Seam Carving: intelligent afbeeldingen en video verkleinen

Bachelorscriptie Cognitieve Kunstmatige Intelligente, Universiteit Utrecht
7.5 ECTS

9 juli 2011

Auteur
Sander Knape

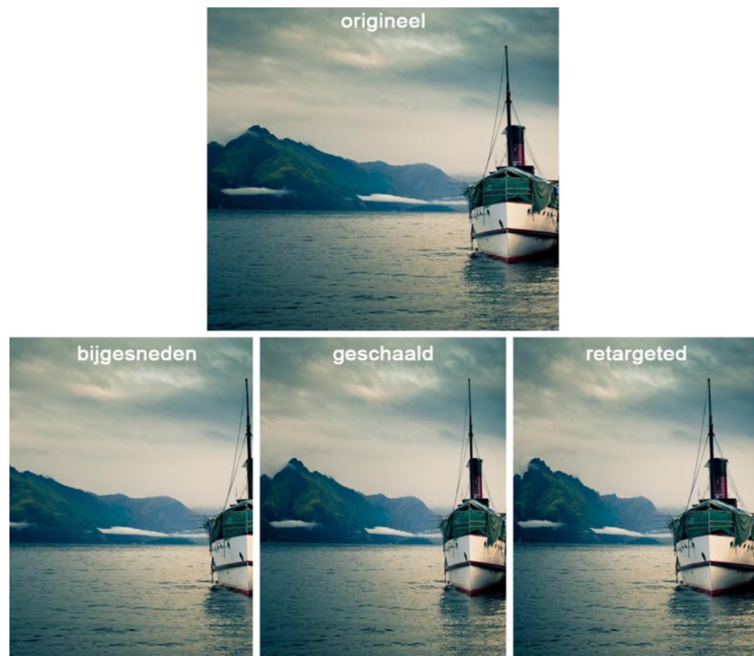
Begeleiders
dr. G.A.W. Vreeswijk
T.C. van Dijk, MSc

Abstract

De huidige manieren om afbeeldingen en video's te verkleinen in de hoogte of breedte houden geen rekening met de semantische inhoud van de afbeelding. Met de techniek *seam carving* wordt dit wel gedaan. Elke pixel krijgt hierbij een energiewaarde en om een afbeelding te verkleinen wordt een *seam* berekend, een horizontaal of verticaal pad om een plaatje respectievelijk verticaal of horizontaal met 1 pixel te verkleinen. De seam volgt een pad met de laagste energiewaarde om ervoor te zorgen dat de belangrijke onderdelen van het plaatje behouden blijven. Het is ook mogelijk om dit in een video te doen, waarin of voor elk frame een 1-dimensionaal pad wordt gevonden, of over de gehele 3D set een 2-dimensionaal vlak wordt berekend. In deze paper beschrijf ik, aan de hand van mijn eigen implementatie, een aantal technieken en algoritmen die hiervoor gebruikt kunnen worden.

Inleiding

Seam carving, oorspronkelijk gecreëerd met als doel om afbeeldingen zonder vervormingen weer te kunnen geven op verschillende media zoals smartphones, tablets en computers, is ontwikkeld door [Avidan & Shamir, 2007]. Waar het lettertype van de tekst op een webpagina verkleind of vergroot kan worden voor de verschillende soorten media, is dit voor afbeeldingen niet het geval. In figuur 1 zijn verschillende manieren te zien om een foto te verkleinen, waaronder d.m.v. seam carving. Het probleem met bijsnijden is duidelijk: veel inhoud van het plaatje verdwijnt en in dit geval moet er gekozen worden tussen het volledig in beeld laten van de boot, de berg, of een beetje van beide. Bij het schalen blijft wel de volledige inhoud in beeld maar de keerzijde van de medaille is dat er in dit geval vervormingen optreden, vooral goed te zien in de boot. Seam carving behoud vooral de belangrijke data en de boot ziet er dan ook vrijwel exact hetzelfde uit waar de berg een stuk kleiner is gemaakt. Vervormingen zijn echter nauwelijks te zien.



Figuur 1: De verschillen tussen bijsnijden, schalen en verkleinen met behulp van *seam carving* .

Hetzelfde is mogelijk voor video. In dit geval wordt er van een andere techniek gebruik gemaakt, omdat de oorspronkelijke techniek niet op de data van een video werkt. Voor video is het vooral belangrijk dat niet elk frame apart wordt bekeken. maar er rekening wordt gehouden met de omliggende frames.

Voor beide soorten media –afbeeldingen en video - heb ik een seam carving implementatie geschreven. Dit paper zal voornamelijk ingaan op de gebruikte algoritmen, implementatie en de resultaten van deze implementatie. Als richtlijnen voor de implementatie is [Avidan & Shamir 2007] gebruikt voor de afbeeldingen en [RubinStein, Avidan, Shamir 2008] voor de video's. Het doel van dit paper is niet om deze resultaten te verbeteren maar vooral om de gebruikte technieken te implementeren en op deze manier kennis te maken met de meer geavanceerde technieken binnen Image Manipulation.

Eén van de vele onderwerpen binnen de Kunstmatige Intelligentie waar momenteel onderzoek naar wordt gedaan is objectherkenning en beeldverwerking. Met behulp van bijvoorbeeld randdetectie wordt een poging gedaan om objecten binnen een plaatje te identificeren [Rosenfeld & Thurston, 1981]. De toepassingen hiervoor zijn eindeloos; een uitgebreid overzicht hiervan is te vinden in [Hirano et al, 2006]. Seam carving werkt tot een zeker niveau op een soortgelijke manier: met behulp van een saliency map of een edge detection algoritme wordt bekeken waar de meeste energy (intensiteit, lokaal verschil) in het plaatje zit. Vervolgens wordt *alleen deze informatie* gebruikt om de

belangrijke delen van het plaatje zo intact mogelijk te houden tijdens het verkleinen van het plaatje. Daarmee is deze techniek ook minder computationeel zwaar terwijl dit toch voor goede resultaten zorgt.

De opbouw van de paper is als volgt. Eerst vertel ik nog wat meer achtergrondinformatie om daarna over te gaan op uitleggen hoe een seam in elkaar zet. Hierna ga ik verder met het uitleggen van de verschillende energiefuncties en hoe ik hier gebruik van maak met twee verschillende algoritmen om een seam in een plaatje te vinden. Dit doe ik met dynamisch programmeren en met een flow network. Hierna ga ik over op video en hoe ik het flow network algoritme hiervoor op een 3D set toe pas. Daarna volgen nog wat concluderende woorden.

Achtergrond

De laatste jaren zijn er meerdere onderzoeksresultaten verschenen waarin wordt gepoogd afbeeldingen op zo'n manier te verkleinen dat de belangrijke inhoud bewaard blijft. Een voorbeeld komt van [Suh et al, 2003], waarin met behulp van een *saliency map* wordt berekend waar zich de belangrijkste inhoud van een plaatje bevind, om vervolgens automatisch door bij te snijden een thumbnail te maken. In een ander voorbeeld van [Chen et al, 2003] wordt ook de belangrijkste inhoud van een plaatje bepaald. Alleen dit deel van de afbeelding kan daarna gebruikt worden om op een kleiner display - zoals die van een smartphone – weer te geven.

Naast deze bottom-up methodes zijn er ook verschillende top-down manieren om de belangrijkste inhoud van een plaatje te bepalen. Zo kan dit door met behulp van eye tracking te kijken waar mensen hun blik op richten [Santella et al, 2006]. Ook kan het met behulp van gezichtsherkenning om te zorgen dat gezichten niet vervormd worden [Viola & Jones, 2001] door de pixels van het gezicht kunstmatig hogere energiewaarden te geven. Een andere methode zou zijn via een algemenere vorm van object herkenning

Ook voor video zijn er al verschillende methodes ontwikkeld om deze op een slimme manier te verkleinen. Eén zo'n methode is de video te segmenteren in voor- en achtergrond lagen [Tao et al, 2007].

Seams

Wanneer het doel is om de minst belangrijke data uit een afbeelding als eerste te verwijderen, zou een eerste stap zijn om te beginnen met simpelweg de minst belangrijke pixels uit de afbeelding te verwijderen. Dit is wat Avidan & Shamir initiëel probeerde. Bepalen hoe belangrijk een pixel is wordt gedaan met een *energiefunctie*. Hier kom ik in de volgende sectie op terug. Laten we er in deze sectie nog van uit gaan dat elke pixel ook daadwerkelijk een berekende energiewaarde heeft. Stel dat het de bedoeling is om een plaatje in de breedte met 100 pixels te verkleinen. Een methode is dan om per rij de 100 minst belangrijke pixels te selecteren en deze te verwijderen waarna alle overgebleven

pixels zoveel mogelijk naar links worden geschoven om de lege gaten op te vullen. Een probleem waar Avidan & Shamir hiermee tegenaan liepen was een zigzag effect. Omdat de pixelkeuze onafhankelijk is van de omliggende rijen was er geen samenhang binnen de verwijderde pixels te vinden. De oplossing hiervoor was het vinden van een pad – in deze situatie verticaal – waarbij er alsnog in elke rij een pixel wordt gekozen maar deze pixel verbonden is aan de pixels die in de rij erboven en eronder gekozen zijn. Dit zorgde voor een aanzienlijk beter resultaat, en het begrip *seam carving* was hiermee geboren [Avidan & Shamir, 2007]. In Figuur 2 is een seam in een plaatje te zien.

In de volgende sectie bespreek ik het bepalen van de energiewaarde van een pixel. Daarna kom ik weer terug op seams en bespreek ik de verschillende algoritmen die ik gebruikt heb om een seam te vinden.

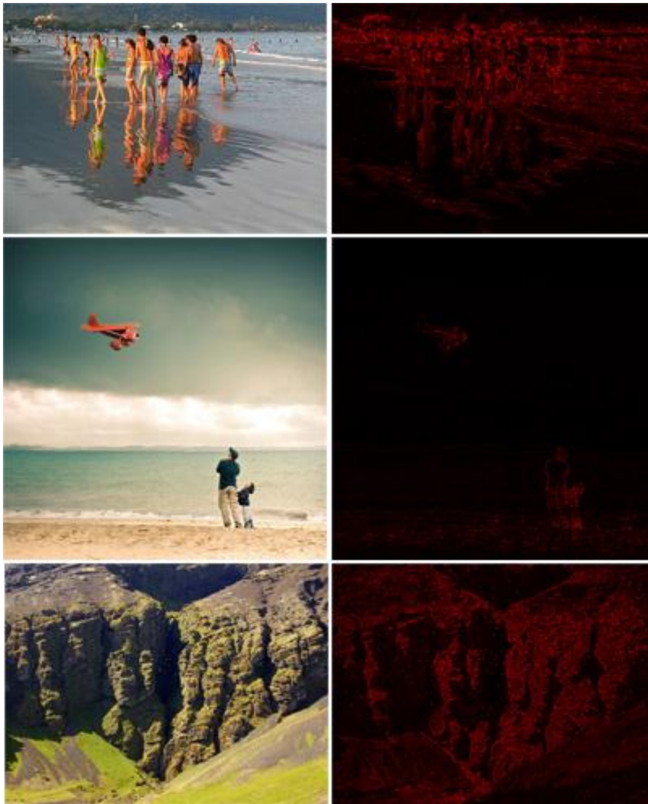
Energiewaarden

Het seam carving algoritme zoekt een pad van de laagste energie, horizontaal of verticaal, binnen een plaatje. Om te beginnen zal er dus een energiefunctie moeten worden gebruikt om deze energiewaarden te bereken. In Figuur 3 is een manier te zien om dit vervolgens grafisch te weergeven: des te roder de pixel, des te meer energie deze pixel heeft.

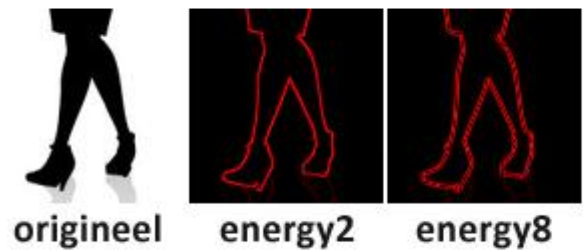
Avidan en Shamir hebben meerdere methoden om de energiewaarde te berekenen getest, waaronder *Histogram of Gradients*, *saliency measure* en de *Harris-corners Measure*. Niet geheel onverwacht was de conclusie dat geen van de methoden op alle afbeeldingen die ze getest hadden het beste resultaat gaf [Avidan & Shamir, 2007]. Eén van de methoden die het beste werkte was de *Histogram of Gradients* methode. Deze methode berekent het verschil in kleurwaarde van elke pixel met een bepaald aantal van de acht directe buren [Dalal & Triggs, 2005]. Ik heb hier drie verschillende versies van geïmplementeerd. De eerste versie kijkt naar de rechter- en onderbuur (*Energy2*), de tweede naar de 4 niet-diagonale buren (*Energy4*) en de laatste naar alle acht de omringende buren (*Energy8*). Mijn eerste doel was uit te zoeken welke van de energiefuncties in de meeste gevallen de beste resultaten gaf. *Energy4* bleek hiervan niet erg interessant omdat deze tussen de twee 'extremen' in zat: de verschillen tussen *Energy2* en *Energy8* waren interessanter om te vergelijken. In Figuur 4 zijn deze beide versies te zien.

Een belangrijk verschil tussen *Energy2* en *Energy8* is dat *Energy2* bij hoge contrasten een duidelijke grens van één pixel breed heeft, terwijl *Energy8* hier grenzen van twee pixels breed heeft. Dit komt doordat *Energy2* alleen naar de rechter- en onderbuur kijkt. De andere kant van de contrastlijn kijkt niet naar de linker- en bovenbuur, dus aan deze kant van de lijn wordt geen hoge energiewaarde gemeten. *Energy8* kijkt naar alle acht de buren dus hier zal aan alle kanten van de contrastlijn een hogere energiewaarde gevonden worden. Dit verklaart het verschil te zien in Figuur 4.

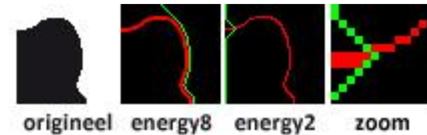
Een gevolg hiervan is te zien in Figuur 5. De rode pixels zijn de energie waardes, de groene lijnen zijn de gevonden seams. Te zien is dat in de *Energy8* versie de seam netjes om de zwarte figuur heen gaat. Bij *Energy2* vind het algoritme een pad door de lijn heen waardoor het pad door de figuur heen gaat. Omdat de figuur zelf compleet zwart is, zijn hier geen kleurverschillen: de energiewaarden zijn hier allemaal nul waardoor het pad vervolgens rechtdoor naar beneden gaat. De reden dat *Energy2* het figuur in gaat is te zien in het ingezoomde plaatje in Figuur 5. Omdat deze figuur maar naar twee buren kijkt is er hier geen sprake van een dubbele rand. Op hoeken ontstaat daardoor een gekartelde energielijn. Omdat het pad ook diagonale stappen mag zetten kan deze dus door de lijn heen gaan



Figuur 3: De energiewaarden van verschillende afbeeldingen.



Figuur 4: Het verschil tussen *Energy2* en *Energy8.*



Figuur 5: In sommige gevallen zorgen de verschillende energiefuncties ook voor verschillende resultaten.

zonder over een pixel met een hoge energiewaarde heen te moeten lopen. Omdat het de bedoeling is dat voorgrond objecten binnen een figuur juist niet worden verkleind, lijkt dit een stevige tekortkoming van *Energy2* te zijn.

In Figuur 5 heeft de rand van de zwarte figuur geen *anti-aliasing.* Anti-aliasing is een techniek om kartellijnen minder duidelijk weer te geven door de pixels om de rand heen een kleur te geven wat tussen de achtergrondkleur en de figuur-kleur in ligt[1]. Het contrast wordt hierdoor minder, omdat de pixels om de rand heen nu ook voor energieverschillen zorgen. Hierdoor is het voor *Energy2* moeilijker om binnen een figuur te komen, omdat de lijn bij randen dikker wordt. Bij *Energy8* word de rand echter nog dikker, waardoor *Energy2* alsnog vaker binnen een figuur komt. Na veel foto's getest te hebben kwam ik ook tot de observatie dat *Energy8* vaak betere resultaten geeft. Om deze reden wordt er in de rest van het onderzoek gebruik gemaakt van *Energy8*.

Seams: Dynamisch Programmeren

Het eerste algoritme dat ik heb gebruikt is een DP-algoritme. De implementatie hiervan is gebaseerd op de algemene uitleg van het DP-algoritme in [Cormen et al, 2009]. Er is zowel sprake van een *optimal substructure* en *overlapping subproblems*. Het algoritme bouwt eerst een matrix op waarin voor elke pixel het korste pad ernaartoe wordt berekend (*optimal substructure*). Door deze waarden op te slaan hoeven we deze pixels niet telkens opnieuw te berekenen wanneer een andere pixel de waarde van deze pixel nodig heeft om zijn eigen waarde te berekenen (*overlapping subproblems*).

[1] Anti-aliasing wordt erg helder uitgelegd op http://nl.wikipedia.org/wiki/Anti-aliasing

Het doel is een pad te vinden dat zo “goedkoop” mogelijk is: de totale som van de pixels die weggehaald gaan worden moet zo laag mogelijk zijn. Dit is als volgt gedefinieerd waabij s* de optimale seam is, s een willekeurige seam en E(s) de energiewaarde van de gegeven seam.

$$s^* = \min_{s} E(s)$$

De optimale seam wordt gevonden door van alle mogelijke seams te seams te nemen met de minste energy. Het vinden van het verticale pad werkt als volgt. Van de bovenste rij pixels wordt de energiewaarde in een nieuwe matrix opgeslagen. Van elk rij daaronder wordt voor elke pixel de goedkoopste van de drie bovenburen genomen – door één van deze pixels te kiezen is het uieindelijke pad aaneengesloten - en wordt dit opgeteld bij de energiewaarde van deze pixel. Deze nieuwe waarde van de pixel is dan de som van het goedkoopste pad om bij deze pixel te komen. In formele form;

$$M(i,j) = e(i,j) + \min\big(M(i-1,j-1), M(i-1,j), M(i-1,j+1)\big)$$

Hier staat M voor de cumulatieve energiewaarde voor het pad van de bovenste rij naar deze pixel. De *e* staat voor de energiewaarde voor die pixel.

In Figuur 6 staat een voorbeeldafbeelding met de energie- en padwaarden die met de bovenstaande methode wordt gevonden. Wanneer de volledige matrix voor de padwaarden is gevuld, bekijken we de onderste rij en nemen we de laagste waarde die we kunnen vinden: dit is de eindbestemming van het goedkoopste pad mogelijk (als er meerdere pixels met dezelfde waarde zijn is het afhankelijk van de implementatie welke wordt gekozen: in mijn implementatie wordt altijd de meest linker gekozen). Vervolgens vinden we het uiteindelijke pad door van de bovenste drie buren steeds de laagste waarde te nemen.

0	255	255	255	255	255	255	0	0	0
0	192	255	255	192	255	255	0	0	0
0	96	192	192	96	192	255	93	0	0
0	0	0	0	0	96	255	255	255	255
0	0	0	0	0	0	96	192	255	255
255	255	192	96	0	0	96	192	255	255
255	255	255	255	96	0	192	255	255	255
0	0	93	255	192	0	192	255	255	255
0	0	0	255	255	0	96	192	255	255
0	0	0	255	255	0	0	0	0	0

energiewaarden

0	255	255	255	255	255	255	0	0	0
0	192	510	510	447	510	255	0	0	0
0	96	384	639	543	447	255	93	0	0
0	0	96	384	447	351	348	255	255	255
0	0	0	96	351	348	351	447	510	510
255	255	192	96	96	348	444	543	702	765
510	447	351	351	192	96	540	699	798	957
447	351	444	447	288	96	288	795	954	1053
351	351	351	543	351	96	192	480	1050	1209
351	351	351	606	351	96	96	192	480	1050

padwaarden

plaatje

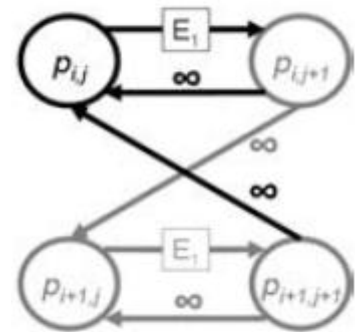
Figuur 6: Energie- en padwaarden die gevonden worden tijdens het berekenen van een seam met het DP-algoritme. Voor het gemak zijn zowel het plaatje als de energiewaarden in een grid gezet. Het plaadje zelf bestaat uit alleen zwarte en witte pixels.

Seams: Flow Network

Een andere methode die ik heb gebruikt om een seam binnen een plaatje te vinden is met behulp van een flow network. De reden om deze methode ook te gebruiken is dat Avidan & Shamir in hun vervolgonderzoek naar seam carving voor video’s gebruik maken van dit algoritme, omdat het DP-algoritme niet te gebruiken is op een 3D matrix [RubinStein, Avidan & Shamir, 2008]. Voordat ik de stap maakte naar video, heb ik dit algoritme eerst op 2D afbeeldingen toegepast. Rubenstein, Avidan en Shamir stellen in hun artikel dat het gebruik van een flow network dezelfde seams zou moeten opleveren als het DP algoritme (wanneer beide dezelfde energiefunctie gebruiken). Dit is iets was ik met eigen ogen wilde zien en het was een goed opstapje naar werken met een 3D matrix.

Een flow network werkt met nodes en edges. Er is dus een mapping nodig van de pixels naar deze nodes en edges. Deze mapping is goed te zien in Figuur 7. Voor elke pixel wordt een node aangemaakt. Elke node krijgt 4 edges;

- Een edge naar voren (rechts). De capaciteit van deze edge is de energiewaarde van de pixel, in Figuur 7 te zien als E_1.
- Een edge naar achteren (links). Deze edge is nodig om de *monotoniciteit* (zie volgende paragraaf) te bewaren, en heeft als capaciteit oneindig.
- Twee edges diagonaal naar achteren (links). Deze edges zijn nodig om de *connectiviteit* (zie volgende paragraaf) te bewaren en hebben beide ook als capaciteit oneindig.



Figuur 7: Mapping van pixels naar een graaf. Voor elke pixel wordt een node gemaakt met bijbehorende edges.

RubinStein, Avidan en Shamir stellen dat een minimal cut moet worden gevonden om een seam te vinden. Deze moet zich aan een aantal eisen om de juiste seam te vinden houden: de seam moet exact 1 keer over elke rij gaan (de *monotonicity* eis) en de pixels van de seam moeten aaneengesloten zijn (de *connectivity* eis). Zoals hiervoor beschreven zijn er een aantal extra edges nodig om dit te bereiken; de uitleg hiervan wordt beschreven in [RubinStein, Avidan & Shamir, 2008].

Om de minimal cut te vinden maak ik gebruik van het Edmonds-Karp algoritme, een variatie op het Ford-Fulkerson algoritme [Cormen et al, 2009]. Ford-Fulkerson zoekt een augmenterend pad – en pad met beschikbare capaciteit – om de capaciteit hiervan vervolgens te verlagen. Edmonds-Karp maakt gebruik van een breadth-first search om te garanderen dat het pad dat gevonden wordt het kortste pad is. Dit heeft wel als gevolg dat wanneer er eenmaal een pad gevonden wordt een groot deel van de graaf bekeken is. Hier kom ik op terug in het volgende gedeelte. Naast de edges die ik hierboven beschreven heb maak ik een node SOURCE aan, met edges naar alle pixels in de meest linker rij van het plaatje. Ook maak ik een node TARGET aan, met edges die komen van alle pixels in de meest rechter rij van het plaatje. Alle edges van de SOURCE en naar de TARGET hebben capaciteit oneindig.

Het algoritme zoekt een pad van de SOURCE naar de TARGET. Wanneer een pad gevonden is, wordt de capaciteit van elke edge verlaagd met de laagste capaciteit die in dat pad aanwezig is: hiermee wordt de capaciteit van minstens één edge dus 0. Van elke node wordt een "kleur" bijgehouden, een variabele die bijhoud of de node al bekeken is of niet. De standaard terminologie hiervoor is dat nodes standaard wit zijn, en wanneer ze bekeken worden grijs worden gemaakt. Op het moment dat het algoritme geen pad meer kan vinden omdat de edges niet meer de benodigde capaciteit hebben, hebben de nodes allemaal wel een kleur in deze eindtoestand. Met behulp van deze kleuren wordt de minimal cut gevonden.

G	W	W	W	W	W	W	W	W	W
G	W	W	W	W	W	W	W	W	W
G	G	W	W	W	W	W	W	W	W
G	G	G	W	W	W	W	W	W	W
G	G	G	G	W	W	W	W	W	W
G	G	G	G	G	W	W	W	W	W
G	G	G	G	G	G	W	W	W	W
G	G	G	G	G	G	W	W	W	W
G	G	G	G	G	G	W	W	W	W
G	G	G	G	G	G	W	W	W	W

Figuur 8: Met behulp van het Flow Network algoritme wordt hetzelfde pad gevonden als met het DP-algoritme. De "G"-waarden zijn voor grijze pixels, de "W"-waarden voor witte pixels.

In Figuur 8 is zo'n eindtoestand te vinden voor hetzelfde plaatje als in Figuur 6. Als we alle grijze nodes pakken waar

aan de rechterkant een witte node grenst, hebben we de minimal cut te pakken. In vergelijking met Figuur 6 is te zien dat met deze minimal cut dezelfde seam wordt gevonden als met het DP-algoritme.

Dynamisch Programmeren vs. Flow Network

Avidan & Shamir hebben bewust gekozen voor het DP-algoritme voor de 2D afbeeldingen, omdat dit een simpele en snelle manier is om de seams te berekenen. We kunnen dus verwachten dat het Flow Network algoritme langzamer is. Tijdens het uitvoeren van beide algoritmes op afbeeldingen bleek al snel dat dit ook het geval was. Om de reden hiervan te achterhalen heb ik een kleine analyse gedaan.

Tijdens het DP-algoritme heeft elke pixel de informatie van de drie bovenliggende pixels nodig om tot een padwaarde te komen. Door de pixels van boven naar onder te doorlopen moeten we elke pixel exact één keer bekijken. Het algoritme heeft dus een looptijd van O(X * Y), waarbij X de breedte van het plaatje in pixels is, en Y de hoogte van het plaatje in pixels.

Bij het Flow Network algoritme ligt dit anders. Het algoritme houdt pas op wanneer er geen pad van de SOURCE naar de TARGET meer gevonden kan worden. Omdat de edges geen “sprongen” maken tussen pixels die meer dan één stap van elkaar afliggen, is het pad van de SOURCE naar de target dus op zijn minst de breedte van het plaatje + 2. Omdat we een BFS algoritme gebruiken is het aannemelijk dat minstens 90% van alle pixels is bekeken op het moment dat er een pad wordt gevonden (zolang de edges dit toelaten). Het rendement is laag: om de capaciteit van één pad te verlagen moet een erg groot deel van het plaatje worden doorlopen. Daarbij is het ook nog eens zo dat worst-case maar één van de edges op het gevonden pad “op slot wordt gezet”, dat wil zeggen dat de capaciteit hiervan 0 wordt. Het vinden van één pad is dus maar een kleine stap in het hele proces.

Hoeveel paden er gevonden worden hangt af van de hoeveelheid energie die zich in het hele plaatje bevindt. In de voorbeelden die ik heb gebruikt maak ik gebruik van maar 2 kleuren: er is een hoog contrast tussen de kleuren en het is makkelijk te zien of de seams die gevonden worden er uitzien alsof ze kloppen. In foto’s zijn echter veel verschillende kleuren en pixels die naast elkaar liggen zullen vaak wel op zijn minst een klein beetje verschillen. Hierdoor zit er veel energie in het plaatje wat als resultaat heeft dat er veel paden gevonden zullen moeten worden voordat er eindelijk genoeg edges op slot zijn gezet waardoor er geen pad meer te vinden is.

Om bovenstaande te kunnen bevestigen heb ik een kleine test met 10 verschillende afbeeldingen

breedte	hoogte	#pixels	gemiddeld aantal bekeken nodes per gevonden pad	percentage bekeken nodes per gevonden pad	aantal gevonden paden	totaal aantal bekeken nodes
500	100	50000	49831	99.66200%	68	3388508
240	120	28800	28213	97.96181%	910	25673830
250	167	41750	40843	97.82754%	1081	44151283
500	500	250000	240464	96.18560%	1159	278697776
350	500	175000	167471	95.69771%	1959	328075689
400	300	120000	113355	94.46250%	1004	113808420
424	660	279840	263689	94.22849%	1363	359408107
100	75	7500	6824	90.98667%	61	416264
300	300	90000	74400	82.66667%	37	2752800
151	72	10872	8204	75.45990%	76	623504

Tabel 1: De resultaten van een aantal tests op verschillende afbeeldingen met het Flow Network algoritme.

gedaan. Op twee afbeeldingen na zijn al deze afbeeldingen 'drukke' afbeeldingen waar veel energie in zit. De resultaten staan in Tabel 1. Ik geef hierin de dimensies van het plaatje aan en voor het gemak het aantal pixels dat in het plaatje zitten. Daarnaast laat ik zien hoeveel nodes (voor elke pixel bestaat één node) er gemiddeld worden bekeken tijdens het zoeken naar een pad. Ook is te zien hoeveel procent dit van het totaal is. Daarnaast laat ik zien hoeveel paden er in totaal worden gevonden en hoeveel nodes er in totaal worden bekeken.

Wat vooral opvalt is dat inderdaad een erg groot deel van alle pixels wordt bekeken tijdens het zoeken naar een pad. De twee afbeeldingen die onderaan de grafiek staan zijn simpele logo's op een matte achtergrond met weinig energie in het plaatje. Interessant is te zien dat er voor deze afbeeldingen relatief weinig paden gevonden worden en dat er een kleiner aandeel aan nodes word bekeken tijdens het zoeken. De andere afbeeldingen zijn foto's van steden, mensen en landschappen. Met de grote hoeveelheid energie in deze afbeeldingen is het blijkbaar nodig om veel paden te zoeken waarbij het steeds nodig is om een groot deel van alle pixels te doorlopen. Een laatste opmerking die ik wil maken is dat de afbeeldingen nog relatief klein zijn: de grootste is 424x60 pixels. Hiervoor was het nodig om ruim 359 *miljoen* nodes te bekijken.

Ik denk dat het hiermee wel duidelijk is waarom Avidan & Shamir in de eerste instantie voor Dynamisch Programmeren hebben gekozen om een seam te vinden in de 2D afbeeldingen. Het probleem is echter dat dit algoritme niet werkt op een 3D matrix. Voor video hebben ze daarom wel moeten kiezen voor het flow network algoritme. Dit bespreek ik in de volgende sectie.

Video

In [RubinStein, Avidan, Shamir, 2008] wordt besproken hoe met het flow network algoritme een 2D seam in een 3D matrix wordt gevonden. Aangezien ik dit algoritme al voor een 2D matrix had geimplementeerd wilde ik eerst kijken wat de resultaten zijn van elk frame apart nemen en daar een 'standaard' 1D seam in vinden. Deze resultaten bespreek ik eerst, daarna bespreek ik het algoritme dat op een 3D matrix werkt.

Seam berekenen per frame

In de sectie over seams werd beschreven waarom het nodig was een aaneengesloten pad te vinden in plaats van op elke rij de pixels met de minste energie te verwijderen. Dit zorgde voor een karteleffect en een soortgelijk effect is te zien wanneer de seams per frame worden berekend zonder met de omliggende frames rekening te houden. In Figuur 9 staan twee verschillende frames van een video, voor en na het verwijderen van 10 seams. Tussen deze frames zitten nog 4 andere frames, waarbij bij elk frame het plaatje een stukje naar links gaat. Te zien is dat in de bovenste frame alle seams links van het oog zijn gevonden. In de onderste frame zijn de frames allemaal tussen de ogen in weggehaald. Bij het afspelen van de video[2] zorgt dit voor een schokkerig beeld. Dit was duidelijk zichtbaar op een aantal video's waar een aantal seams zijn verwijderd. RubinStein, Avidan en Shamir hebben een video gemaakt waar deze vervormingen goed te zien zijn[2].

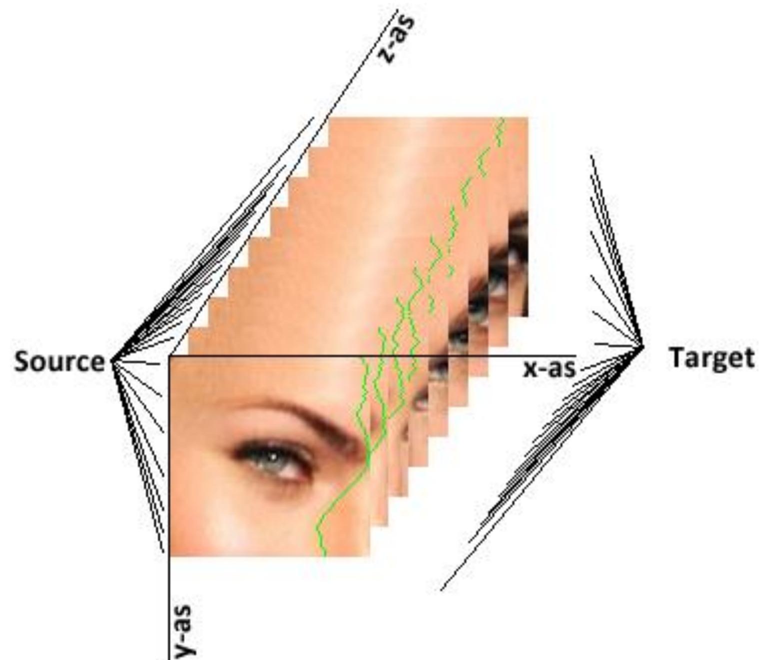
Figuur 9: Twee frames binnen één video.

[2] www.youtube.com/watch?v=AJtE8afwJEg

Duidelijk is dat deze methode voor soortgelijke problemen zorgt als bij de seams waar de pixels met de minste energie in elke rij worden verwijderd. Ook in de 3D set is het nodig om rekening te houden met de omliggende frames.

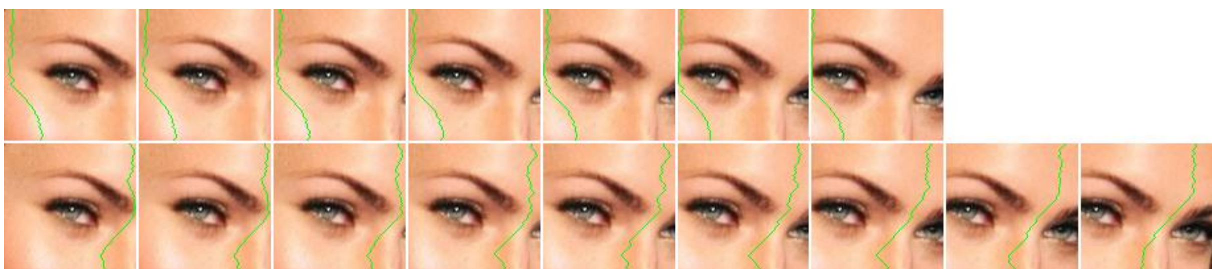
Seam berekenen over de gehele 3D set

De stap van een 2D naar 3D flow network algoritme is relatief simpel. Er moeten nodes aangemaakt worden voor elke pixel in elk plaatje in *elk frame*. Het aantal nodes stijgt dus aanzienlijk. De edges die aangemaakt moet worden zelfde zijn dezelfde als voor de 2D matrix om de monotoniciteit en de connectiviteit te behouden [RubinStein, Avidan, Shamir 2008]. Hier komen nog wel een aantal edges bij. Exact dezelfde edges als op de x-as komen op de tijd-as. Dus een edge van een pixel naar dezelfde pixel op het volgende frame met dezelfde (x, y) locatie heeft als capaciteit de energiewaarde van de pixel waar de edge vandaan komt. Er gaan ook drie edges naar het vorige frame op dezelfde (x, y) locatie, op (x, y – 1) en (x, y + 1) met als capaciteit oneindig. Zie Figuur 7 voor een grafische weergave. Ook vanaf de virtuele SOURCE en naar de virtuele TARGET zijn edges met capaciteit oneindig nodig. De SOURCE heeft edges naar de alle pixels van de meest linker rijen van elk frame. De pixels van de meest rechter rijen hebben allemaal een edge naar de TARGET.



Figuur 10: Een 3D overzicht van een filmpje met 9 frames.

Omdat de seams nu ook in de tijd aan elkaar verbonden zijn, is het onoverkomelijk dat een seam in een frame een hogere energiewaarde zal moeten nemen om de energiewaarde van het volledige 2D vlak zo laag mogelijk te houden. Dit is tegelijkertijd ook een goede manier om te experimenteren en te zien hoe het algoritme zich in verschillende situaties gedraagt. In Figuur 11 is zo'n duidelijk verschil te zien. Hier worden twee verschillende video's onder elkaar getoond. De bovenste video heeft 7 frames, de onderste heeft er 9. Het filmje betreft een afbeelding die elk frame een stukje naar links wordt geschoven. De eerste 7 frames zijn voor beide video's exact hetzelfde: in de twee extra frames van de onderste video wordt de afbeelding nog wat extra naar links geschoven, waardoor het oog deels van het beeld verdwijnt. In de bovenste video bevinden de frames zich allemaal in de initieel goedkopere linkerkant van het plaatje: rechts beginnen zou betekenen dat een stuk wenkbrauw ook zou verdwijnen wat veel energie kost. In het laatste frame kan het nog nét en wordt de seam helemaal naar links gedrukt.



Figuur 11: De frames van twee verschillende video's. De eerste 7 frames zijn van beide video's hetzelfde, de twee extra frames in de onderste video zorgen voor compleet andere seams.

In de onderste video begint de seam wél aan de rechterkant en neemt hier een stuk wenkbrauw mee. Als de seam weer aan de linkerkant zou beginnen, zou deze in de laatste twee frames een stuk oog moeten meenemen. Dit kost veel meer energie dan het stuk wenkbrauw wat het anders moet meenemen. Duidelijk is dat de seam in dit video in de eerste frames lijkt te anticiperen wat er later komt en zodoende dus rechts begint.

Snelheid van seams over een 3D set

Met het flow network algoritme toegepast op de 2D afbeeldingen bleek al dat dit een erg traag algoritme is. De reden hiervoor was dat een groot aantal van de nodes bekeken moest worden om een pad te vinden, iets wat relatief vaak herhaald moet worden. Helaas is er hetzelfde probleem in de 3D situatie. Ook hier wordt vaak meer dan 90% van de totale video bekeken om vervolgens 1 pad te vinden. Het betreft hier dan ook veel meer pixels: bovenstaande video van 9 frames groot is 100 pixels groot in zowel de breedte als de hoogte. In totaal zijn dit 100 x 100 x 9 = 90.000 pixels of nodes. Bij een 2D plaatje zijn we deze aantallen ook al tegengekomen (een 500 x 500 plaatje is al 250.000 pixels). Voor bovenstaande video van 9 frames werden ruim 172 miljoen nodes bekeken. Minder dus dan bij een aantal afbeeldingen in Tabel 1, maar onthoudt dat 9 frames voor een video niets voorstelt. Om de beweging van een video zonder schokken te zien zijn 25 frames per seconde nodig. Met 9 frames hebben we dus amper 1/3 seconde. Daarbij wordt er met deze berekeningen ook maar één seam weggehaald. Er is dus nog flink wat optimalisaties nodig om video's van een aantal minuten lang met een redelijke resolutie binnen een afzienbare tijd te verwerken.

BFS vs. DFS

In het flow network algoritme maakte ik constant gebruik van breadth-first search (BFS). In de voorgaande analyses voor de trage werking van dit algoritme leek BFS de hoofdreden van dit probleem te zijn. Om het kortste pad van de SOURCE naar de TARGET te vinden worden veel extra nodes bekeken. Waarom zou het nodig zijn om het per sé het korste pad te zoeken? Hoofddoel is om zo snel mogelijk alle edges op slot te zetten zodat er geen pad meer gevonden kan worden. Reden genoeg om een variant van BFS te gebruiken: depth-first search (DFS) (kijk voor een uitgebreide uitleg van beide zoekalgoritmen in [Cormen et al, 2009]).

Het Edmonds-Karp algoritme dat gebruikt wordt om de minimal cut en hiermee de seam te vinden heeft BFS nodig om gegarandeerd de minimal cut te vinden. Met gebruik van DFS is dit niet meer gegarandeerd [Cormen et al, 2009]. Een test met DFS wijst uit dat het aantal bezochte nodes aanzienlijk minder is dan met gebruik van BFS. Waar voor het bovenstaande video met 9 frames ruim 172 miljoen nodes werken bekeken met BFS, waren dit er ruim 7 miljoen met DFS. Echter, de *connectivity* eis in de tijd wordt nu geschonden: de seams van de frames liggen niet meer maximaal 1 pixel van elkaar af. Verder onderzoek is nodig om uit te zoeken waarom dit gebeurt en of DFS alsnog wel gebruikt kan worden om voor een flinke prestatieverbetering te zorgen.

Conclusie

In dit paper heb ik meerdere technieken gebruikt om zowel in 2D afbeeldingen als in 3D video's *seams* te vinden en deze te gebruiken om beide media in de hoogte en breedte te verkleinen met behoud van de belangrijkste inhoud. Een aantal al bekende technieken zoals kijken naar de energie die juist ontstaat bij het verwijderen van een seam (*Forward Energy*, zie [RubinStein, Avidan, Shamir, 2008]) heb ik helaas niet geïmplementeerd. Deze en andere soort technieken zoals DFS voor de flow

network zijn interessante technieken om nog verder te implementeren en zo een beter resultaat te krijgen.

Duidelijk is geworden dat de techniek helaas nog net niet ver genoeg is om in grote schaal ingezet te worden door bijvoorbeeld een kleinere versie van een plaatje weer te geven op een kleiner beeldscherm zoals die van een smartphone. Het is niet te voorkomen dat er in een plaatje waar alleen maar belangrijke informatie zit deze informatie vervormd of verdwijnt. De keuze zal eerst gemaakt moeten worden om dit als een nodige tekortkoming te zien en toch in te zetten, of een manier te vinden om deze afbeeldingen te identificeren en deze dan op de normale manier te weergeven. Verder onderzoek van onder andere Avidan en Shamir zal hier ongetwijfeld meer duidelijk over scheppen.

Met technieken als Forward Energy en Face Recognition werkt het algoritme alleen maar beter en verder onderzoek zal er toe leiden dat de resultaten zowel beter als sneller berekent kunnen worden.

Referenties

S. Avidan, A. Shamir, “Seam carving for content aware image resizing”, ACM Trans. Graph. vol. 26, no. 3, 2007

L. Chen, X. Xie, X. Fan, W. Ma, H, Zhang and H. Zhou, “A visual attention model for adapting images on small displays”, Multimedia Systems 9, 4, 353–364, 2003

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, “Introduction to Algorithms, third edition”, MIT Press / McGraw Hill, 2009

N. Dalal and B. Triggs, “Histograms of Oriented Gradients for Human Detection”. In CVPR, pages 886-893, 2005

Y. Hirano et al., "Industry and Object Recognition: Applications, Applied Research and Challenges", J. Ponce et al. (Eds): Toward Category-Level Object Recognition, LNCS 4170, pp. 49-64, 2006

A. Rosenfeld, M. Thurston, “Edge and curve detection for visual scene analysis”, IEEE Trans. Comput, C-20, 562-569, 1981

M. Rubenstein, S. Avidan, A. Shamir, “Improved seam carving for video retargeting”, ACM Trans. Graph, vol. 27, no. 3, 2008

A. Santella, M. Agrawala, D. Decarlo, D. Salesin and M.Cohen, “Gaze-based interaction for semiautomatic photo cropping”, ACM Human Factors in Computing Systems (CHI), 771–780, 2006

B. Suh, H. Ling, B. B. Bederson and D. W. Jacobs, “Automatic thumbnail cropping and its effectiveness”, UIST ’03: Proceedings of the 16th annual ACM symposium on User interface software and technology, ACM Press, New York, NY, USA, 95–104, 2003

C. Tao, J. Jia and H. Sun, “Active window oriented dynamic video retargeting”, Proceedings of the Workshop on Dynamical Vision”, ICCV 2007

P. Viola and M. Jones, “Rapid object detection using a boosted cascade of simple features”, Conference on Computer Vision and Pattern Recognition (CVPR), 2001