Implementatieplan Edge Detection

Zehna van den Berg 1662506
Remco Nijkamp 1657833

Datum 11-5-2016

Inhoud

1. Doel

Edge detection is een onderdeel van de pre-processing fase. Het zorgt ervoor dat het herkennen van features in een afbeelding veel gemakkelijker gaat. Het doel is dus het maken van een afbeelding met daarin de edges van een foto. Vanuit hier kan dan, na het uitvoeren van een threshold, gezocht worden naar speciale features binnen de foto, zoals de contouren van een gezicht.
Een subdoel is het zoeken naar de meest geschikte kernel voor het uitvoeren van de edge detection.

2. Methoden

Nodig voor edge detection zijn neighborhood operations. Deze kijken niet alleen naar de pixels individueel maar juist ook naar de omringende pixels. Waarom dit nodig is voor edge detection is vrij simpel te verklaren en eigenlijk heel logisch. Edges zijn eigenlijk niks anders dan sterke overgangen in intensiteit (of grijswaarde) tussen de verschillende pixels die aan elkaar grenzen. De uitleg geeft al aan dat hierbij dus gekeken moet worden naar een aantal pixels om de conclusie te kunnen trekken dat hier sprake is van een edge.

Bij het kijken naar omringende pixels wordt dus gesproken van een neighborhood. Deze neighborhood is ook wel bekend als de kernel of de mask die over een afbeelding heen wordt gegooid.

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

De meest voorkomende kernel is een 3x3 neighborhood. Maar er komen ook kernels voor van bv 5x5 of 9x9 pixels. Ook hoeft een kernel niet altijd vierkant te zijn. Andere veel voorkomende vormen zijn bv rond en “plusvormig”. Bij de kernel wordt de nieuwe waarde van de middelste pixel bepaald door de oude waarde van zichzelf en de 8 omringende oude pixelwaarden. Bij het berekenen van de nieuwe pixelwaarde kunnen alle pixels uit de kernel even zwaar worden meegerekend, maar dit hoeft niet het geval te zijn.

Voor de kernels wordt een grove scheiding gemaakt tussen low pass filters en high pass filters. Deze laatste kan worden gebruikt om scherpe overgangen in een afbeelding naar voren te laten komen en dus is deze ook geschikt voor edge detection.
Bij high pass filters worden in de kernel zowel positieve als negatieve gewichten meegenomen.

$$h_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}, \quad h_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

3x3 Prewitt kernel

$$h_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad h_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

3x3 Sobel kernel

Opvallend aan deze Prewitt en Sobel kernels is dat de som van de waardes in de kernels telkens gelijk is aan nul.

2.1. Mogelijkheden

2.1.1. Prewitt

De Prewitt methode maakt gebruik van een 3x3 kernel met gelijk gewicht aan beide zijden waarbij de ene kant het negatieve is van de positieve kant. Hierbij wegen de waardes van de buren dus even zwaar mee als de as waarop gekeken wordt. Deze methode maakt gebruik van aparte x- en y-kernels. Dit zorgt ervoor dat het algoritme sneller is maar diagonale edges minder goed herkent.

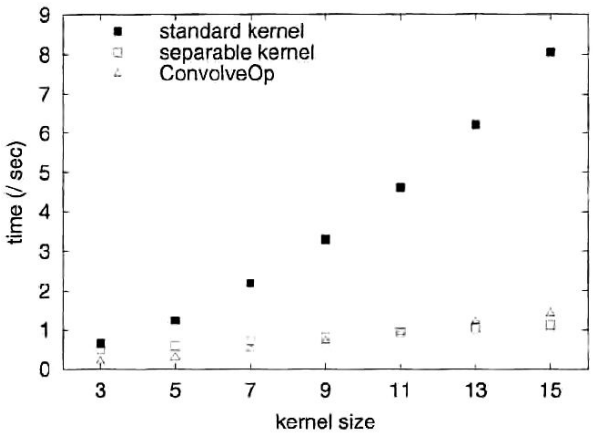
2.1.2. Sobel

Sobel kernels werken op hetzelfde principe als de Prewitt methode maar verschilt op een punt. Namelijk dat de waarde van de pixels op de as waar op dat moment gekeken wordt een hoger gewicht heeft dan zijn buren.

2.1.3. Laplacian

De Laplacian maakt gebruik van 2de orde afgeleide waardoor je dus het verschil meet tussen het ontstaan van edges. Wanneer het resultaat dat uit de 2de afgeleide komt het nulpunt doorsnijd tussen 2 pieken betekent het dat daar een edge is. De Laplaciaan is echter zeer sensitief tegen ruis. Hierdoor wordt er meestal eerst een Gaussian filter gebruikt om de ruis te verminderen zodat er minder ongelijkheden uitkomen.

2.1.4. Formaat Kernels



Kernels kunnen in verschillende maten genomen worden. Ze zijn altijd rechthoekig maar kunnen eventueel ook een convolutie uitvoeren in een plus vorm. Dit wordt gedaan door de hoeken van het rechthoek op 0 te zetten en zo dus niet te laten meetellen.
De afmetingen in de x en y richting beïnvloeden de edge detection ieder op een andere manier.
Als we stellen dat we een edge willen detecteren op de x-as dan zullen de assen op de volgende manieren gedragen. Een grotere kernel in de x richting zal de edge detection smoothen met een minder scherpe edge overgang. Een grotere lengte in de y-richting zal het verlengde van de lijn meer meetellen waardoor korte lijnen minder gedetecteerd worden.

2.2. Gekozen methoden

We hebben ervoor gekozen de volgende methoden te gaan onderzoeken. Het gaat ons bij het onderzoeken vooral naar het resultaat van het plaatje dat we over zullen houden na de edge detection. Hoe beter de belangrijke edges zichtbaar zijn en de rest niet, hoe beter het resultaat. Verder zal een timer worden meegenomen om te kunnen kijken hoe lang een bewerking met een kernel duurt. De tijd kan uiteindelijk een beslissende factor worden als het verschil tussen de verschillende kernels niet duidelijk naar voren komt.

We willen de Prewitt uitzetten tegen de Sobel. Dit doen we verder nog voor verschillende formaten van beide kernels. Dus voor het onderzoek gaan we gebruiken:

- Prewitt in 3x3, 5x5 en 9x9
- Sobel in 3x3, 5x5 en 9x9

Er is gekozen om voor nu niet de Laplacian mee te nemen, aangezien deze in de standaard software al gebruikt werd. Daar werd een 9x9 Laplacian kernel gebruikt.

3. Keuze

De grootste keuze die we hebben gemaakt ligt hem vooral in de manier waarop we de Kernel klasse hebben gemaakt.
De kernel is uitgewerkt in de klasse Kernel.
Er is voor gekozen een algemene Kernel klasse te maken, van waaruit de gebruiker een aantal verschillende soorten Kernels kan maken. De types die we er voor nu in hebben gezet zijn Prewitt, Sobel, Laplacian, Gaussian en gewoon een kernel waarvan alle waardes gelijk zijn aan elkaar. Het idee hierachter is dat zo niet alleen kernels voor edge detection gebruikt kunnen worden, maar voor veel meer functies.

Bij het aanmaken van een kernel, dus bij het aanroepen van de constructor, moet het formaat van de kernel in x en y richting worden meegegeven. Hier de width en height van de kernel genoemd. Een eis die we aan de kernel hebben gesteld, is dat hij een middelste waarde moet hebben. Deze wordt bij het verwerken van de kernel over de afbeelding, op de pixel geplaatst die een nieuwe waarde moet krijgen. De eis die hiervoor geldt is dus over de opgegeven waarden oneven zijn. Is dit niet het geval, dan kan er geen middelste waarde worden gekozen en is de kernel dus niet geschikt voor gebruik binnen onze code. Dit wordt dan ook gecheckt binnen de constructor.

Er is voor gekozen om de kernels zo te maken dat ze aan te passen zijn naar verschillende formaten. Zo zijn de Prewitt en de Sobel kernels standaard 3x3, maar wij hebben er een nieuwe interpretatie op bedacht zodat ook andere formaten mogelijk zijn, zoals 5x5 of 7x7. Maar zelfs tot vierkante kernels gaat onze implementatie zich niet beperken. Over deze implementatie verderop meer. De breedte en hoogte van de kernel kunnen los van elkaar op worden gegeven. Dus ook kernels van 3x5 of 5x9 zulle mogelijk worden.

We zoomen even in op de Prewitt en de Sobel kernels, aangezien we deze voor de testen willen gaan gebruiken. Hier zal ook onze eigen interpretatie even worden uitgelegd.

De Prewitt en de Sobel kernel zijn beiden kernels die uit elkaar getrokken kunnen worden tot een horizontaal en een verticaal deel. In de implementatie hebben we de horizontale kernels gemaakt. We hebben ervoor gekozen om niet de kernel een kwart slag te draaien voor het uitvoeren van de verticale edge detection, maar om de volgorde van de pixels die worden doorlopen om te draaien. Dit komt eigenlijk op hetzelfde neer.

3.1 Prewitt opbouw

Zoals al eerder aangegeven is een originele Prewitt kernel een kernel van 3x3. Het bestaat uit een horizontaal en een verticaal deel die om beurten worden uitgevoerd. De twee afbeeldingen die dit oplevert worden vervolgens samengevoegd. Aangezien wij in onze implementatie niet te waarden in de kernel “draaien” maar de volgorde van de pixels van de afbeeldingen andere doorlopen, hebben wij voor de Prewitt enkel een horizontale implementatie gemaakt.

Het horizontale deel van de originele Prewitt kernel ziet er als volgt uit:

-1	-1	-1
0	0	0
1	1	1

3x3 Prewitt horizontaal

Wat opvalt is dat de middelste rij 0 is en de de rij daarboven precies het negatieve is van de rij onder de middelste rij. Ook zijn alle waarden op die rijen gelijk aan elkaar. Zo onstond het idee om voor de nieuwe implementatie dit zo te houden voor uitbreiding in de breedte, maar voor uitbreiding in de hoogte gewoon de rij met -1 of 1 waarden dubbel voor te laten komen. Zou zou een 5x5 kernel er zo uit komen te zien:

-1	-1	-1	-1	-1
-1	-1	-1	-1	-1
0	0	0	0	0
1	1	1	1	1
1	1	1	1	1

5x5 Prewitt horizontaal

3.2. Sobel opbouw

Ook de Sobel kernel heeft eigenlijk een vaste grootte van 3x3 en ziet er als volgt uit:

-1	-2	-1
0	0	0
1	2	1

3x3 Sobel horizontaal

Ook hier hebben we enkel het horizontale deel van de kernel gemaakt, omdat we dus niet de kernel “draaien”, maar de volgorde van de pixels van de afbeelding als het ware omdraaien.
Om ook voor de Sobel andere formaten kernels beschikbaar te maken moest ook hier een nieuwe implementatie komen. Opvallend aan de originele kerel is dat hij eigenlijk net zo is als de Prewitt kernel, alleen geeft hij de pixels dicht bij de middelste waarde net wat meer gewicht mee. Dit bracht ons op het idee om deze rij als volgt te implementeren. De buitenste waarden van de rij worden op 1, dan wel -1 gezet en hoe verder je naar het midden komt, hoe groter de waarde positief dan wel negatief wordt. Als de kernel daarentegen in de hoogte groeit, willen we eigenlijk niet daar ook extra gewicht aan toe gaan voegen, omdat deze zegmaar steeds verder van de “lijn” afraken die we proberen te detecteren. Daarom is ervoor gekozen deze rijden net als bij de Prewitt op -1, dan wel 1 te zetten.

Een 5x5 Sobel kernel volgens onze implementatie ziet er zo uit:

-1	-1	-1	-1	-1
-1	-2	-3	-2	-1
0	0	0	0	0
1	2	3	2	1
1	1	1	1	1

5x5 Sobel horizontaal

4. Implementatie

Voor de implementatie zijn twee delen van belang. Ten eerste moest een klasse worden gemaakt die zorgt voor de opslag van de kernel. Ten tweede natuurlijk een klasse met methoden voor het toepassen van de kernel op de afbeelding.

4.1. Kernel

Hieronder zullen een aantal methodes binnen de Kernel klasse worden besproken.

4.1.1. Kernel Constructor

```
Kernel::Kernel(int w, int h, int mode, int strength) {
        width = w;
        height = h;
        size = width * height;
        if (width % 2 && height % 2) { fillKernel(mode, strength);}
}
```

De constructor maakt een Kernel aan. Hij slaat eerst zijn width (breedte) en height (hoogte) op en berekent met deze waarden de size (grootte). Daarna roept hij de functie fillKernel aan, waar, aan de hand van mode die bij het aanmaken van de kernel is meegegeven, de kernel juiste methode wordt aangeroepen die bij de juiste soort kernel hoort. Strength kan worden meegegeven om de sterkte van de waardes in de kernel te kunnen bepalen. Standaard is de sterkte 1.

4.1.2. fillKernel

```
void Kernel::fillKernel(int mode, int strength) {
        data = new int[size];
        switch (mode) {
                case LAPLACIAN:  fillLaplacianKernel(strength);  break;
                case PREWITT:    fillPrewittKernel(strength);    break;
                case SOBEL:      fillSobelKernel(strength);      break;
                case GAUSSIAN:   fillGaussianKernel(strength);   break;
                default:         fillNormalKernel(strength);     break;
        }
}
```

Deze methode maakt een int array aan waarin de waardes van de kernel opgeslagen zullen worden. Verder zit er een switch statement in om de methodes voor het vullen van deze kernel met de juiste waarden aan te roepen.

4.1.3. Prewitt

```
void Kernel::fillPrewittKernel(int strength){
	...
	for (int i = 0; i < size; ++i) {
		if (i < beginMidden) { data[i] = -strength;}
		else if (i >= eindMidden) { data[i] = strength;}
		else data[i] = 0;
	}
}
```

In deze methode wordt de array van de kernel gevuld met waardes die overeenkomen met een Prewitt kernel. Op het deel van de puntjes worden een aantal int waardes die binnen de methode van belang zijn aangemaakt. Verder wordt de kernel doorlopen en worden alle waarden boven de middelste rij op -1 gezet, de middelste rij op 0 en de waarden onder de middelste rij op 1. Wij hebben ervoor gekozen om alle kernels een custom size te kunnen geven, dus ook de Prewitt. In hoofdstuk 3 is te zien hoe de kernel zich gedraagt als hij groter wordt gemaakt dan 3x3.

4.1.4. Sobel

```
void Kernel::fillSobelKernel(int strength){
	...
	int tPos = 0;
	int tNeg = 0;
	for (int i = 0; i < size; ++i) {
		if (i < voorMiddenRij) { data[i] = -1; }
		else if (i < beginMidden) {
			if (i < (voorMiddenRij + ((width / 2) + 1))) { tNeg--; }
			else tNeg++;
			data[i] = tNeg;
		}
		else if (i < eindMidden) { data[i] = 0;}
		else if (i < eindNaMiddenRij) {
			if (i < (eindMidden + ((width / 2) + 1))) { tPos++;}
			else tPos--;
			data[i] = tPos;
		}
		else data[i] = 1;
	}
}
```

In deze methode wordt de array van de kernel gevuld met waardes die overeenkomen met een Sobel kernel. Op het deel van de puntjes worden een aantal int waardes die binnen de methode van belang zijn aangemaakt. Het gedrag van de kernel zoals besproken in hoofdstuk 3 is hier duidelijk te zien. De kernel wordt als het ware opgedeeld in 5 stukken; de middelste rij, de rij direct voor de middelste rij, de rij(en) nog voor deze rij (deze zijn niet altijd aanwezig!), de rij direct na de middelste rij en de rij(en) daar weer na (ook deze zijn niet altijd aanwezig!).

4.2. Edge detection methode

Voor de Edge detection hebben we het proces opgedeeld in 3 methodes om het overzichtelijker te houden en makkelijk kleine aanpassingen te maken in herhaalde stukken code.

4.2.1 stepEdgeDetection

Deze methode zit standaard al in de header file van het aangeleverde project. Hier worden de objecten gemaakt die voor de resterende berekeningen nodig zijn.

```
IntensityImage * StudentPreProcessing::stepEdgeDetection(const IntensityImage &image) const {
	//time calculation
	clock_t time = clock();

	IntensityImageStudent nImage(image.getWidth(), image.getHeight());
	Kernel kern(5, 5,PREWITT, 1);
	KernelAppliance(nImage, image, kern);

	time = clock() - time;
	std::cout << "The amount of time spent calculating: " << time << " milliseconds \n";
	return new IntensityImageStudent(nImage);
}
```

Hier wordt ook de tijd opgemeten zodat we kunnen bekijken hoelang onze berekeningen erover doen. Naast het aangeleverde plaatje wordt de kernel gemaakt die erover heen gehaald wordt en een nieuw plaatje gemaakt om alle nieuwe waardes in op te slaan.

4.2.2 KernelAppliance

```
IntensityImage & StudentPreProcessing::KernelAppliance(IntensityImage& newImage, const IntensityImage & oldImage, const
Kernel & kern) const{
	int kernVerOff = kern.getHeight() / 2;
	int kernHorOff = kern.getWidth() / 2;

	for (int y = 0; y < oldImage.getHeight(); y++){
		for (int x = 0; x < oldImage.getWidth(); x++){
			double result = 0;

			if (y >= kernVerOff && y < oldImage.getHeight() - kernVerOff){
				if (x >= kernHorOff && x < oldImage.getWidth() - kernHorOff){
					//Checks the kernel in all directions
					result += kernelSum(kernHorOff, kernVerOff, oldImage, kern, x, y, true,
true);
					result += kernelSum(kernHorOff, kernVerOff, oldImage, kern, x, y, true,
false);
				}
			}
			int nPixel = (int)round(result);
			if (nPixel < 0){ nPixel = 0; }
			if (nPixel > 255){ nPixel = 255; }
			newImage.setPixel(x, y, nPixel);
		}
	}
	return newImage;
}
```

Allereerst wordt de grote van de kernel bepaald om te berekenen hoeveel offset die heeft. Hiermee kan bepaald worden vanaf welke pixels de kernel over het plaatje gedaan kan worden. Want als het buiten het plaatje zou vallen dan kunnen die waardes dus niet meegenomen worden. Als de kernel niet gebruikt kan worden dan wordt het resultaat op 0 gehouden en dus met zwart aangevuld. Voor de pixels die wel gecheckt kunnen worden wordt het masker er op 4 verschillende manieren opgelegd om vanaf alle kanten te kunnen bekijken.
Voor het berekenen van de kernelwaarde wordt de kernelSum methode aangeroepen.

4.2.3 kernelSum

```
double StudentPreProcessing::kernelSum(int kernHorOff, int kernVerOff, const IntensityImage & oldImage, const Kernel & kern, int x, int y, bool xAdd, bool yAdd) const{
	double result = 0;
	int value = 0;
	for (int yWeight = -1 * kernVerOff; yWeight <= kernVerOff; yWeight++){
		for (int xWeight = -1 * kernHorOff; xWeight <= kernHorOff; xWeight++){
			double temp = kern.getValue(value) * ((int)oldImage.getPixel((xAdd ? x + xWeight : x - xWeight), (yAdd ? y + yWeight : y - yWeight)));
			result += temp;
			value++;
		}
	}
	//Rotated
	value = 0;
	for (int yWeight = -1 * kernVerOff; yWeight <= kernVerOff; yWeight++){
		for (int xWeight = -1 * kernHorOff; xWeight <= kernHorOff; xWeight++){
			double temp = kern.getValue(value) * ((int)oldImage.getPixel((xAdd ? x + yWeight : x - yWeight), (yAdd ? y + xWeight : y - xWeight)));
			result += temp;
			value++;
		}
	}
	if (result < 0){ result = result * -1; }
	result = result * 1; //Change this to change the output of each kernel. 1 for 3x3, 0.4 for 5x5 recommended. This is solely for the Threshold.
	return result;
}
```

Deze methode bevat veel parameters die al aangemaakt zijn in de vorige 2 methodes. Hij telt het resultaat van de standaard kernel en de geroteerde kernel bij elkaar op. Vervolgens zijn er nog twee booleans waarmee de kernel nog geflipt kan worden. Hiermee wordt er vanaf meerdere hoeken bekeken hoe het verschil zich gedraagt en kan dus beter het verloop van de edge bepaald worden.

5. Evaluatie

5.1. Onderzoeksvraag

Welke kernel geeft een “beter” resultaat voor edge detection, Prewitt of Sobel?
Hierbij zullen verschillende subvragen gesteld worden, zoals:

- *Welke kernel produceert het snelst een afbeelding?*
- *Welke kernel produceert de afbeelding die voor menselijk oog het duidelijkst edges aangeeft?*
- *Welke kernel produceert de afbeelding die het best voor vervolgstappen geschikt is?*
- *Welk formaat kernel geeft het “beste” resultaat?*

Het woord “beter” is een beetje een vaag begrip. Er wordt hier bedoeld dat de afbeelding die ontstaat het duidelijkst de juiste edges weergeeft en zoveel mogelijk onzin weglaat. Hierbij moet de opgeleverde afbeelding dus weer bruikbaar zijn voor de volgende stappen binnen de software die al beschikbaar was voor het vinden van features binnen de afbeelding. Ook wordt hierbij gekeken hoe snel de kernel uit te voeren is.

5.2. Experimenten

We willen gaan testen welke kernel beter werkt voor het vinden van edges, en welk formaat daar dan het best geschikt voor is. Dit willen we gaan doen door een aantal keer een edge detection uit te voeren, voor verschillende formaten van zowel de Prewitt als de Sobel kernel. Ook zullen we hiervoor een aantal verschillende afbeeldingen gebruiken, omdat het ook toeval kan zijn dat een kernel precies bij die afbeelding goed resultaat levert. Achteraf willen we eerst een conclusie trekken met de tussenkomst van een mens. Dus wij gaan zelf bekijken op welk plaatje we het beste de edges terug kunnen vinden. Verder willen we het plaatje dat de edge detection heeft opgeleverd doorgeven aan de volgende stap in de keten, om te kijken of de applicatie hem ook goed kan gebruiken bij het maken van een treshold en het vinden van features in de afbeelding. Tot slot wordt het tijdsaspect meegenomen. Als een kernel heel goed resultaat oplevert, maar daarentegen heel langzaam is ten opzichte van een andere kernel, kan het zijn dat alsnog de kernel die iets minder resultaat levert, maar wel heel snel is uiteindelijk het meest geschikt wordt gevonden.

5.3. Verwachting

De verwachte uitkomst is dat de Sobel een beter resultaat op zal gaan leveren dan de Prewitt. Dit aangezien in de Sobel iets meer gewicht uitdeelt aan pixels die zich dicht rond de kern bevinden. Hierdoor zou een overgang van een kleiner omvang ook al gedetecteerd moeten worden en duidelijkere lijnen te weergeven. De verwachting is dat het verschil in snelheid te verwaarlozen valt.