Meetrapport snelheid

Imageshell en grayscale

Gerrit van Os & Bryan Campagne

Inhoud

1. Doel

In dit onderzoek zal er gekeken worden naar de snelheid van de imageshell en het gekozen grayscale algoritme.In dit ondezoek zullen een aantal tests gedaan worden die de door ons geïmplementeerde image shell en grayscale algoritme gaan vergelijken met de default implementatie.

2. Hypothese

We verwachten dat onze implementatie van de image shell en gray scaling sneller zijn dan die van de standaard implementatie. Dit denken wij omdat onze imageshell gebruik maakt van een 1d vector die in theorie vrij snel moeten zijn omdat er maar 1 array acces nodig is. Voor de gray scaling gaan we er ook van uit dat deze sneller is, dit vooral omdat wij een relatief simpele rekensom gebruiken. We kunnen niet inzien wat de default implementatie gebruikt maar we gaan ervan uit dat deze een iets moeilijker algoritme gebruikt.

3. Werkwijze

Er zullen een aantal testen uitgevoerd worden, zowel voor de image shell als de grayscale hieronder zullen deze per onderdeel beschreven worden.

3.1 Algemeen

Voor het timen van de code zal er gebruik gemaakt worden van een std::chrono::high_resolution_clock voorafgaand aan het runnen van de samplesizes zal de now() methode aangeroepen worden om de huidige tijd op te vragen, na het runnen van de for-loop zal nogmaald de now() functie aangeroepen worden het verschil van deze 2 tijden is de verstreken tijd.

Alle tests zijn gerunt vanuit visual studio in debug build mode. En met als input de foto “male-2.png” vanuit testset A.

Alle test zijn uitgevoerd op een HP Pavilion Power Laptop 15-cb0xx met de volgende specificaties:

- **Processor:** i7-7700hq
- **Geheugen:** 16GB DDR4 2400MHz
- **Opslag:** 256gb nvme ssd

3.2 Grayscale

Voor het testen van het grayscale algoritme zal er een vergelijking gemaakt worden met de default implementatie. Er zal getest worden met verschillende sample sizes: 10,100,1000 en 10.000. Om alleen de grayscale functie te testen zal er een for-loop om de PreProcessingStep1 geplaatst worden. Doormiddel van deze for-loop kan het aantal keer uitvoeren veranderd worden. Hieronder het stuk code wat gebruikt is.

```
auto startGrayScale= std::chrono::high_resolution_clock::now();
for (int i = 0; i <= 10; i++) {
        if (!executor->executePreProcessingStep1(false)) {
                std::cout << "Pre-processing step 1 failed!" << std::endl;
                return false;
        }
}
auto finishGrayScale = std::chrono::high_resolution_clock::now();
elapsedGrayScale = finishGrayScale - startGrayScale;
```

In dit stukje code is de i<=10 bepalend voor het aantal samples.

3.3 Imageshell:

Voor het testen van de imageshell gaan we bijna op dezelfde manier te werkt, echter omdat de imageshel gedurende het gehele programma gebruikt wordt zal er hier gekeken worden naar de runtime van het gehele programma in plaats van naar 1 functie. Vanwege de lengte van deze tests zullen de sample sizes iets anders liggen namelijk: 10,50,100 en 250 de manier van de code timen gaat op dezelfde manier alleen nu zal de for-loop de volledige main beslaan en dus het gehele programma timen. Hieronder is te zien hoe we dit in code geïmplementeerd hebben:

```
int main(int argc, char * argv[]) {
        auto startProgram = std::chrono::high_resolution_clock::now();
        //ImageFactory::setImplementation(ImageFactory::DEFAULT);
        ImageFactory::setImplementation(ImageFactory::STUDENT);
        for (int i = 0; i <= 250; i++) {

--------- code van de originele main

}

auto finishProgram = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double> elapsedProgram = finishProgram - startProgram;
        std::cout << "time elapsed total program: " << elapsedProgram.count() << "\n";
        system("pause");
        return 1;

}
```

4. Resultaten

4.1 Grayscale

Hieronder een tabel van de tijden die gemeten zijn tijdens het testen van de verschillende grayscale algoritmes. Bovenin de tabel is de samplesize weergegeven met daaronder de gemeten tijd in seconden.

Grayscale snelheid	**10**	**100**	**1000**	**10000**
Student implementatie	0.239	2.22	12.5	117.5
Default implementatie	0.39	2.9	23.9	235.7
verschil	0.151	0.68	11.4	118.2

Tevens nog een tabel met de gemiddelde tijd per functie deze is uitgedrukt in milliseconde en gebaseerd op de deling van het resultaat en de samplesize.

Grayscale snelheid gem	**10**	**100**	**1000**	**10000**
Student implementatie	23.9	22.2	12.5	11.75
Default implementatie	39	29	23.9	23.57
verschil	15.1	6.8	11.4	11.82

4.2 Imageshell

Voor de imageshell zijn hieronder de resultaten weergegeven in eenzelfde soort tabel eerst de totaaltijd per run in seconden daarna de gemiddelde tijd in milliseconde.

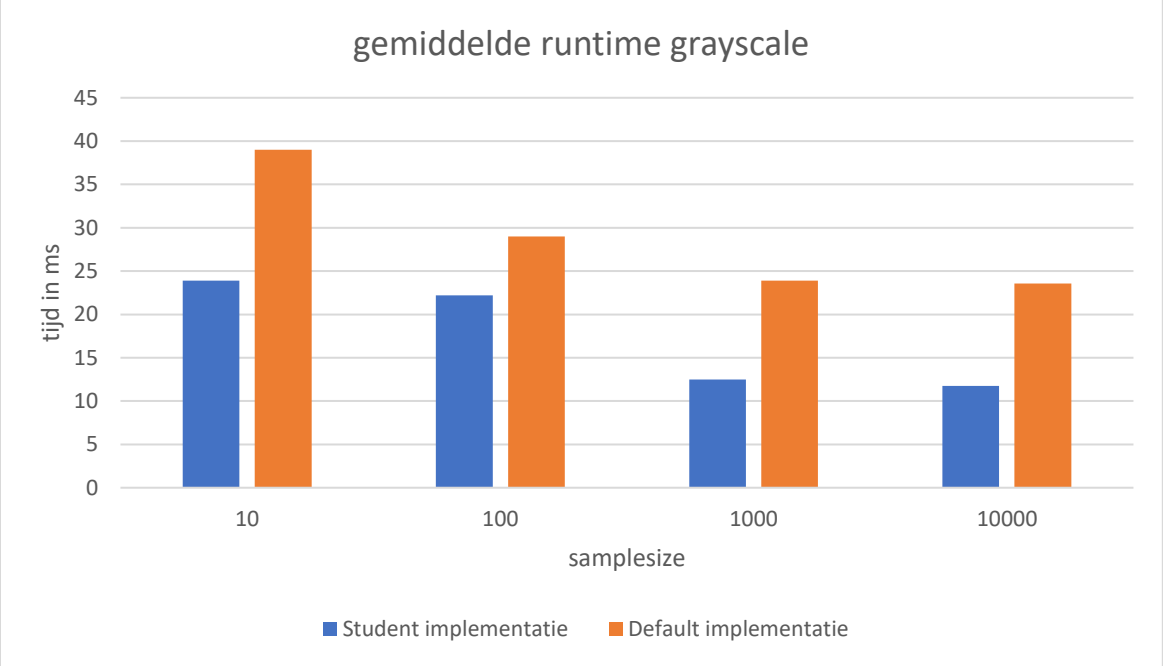
Totale programma	10	50	100	250
Student implementatie	8.6	39.14	76.9	187.62
Default implementatie	4.94	21.7	42.49	NF
verschil	3.66	17.44	34.41	187.62

Totale programma gem	10	50	100	250
Student implementatie	860	782.8	769	750.48
Default implementatie	494	434	424.9	NF
verschil	366	348.8	344.1	750.48

5. Verwerking resultaten

5.1 Grayscale

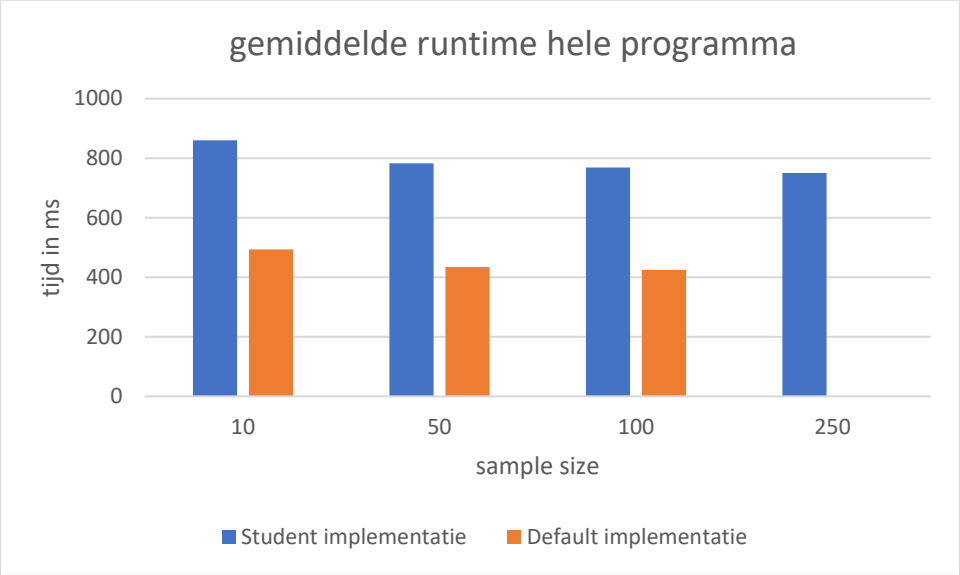
Hieronder hebben we de tabel verwerkt in een grafiek.



In deze grafiek is duidelijk te zien dat de default implementatie altijd trager is dan de door ons geïmplementeerde code. We hebben in deze grafiek gebruik gemaakt van de gemiddelde tijd per run dit zodat het beeld minder vertekend. Als we de totaal tijd gebruikt zouden hebben vallen de resultaten onder de 1000 samples grotendeels weg omdat dit relatief kleine waardes zijn. Omdat het bij ons niet bekend is welke manier van gray scaling gebruikt is in de default implementatie is het lastig te zeggen waardoor het komt dat onze implementatie sneller is.

5.2 Imageshell:

Hieronder zijn de gegevens van de imageshell te vinden in een grafiek.



Op basis van deze gegevens kunnen we een aantal zaken afleiden, de studentimplementatie is over het algemeen ca. 2x zo langzaam als de default implementatie. Echter tijdens het runnen van deze tests zijn er veel errors voorgekomen zie het kopje 7. Evaluatie voor uitgebreidere verklaring van deze errors. Dit heeft er in geresulteerd dat de test van 250 samples voor de default implementatie niet voltooid is.

6. Conclusie

6.1 Grayscale

Voor het grayscale algoritme is er uit de tests gebleken dat onze implementatie sneller is, dit kan meerdere oorzaken hebben maar de voornaamste is dat onze implementatie waarschijnlijk een makkelijkere rekensom.

6.2 Imageshell

Uit de test resultaten is gebleken dat de default imageshell sneller is dan onze geïmplementeerde imageshell. Wij vermoeden dat dit door de overhead van de stl container wordt veroorzaakt. Deze optie blijkt dus niet de goede geweest te zijn. Dit zou ook nog kunnen komen omdat de get/set(i) functie voor onze implementatie direct resultaat geeft maar voor de get/set(x,y) eerst een berekening uit moet voeren.

7. Evaluatie

Tijdens het testen van de imageshell is het meerdere malen voorgekomen dat het programma vast liep op een unhandled exception in de defaultExtraction.cpp. Deze heeft te maken met memory locations dit wordt waarschijnlijk veroorzaakt door de raw pointers die gebruikt worden. Dit heeft erin geresulteerd dat 1 van de tests van de default implementatie niet goed gerunt heeft. We hebben allerlei opties geprobeerd, denk aan rebuilden, herstarten etc. maar hebben het probleem niet op kunnen lossen. Gezien het aanpassen van deze classes etc. niet binnen de scope van de opdracht vallen hebben we het voor nu hierbij gelaten.