Implementatieplan

Imageshell en grayscale

Gerrit van Os & Bryan Campagne

Inhoud

1. Doel

Het doel van deze opdracht bestaat uit een aantal losse onderdelen, er zullen 2 imageshells gemaakt worden. Eén van deze zal zijn voor een RGB-image de andere voor een Intensity image. We streven ernaar om een snellere imageshell te produceren dan op dit moment aanwezig is in de standaard implementatie. Aangezien we niet kunnen zien wat de huidige implementatie is weten we pas na het testen of dit ook daadwerkelijk gelukt is.

Daarnaast zullen we een RGB → Intensity conversie algoritme gaan implementeren. Het doel van dit algoritme is dat deze sneller is dan de standaard implementatie en ook evengoed de facial parameters herkent.

2. methoden

2.1 Imageshell

De belangrijkste keuzes die hier mogelijk zijn hebben te maken met de manier van de pixels opslaan. De keuzes hierin zijn uiteraard groot, denk aan een c stijl array of 1 van de STL-containers zoals std::array<> of std:vector<>. Daarnaast is het nog de vraag hoe de pixels opgeslagen gaan worden. Alle pixels achter elkaar in 1 container of een hoofd container met daarin voor elke rij of kolom een ander container die daadwerkelijk de informatie bevat. De verschillende opties hebben ook voor en nadelen uiteraard. Het gebruik van STL-container is vrij eenvoudig gezien de functionaliteit die deze ondersteunen, echter performance technisch zijn deze wat slechter omdat deze veel overhead hebben.

2.2 Grayscale

Om van een kleurenfoto naar een zwart-wit foto te gaan moet een algoritme worden gebruikt. Hieronder zijn de meest voorkomende methodes beschreven.

2.2.1 Intensity

Intensity is een veelvoorkomende methode. Deze methode is relatief eenvoudig. De drie RGB-waardes worden bij elkaar opgeteld en vervolgens gedeeld door drie. Het gemiddelde van de 3 waardes wordt de nieuwe grijswaarde van een pixel. (GRAY = ((R+G+B) / 3)) Het nadeel van deze manier is dat het slecht de grijstinten representeert ten opzichte van hoe een mens licht ervaart.

2.2.2 Luma

Luma is gebaseerd hoe een mens grijstinten ervaart. Voor deze berekening hebben de waardes R,G en B samen een waarde 1. Waarvan rood 0.2126 representeert, groen 0.7152 en blauw 0.0722. De waarde van elke kleur wordt met zijn coëfficiënt vermenigvuldigd. Daarna worden de waardes bij elkaar opgeteld en dat is de nieuwe grijswaarde van een pixel. (Gray = (R * 0.2126 + G * 0.7152 + B * 0.0722)). Het nadeel is dat niet iedereen zijn ogen uniform zijn en er zijn betere manieren voor machine vision.

2.2.3 Luminance

Luminance lijkt op luma maar de waardes zijn anders verdeeld. Luminance is gericht op machine vision en niet op het menselijk oog. Voor deze berekening hebben de waardes R, G en B samen een waarde 1. Waarvan rood 0.3 representeert, groen 0.59 en blauw 0.11. De waarde van elke kleur wordt met zijn coëfficiënt vermenigvuldigd. Daarna worden de waardes bij elkaar opgeteld en dat is de nieuwe grijswaarde van een pixel.

(Gray = (R * 0.3 + G * 0.59 + B * 0.11)) Het nadeel is dat het minder natuurlijk is voor het menselijk oog.

2.2.4 Value method

Bij value methode wordt de hoogste van de drie RGB-waardes wordt de nieuwe grijswaarden van een pixel. (Gray = max(RGB)) Deze methode wordt het meest gebruikt voor artistieke effecten gebruikt. Voordeel is dat het snel is. Het nadeel is dat de details in de andere kleuren dan de max value kunnen wegvallen en is gevoelig voor hoge helderheden.

2.2.5 Luster

Luster is een methode waar de hoogste en laagste van de drie RGB-waardes worden bij elkaar worden opgeteld. Vervolgens bij gedeeld door 2. De uitkomst hiervan wordt de nieuwe grijswaarden van een pixel. (Gray= (max(RGB) + min(RGB))/ 2) Het voordeel is dat het minder gevoelig is voor hogere helderhedenen is ontworpen voor computer graphics. Het nadeel is minder natuurlijk voor het menselijk oog.

2.2.6 Single color channel

Single color is een methode waar 1 kleur van de RGB-waardes wordt gekozen en vervolgens wordt dat dan die nieuwe grijswaarde van een pixel. (Gray= (R)) Deze methode heeft de minste operaties maar is moeilijk te verwerken door machine vision en mist de details van de andere kleuren.

3. Keuze

3.1 imageshell

We hebben besloten om te kiezen voor een 1d std::vector<> het voordeel hiervan is dat het snel te implementeren is en dat alle functies relatief simpel zijn. Denk hierbij aan de get en set pixel functies deze zijn heel makkelijk te implementeren op deze manier. Tevens is het een random acces container en kan je dus heel snel waardes opvragen. Echter moet voor de get(x,y) en set(x,y) eerst een berekening gedaan worden om de x,y value om te rekenen naar een i value die benodigd is om de waardes in de array op te vragen.

3.2 Grayscale

We gaan kiezen voor Luminance omdat het menselijk oog het ook zo ziet. Wij denken dat gezichtsherkenning gedeelte sneller gaat werken als de grayscaling zo realistisch mogelijk is. De berekening per pixel zou ook sneller moeten zijn omdat de berekening relatief eenvoudig is omdat de coëfficiënt lineair is.

4. Implementatie

4.1 Imageshell

Beide imageshells(intensity en RGB) lijken qua implementatie vrij veel op elkaar, het grote verschil is de inhoud van de pixelStorage. Bij de RGB zal er een RGB-pixel waarde in de vector staat, deze RGB-klasse is gegeven binnen het framework. Bij de intensity shell zal de inhoud alleen een intensity waarde zijn, deze is eveneens gegeven.

Respectievelijk zullen de file's RGBImageStudent.cpp en IntensityImageStudent.cpp ingevuld worden met de implementatie van de volgende functies:

- `set(const int width, const int height)`
- `set(const IntensityImageStudent &other)`
- `setPixel(int x, int y, Intensity pixel)`
- `setPixel(int i, Intensity pixel)`
- `getPixel(int x, int y) const`
- `getPixel(int i) const`
- `verschillende constructors`

Tevens zal er in de file's RGBImageStudent.h en IntensityImageStudent.h een private variabele toegevoegd worden met de naam pixelStorage. Dit zal zoals eerder aangegeven een std::vector<> betreffen.

Voor de RGB image zal deze er als volgt uit zien "std::vector<RGB> pixelStorage"
Voor de intensity image zal deze er als volgt uit zien "std::vector<Intensity> pixelStorage"

4.2 Grayscale

Het grayscale algoritme zal geïmplementeerd in de file StudentPreProcessing.cpp, In deze file bevind zich de volgende functie `stepToIntensityImage(const RGBImage &image) const` in deze functie zal het gekozen grayscale algoritme uitgevoerd worden. Deze bestaat uit een for-loop die alle pixels langsloopt en de RGB-waarde omrekent naar een Intensity waarde en deze vervolgens opslaat in een nieuwe intensity image. Deze image wordt aangemaakt voordat de for-loop gestart wordt.

4.3 Werkinschatting

Het schrijven van de imageshells duurt waarschijnlijk niet heel lang omdat deze praktisch hetzelfde zijn en je dus in werkelijkheid maar 1 imageshell hoeft te schrijven. Het maken hiervan schatten we in op ca. 2uur voor beide imageshells. Documenteren en testen niet meegerekend.

Het implementeren van het grayscale algoritme schatten we in op ca. 1 uur code tijd wederom testen en documenteren niet meegerekend. Dit voornamelijk omdat de waardes al gegeven zijn en we het alleen maar om moeten vormen naar werkende code. Het is inprincipe 1 rekensom en een for-loop met wat extra function calls etc. eromheen om het binnen het framework te laten werken.

5. Evaluatie

Beide delen die binnen deze opdracht geïmplementeerd worden moeten uiteraard getest worden om te checken of we het doel bereikt hebben wat in het begin van dit plan gesteld is. Hieronder volgt een korte beschrijving van de tests die gedaan zullen worden voor beide onderdelen.

5.1 Imageshell

De imageshell zal getest worden op snelheid, dit wordt gedaan door de doorlooptijd van het gehele programma te meten. Dit omdat de efficiëntie van de imageshell van toepassing is gedurende de gehele looptijd van het programma. Deze test zal steeds met dezelfde afbeelding herhaald worden en met alle stappen van het proces op de default implementatie. De doorlooptijd wordt gemeten voor beide implementaties, dus de door ons gemaakte imageshell en de default imageshell. Er zullen een aantal samplesizes gebruikt worden om cherrypicking te voorkomen. Deze samplesizes zijn 10,50,100 en 250. Naderhand zal er gekeken worden of er verschillen zijn tussen de beide implementaties.

5.2 Grayscale

5.2.1 Snelheid:

Het grayscale algoritme zal getest worden op snelheid, dit wordt gedaan door dit stukje van het programma te isoleren en de doorlooptijd van dit stukje te timen. Ook hier zal er gebruik gemaakt worden van een aantal samplesizes echter hier gaan we iets verder en zijn de samplesizes 10,100,1000 en 10.000. Omdat het hier maar een kort stukje code betreft is het rendabeler om dit vaker te runnen. Ook voor deze test geld dat zowel de default implementatie als de student implementatie getest zullen worden.

5.2.2 Kwaliteit:

Het grayscale algoritme zal ook op “kwaliteit” getest worden, hierbij wordt er gekeken naar de facial parameters die het programma genereerd. Deze worden naast elkaar gelegd en vergeleken tevens zal er naar de output afbeeldingen gekeken worden en ook deze zullen vergeleken worden. Dit zal gedaan worden met 2 verschillende afbeeldingen en voor elke afbeelding zal het programma 10x gerunt worden om fouten te voorkomen.

5.3 Oplevering:

Alle tests zullen uitgewerkt worden in meetrapporten, er zullen 2 meetrapporten gemaakt worden 1 met betrekking tot snelheid voor zowel de imageshell als het grayscale algoritme. Tevens zal er een meetrapport gemaakt worden met daarin de vergelijking in kwaliteit. Beide meetrapporten zullen de gegevens verwerken en tot een conclusie komen.

5.4 Werkinschatting:

Het testen van de Imageshell verwachten we vrij lang bezig te zijn, het maken van de test code technisch zal meevallen maar het uitvoeren van de tests vereist al snel veel tijd gezien de samplesizes. Het testen van de grayscale zal ongeveer evenveel tijd kosten, deze tests runnen wel sneller maar door de grotere samplesizes duurt het relatief net zo lang. Voor het testen van de snelheid verwachten we ca. 2 uur nodig te hebben en voor het verwerken van de resultaten en het opstellen van het meetrapport een soortgelijke hoeveelheid in totaal dus ca. 4 uur.

Het testen van de kwaliteit vereist eigenlijk geen tot weinig modificatie van de code dus het uitvoeren van de tests kost hooguit 30min verwachten we. Het verwerken van de data kost wel iets meer tijd maar totaal verwachten we ca. 3 uur nodig te hebben voor het kwaliteit deel.