# Thesis: GPU gebaseerde videocompressie

Tony Bleys

School voor Informatie Technologie
Transnationale Universiteit Limburg
Diepenbeek, 12 augustus 2005

**Abstract**

Met de opkomst van steeds snellere grafische kaarten, alsook de steeds toenemende vraag naar meer rekenkracht lijkt het nuttig om de rekenkracht van de grafische kaart ook te benutten voor andere berekeningen dan alleen maar grafische. In deze thesis gaan we proberen de grafische kaart aan te wenden voor het berekenen van motion estimation. Ook wordt er een overzicht gegeven van populaire motion estimation technieken.

Met dank aan professor Phillipe Bekaert voor zijn begeleiding, Tom Haber en Cederic Vanaken voor hun hulp en aan mijn ouders voor hun steun.

The best way to accelerate a Macintosh is at $9.8m/sec^2$.
-Marcus Dolengo

# Inhoudsopgave

# Lijst van figuren

# Hoofdstuk 1

# Inleidng

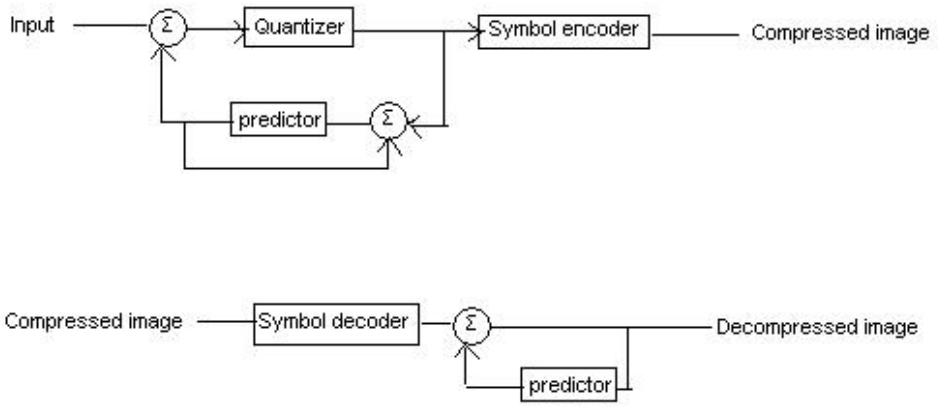
## 1.1 Wat is compressie?

Compressie in deze context slaat op data compressie. Compressie wordt gebruikt om de hoeveelheid data die nodig is om een zekere hoeveelheid informatie op te slaan, te reduceren. Er moet dus een onderscheidt gemaakt worden tussen informatie en de data die deze informatie representeerd. Dezelfde informatie kan door verschillende hoeveelheden data weergegeven worden.

Dit wordt treffend weergegeven door het verschil tussen de beschrijving van een vechtpartij door een journalist in een krantenartikel en de beschrijving van datzelfde gevecht door een van de deelnemers tegen zijn kamaraden na afloop. De journalist spendeert een klein stukje tekst, terwijl de vechtersbaas wel een hele avond kan vullen met het verhaal. Nochtans brengen beide dezelfde informatie over, namelijk de vechtpartij.

Voor het efficiënt weergeven van de informatie is het handig dat de data zo weinig mogelijk redundantie bevat. Er bestaan verschillende soorten redundantie. Zo kan er redundantie zitten in hoe de data gecodeerd is. Een manier om dit te verbeteren is om voor de meest voorkomende waardes de kortste code woorden te gebruiken. Dit is het idee achter Huffman encoding. Een ander soort redundantie hangt af van het type data. Zo kan men in een foto opmerken dat de kleur van een naburige pixels niet veel verschilt van elkaar. Deze soort redundantie wordt interpixel redundantie genoemd. Bij videobeelden komt ook iets dergelijks voor maar dan met overeenkomstige pixels tussen opeenvolgende frames.

Een derde manier om de hoeveelheid data te beperken is door gebruik te maken van de beperkingen van het menselijk waarnemingssysteem. Dit kan gebruikt worden om informatie die niet waarneembaar is door mensen weg te laten. Dit kan gaan van minime kleurverschillen die het menselijk oog niet in staat is om op te merken of hoge tonen die het menselijk oor niet kan waarnemen. Op deze manier kan men de hoeveelheid data die nodig is voor het overbrengen van de waarneembare informatie verkleinen. Een




Figuur 1.1: encoding & decoding schema

voorbeeld van deze compressie methode is de welbekende en populaire MP3 compressie voor muziek.

Merk op dat de laatste manier van compressie het onmogelijk maakt terug te keren tot het orgineel. Met andere woorden er gaat data verloren. Deze methode staat dan ook bekent als lossy compresie in tegenstelleing tot de andere 2 methoden die wel volledig omkeerbaar zijn en daarom lossless compresie genoemd worden.

Compressie methode n kan op elk soort data gebruikt worden, maar is niet op elk soort data even efficiënt. Ze is efficiënter naar mate bepaalde code tekens meer voorkomen dan anderen. Compressie methodes twee en drie zijn afhankelijk van de soort informatie. Bij compressie methode twee moet het mogelijk zijn om een schatting te maken van toekomstige data aan de hand van reeds bekende data. Nu probeert men de volgende data te voorspellen uit de reedsgekende data. Voor een schematische weergave sie figuur 1.1. Indien dit perfect mogelijk was, dan zou men een start waarde kunnen geven en de rest van de data volledig kunnen voorspellen. Nu is het natuurlijk niet mogelijk om inde meeste gevallen, de volgende data perfect te voorspellen. Daarom wordt er gekeken naar het verschil tussen de voorspelde waarde en de waarde uit de oorspronkelijke informatie. Deze verschillen zijn over het algemeen makkelijker om te encoderen en te comprimeren doordat ze redundantie verwijderen.

De derde compresie methode gaat niet echt opzoek naar redundantie, maar naar delen van de informatie die niet weergegeven of waargenomen kan worden. Dit stuk wordt dan verwijderd. Het grootste probleem bij dit soort technieken is vinden welke delen verwaarloosbaar zijn. Een computer kan moeilijk beoordelen welk van twee beelden het best lijkt op het oorspronkelijke. Er kan wel een schatting gedaan worden met verschillende

vergelijkingstechnieken maar de enige die kan oordelen is in principe een mens.

## 1.2 Videocompressie

Bij videocompressie wordt gebruik gemaakt van het feit dat er gewerkt wordt met informatie met een hoge graad van correlaties (tweede compressie methode) en de mogelijheden van het menselijk waarnemingssysteem (derde compressie methode.) De correlaties bestaan er tussen opeenvolgende frames. Het weergegeven beeld veranderd niet fel tussen twee frames in de meeste gevallen.

Om te beginnen heeft men een predctor nodig. In het specifieke geval van videocompressie wil men volgende frames afleiden uit het huidige frame. Dit probeerd men te doen door gebruik te maken van motion estimation. Dit dient dan als predictor. Deze voorspelling is meestal niet helemaal correct en dus moet het verschil beeld nog steeds geëncodeerd worden. Door een goede motion prediction te gebruiken kan men dit verschil zo uniform mogelijk maken, waardoor dit verschil met minder data is op te slaan dan het oorspronkelijke frame.

Er onstaan echter problemen wanneer er in een film een scene wiseling optreed. Dan is het onmogelijk om het volgende frame van het huidige af te leiden. Daarom wordt er bij videocompressie systemen een onderscheidt gemaakt tussen verschillende soorten frames. Men onderscheidt zogenaamde I-frames die zijn afgeleidt van de omliggende frames (meestal alleen het vorige) en de P-frames. P-frames zijn volledige frames die op zichzelf staan. Met andere woorden deze kunnen weergegeven worden met de data voor alleen hun frame en zijn niet afhankelijk van omliggende frames. Op deze manier kan men de data bij scene wisselingen beperken.

Nadat men een goede predictor gevonden heeft moet men ook nog het verschil tussen het voorspelde beeld en het werkelijke beeld opslaan. In het geval van een goede predictor is dit verschil redelijk neutraal. Dit verschil kan dan makkelijk verder gecomprimeerd worden. De meeste winst kan gehaald worden door rekening te houden met het onvermogen van het mesnelijk oog om kleine kleurschakeringen waar te nemen. De meeste gebruikte methoden zijn fouriertransformaties of discrete cosinus transformaties. De compresie wordt dan verwezelijkt door de hogere orde termen weg te laten. Dit zorgt natuurlijk voor een lichte degradatie van de beeldkwaliteit. Bij een te hoge compresie ratio treed er ruis op of komen er encoderingsartefacts voor. Dit is zichtbaar bijvoorbeeld bij een Xvid geëncodeerde film met een te lage bitrate.

## 1.3 Werking van GPU

Een ander belangrijk deel van deze thesis is niet slechts videocompressie, maar het uitvoeren van (delen van) deze compressie op de grafische processor ook wel GPU genoemd. De GPU is tegenwoordig een onmisbaar onderdeel van de hedendaagse computer.

Een GPU haalt tegenwoordig ongeveer 40 GFlop oftewel 40 miljard floting points operaties per seconde. Een moderne CPU daarentegen haalt slechts een 6 GFlop. Dit is een redelijk groot verschil in prestaties. De volgende generatie grafische kaarten zullen nog sneller zijn. Er is reeds een indicatie uitgaande van de specificaties van de Microsoft XBox 360 ™en de Sony Playstation3 ™, welke beiden een performance van 1 á 2 TFlop claimen.

De hoge performance van een GPU is een aanleiding om te proberen deze rekenkracht ook aan te wenden voor andere toepassingen dan grafische berekeningen. Dit is echter niet zo vanzelf sprekend. Een GPU is opgebouwd uit aparte zogenaamde pipelines. Moderne GPU's hebben tot 16 zulke pipelines voor de pixelshader operaties. Dat wil zeggen dat ze 16 verschillende berekeningen tegelijkertijd kunnen uitvoeren. Een CPU daarentegen kan meestal slechts 1 berekening tegelijkertijd, al zien we ook hier een opkomst van de parallellisering met Intels Hyperthreading systeem en de dualcore processoren van zowel Intel als AMD die op de markt aan het verschijnen zijn. Indien men de volledige rekenkracht van de GPU wil gebruiken moet men een algoritme gebruiken dat kan profiteren van het parallellisme van de GPU.

Een ander nadeel van de GPU is dat hij veel gespecialiseerder is. Hij is niet zo geschikt om algemene berekingen uit te voeren als de CPU. Met de opkomst van de programeerbare vertex- en pixel shaders is dit verminderd. Met behulp van shadermodel 3.0 kan men bijna alle bewerkingen ook op een GPU uitvoeren. Door de architectuur van een GPU blijft complexe branching een performance killende operatie.

Een laatste verschil met de hedendaagse CPU is de geheugenbandbreedte. Een GPU heeft tegenwoordig al snel 256MB of meer geheugen tot zijn beschikking. De bandbreedte bedraagd typisch iets van 25Gb/s. Dit is een stuk meer dan dat de CPU tot zijn beschikking heeft. Deze heeft typisch iets van 6Gb/s tot zijn beschikking voor het aanspreken van systeem geheugen. De CPU heeft echter ook on-die cache geheugen waar hij veel sneller toegang tot heeft. dit is wel een stuk kleiner, typisch iets van 512kB tot 1MB.

## 1.4 Verwezelijkingen

Alhoewel het in eerste instantie de bedoeling was om de volledige videocompressie te implementeren op de grafische kaart is dit niet gelukt. Ik ben er slechts in geslaagd om de eerste helft van de videocompressie te implementeren. Met mijn programma kan men momenteel motion prediction uitvoeren met behulp van blok matching algoritmen en kan men diferential beelden berekenen. Eerste testen met wavelet encoding voor de rest van een videocompressie programma duiden wel aan dat wavelet compressie te traag is om te gebruiken voor videocompressie.

# Hoofdstuk 2

# Motion Prediction technieken

Motion estimation of optical flow probeerd de verandering van opeenvolgende beelden te voorspellen. Dit is handig voor een veelvoud van toepassingen. Zo kan men het gebruiken voor het volgen en herkennen van objecten, wat onder andere bruikbaar is voor virtual reality en augmented reality. Ook kan het gebruikt worden voor het bepalen van time-of-collision. Er bestaan verschilledne technieken die hieronder kort besproken zullen worden.

## 2.1 Differentiële Technieken

De eerste basis van de differentële technieken is gelegd door eerste orde afgeleiden van een simpele translatie van het beeld dus:

$$I(x,t) = I(x - vt, 0) \tag{2.1}$$

waar I de intensiteit van het beeld is op plaats $x$ en tijdstip $t$. De vector $v$, in het tweede deel van de vergelijking, geeft de snelheid waarmee het beeld translateerd aan. Formule 2.1 zegt dus dat men het beeld op tijdstip t kan vinden door het beeld te nemen op tijdstip 0 en alle pixels op te schuiven in de richting van v en dat de lengte van deze verschuiving afhangt van de verlopen tijd.

Indien we aannemen dat de intensiteit niet veranderd doorheen de tijd dwz $dI(x,t)/dt = 0$ kunnen we een beperking uitrekenen voor de gradiënt. Deze restricite komt neer op:

$$\nabla I(x,t) \cdot v + I_t(x,t) \tag{2.2}$$

Het probleem is dat $v$ uit vergelijking 2.1 bestaat uit twee componenten. Er zijn dus in feite twee onbekenden en slechts 1 vergelijking. Bijgevolg zijn

er oneindig veel oplossingen Om tot een unieke oplossing te komen moet men dus een bijkomende restrectie opleggen.

Tweede orde differentiële methode maken gebruik van de Hessiaan (= de tweede alfgeleide van $I$) als bijkomende restrictie.

$$\begin{bmatrix} I_{xx}(x,t) & I_{yx}(x,t) \\ I_{xy}(x,t) & I_{yy}(x,t) \end{bmatrix} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} + \begin{pmatrix} I_{tx}(x,t) \\ I_{ty}(x,t) \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \tag{2.3}$$

Vergelijking , samen met vergelijking bepalen een overgedetermineerd systeem van lineare vergelijkingen. Het probleem hierbij is dat het in sommige gevallen moeilijk is om 2de orde afgeleiden nauwkurig genoeg te meten.

### 2.1.1 Horn en Schunk

Horn en Schunk gebruiken in [HS81] een globale smoothness restrictie om de optical flow te schatten. Zij keizen ervoor om de absolute gradiënt van de snelheid te minimalizeren:

$$\int [(\nabla I \cdot v + I_t)^2 + \lambda^2(\|\nabla u\|_2^2 + \|\nabla u\|_2^2)]dx \tag{2.4}$$

Vergelijkingen 2.1 en 2.4 kunnen uitgewerkt worden tot een paar van recursieve vergelijkingen:

$$u^{k+1} = \bar{u}^k - \frac{I_x(I_x\bar{u}^k + I_y\bar{v}^k + I_t)}{\alpha^2 + I_x^2 + I_y^2} \tag{2.5}$$

en

$$v^{k+1} = \bar{v}^k - \frac{I_y(I_x\bar{u}^k + I_y\bar{v}^k + I_t)}{\alpha^2 + I_x^2 + I_y^2} \tag{2.6}$$

waarbij $k$ staat voor het iteratie nummer, $\bar{u}^k$ en $\bar{v}^k$ staan voor een lokaal gemiddelde van respektivelijk $u^k$ en $v^k$ en waarbij de initiële waarden $u^0$ en $v^0$ op 0 gezet worden. Deze twee vergelijkingen kunnen dan op iterative wijze opgelost worden.

### 2.1.2 Lucas en Kanade

Lucas en Kanade gebruiken net als Horn en Schunk een smoothness restrictie, maar hun methode zoals beschreven in [LK81] gebruikt een presmoothing algoritme. Zoals:

$$\min \sum_{x\in\Omega} W^2(x)[\nabla I(x,t) \cdot v + I_t(x,t)]^2 \tag{2.7}$$

waarbij $W(x)$ een vesnterfunctie is die meer gewicht geeft in het centrum van de omgeving $\Omega$

Deze vergelijkingen kunnen herleid worden tot

$$\begin{aligned} v &= [A^T W^2 A]^{-1} A^T W b \\ A &= [\nabla I_t(x_1, \ldots, \nabla I_t(x_n]^T \\ W &= Diag[W(x_1, \ldots, W(x_n] \\ b &= -[I_t(x_1, \ldots, I_t(x_n]^T \end{aligned}$$

Eén voordeel ten opzichte van de vorige methode van Horn en Schunk is dat deze methode van Lucas en Kanade komt met een ingebouwde betrouwbaarheidsmetriek zijnde de kleinste eigenwaarde $\lambda_1$ uit

$$A^T W^2 A = \begin{bmatrix} \Sigma W^2 I_x^2 & \Sigma W^2 I_x I_y \\ \Sigma W^2 I_x I_y & \Sigma W^2 I_y^2 \end{bmatrix} \tag{2.8}$$

Deze zorgt voor een manier om een onderscheid te kunnen maken tussen de 2D snelheid en de normale snelheid.

Volgens Barron et al. in [BFBB92] is dit een van de betrouwbaarste algoritmen uit hun test.

### 2.1.3 Nagel

Een andere manier om een extra restrictie op te leggen is bedacht door Nagel. Nagel was één van de eerste die bedacht om de restrictie te laten afhangen van de oriëntatie. Dit was een poging om het oclusie probleem op te lossen. Het komt erop neer dat de smoothness niet afgedwongen wordt op plaatsen waar de gradiënt erg stijl is. Deze plaatsen zijn randen van een voorwerp. Het probleem is wiskundig te schrijven als het minimaliseren van:

$$\int\int (\nabla I^T v + I_t)^2 + \frac{\alpha^2}{\|\nabla I\|_2^2 + 2\delta}[(u_x I_y - u_y I_x)^2 + (v_x I_y - v_y I_x)^2 + \delta(u_x^2 + u_y^2 + v_x^2 + v_y^2)]dxdy \tag{2.9}$$

waarbij $\delta$ een vaste constante is. Nagel raad aan om deze te fixeren op 1.0. $\alpha$ wordt in de meeste gevallen in de testen van Barron et al. gefixeerd op 0.5.

Dit minimalizatie probleem kan worden opgelost door het gebruik van Gauss-Seidel iteraties, welke er als volgt uitzien:

$$u^{k+1} = \xi(u^k) - \frac{I_x(I_x\xi(u^k) + I_y\xi(v^k) + I_t)}{\alpha^2 + I_x^2 + I_y^2} \tag{2.10}$$

en

$$v^{k+1} = \xi(u^k) - \frac{I_y(I_x\xi(u^k) + I_y\xi(v^k) + I_t)}{\alpha^2 + I_x^2 + I_y^2} \tag{2.11}$$

hierbij is k het aantal ieteraties en $\xi(r)$ met $r$ $u$ of $v$ wordt gegeven door:

$$\xi(r) = \bar{r}^k - 2I_x I_y r_x y - q^T(\nabla r^k) \tag{2.12}$$

waarbij $q$ staat voor:

$$q = \frac{1}{I_x^2 + I_y^2 + 2\delta} \nabla I^T \left[ \begin{pmatrix} I_{yy} & -I_{xy} \\ -I_{yx} & I_{xx} \end{pmatrix} + 2 \begin{pmatrix} I_{xx} & I_{yx} \\ I_{xy} & I_{yy} \end{pmatrix} W \right] \tag{2.13}$$

met W de gewichtsmatrix:

$$W = \frac{1}{I_x^2 + I_y^2 + 2\delta} \begin{pmatrix} I_y^2 + \delta & -I_x I_y \\ -I_x I_y & I_x^2 + \delta \end{pmatrix} \tag{2.14}$$

## 2.2 Energie Gebaseerde Technieken

Een andere manier om optical flow te berekenen is door middel van bandpass-filters. Elk van deze bandpass-filters is afgesteld op een andere snelheid en oriëntatie. Dit soort methode worden ook wel eens frequentie gebaseerde technieken genoemd om wille van de constructie van snelheidsfilters in het frequentie domein. De fourier getransformeerde van een translateerend 2D beeld is:

$$\hat{I}(k, \omega) = \hat{I}(k, 0)\delta(\omega + v^T k) \tag{2.15}$$

Waarbij $\delta$ staat voor de dirac-delta, $k$ voor de ruimtelijke frequentie en $\omega$ voor de temporele frequentie. Uit vergelijking 2.15 volgt dat alle oplossingen liggen op een vlak dat door de oorsprong gaat.

### 2.2.1 Heeger

Heeger maakt gebruik van een least-square fit om de spatio-temporele energie te mappen op een vlak uit de frequentie-ruimte. Om de lokale energie uit het beeld te halen gebruikt Heeger zogenaamde Gabor filters. Hij gebruikt 12 van deze filters op elke schaal, elk afgestemd op een andere oriëntatie en verschillende temporele frequënties. In het ideale geval en met een enkele translatie beweging, zijn de responties van deze filters geconcentreerd in een vlak in de frequëntie-ruimte.

Heeger gebruikt dan de verwachte responties van het translateren van witte ruis, zijnde:

$$R(u, v) = \exp \frac{-4\pi^2 \sigma_x^2 \sigma_y^2 \sigma_z^2 \sigma_t^2 (u k_x + v k_y + \omega)}{(u\sigma_x\sigma_t)^2 + (v\sigma_y\sigma_t)^2 + (\sigma_x\sigma_y)^2} \tag{2.16}$$

waarbij de respectieve $\sigma$'s staan voor de standaard deviaties van de Gaussiaanse component van de Gabor filters.

Laat $M_i, 1 \leq i \leq 12$ de verzameling zijn van filters met dezelfde oriëntatie. $\bar{m}_i$ is dan de som van de gemeten energie van deze verzameling filters, terwijl $\bar{R}_i$ de som van de voorspelde waarden van deze verzameling filters is. Dus:

$$\bar{m}_i = \sum_{j \in M_i} m_j \tag{2.17}$$

en

$$\bar{R}_i = \sum_{j \in M_i} R_j(u, v) \tag{2.18}$$

De least square fit voor $(u, v)$ minimalizeerd dan deze functie:

$$f(u, v) = \sum_{i=1}^{12} [m_i - \bar{m}_i \frac{R_i(u, v)}{\bar{R}_i(u, v)}]^2 \tag{2.19}$$

Op deze manier wordt het verschil tussen de verwachte en de gemeten waarden het kleinst.

## 2.3 Fase Gebaseerde Technieken

Fase gebaseerde technieken lijken redelijk veel op de voorgaande methode voor het bepalen van de optical flow, maar in dit geval wordt er gebruik gemaakt van het fase gedrag om de snelheid te bepalen.

### 2.3.1 Fleet en Jepson

Fleet en Jepson maken ook gebruik van een familie van op snelheid afgestelde filters. Elk van deze filters is complex en kan geschreven worden als volgt:

$$R(x, t) = \rho(x, t) \exp^{[i\phi(x,t)]} \tag{2.20}$$

waarbij $\rho(x, t)$ en $\phi(x, t)$ staan voor respectievelijk de amplitude en de fase van $R$. De normale component van de snelheid wordt gegeven door $v_n = sn$ waarbij

$$s = \frac{-\phi_t(x, t)}{\parallel \nabla\phi(x, t) \parallel} \tag{2.21}$$

en

$$n = \frac{\nabla\phi(x, t)}{\parallel \nabla\phi(x, t) \parallel} \tag{2.22}$$

In feite is dit een differentiële techneik toegepast op de fase in plaats van op de intensiteit.

Men kan dit doen omdat de fase contouren een goede benadering vormen van het geprojecteerde bewegingsveld. Het gebruik van de fase in plaats van de amplitude zoals in bovenstaande methode heeft als voordeel dat de fase in het algemeen stabiler is dan de amplitude en dus de voorkeur verdient.

## 2.4 Regio Gebaseerde Matching

Regio Gebaseerde matching hangt niet af van de gradient zoals de meeste andere technieken. Het is niet altijd mogelijk om accuraat numerieke differentiatie te doen zoals nodig is voor de bovenstaande technieken. Dat kan verschilledne redenen hebben. Zo kan het zijn dat men slechts een klein aantal frames heeft om mee te werken of de frames die gebruikt kunnen worden bevatten veel ruis of iets dergelijks. In zo een geval kan men beter regio gebaseerde matchting gebruiken. De snelheidheid is dan een translatie $t = (t_x, t_y)$. De beste match wordt dan gevonden door het minimalizeren van een metriek of een gelijk soortige voorwaarde. Een veelgebruikte metriek is de som van de kwadratische verschillen.

$$SKV_{1,2}(x,d) = \sum_{j=-n}^{n} \sum_{i=-n}^{n} W(i,j)[I_1(x+(i,j)) - I_2(x+d+(i,j))]^2 \quad (2.23)$$

waarbij $W(i,j)$ een vensterfunctie is.

### 2.4.1 Anandan

Anadan gebruikt in [Ana89] een algoritme dat begint met een heel ruwe benadering van het beeld en steeds verder gaat verfijnen. Op deze manier kan hij eerst de grove bewegingen eruit plukken en zo steeds kleinere verplaatsingen vinden bij opeen volgende stappen.

Anadan maakt gebruikt van de zogeheten Laplaciaanse piramides. Met behulp van deze Laplaciaanse piramides kan hij belangrijke beeldelementen zoals randen beter gebruiken. Ook helpen ze bij het bepalen van de grove bewegingen. Voor de vensterfunctie W gebruikt hij een Gaussiaan van 5 pixels bij 5 pixels.

Op het grofste niveau neemt hij aan dat pixels zich niet sneller verplaatsen dan 1 pixel per frame. Er wordt gezocht naar minima van de SKV door te kijken naar vlakjes van 3 bij 3 pixels. Met Beaudet operatoren kan hok dan de parameters bepalen van het SKV oppervlak. Ook gebruikt hij een vlakheidsrestrictie op de snelheden. Na het uitreken hiervan leidt dit tot de volgende vergelijking, die volgens Anadan genoeg convergeerd na 10 iteraties:

$$v[k+1 = \bar{v}^k + \frac{c_{mmax}}{c_{max}+1}[(v_0 - \bar{v}^k)\dot{e}_{max}]e_{max} + \frac{c_{min}}{c_{min}+1}[(v_0 - \bar{v}^k)\dot{e}_{min}]e_{min} \quad (2.24)$$

Voor de volgende, fijnere stap worden de verkregen resultaten geprojecteerd op een 4 bij 4 gebied. Op deze wijze krijgt elke pixel op het fijnere niveau 4 initiële beginwaarden. Als zoekgebied voor de SKV gebruikt men dan de unie van de 3 bij 3 pixel gebiedjes uit de initiële waarden.

### 2.4.2 Singh

Singh gebruikt een 2 staps benadering. In de eerste stap worden de SKV waarden berekend tussen drie opeenvolgende frames. Om de motion estimation van een geven frame te zoeken word het voorgaande en het volgende frame gebruikt.

$$SKV_0(x,d) = SKV_{0,1}(x,d) + SKV_{0,-1}(x,-d) \tag{2.25}$$

met SKV zoals in vergelijkin 2.23 Door zoals in vergelijking 2.25 te zien is, de som te gebruiken van de SKV van twee opeenvolgende frames wordt de invloed van ruis en alisasing door een herhalend texture geminimaliseerd.

Singh zet dan $SKV_0$ om in een kansverdeling, $R_e = e^{-kSKA_0}$ met $k = \frac{-\ln(0.95)}{\min(SKA_0)}$. De subpixel snelheid is dan het gemiddelde van deze verdeling. Dit werkt alleen goed als de kansverdeling nagenoeg symmetrisch is over de verplaatsing. Singh raad daarom ook het gebruik van de Laplaciaanse piramides en een grof naar fijn benadering aan. Dit heeft als bijkomend voordeel dat ook grotere verplaatsingen gededecteerd kunnen worden.

## 2.5 Block Matching

De voorgaande technieken leveren allemaal wel redelijke tot goede resultaten op voor motion estimation, afhankelijk van het soort beweging, maar deze technieken zijn vooral ontworpen met motion estimation als doel, en niet als middel. Voor video-compressie moet de motion estimation niet alleen goed zijn, hij moet vooral snel zijn. Als er gekeken wordt naar gebruikelijke video-compressie algoritmen, zoals daar zijn Xvid, Divx... zien we dat er gebruik gemaakt wordt van block matching algoritmen.

Block matching algoritmen hebben als voordeel dat ze snel en robust zijn. Ze zijn niet zo accuraat als andere methoden, maar ze zijn goed genoeg voor video-compressie. Er bestaan verschillende soorten block matching algoritmen. Deze hebben echter een groot deel gemeenschappelijk. Om te beginnen wordt een frame opgedeeld in rechthoekige blokken. Een typische grootte voor deze blokken is 16 pixels op 16 pixels. Echter kleinere of grotere blokken zijn ook mogelijk, voor meer accuratie of meer snelheid respectivelijk. Beweging binnen een blok wordt aangenomen uniform te zijn. Voor elk blok wordt er nu in het vorige frame gezocht naar de beste gelijkenis (=matching). het verschil in positie tussen het blok in het huidige frame en het vorige frame is dan de bewegingsvector voor dat blok. Block matching algoritmen nemen een maximale verplaatsing van een blok.hierdoor wordt een gebied afgebakend,het zogenaamde search-window, waarbinnen er gezocht wordt naar de beste match. Typische grootten voor deze maximale verplaatsing is 8, 16 of 32 pixels. Voor het zoeken naar de match gebruikt men verschilende methodes waarvan er nu enkelen in nader detail besproken worden.

### 2.5.1 Full Search Algoritme

Bij deze methode gaat men zoals de naam al aangeeft, elk mogelijk blok bekijken binnen het zoekvenster en zo de beste match kiezen. Het voordeel van deze methode is dat men zeker de beste keuze maakt uit alle blokken in het zoekvenster. Echter, men moet veel blokken bekijken zodat deze methode traag wordt. Er zijn dus andere methoden ontwikkled waarbij men een goede match kan selecteren zonder alle mogelijkheden af te gaan.

### 2.5.2 Secutive Elimination Algoritm

Het secutive elimination algoritme is een verbetering van het full search algoritme. Door gebruik te maken van de som van de absolute fout als selectie methode bij het bepalen van de beste match kan men vele mogelijkheden weglaten zonder deze volledig te moeten controleren.

Dit werkt met behulp van de driehoeksongelijkheid. Men kan een ondergrens van de som van de absolute fout bepalen uit het verschil van de sommen van de blokken uit het eerste en het tweede frame. Deze kunnen snel en tegen lage kost berekend worden. Op deze manier kunnen veel blokken ge/ëlimineerd worden zonder dat men ze moet controleren en kan men de snelheid ten op zichte van full search verbeteren.

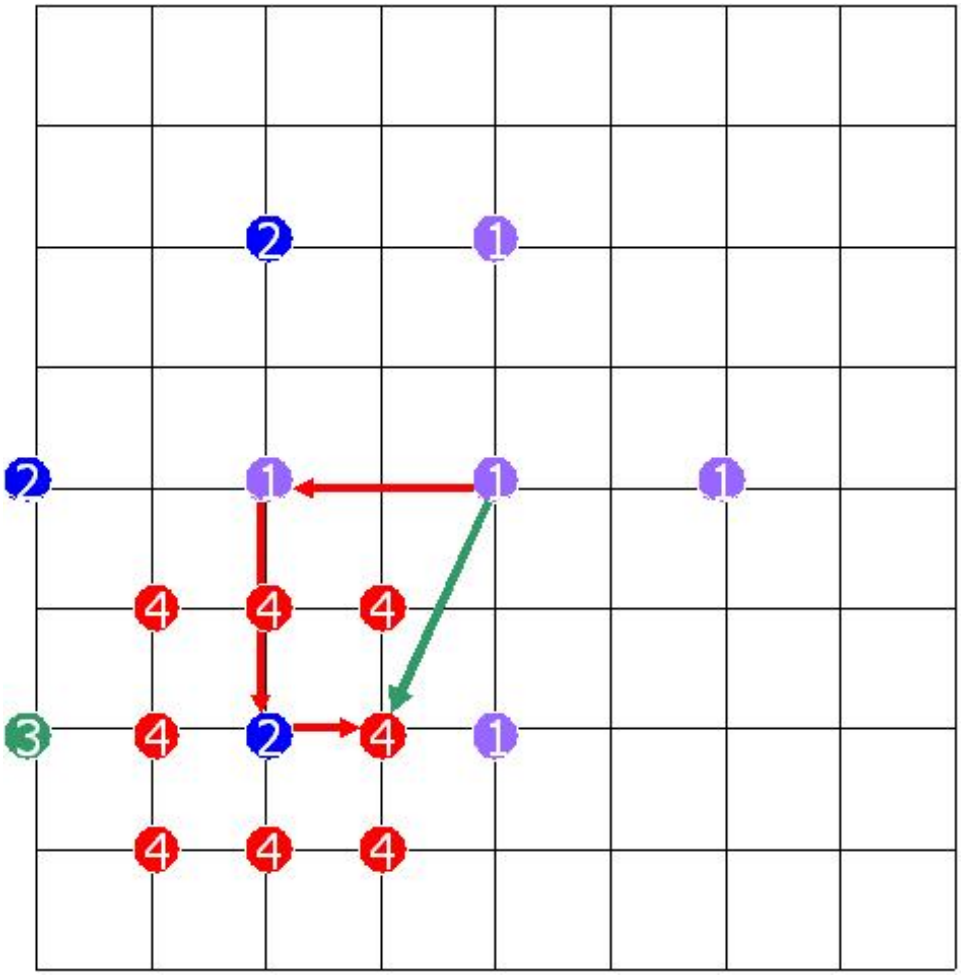
### 2.5.3 Logarimtic search

Logaritmisch zoeken werkt in een aantal stappen. Als eerste stap wordt het centrale punt en de 4 punten er rond op het midden tussen het centrum en de rand van het zoekvenster onderzocht, dus 4 punten op een afstand die de helft is van de maximaal mogelijke beweging. Uit deze 5 mogelijkheden wordt de beste match gekozen. Indien de beste match op de rand van het zoekvenster ligt, of als het het centrale punt was uit de vorige stap halveer de zoekafstand. Zoek wederom uit 5 punten de beste match. Herhaal dit tot de afstand gelijk is aan 1. Neem dan de beste match uit het centrale punt en de 8 aanliggende punten. Dit is dan de oplossing volgens logaritmic search. Voor een duidelijk diagram zie figuur 2.1.

Er dient te worden opgemerkt dat deze oplossing niet in alle gevallen optimaal is, er kan ergens in het zoekvenster een betere match bestaan, maar in de meeste gevallen is de gevonden match goed genoeg. Het voordeel van deze methode is wel dat er veel minder punten bekeken moeten worden dan onder full search, waardoor deze methode sneller is.

### 2.5.4 Three-Step Search

Three-step search, kortweg tss, vindt een match in juist geteld 3 stappen, vandaar ook de naam. Tss begint door de 8 punten rondom het centrum, in het midden tussen dat centrum en de rand. Uit deze punten en eht centrum

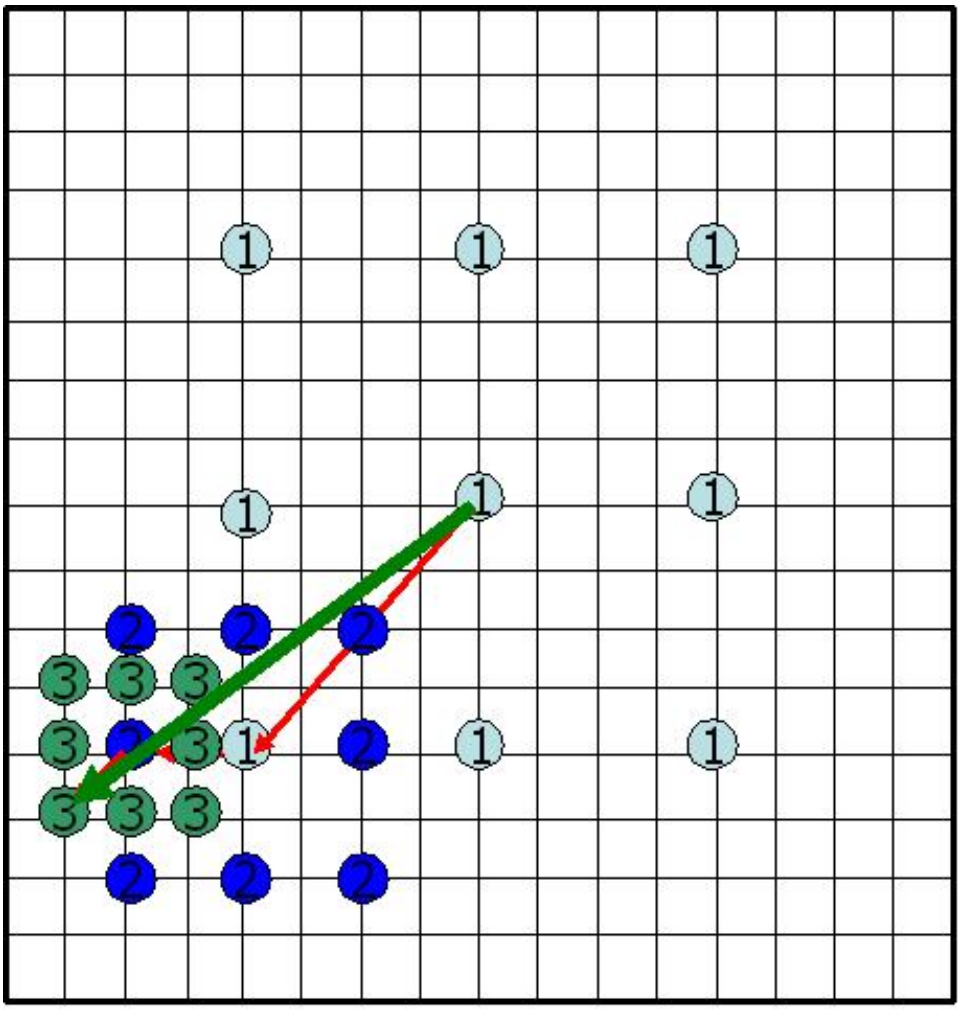



Figuur 2.1: logarimic search voorbeeld.

wordt de beste overeenkomst gekozen. Dit wordt het nieuwe centrum. De afstand wordt gehalveerd en er worden op nieuw 8 omliggende punten getest. De beste daarvan wordt opnieuw gebruikt als centrum en de afstand wordt opnieuw gehalveerd. Ook nu weer worden de 8 omliggende punten getest en de beste daarvan wordt als resultaat teruggeven. Zie figuur 2.2 voor een voorbeeld hoe three-step search te werk gaat.

Het voordeel van tss is dat men precies kan zeggen hoeveel punten er onderzocht worden en dat dit aantal constant is. Men weet dus op voorhand hoelang het zal duren. Nadeel is wel dat het resultaat niet altijd goed is. Om het resultaat te verbeteren zijn er een aantal wijzigingen voorgesteld.

Zo is er New three step search waarbij men in het centrum van het zoekvenster 8 extra punten gebruikt. Dit omdat onderzoek aantoont dat de beste match meestal in het centrum zit. De resultaten van n3ss zijn iets



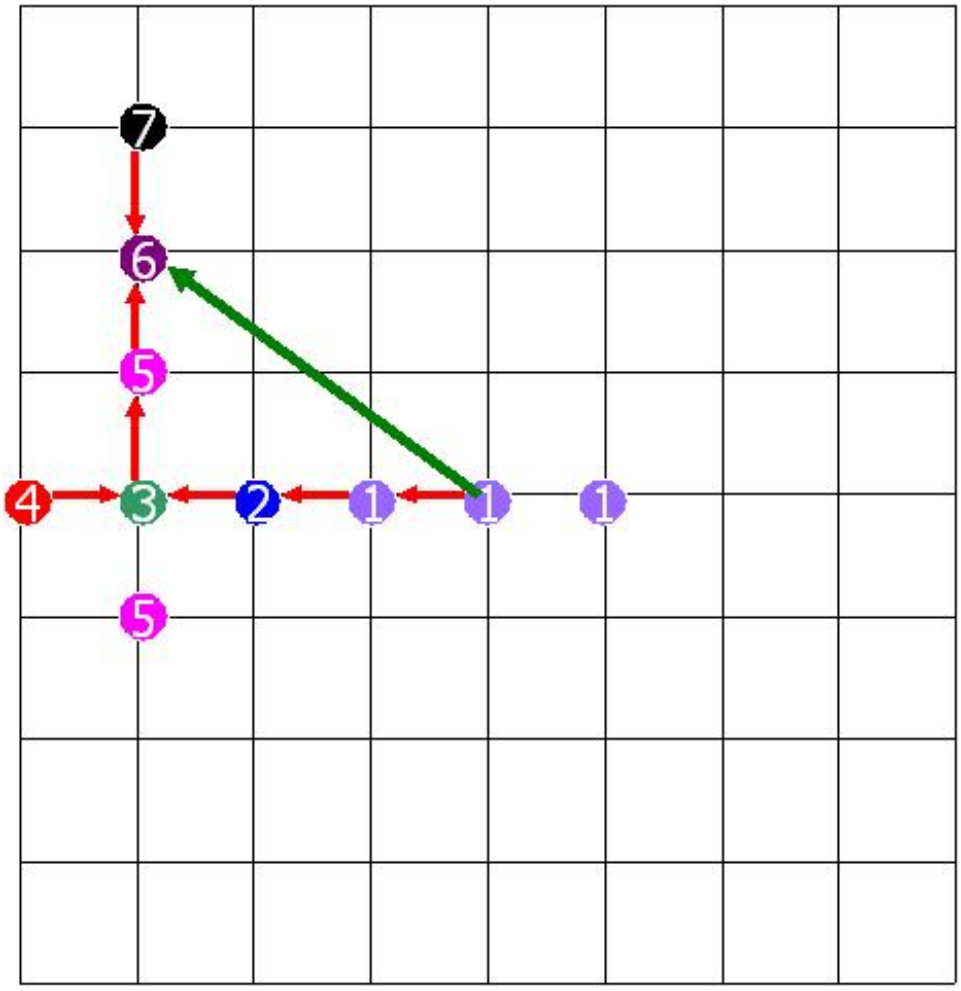

Figuur 2.2: three-step search voorbeeld.

beter en meer robust dan van tss.

Een andere verbetering is four step search. Bij deze methode onderzoekt men ook de 8 punten rondom het centrum op een afstand 2. Indien het centrum de beste match is, onderzoekt men de de 8 aanliggende punten. Indien niet pakt men de beste match als centrum en herhaalt de vorige stap. Indien men 3 keer herhaalt zonder dat de beste match in het centrum ligt neem dan de 8 aanliggende punten van de laatste beste match en onderzoek deze. De beste match wordt teruggeven als resultaat. Deze methode is iets beter dan tss en minder zoekpunten dan n3ss.

### 2.5.5 Conjugate Direction Search

Conjugate Direction search begint door het centrumpunt en de 2 aanliggende horizontale punten te onderzoeken. In de richting van de beste match wordt het volgende aan liggende punt onderzocht. Dit gaat steeds verder tot er geen verbetering meer optreed. dan onderzoekt men de 2 aan liggende verticale punten en herhaalt het hele proces maar nu in verticale richting. Dit is duidelijk te zien op figuur 2.3 Wanneer er geen verbetering meer optreed is de beste match gevonden.

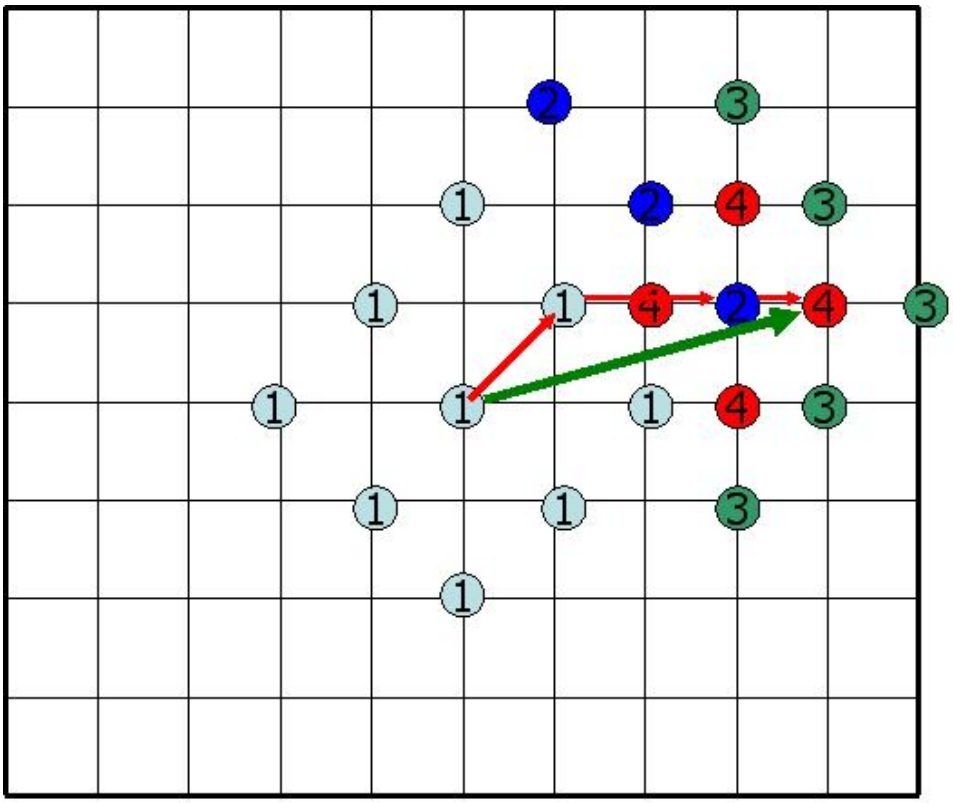



Figuur 2.3: conjugate direction search voorbeeld.

### 2.5.6 Diamond Search

Diamond search gebruikt een ruitpatroon om de zoekruimte af te tasten. Eigelijk zelfs 2 van deze patronen, n groot bestaande uit 8 punten en n klein

gevormd door vier punten. Om te beginnen wordt het centrum en de punten op het grote ruitpatroon onderzocht. Uit deze punten wordt de beste match gekozen. Indien dit niet het centrale punt is, wordt dit punt gebruikt als nieuw centrum waarrond opnieuw een grote ruit gevormd wordt. Zodra het centrale punt gekozen is, wordt de kleine ruit gebruikt. men controleert dan de punten in de kleine ruit en het centrale punt. Men kan dit process stap voor stap zien op figuur 2.4 De beste match is dan het resultaat van diamond search.



Figuur 2.4: diamond search voorbeeld2.

Diamond search is preciser dan three-step search, hij behaald ongeveer dezelfde prestaties als new three step en four step search, maar is een stuk eenvoudiger.

# Hoofdstuk 3

# Implementatie

De implementatie bestaat uit twee delen. Voor de eigelijke algortimes die door de GPU afgehandeld worden, is er gebruik gemaakt van pixel shaders. Deze zijn geschreven in Cg. Het andere deel is geschreven in OpenGL. Er zijn twee versies van de OpenGL code. De eerste versie maakt gebruikt van een Qt framework en is een opzichzelf staande applicatie. De tweede versie is een losse klasse die gebruikt maakt van glut en makkelijk gebruikt kan worden in een ander programma. Deze klasse zorgt dan zelf voor de afhandeling van de nodige environment settings zodat de programmeur er niet hoeft aan te denken. Hij kan dan deze klasse met behulp van een simpele functioncall waaraan de twee frames meegegeven worden, de motion estimation laten bereken.

## 3.1 OpenGL

Het OpenGL gedeelte is minder belangrijk dan het shader gedeelte. Er is gekozen voor OpenGL om twee reden. Ik had voorgaande ervaringen met openGL en de keuze om Cg tegebruiken als taal voor het programeren van de shaders. Een andere optie had kunnen zijn het gebruik van Microsoft DirectX$^{TM}$in combinatie met HLSL (High-Level Shader Language) maar ik had nog nooit gewerkt met DirectX of HLSL terwijl ik wel ervaring had met OpenGL en Cg. Het is wel mogelijk om Cg tegebruiken onder DirectX, maar de keuze voor Cg laat toe te kiezen voor OpenGL. Bijkomend voordeel is dat het mogelijk is om OpenGL code zonder veel problemen te porten naar verschillende operatingsystemen terwijl DirectX toch sterk gebonden is aan Microsoft Windows$^{TM}$.

Zoals eerder vermeld zijn er twee versies van de OpenGL code. Op zich is er geen verschil tussen de OpenGL code zelf, alleen moet er bij de glut versie iets meer geïnitialiseerd worden terwijl dat in Qt allemaal door het Qt framework geregeld wordt. De OpenGL code is dan ook betrekkelijk simpel. Het enige dat deze code moet doen is het aanvoeren van de data,

het oproepen van de shaders en het terug lezen van het resultaat.

Om dit te bewerkstelligen worden de frames ingelezen als standaard OpenGL textures. Dan wordt er een rechthoek getekend waarvan het oppervlak gekluerd wordt door een pixelshader. De input van deze pixelshader bestaat uit twee textures die twee opeenvolgende frames voorstellen. De grootte van de rechthoek is in dit geval gelijk aan de resolutie van het beeld waarop men motion detection wilt doen gedeeld door de grootte van de blokjes waarin het beeld verdeeld wordt. Er is dus 1 pixel per blokje uit het oorspronkelijke beeld. Standaard gebruiken we blokjes met een grootte van 16 pixels op 16 pixels. De oppervlakte van de rechthoek die we gaan tekenen is dus 256 keer kleiner dan de oppervlakte van de oorspronkelijke frames.

Nadat men de rechthoek getekent heeft en dus de shader heeft uitgevoerd wordt het resultaat uit het geheugen van de grafische kaart teruggelezen naar het processor geheugen. Op deze manier kan dit resultaat worden opgeslagen en later worden gebruikt voor de recontructie van het beeld.

Er is ook een functie om het verschil te berekenen tussen het voorspelde beeld en het oorspronkelijke beeld. Deze functie gebruikt als invoer drie textures. Ten eerste het frame waarmee we onze voorspelling gaan doen. Ten tweede de texture gegenereerd door de motion estimation shader. Ten slotte het oorspronkelijk beeld. Met behulp van de eerste twee textures wordt het oorspronkelijke beeld gereconstrueerd. Dit beeld wordt dan vergeleken met het oorspronkelijk beeld. Om deze shader te runnen wordt evens een viewport vullende rechthoek getekend, alleen is deze evengroot als de resolutie van het oorspronkelijke beeld. Ook dit resultaat kan worden terug gelezen naar het processor geheugen voor opslag of voor verdere bewerking zoals bijvoorbeeld fourier transformatie. Met behulp van dit verschilbeeld en het bijhorende motion estimation texture kan men indien het vorige frame gekend is, het oorspronkelijke beeld terug opbouwen.

## 3.2 Shaders

Er wordt gebruik gemaakt van shaders om de verschillende blokmatching algoritmes uit te voeren. De gebruikte shaders zijn pixelshaders. Er is gebruik gemaakt van de complexe branching die beschikbaar is geworden vanaf shadermodel 3.0. Dit houdt wel in dat de huidige implementatie slechts werkt op de laatste generatie Nvidia grafische kaarten daar dit de enige zijn die shadermodel 3.0 reeds implementeren. De volgende generatie Ati kaarten zullen ook shadermodel 3.0 gaan ondersteunen waardoor in principe de shaders ook op deze kaarten uitgevoerd kunnen worden.

De shaders werden geschreven in Cg, de op C gelijkende programeertaal die door Nvidia ontwikkeld is om shaders in te programeren. Hiervoor werd gekozen wegens het gebruikgemak van deze programeertaal. Ook had ik reeds een korte kennismaking met Cg achter de rug. Het werken in Cg

gaat meteen redelik vlot door de grote gelijkenis met C. Er kunnen wel geen pointers gebrukt worden in Cg maar daar de basis algoritmen eenvoudig van opzet zijn, vormt dit geen hindernis. Ik moest alleen gewoon worden aan de ingebouwde vectorrekende van Cg. Tegenwoordig kunnen vectorberekeningen met vectoren bestaande uit vier elementen of minder rechtstreeks in de GPU hardware worden uitgevoerd. Dit heeft tot gevolg dat 4 getallen optellen in fiete slechts één operatie is, mits deze getallen in een vector zitten. Een ander toffe mogelijkheid van Cg vind ik de zogenaamde swivle operator. Met deze operator kan men eenvoudig vectoren permuteren en componenten opvragen.

## 3.3 Algoritmes

Ik heb twee motion estimation algoritmen geïmplementeerd in Cg. Deze twee zijn three-stepsearch(tss) en diamondsearch(ds). Het eenvoudigste algoritme is zoals reeds eerder besproken tss.

### 3.3.1 Three-Step Search

Het tweede frame wordt opgedeeld in blokken van 16 bij 16 pixels. Dan wordt er gezocht naar een blok in het eerste frame dat het best overeenkomt met het blok in het tweede frame. Daartoe vergelijken we de pixels uit het blok uit het tweede frame met de pixels uit de kandidaat blokken uit het eerste frame. Bij de eerste stap zijn de kandidaat blokken het overeenkomstige blok en de blokken 4 pixels naar boven, onder, links of rechts verschoven. Uit deze negen blokken kiezen we het blok dat het minst verschild volgens de som van de absolute error. Dan gaan we verder met de tweede stap. We vergelijken weer ons blok uit het tweede frame met kandidaat blokken uit het eerste frame. Deze keer zijn de kandidaat blokken het juist gekozen blok en de blokken errond die 2 pixels verschoven zijn. Uit deze negen blokken wordt weer het beste blok gekozen. Ten slotte herhalen we de gehele procedure nog 1 keer maar deze keer zijn de kandidaat blokken slechts 1 pixel verschoven. Zo vinden we de beste match voor ons blok. De resultaat vector wordt gescaleerd zodat zijn componenten tussen 0 en 1 liggen. Dit is nodig omdat we een kleur terug geven en een kleur ligt als float waarde nu eenmaal tussen 0 en 1. Dit doen we zo voor elk blok. Voor elk blok zijn er altijd 27 blokvergelijkingen nodig. De performance zal dus ook niet veel schommelen.

### 3.3.2 Diamond Search

Bij diamond search gaat het iets anders. Ook hierwordt het tweede frame opgedeeld in blokken van 16 bij 16 pixels en worden deze vergeleken met kandidaat blokken uit het eerste frame. De kandidaat blokken zijn bij diamondsearch gegroepeerd in een ruit rondom het centrale blok. Er zijn twee

mogelijke ruiten. Er is de grote ruit die de 8 blokken op een grid afstand 2 bevat en er is de kleine ruit die de 4 direct aanliggende blokken bevat. Zoals altijd is get centrale blok ook een kandidaatsblok. Zolang het laatst gekozen kandidaatblok zich in het zoekvenster bevindt en dit kandidaatsblok niet het centrum van de ruit was, gebruikt men de grote ruit. Zodra men op de rand van het zoekvenster beland, of het centrale blok kiest, doet men nog een laatste test met de kleine ruit. Dit geeft dan het resultaat. Ook hier wordt het resultaat herleid naar waardes tussen 0 en 1.

# Hoofdstuk 4

# Resultaten

## 4.1 snelheid

Beide algoritmen lopen niet in realtime. Het three-step algoritme haalt ongeveer 15 frames per seconde op een NVidia gf6800gt, terwijl het diamondsearch algoritme loopt tegen 21 frames per seconde, beide bij een resolutie van 576 bij 320. Dit is niet echt snel als we bedenken dat op de CPU een Xvid of Divx filmpje kunnen bekijken welke evens gebruik maakt van diamondsearch. De CPU haalt dus een veel hogere framerate en het moet niet eens een moderne CPU zijn, terwijl de gf6800gt toch 1 van de snelste grafische kaarten is die momenteel verkrijgbaar zijn.

Als we ook naar andere resoluties kijken (zie grafiek **??**) zien we dat bij hogere resoluties threestep search minder frames per seconde kan uitrekenen. Diamondsearch daarentegen is beter bestand tegen een grotere resolutie. Ook is te zien dat op de zelfde resolutie diamondsearch sneller is dan threestep search. Diamond search loopt zelfs in realtime bij een resolutie van 384 op 284.

De reden dat de snelheid tegenvalt kan liggen aan het feit dat de Cg compiler nog niet optimaal is, en al zeker niet voor shadermodel 3.0. Een andere reden kan zijn dat het gebruik van dit shadermodel en de allerhoogste precisie de performance naar beneden haalt. Zie figuren 4.1 en 4.2 voor een voorbeeld van beide algoritmen in normale omstandigeheden. Merk opdat de reslutaten niet 100% gelijk zijn.

## 4.2 Testen

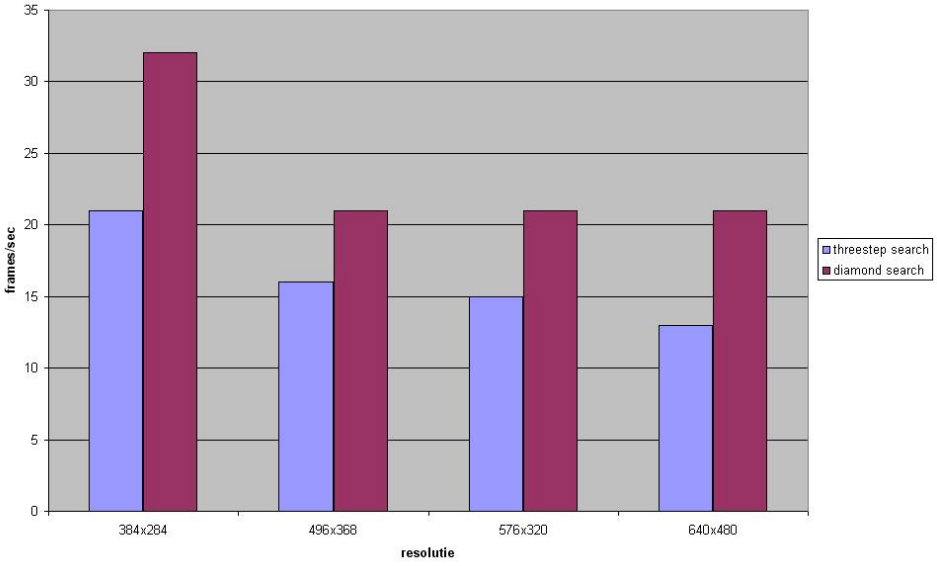
Als eerste test gebruiken we een voorbij vliegend ruimteschip. Dit is een een translaterend object, het verschuift over het beeld. In de bijhorende grafiek (4.5)kunnen we duidelijk zien dat diamondsearch een iets beter resultaat geeft. Merk op dat het differentiëelbeeld (figuur 4.4 bijna volledig grijs is wat er op wijst dat het verschil tussen het gegenereerde beeld en het werkelijke

beeld vrijwel niet bestaande is.

Het tweede voorbeeld is een scene waarin een figuur op de voorgrond een monoloog geeft, terwijl er in de achtergrond bijna geen beweging te zien is. Als we echter het differentiëel beeld en het motion field vergelijken (zie figuren 4.6 en 4.7) zien we dat een deel van de beweging in beeld niet gedetecteerd word. In deze sequentie gedraagt diamondsearch zich op frames 4 en 5 niet echt normaal en geeft grotere afwijkingen dan threestep search (zie grafiek 4.8.)

De volgende test gebruikt meer beweging: een persoon stuikt in elkaar terwijl de camera deze beweging volgt. We zien hier duidelijk dat de waardes van de mean square error bij beide algoritmes de hoogte ingaan (zie grafiek 4.11. Ook hier is de prestatie van diamond search net iets beter dan die van threestep search. Merk op dat nar het einde van de sequentie, wanneer de beweging afneemt de resultaten beter worden. Ook aan het differentie beeld (figuur 4.10 is duidelijk te zien dat in deze scene sterke afwijkingen voorkomen.

Ten slotte wordt er getoont wat er gebeurd bij een viewpoint verandering (figuur 4.12.) Daar de twee frames eigelijk geen onderlinge relatie hebben is het vectorveld random. Dit wordt opgelost met keyframes in moderne vidoecompressie software.

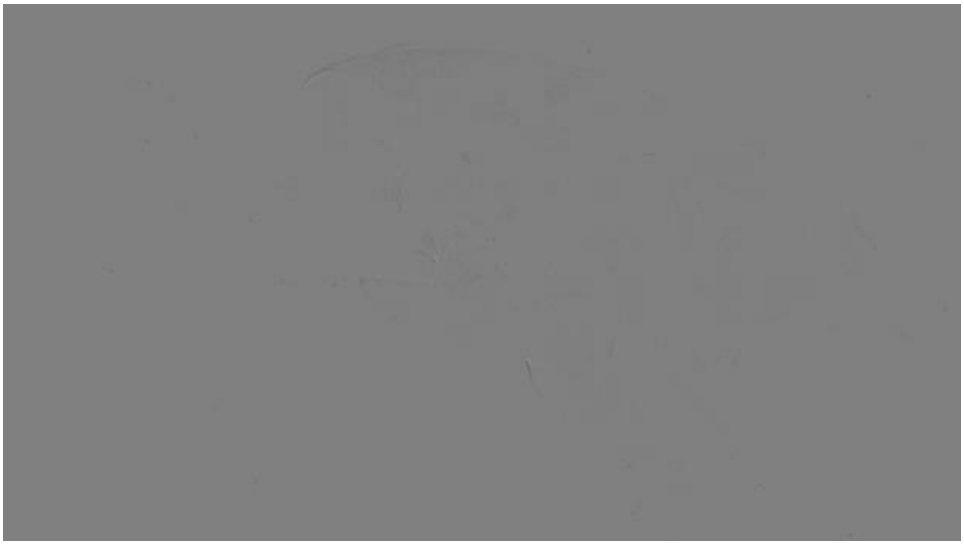



Figuur 4.1: vergelijking van snelheden

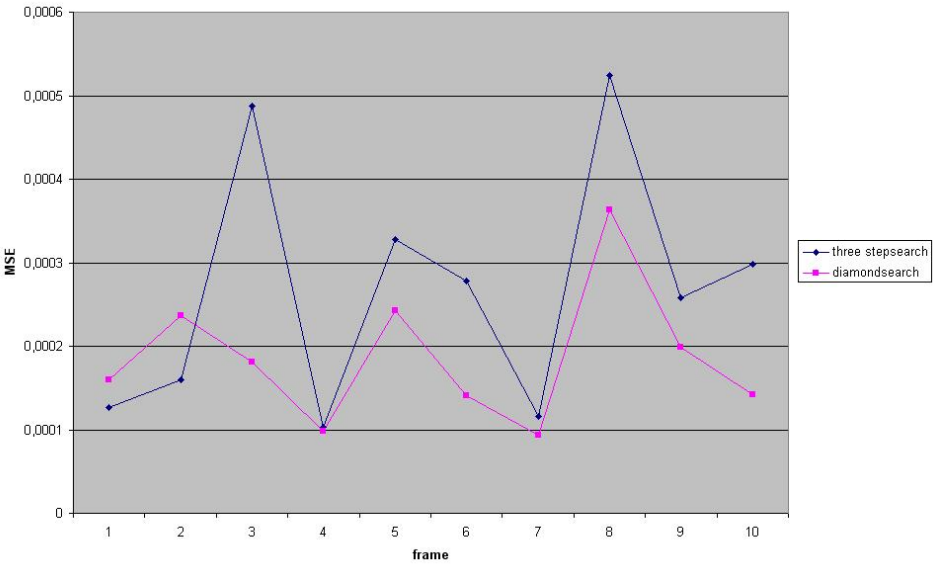

Figuur 4.2: three-step search voorbeeld, een man in een auto


Figuur 4.3: diamond search voorbeeld, zelfde man in een auto


Figuur 4.4: Translaterend object motion field

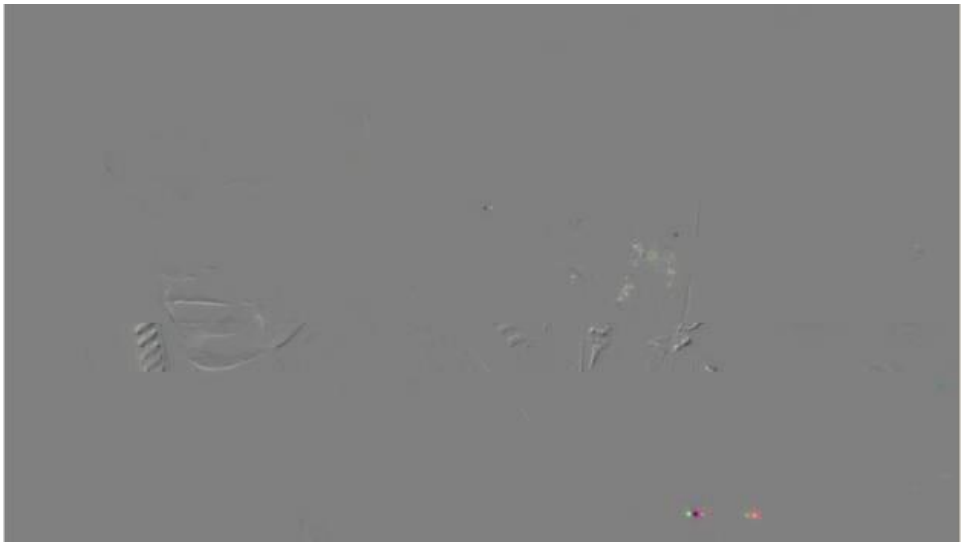


Figuur 4.5: Translaterend object differentiëel beeld

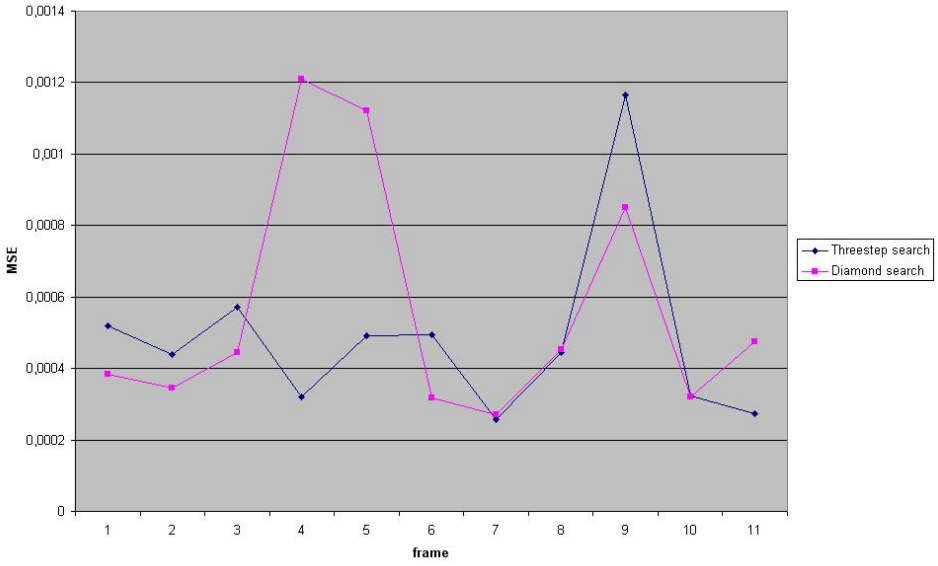



Figuur 4.6: Translaterend object vergelijkende grafiek

Figuur 4.7: Monoloog motion field

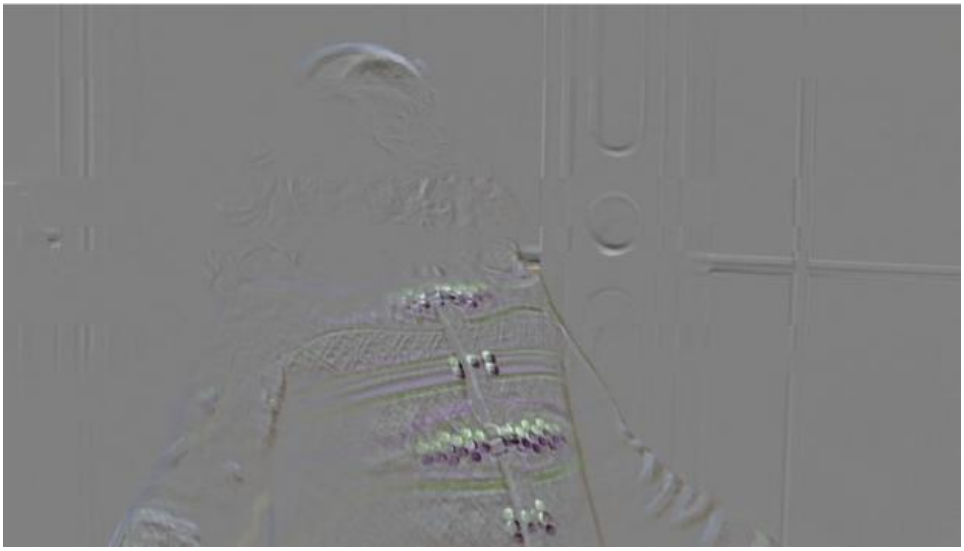


Figuur 4.8: Monoloog differentiëel beeld

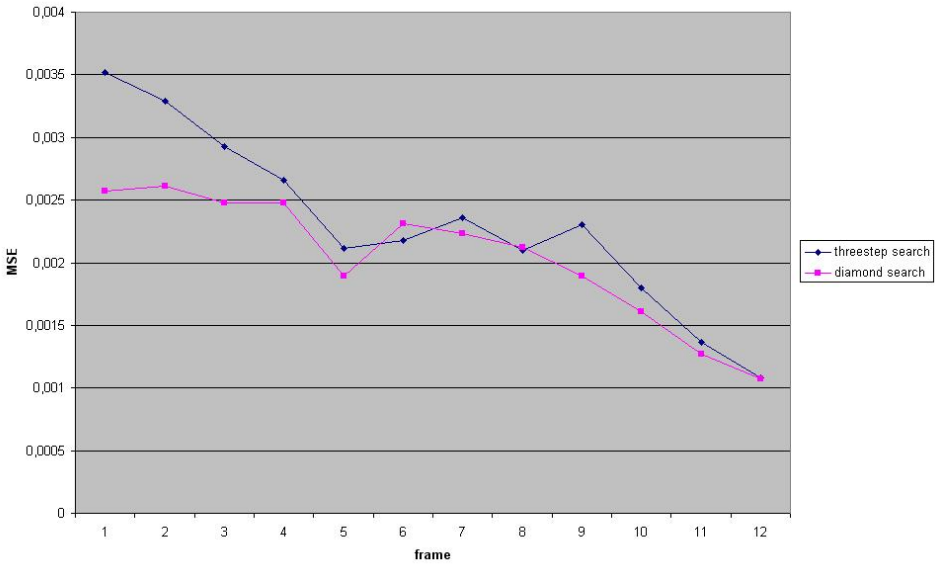



Figuur 4.9: Monoloog vergelijkende grafiek


Figuur 4.10: motion field met veel beweging



Figuur 4.11: veel beweging differentiëel beeld




Figuur 4.12: veel beweging vergelijkende grafiek

Figuur 4.13: viewpoint verandering

# Hoofdstuk 5

# Suggesties voor toekomstig werk

## 5.1 Verbeteringen

Als eerste verbetering zou het geheel sneller gemaakt moeten worden. Dit kan op verschillende manieren. Zo zou men kunnen proberen om de shaders in asembler te schrijven en zo de soms nogal klunzige Cg compiler omzeilen. Een andere mogelijkheid zou zijn om te proberen de branching instructies te elimineren. Deze branching instructies kunnen er voor zorgen dat het hele programma vertraagt, daar de hardware niet efficiënt overweg kan met deze instructies.

Een tweede verbetering zou zijn het dynamisch kunnen instellen van de window en blokgrootte en niet met defines zoals nu het geval is.

Een derde verbetering zou kunnen zijn het gebruiken van een ruw naar fijn benadering. Beginnen met het beeld op te delen in een aantal blokken bijvoorbeeld een viertal. Indien de het verschil tussen de twee gematchte blokken nog te groot is, delen we dat blok opnieuw op in een aantal blokken en herhalen de procedure. Op deze manier kan men vrij goede benaderingen maken met waarschijnlijk minder berekeningen.

## 5.2 Uitbereidingen

Als eerste uitbereiding zou ik voorstellen om ook de rest van de videocompressie te implementeren op de grafische kaart. Waarschijnlijk is het gebruik van discrete cosinus tranformaties of fast fourier transformaties het beste om te gebruiken, wegens de snelheid. Het is ook mogelijk om, zoals oorspronkelij gepland gebruik te maken van wavelet transformaties maar die gaan volgens mij niet genoeg snelheid opleveren. Het is wel zo dat wavelet transformaties voor een betere compressie kunnen zorgen.

Als tweede uitbereiding kan men simpelweg bijkomende algoritmes implementeren. De beste keuze blijven de blokmatching algoritmes. De anderen zijn waarschijnlijk te traag om gebruikt te worden voor videocompressie. Ze kunnen wel nut hebben bij andere toepassingen zoals virtual reality of object herkenning.

# Bibliografie


[Ana89] P. Anandan. A computational framework and an algorithm for the measurement of visual motion. *IJCV*, 2(3):283–310, 1989.

[BA83] Peter J. Burt and Edward H. Adelson. The laplacian pyramid as a compact image code. *IEEE Transactions on Communications*, COM-31,4:532–540, 1983.

[BFBB92] J.L. Barron, D.J. Fleet, S.S. Beauchemin, and T.A. Burkitt. Performance of optical flow techniques. *CVPR*, 92:236–242, 1992.

[BYJF97] M. J. Black, Y. Yacoob, A. D. Jepson, and D. J. Fleet. Learning parameterized models of image motion. In *IEEE Proc. Computer Vision and Pattern Recognition, CVPR-97*, pages 561–567, Puerto Rico, June 1997.

[CRS98] Giancarlo Calvagno, Roberto Rinaldo, and Luciano Sbaiz. Three-dimensional motion estimation of objects for video coding. *IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS*, 16(1):86–97, 1998.

[HS81] B. Horn and B. Schunk. Determing optical flow. *Artificial Intelligence*, 17:185–203, 1981.

[LHH+98] Hongche Liu, Tsai-Hong Hong, Martin Herman, Ted Camus, and Rama Chellappa. Accuracy vs efficiency trade-offs in optical flow algorithms. *Computer Vision and Image Understanding: CVIU*, 72(3):271–286, 1998.

[LK81] B.D. Lucas and T. Kanade. An iterative image registration technique with an application to stereo vision. In *IJCAI81*, pages 674–679, 1981.

[LO00] K. Lengwehasatit and A. Ortega. Computationally scalable partial distance based fast search motionestimation. In *Image Processing, 2000. Proceedings. 2000 International Conference on*, volume 1, 2000.

[LSP98] T. Le, W. Snelgrove, and S. Panchanathan. Simd processor arrays for image and video processing: a review, 1998.

[LV95] S.H. Lai and B.C. Vemuri. Robust and efficient algorithms for optical flow computation. In *SCV95*, page 9A Optical Flow II, 1995.

[TAL+00] Alexis Michael Tourapis, Oscar C. Au, Ming Lei Liou, Guobin Shen, and Ishfaq Ahmad. Optimizing the mpeg-4 encoder - advanced diamond zonal search. In *Proceedings of 2000 International Symposium on Circuits and Systems (ISCAS-2000)*, Geneva, Switzerland, May 2000. IEEE.

[TMB+03] Stanimire Tomov, Michael D. McGuigan, Robert Bennett, Gordon Smith, and John Spiletic. Benchmarking and implementation of probability-based simulations on programmable graphics cards. *CoRR*, cs.GR/0312006, 2003.

[WM95] Joseph Weber and Jitendra Malik. Robust computation of optical flow in a multi-scale differential framework. *International Journal of Computer Vision*, 14(1):5–19, 1995.