Implementatieplan Practicum 1

1 | Namen en datum

Utrecht, 5 februari 2019

Dylan Rakiman
Joost van Lingen

2 | Doel

Het doel van deze implementatie is het versnellen van het inladen van een plaatje in het werkgeheugen. Dit hopen we voor elkaar te krijgen door de benodigde hoeveelheid werkgeheugen te verminderen. Hierdoor hoeven er minder aanroepen naar het geheugen gedaan te worden wat veel tijd bespaart.

3 | Methoden

Methode 1: Sla elke pixel van een RGB-image op in een tweedimensionale array.

Methode 2: Sla elke pixel van een RGB-image op in een eendimensionale array.

Methode 3: Sla elke pixel van een RGB-image op in 24-bits in het geheugen. Dit is de minst geheugen-intensieve manier om een RGB-image op te slaan, maar vereist wel meerdere bitwise operaties om individuele gegevens uit het plaatje te halen omdat de data niet aligned is met de geheugenadressen van een x86- of x64-machine.

Methode 4: Sla elke pixel van een RGB-image op in 32 bits in het geheugen.. Hierbij worden de waarden ‘r’, ‘g’ en ‘b’ respectievelijk in de eerste 24 bits geplaatst. De laatste 8 bits worden leeg gelaten.

Methode 5: Sla elke kleurwaarde van een pixel op in een integer.

Methode 6: Sla elke kleurwaarde van een pixel op in een char.

Methode 7: Sla kleurwaardes op met signed variabelen.

Methode 8: Sla kleurwaardes op met unsigned variabelen.

4 | Keuze

Methode 1 is sneller wanneer wij kriskras pixels willen opvragen of veranderen in ons programma. We kunnen namelijk x- en y-waarden in de array indices stoppen om toegang tot de gewenste pixel te krijgen, zonder extra berekeningen te hoeven maken. We moeten echter wel de functies met een (int i)-parameter een extra berekening laten maken door bijvoorbeeld i modulo getWidth() te doen om de y-waarde te krijgen en i modulo getHeight() te doen om de x-waarde te krijgen.

Methode 2 is echter beter geschikt voor het inladen van een plaatje, waarbij de functies met een (int i)-parameter met een simpele for-loop aangeroepen kunnen worden, zonder verdere berekeningen in de achtergrond te hoeven doen. Dit betekent echter wel dat de functies met (int x, int y)-parameters wel weer berekeningen moeten doen. Aangezien ons doel is om het inladen van een plaatje te versnellen kiezen wij voor deze methode.

Wij kiezen ervoor om methode 3 en 4 te implementeren en de resultaten van de meetrapporten met elkaar te vergelijken. Methode 3 neemt minder geheugen op maar vereist soms toegang tot twee geheugenadressen en gebruikt meerdere bitwise operaties om gegevens te achterhalen of veranderen. Methode 4 neemt meer geheugen op maar is wel memory-aligned en zou dus efficiënter kunnen werken door de prefetcher; het inladen van een plaatje is immers relatief makkelijk te 'voorspellen'. Waar bij methode 4 consistent om de één of twee pixels (afhankelijk van 32- of 64-bits geheugen) een nieuw geheugenadres geprefetcht moet worden, is het relatief een stuk lastiger voor de prefetcher om voor methode 3 te voorspellen wanneer hij een nieuwe geheugenplek moet inladen.

De keuze tussen methode 5 (waarbij int8_t voor methode 3 en int32_t voor methode 4 gebruikt wordt) en methode 6 is vrij simpel: de c99 standaard specificeert int8_t als exact 8 bits en int32_t als exact 32 bits. Voor ons gebruik is het vaststellen van de exacte hoeveelheid bits van groot belang en het gebruik van fixed-width variabelen is dus in dit geval beter. Daarom kiezen wij voor methode 5.

De keuze tussen methode 7 en 8 is vrij simpel. Methode 8 heeft namelijk ofwel dezelfde ofwel betere performance dan methode 7; in het geval van delen door 2^n, delen door een constante of moduleren met 2 om te kijken of het getal even of oneven is, zal methode 8 sneller werken dan methode 7. Daarom kiezen wij dus voor methode 8.

5 | Implementatie

Voor beide implementaties maken we een array aan in de klasse "RGBImageStudent". Ook passen we de volgende functies aan zodat deze array correct aangeroepen/aangepast wordt:

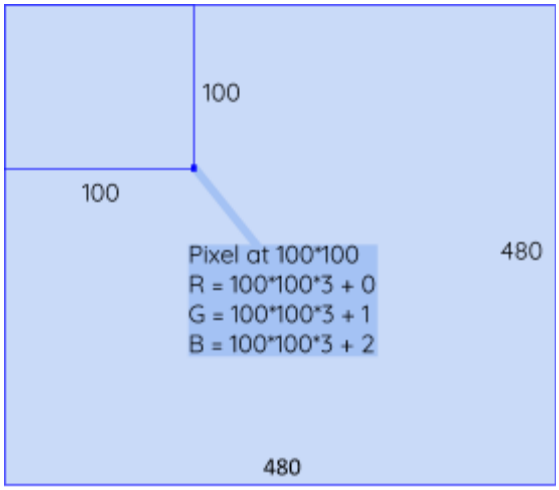
```
void set(const int width, const int height);
void set(const RGBImageStudent &other);

void setPixel(int x, int y, RGB pixel);
void setPixel(int i, RGB pixel);

RGB getPixel(int x, int y) const;
RGB getPixel(int i) const;
```

Methode 3

We maken een array aan van uint32_t en stoppen de waardes ‘r’ , ‘g’ en ‘b’ in de eerste drie bytes respectievelijk. De laatste byte laten we leeg. Één uint32_t stelt dus één pixel voor. De grootte van de array in bits is dus 32 maal de hoogte van het plaatje maal zijn breedte.

Methode 4

We maken een array aan van uint8_t en zorgen dat de grootte van de array driemaal de hoogte van het plaatje maal zijn breedte is. Vervolgens stoppen we omstebeurten de ‘r’, ‘g’ en ‘b’-waardes in hun respectievelijke uint8_t’s. Hierdoor kunnen we we een pixel als volgt kunnen oproepen: Als men vraagt om een pixel op een bepaalde hoogte en breedte, doen we die hoogte maal die breedte maal drie om op de juiste plek te komen. Op die plek zit de ‘r’-waarde opgeslagen; om de ‘g’-waarde te krijgen tellen we één bij deze plek op en om de ‘b’-waarde te krijgen tellen we twee bij deze plek op.



Visualisatie van de locatie van de kleurdata in het 1-dimensionaal array.

6 | Evaluatie

Om erachter te komen of methode 2 of 3 beter werkt zullen we experimenten doen die de snelheden van beide methoden met elkaar vergelijken. We zullen tests uitvoeren om de snelheid te controleren tussen de twee methodes door meerdere plaatjes te testen op meerdere systemen en het programma een tijdmeting te laten doen voor het inladen van elk plaatje. Ook zullen we tests doen voor het aanpassen van enkele/meerdere pixels.