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 of aanpassen 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 apart, memory-aligned adres. Op een x64-machine houdt dit dus in dat elke pixel in zijn eigen 64-bits geheugenadres geplaatst zal worden. Deze methode zal echter niet (goed) kunnen werken op een x86-machine.

Methode 2: Sla elke pixel van een RGB-image op in een 32-bits memory-aligned adres. Op een x86-machine zal dit hoogstwaarschijnlijk hetzelfde effect hebben als methode 1 maar op een x64-machine kan dit de hoeveelheid geheugen-oproepen halveren, aangezien er twee pixels in een geheugenadres passen.

Methode 3: Sla elke pixel van een RGB-image op door middel van 24-bits non-aligned stappen over 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.

4 | Keuze

Wij kiezen ervoor om methode 2 en 3 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 2 neemt meer geheugen op maar kan in theorie veel sneller werken omdat er maar één geheugenadres-oproep nodig is en deze oproep door gebruik van de prefetcher, wat alleen bij memory-aligned gegevens kan, versneld kan worden.

Methode 1 houden we buiten beschouwing omdat deze methode dezelfde voordelen als methode 2 biedt maar twee keer meer geheugengebruik vereist.

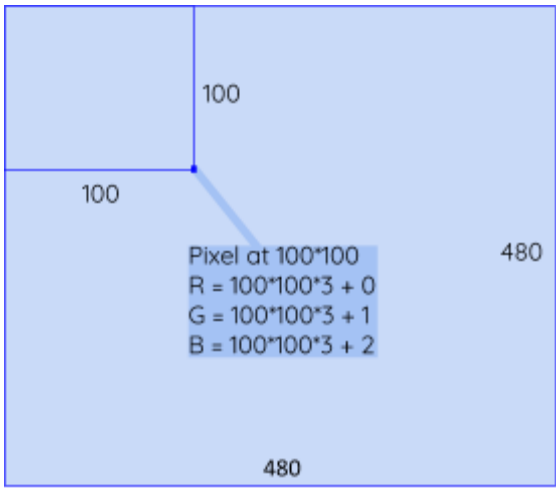
5 | Implementatie

Methode 2

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 3

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.

Om te kijken of de implementaties correct zijn zullen we ook een meting doen aan de hoeveelheid geheugen die verbruikt wordt voor het opslaan van een plaatje; theoretisch gezien zou het verschil in geheugengebruik rond de 25% moeten liggen.