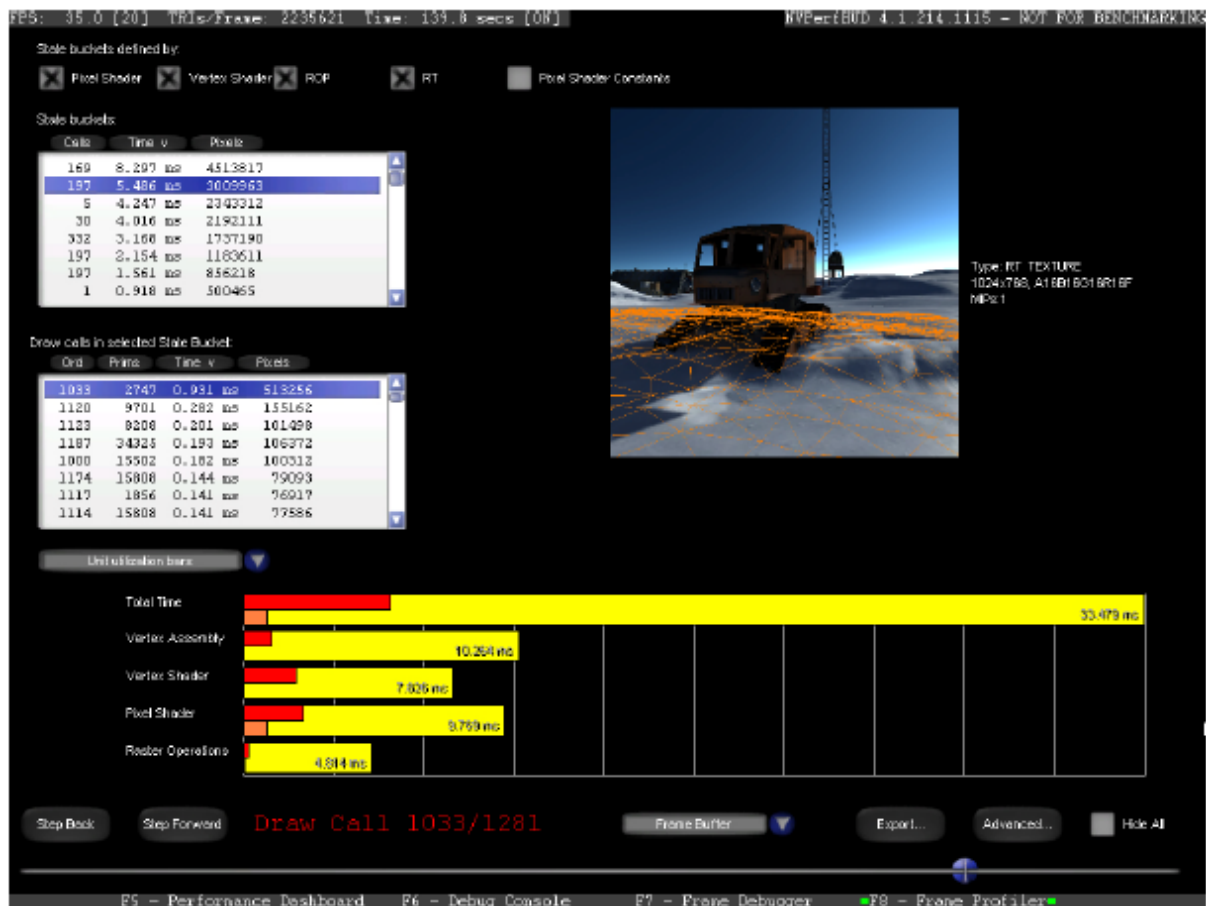


Figuur 27 Screenshot Performance Dashboard van NVPerfHUD

Een tweede modus van NVPerfHUD is de *Debug Console*. Dit is een klein venster dat over de applicatie gelegd wordt waarin standaard alle DirectX debug berichten worden afgebeeld. Door de programmeur zelf gegenereerde debug informatie wordt hier ook in weergegeven wanneer dit gedaan wordt door de functie *OutputDebugString()*.

De derde modus is de *Frame Debugger*. Hiermee kan de uitvoer van de applicatie bevroren worden. Bovendien kan per tekeninstructie door de opbouw van een frame gestapt worden. Op deze wijze kan makkelijk bepaald worden welke instructie verantwoordelijk is voor mogelijk afwijkend gedrag dat geconstateerd wordt. In elke fase kan zeer gedetailleerd de staat van onder andere de rasterizer, de vertex processor en de fragment processor bekeken worden. Hierin is bijvoorbeeld precies te zien wat de shader instructies zijn en wat de hieraan meegegeven constanten en textures zijn. In het geval van de rasterizer wordt informatie geven zoals de huidige staat en instellingen van alpha- stencil- en depth-tests.

De laatste modus van NVPerfHUD is die van *Frame Profiler*. Wanneer deze modus gebruik worden, kunnen nog meer gedetailleerde statistieken getoond worden van een grote hoeveelheid counters die in de GPU bijgehouden worden. Deze gegevens zijn afkomstig van de geïnstrumenteerde driver zoals beschreven in Hoofdstuk 6.1.1. Aan de hand hiervan kan nauwkeurig bepaald worden wat de bottleneck van de applicatie is.



Figuur 28 Screenshot Frame Profiler van NVPerfHUD

Om te voorkomen dat onbevoegden toegang kunnen krijgen tot de zeer uitgebreide debug en performance gegevens, moet de mogelijkheid voor het gebruik van NVPerfHUD in de broncode van de applicatie expliciet geactiveerd worden. Dit moet bij het aanmaken van een Direct3D device aangegeven worden. Een herbruikbaar en eenvoudig aan of uit te zetten methode gegeven door NVIDIA is te zien in Code fragment 3. Wanneer de applicatie geactiveerd is voor het gebruik van NVPerfHUD, kan dit eenvoudig hierin gestart worden door de executable van de applicatie als parameter aan de NVPerfHUD executable mee te geven.

```
// Set default settings
UINT AdapterToUse=D3DADAPTER_DEFAULT;
D3DDEVTYPE DeviceType=D3DDEVTYPE_HAL;

#ifdef ACTIVATE_NVPERFHUDD
// Look for 'NVIDIA NVPerfHUD' adapter
// If it is present, override default settings

    for (UINT Adapter=0;Adapter<g_pD3D->GetAdapterCount();Adapter++)
    {
        D3DADAPTER_IDENTIFIER9 Identifier;
        HRESULT Res;
        Res = g_pD3D->GetAdapterIdentifier(Adapter,0,&Identifier);
        if (strcmp(Identifier.Description,"NVIDIA NVPerfHUD") == 0)
        {
            AdapterToUse=Adapter;
            DeviceType=D3DDEVTYPE_REF;
            break;
        }
    }
#endif

// Create the device using the selected adapter and device type
g_pD3D->CreateDevice( AdapterToUse, DeviceType, hwnd,
                    D3DCREATE_HARDWARE_VERTEXPROCESSING,
                    &d3dpp, &g_pd3dDevice);
```

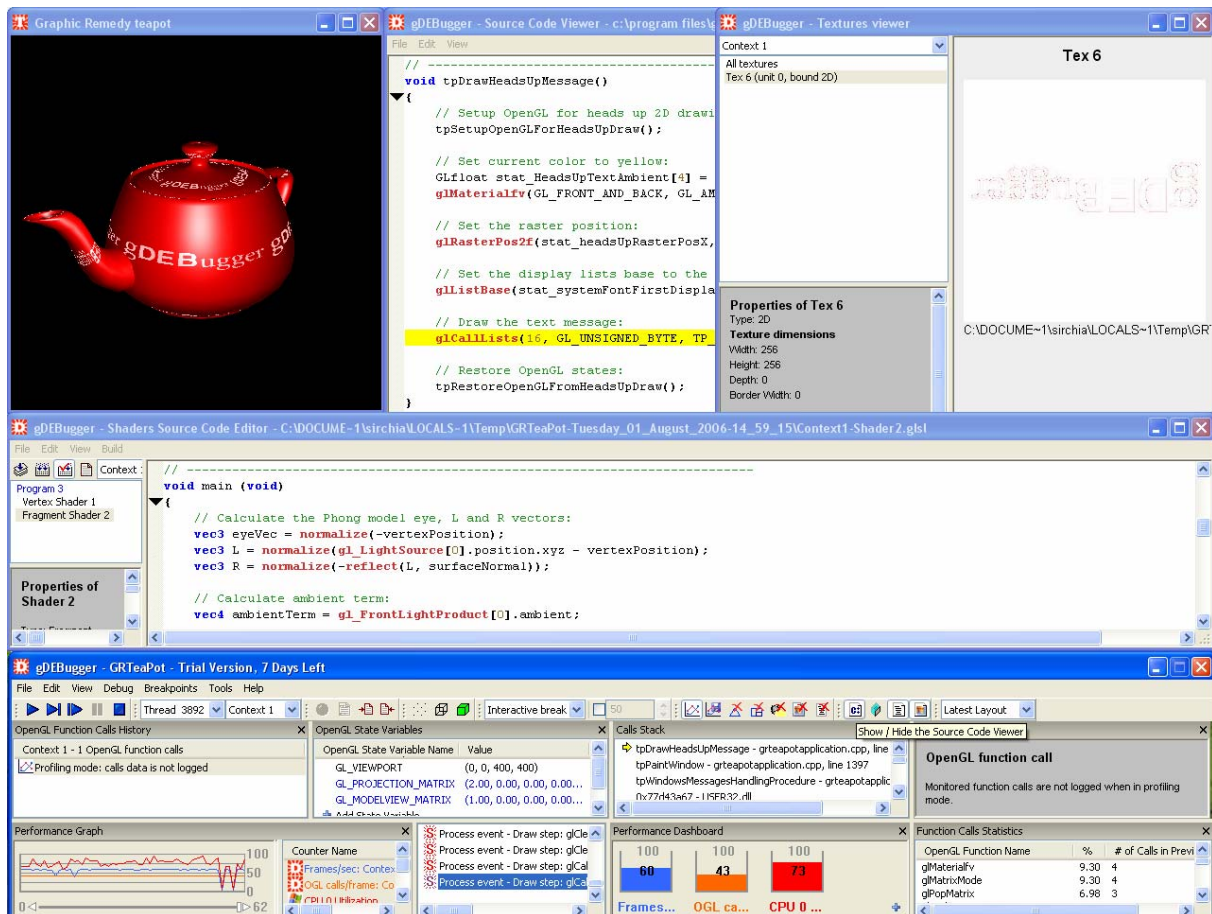
Code fragment 3 Door NVIDIA aangeraden opt-in mechanisme van NVPerfHUD

NVPerfHUD biedt veruit de meeste mogelijkheden van alle beschikbare debug en profiling tools. Hiermee kan men grafische applicaties op alle mogelijke wijzen debuggen en profileren. De geboden debug-mogelijkheden kunnen tekort schieten wanneer een GPGPU applicatie ontwikkeld wordt en de data weergegeven als afbeelding weinig of geen zinnige informatie naar de ontwikkelaar uitdraagt. In zo'n geval kunnen echter wel de methoden beschreven in Hoofdstuk 7.2 gebruikt worden, en kan de verkregen resultaten afgebeeld worden in de *Debug Console* mode van het programma.

6.5 gDEBugger

gDEBugger is een commercieel product dat ontwikkeld wordt door Graphic Remedy. Het bedrijf werkt hierin nauw samen met NVIDIA waardoor al erg vroeg ondersteuning voor het uitlezen van counters van de geïnstrumenteerde driver zoals vermeldt in Hoofdstuk 6.1.1 geboden kon worden. De applicatie is zoals de naam doet vermoeden in de eerste plaats een debugger. Het biedt echter ook verscheidene mogelijkheden voor het profileren van een applicatie.

In Figuur 29 zijn in één screenshot een groot aantal features, die geboden wordt door gDEBugger weergegeven. De applicatie, die onderzocht wordt, is in het venster linksboven te zien. De GUI van gDEBugger bestaat hoofdzakelijk uit het actieve scherm dat onderin weergegeven wordt. Hierin zijn op dit moment alle mogelijke opties weergegeven. Het is echter wel mogelijk om hierin een selectie te maken, bijvoorbeeld wanneer de werkzaamheden zich vooral richten op het debuggen dan wel profileren van de applicatie.



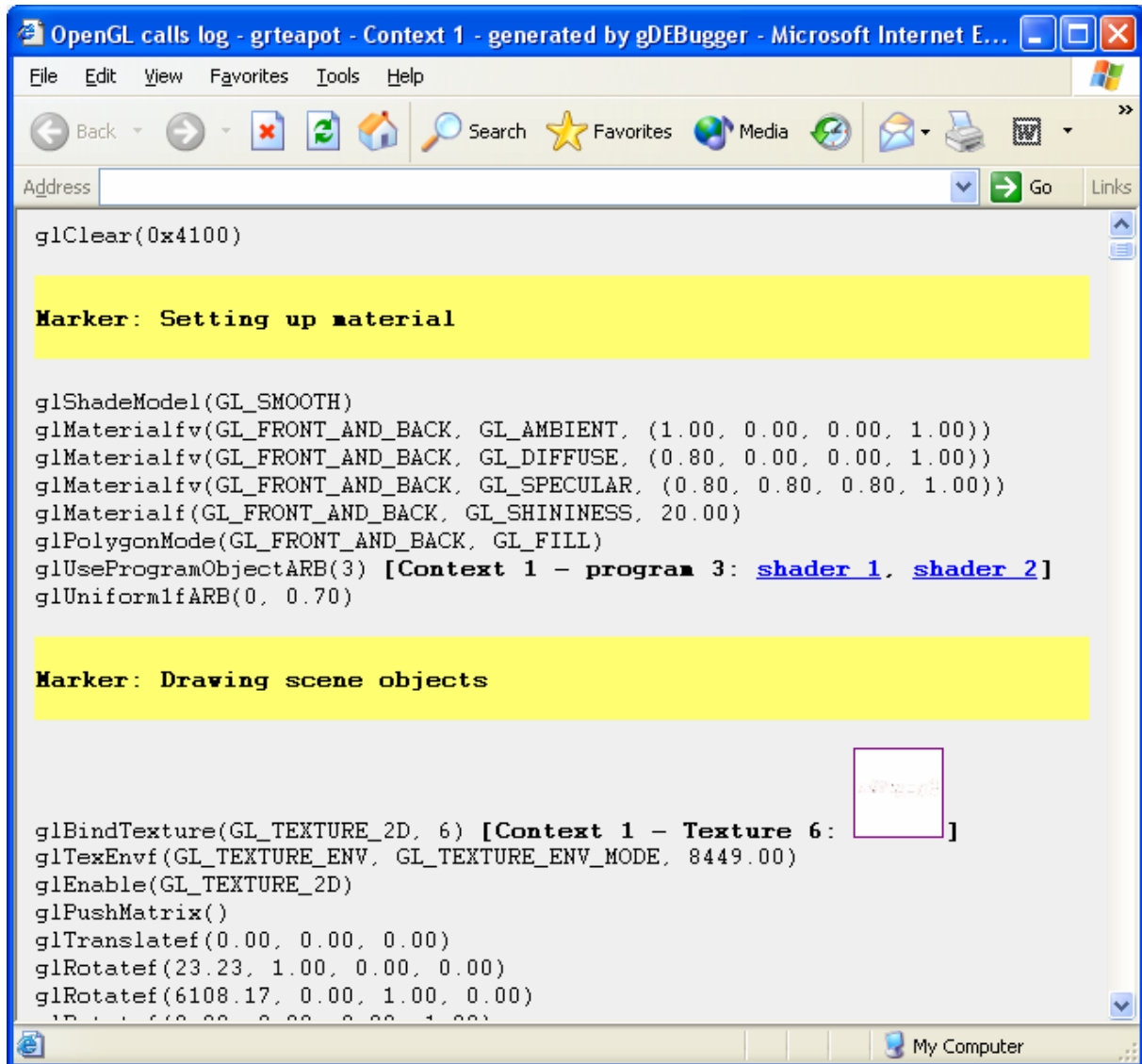
Figuur 29 Screenshot van gDEBugger en bijbehorende viewers

De debug mogelijkheden van gDEBugger zijn vrij goed vergelijkbaar met een reguliere ‘CPU’ debugger. Deze zijn echter specifiek gericht op de OpenGL API. Zo kan de historie van aangeroepen OpenGL functies worden bijgehouden en is het mogelijk om de waarde van een grote hoeveelheid OpenGL state variables bij te houden. Het uitvoeren van de applicatie kan handmatig door de gebruiker gepauzeerd worden. Daarnaast kunnen er breakpoints gedefinieerd worden op een willekeurige set van OpenGL commando’s of in geval een OpenGL error geconstateerd wordt. Tenslotte is het nog mogelijk om de applicatie vertraagd af te laten spelen of zelfs handmatig per tekenopdracht hier doorheen te lopen. Dit alles kan, indien beschikbaar, in relatie gebracht worden met de broncode van de applicatie, te zien midden-boven in Figuur 29.

Naast de OpenGL functies kunnen ook gebruikte shaders onder de loep genomen worden. Voorwaarde hiervoor lijkt echter wel te zijn dat gebruik wordt gemaakt van GLSL shaders. Het is mij niet gelukt om bijvoorbeeld gebruikte Cg shaders weer te geven. GLSL shaders worden overzichtelijk per programma (set van vertex en/of fragment shader) weergegeven. Hiervan kan de broncode worden ingezien. Verder is zichtbaar welke waarden er op dat moment aan de uniforme parameters zijn toegekend. Het is zelfs mogelijk om wijzigingen in de broncode aan te brengen, dit te compileren en te gebruiken zonder de applicatie opnieuw te hoeven starten.

Verder kunnen alle op dat moment gedefinieerde en gebruikte textures, alsmede alle attributen hiervan, worden bekeken. Dit is rechtsboven in Figuur 29 te zien.

Naast de mogelijkheid om 'live' deze gegevens in te zien, geeft gDEBugger ook de mogelijkheid om de gegevens in een HTML log bestand op te slaan. Hierin zijn, zoals in Figuur 30 te zien is, alle OpenGL instructies opgeslagen. Alle gebruikte shaders zijn als hyperlinks weergegeven. Een klik hierop toont de broncode van de betreffende shader. De gebruikte textures worden in thumbnail vorm in het logbestand weergegeven. De volledige texture in oorspronkelijk formaat is te zien door hierop te klikken.



Figuur 30 HTML log bestand gegenereerd door gDEBugger

Naast al deze debug mogelijkheden biedt het programma ook nog uitgebreide mogelijkheden om een applicatie te profileren. Mogelijkheden hiertoe bevinden zich voornamelijk in het onderste gedeelte van GUI in Figuur 29. Er kunnen hierin alle NVIDIA specifieke counters van de geïnstrumenteerde drivers bijgehouden worden, alsmede Windows gerelateerde counters zoals CPU en geheugen gebruik. Verder worden door gDEBugger zelf nog enkele counters aangeboden, die informatie per context bijhouden. Hiervoor is momenteel de keuze tussen frames per seconde, OpenGL aanroepen per frame, de hoeveelheid gebruikte textures en de hoeveelheid verwerkte texels. De verzamelde gegevens worden zowel in een dashboard getoond, als in een lopende grafiek. De eerste is een staafdiagram met de huidige waarde en het maximum. In de tweede is het verloop met geschiedenis zichtbaar.

Verder wordt er per frame bijgehouden hoe vaak welke OpenGL functies aangeroepen worden. Ook is zichtbaar hoeveel procent van de totale aanroepen door die functie wordt opgemaakt. Op deze wijze kunnen bijvoorbeeld redundante functie aanroepen worden geïdentificeerd.

Naast als deze statistieken biedt gDEBugger nog de mogelijkheid om enkele eenvoudige tests uit te voeren voor het bepalen van een mogelijke bottleneck door het uitschakelen van delen van de pipeline. De eerste mogelijkheid hiervoor is het elimineren alle tekencommando's (*glVertex*, *glDrawElements*, *glDrawPixels*, etc). Wanneer in dit geval de performance niet significant toeneemt, vormt de CPU of de bus de bottleneck. Deze hebben immers als enige geen verminderde last gekregen. Op soortgelijke wijze kan de bottleneck analyse vergemakkelijkt worden door het gebruik van een 1x1 pixel viewport, het uitschakelen van standaard OpenGL belichting, geforceerd gebruik van 2x2 textures en het vervangen van alle fragment shaders door een zeer simpel exemplaar. Zie Hoofdstuk 7.1.1 voor hulp bij het analyseren van de hierdoor verkregen informatie.

Met gDEBugger beschikt de ontwikkelaar over een schat aan informatie over zijn applicatie. Het biedt zeer uitgebreide debug en profileer mogelijkheden. Een nadeel is echter dat de applicatie momenteel uitsluitend voor Windows beschikbaar is. De ontwikkelaars hebben echter laten weten dat een nieuwe versie met Linux ondersteuning in ontwikkeling is en uiterlijk begin 2007 uitgebracht zal worden.

6.6 GLIntercept

GLIntercept is een toolkit, die geïmplementeerd is als extra laag tussen de applicatie en OpenGL. De interface wordt gedaan door de applicatie gebruik te laten maken van de meegeleverde *opengl32.dll* als OpenGL bibliotheek. Deze stelt GLIntercept in staat om de OpenGL API aanroepen op te vangen, en hiervan gegevens te loggen alvorens de werkelijke commando's uitgevoerd worden.

Met behulp van een *gliConfig.ini* file, waarvan verscheidene voorbeelden zijn meegeleverd, kunnen de verscheidene debug en log mogelijkheden aan- dan wel uitgezet worden. De gegenereerde logbestanden kunnen zowel in een tekst bestand als in een XML bestand opgeslagen worden. De laatste vorm lijkt sterk op de log mogelijkheden van gDEBugger. Een voorbeeld hiervan is te zien in Figuur 31. Wanneer gekozen wordt om ook afbeeldingen, zoals textures of de inhoud van het framebuffer, en het gebruik van shaders te loggen, worden deze ook als hyperlinks gerepresenteerd.

Bij de log mogelijkheden hoort onder andere het bijhouden van alle OpenGL commando's met hun argumenten. Hiervan kan ook in microseconden nauwkeurig worden bijgehouden hoe lang deze commando's nodig hadden om uitgevoerd te worden. Hierbij moet wel rekening gehouden worden met kleine onnauwkeurigheden doordat GLIntercept zelf een kleine overhead toevoegt.

Verder kunnen afbeeldingen van alle gebruikte textures en de inhoud van het framebuffer opgeslagen worden. Voor deze laatste wordt de mogelijkheid geboden om het framebuffer voor en na een tekenoperatie op te slaan en zelfs om de verschillen tussen beide afbeeldingen op te slaan. Verder kan ook de inhoud van het z-buffer en stencil-buffer opgeslagen worden.

```

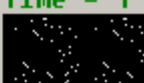
glBindTexture(GL_TEXTURE_2D,1); Time = 4 us
glCopyTexSubImage2D(GL_TEXTURE_2D,0,0,0,0,0,512,512); Time = 74 us
glGetError(); = GL_NO_ERROR Time = 2 us
glGetError(); = GL_NO_ERROR Time = 0 us
glBindProgramARB(GL_FRAGMENT_PROGRAM_ARB,1(src1)); Time = 3 us
glGetError(); = GL_NO_ERROR Time = 0 us
glEnable(GL_FRAGMENT_PROGRAM_ARB); Time = 2 us
glGetIntegerv(GL_ACTIVE_TEXTURE,0x12fe88); Time = 4 us
glActiveTextureARB(GL_TEXTURE0); Time = 1 us

glBindTexture(GL_TEXTURE_2D,1); Time = 2 us
glActiveTextureARB(GL_TEXTURE0); Time = 0 us

glBegin(GL_QUADS); Shaders=(FP1    Color(); Time = 13 us
glTexCoord2f(0.000000,0.000000); Time = 2 us
glVertex3f(-1.000000,-1.000000,-0.500000); Time = 8 us
glTexCoord2f(1.000000,0.000000); Time = 0 us
glVertex3f(1.000000,-1.000000,-0.500000); Time = 0 us
glTexCoord2f(1.000000,1.000000); Time = 0 us
glVertex3f(1.000000,1.000000,-0.500000); Time = 0 us
glTexCoord2f(0.000000,1.000000); Time = 0 us
glVertex3f(-1.000000,1.000000,-0.500000); Time = 0 us
glEnd(); Time = 4 us
glDisable(GL_FRAGMENT_PROGRAM_ARB); Time = 2 us

glBindTexture(GL_TEXTURE_2D,1); Time = 4 us
glCopyTexSubImage2D(GL_TEXTURE_2D,0,0,0,0,0,512,512); Time = 72 us
glViewport(0,0,512,512); Time = 14 us

glBindTexture(GL_TEXTURE_2D,1); Time = 3 us
glEnable(GL_TEXTURE_2D); Time = 1 us

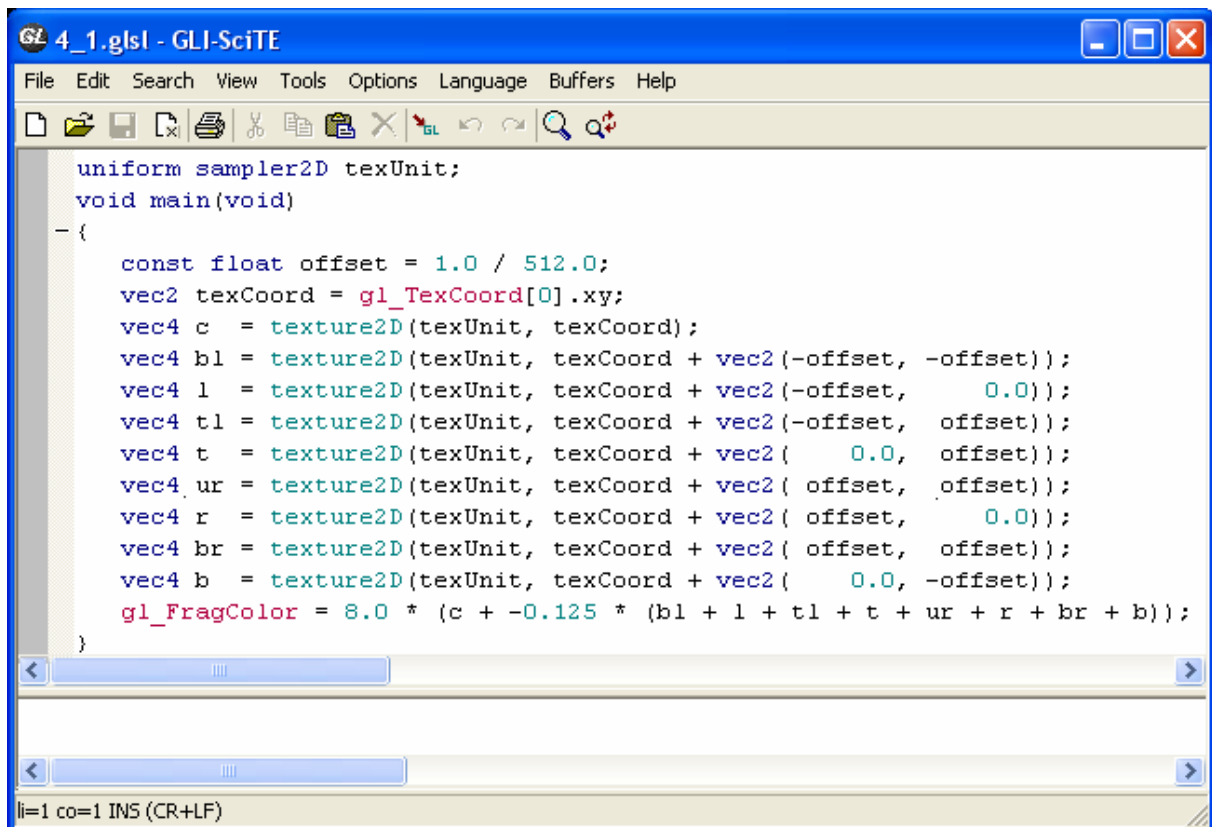
glBegin(GL_QUADS); Color( ); Time = 13 us

```

Figuur 31 Voorbeeld XML output van GLIntercept

Naast de uitgebreide ingebouwde log mogelijkheden biedt de architectuur van GLIntercept ook mogelijkheid voor het toevoegen van plugins. Het gebruik hiervan kan in dezelfde *gliConfig.ini* file geregeld worden. Standaard worden er al verscheidene plugins zoals de mogelijkheid om de hoeveelheid aanroepen per OpenGL functie bij te houden, en een runtime shader editor. Deze laatste is de meest noemenswaardige. Een screenshot hiervan is te zien in Figuur 32. Met deze editor kan op runtime een lijst met gebruikte shaders afgebeeld worden, die afzonderlijk te bekijken, bewerken en zelfs te compileren en te gebruiken zijn. Hierdoor kan het resultaat van een wijziging in de shader direct zichtbaar gemaakt worden zonder dat de applicatie opnieuw opgestart hoeft te worden.

In deze applicatie is het, in tegenstelling tot gDEBugger, wel mogelijk om Cg shaders te detecteren en weer te geven. Wel moet hierbij rekening gehouden worden dat de Cg shaders niet in de vorm van Cg broncode, maar in gecompileerde vorm met op Assembler lijkende instructies aan OpenGL aangeboden wordt. Dit is dan ook de enige vorm waarover GLIntercept in deze editor kan beschikken.



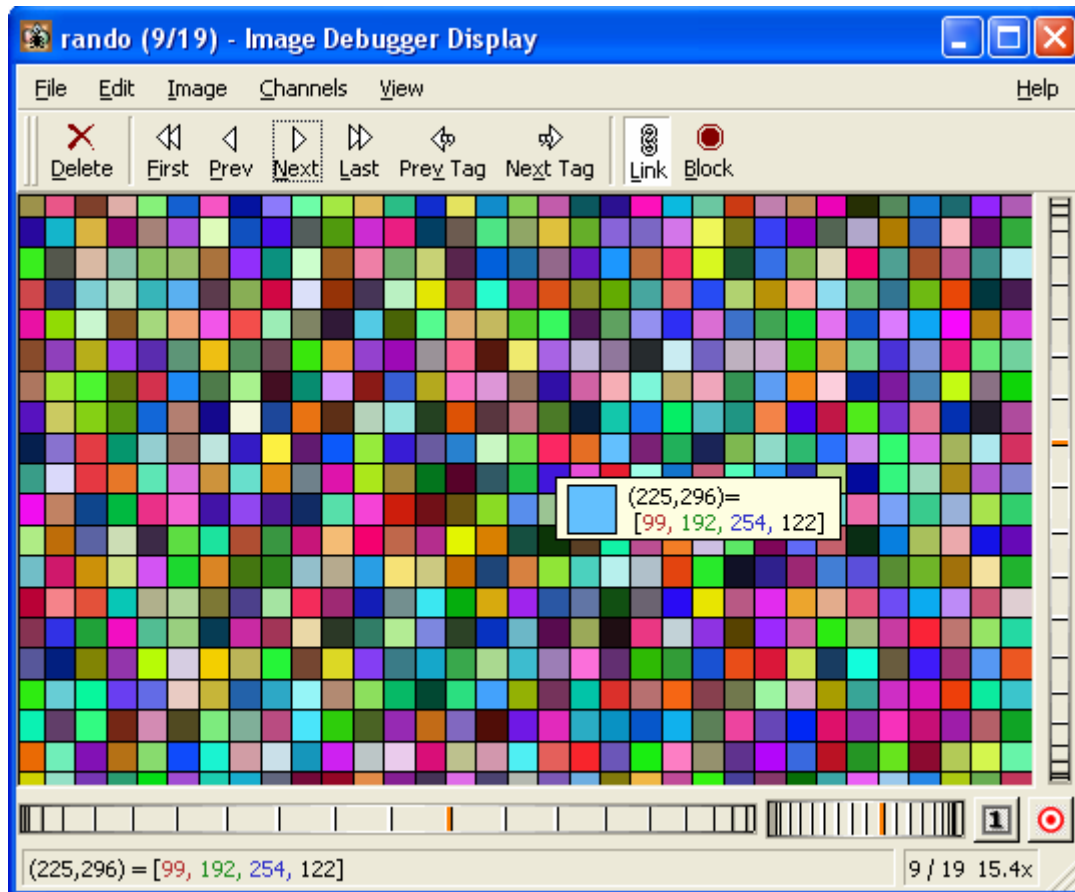
Figuur 32 Screenshot van de runtime shader editor plugin van GLIntercept

Van GLIntercept is de broncode vrijgegeven wat het mogelijk maakt om desgewenst zelf aanpassingen te maken en belangrijker te kunnen compileren voor andere platformen dan Windows. Volgens de readme moet het bijvoorbeeld mogelijk zijn om de toolkit te compileren en te gebruiken onder Linux. Hiervoor is helaas geen makefile of iets dergelijks aanwezig en kan dus een project op zich worden voor de ontwikkelaar.

6.7 IMDebug

In veel gevallen wil een ontwikkelaar de waarde van een variabele als debug output bekijken om aan de hand daarvan conclusies te trekken over de oorzaak van een bepaalde fout. In het geval de data een afbeelding of een andere vorm van een array met veel data betreft, is deze werkwijze een stuk lastiger. Bij het geven van een afbeelding als debug output komt normaliter veel meer kijken dan bijvoorbeeld een *printf* commando. IMDebug probeert de ontwikkelaar hierin tegemoet te komen.

Met behulp van de IMDebug bibliotheek kan met één commando de inhoud van een array visueel worden weergegeven. Hiervoor wordt een externe display applicatie gestart waarin alle debug output afgebeeld wordt. Een screenshot van deze applicatie is te zien in Figuur 33. In dit venster kan de afbeelding op verscheidene manieren geanalyseerd worden. Zo worden er mogelijkheden geboden om de waarden te schalen of een bias te geven, en kunnen de exacte pixelwaarden onder andere met behulp van een tooltip worden weergegeven. Verder kunnen de afzonderlijke kleurkanalen geïsoleerd worden en kan er vrij op de afbeelding ingezoomd en geschoven worden.



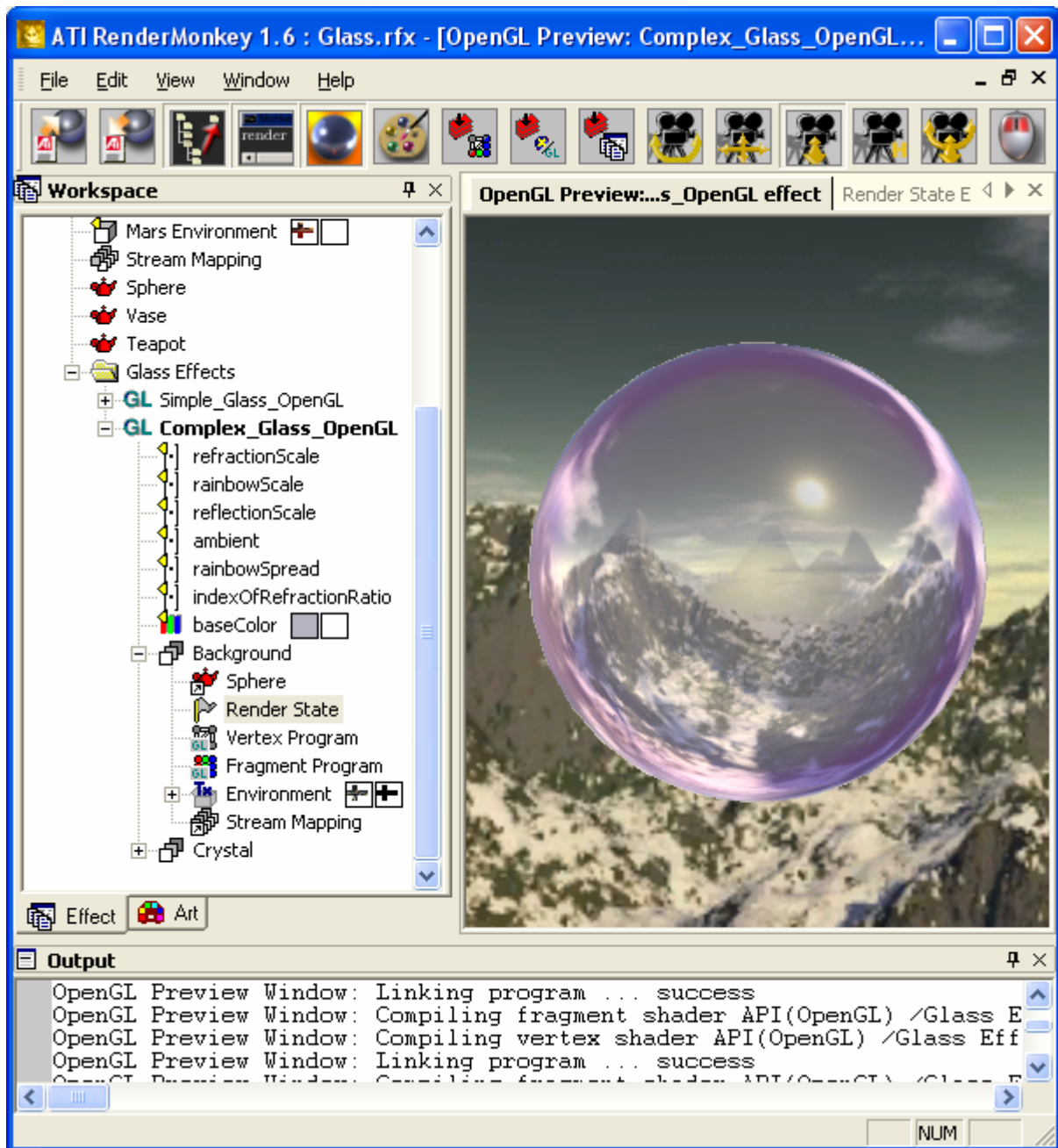
Figuur 33 Afbeeldinganalyse met IMDebug

Net als voor GLIntercept geldt dat de applicatie voor Windows is ontwikkeld. Er is echter wel broncode beschikbaar en voor de GUI is gebruik gemaakt van de platform onafhankelijke FOX Toolkit. Het moet dus mogelijk zijn de applicatie ook onder Linux te gebruiken, maar dit porten wordt tot nu toe wederom aan de ontwikkelaar zelf over gelaten.

6.8 Rendermonkey

ATI's Rendermonkey is een ontwikkelomgeving, die enigszins vergelijkbaar is met FX Composer van NVIDIA. Beide applicaties stellen de ontwikkelaar namelijk in staat een effect bestaande uit één of meer shaders te beschrijven en te ontwikkelen.

Één van de belangrijkste verschillen is echter dat Rendermonkey naast HLSL ook het bewerken van GLSL shaders toelaat. Daarbij kunnen ook instellingen van de OpenGL omgeving beïnvloed worden. Hiermee wordt de mogelijkheid gecreëerd om als shader ontwikkelomgeving voor generieke OpenGL toepassingen te gebruiken. Wel moet hierbij opgemerkt worden dat het niet mogelijk is om code te laten genereren, die direct in een OpenGL applicatie geplaatst kan worden. Van het gehele effect zijn uitsluitend de shader programma's zelf rechtstreeks te gebruiken door deze te kopiëren. Verdere OpenGL omgevingsvariabelen, textures en shader parameters moeten door de programmeur zelf in de applicatie overgenomen worden van het ontworpen effect.



Figuur 34 Screenshot van Rendermonkey

Ondanks dit gemis kan Rendermonkey een nuttige tool zijn in het ontwikkelproces van een GPU algoritme. Het biedt namelijk, zeker in vergelijking met andere soortgelijke applicaties, veel mogelijkheden voor het beïnvloeden van (vrijwel) alle omgevingsvariabelen van OpenGL. Daarnaast zijn er nog verscheidene voorgedefinieerde attributen aan shaders mee te geven, zoals een tijdsafhankelijke waarde, positie van de camera of de hoeveelste pass op dat moment uitgevoerd wordt.

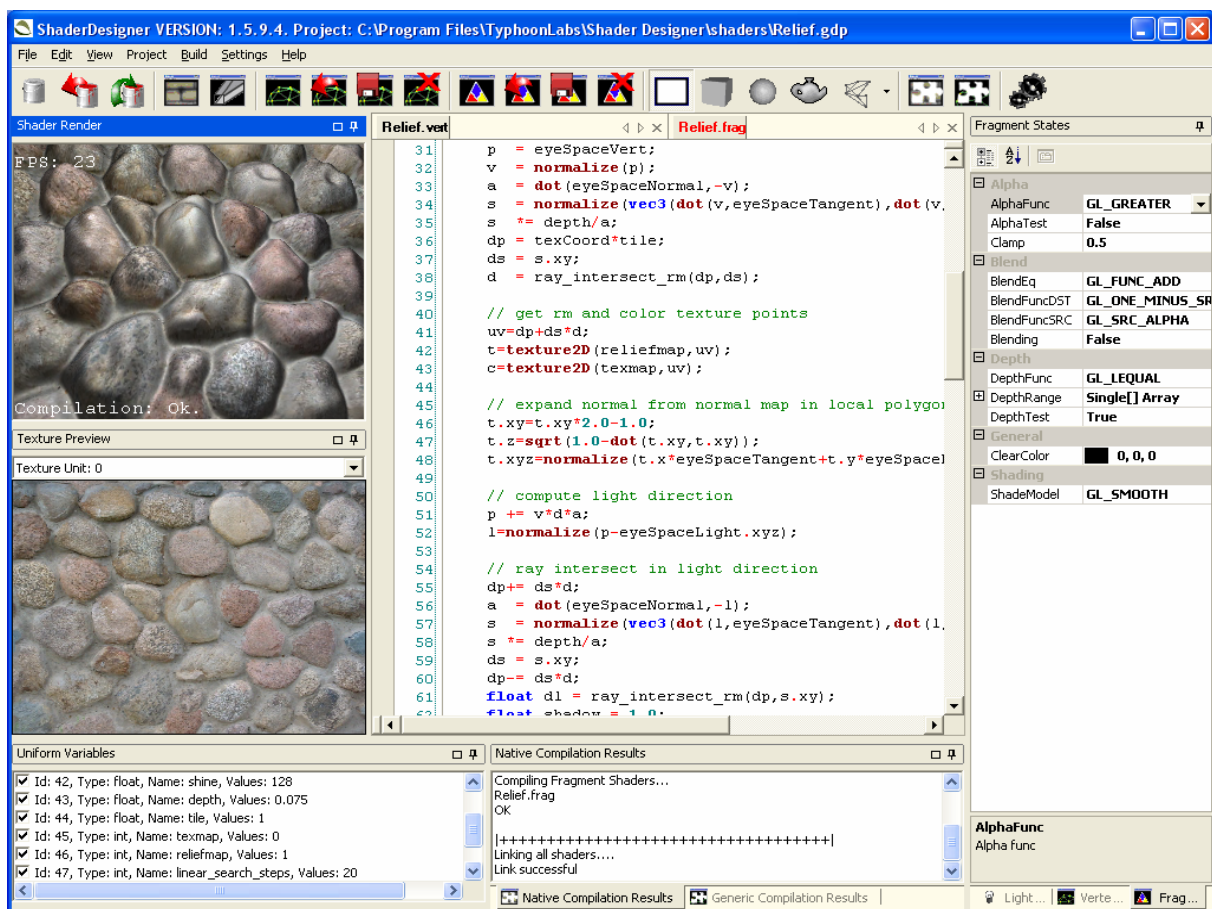
Een ander sterk punt is dat het makkelijk mogelijk is om algoritmen bestaande uit meerdere passes te definiëren. Hierbij kunnen desgewenst render-to-texture operaties zoals beschreven in Hoofdstuk 7.1.3 toegepast worden. Voor debug doeleinden is het mogelijk om deze tussenstappen ook naar het scherm te laten renderen, zodat eventuele problemen geïdentificeerd kunnen worden.

Rendermonkey is opgezet met een plugin architectuur. Hierdoor is het mogelijk nieuwe functionaliteit toe te voegen zonder de gehele applicatie opnieuw te compileren. Dit betekent ook dat door derden gemaakte toevoegingen, zoals model of texture laders of generatoren, in het programma gebruikt kunnen worden. Hiervoor is een plugin SDK beschikbaar gesteld, die ook gebruikt is voor het ontwikkelen van alle reeds beschikbare functionaliteit. Op dit moment zijn er echter nog vrij weinig additionele plugins beschikbaar. Een zoektocht op het internet levert slechts enkele export plugins naar effecten in andere formaten, toolkits of applicaties op, zoals Maya en de open source game engine OGRE.

Rendermonkey is uitsluitend te gebruiken onder Windows. Gezien het feit dat de belangrijkste doelgroep wordt gevormd door content ontwikkelaars in de spelindustrie, zal hier naar alle waarschijnlijkheid geen verandering in komen. Rendermonkey hoeft echter niet gelijktijdig met de beoogde doelapplicatie gebruikt te worden. Het kan in plaats daarvan efficiënt gebruikt worden om snel een prototype van een GPU algoritme te ontwikkelen. De hieruit ontstane GLSL shaders kunnen direct toegepast worden in de doelapplicatie.

6.9 Shader Designer

Dit programma van Typhoon Labs probeert wat functionaliteit in de buurt van ATI's Rendermoney te komen. Het is ook een ontwikkelomgeving voor GLSL shaders waarbij ook alle overige omgevingsvariabelen van OpenGL in te stellen zijn. Toch schiet het programma nog op enkele punten tekort ten opzichte van Rendermonkey.



Figuur 35 Screenshot van Shader Designer

Een van de belangrijkste tekortkomingen is de onmogelijkheid van het definiëren van meerdere renderpasses. Dit maakt het onmogelijk om met dit programma complexe algoritmen, bestaande uit meerdere fragment shaders, te implementeren. Daarnaast ontbreekt de mogelijkheid om zelfgemaakte objecten in te laden. Hiervoor kan uitsluitend gebruik worden gemaakt van de bij het programma meegeleverde modellen.

Net als Rendermonkey biedt Shader Designer ook niet de mogelijkheid om het opgezette prototype direct om te zetten naar een blok OpenGL code. Ook hiervoor geldt dat uitsluitend de GLSL shaders gekopieerd kunnen worden en dat de rest van de instellingen aan de OpenGL omgeving handmatig overgenomen moet worden.

Een voordeel van Shader Designer is dat het wel beschikbaar is onder Linux. De ontwikkelaars hebben echter toegegeven dat er hierin nog een groot aantal, niet nader gespecificeerde problemen zitten. Verder hebben zij de ontwikkelingen aan deze Linux versie gestopt. De broncode van het programma is zonder verdere opgaaf van redenen niet vrijgegeven, ondanks dat dit voor de Windows versie wel gedaan is.

6.10 GLTrace

GLTrace is zeer vergelijkbaar met GLIntercept. Het bestaat uit een vervangende *opengl32.dll* en een *ini* file voor configuratie van de mogelijkheden. Het is hierin echter een stuk beperkter dan GLIntercept. Het biedt enkel de mogelijkheid om alle OpenGL functie aanroepen en hun argumenten te loggen in een tekst bestand. Aan het eind worden van deze aanroepen nog het totale aantal aanroepen per functie genoteerd.

Het enige voordeel dat GLTrace heeft ten opzichte van GLIntercept, is dat de broncode gepaard gaat met een *readme* met instructies en een Linux *makefile*. Het is hiermee dus zeker mogelijk om deze applicatie te porten naar Linux.

6.11 BuGLE

BuGLE is een debug toolkit voor het analyseren van een OpenGL applicatie. Ook deze toolkit werkt door het afvangen van OpenGL functies, en de mogelijkheid hieraan acties te koppelen. Hiertoe wordt het programma geleverd met een aantal zogenaamde filters zoals het verzamelen van statistieken en het loggen van commando's. Een unieke mogelijkheid hierbij is het bijhouden van de gebruikte OpenGL extensies.

Naast de genoemde logmogelijkheden biedt BuGLE de mogelijkheid om screenshots en zelfs een video opname te maken van de inhoud van de framebuffer. Ook kan de output zelf beïnvloed worden door uitsluitend te laten renderen in *wireframe* mode of uitsluitend de *frontbuffer* te gebruiken. Met deze laatste optie wordt ook de opbouw van de scene direct zichtbaar voor de ontwikkelaar.

Naast deze filters wordt ook een naar *gdb* gemodelleerde debugger bijgeleverd. Deze applicatie, *gldb* genaamd, is speciaal toegespitst op het debuggen van grafische applicaties. Dit wordt gerealiseerd door de debugger de mogelijkheid te geven breakpoints in te stellen op OpenGL functies. Ook kan de debugger de applicatie automatisch onderbreken wanneer een OpenGL error geconstateerd is. Wanneer de applicatie onderbroken wordt door één van deze mogelijkheden of door gebruikelijke interventie van de ontwikkelaar, kan tevens inzicht gegeven worden in de huidige OpenGL staat. De inhoud van alle variabelen die deze staat opmaken kunnen dan in de debugger zichtbaar gemaakt worden.

Verrassend is dat BuGLE als enige puur ontwikkeld is voor het gebruik in Linux. De ontwikkelaars geven dan ook geen enkele support voor het gebruik van de applicatie onder Windows. De broncode van de toolkit is echter wel beschikbaar. Daarmee is het dus mogelijk om BuGLE naar Windows te porten, maar dit zal naar verwachting geen eenvoudige klus worden.

7 Performance en Debugging tips

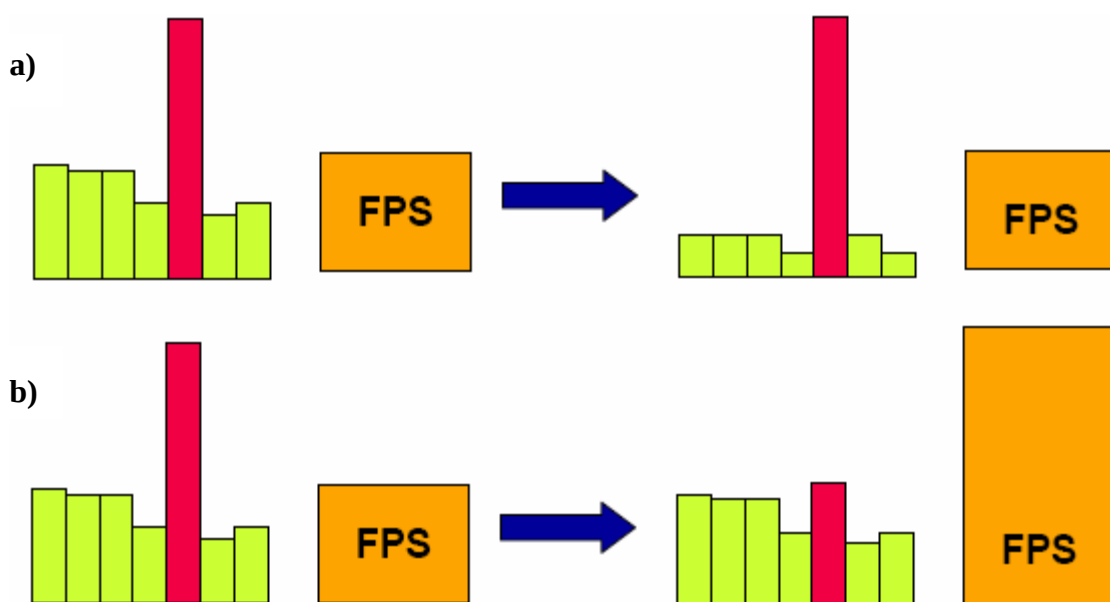
Wanneer bij ontwikkeling van een toepassing gebruik wordt gemaakt van GPU programmeren, kan de programmeur voor nieuwe problemen komen te staan. Hiermee moet op een andere manier worden omgesprongen dan problemen bij het programmeren voor de CPU. Ook bij optimalisatie van applicaties, die van de GPU gebruik maken, spelen andere aspecten een rol. In de komende subhoofdstukken worden enkele technieken en methoden geboden die de programmeur kan aanwenden om efficiënt gebruik te maken van de GPU en problemen te debuggen.

7.1 Performance optimalisatie

In Hoofdstuk 3 is te zien dat de architectuur van de GPU sterk afwijkt van de CPU. Zoals voor alle andere architecturen geldt ook voor de GPU dat maximale performance gehaald wordt wanneer alle onderdelen van de architectuur in gelijke mate belast worden. Verder moet er zo efficiënt mogelijk gebruik gemaakt worden van de geboden mogelijkheden. De NVIDIA GPU Programming Guide [17] geeft een vrij grote hoeveelheid methoden die hierin kunnen ondersteunen. In de volgende paragrafen staat uitgelegd hoe de bottleneck geïdentificeerd kan worden. Verder worden enkele technieken geboden, die tot hogere performance kunnen leiden.

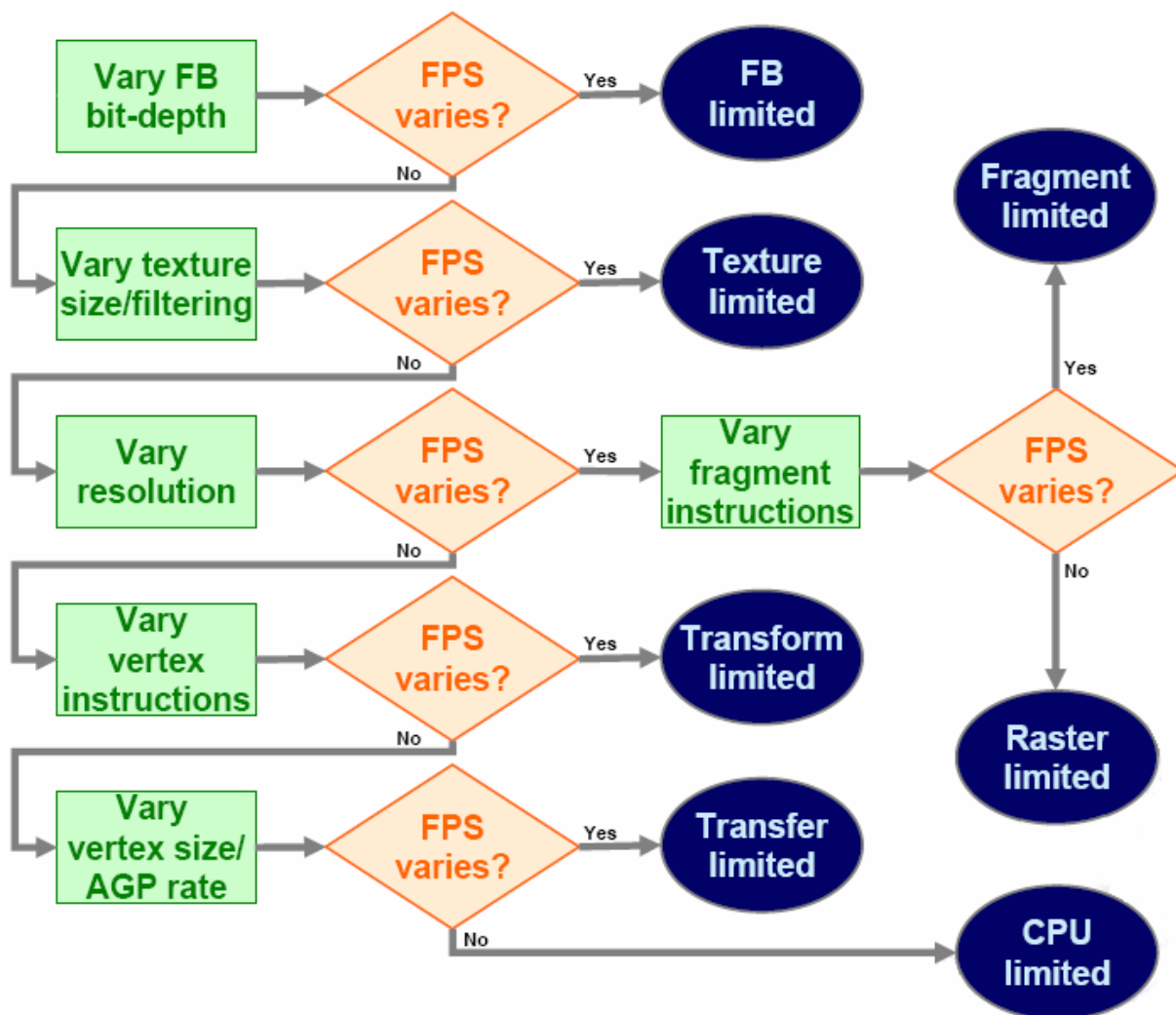
7.1.1 Identificeer de bottleneck

Het lijkt misschien voor de hand liggend, maar gezien de pipeline architectuur van de GPU is het extra belangrijk om te weten waar de bottleneck in een applicatie ligt. De verschillende stadia in de pipeline worden immers, gekeken naar 1 data element, sequentieel uitgevoerd. Zo wordt de snelheid van de gehele keten bepaald door de zwakste schakel.



Figuur 36 Twee mogelijkheden voor het bepalen van een bottleneck [18]

Het bepalen van de bottleneck kan, zoals te zien is in Figuur 36, op twee manieren. Wanneer het verlagen van de werklust van de andere stadia weinig of geen effect heeft, kunnen deze als potentiële bottlenecks afgeschreven worden. Evenzo wordt de bottleneck geïdentificeerd indien het verlagen van de werklust van een bepaald stadium van de pipeline een positief effect heeft. Waar hierbij voor opgepast moet worden, is dat bepaalde incorrecte werkwijzen hierbij niet alleen de werklust van het stadium dat onderzocht wordt verlagen, maar ook van één of meer andere stadia. Zo zal bijvoorbeeld het halveren van de hoeveelheid geometrie, met als doel te onderzoeken of de vertex shader de bottleneck is, geen juiste werkwijze zijn. Dit zal namelijk ook zeker invloed hebben op de transfer tussen CPU en GPU. Mogelijk wordt hierdoor ook de hoeveelheid werk voor de rasterizer en de fragment shader verminderd, indien dit een kleinere hoeveelheid pixels oplevert. In Figuur 37 is een flowchart aangegeven om systematisch de bottleneck van een applicatie te achterhalen.

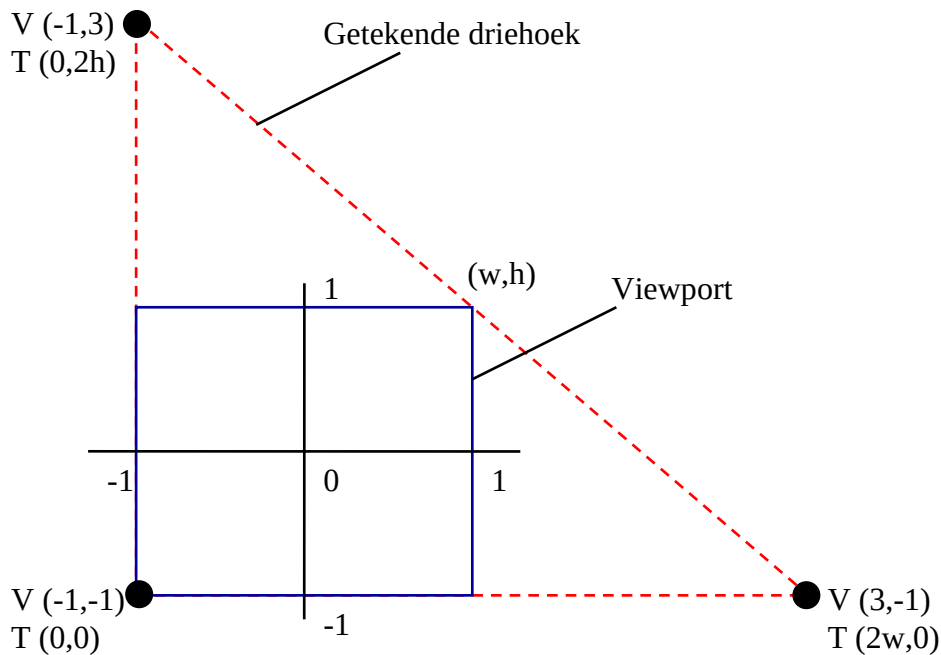


Figuur 37 Bottleneck identificatie flowchart [18]

Wanneer op deze wijze de ergste bottleneck bepaald is, kunnen doelgericht pogingen ondernomen worden om de prestaties te verbeteren. In de NVIDIA GPU Programming Guide [17] staan per categorie enkele mogelijkheden hoe dit bewerkstelligd kan worden.

7.1.2 Render met een triangle in plaats van quad

In Hoofdstuk 5.3 is een makkelijke en voor de hand liggende methode weergegeven voor het verwerken van data in een tweedimensionale array (texture) door het tekenen van één quad. Deze methode is makkelijk toepasbaar en erg intuïtief maar haalt helaas niet de optimale performance. Een betere variant kan uitgelegd worden aan de hand van Figuur 38. Hierin is te zien dat het mogelijk is een rechthoek met breedte w en hoogte h te definiëren door een driehoek te tekenen met tweemaal zo grote zijden. Daarbij moet wel opgelet worden dat de op dat moment actieve viewport exact $w \times h$ pixels beslaat. De delen van de geometrie die buiten de viewport vallen, worden niet door de rasterizer en daarmee verdere stadia in de pipeline verwerkt.

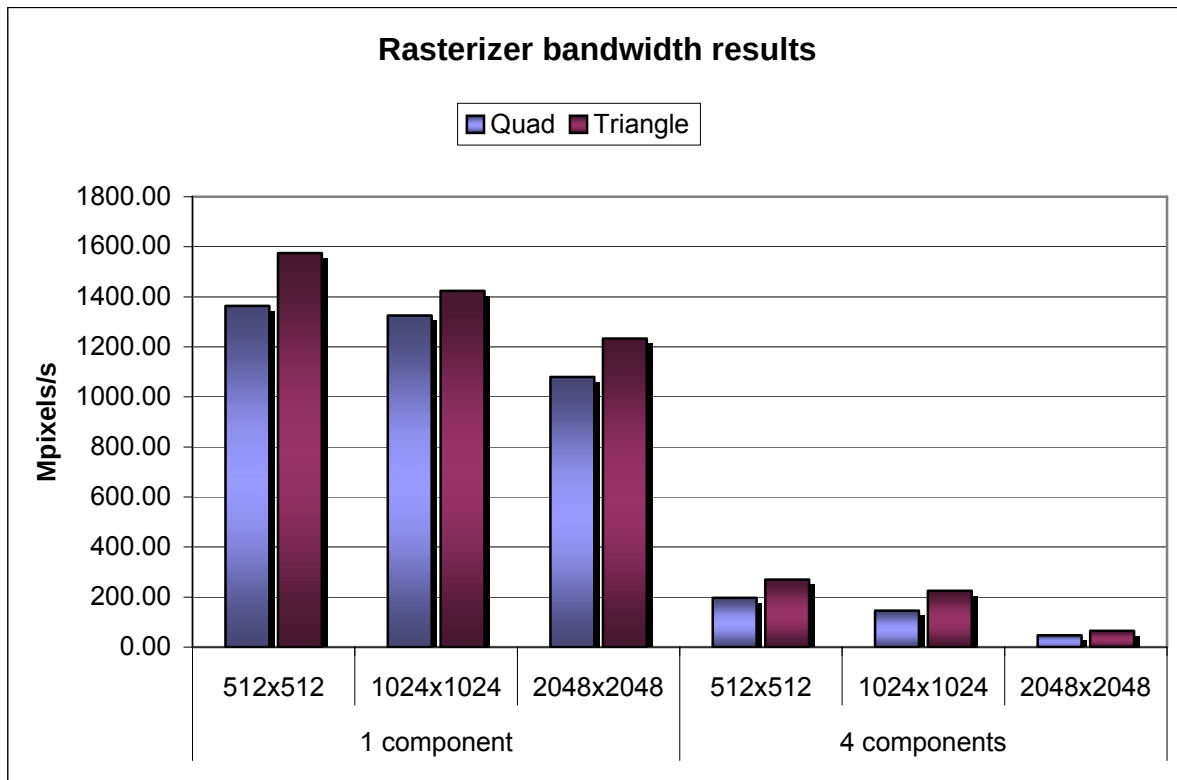


Figuur 38 Rasterisatie van een rechthoekig vlak door één driehoek

De oorzaak van dit verschil in performance wordt duidelijk wanneer gekeken wordt naar de OpenGL 2.0 specificatie [7]. Wanneer een quad door de rasterizer verwerkt wordt, wordt deze eerst in twee driehoeken verdeeld op een diagonaal van de quad. Vervolgens worden deze twee driehoeken beiden door de rasterizer verwerkt. Een probleem ontstaat echter bij de fragments, die zich op de diagonaal tussen beide driehoeken bevinden. Volgens de specificatie wordt zo'n fragment uitsluitend door één driehoek voortgebracht. Hierbij wordt gekeken naar de driehoek die het centrum van de fragment bevat. Men kan eenvoudig nagaan dat het testen van deze conditie voor alle fragments rond de diagonaal extra bewerkingen kost. Deze kunnen uitgespaard worden door het genereren van één enkele driehoek. De resultaten en behaalde winst zijn gemeten met GPUBench [W27] op een GeForce 6600 GT. Zie hiervoor Tabel 9 en Figuur 39.

	1 component			4 components		
	512x512	1024x1024	2048x2048	512x512	1024x1024	2048x2048
Quad (Mpix/s)	1363.70	1325.19	1079.89	196.77	146.46	47.33
Triangle (Mpix/s)	1574.75	1424.04	1233.72	270.55	224.76	65.26
Speedup	15.48 %	7.46 %	14.24 %	37.50 %	53.46 %	37.88 %

Tabel 9 Verschil in bandbreedte in rasterizer tussen triangle en quad rendering



Figuur 39 Verschil in bandbreedte in rasterizer tussen triangle en quad rendering

7.1.3 Render to Texture met Framebuffer Objects

Zoals ook vermeld in Hoofdstuk 5, is het mogelijk om meerdere fragment programma's achtereenvolgens uit te voeren, waarbij de uitvoer van het eerste programma als invoer voor het volgende programma dient. Dit kan het gemakkelijkst bewerkstelligd worden door de uitvoer zoals gewoonlijk naar het scherm of naar een *offscreen buffer*. Nadat de tekenoperatie klaar is, kan dit geheugen middels het OpenGL commando *glCopyTexSubImage* gekopieerd worden naar een reeds bestaande texture. Deze methode werkt op zich prima maar presteert niet optimaal door het herhaaldelijk kopiëren van de texture. Het probleem zal alleen maar meer merkbaar zijn wanneer grote textures gebruikt worden of het algoritme bestaat uit een grote hoeveelheid iteraties van dezelfde of verschillende fragment programma's.

Door gebruik te maken van `GL_EXT_framebuffer_object` [20], een door het ARB goedgekeurde OpenGL extensie, kan de uitvoer van een fragment programma rechtstreeks in een texture geplaatst worden, zonder dat deze achteraf nog hierin gekopieerd hoeft te worden. Het *Framebuffer Object* biedt een aantal aanknopingspunten waaraan bijvoorbeeld textures gekoppeld kunnen worden. Aan het gebruik van deze aanknopingspunten zijn echter wel enige restricties gebonden. Zo kunnen er bijvoorbeeld aan de verschillende punten uitsluitend textures van dezelfde grootte en typen gekoppeld worden, voor een complete beschrijving: zie [20]. Het is echter wel mogelijk om meerdere verschillende Framebuffer Objects te definiëren waardoor het toch mogelijk is om binnen één applicatie de uitvoer van verschillende fragment programma's in verschillende formaten en typen rechtstreeks in een texture weg te schrijven.

Een klein voorbeeld van hoe deze extensie te gebruiken is staat weergegeven in Code fragment 4. Voor verdere informatie en voorbeelden, zie [20].

```
//Initialization
glGenFramebuffersEXT(1, &fb);
glGenTextures(1, &color_tex);

glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);

// initialize color texture as usual
glBindTexture(GL_TEXTURE_2D, color_tex);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB8, 512, 512, 0,
             GL_RGB, GL_INT, NULL);

//attach the texture to attachment point 0
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                          GL_COLOR_ATTACHMENT0_EXT,
                          GL_TEXTURE_2D, color_tex, 0);

//Usage in draw loop
glBindTexture(GL_TEXTURE_2D, 0);
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);

<draw to the texture>

glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
```

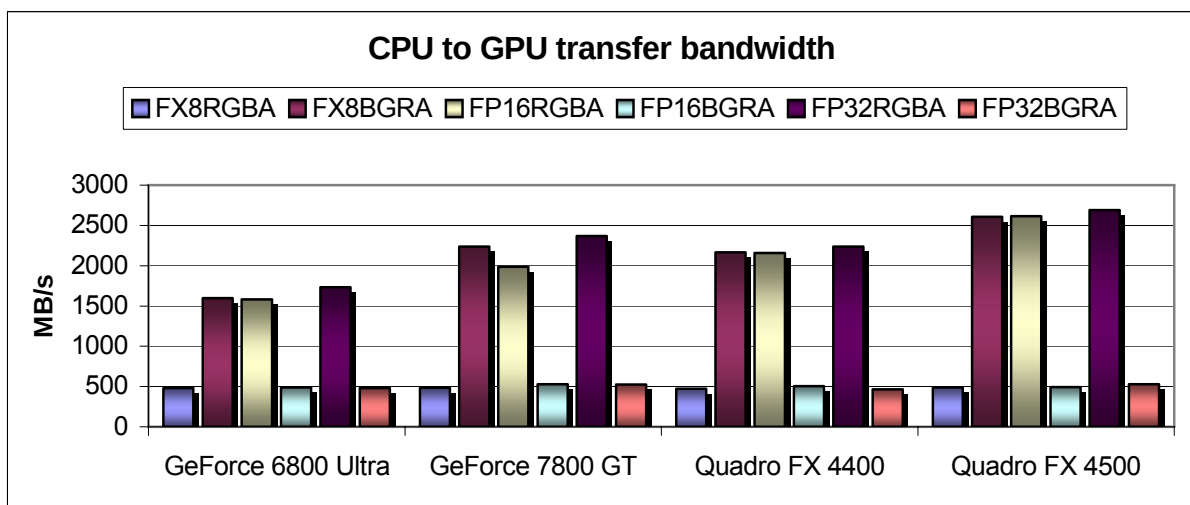
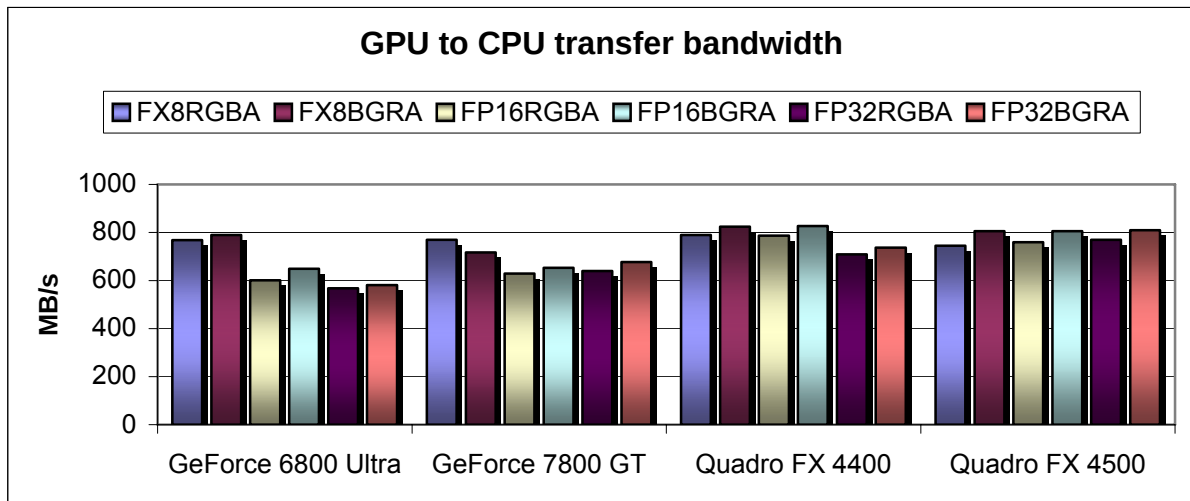
Code fragment 4 Gebruik Framebuffer Object

7.1.4 Datatransfer van en naar de GPU

Omdat data in eerste instantie op de CPU gegenereerd/geladen wordt, is het noodzakelijk deze data te transporteren naar de GPU voor verdere verwerking. Ook is het mogelijk dat, met name in GPGPU toepassingen, de resultaten van de bewerkingen weer teruggelezen moeten worden naar de CPU. Deze datatransfer kost uiteraard tijd en wordt met de normaal gebruikte OpenGL commando's *glTexSubImage* en *glReadPixels* met tussenkomst van de CPU uitgevoerd. Alle tijd die ingenomen wordt in deze transfers heeft hierdoor ook direct invloed op de CPU belasting. Het is dus van belang dat deze transfer zo efficiënt mogelijk verloopt.

7.1.4.1 Datatypes, -formaten en -indelingen

Een belangrijke factor hierin is, naast het formaat en hoeveelheid kanalen van de data, ook de volgorde waarin de elementen worden aangeboden. Voor data met 8 bit per kanaal hanteren de videokaarten van NVIDIA intern een data-indeling conform de Microsoft GDI indeling, in de volgorde BGRA. Wanneer data aan de GPU aangeboden wordt wat in CPU geheugen in de volgorde RGBA staat, moeten bij de transfer alle bytes in het rode en blauwe kanaal omgewisseld worden. Dit gaat ten koste van de effectieve bandbreedte, zoals te zien is in metingen van NVIDIA in Tabel 10 en Figuur 40. De transfertijden duren langer, en daarmee wordt de CPU ook langer bezet gehouden. Merk echter op dat dit precies omgekeerd is bij de 16 en 32 bit floating point textures. Deze worden dus efficiënter in RGBA formaat verstuurd.



Figuur 40 Transfer bandbreedtes bij verschillende texture formaten en volgordes [19]

Van GPU naar CPU	FX8RGBA	FX8BGRA	FP16RGBA	FP16BGRA	FP32RGBA	FP32BGRA
GeForce 6800 Ultra	768 MB/s	790 MB/s	602 MB/s	650 MB/s	568 MB/s	581 MB/s
GeForce 7800 GT	770 MB/s	718 MB/s	630 MB/s	653 MB/s	640 MB/s	677 MB/s
Quadro FX 4400	789 MB/s	824 MB/s	787 MB/s	827 MB/s	710 MB/s	738 MB/s
Quadro FX 4500	745 MB/s	805 MB/s	760 MB/s	806 MB/s	770 MB/s	810 MB/s

Van CPU naar GPU	FX8RGBA	FX8BGRA	FP16RGBA	FP16BGRA	FP32RGBA	FP32BGRA
GeForce 6800 Ultra	480 MB/s	1596 MB/s	1583 MB/s	490 MB/s	1734 MB/s	480 MB/s
GeForce 7800 GT	483 MB/s	2238 MB/s	1987 MB/s	527 MB/s	2367 MB/s	525 MB/s
Quadro FX 4400	471 MB/s	2167 MB/s	2157 MB/s	503 MB/s	2237 MB/s	463 MB/s
Quadro FX 4500	490 MB/s	2605 MB/s	2613 MB/s	493 MB/s	2689 MB/s	527 MB/s

Tabel 10 Transfer bandbreedtes bij verschillende texture formaten en volgordes [19]

Om deze metingen te controleren met bekende hardware en om ook andere formaten te kunnen testen, heb ik zelf een applicatie geschreven die soortgelijke tests uitvoert. Deze applicatie test voor verschillende datatypes (8-bit en 16-bit unsigned byte, 16-bit en 32-bit float) de transfer van 1 tot 4 kanalen data in textures met van 1 tot 4 kanalen. Hierbij wordt ook gelet op de verschillen tussen transfer in de formaten RGB(A) en BGR(A). De details en resultaten van mijn eigen metingen zijn te vinden in Bijlage C.

Hierin valt op dat er toch een groot verschil in effectieve bandbreedte te meten is wanneer verschillende formaten textures gebruikt worden. Vreemd genoeg neemt deze juist toe wanneer gebruik wordt gemaakt van kleinere textures zoals in de 256 x 256 testen. Dit gaat tegen de verwachtingen in omdat, in geval van grote datatransfers, de overhead van het starten van de transfer relatief klein is. De verschillen tussen de beide resultaten kunnen deels te wijten zijn aan de inherente onnauwkeurigheid van de manier van meten. Testen om deze afwijkingen te meten hebben echter uitgewezen dat schommelingen groter dan 5% zeer zeldzaam zijn. Alle verschillen groter dan 5% moeten dus wel degelijk als serieus verschil in de effectieve bandbreedte opgevat worden. Hiermee kan geconstateerd worden dat de verschillen tussen RGBA en BGRA transfers voor 8-bit unsigned byte, zoals aangegeven door NVIDIA, minder merkbaar worden naarmate de texture omvang toeneemt. Tevens is verschil uitsluitend merkbaar wanneer gebruik wordt gemaakt van 4-kanaals textures. Dit doordat bij conversie tussen RGBA en BGRA selectief twee kanalen omgewisseld moeten worden, terwijl voor de conversie tussen RGB en BGR simpelweg de hele set van drie waarden omgekeerd kan worden ingelezen.

Verder wijst deze test op verschillen tussen OpenGL extensies van ARB en NVIDIA, die hetzelfde doel hebben, namelijk het gebruik van 16-bit dan wel 32-bit float textures. Ondanks het feit dat de resultaten getest zijn op een videokaart van NVIDIA, scoren de extensies van ARB op dit gebied gelijkwaardige of hogere bandbreedtes. In bepaalde gevallen, vooral met 16-bit textures, zijn deze verschillen extreem hoog. Het is dus op basis van deze testen aan te raden om hiervoor altijd de ARB extensies te gebruiken.

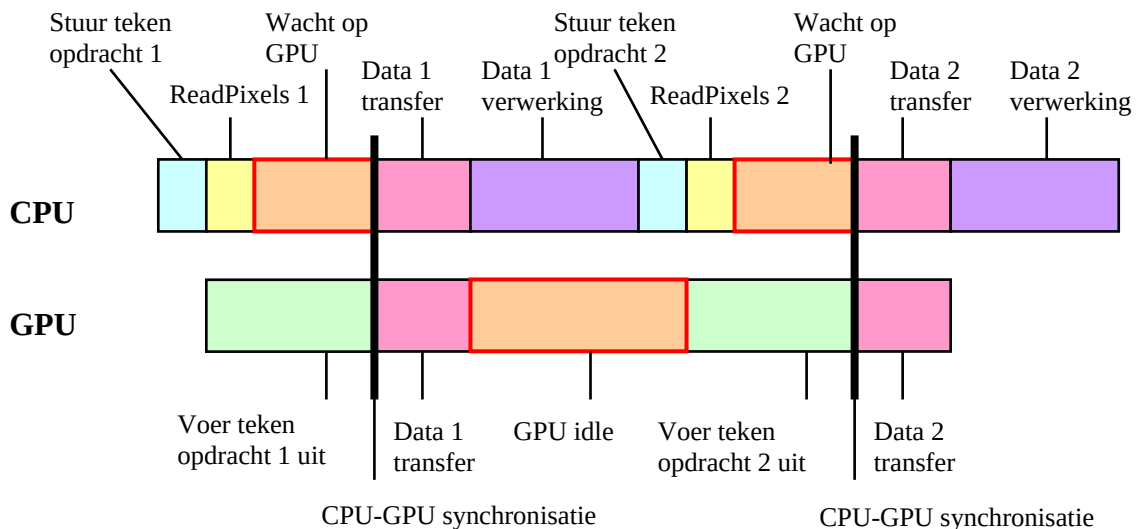
Uit deze resultaten kan verder een interessante conclusie getrokken worden, wanneer er 3-kanaals data wordt in- of uitgelezen. In dat geval geniet het gebruik van 4-kanaals float textures de voorkeur boven 3-kanaals float textures. In alle metingen worden hiermee significant hogere effectieve bandbreedtes gemeten.

Er is dus grote variatie te meten tussen de verschillende transfermogelijkheden, gebruikte extensies en datatypes. De exacte resultaten zijn echter afhankelijk van verschillende factoren zoals het type videokaart, de gebruikte driver versie en of het een AGP of PCI-Express bus betreft. Vandaar het advies om deze meetresultaten slechts te gebruiken als richtlijn voor de juiste keuze en voor de bewustwording van de grote performance verschillen tussen de verschillende methoden. Dergelijke testen, uitgevoerd op het beoogde systeem met relevante parameters zoals bijvoorbeeld de grootte van de texture, moeten gedaan worden om zeker te zijn van optimale performance van een bepaalde toepassing.

7.1.4.2 Vermijden van CPU - GPU synchronisatie bij readback operaties

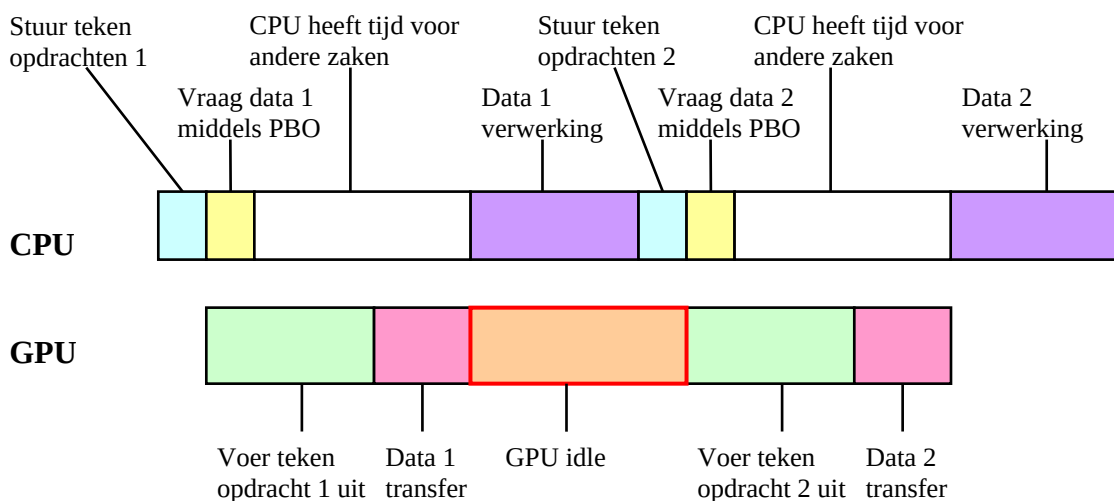
Wanneer data van de GPU wordt teruggelezen, loopt men doorgaans tegen een grote terugval van de performance aan. De oorzaak hiervoor wordt vaak in eerste instantie gezocht in de busbandbreedte. Zeker wanneer gebruik wordt gemaakt van een AGP videokaart, gezien deze een asymmetrische bus heeft, welke twee maal zo snel is in de richting van CPU naar GPU als andersom.

Dit vormt doorgaans echter niet het grootste probleem. Een erg belangrijk punt om rekening mee te houden is het feit dat operaties die data terug te lezen, zoals *glReadPixels*, dit doen uit het framebuffer. Gezien tekenoperaties hun resultaten wegschrijven in deze framebuffer, moeten deze eerst allemaal uitgevoerd worden om de correcte informatie te kunnen verkrijgen. Daarna pas kan de CPU de gevraagde data ontvangen. Gedurende dit hele proces wordt de CPU bezet gehouden en is deze niet beschikbaar om andere bewerkingen uit te voeren. Daarnaast betekent het synchroniseren van de CPU met de GPU dat de GPU pipeline leegloopt en dus direct na de transfer volledig inactief is. Dit probleem is schematisch weergegeven in Figuur 41.



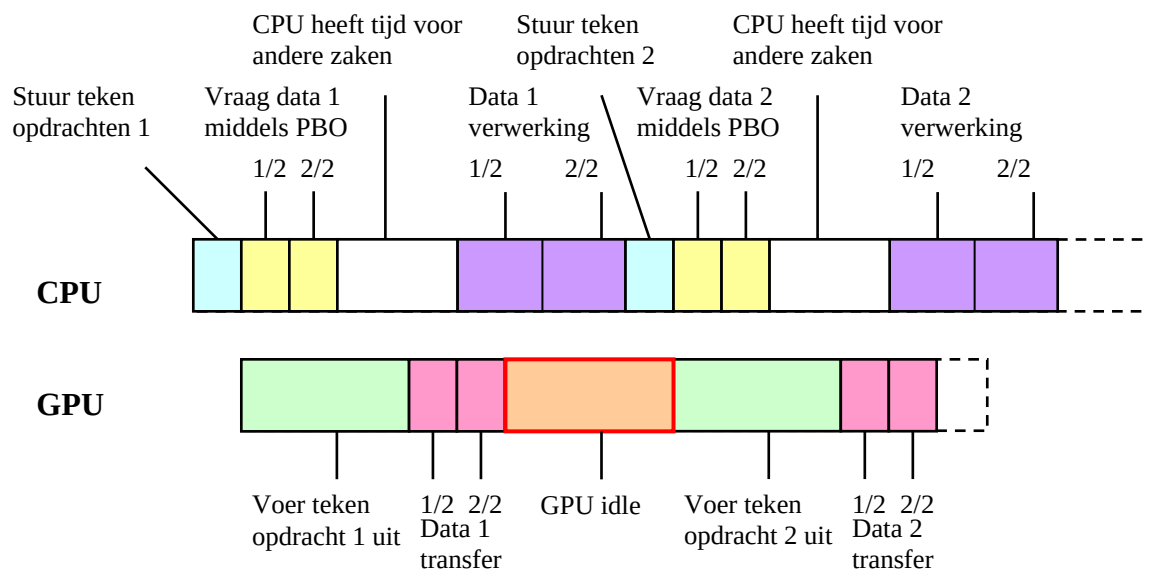
Figuur 41 Vertraging van GPU en CPU door *glReadPixels*

Om dit probleem deels tegen te gaan is de OpenGL extensie Pixelbuffer Object (PBO) [21] gemaakt. Deze maakt het mogelijk om de datatransfer middels *DMA* te laten plaatsvinden. Hierdoor kan de CPU andere taken verrichten gedurende de tijd, die de GPU nog nodig heeft om alle tekenoperaties te voltooien en de volledige transfer tijd, zie Figuur 42.



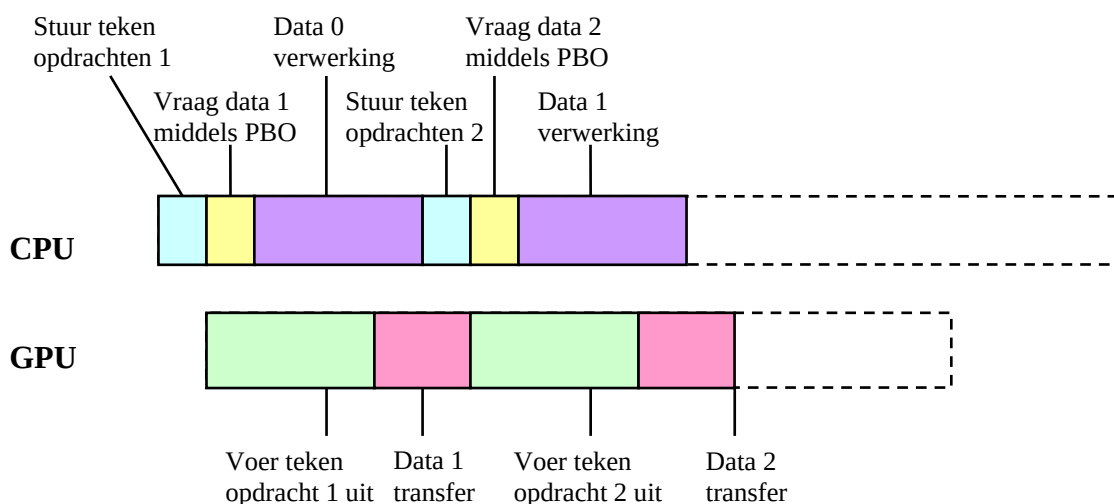
Figuur 42 Lagere CPU belasting door gebruik van PBO's

Van deze eigenschap van PBO's kan handig gebruik gemaakt worden door grote transfers op te delen in twee of meer delen. Hierdoor kan meteen met de verwerking van de eerste data begonnen worden terwijl de transfer van de overige data nog in volle gang is, zie ter illustratie Figuur 43. Dit heeft als resultaat dat de volledige doorvoertijd verkort kan worden.



Figuur 43 Asynchrone datatransfer door gebruik van PBO's

Wanneer het geen probleem is dat data pas één frame later verwerkt wordt dan het gegenereerd is, is het mogelijk een nog grotere winst te halen. Door alternerend de data van de vorige frame te verwerken en de huidige frame aan te sturen, kunnen de CPU en GPU optimaal parallel blijven werken. Deze techniek werkt optimaal wanneer de verwerking van de data, eventueel samen met andere taken van de CPU, ongeveer gelijk is aan de totale verwerkingstijd van de GPU. Deze methode is schematisch weergegeven in Figuur 44.



Figuur 44 Optimaal parallel gebruik van CPU en GPU door gebruik van PBO's

7.1.5 Efficiënte branching in GPU programma's

Zoals in Hoofdstuk 3.3.2 is uitgelegd, gaat de GPU en hierin met name de fragment processor nog niet optimaal om met branching in programma's. Het is dan ook raadzaam dit zoveel mogelijk te vermijden. Uiteraard is het voor correcte werking van veel algoritmen onmogelijk deze simpelweg achterwege te laten. Met behulp van de in deze paragraaf gegeven mogelijkheden kunnen de branches echter vervangen worden door meer efficiënte constructies.

7.1.5.1 Gebruik van wiskunde

De eerste mogelijkheid is om de condities te vervangen door een wiskundige operatie. Dit is bijvoorbeeld bruikbaar, wanneer een gegeven of berekende waarde afhankelijk van een conditie wel of niet bij een totaal opgeteld moet worden. Zie Code fragment 5 voor een voorbeeld.

```
//useAorB is 0 or 1  
  
if (useAorB == 1)  
    total += A;  
else  
    total += B;  
  
//replace by:  
  
total += useAorB * A + (1-useAorB) * B;
```

Code fragment 5 Vervang branch code door wiskundige operaties

7.1.5.2 Gebruik van conditionele statements

In bepaalde gevallen is het niet mogelijk om met enkel wiskundige operaties het effect van een branch te verkrijgen. In enkele van deze gevallen is de overhead van de flow control statements, zoals vermeld in Tabel 2 op pagina 17, te omzeilen door gebruik te maken van conditionele expressies. Een voorbeeld hiervan is gegeven in Code fragment 6.

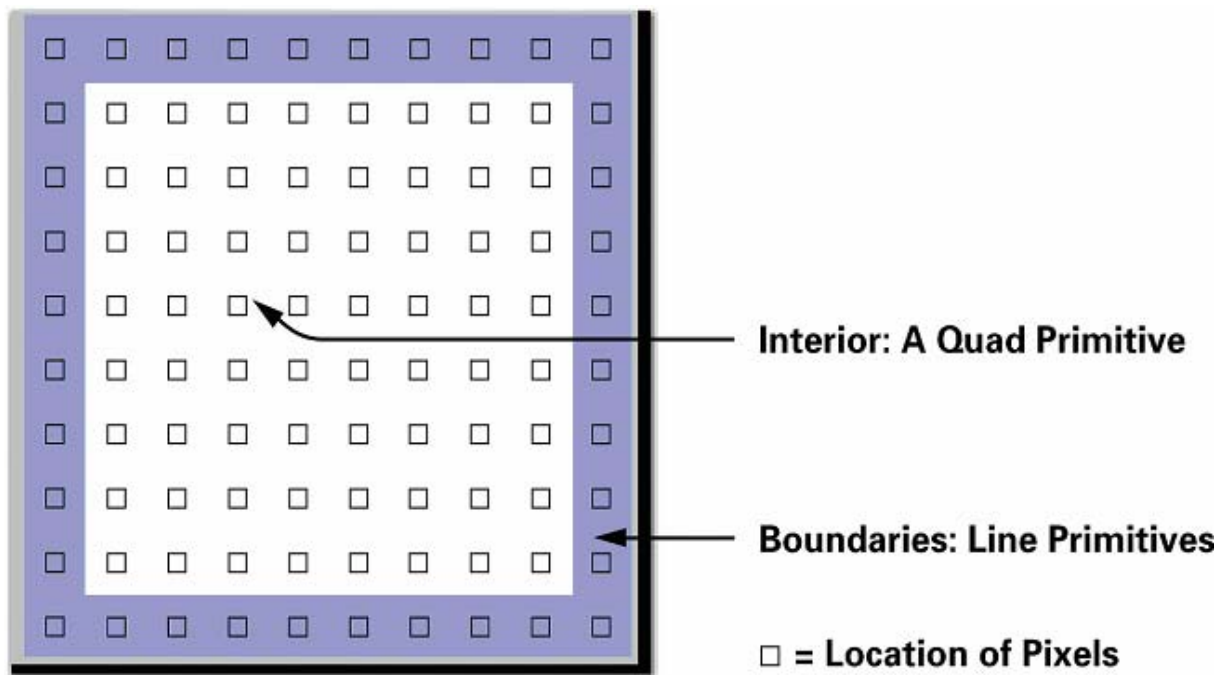
```
if (myValue > threshold)  
    total += myValue;  
else  
    total += otherValue;  
  
//replace by:  
  
total += (myValue > threshold ? myValue : otherValue);
```

Code fragment 6 Vervang branch code door een conditionele expressie

7.1.5.3 Opdelen van genererende geometrie

Een andere situatie die in de praktijk regelmatig voorkomt, is dat fragmenten uit bepaalde locaties altijd dezelfde weg in een branch kiezen. Deze situatie doet zich bijvoorbeeld voor wanneer convolutie filter andere berekeningen moet doen wanneer het fragment zich aan de rand van het geheel bevindt. Alle fragmenten langs de randen voeren altijd deze alternatieve berekeningen uit, terwijl alle andere fragmenten altijd de standaard berekeningen uitvoeren.

Een efficiënte oplossing hiervoor is het definiëren van twee fragment programma's, één voor de randen en één voor de standaard berekeningen, en deze met verschillende delen geometrie aan te roepen zoals weergegeven in Figuur 45. Hierdoor kan de branching code geheel uit de fragment programma's verdwijnen.



Figuur 45 Implementeren van statische branches met verschillende delen geometrie [24]

7.1.5.4 Gebruik van Z-culling

Als laatste mogelijkheid wordt hier het gebruik maken van *Z-culling* behandeld. Hiervoor moet eerst het *Z-buffer* gevuld worden met verschillende waarden waarbij de waarde aangeeft of een bepaalde branch wel of niet genomen moet worden, bijvoorbeeld 0 en 1. Vervolgens kan, gebruik makend van wederom twee verschillende fragment programma's, het volledige gebied twee maal met de verschillende z-waarden getekend worden. Hierbij wordt in beide gevallen gekeken, of de nieuwe z-waarde gelijk is aan de waarde welke op dat moment in het z-buffer staat. Op deze wijze worden uitsluitend de fragmenten verwerkt waarvoor ook 0 in het z-buffer staat, wanneer een quad getekend wordt met z-waarde 0. Ook in dit geval kan de branching code geheel uit de fragment programma's verdwijnen.

7.1.6 Profielen, datatypes en data-indeling

De keuze van profielen, datatypes en de indeling van de data kunnen grote invloed hebben op de totale prestatie van een programma. In dit hoofdstuk worden enkele punten genoemd waarop gelet dient te worden bij het maken van een GPU programma.

7.1.6.1 Profielen

Zoals in Hoofdstuk 4.2.1 is genoemd bestaat de mogelijkheid om snelheidswinst te behalen door een zo 'oud' mogelijk profiel te kiezen voor het compileren van een Cg programma. Dit staat onder andere vermeld in [17]. In praktische testcases heb ik een snelheidswinst gemeten van rond de 40% wanneer gebruikte profiel veranderd werd van fp40 naar fp30. Een exacte verklaring hiervoor is helaas in de literatuur niet gevonden. Wel wordt er aangegeven dat de krachtigere en meer flexibelere mechanismen mogelijk gemaakt door de nieuwere profielen een rol spelen. Ter vergelijking kan er gekeken worden naar het verschil tussen een programma geschreven in C en C++. C++ is op te vatten als een superset van C, net zoals een shader model 3.0 profiel een superset is van een 2.0 profiel. Net zoals bij de verschillende profielen zijn C++ programma's doorgaans net wat langzamer dan C programma's, al is het verschil tussen C en C++ veel minder drastisch.

7.1.6.2 De invloed van datatypes op de snelheid van berekeningen

Net zoals bij profielen is het ook aan te raden om het datatype met de minimaal benodigde precisie te kiezen voor de berekeningen. Naast het bekende 32 bit *float* type met een teken bit, een 32 bit mantisse en een 8 bit exponent (s32e8) is ook het *half* type (s10e5) en in Cg ook het *fixed* type (s1.10) beschikbaar. Deze datatypes zijn gegeven in aflopende precisie en daarmee in oplopende snelheid van het shader programma. NVIDIA geeft in [17] de volgende richtlijnen voor het gebruik van deze datatypes:

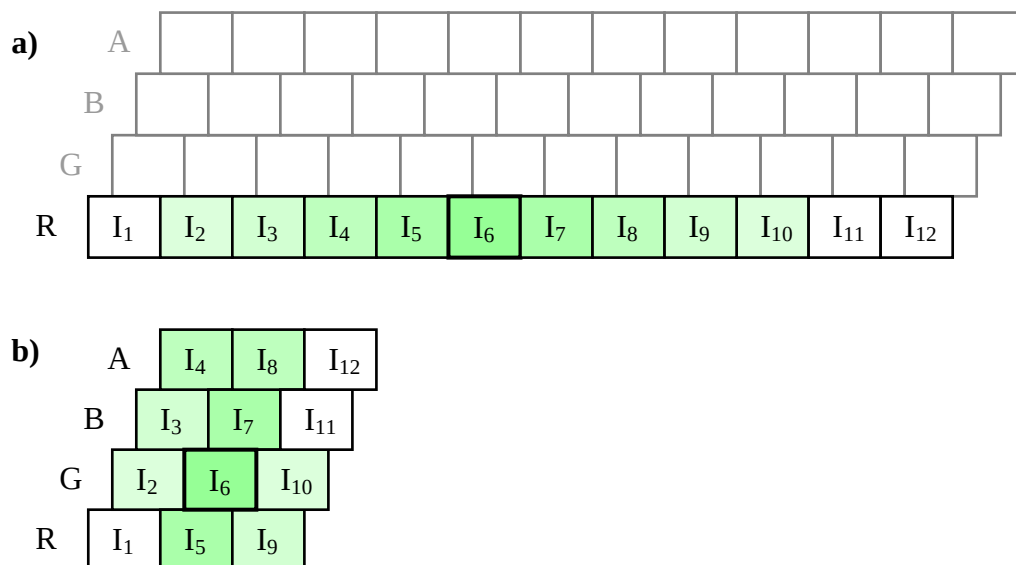
- ❑ Het *fixed* type is het snelste en zou gebruikt moeten worden voor weinig precieze calculaties zoals kleurinformatie
- ❑ Wanneer floating-point precisie vereist is, wordt aangeraden het *half* type te gebruiken
- ❑ Uitsluitend wanneer de hoogst mogelijke precisie vereist is, moet het *float* type gebruikt worden

Naast het positieve effect op de transfersnelheden zoals vermeld in Hoofdstuk 7.1.4 heeft de keuze voor een kleiner datatype dus ook invloed op de totale verwerkingssnelheid van de berekeningen.

7.1.6.3 Aanpassen van de indeling en gebruik van data naar de toepassing

Behalve de keuze van het datatype van de textures en berekeningen kan ook de verdeling van die data een grote rol spelen in de performance. Stel bijvoorbeeld dat met behulp van de GPU getracht wordt om een convolutiefilter op de grijswaarden afbeelding te versnellen, waarna het resultaat weer teruggelezen wordt naar de CPU. Zo'n afbeelding bevat dus per pixel één waarde. Een voor de hand liggende geheugenindeling hiervoor is gegeven in Figuur 46a. Hierin is te zien dat in één enkel kanaal de intensiteit van de betreffende pixels is opgeslagen. Wanneer een convolutiefilter uitgevoerd wordt waarbij naast de oorspronkelijke waarde ook de waarden van de vier linker- en de vier rechterburen van de te berekenen pixel beschouwd moeten worden, heeft dit per pixel 9 texture lookups tot gevolg. Voor een afbeelding van 100x100 pixels, betekent dit dat er in totaal 90000 texture lookups gedaan moeten worden.

Door nu echter de informatie van de afbeelding niet enkel in één kanaal in te lezen, maar in plaats daarvan in alle vier beschikbare kanalen, kan significante winst behaald worden. De datarepresentatie is zichtbaar gemaakt in Figuur 46b.



Figuur 46 Efficiënt geheugengebruik kan algoritmen drastisch versnellen

Door deze wijze van het inlezen van data, wordt de totale texturegrootte in het geheugen over de breedte 4 keer kleiner. Dit betekent ook dat het fragment programma met het convolutiefilter 4 keer minder aangeroepen wordt. Per aanroep van dit programma, waarin nu 4 uitvoer waarden tegelijk berekend worden, zijn er ook nog maar 3 texture lookups noodzakelijk om alle benodigde data te bemachtigen. Dit betekent dat de totale hoeveelheid texture lookups nodig om de gehele afbeelding te verwerken gedaald is van 90000 naar 7500. Wanneer de data weer op dezelfde manier teruggelezen wordt als dat het oorspronkelijk naar de GPU was gestuurd, is de geheugenindeling voor de CPU weer identiek aan de oorspronkelijke situatie. Hierin zijn dus geen negatieve effecten te ondervinden aan het gebruik van de alternatieve datarepresentatie.

7.2 Debug methoden

Helaas is het debuggen van OpenGL en in het bijzonder GPU programma's erg lastig zonder gespecialiseerde tools zoals behandeld in Hoofdstuk 6. In dit hoofdstuk worden enkele technieken onder de aandacht gebracht, die de ontwikkelaar kunnen helpen bij het analyseren van fouten in een GPU applicatie. Het op een zinnige wijze toepassen van deze technieken is echter sterk afhankelijk van het specifieke probleem en vergt altijd enige creativiteit van de ontwikkelaar.

7.2.1 Debuggen met afbeeldingen

Wanneer de GPU applicatie een afbeelding als input en/of output heeft, is het in de meeste gevallen het meest praktisch om ook de debug informatie hiervan in de vorm van een afbeelding te beschouwen. Een groot voordeel van deze methode is dat een zeer grote hoeveelheid informatie overzichtelijk in realtime geïnterpreteerd kan worden. Hierbij blijven ook duidelijk alle onderlinge relaties zoals de locaties van pixels zichtbaar.

Als voorbeeld kan een GPU programma beschouwd worden, dat een beeldbewerkend algoritme implementeert. Wanneer de uitvoer hiervan totaal niet overeenkomt met de verwachte of een referentie uitvoer, bijvoorbeeld afkomstig van een programma zoals Matlab, zou in eerste instantie in de shader de initiële invoer ongewijzigd weergegeven kunnen worden. Wanneer ook dit niet goed gaat, kunnen als volgende test de texture coördinaten als uitvoer gebruikt worden. Indien een texture volledig geïndexeerd wordt, moeten deze dus variëren van (0,0) linksonder tot (*breedte*, *hoogte*). Door deze coördinaten te normaliseren naar het bereik [0..1] en hiervan de x-coördinaat in het rode en de y-coördinaat in het groene kanaal te plaatsen, kan makkelijk gezien worden of de fout wellicht in de texture indexing zit. Een voorbeeld van een dergelijke output is te zien in Figuur 47.



Figuur 47 Voorbeeld van texture coördinaten als uitvoer

Indien de indexering ook onjuiste resultaten weergeeft, is het probleem hoogstwaarschijnlijk het gevolg van één of meerdere fouten in de gegeven geometrie of viewport en projectie instellingen. Wanneer uit deze test blijkt dat de coördinaten wel correct ontvangen worden, kan het probleem gezocht worden in het incorrect laden van de te verwerken beeldinformatie.

Bij deze methode is het wel belangrijk om er rekening mee te houden dat de uitvoer naar het beeldscherm altijd genormaliseerd moet worden naar een waarde in het bereik [0..1]. Lagere dan wel hogere waarden worden door de GPU uitgevoerd als 0 respectievelijk 1. Hierdoor kunnen de onderlinge verhoudingen tussen de te analyseren waarden dusdanig verstoord worden dat de uitvoer niet meer te gebruiken is voor het (correct) bepalen van een mogelijke fout. Hiermee wordt helaas in veel gevallen de exacte waarde aangetast, maar op deze wijze kan er aan de hand van over de gehele afbeelding geldende verhoudingen toch waardevolle informatie vergaard worden over de mogelijke oorzaak van de fout.

Hierdoor kunnen alle relevante waarden onderzocht worden door de waarde deel uit te laten maken van de uitgevoerde kleur. Zoals gedemonstreerd is met de texture coördinaten hoeven dit niet perse pixels met een rood groen en blauw kanaal te zijn. Dit kunnen evengoed ook abstractere waarden zijn zoals bijvoorbeeld de hoeveelheid omliggende pixels met een hogere kleurintensiteit dan de huidige pixels. In dit geval kan deze bepaalde waarde gedeeld door 8 in het rode kanaal geplaatst worden. Immers, elke pixel heeft maximaal 8 burens waardoor deze waarde een ongenormaliseerd bereik heeft van [0..8].

7.2.2 Debuggen met exacte waarden

In bepaalde gevallen kan echter niet makkelijk de benodigde informatie geëxtraheerd worden uit een afbeelding. Wanneer naburige pixels bijvoorbeeld totaal geen relatie met elkaar hebben of het als juist van cruciaal belang is de exacte waarde van een berekening te weten. Wanneer dergelijke gevallen zich voordoen bij programmeren op de CPU, laat de ontwikkelaar deze waarde doorgaans in een console afbeelden. De programmeur kan dan aan de hand van dergelijke waarden conclusies trekken over de mogelijk oorzaak van het probleem. Wanneer deze situatie zich voordoet bij het programmeren op de GPU is deze stap echter niet zo gemakkelijk te nemen. In shaders kan niet direct een waarde op het scherm afgebeeld worden, gezien alle directe uitvoer de vorm van een getekende afbeelding heeft. Hiervoor moet de betreffende data eerst teruggelezen worden naar de CPU. Dit teruglezen naar de CPU vergt helaas nog wel enige handelingen en het kan daardoor veel tijd kosten om telkens weer de statements voor het uitvoeren van deze actie toe te voegen en te verwijderen. Dit probleem neemt alleen maar toe wanneer de ontwikkelaar geïnteresseerd is in meerdere waarden tegelijk.

Op de Game Developers Conference van 2006 is door Mark Harris een techniek uitgelegd om dit proces enigszins te vergemakkelijken [25]. Hiervoor wordt gebruik gemaakt van de mogelijkheid om macro's te gebruiken in GPU programma's. De voorgestelde macro is gegeven in Code fragment 7. Het gebruik hiervan in een shader staat weergegeven in Code fragment 8. Hierin worden twee waarden geselecteerd om als uitvoer te dienen, namelijk *temp1* en *temp2*. Ten slotte wordt wanneer *debugSelector* gelijk is aan 0 de oorspronkelijke uitvoer gegenereerd.

```
#ifdef DEBUG_SHADER
    #define CG_PRINTF(index, variable) \
        if (debugSelector == index) \
            DEBUG_OUT = variable;
#else
    #define CG_PRINTF(index, variable)
#endif
```

Code fragment 7 Debug macro voor gebruik in shaders

```
float4 foo( float2 coords, // other params
#ifdef DEBUG_SHADER
    uniform float debugSelector
#endif
) : COLOR0
{
#ifdef DEBUG_SHADER
    float4 DEBUG_OUT;
#endif

    float4 temp1 = complexCalc1();
    float4 temp2 = complexCalc1(temp1);
    float4 ret = complexCalc3(temp2);

    CG_PRINTF(1, temp1);
    CG_PRINTF(2, temp2);

#ifdef DEBUG_SHADER
    CG_PRINTF(0, ret);
    return DEBUG_OUT;
#else
    return ret;
#endif
}
```

Code fragment 8 Gebruik van de macro in een shader

In Code fragment 9 wordt de C++ functie beschreven, die in een lus alle verschillende debug uitvoer mogelijkheden afgaat, en deze van de GPU terugleest. Als laatste wordt het programma uitgevoerd met de waarde 0 als *debugSelector*. Hiermee wordt de oorspronkelijke uitvoer gegenereerd zodat eventuele andere renderpasses, die afhankelijk zijn van deze uitvoer, zo goed mogelijk kunnen blijven functioneren. Hiermee wordt het teruglezen van één of meerdere waarden uit de GPU vereenvoudigd tot een uitvoer statement in de shader en het aanroepen van een debug functie in de C++ code. Wanneer de debug uitvoer tijdelijk ongewenst is, kan deze simpelweg uit de code gecompileerd worden zonder dat alle individuele aanpassingen telkens verwijderd en daarna weer toegevoegd moeten worden.

```
debugProgram( CGProgramprog, x, y, w, h, float** debugData)
{
    CGParampsel= cgGetNamedParameter(prog, "debugSelector");
    for (int selector = 1; selector < 100; ++selector)
    {
        if (debugData[selector] == 0) break;

        // run program with debug selector
        cgGLBindFloat1f(psel, selector);
        runProgram( prog, x, y, w, h);

        // read back results
        glReadPixels(x, y, w, h, GL_RGBA, GL_FLOAT,
                    debugData[selector]);
    }

    cgGLBindFloat1f(psel, 0);
    runProgram( prog, x, y, w, h); // run program as normal
}
```

Code fragment 9 C++ code voor het gebruik van de debug output

8 Multi-GPU aspecten

Met de opkomst van PCI-Express als vervanger voor AGP is het in moderne systemen mogelijk om meerdere grafische kaarten in één systeem te gebruiken. Dit biedt verscheidene mogelijkheden. Zowel ATI als NVIDIA hebben een technologie ontwikkeld om twee grafische kaarten voor de gebruiker transparant te laten samenwerken. Dit zijn respectievelijk *Crossfire* en *SLI*. Daarnaast is het natuurlijk ook mogelijk om beide grafische kaarten onafhankelijk van elkaar aan te spreken. In dit hoofdstuk wordt kort ingegaan op beide methoden en de hierdoor geboden perspectieven voor GPGPU applicaties. Vanwege de focus van het onderzoek op grafische kaarten van NVIDIA wordt hierin uitsluitend SLI behandeld.

8.1 GPUs in SLI configuratie

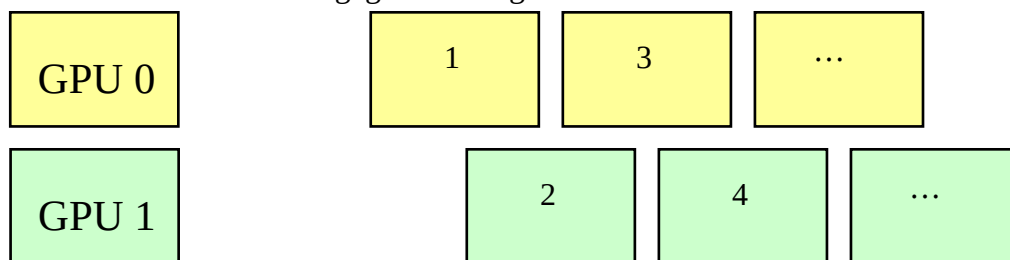
Voor een SLI configuratie worden de volgende eisen gesteld:

- ❑ Beide GPUs moeten van het zelfde type zijn en dus op dezelfde frequenties zijn afgesteld.
- ❑ Het moederbord moet ondersteuning hebben voor SLI, bijvoorbeeld in de vorm van een nForce chipset.

Wanneer GPUs in SLI configuratie gebruikt worden, kunnen ze op verschillende manieren samenwerken om tot snellere dan wel betere resultaten te komen. Een groot voordeel hierbij is dat de programmatuur ongewijzigd kan blijven, doordat GPUs in SLI configuratie voor de software beschouwd worden als één ‘snellere’ GPU. In de volgende paragrafen worden de verschillende modi behandeld waarin een SLI configuratie kan opereren.

8.1.1 AFR

De eerste en volgens NVIDIA de meest gunstige modus is Alternate Frame Rendering of *AFR*. Hierbij worden beide GPUs parallel gebruikt om altemnerend één frame te renderen. Deze methode is in schema weergegeven in Figuur 48.



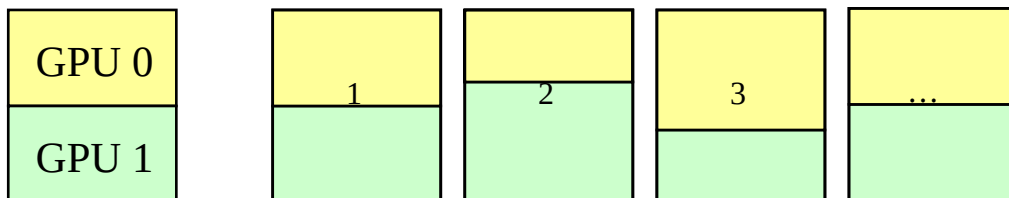
Figuur 48 Twee GPUs in AFR configuratie

Een vereiste voor het goed presteren van deze methode is echter dat alle frames onafhankelijk moeten zijn van elkaar. Wanneer bijvoorbeeld het resultaat van een render tot texture (*RTT*) operatie in frame 1 ook gebruikt moet worden in frame 2, moet GPU 1 wachten tot GPU 0 deze getekend heeft. Daarna moet de data nog gekopieerd worden naar GPU 1 en pas dan kan deze beginnen aan het tekenen van frame 2. In het ergste geval wordt het resultaat van de *RTT* operatie pas aan het einde van de frame berekend, waardoor alle frames sequentieel getekend worden. Dit heeft tot gevolg dat er totaal geen winst verkregen wordt van de SLI configuratie.

In verscheidene gevallen van GPGPU toepassingen speelt deze situatie een grote rol. Bijvoorbeeld in geval van simulatie van fysische aspecten zoals vloeistof dynamica of bepaalde beeldverwerkingalgoritmen waarbij een (beperkte) geschiedenis van beelden bijgehouden wordt voor bewegingsdetectie. In dergelijke situaties biedt deze vorm van SLI dan ook slechts een zeer geringe performancewinst.

8.1.2 SFR

Een andere belangrijke modus van SLI is Split Frame Rendering of *SFR*. Hierin werken beide GPUs gezamenlijk aan één frame per keer. Hierbij wordt de te tekenen geometrie van één frame gesplitst in twee helften, waarbij de bovenste helft door GPU 0 en de onderste door GPU 1 verwerkt wordt. De driver controleert elke frame hoeveel tijd de beide GPUs gedaan hebben over hun deel en verandert aan de hand van deze gegevens de exacte locatie van de splitsing binnen het volgende frame. Op deze wijze wordt getracht een optimale balans te behouden tussen de werklust van GPU 0 en 1. Deze werkwijze is schematisch weergegeven in Figuur 49.



Figuur 49 Twee GPUs in SFR configuratie

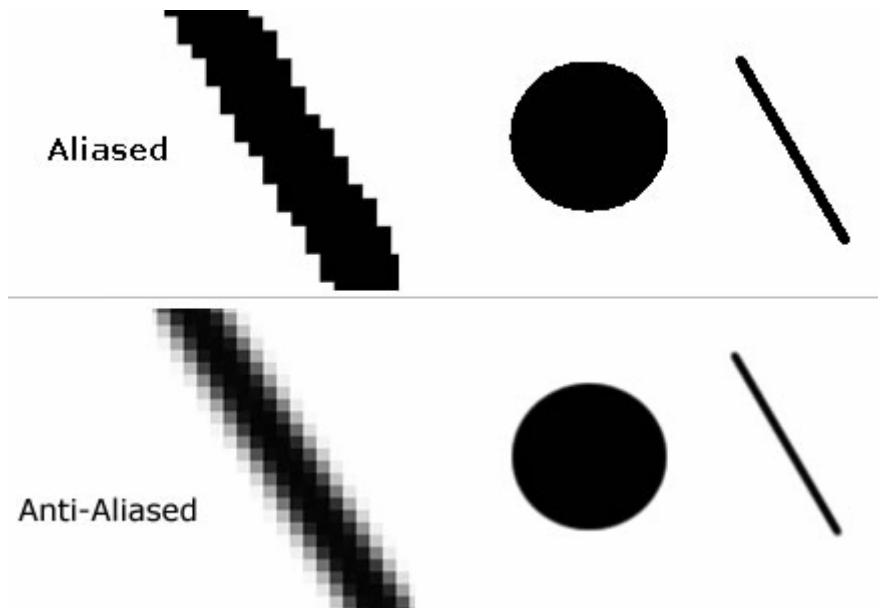
Een nadeel van deze methode ten opzichte van AFR is dat er altijd een hogere communicatieoverhead is. Wanneer RTT wordt toegepast in deze methode, wordt door elk van de GPUs een deel van deze texture gegenereerd. Wanneer deze texture uiteindelijk gebruikt wordt, moeten beide GPUs hun berekende deel doorgeven aan de andere.

Daarnaast moet de geometrie op de scheiding tussen de twee delen van het te tekenen frame opgedeeld worden. Dit om te voorkomen dat beide GPUs alle geometrie te verwerken krijgen en vervolgens effectief hier maar een deel van tekenen. De algoritmen, die deze opdeling bewerkstelligen, zijn vrij goed maar helaas niet perfect. Hierdoor krijgen beide GPUs toch enige overlap in geometrie te verwerken.

Ondanks de communicatieoverhead en de lichte redundantie in de berekeningen kan de extra GPU in deze SLI modus toch voor een nettowinst zorgen. Deze is echter in alle gevallen kleiner dan een AFR configuratie onder gunstige omstandigheden. Wanneer een AFR configuratie echter niet mogelijk is, kan met SFR toch nog performancewinst behaald worden ten opzichte van het gebruik van één enkele GPU.

8.1.3 SLI anti-aliasing

Een andere mogelijkheid voor het gebruik van SLI heeft niet als doel een hogere *framerate* te behalen, maar gebruikt de extra GPU voor het toepassen van betere anti-aliasing. Anti-aliasing zorgt er normaliter voor dat schuine randen van getekende objecten minder kartelig lijken, zie Figuur 50. Dit wordt gedaan door de kleurinformatie van omliggende pixels mee te rekenen bij het bepalen van de exacte uitvoer. Door dit gebied groter te maken en meer omliggende samples te nemen kan hierin een beter resultaat verkregen worden.



Figuur 50 Verschil tussen aliased en anti-aliased graphics

Wanneer SLI anti-aliasing geactiveerd wordt, tekenen beide GPUs de complete frame. Het verschil in deze twee frames zit echter in de beschouwde gebieden waaruit samples genomen worden voor anti-aliasing. Op deze manier wordt het grote samplegebied nodig voor de meest geavanceerde filtering opgedeeld over de twee GPUs zodat de gedeelde anti-aliasing parallel uitgevoerd kan worden. Daarna worden, in een kleine extra stap, de beide parallel berekende delen samengevoegd. Hierdoor kan een hoge kwaliteit anti-aliasing uitgevoerd worden met minder performanceverlies dan wanneer één enkele GPU hiervoor verantwoordelijk is.

Anti-aliasing zelf is een vorm van beeldverbeterende filtering, die voor vrijwel alle GPGPU bewerkingen uitgeschakeld moet zijn. Hierdoor wordt immers de uitvoer van de fragment programma's beïnvloed door andere factoren dan door de geïmplementeerde algoritmen. Vandaar dat deze vorm van SLI niet zinvol is voor GPGPU toepassingen.

8.2 Onafhankelijk gebruik van meerdere GPUs

Waar het gebruik van meerdere GPUs in SLI door communicatieoverhead en dergelijke nooit tot een snelheidswinst van factor 2 kan komen, biedt onafhankelijk gebruik van twee GPUs uiteraard wel deze mogelijkheid. Een probleem hierbij is echter dat het probleem waarvoor een GPGPU applicatie geschreven is, moet worden opgesplitst in twee of meerdere delen. Wanneer de applicatie bijvoorbeeld beeldverwerkende operaties op beelden van meerdere camera's moet uitvoeren, kan het werk per camera verdeeld worden over de beschikbare GPUs. In geval van een particle simulatie, kan de totale set van particles in twee groepen verdeeld worden, die over de GPUs verdeeld worden.

Het nadeel van deze methode is echter dat er per toepassing gekeken moeten worden naar een geschikte verdeling en moet deze ook handmatig geïmplementeerd worden. Hierdoor is dit gebruik van meerdere GPUs minder eenvoudig schaalbaar dan een SLI configuratie. Het kan echter zeker de moeite zijn toch deze investering in de implementatie te maken.

Wanneer bijvoorbeeld 12 camera's verwerkt moeten worden en één GPU kan er 3 tegelijk verwerken, betekent dit dat er in totaal 4 PC's nodig zijn. Door echter 2 GPUs per PC te gebruiken, kan dit aantal al gehalveerd worden tot 2 PC's. Zeker wanneer ruimte een belangrijke factor is kan dit zeer voordelig zijn. Hierbij moet echter wel rekening gehouden worden met het feit dat de bandbreedte van de gebruikte databus niet verdubbeld. In extreme gevallen kan dit dan ook voor een nieuwe bottleneck zorgen.

Voor toepassing van deze techniek moeten meerdere grafische contexten aangemaakt worden. Hierbij dient er dus één context per GPU gebruikt te worden. Voor het aanmaken van deze contexten kan onder Linux zoals gewoonlijk gebruik gemaakt worden van de *GLX* bibliotheek. Hierbij dient bij het commando *glXChooseVisual* de keuze voor het scherm, en daarmee de GPU, gemaakt te worden. De hiermee verkregen *XVisualInfo* kan vervolgens meegegeven worden aan het *glXCreateContext* commando voor het daadwerkelijke aanmaken van de context.

8.3 De toekomst op het gebied van multi-GPU

Vrij recente ontwikkelingen laten zien dat de parallelisatie trend niet alleen binnen de GPU architectuur, maar ook op een hoger niveau in aantallen GPUs verder doorontwikkeld wordt. Zo heeft NVIDIA de GeForce 7950 GX2 en Quadro FX 4500 X2 uitgebracht. Deze kaarten zijn in feite een samensmelting van twee grafische kaarten op één enkele aansluiting, zie Figuur 51. Deze kaart biedt de mogelijkheid om met gebruik van slechts één PCI-E slot een SLI configuratie te verkrijgen.



Figuur 51 GeForce 7950 GX2

Zelf heb ik helaas geen tests met deze kaart kunnen doen maar uit de FAQ over deze kaart op de site van NVIDIA [W28] is gebleken dat de kaart in 'multi-monitor mode' twee beeldschermen kan aansturen waarbij door elk van de GPUs één scherm aangestuurd wordt. Dit zou betekenen dat de methode voor het gebruik van meerdere GPUs zoals gepresenteerd in Hoofdstuk 8.2 ook moet functioneren met het gebruik van deze kaart.

Een andere interessante mogelijkheid die deze kaart biedt is het gebruik van Quad SLI. Hiermee wordt het gebruik van vier grafische kaarten in SLI modus aangeduid. Naast de transparante mogelijkheden die Quad SLI biedt, is het gebruik van vier onafhankelijke GPUs in één PC natuurlijk ook erg interessant. Het geschetste probleem met de 12 camera's zou in zo'n geval op slechts één PC uit te voeren zijn. Uiteraard mits de bandbreedte van de databus hierbij niet voor problemen zorgt.



Figuur 52 Quad SLI opstelling

Het gebruik van vier GPUs, in ieder geval in Quad SLI opstelling, is echter nog sterk in ontwikkeling. De stabiliteitsissues, waar NVIDIA bij het uitbrengen van Quad SLI mee geconfronteerd werd, zijn inmiddels vrijwel opgelost. Wel wordt er nog hard gewerkt aan optimale aansturing van de vier grafische kaarten. Zeker bij gebruik van Quad SLI is het dan ook verstandig om altijd de meest recente drivers te gebruiken.

9 Overige ontwikkelingen

Behalve ontwikkelingen op het gebied van GPU programmeren, zijn er ook andere technologieën in opkomst die mogelijk voor dezelfde toepassingen gebruikt kunnen worden. Gezien dit verslag als doel heeft de aspecten van GPU programmeren in kaart te brengen wordt hierbij niet erg diep ingegaan op bijvoorbeeld de architectuur van deze alternatieven. In plaats daarvan zal de nadruk liggen op de vergelijking met de ontwikkelingen op GPU gebied.

9.1 Multi-Core CPU

Tot enige tijd geleden werden de extra transistors, die in overeenstemming met Moore's law konden worden toegevoegd, ingezet om de kloksnelheid van de CPU te verhogen. De grenzen van deze methode voor het verbeteren van de processorprestaties lijken echter langzaam maar zeker dichterbij te komen. Daardoor lijkt de trend op het gebied van CPU ontwikkeling ook te liggen in parallelisme. Dit vertaalt zich in het gebruik van multi-core CPUs. Deze ontwikkeling is vergelijkbaar met het gebruik van meerdere vertex en fragment processors in één grafische kaart.

Een voordeel dat deze technologie heeft in vergelijking met GPUs is dat beide processor cores volwaardige algemene verwerkingseenheden zijn waarop alle denkbare bewerkingen mogelijk zijn die op huidige desktop processors toegepast kunnen worden. Hierdoor kunnen paralleliseerbare algoritmes vrijwel altijd en met relatief weinig moeite omgezet worden om gebruik te maken van de extra processor core(s).

Een probleem is echter dat er vrij veel processor cores nodig zijn om de pure rekencapaciteit van huidige generaties GPUs te evenaren. Dit zal met de huidige situatie en vooruitzichten niet op korte termijn gerealiseerd worden. Zeker niet wanneer ook naar de prijs/prestatie factor gekeken wordt. Wel is het zo dat met het gebruik van SSE instructies, in ieder geval ten dele, hetzelfde effect bereikt kan worden als het parallelisme op instructie niveau zoals die in de GPU beschikbaar is.

Een andere factor die een grote rol kan spelen in het verschil van prestatie tussen de (multi-core) CPU en GPU is het grote verschil in geheugen bandbreedte. Dit neemt met de toevoeging van meerdere cores aan een CPU niet toe en kan daardoor nog altijd een bottleneck blijven. Wel is het zo dat, wanneer meerdere processors aangesloten zijn op een geheugenbus, de bandbreedte van deze geheugenbus beter wordt benut doordat deze meer verzoeken te verwerken krijgt en dus minder tijd heeft om inactief te zijn.

9.2 Cell processor

De Cell processor is een vrij nieuwe processor, ontwikkeld door Sony en IBM. Het is in feite een wat meer gespecialiseerdere vorm van de hierboven vermelde multi-core CPUs daar deze niet te vergelijken is met een aantal complete CPUs in één behuizing. De informatie die hier gebruikt wordt is afkomstig van de websites [W24][W25] en afgestemd op de Cell processor zoals deze gepland staat voor gebruik in de nieuwe Playstation 3.

De processor bestaat uit één Power Processing Element (*PPE*) en acht Synergistic Processing Elements (*SPE*) waarvan er standaard één is uitgeschakeld om een hogere productie opbrengst te kunnen realiseren. De PPE is enigszins vergelijkbaar met bestaande PowerPC processors, maar dient daarnaast hoofdzakelijk als ‘manager’ over de SPEs, die het grootste gedeelte van de rekenlast voor hun rekening moeten nemen. Cell processors beschikken over een vergelijkbare rekenkracht en geheugenbandbreedte als de huidige generaties GPUs. Ook zijn er prototypes voor servers gebaseerd op meerdere Cell processors in ontwikkeling.

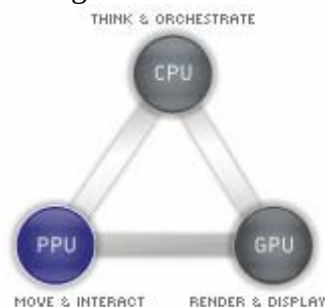
Een voordeel dat de Cell processor heeft ten opzichte van de GPU, is dat de SPEs beter geschikt zijn om algemene taken uit te voeren dan de fragment processors van de GPU. Daarnaast kunnen ze ook afhankelijk van de uit te voeren instructies in serie gebruikt worden om bewerkingen snel en efficiënt uit te voeren. Op de GPU voeren alle fragment processors daarentegen altijd dezelfde instructies tegelijk uit op meerdere data elementen.

Een voordeel van het gebruik van de GPU ten opzichte van de Cell processor is dat de gemiddelde PC tegenwoordig al wel uitgerust is met een GPU en algoritmen die hiervoor gemaakt zijn dus ook op al die PC's werken. Daarnaast kunnen algoritmen op de GPU nog een voordeel hebben wanneer deze bijvoorbeeld beeldbewerkende operaties beschrijven. Wanneer de berekeningen voltooid zijn en deze op het scherm afgebeeld moeten worden, is alle data reeds beschikbaar en kan deze afgebeeld worden. In geval van een Cell processor moet alle data eerst nog naar de grafische kaart getransporteerd worden.

De Cell processor en toepassingen hiervan zijn op dit moment nog in ontwikkeling en dus nog niet commercieel verkrijgbaar. Op dit moment zijn er dan ook nog geen ontwikkelomgevingen of compilers beschikbaar die automatisch gebruik kunnen maken van de rekenkracht van de SPEs. Software ontwikkeling voor een Cell processor is daarmee op dit moment niet minder complex dan voor een GPU. In de toekomst kan hier naar verwachting echter wel verandering in komen.

9.3 PhysX processor

De PhysX processor is momenteel nog in ontwikkeling door AGEIA [W26] en zal de eerste *PPU* of Physical Processing Unit worden. Deze ontwikkeling doet sterk denken aan de geschiedenis van de GPU. Deze is immers ontwikkeld om grafische bewerkingen uit handen van de CPU te nemen. De PhysX processor moet ditzelfde gaan doen voor allerlei natuurkundige bewerkingen. Hierbij valt te denken aan dynamica van vloeistoffen, kleding, of rondvliegend puin na een explosie. De processor is dan ook ontworpen met de computerspel industrie als voornaamste doelgroep. Deze kunnen dan (een gedeelte van) de fysica berekeningen in game engines overdragen aan de PhysX processor. Het model van de rolverdeling dat AGEIA hiermee voor ogen heeft is te zien in Figuur 53.



Figuur 53 De rolverdeling binnen een computer met een PhysX processor [W26]

Een PCI-32 insteekkaart moet deze PhysX processor gaan huisvesten. Hiermee ligt meteen vast dat deze processor maximaal slechts 133 MB/s ($33.33 \text{ MHz} \times 32 \text{ bit} \times 1 \text{ byte} / 8 \text{ bit}$) bandbreedte kan gebruiken voor communicatie met de CPU. Grafische kaarten in een PCI-Express slot hebben hiervoor echter maximaal 4GB/s beschikbaar. Dit verschil kan, voor toepassingen waarbij hoge bandbreedte van en naar de processor noodzakelijk is, grote gevolgen hebben voor de totale performance.

De geheugenbandbreedte op de kaart zelf zal vergelijkbaar zijn, gezien hiervoor GDDR3 gebruikt wordt. Een verschil zit echter wel in de hoeveelheid beschikbaar geheugen. Op de PhysX insteekkaart zal aanvankelijk 128 MB beschikbaar zijn, terwijl een mainstream grafische kaart tegenwoordig 256 MB tot zijn beschikking heeft. High-end grafische kaarten hebben zelfs al enige tijd beschikking over 512 MB en de nieuwste modellen zelfs over 1024 MB

Om de PhysX processor te programmeren, is een API beschikbaar gesteld. Deze API lijkt op dit moment nog sterk toegespitst te zijn op de fysische bewerkingen waarvoor het ontworpen is, en in mindere mate te gebruiken voor andere, algemene toepassingen. Met het in de handel brengen van de processor, dat aan het einde van 2005 gepland staat, zal hier naar verwachting voorlopig nog geen verandering in komen.

Mocht dit in de toekomst wel het geval zijn, dan is het nog de vraag of de PPU net zo populair zal worden als de GPU op dit moment is. Dit met het oog op de toepasbaarheid van programma's die hiervoor geschreven zijn. Een programma geschreven om gebruik te maken van de rekenkracht van de GPU zal tegenwoordig op de gemiddelde PC bijvoorbeeld al uitgevoerd kunnen worden. Daarbij komt dat er ook al verscheidene succesvolle pogingen gedaan zijn om fysica berekeningen te doen op de GPU, zie bijvoorbeeld [25]. Ook heeft NVIDIA samen met Havok, één van de meest vooraanstaande producenten van fysica middleware, in 2005 een onderzoeksproject gestart naar verwerking van fysica op de GPU. Met de resultaten van dit onderzoek is de Havo SDK uitgebreid met GPU acceleratie welke zelfs gebruik kan maken van meerdere grafische GPUs, al dan niet in SLI modus. Deze worden dan waar nodig ingezet voor grafische en/of fysische bewerkingen. Het is dus goed mogelijk dat een processor, puur toegewijd op deze fysica berekeningen, geen toekomst heeft en dat in plaats daarvan de taak overgenomen zal worden door de GPU.

10 De Toekomst

Naast de in Hoofdstuk 1 vermelde ontwikkelingen op het gebied van rekenkracht per processor gaan ook de ontwikkelingen rondom het gebruik van meerdere GPUs in één systeem met grote stappen vooruit. Het gebruik van twee GPUs in één systeem is al goed mogelijk en sinds kort worden ook Quad SLI systemen met vier GPUs ondersteund door NVIDIA. Dit biedt grote mogelijkheden voor het gebruik van één enkele PC voor berekeningen waarvoor normaliter specialistische supercomputers nodig zijn.

Verder zijn er grote ontwikkelingen aangekondigd op het gebied van de GPU architectuur en de programmeerbaarheid hiervan. Met Direct3D 10 op komst zijn de ook mogelijkheden van de 4^e generatie programmeerbare GPUs en daarmee Shader Model 4.0 gedefinieerd. Een overzicht van een groot deel van de veranderingen is terug te vinden in Tabel 11. De hierin vermelde waarden zijn de minimale specificaties waar een GPU aan moet voldoen om Shader Model 4.0 compatibiliteit te mogen claimen.

Daarnaast is de introductie van de *Unified Shader* architectuur erg belangrijk. Hiermee verdwijnt het hardwarematige verschil tussen een vertex processor en een fragment processor. Deze techniek wordt ook al toegepast in de ATI videok kaart, die in de XBOX 360 is geïntegreerd. Momenteel kan een algoritme erg zwaar leunen op de rekenkracht van de fragment processors terwijl de vertex processors vrijwel niets te doen hebben. Met deze technologie worden altijd alle processors optimaal benut.

Feature	SM 1.1, 2001	SM 2.0, 2002	SM 3.0, 2004 ⁶	SM 4.0, 2006
Instruction slots ⁷	128	256	≥512	≥64K
	4+8 ⁸	32+64 ³	≥512	
Constant registers ²	≥96	≥256	≥256	16x4096
	8	32	224	
Temporary registers ²	12	12	32	4096
	2	12	32	
Input registers ²	16	16	16	16
	4+2 ⁹	8+2 ⁴	10	32
Render targets	1	4	4	8
Samplers	8	16	16	16
Textures ²			4	128
	8	16	16	
2D texture size			2Kx2K	8Kx8K
Integer operations				✓
Load operation				✓
Sample offsets				✓
Transcendental operations ²	✓	✓	✓	✓
		✓	✓	
Derivative operation			✓	✓
Flow control ²		Static	Static/Dynamic	Dynamic
			Static/Dynamic	

Tabel 11 Vergelijking mogelijkheden Shader Model 4.0 met oudere versies [23]

⁶ Specificatie vrijgegeven in 2002, hardware in 2004

⁷ Stippellijn scheid vertex shader (boven) en pixel shader (beneden)

⁸ Texture lookups + rekenkundige instructies

⁹ Texture + Color registers

Een ander belangrijke vernieuwing is de toevoeging van een nieuw programmeerbaar element: de *Geometry Shader*. In de geometry processor kan, op basis van alle vertices van een geometrie primitieve (bijvoorbeeld een driehoek of lijn) en alle aangrenzende vertices, bepaald worden of de primitieve verder verwerkt moet worden. Ook kunnen op basis van deze gegevens nieuwe primitieven gedefinieerd worden. Deze shader wordt per primitieve éénmaal uitgevoerd.

Tenslotte kunnen er op het gebied van debug en profile applicaties voor de GPU nog grote ontwikkelingen verwacht worden, zeker voor het Linux platform. Zoals in Hoofdstuk 6 is verteld zijn er hiervoor op dit moment nog weinig tools van hoge kwaliteit beschikbaar. Door verscheidene ontwikkelaars is hiervoor echter binnen afzienbare tijd beterschap beloofd. Uiteraard wordt er nog sterk doorontwikkeld op de bestaande applicaties waardoor de ontwikkelaar continu nieuwe mogelijkheden tot zijn beschikking krijgt om hem te ondersteunen in het ontwikkelproces.

11 Conclusies en aanbevelingen

Uit het onderzoek is gebleken dat GPUs in de laatste jaren verwerkingssnelheden hebben ontwikkeld, die vele malen hoger liggen dan die van de snelste CPUs van Intel of AMD. Daarbij zijn ze geëvolueerd van vrij eenzijdige, specialistische processors naar meer generiek te programmeren stream processors. De zinvolle toepassingsgebieden van deze technologie laten zich dan ook zeker niet beperken tot grafische toepassingen waarbij indrukwekkende effecten op het scherm gegenereerd worden. Ook in andere applicaties wordt de GPU al succesvol gebruikt. Voorbeelden hiervan zijn simulaties, zoals die van fysische modellen, en digitale signaalverwerking, zoals die van audio of MRI scans.

Voor het platform onafhankelijk aanwenden van rekenkracht van GPUs kan het beste gebruik worden gemaakt van Cg wanneer vast staat dat een NVIDIA GPU gebruikt zal worden. Dit vanwege de hogere maximaal haalbare performance en uitgebreidere, praktischere runtime library. Wanneer het ontwikkelproces echter sterk leunt op de in dit verslag genoemde ontwikkelapplicaties of wanneer een GPU van een andere fabrikant gebruikt zal worden, geniet GLSL de voorkeur. In het geval optimale performance niet noodzakelijk is of wanneer er geen of onvoldoende kennis van GPU programmeren is, kan voor GPGPU toepassingen ook overwogen worden om de programmeertaal Sh te gebruiken. Hiermee kunnen de details van de grafische API en specifieke GPU programmeertalen geabstraheerd worden.

Uit het onderzoek is verder naar voren gekomen dat er verscheidene ontwikkelapplicaties voor de GPU beschikbaar zijn. Deze kunnen assisteren in het debuggen, profileren of ontwikkelen van een GPU toepassing. De laatstgenoemde groep kan gebruikt worden voor het maken van een prototype. Het prototype resulteert in een shader, die rechtstreeks in de beoogde toepassing kan worden opgenomen. Het grootste gedeelte van de beschikbare applicaties is gericht op de game-industrie, en daarmee op het gebruik in grafische toepassingen. Ook zijn veel applicaties nog niet op andere platformen dan Windows beschikbaar. Desondanks kunnen zelfs op Linux gebaseerde GPGPU toepassingen nog steeds baat hebben bij het gebruik van de ontwikkelapplicaties. Dit mede gezien het feit dat de shaders zelf platform onafhankelijk zijn.

Voor optimale performance is gebleken dat diepgaande kennis over de architectuur en werking van de GPU belangrijk is. Met die kennis zijn zwakke punten en eigenaardigheden bekend, en kunnen deze in veel gevallen vermeden worden. Dit wordt bereikt door optimaal, en in sommige gevallen creatief, gebruik te maken van alle mogelijkheden van de architectuur en de daarvoor beschikbare API.

Verder blijkt uit het onderzoek dat het op verschillende wijzen mogelijk is om meerdere GPUs in één systeem te gebruiken. De verschillende vormen van SLI maken het mogelijk om de extra toegevoegde rekenkracht op transparante wijze te benutten. Wanneer dit door de aard van het probleem niet mogelijk is, kan de programmeur zelf de bewerkingen verdelen over meerdere, onafhankelijk gebruikte GPUs. Vaak levert deze laatste optie het beste resultaat.

Tenslotte kan geconcludeerd worden dat de pure rekenkracht per GPU ongeveer een factor drie sneller toeneemt dan bij de CPU het geval is. Daarbij geven de specificaties voor volgende generaties aan dat de programmeerbaarheid van GPUs alleen nog maar toeneemt. Het is in ieder geval duidelijk, dat de uitspraak van Tim van Hook: *“All processors aspire to be general-purpose”* [29] zeker voor de GPU lijkt te kloppen.

Referenties

- [1] **Cg Toolkit, User's Manual – A Developer's Guide to Programmable Graphics**, NVIDIA Corporation, January 2004
- [2] **Metaprogramming GPU's with Sh**, Michael McCool, Stefanus Du Toit, A. K. Peters, Ltd, 2004
- [3] **GPU Gems**, Randima Fernando, Addison Wesley, March 2004
- [4] **GPU Gems 2**, Edited by Matt Phar and Randima Fernando, Addison Wesley, March 2005
- [5] **The Cg Tutorial**, Randima Fernando, Mark J. Kilgard, Addison Wesley, February 2005
- [6] **OpenGL Programming guide**, Architecture Review Board, Addison Wesley
- [7] **OpenGL Graphics System: A specification**, Mark Segal, Kurt Akeley, Version 2.0, October 2004
- [8] **OpenGL Shading Language**, Randi J. Rost, Addison Wesley, February 20, 2004
- [9] **3D Games – Real-time Rendering and Software Technology**, Alan Watt, Fabio Policarpo, Volume one, Addison-Wesley, 2001
- [10] **Leren Communiceren**, Michaël Steehouder, Wolters-Noordhoff, mei 1998
- [11] **Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis**, Chris J. Thompson, Sahngyun Hahn, Mark Oskin. University of Washington
- [12] **Shader Model 3.0**, Ashu Rege, NVIDIA Developer Technology Group, GPU Jackpot 2004 presentations
- [13] **Evolution of the Graphical Processing Unit**, Thomas Scott Crow, University of Nevada, December 2004
- [14] **Streaming Architectures and GPUs**, Ian Buck, Bill Dally, Pat Hanrahan, Stanford University, February 2004
- [15] **Accelerating Double Precision FEM Simulations with GPU's**, Dominik Goddeke, Robert Strzodka, Stefan Turek, University of Dortmund, Caesar Research Centre
- [16] **Cg Reference Manual**, NVIDIA Corporation, August 2005
- [17] **GPU Programming Guide**, NVIDIA Corporation, August 2005
- [18] **Optimizing the graphics pipeline**, Matthias M Wloka, NVIDIA Corporation, Eurographics conference 2004
- [19] **Fast texture downloads and readbacks using Pixelbuffer Objects in OpenGL**, NVIDIA Corporation, August 2005
- [20] **GL_EXT_framebuffer_object OpenGL extension**, Architecture Revision Board, September 2005
- [21] **GL_EXT_pixel_buffer_object OpenGL extension**, NVIDIA Corporation, Ralf Bierman, Derek Cornish et al., March 2004
- [22] **Release notes for NVIDIA OpenGL Shading Language support**, NVIDIA Corporation, August 2004
- [23] **The Direct3D 10 System**, David Blythe, Microsoft Corporation, 2006
- [24] **GPU Computation strategies and tricks**, Ian Buck, Siggraph 2005
- [25] **GPGPU: Lessons learned**, Mark Harris, Game Developers Conference 2006
- [26] **NVIDIA OpenGL update**, Simon Green, Game Developers Conference 2006
- [27] **OpenGL performance tools**, Jeff Kiel, Game Developers Conference 2006
- [28] **GPU Performance Optimization with NVPerfHUD**, NVIDIA Corporation, May 2006
- [29] **Keynote**, Tim van Hook, Graphics Hardware 2001

Websites

- [W1] **GPGPU.org**
<http://www.gpgpu.org/>
- [W2] **GLSL Tutorial**
<http://www.lighthouse3d.com/opengl/glsl/index.php?intro>
- [W3] **OpenGL Architecture Review Board**
<http://www.opengl.org/about/arb/overview.html>
- [W4] **Cg and HLSL FAQ, Alex Calvrak, June 2003**
<http://www.fusionindustries.com/content/lore/code/articles/fi-faq-cg.php3>
- [W5] **GLSL, Wikipedia, July 2005**
<http://en.wikipedia.org/wiki/GLSL>
- [W6] **Sh, Embedded Metaprogramming Language website**
<http://libsh.org/>
- [W7] **Brook for GPU, Stanford University**
<http://graphics.stanford.edu/projects/brookgpu/>
- [W8] **Keynote, Tim van Hook, Graphics Hardware 2001**
- [W9] **Moore's Law, Webopedia**
http://www.webopedia.com/TERM/M/Moores_Law.html
- [W10] **GeForce 7800 GTX, Tom's Hardware Guide**
http://graphics.tomshardware.com/graphic/20050622/NVIDIA_7800_gtx-01.html
- [W11] **The P4 600 Series and Extreme Edition 3.73 GHz, Tom's Hardware Guide**
<http://www.tomshardware.com/cpu/20050221/>
- [W12] **NVIDIA SDK, NVIDIA Corporation**
http://developer.NVIDIA.com/object/sdk_home.html
- [W13] **Fx Composer, NVIDIA Corporation**
http://developer.NVIDIA.com/object/fx_composer_home.html
- [W14] **NVPerfHUD, NVIDIA Corporation**
http://developer.NVIDIA.com/object/nvperfhud_home.html
- [W15] **NVShaderPerf, NVIDIA Corporation**
http://developer.NVIDIA.com/object/nvshaderperf_home.html
- [W16] **gDEBugger**
<http://www.gremedy.com/>
- [W17] **GLIntercept**
<http://glintercept.nutty.org/>
- [W18] **IMDebug**
<http://www.billbaxter.com/projects/imdebug/>
- [W19] **ATI RenderMonkey**
<http://www.ati.com/developer/rendermonkey/>
- [W20] **NVPerfKit**
http://developer.NVIDIA.com/object/nvperfkit_home.html
- [W21] **Shader Designer**
<http://www.typhoonlabs.com/index.php?action=developer.htm>
- [W22] **GLTrace**
<http://www.hawksoft.com/gltrace/>
- [W23] **BuGLE**
<http://bugle.sourceforge.net/>
- [W24] **Cell Processor, Wikipedia**
[http://en.wikipedia.org/wiki/Cell_\(microprocessor\)](http://en.wikipedia.org/wiki/Cell_(microprocessor))
- [W25] **Cell architecture explained, Nicholas Blachford, version 2, 2005**

-
- [W26] http://www.blachford.info/computer/Cell/Cell0_v2.html
PhysX Processor, AGEIA
<http://www.ageia.com/>
- [W27] **GPUBench**
<http://graphics.stanford.edu/projects/gpubench/>
- [W28] **GeForce 7950 GX2 FAQ**, NVIDIA
http://www.nvidia.com/page/7950_faq.html

Begrippenlijst

#

1GL

Eerste generatie programmeertaal. Hiermee wordt programmeren in machine instructies aangeduid.

2GL

Tweede generatie programmeertaal. Deze vorm van programmeertaal is vergelijkbaar met een *1GL*, hij is echter syntactisch enigszins aangepast zodat het voor mensen beter leesbaar en dus beter te programmeren is. Alle procedures welke een programmeur zou willen uitvoeren, moeten instructie voor instructie gedefinieerd worden. Een CPU variant hiervoor is bijvoorbeeld Assembly.

3GL

Derde generatie programmeertaal. In deze vorm hoeft de programmeur zich niet meer te bekommeren om aparte instructies maar kunnen applicaties op een hoger, procedureel niveau gedefinieerd worden. Voorbeelden hiervan zijn bijvoorbeeld C(++) voor de CPU en Cg of GLSL voor de GPU.

4GL

Vierde generatie programmeertaal. Met deze generatie programmeertalen kan de programmeur zijn applicaties op een nog hoger conceptueel niveau definiëren. In plaats van te beschrijven hoe iets moet gebeuren, wordt er enkel beschreven wat er moet gebeuren. De vele beschikbare libraries voor C(++) doen de grens tussen een 3GL en 4GL vervagen. Voor GPU programmeren zouden bijvoorbeeld Sh en Brook als 4GL beschouwd kunnen worden.

A

Alpha

De mate van transparantie van een *pixel*.

Alpha-test

Een test die aan het einde van de *pipeline* gedaan wordt om te zien of de *alpha* waarde van het *fragment* aan gestelde eisen voldoet.

AFR

Staat voor Alternate Frame Rendering. De aanduiding voor een operatie modus van *SLI* waarin twee grafische kaarten afwisselend één frame renderen.

API

Application Programming Interface. Een set van *library* functies welke aangeroepen kunnen worden door een programma.

ARB

Architecture Revision Board. Commissie van organisaties samen verantwoordelijk voor het beheren en wijzigen van de *OpenGL* specificatie.

B

Backend

Een beoogde architectuur voor een bepaalde operatie zoals bijvoorbeeld het compileren van een programma.

Branching

Het nemen van data afhankelijke paden in een programma zoals een if-else statement.

Buffer

Een plek in het geheugen. In grafische context: het geheugen waar de getekende beelden in opgeslagen wordt.

C

Cell

Een processor gemaakt door Sony en IBM die gebruikt gaat worden in de nieuwe Playstation 3 en is opgebouwd uit één *PPE* en acht *SPEs*.

Cg

C for Graphics/*GPU*, programmeertaal waarin *shaders* geschreven kunnen worden welke via *OpenGL* of *DirectX* gebruikt kunnen worden. Ontwikkeld door *NVIDIA* [1].

Cg profiel

Met profielen wordt in Cg een set mogelijkheden voor GPU programma's aangeduid. Deze zijn vergelijkbaar met het concept van *Shader Models*.

Clock Cycle

Een (zeer kleine) hoeveelheid tijd die als atomaire tijdseenheid gehanteerd wordt. Een operatie kan dus nooit minder dan één clock cycle in beslag nemen.

Convolutie Filter

Een bepaalde bewerkingsfilter die met behulp van een matrix uitgevoerd wordt op een beeld om een bepaald effect te krijgen zoals blur, edge detection, etc.

COTS

Commercial Off The Shelf, een benaming die gebruikt wordt voor producten die reeds ontwikkeld en commercieel verkrijgbaar zijn.

CPU

Central Processing Unit, De processor waar programma's op uitgevoerd worden.

Crossfire

Een technologie van ATI om meerdere grafische kaarten voor de gebruiker transparent te laten samenwerken alsof het één snellere kaart is.

D

DirectX

Een *API* voor directe toegang tot hardware componenten dat door Microsoft ontwikkeld is. Het Direct3D gedeelte hiervan is vergelijkbaar met *OpenGL*.

Depth-test

Een test die aan het einde van de *pipeline* gedaan wordt om te zien of de diepte van het *fragment* aan gestelde eisen voldoet. Zo kan getest worden of een te tekenen pixel voor een reeds getekende pixel moet liggen en de eerste overschrijven.

DMA

Staat voor Direct Memory Access. Een techniek welke het mogelijk maakt om randapparatuur, zoals in dit geval de videokaart, data van en naar het geheugen te kopiëren zonder directe sturing van de CPU.

F

FBO

Zie *Framebuffer object*.

FPS

Frames per Second, zie *framerate*.

Fragment

Een 'potentiële *pixel*'. Deze benaming wordt gegeven aan pixels die nog verwerkt worden. Hiervan is het nog niet zeker of deze daadwerkelijk in het uiteindelijke beeld terecht komt doordat er aan het einde van de pipeline nog aan verscheidene tests zoals *alpha*-, *stencil*- en *depth-test* onderworpen wordt. Pas als het door alle tests heen is gekomen en in het *framebuffer* terecht is gekomen wordt het een pixel genoemd.

Fragment processor

Het stuk hardware dat onderdeel uitmaakt van de *GPU*, dat in staat is om een *fragment program* uit te voeren.

Fragment Program

Zie pixel shader.

Fragment Shader

Zie pixel shader.

Frame

Een beeld volledig getekend beeld.

Framebuffer

Een *buffer* waar een *frame* in kan worden opgeslagen, na een tekenoperatie staat het resultaat in een framebuffer.

Framebuffer object

Een extensie op *OpenGL*, o.a. te gebruiken om de uitvoer van een *pixel shader* rechtstreeks in een *texture* te plaatsten.

Framerate

De snelheid waarmee het beeld wordt geüpdate in hertz. Men zegt 25 hertz als een programma of camera 25 *frames* per seconde levert.

FX

Een door NVIDIA gedefinieerd formaat voor een materiaal of effect dat voor een model gebruikt kan worden. Deze effecten zijn niet equivalent aan shaders, maar bevatten doorgaans wel één of meerdere shaders.

G

Gather-operation

Een bewerking die voor het berekenen van een *pixel* informatie uit andere (nabij gelegen) pixels uitleest en verwerkt. Zie ook *scatter-operation*.

Geometry Shader

Een nieuwe programmeerbare stap in de *grafische pipeline* geïntroduceerd in *Shader Model 4.0*. Hierin kan op basis van alle vertices van een geometrie primitieve (bv. Driehoek of lijn) en alle aangrenzende vertices bepaald worden of deze primitieve verder verwerkt moet worden. Ook kunnen op basis van deze gegevens nieuwe primitieven gedefinieerd worden. Deze *shader* wordt per primitieve éénmaal uitgevoerd.

GLSL

OpenGL Shading Language, een taal om de GPU te programmeren die ontwikkeld is door het ARB als onderdeel van *OpenGL*.

GLX

OpenGL for X, *library* functies die het mogelijk maken om *OpenGL* te gebruiken binnen X.

GPU

Graphical Processing Unit, de processor van een *videokaart*, of gebruikt om te verwijzen naar de een *videokaart* in zijn geheel.

GPGPU programming

General Purpose GPU programming, het gebruik van *grafische hardware* voor uitvoeren van niet noodzakelijk grafische berekeningen.

Grafische hardware

Zie videokaart en GPU.

Grafische pipeline

Zie Rendering Pipeline.

H

HLSL

High Level Shading Language, de afsplitsing van Microsoft van hun samenwerking met *NVIDIA* om een programmeertaal te ontwerpen. Geschikt om *shaders* te maken voor *DirectX*.

K

Kanaal

Zie Kleurenkanaal.

Kleurenkanaal

Een bepaalde kleur, bijvoorbeeld rood. Met het rode kleurkanaal worden alleen de rode *pixelwaardes* uit een beeld bedoeld.

L

Library

Een reeks verwante functies, klassen en/of structures in een library-bestand welke vervolgens gebruikt kunnen worden door programma's die deze library willen gebruiken.

Lookups

In grafische context wordt hiermee het opzoeken/opvragen van de pixelwaardes op een bepaalde locatie van een texture bedoeld.

M

Matrix

Een 2 dimensionale array met een bepaalde hoogte en breedte. Bijvoorbeeld een 3x3 matrix welke 9 waarden bevat.

MIMD

Multiple Instructions Multiple Data. Een architectuur van deze vorm kan meerdere instructies op een (grote) dataset uitvoeren.

N

NVIDIA

Grote fabrikant van onder meer *videokaarten*, ontwikkelaar van *Cg*.

O

Offscreen buffer

Een buffer waarin grafica getekend wordt, maar niet direct op het scherm zichtbaar is. Bijvoorbeeld een *p-buffer*.

OpenGL

Open Graphics *Library*. Specificatie van een *API* voor het genereren van 2D en 3D beelden. Vrijwel alle grafische kaarten kunnen met deze API worden aangesproken.

OpenGL extensie

Een uitbreiding op de basisfunctionaliteit van *OpenGL*. Nieuwe mogelijkheden van *GPUs* of toepassingen hiervan kunnen in eerste instantie aangesproken worden door gebruik te maken van deze extensies. Deze kunnen ook door de fabrikanten van grafische kaarten zelf gemaakt worden. Wanneer een uitbreiding door alle fabrikanten ondersteund kan worden, maakt deze kans om door de *ARB* opgenomen te worden in de *OpenGL* specificatie.

OpenGL state

Een grote hoeveelheid variabelen die de huidige staat van de *OpenGL* omgeving bevatten. Hierin worden bijvoorbeeld transformatie matrices, licht, materiaal en mist parameters opgeslagen.

P

P-Buffer

Een buffer in het geheugen waar geen venster aan gekoppeld is.

PBO

Zie *Pixelbuffer Object*.

Pipeline

Zie *Rendering Pipeline*.

Pixel

Een punt van een beeld.

Pixelbuffer Object

Een extensie op *OpenGL*, o.a. te gebruiken om transfer van pixel data in *textures* middels *DMA* te laten plaatsvinden.

Pixel Shader

Programma welke de standaard berekeningen van *OpenGL* of *DirectX* om de kleur van een *pixel* bepalen vervangt.

Pixel Waardes

De waarde van een pixel in een bepaald kleurformaat. In een monochroom beeld zou wit bijvoorbeeld 255 voorstellen en zwart 0. Bij een RGB (red, green, blue) formaat staat (0,255,0) voor volledig groen.

PPE

Power Processing Element, onderdeel van de *Cell* processor dat qua mogelijkheden vergelijkbaar is met de huidige x86 processors. Deze stuurt echter het grootste gedeelte van de aangeboden berekeningen door aan de *SPEs* die in diezelfde *Cell* processor verwerkt zijn.

Profiel

Zie *Cg profile*.

PPU

Physics Processing Unit, een processor die geheel is toegespitst op natuurkundige berekeningen zoals het verwerken van krachten op objecten.

Q

Quadro

Een krachtig type *videokaart* van het merk NVIDIA, gericht op de zakelijke (high-end) markt.

R

Raster eenheid

Onderdeel van de *grafische pipeline* die de lijnen en vlakken van een 3D wereld opdeelt in *pixels* welke uiteindelijk op het scherm getekend kunnen worden. Zorgt tevens voor interpolatie van bijvoorbeeld *texture coördinaten* welke tussen twee of meerdere *vertices* liggen.

Renderen

Het tekenen van bepaalde delen van bijvoorbeeld een OpenGL, een DirectX of een Java2D scene.

Render pass

Het verwerken en tekenen van één set samenhangende geometrie.

RTT

Staat voor Render To Texture. Hiermee wordt de operatie bedoeld waarmee het resultaat van een tekenoperatie direct in een *texture* wordt weggeschreven.

Rendering Pipeline

De volgorde van acties de *videokaart* doorloopt bij het renderen, waarvan enkele acties aan te passen zijn door middel van de *shaders*.

RGBA: Red Green Blue Alpha

Een type colorspace welke iedere kleur kan vormen door een bepaalde waarde in te vullen voor de R, G en B, (rood, groen en blauw) van de kleur. *Alpha* geeft de mate van transparantie weer.

S

Scatter-operation

Een bewerking die bij het verwerken van een *pixel* informatie naar andere (nabij gelegen) pixels wegschrijft. Zie ook *gather-operation*.

SDK

Software Development Kit, een verzameling bestanden van verschillende types die nodig zijn om software in een bepaalde taal en/of met een bepaalde *API* te ontwikkelen.

SFR

Staat voor Split Frame Rendering. De aanduiding voor een operatie modus van *SLI* waarin twee grafische kaarten beiden een helft van één frame renderen.

Sh

Een metaprogrammeertaal ontworpen op de Universiteit van Waterloo welke tracht GPU programmeren mogelijk te maken met minimale kennis van hoe een grafische omgeving zoals OpenGL opgezet moet worden.

Shaders

Kleine programma's die delen van de *rendering pipeline* aanpassen. Zie *vertex shader* en *pixel shader*.

Shader Model

Begrip geïntroduceerd in de *DirectX* specificatie om de mogelijkheden en beperkingen van verschillende *GPU* generaties aan te duiden.

SIMD

Single Instruction Multiple Data. Een architectuur van deze vorm voert dus één enkele instructie op een (grote) dataset uit. Wanneer er op de dataset meerdere instructies uitgevoerd moeten worden vergt dit dus meerdere iteraties.

SLI

Staat voor Scalable Link Interface. Een technologie van NVIDIA om meerdere grafische kaarten voor de gebruiker transparent te laten samenwerken alsof het één snellere kaart is.

SPE

Synergistic Processing Element, een *Cell* processor heeft meerdere van deze elementen aan boord. Deze elementen vormen samen het grootste gedeelte van de rekencapaciteit van de *Cell* processor.

Stencil-test

Een test die aan het einde van de *pipeline* gedaan wordt om te zien of het *fragment* al dan niet aan gestelde voorwaarden voldoet en het *framebuffer* mag veranderen.

Stream programmeren

Een vorm van programmeren op een *SIMD* of *MIMD* manier. Een stream programma voert dus één of meerdere instructies uit op een dataset en kan dit (gedeeltelijk) parallel doen.

T

Texture

Beeld in het geheugen van de *GPU*.

Texture coördinaten

Coördinaten die aangeven welke positie van de *texture* gebruikt moet worden voor het tekenen van een *pixel*

U

Unified shader

Een nieuwe ontwikkeling waarbij het hardwarematige onderscheid tussen de *fragment processor* en de *vertex processor* verdwenen is. Een unified shader kan zowel als fragment als vertex processor ingezet worden.

V

VBO

Zie *Vertex buffer object*.

Vertex

Een punt in een 3D wereld, bevat minimaal x-, y- en z-coördinaten en kan verder nog informatie als een normaal, kleur of texture coördinaten bevatten.

Vertex buffer object

Een extensie op *OpenGL*, o.a. te gebruiken om transfer van vertex data middels *DMA* te laten plaatsvinden.

Vertex processor

Het stuk hardware dat onderdeel uitmaakt van de *GPU*, dat in staat is om een *vertex program* uit te voeren.

Vertex program

Zie Vertex shader

Vertex Shader

Programma welke op de *GPU* gedraaid kan worden. Vervangt het vertex processing deel van de standaard *rendering pipeline* van *OpenGL* of *DirectX*.

Videokaart

Hardware waarop zich een *GPU* bevindt, waarmee grafische bewerkingen versneld worden en waarop zich tevens videogeheugen bevindt waarin bijvoorbeeld textures opgeslagen kunnen worden.

X

X

X is een programma voor Linux/Unix welke het mogelijk maakt om grafisch te werken. Hierin worden onder andere een grafische desktop geladen (window manager) en kan gebruikt worden om *OpenGL* weer te geven dankzij *GLX*.

Z

Z-buffer

Het *buffer* waarin diepte informatie wordt opgeslagen.

Z-culling

De procedure waarbij de diepte van een te verwerken *fragment* vergeleken wordt met de corresponderende waarde in het *z-buffer*. Er kan bijvoorbeeld gekeken worden of de waarde groter, gelijk of juist kleiner dan de waarde uit het *z-buffer* is. Wanneer deze niet aan de door de programmeur gestelde voorwaarden voldoet, wordt het fragment niet verder verwerkt.

Bijlage A Omschrijving afstudeeropdracht

Vestiging	Hengelo
Opdrachtnr	052050
Afdeling	Combat Systems - Projects & Bids
Studierichting	IT - (Technische) Informatica
Niveau	WO
Achtergrond	De laatste jaren zijn er grote ontwikkelingen geweest op het gebied van GPU's (Graphical Processing Units) in PC's. Bekende GPU voorbeelden zijn de NVIDIA GeForce en ATI Radeon graphics processors die op de grafische kaart in moderne game PC's te vinden zijn. Deze GPU's zijn in de afgelopen generaties steeds beter programmeerbaar geworden dmv. zgn. shader programming. Daarmee kunnen eigen algoritmes op de GPU uitgevoerd worden; niet alleen voor 3D graphics toepassingen, maar ook bv. voor zware parallele wiskundige berekeningen en realtime image processing. Op het gebied van GPU's en het gebruik ervan als snelle co-processor naast de CPU zijn de komende jaren nog grote ontwikkelingen te verwachten.
Titel Opdracht	Onderzoek naar state-of-the-art in GPU technology tbv. high-performance computing en 3D visualisatie
De Opdracht	In deze onderzoeksopdracht moet een literatuurstudie en marktonderzoek uitgevoerd worden naar de state-of-the-art van GPU technologieën en oplossingen. De stand van zaken, trends en roadmaps rondom GPU's, shader programming (zie ook http://developer.NVIDIA.com) en GPGPU (General Purpose Computing on Graphical Processing Units (zie ook www.gpgpu.org)) moet in kaart gebracht worden. Onderzoeksthema's zijn ondermeer: architectuur van GPU's, vergelijking NVIDIA en ATI GPU's, vergelijking shader model programming talen (Cg, HLSL, GLSL) zowel onderling als met C/C++, richtlijnen+tips en tricks voor het optimaal implementeren van algoritmes mbv. shaders, ontwikkelomgevingen onder Windows en Linux tbv. Shader model programming, performance-karakteristieken, -analyse en debug middelen, multi-GPU aspecten, SLI (Scalable Link Interface), advanced graphics techniques (bv. rendering to texture, volume rendering, large model visualisation), en het onderzoeken van GPGPU aspecten (General Purpose Computing on GPUs), incl. vergelijking met traditionele CPU gebaseerde oplossingen. Aan de hand van een aantal voorbeelden uit de Thales praktijk moet vervolgens de opgedane kennis in de praktijk worden gebracht door het implementeren van een aantal representatieve testcases (bv. beeldcorrectie bij radar video display, object detectie in Infrarood video, realtime 3D graphics). De opgedane ervaringen tijdens deze ontdekkingsreis in de fascinerende wereld van GPU programming moeten in een gedegen onderzoeksrapport vastgelegd worden.
Affiniteit	Ontdekkingsreiziger op Internet, realtime 3D graphics, computerhardware, performance analyse, game-engine development, wiskunde
Afdelings-stagecoördinator	Ingo Verhaak
Stagebegeleider	Ronald Boers

Bijlage B Vergelijkingstabel HLSL, GLSL en Cg

Gebaseerd op beschikbare informatie op april 2005. Door Mark Kilgard, hand-out op SIGGRAPH 2005.

	DirectX 9 HLSL	GLSL	Cg
Availability			
Available today	Yes	Yes	Yes
Installation/upgrade requirements	Part of DirectX 9; comes with XP or a free Windows update	May need driver update from hardware vendor for ARB extensions or OpenGL 2.0	User level libraries so no driver upgrade typically required
Time of first release	March 2003, DirectX 9 ship	June 2003, ARB standards approved; implementations in late 2003	December 2002 1.0 release
Current version	DirectX 9.0c	1.1	Cg 1.4
Standard maker	Microsoft	OpenGL Architectural Review Board	NVIDIA
Implementer	Microsoft	Each OpenGL driver vendor	NVIDIA
3D Graphics API Support			
OpenGL	No	Yes	Yes
Direct3D	Yes	No	Yes
One shader can compile for either API	No	No	Yes
OpenGL Specifics			
OpenGL 1.x support	n/a	Needs ARB GLSL extensions	Needs ARB or NV low-level assembly extensions
Multi-vendor ARB_vertex_program support	n/a	No	Yes
Multi-vendor ARB_fragment_program support	n/a	No	Yes
NVIDIA OpenGL extension support	n/a	No	Yes, profiles for fp20, vp20, fp30, vp30, fp40, and vp40
Relationship to OpenGL standard	n/a	Part of core OpenGL 2.0 standard	Layered upon ARB-approved assembly extensions
Access to OpenGL state settings	n/a	Yes	Yes
Open Source OpenGL rendering support (via Mesa)	n/a	No Mesa support yet	Yes, no changes required
Language tied to OpenGL	n/a	Yes	No, API independent

	DirectX 9 HLSL	GLSL	Cg
Direct3D Specifics			
DirectX 8 support	Requires DirectX 9 upgrade but supports DirectX 8-class hardware profiles	n/a	Yes
DirectX 9 support	Yes	n/a	Yes
GPU Hardware Support			
NVIDIA DirectX 9-class GPU's	Yes	Yes	Yes
ATI DirectX 9-class GPU's	Yes	Yes	Yes
3Dlabs DirectX 9-class GPU's	Yes	Yes	Yes
DirectX 8-class GPU's	Yes, with ps1.x and vs1.1 profiles	No	Yes, fp20 and vp20 profiles
Graphics Hardware Feature Support			
Vertex textures	Yes, if hardware supports vs3.0 profile	Yes, if hardware supports	Yes, if hardware supports vp40 profile
Dynamic vertex branching	Yes, if hardware supports vs2.0 profile	Yes, if hardware supports	Yes, if hardware supports vp30 or vp40 profiles
Dynamic fragment branching	Yes, if hardware supports ps3.0 profile	Yes, if hardware supports	Yes, if hardware supports fp40 profile
Fragment depth output	Yes	Yes	Yes
Multiple render targets	Yes, if hardware supports	Yes, if hardware supports	Yes, if hardware supports fp40 profile
1D, 2D, 3D, and cube map texture support	Yes	Yes	Yes
Shadow mapping	Yes	Yes, but needs special shadow texture fetch routines to be well-defined	Yes
Point size output	Yes	Yes	Yes
Clipping support	Output clip coordinate(s)	Output clip position	Output clip coordinate(s)
Texture rectangles	No	Yes, when ARB_texture_rectangle is supported	Yes, when ARB_texture_rectangle is supported
Access to fragment derivatives	Yes, when supported by the fragment profile	Yes	Yes, when supported by the fragment profile
Front and back vertex color outputs for two-sided lighting	No	Yes	Yes, for OpenGL
Front facing fragment shader input	Yes	Yes	Yes, for fp40 profile
Window position fragment shader input	Yes, for ps2 and ps3 profiles	Yes	Yes, for all but fp20 profile

	DirectX 9 HLSL	GLSL	Cg
Language Details			
C-like languages	Yes	Yes	Yes
Language compatibility with Microsoft's DirectX 9 HLSL	Yes	No	Yes
Vertex and fragment shaders written as separate programs	Yes	Yes	Yes
C operators with C precedence	Yes	Yes	Yes
C control flow (if, else, for, do while)	Yes	Yes	Yes
Vector data types	Yes	Yes	Yes
Fixed-point data type	No	No. has fixed reserved word	Yes, fixed
Matrix arrangement	Column major by default	Column major	Row major by default
Non-square matrices	Yes	No	Yes
Data type constructors	Yes	Yes	Yes
Structures and arrays	Yes	Yes	Yes
Function overloading	Yes	Yes	Yes
Function parameter qualifiers: in, out, and inout	Yes	Yes	Yes
Type qualifiers: uniform, const	Yes	Yes	Yes
Shader outputs readable	Yes	No	Yes
Texture samplers	Yes	Yes	Yes
Keyword discard to discard a fragment	Yes	Yes	Yes
C and C++ style comments	Yes	Yes	Yes
C preprocessor support	Yes	Yes	Yes
Vector write masking	Yes	No	Yes
Vector component and matrix swizzling	Yes	No	Yes
Vector ?: operator	Yes	No, Boolean only	Yes
Vector comparison operators	Yes	No, must use lessThan, etc. standard library routines	Yes
Semantic qualifiers	Yes	No	Yes
Array dimensionality	Multidimensional	1D only	Multi-dimensional
Un-sized arrays (dynamic sizing)	No	No	Yes

	DirectX 9 HLSL	GLSL	Cg
Shader Linking Support			
Sub-shader support via interface mechanism	No	No, but keyword reserved	Yes, in Cg 2.0
Separate compilation and linking	D3DX utility library: Fragment Linker	Yes	No, but interfaces provide a structured form of program reconfiguration
Cross domain linking by varying name	No	Yes	No
Standard Library			
Standard function library	Yes	Yes	Yes
Standard function library compatibility with Microsoft's DirectX 9 HLSL	Yes	No	Yes
Operating System Support			
Windows support (98/2000/XP)	Yes	Yes	Yes
Legacy Windows 95 support	No	Yes	Yes
Legacy Windows NT 4.0 support	No	Yes	Yes
Linux	No	Yes	Yes
Mac OS X	No	Yes	Yes (since 1.2)
Shader Meta File Format Support			
Standard shader meta file format	Yes (FX)	No	Yes (CgFX)
Shader meta file format compatible with DirectX 9 Effects format	Yes	n/a	Yes
Specification of multiple techniques for an effect	Yes	n/a	Yes
Virtual machine for CPU shader execution	Yes	No	Yes
Annotations	Yes	n/a	Yes
State assignments	Yes	n/a	Yes
Run-time API	Yes	n/a	Yes
Direct3D loader	Yes	n/a	Yes
OpenGL loader	n/a	n/a	Yes
Free shader meta file viewer available	FX Composer	n/a	FX Composer
Debugging			
Interactive shader debugger	Visual Studio .NET Shader Debugger	No	No

	DirectX 9 HLSL	GLSL	Cg
Documentation			
Specification or definitive documentation	MSDN Direct3D 9 documentation	The OpenGL Shading Language specification	The Cg Language Specification
Tutorial book	Various books	OpenGL Shading Language (Rost)	The Cg Tutorial (Fernando & Kilgard)
User's manual	MSDN Direct3D 9 documentation	No	Cg User's Manual
Japanese documentation available	Yes	No	Yes
Example Code			
Sources of example code	Microsoft DirectX 9 SDK examples	ATI, 3Dlabs, NVIDIA SDK examples	Cg Tutorial examples, NVIDIA SDK examples
Miscellaneous			
Standalone command line compiler	Yes (fxc)	No	Yes (cgc)
Generates human-readable intermediate assembly code	Yes	No	Yes
Supports ILM's OpenEXR halfprecision data type	Yes	Reserved word for half; NVIDIA supports it	Yes
Open source language parser available	No	Yes	Yes
Compiler tied into graphics driver	No	Yes, compiled result depends on end-user hardware and driver version	No
Multi-vendor extension mechanism	No	Yes	No
No-fee redistributable library	Yes, Windows only	n/a	Yes, all platforms
What its name stands for	High Level Shader Language	OpenGL Shading Language	C for Graphics

Bijlage C Resultaten bandbreedte tests

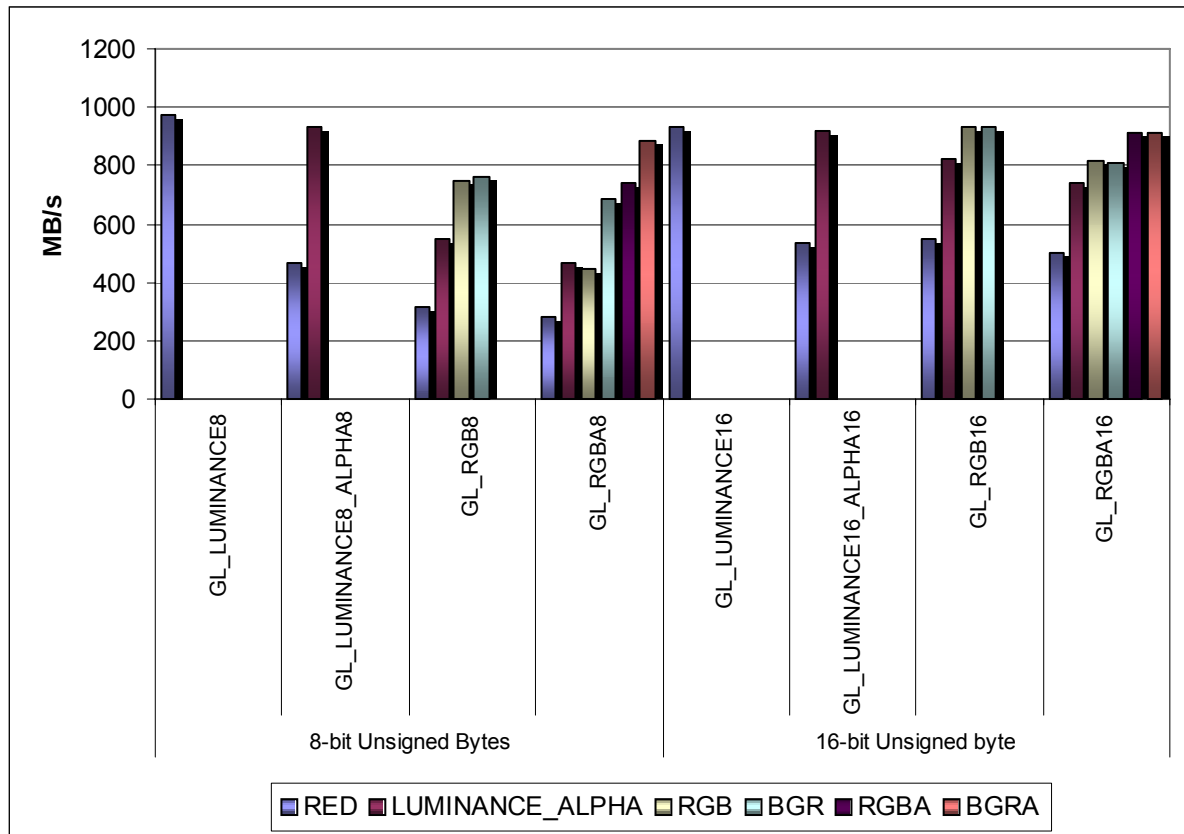
Deze applicatie test voor verschillende datatypes (8-bit en 16-bit unsigned byte, 16-bit en 32-bit float) de transfer van 1 tot 4 kanalen data in textures met van 1 tot 4 kanalen. Hierbij wordt ook gelet op de verschillen tussen transfer in de formaten RGB(A) en BGR(A). De data wordt middels de functie *glTexSubImage* naar de GPU gestuurd. Met *glGetTexImage* wordt deze weer teruggelezen. Uit vroege tests is gebleken dat de eerste transfer significant langer duurt dan alle volgende transfers. Nader onderzoek hiernaar heeft uitgewezen dat dit komt doordat de data na de eerste transfer al in de cache van de CPU aanwezig is. Om dit te vermijden, wordt bij elke transfer een nieuw geheugenblok vrijgemaakt van waaruit de data ingelezen wordt. De metingen worden verricht door de verschillen in de systeemtijd in microseconden voor en na de transfer te meten.

Om de metingen zo nauwkeurig mogelijk te maken, wordt de CPU scheduling van het meetproces aangepast naar een realtime proces met hoge prioriteit. Ook wordt de CPU affiniteit gezet op uitsluitend de eerste CPU, om context switches hierop te voorkomen. Om deze aanpassingen toe te laten, moet de applicatie als root gestart worden.

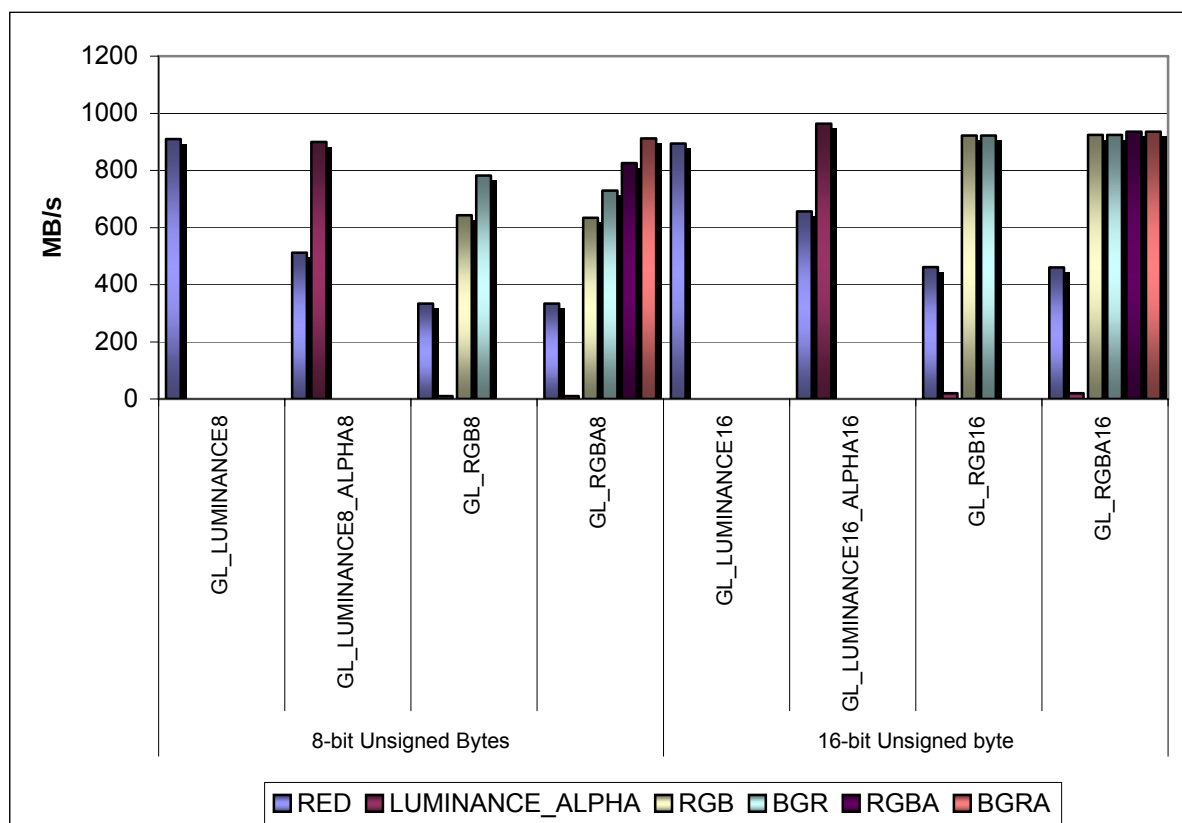
```
#tex_test -?

Usage: ./tex_test <options>
Options
-d, --noDownload
    do not perform download test
-r, --noReadback
    do not perform readback test
-a, --noARB
    do not perform ARB floating point tests
-n, --noNV
    do not perform NV floating point tests
-u, --noUnsigned
    do not perform unsigned byte tests
-w, --width=NUM
    width of texture to transfer (256)
    2 <= NUM <= 4096
-h, --height=NUM
    height of texture to transfer (256)
    2 <= NUM <= 4096
-i, --iterations=NUM
    number of iterations to perform per test
    output is the median of these iterations (100)
    NUM > 0
-o, --output=STRING
    Specifies the type of output
    mbps: transfer bandwidth in MB/s (default)
    ms:   milliseconds per transfer
    us:   microseconds per transfer
```

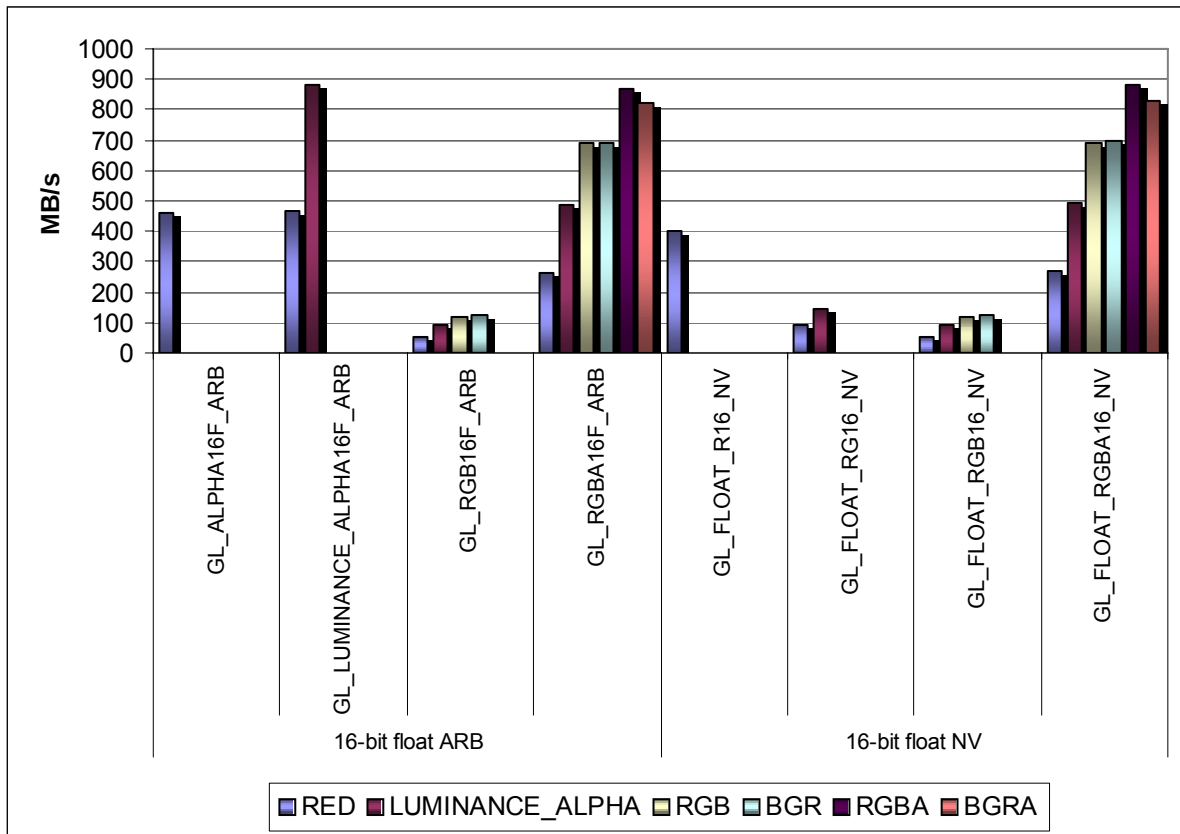
De resultaten van de hieronder vermelde tests zijn verkregen door elke transfer 500 keer uit te voeren. Door de applicatie wordt van deze 500 resultaten de mediaan bepaald om een goede schatting te krijgen van de te verwachten waarde, zonder hierbij beïnvloed te worden door grote uitschieters zoals het geval is bij gebruik van het gemiddelde. De tests zijn uitgevoerd op een GeForce 6600 GT AGP videokaart met texture groottes van 256x256 en 1024x1024.



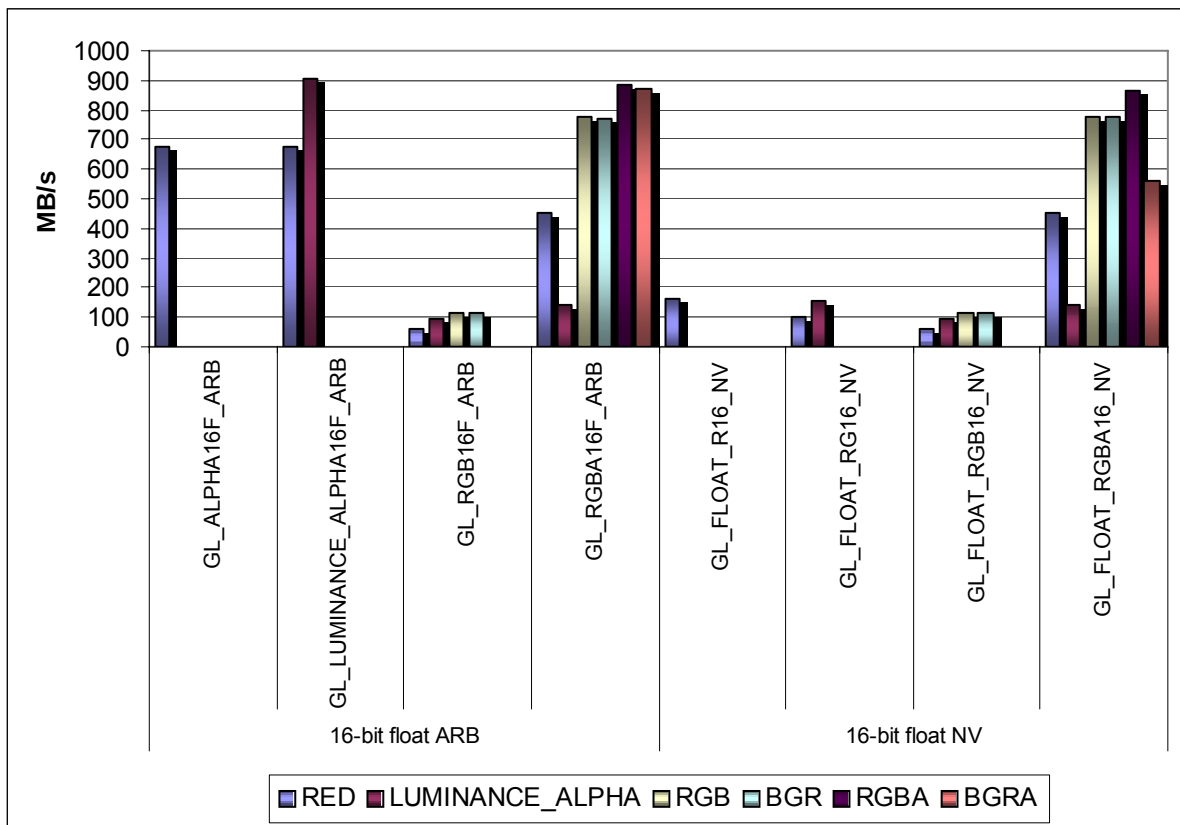
Figuur 54 Unsigned bytes download 1024 x 1024



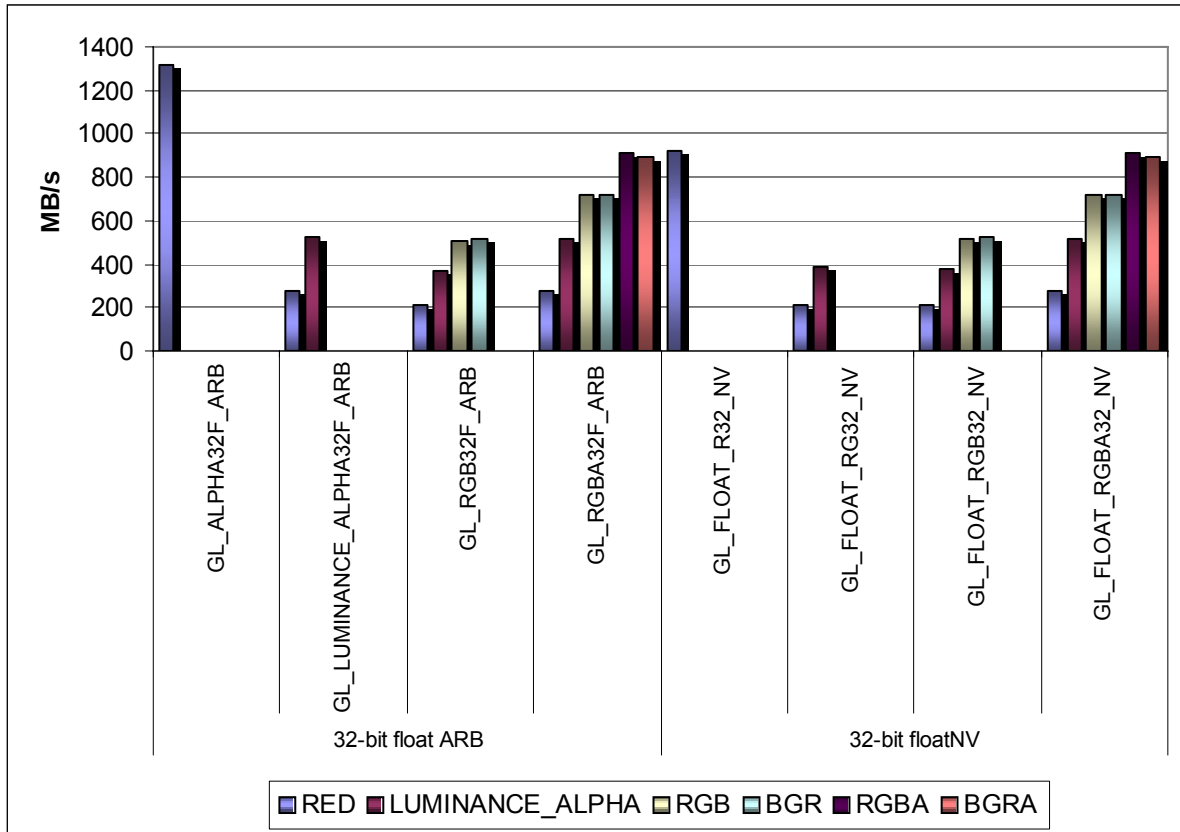
Figuur 55 Unsigned bytes readback 1024 x 1024



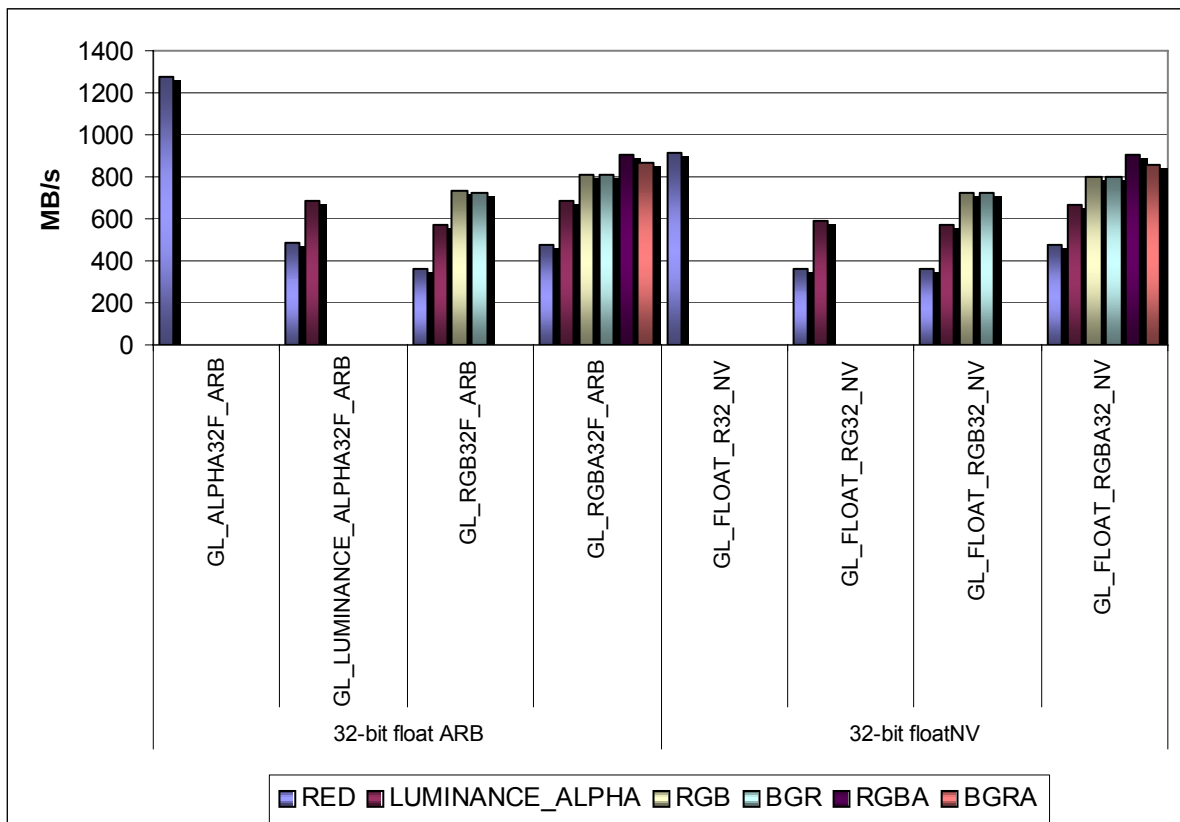
Figuur 56 16-bit float download 1024 x 1024



Figuur 57 16-bit float readback 1024 x 1024



Figuur 58 32-bit float download 1024 x 1024



Figuur 59 32-bit float readback 1024 x 1024

Internal Format	RED	LUM_ALPHA	RGB	BGR	RGBA	BGRA
GL_LUMINANCE8	972.76	-	-	-	-	-
GL_LUMINANCE8_ALPHA8	466.20	932.40	-	-	-	-
GL_RGB8	313.09	548.55	750.19	761.42	-	-
GL_RGBA8	279.88	463.39	447.29	685.40	737.46	883.00

Tabel 12 8-bit unsigned byte download 1024 x 1024

Internal Format	RED	LUM_ALPHA	RGB	BGR	RGBA	BGRA
GL_LUMINANCE16	932.40	-	-	-	-	-
GL_LUMINANCE16_ALPHA16	533.76	915.96	-	-	-	-
GL_RGB16	547.35	822.20	932.55	935.31	-	-
GL_RGBA16	501.13	743.91	814.77	810.37	912.41	911.27

Tabel 13 16-bit unsigned byte download 1024 x 1024

Internal Format	RED	LUM_ALPHA	RGB	BGR	RGBA	BGRA
GL_ALPHA16F_ARB	463.18	-	-	-	-	-
GL_LUMINANCE_ALPHA16F_ARB	465.98	881.64	-	-	-	-
GL_RGB16F_ARB	50.89	90.35	121.68	123.02	-	-
GL_RGBA16F_ARB	265.22	487.03	692.20	691.56	871.36	822.88

Tabel 14 16-bit float ARB download 1024 x 1024

Internal Format	RED	LUM_ALPHA	RGB	BGR	RGBA	BGRA
GL_ALPHA32F_ARB	1314.49	-	-	-	-	-
GL_LUMINANCE_ALPHA32F_ARB	280.50	523.63	-	-	-	-
GL_RGB32F_ARB	207.99	372.35	504.63	516.35	-	-
GL_RGBA32F_ARB	276.55	516.93	720.50	721.54	913.40	895.56

Tabel 15 32-bit float ARB download 1024 x 1024

Internal Format	RED	LUM_ALPHA	RGB	BGR	RGBA	BGRA
GL_FLOAT_R16_NV	404.53	-	-	-	-	-
GL_FLOAT_RG16_NV	89.47	146.42	-	-	-	-
GL_FLOAT_RGB16_NV	50.92	90.33	121.50	122.96	-	-
GL_FLOAT_RGBA16_NV	268.78	490.32	693.64	699.55	880.77	830.05

Tabel 16 16-bit float NV download 1024 x 1024

Internal Format	RED	LUM_ALPHA	RGB	BGR	RGBA	BGRA
GL_FLOAT_R32_NV	925.28	-	-	-	-	-
GL_FLOAT_RG32_NV	216.09	385.60	-	-	-	-
GL_FLOAT_RGB32_NV	213.07	381.26	515.02	526.64	-	-
GL_FLOAT_RGBA32_NV	277.49	518.17	722.02	722.89	915.59	897.52

Tabel 17 32-bit float NV download 1024 x 1024

Internal Format	RED	LUM_ALPHA	RGB	BGR	RGBA	BGRA
GL_LUMINANCE8	909.92	-	-	-	-	-
GL_LUMINANCE8_ALPHA8	511.77	899.28	-	-	-	-
GL_RGB8	334.22	10.04	642.95	781.66	-	-
GL_RGBA8	333.78	9.95	633.98	729.04	826.28	911.99

Tabel 18 8-bit unsigned byte readback 1024 x 1024

Internal Format	RED	LUM_ALPHA	RGB	BGR	RGBA	BGRA
GL_LUMINANCE16	894.45	-	-	-	-	-
GL_LUMINANCE16_ALPHA16	656.38	963.39	-	-	-	-
GL_RGB16	461.57	20.06	921.66	921.80	-	-
GL_RGBA16	461.04	19.88	924.64	924.07	935.45	935.34

Tabel 19 16-bit unsigned byte readback 1024 x 1024

Internal Format	RED	LUM_ALPHA	RGB	BGR	RGBA	BGRA
GL_ALPHA16F_ARB	672.49	-	-	-	-	-
GL_LUMINANCE_ALPHA16F_ARB	677.05	908.27	-	-	-	-
GL_RGB16F_ARB	58.17	97.64	115.18	114.71	-	-
GL_RGBA16F_ARB	449.44	139.38	774.09	772.50	881.93	874.89

Tabel 20 16-bit float ARB readback 1024 x 1024

Internal Format	RED	LUM_ALPHA	RGB	BGR	RGBA	BGRA
GL_ALPHA32F_ARB	1277.55	-	-	-	-	-
GL_LUMINANCE_ALPHA32F_ARB	489.42	681.61	-	-	-	-
GL_RGB32F_ARB	361.21	575.13	731.48	724.51	-	-
GL_RGBA32F_ARB	476.13	682.54	807.32	811.30	905.95	866.46

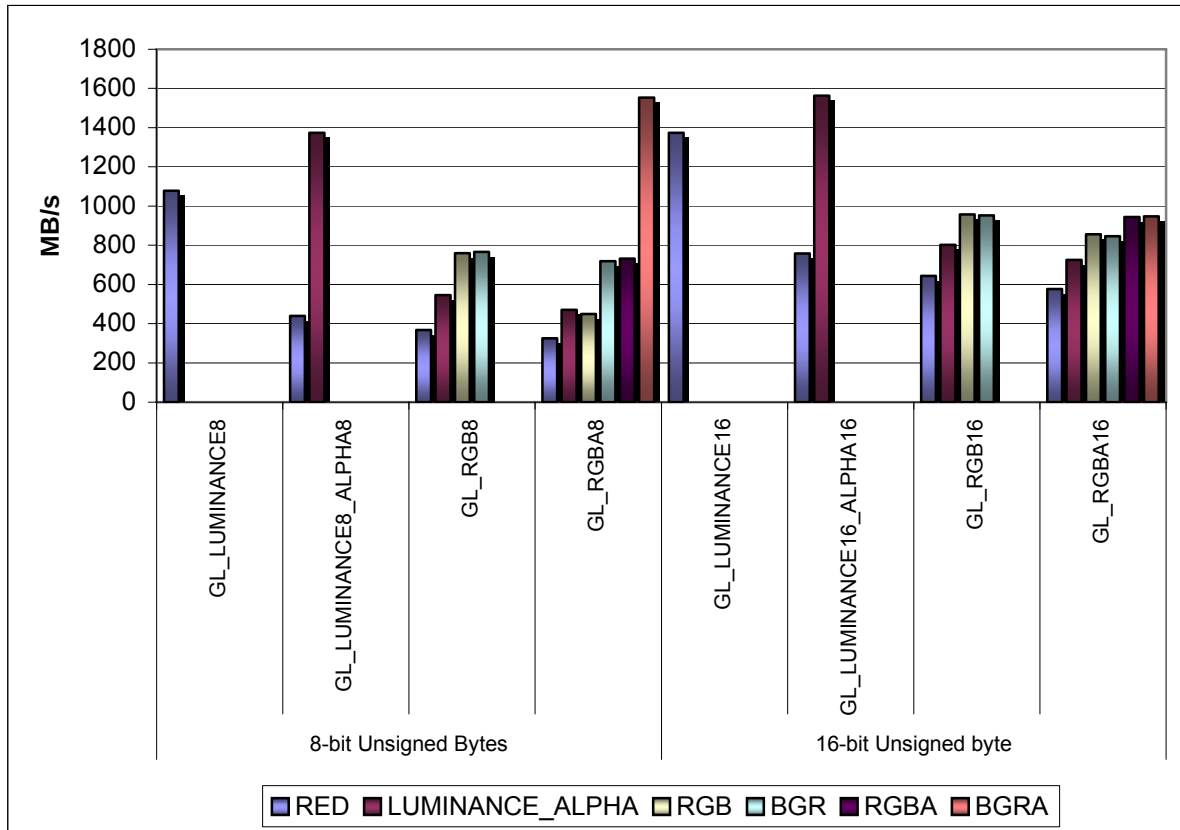
Tabel 21 32-bit float ARB readback 1024 x 1024

Internal Format	RED	LUM_ALPHA	RGB	BGR	RGBA	BGRA
GL_FLOAT_R16_NV	161.07	-	-	-	-	-
GL_FLOAT_RG16_NV	102.15	153.47	-	-	-	-
GL_FLOAT_RGB16_NV	58.10	97.64	115.04	114.52	-	-
GL_FLOAT_RGBA16_NV	451.88	139.32	776.20	776.00	865.05	562.79

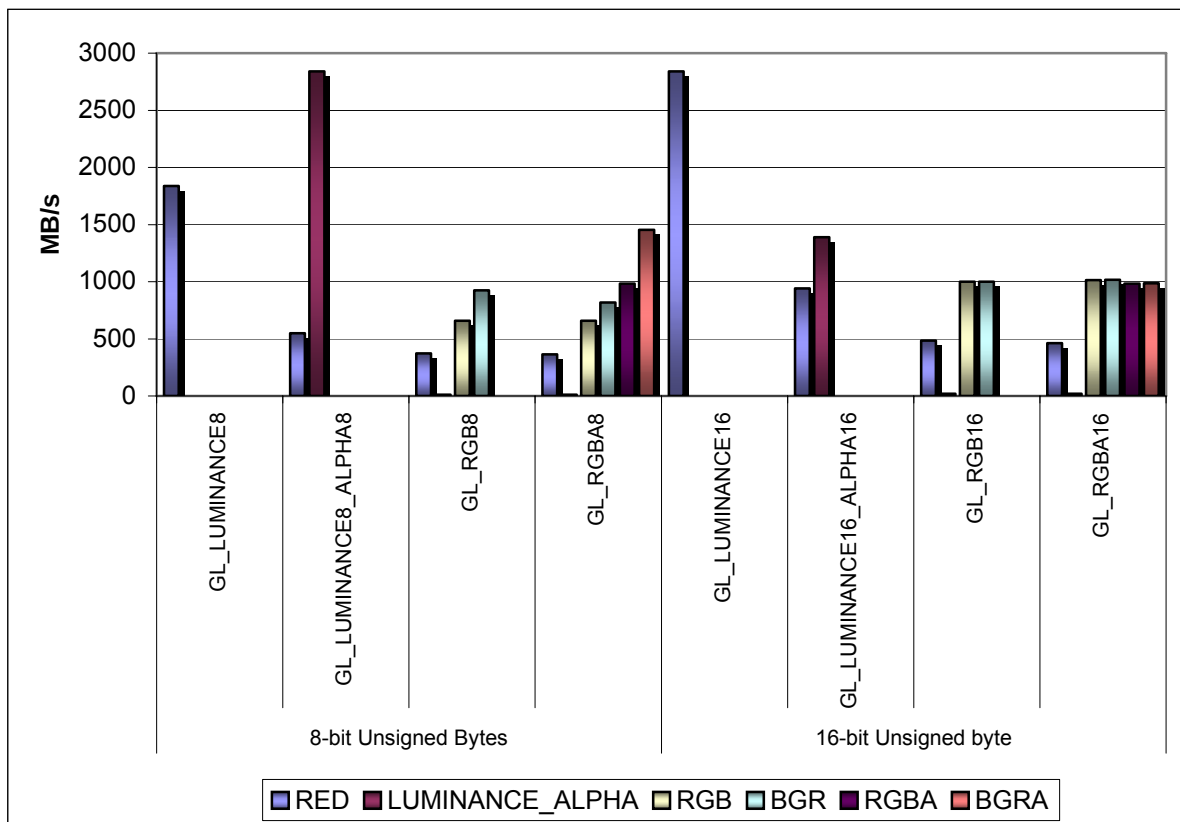
Tabel 22 16-bit float NV readback 1024 x 1024

Internal Format	RED	LUM_ALPHA	RGB	BGR	RGBA	BGRA
GL_FLOAT_R32_NV	918.27	-	-	-	-	-
GL_FLOAT_RG32_NV	361.96	589.45	-	-	-	-
GL_FLOAT_RGB32_NV	360.04	571.80	728.33	721.63	-	-
GL_FLOAT_RGBA32_NV	475.00	666.11	804.18	804.67	903.65	861.33

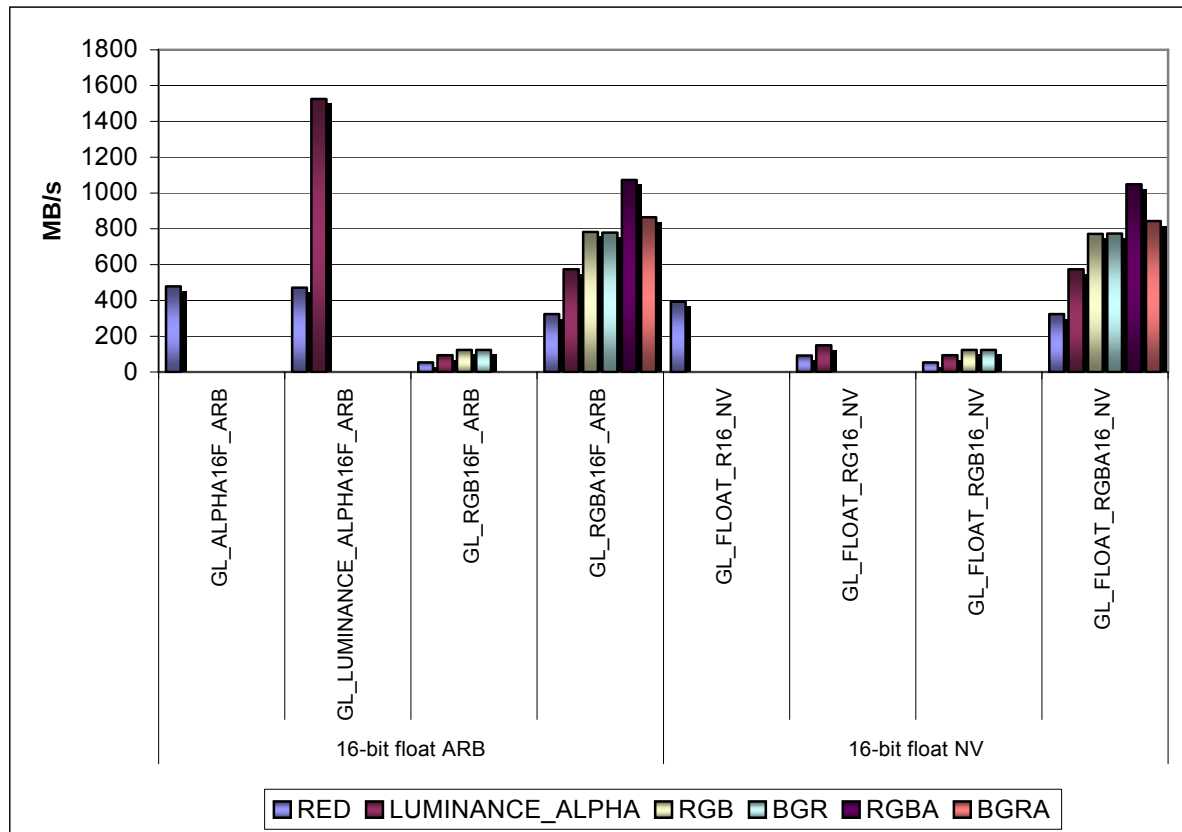
Tabel 23 32-bit float NV readback 1024 x 1024



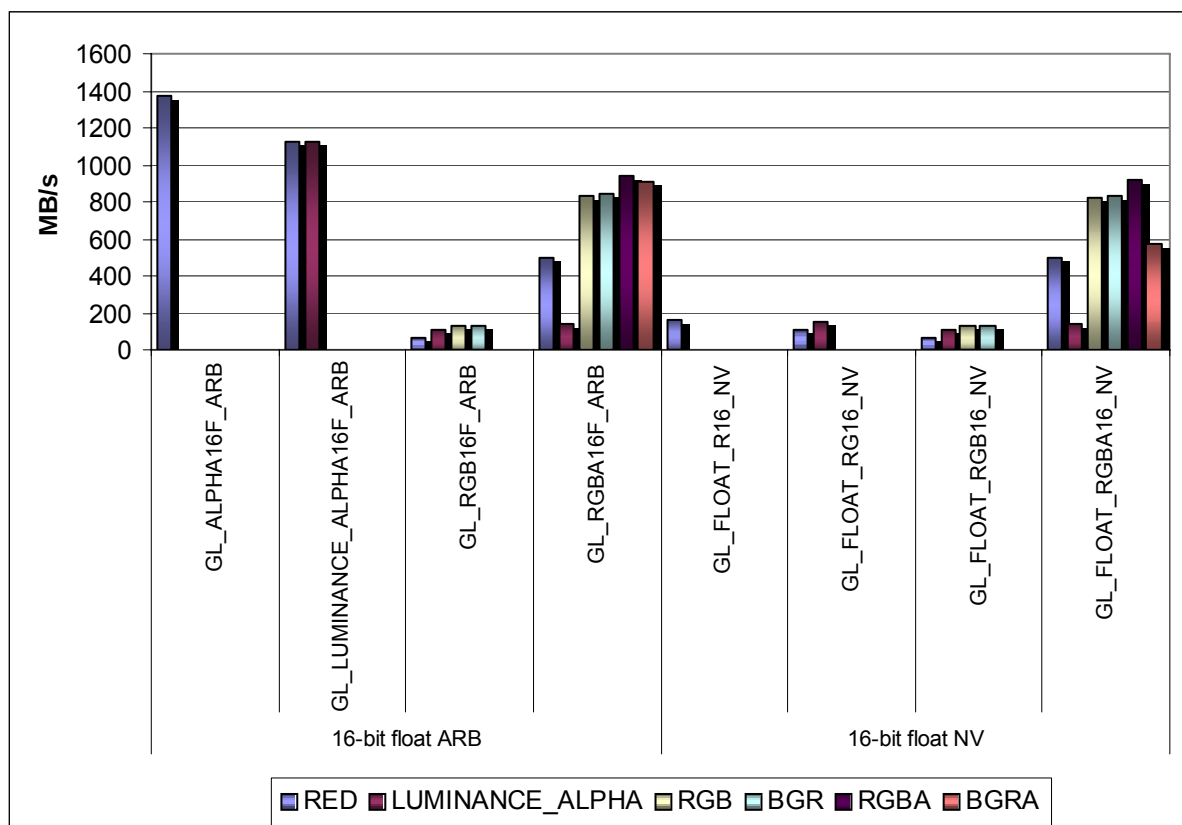
Figuur 60 Unsigned bytes download 256 x 256



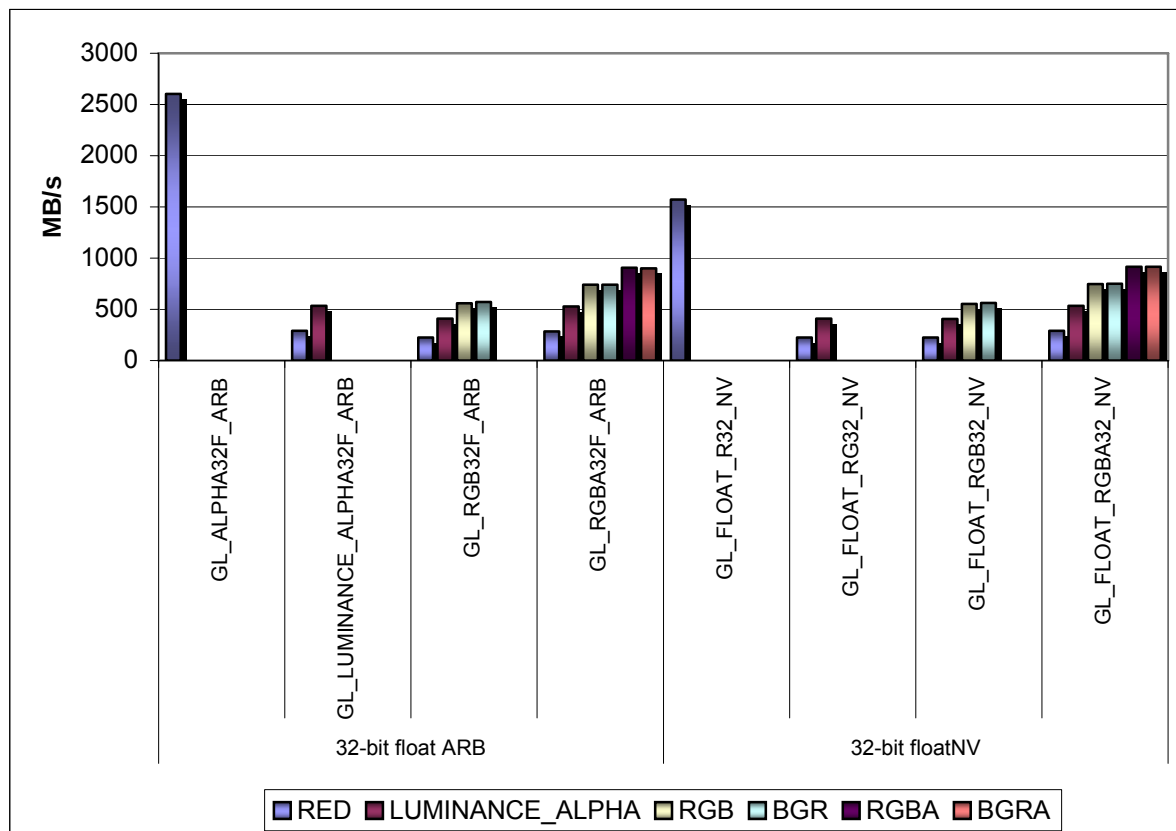
Figuur 61 Unsigned bytes readback 256 x 256



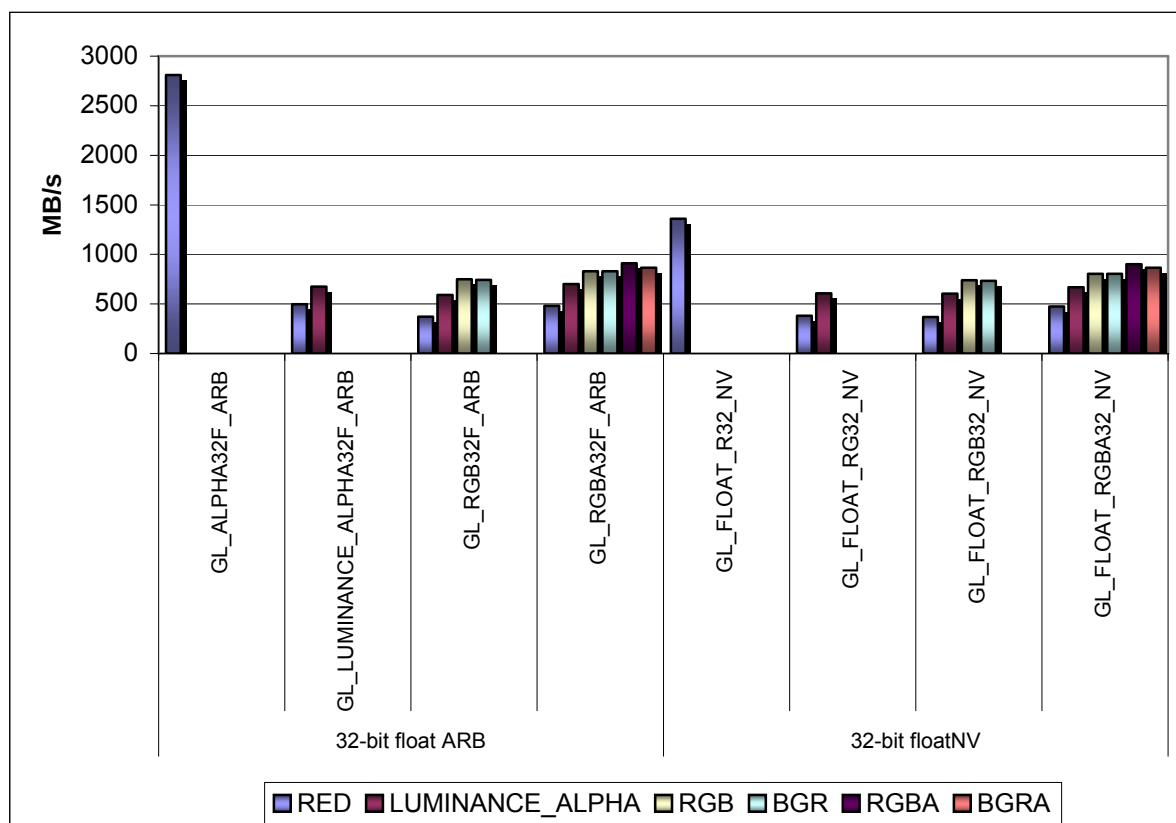
Figuur 62 16-bit float download 256 x 256



Figuur 63 16-bit float readback 256 x 256



Figuur 64 32-bit float download 256 x 256



Figuur 65 32-bit float readback 256 x 256

Internal Format	RED	LUM_ALPHA	RGB	BGR	RGBA	BGRA
GL_LUMINANCE8	1077.59	-	-	-	-	-
GL_LUMINANCE8_ALPHA8	440.14	1373.63	-	-	-	-
GL_RGB8	367.65	545.85	759.11	765.31	-	-
GL_RGBA8	325.52	469.92	449.64	718.39	730.99	1552.80

Tabel 24 8-bit unsigned byte download 256 x 256

Internal Format	RED	LUM_ALPHA	RGB	BGR	RGBA	BGRA
GL_LUMINANCE16	1373.63	-	-	-	-	-
GL_LUMINANCE16_ALPHA16	757.58	1562.50	-	-	-	-
GL_RGB16	644.33	801.28	956.63	951.78	-	-
GL_RGBA16	576.04	724.64	856.16	846.50	943.40	946.97

Tabel 25 16-bit unsigned byte download 256 x 256

Internal Format	RED	LUM_ALPHA	RGB	BGR	RGBA	BGRA
GL_ALPHA16F_ARB	478.93	-	-	-	-	-
GL_LUMINANCE_ALPHA16F_ARB	471.70	1524.39	-	-	-	-
GL_RGB16F_ARB	54.73	94.30	123.19	123.48	-	-
GL_RGBA16F_ARB	323.83	573.39	782.88	779.63	1072.96	865.05

Tabel 26 16-bit float ARB download 256 x 256

Internal Format	RED	LUM_ALPHA	RGB	BGR	RGBA	BGRA
GL_ALPHA32F_ARB	2604.17	-	-	-	-	-
GL_LUMINANCE_ALPHA32F_ARB	289.69	533.62	-	-	-	-
GL_RGB32F_ARB	225.02	408.50	559.28	570.34	-	-
GL_RGBA32F_ARB	284.41	526.87	739.64	740.38	905.80	900.09

Tabel 27 32-bit float ARB download 256 x 256

Internal Format	RED	LUM_ALPHA	RGB	BGR	RGBA	BGRA
GL_FLOAT_R16_NV	393.08	-	-	-	-	-
GL_FLOAT_RG16_NV	92.66	148.81	-	-	-	-
GL_FLOAT_RGB16_NV	54.75	94.30	123.23	123.44	-	-
GL_FLOAT_RGBA16_NV	323.83	573.39	771.60	773.20	1048.22	843.17

Tabel 28 16-bit float NV download 256 x 256

Internal Format	RED	LUM_ALPHA	RGB	BGR	RGBA	BGRA
GL_FLOAT_R32_NV	1572.33	-	-	-	-	-
GL_FLOAT_RG32_NV	225.84	408.50	-	-	-	-
GL_FLOAT_RGB32_NV	224.22	404.86	553.91	563.49	-	-
GL_FLOAT_RGBA32_NV	290.36	533.62	747.76	748.50	915.75	914.91

Tabel 29 32-bit float NV download 256 x 256

Internal Format	RED	LUM_ALPHA	RGB	BGR	RGBA	BGRA
GL_LUMINANCE8	1838.24	-	-	-	-	-
GL_LUMINANCE8_ALPHA8	548.25	2840.91	-	-	-	-
GL_RGB8	372.02	9.95	657.89	923.65	-	-
GL_RGBA8	363.37	10.05	657.89	818.78	984.25	1453.49

Tabel 30 8-bit unsigned byte readback 256 x 256

Internal Format	RED	LUM_ALPHA	RGB	BGR	RGBA	BGRA
GL_LUMINANCE16	2840.91	-	-	-	-	-
GL_LUMINANCE16_ALPHA16	939.85	1388.89	-	-	-	-
GL_RGB16	484.50	19.87	1000.00	1000.00	-	-
GL_RGBA16	461.25	20.06	1013.51	1016.26	982.32	986.19

Tabel 31 16-bit unsigned byte readback 256 x 256

Internal Format	RED	LUM_ALPHA	RGB	BGR	RGBA	BGRA
GL_ALPHA16F_ARB	1373.63	-	-	-	-	-
GL_LUMINANCE_ALPHA16F_ARB	1126.13	1121.08	-	-	-	-
GL_RGB16F_ARB	65.69	109.46	125.08	124.34	-	-
GL_RGBA16F_ARB	492.13	144.68	835.19	840.81	945.18	909.09

Tabel 32 16-bit float ARB readback 256 x 256

Internal Format	RED	LUM_ALPHA	RGB	BGR	RGBA	BGRA
GL_ALPHA32F_ARB	2808.99	-	-	-	-	-
GL_LUMINANCE_ALPHA32F_ARB	498.01	675.68	-	-	-	-
GL_RGB32F_ARB	370.92	590.32	747.76	742.57	-	-
GL_RGBA32F_ARB	482.63	702.25	828.73	828.73	909.09	866.55

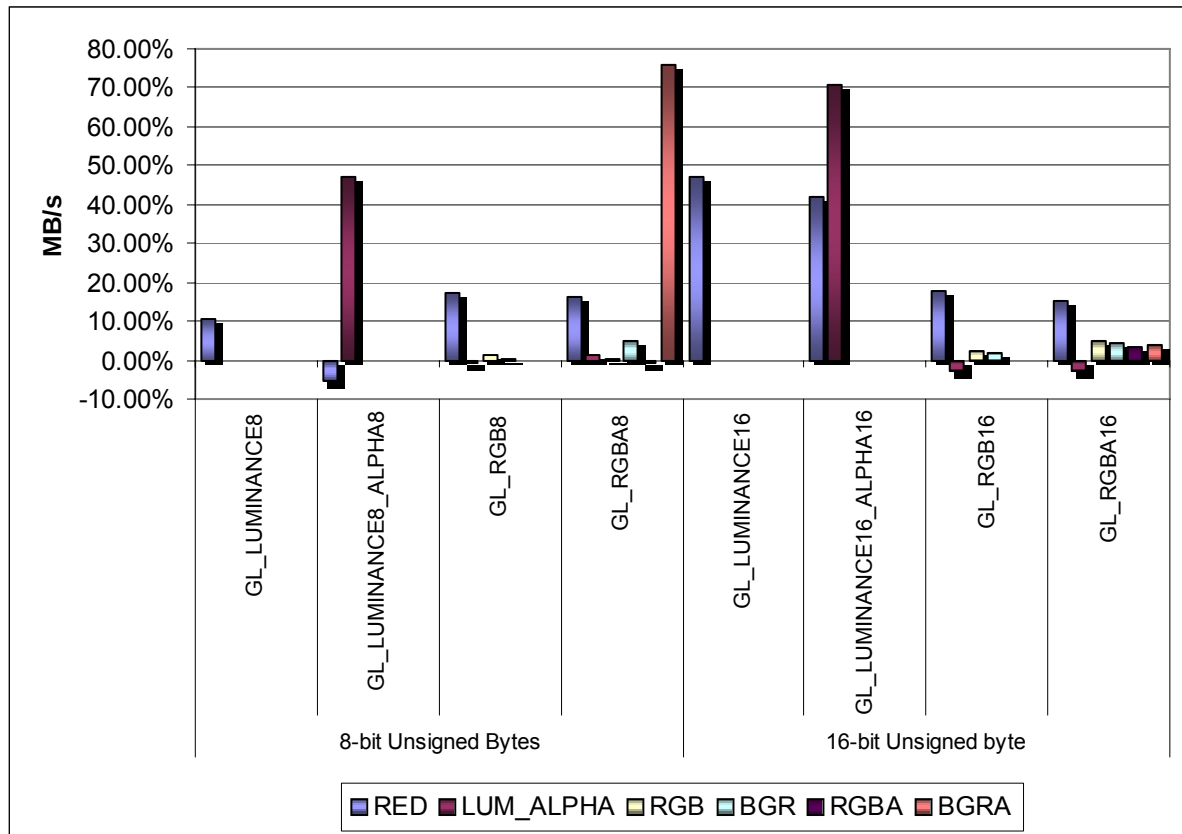
Tabel 33 32-bit float ARB readback 256 x 256

Internal Format	RED	LUM_ALPHA	RGB	BGR	RGBA	BGRA
GL_FLOAT_R16_NV	165.13	-	-	-	-	-
GL_FLOAT_RG16_NV	103.91	156.45	-	-	-	-
GL_FLOAT_RGB16_NV	65.82	109.27	124.88	124.63	-	-
GL_FLOAT_RGBA16_NV	496.03	144.76	820.57	827.81	922.51	570.13

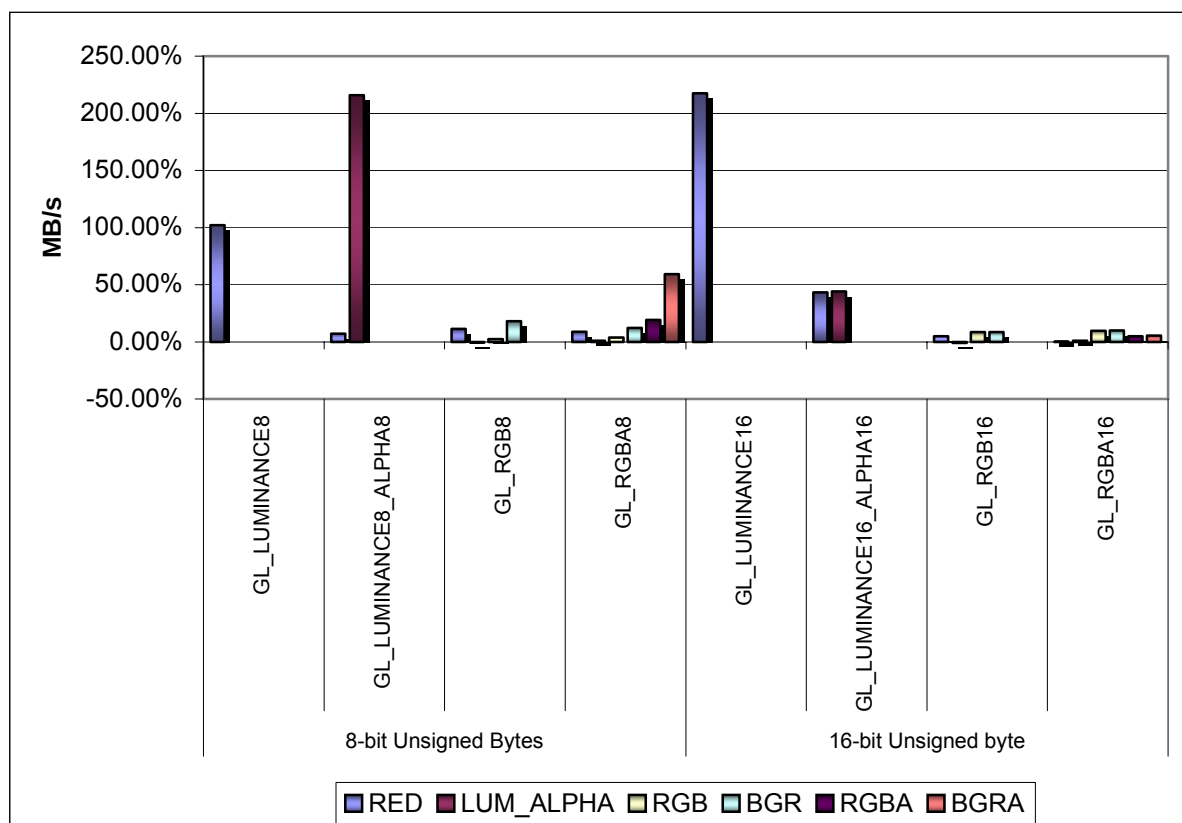
Tabel 34 16-bit float NV readback 256 x 256

Internal Format	RED	LUM_ALPHA	RGB	BGR	RGBA	BGRA
GL_FLOAT_R32_NV	1358.70	-	-	-	-	-
GL_FLOAT_RG32_NV	380.52	608.27	-	-	-	-
GL_FLOAT_RGB32_NV	367.65	603.86	739.64	733.86	-	-
GL_FLOAT_RGBA32_NV	476.19	667.56	803.86	803.00	902.53	865.05

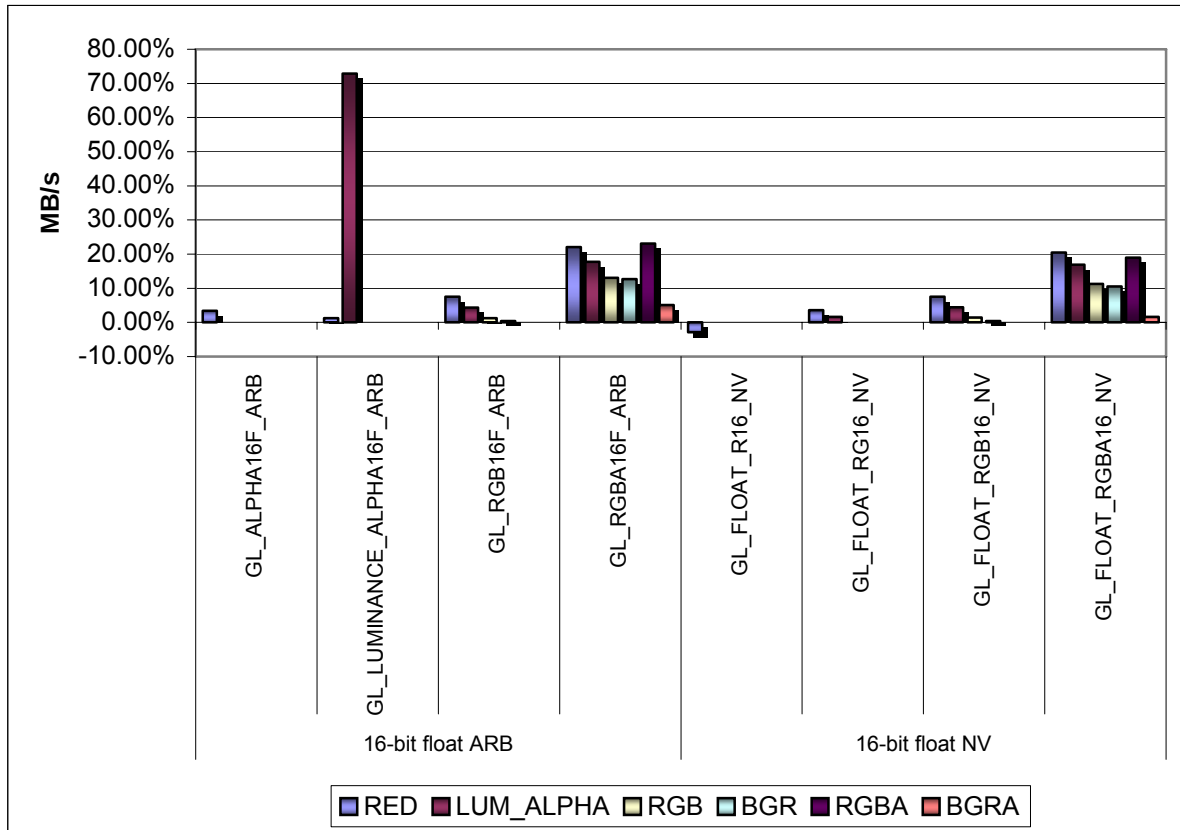
Tabel 35 32-bit float NV readback 256 x 256



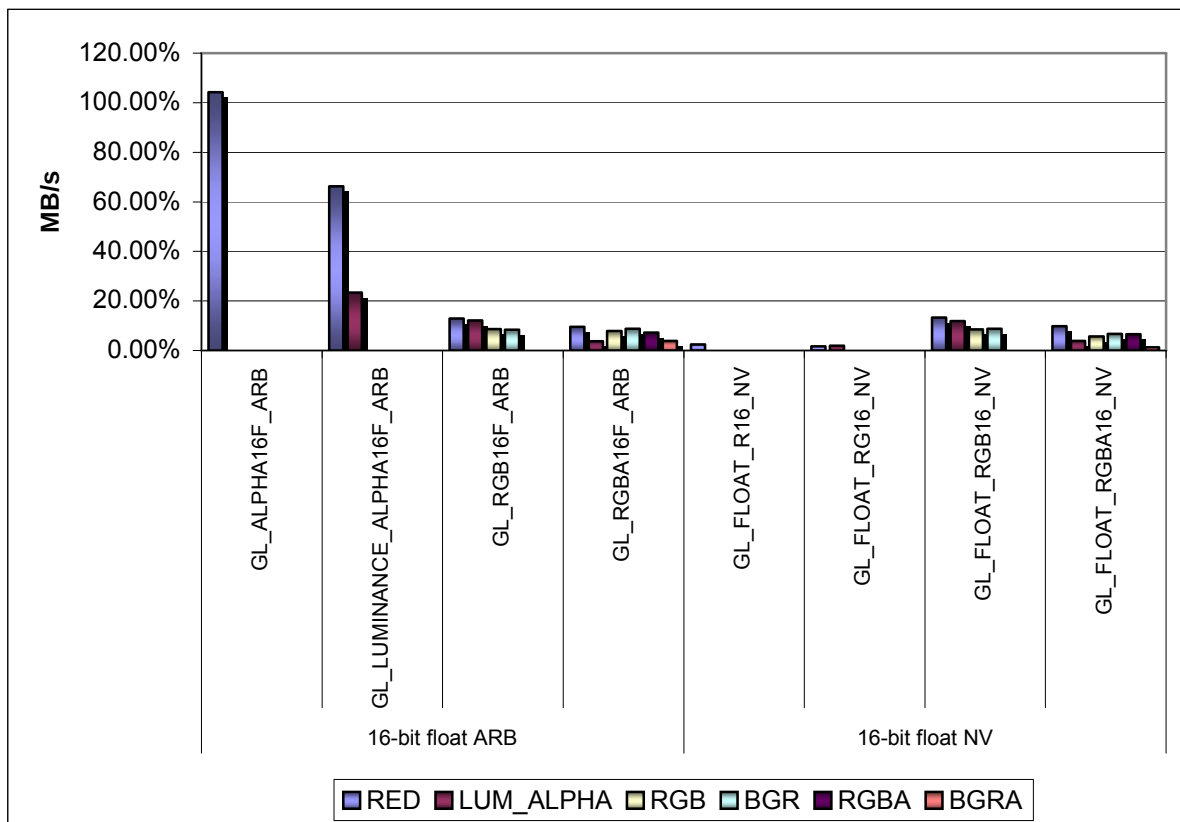
Figuur 66 Unsigned bytes download verschil tussen dimensies 256 en 1024



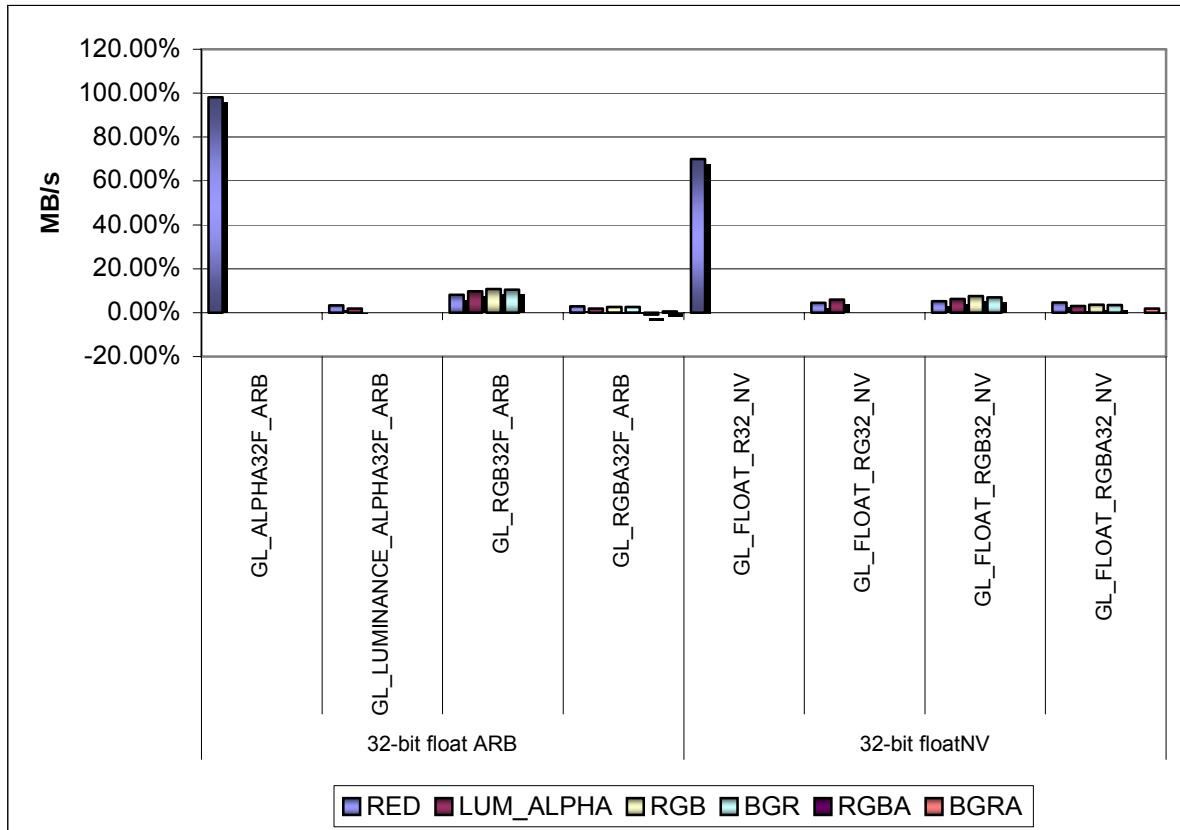
Figuur 67 Unsigned bytes readback verschil tussen dimensies 256 en 1024



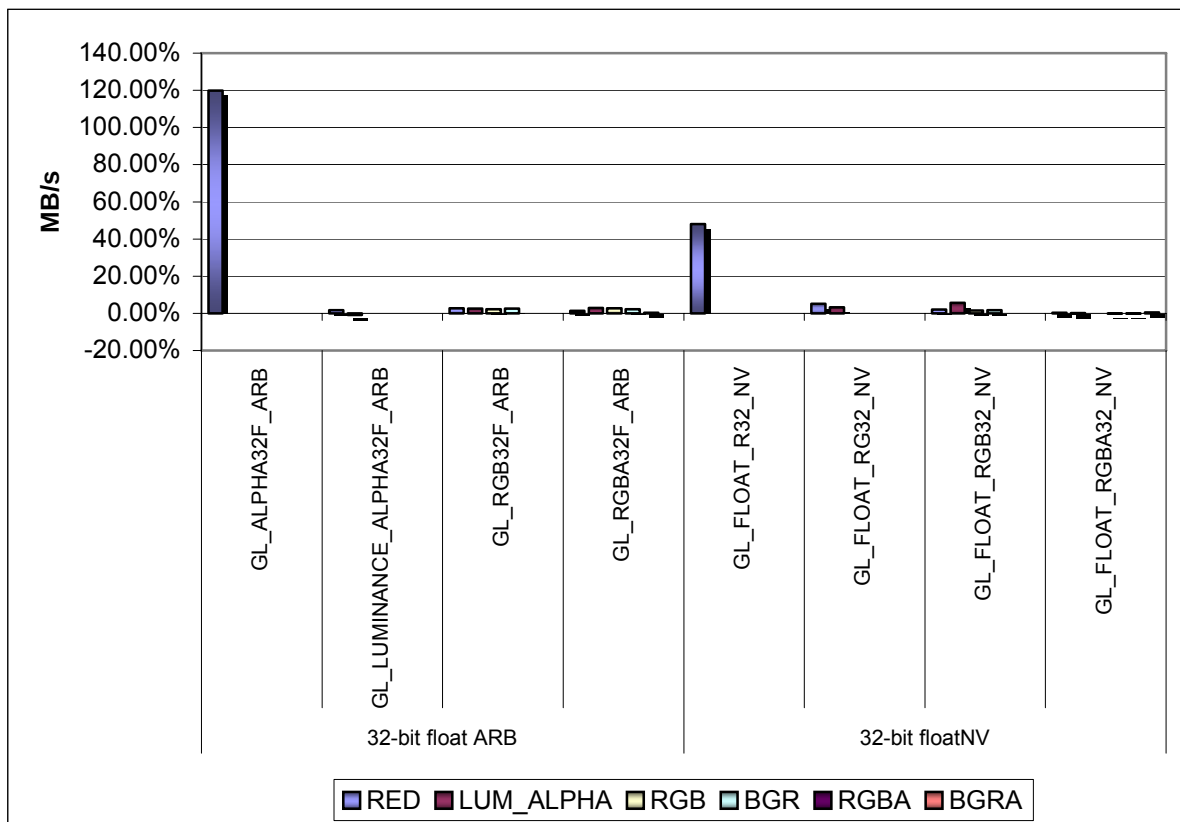
Figuur 68 16-bit float download verschil tussen dimensies 256 en 1024



Figuur 69 16-bit float readback verschil tussen dimensies 256 en 1024



Figuur 70 32-bit float download verschil tussen dimensies 256 en 1024



Figuur 71 32-bit float readback verschil tussen dimensies 256 en 1024

Internal Format	RED	LUM_ALPHA	RGB	BGR	RGBA	BGRA
GL_LUMINANCE8	10.78%	-	-	-	-	-
GL_LUMINANCE8_ALPHA8	-5.59%	47.32%	-	-	-	-
GL_RGB8	17.43%	-0.49%	1.19%	0.51%	-	-
GL_RGBA8	16.31%	1.41%	0.52%	4.81%	-0.88%	75.85%

Tabel 36 8-bit unsigned byte download verschil tussen dimensies 256 en 1024

Internal Format	RED	LUM_ALPHA	RGB	BGR	RGBA	BGRA
GL_LUMINANCE16	47.32%	-	-	-	-	-
GL_LUMINANCE16_ALPHA16	41.93%	70.59%	-	-	-	-
GL_RGB16	17.72%	-2.54%	2.58%	1.76%	-	-
GL_RGBA16	14.95%	-2.59%	5.08%	4.46%	3.40%	3.92%

Tabel 37 16-bit unsigned byte download verschil tussen dimensies 256 en 1024

Internal Format	RED	LUM_ALPHA	RGB	BGR	RGBA	BGRA
GL_ALPHA16F_ARB	3.40%	-	-	-	-	-
GL_LUMINANCE_ALPHA16F_ARB	1.23%	72.90%	-	-	-	-
GL_RGB16F_ARB	7.54%	4.37%	1.25%	0.37%	-	-
GL_RGBA16F_ARB	22.10%	17.73%	13.10%	12.73%	23.14%	5.13%

Tabel 38 16-bit float ARB download verschil tussen dimensies 256 en 1024

Internal Format	RED	LUM_ALPHA	RGB	BGR	RGBA	BGRA
GL_ALPHA32F_ARB	98.11%	-	-	-	-	-
GL_LUMINANCE_ALPHA32F_ARB	3.27%	1.91%	-	-	-	-
GL_RGB32F_ARB	8.19%	9.71%	10.83%	10.46%	-	-
GL_RGBA32F_ARB	2.84%	1.92%	2.66%	2.61%	-0.83%	0.51%

Tabel 39 32-bit float ARB download verschil tussen dimensies 256 en 1024

Internal Format	RED	LUM_ALPHA	RGB	BGR	RGBA	BGRA
GL_FLOAT_R16_NV	-2.83%	-	-	-	-	-
GL_FLOAT_RG16_NV	3.57%	1.63%	-	-	-	-
GL_FLOAT_RGB16_NV	7.54%	4.40%	1.43%	0.38%	-	-
GL_FLOAT_RGBA16_NV	20.48%	16.94%	11.24%	10.53%	19.01%	1.58%

Tabel 40 16-bit float NV download verschil tussen dimensies 256 en 1024

Internal Format	RED	LUM_ALPHA	RGB	BGR	RGBA	BGRA
GL_FLOAT_R32_NV	69.93%	-	-	-	-	-
GL_FLOAT_RG32_NV	4.51%	5.94%	-	-	-	-
GL_FLOAT_RGB32_NV	5.23%	6.19%	7.55%	7.00%	-	-
GL_FLOAT_RGBA32_NV	4.64%	2.98%	3.56%	3.54%	0.02%	1.94%

Tabel 41 32-bit float NV download verschil tussen dimensies 256 en 1024

Internal Format	RED	LUM_ALPHA	RGB	BGR	RGBA	BGRA
GL_LUMINANCE8	102.02%	-	-	-	-	-
GL_LUMINANCE8_ALPHA8	7.13%	215.91%	-	-	-	-
GL_RGB8	11.31%	-0.83%	2.32%	18.17%	-	-
GL_RGBA8	8.87%	0.97%	3.77%	12.31%	19.12%	59.38%

Tabel 42 8-bit unsigned byte readback verschil tussen dimensies 256 en 1024

Internal Format	RED	LUM_ALPHA	RGB	BGR	RGBA	BGRA
GL_LUMINANCE16	217.61%	-	-	-	-	-
GL_LUMINANCE16_ALPHA16	43.19%	44.17%	-	-	-	-
GL_RGB16	4.97%	-0.91%	8.50%	8.48%	-	-
GL_RGBA16	0.05%	0.88%	9.61%	9.98%	5.01%	5.44%

Tabel 43 16-bit unsigned byte readback verschil tussen dimensies 256 en 1024

Internal Format	RED	LUM_ALPHA	RGB	BGR	RGBA	BGRA
GL_ALPHA16F_ARB	104.26%	-	-	-	-	-
GL_LUMINANCE_ALPHA16F_ARB	66.33%	23.43%	-	-	-	-
GL_RGB16F_ARB	12.93%	12.11%	8.60%	8.39%	-	-
GL_RGBA16F_ARB	9.50%	3.80%	7.89%	8.84%	7.17%	3.91%

Tabel 44 16-bit float ARB readback verschil tussen dimensies 256 en 1024

Internal Format	RED	LUM_ALPHA	RGB	BGR	RGBA	BGRA
GL_ALPHA32F_ARB	119.87%	-	-	-	-	-
GL_LUMINANCE_ALPHA32F_ARB	1.76%	-0.87%	-	-	-	-
GL_RGB32F_ARB	2.69%	2.64%	2.22%	2.49%	-	-
GL_RGBA32F_ARB	1.36%	2.89%	2.65%	2.15%	0.35%	0.01%

Tabel 45 32-bit float ARB readback verschil tussen dimensies 256 en 1024

Internal Format	RED	LUM_ALPHA	RGB	BGR	RGBA	BGRA
GL_FLOAT_R16_NV	2.52%	-	-	-	-	-
GL_FLOAT_RG16_NV	1.72%	1.94%	-	-	-	-
GL_FLOAT_RGB16_NV	13.30%	11.91%	8.55%	8.82%	-	-
GL_FLOAT_RGBA16_NV	9.77%	3.90%	5.72%	6.68%	6.64%	1.30%

Tabel 46 16-bit float NV readback verschil tussen dimensies 256 en 1024

Internal Format	RED	LUM_ALPHA	RGB	BGR	RGBA	BGRA
GL_FLOAT_R32_NV	47.96%	-	-	-	-	-
GL_FLOAT_RG32_NV	5.13%	3.19%	-	-	-	-
GL_FLOAT_RGB32_NV	2.11%	5.61%	1.55%	1.69%	-	-
GL_FLOAT_RGBA32_NV	0.25%	0.22%	-0.04%	-0.21%	-0.12%	0.43%

Tabel 47 32-bit float NV readback verschil tussen dimensies 256 en 1024