Abstract

We have made a software-simulation of a digital camera with a filter function and a decompression function. We have tested the program on an ARM-processor and after analyzing the running times, we propose to change the filter function to hardware, as this will drastically improve the overall running time of the program.

Indhold

1 Introduktion

Dette projekt er den første del af to, der omhandler konstruktionen af et fuldt funktionelt digitalkamera, hvor vi i denne del fokuserer på softwaren. Dette betyder at programmet begrænses på visse områder, altså foretager vi nogle antagelser, som afgrænser problemstillingen. Vi fik den basale c kode udleveret, og så var det ellers vores opgave, at implementere de features vi mente der burde være i et digitalt kamera.

I rapporten vil der fokuseres på software implementeringen af programmet, hvor kildekoden er forklaret, samt de testmetoder, der anvendes til at bestemme hvordan softwaren skal gennemføres på hardwaren. Endvidere vil programstrukturen i de tilhørende funktioner blive gennemgået og vi vil opstille en analyse med vægt på optimering.

2 Analyse

I denne sektion vil vi analysere problemet, samt de specifikke krav opgaven, og vi selv, stiller til det færdige produkt.

2.1 Problemstillingen

Vi skal udforme et program til et digitalkamera, som vi alle kender det. Der findes i dag flere tusinde varianter af disse digitalkameraer, fælles for dem alle er, at de kan tage billeder og gemme dem digitalt på en eller anden form. Det er disse basiske funktioner, vi vil forsøge at implementere i vores egen simple udgave. I denne sammenhæng er der opstillet specifikke krav, for hvad kameraet som minimum skal kunne, hvilket vi kommer nærmere ind på i en kommende sektion.

Endvidere skal vi analysere på programmets køretider, optimerer vores algoritmer og til sidst sammenligne køretiderne på henholdsvis en PC og på den ARM processor programmet senere skal integreres i.

2.2 Funktioner

Som hentydet ovenfor har vi nogle krav til vores program, om nogle funktioner som kameraet skal have. Disse funktioner er beskrevet nedenfor, både generelt og specifikt i forhold til vores program. Desuden har vi skrevet noggle få ord om hvad vi forventer at køretiden nogenlunde bliver.

2.2.1 Bitmap

Bitmap, populært kaldet BMP, er et format benyttet til at lagre digitale billeder på. En bitmap fil er opbygget af først en BMPFILEHEADER, der indeholder information om billedet, såsom billedets størrelse, type mv. Dette efterfølges af en BMPINFOHEADER der indeholder information om selve billedet, f.eks. billedets højde og bredde.

Et sædvanligt ukomprimeret bitmap indeholder endvidere et gitter af pixels, hvor der på hver plads er gemt en farvedybde af 1, 4, 8, 16, 24, 32, 48, eller 64 bit per pixel. Igennem denne opgave vil vi hovedsageligt beskæftige os med farvedybderne 8-bit, der i vores opgave bruges til gråskalabilleder, og 24-bit, der bruges til de almindeligt farvede billeder. Antallet af bit per pixel og selvfølgelig også antallet af pixels, har direkte indflydelse på filens størrelse.

Vores program
Vores program skal kunne åbne, læse og ændre i bitmap-billedfiler for til sidst at gemme dem igen. Typen af bitmap filer vi skal kunne åbne er 24-bit. Vi kan nøjes med at arbejde med gråskala billeder, for derved at kunne nøjes med at gemme dem som 8-bit billeder, istedet for 24-bit. Herved skal vi dog så også

implementerer en algoritmen til at skifte fra 24 til 8 bit.

Ydeevne
Køretiden på at åbne og gemme filer kommer an på hvordan funktionerne er lavet i det valgte sprog, altså hovedsageligt hvilket sprog vi bruger, samt selvfølgelig størrelsen af filen. I denne opgave bruges C, så køretiden er nok hurtigere end i de fleste andre sprog. Samtidig bruger vi forholdsmæssige små filer, så alt i alt forventer vi en ret hurtig køretid.

2.2.2 RLE komprimering

RLE (Run-length encoding) er en simpel data komprimeringsalgoritme, der ofte benyttes i forbindelse med komprimering af 8-bit bitmap filer, som er det vi arbejder med i denne opgave. Kort sagt går denne algoritme ind og kigger på, hvor mange af de samme farver der sidder ved siden af hinanden i vores bitmap gitter, og reducerer dem så til en enkelt farveværdi og et fortløbende antal. Hvis der f.eks. er 4 bytes ved siden af hinanden, der indeholder samme farveværdi, ville disse 4 bytes blive reduceret til 2 (farve og antal) efter komprimering. Dette er smart fordi, at man kan gøre billeder med mange af de samme farver betydeligt mindre.

Vores program
Vi indbygger komprimering i vores program. Man kan ente vælge at sørge for at programmets dele (f.eks. filteret) både virker på komprimerede og ukomprimerede filer, eller kun ukomprimerede filer. Vælger man det første kan de forskellige funktioner blive en del mere komplicerede at implementerer. Vælger man det sidste kan programmet enten ikke åbne komprimerede billeder eller også er man nød til at implementerer en dekomprimeringsalgoritme.

Vi har i vores opgave valgt den sidste løsning, altså kun at arbejde med ukomprimerede billeder, og så implementerer en dekomprimeringsalgoritme.

Ydeevne
Den algoritmiske køretid af komprimerings algoritmen forventes at blive lig antallet af pixels i billedet, da hvert pixel skal undersøges. Komprimeringen tager derfor nok ikke særlig lang tid, men nok dog længerer tid end at åbne og gemme filer uden komprimering/dekomprimering.

2.2.3 Filter

Et filter i denne sammenhæng er en maske der tildeles hver eneste pixel. En maske der kigger på den valgte pixel, samt de nærmest omkringliggende og manipulerer dets værdi for at skabe forskellige effekter. Herved kan man for eksempel fremhæve kanter (linjer der adskiller områder med høj kontrast) eller lave 3D effekter (selvfølgelig uden at ændre at vi stadig bare har et almindelig 2D billed), samt meget mere.

Vores program
Der er i teorien ingen grænser for vore store filterne kan være (i højde x bredde), men større filtrer kræver flere beregninger. Der findes ligeledes flere filter algoritme. I vores program bruger vi "2D convolution" algoritmen med filter matrixer som vi har fået udleveret i opgaven.

Ydeevne
Eftersom at hvert pixel skal manipuleres og hver manipulering skal igennem hele filteret er køretiden her nok den højeste vi har blandt programmets funktioner. Vi forventer at køretiden er høj, og den afhænger af billedets størrelse, samt filterets størrelse.

3 Design

3.1 Overordnet arkitektur

Vi har i stor grad brugt de tildelte program filer, med den dertilhørende program arkitektur. Vi har dog oprettet de nye C filer, filter.c og mp.c samt deres dertilhørende header filer. Vi har altså hele softwaren for kameraets LCD i lcd.c og hele kameraets CCD i ccd.c. Alt der omhandler behandlign og læsning af bmp filer ligger i bmp.c, udover det der høre til filtrering som ligger i filter.c. Til sidst er alt der omhandler Micro Processoren beliggende i mp.c.

Vi har valgt denne strukturering da det er godt opdelt i forhold til selve hardware strukturen i kameraet, og det giver en overskuelig opdeling af programmet, og desuden ville det være et for stort og unødvændigt arbejde at ændre det udleverede kode.

3.2 Bitmapfils behandling

Vores program kan indlæse bitmapfiler af typen 24-bit og 8-bit, både komprimerede og ikke komprimerede. Først indlæses BITMAPFILEHEADER, derefter indlæses BMPINFOHEADER og til sidst indlæses hele bitmap billed delen, samt farvepaletten hvis denne findes. Det hele gemmes i programmet.

Vi har som sagt lavet en program del for CCD, LCD og MP så derfor skal disse billedinformationer flyttes før at billedet kan behandles et andet sted i programmet (billedet startes med at blive indlæst i CCD, når billedet "bliver taget").

3.3 Komprimerings algoritmen

Til komprimering bruger vi en algoritme der tager et ukomprimeret billed ind og leverer et komprimeret billed tilbage. Algoritmen går hver pixel i bitmappen igennem, én for én. Den starter ved en pixel og tjekker så den næste pixel, for at

se om de har samme farve værdi. Har de det tæller den en op, så vi hele tiden har det nuværende antal ens pixels i træk. Herefter tjekkes den næste pixel osv., indtil at vi finder en pixel med en ny farveværdi.

Når den når hertil skriver den komprimerings byte koden ind i en ny midlertidigt index til at gemme vores komprimerede billed, dvs. en byte for antallet vi kom frem til og en byte for den pågældende farvekode, som beskrevet i RLE-Komprimering under analyse.

Denne process gentages indtil hele billedet er blevet analyseret.

Dekomprimeringsalgoritmen virker på samme måde, bare hvor den tager et komprimeret billed ind, og indlæser to bytes af gangen fra billedst pixel index. I et nyt index gemmer den så et bestemt antal identiske bytes med den pågældende farvekode, svarende til det indlæste antal i den første indlæste byte. Dette går den med hvert sæt af bytes hele vejen igennem det komprimerede billede.

3.4 Filter algoritmen

Filter algoritmen tager et pixel index fra en bitmapfil, samt et filter som input. Dette gør at det er let at skifte det anvendte filter.
Algoritmen går hver pixel igennem, en af gangen, startende fra nederste venstre hjørne og så mod højre, en række af gangen, indtil den når sidste pixel i øverste højre hjørne. For hver pixel kører den vores opgivet "2D convolution" algoritme:

$$I'[r,c] = \frac{1}{\sum_{i,j} F[i,j]} \sum_{i=-\frac{m-1}{2}}^{\frac{m-1}{2}} \sum_{j=-\frac{m-1}{2}}^{\frac{m-1}{2}} F[i,j]I[r+i,c+j]$$

Dvs. den tager alle værdierne fra de pixels der ligger rundt om i et område svarende til størrelsen på filteret og ganger med den tilsvarende filterværdi. Summen normaliseres herefter ved at blive ganget med $\frac{1}{\sum_{i,j} F[i,j]}$, og dette endelige resultat gemmes som den nye værdi i vores pixel index. Har vi f.eks. et udsnit af et billed med værdierne

124	128	132	132
122	130	143	144
117	132	147	152
111	125	135	141
99	108	120	126

og et filter med værdierne

0	1	0
1	1	1
0	1	0

og bruger vi algoritmen på det felt der er markeret med rød vil vi få det følgende resultat:
$\frac{1}{0+1+0+1+1+1+0+1+0} * (0*99+1*108+0*120+1*111+1*125+1*135+0*117+1*132+0*147) = \frac{1}{5} * (108+111+125+135+132) = \frac{1}{5} * 611 = 122,2.$
125 vil altså blive skiftet ud med 122 (der rundes ned). Det pågældende filter vil sløre billedet, en såkaldt "blur effekt".

Når filteret påføres de pixels der ligger ved kanten (eller tæt på kanten for større filtrer), således at filteret prøver at regne med nogen værdier der "ligger uden for billedet", altså ikke findes bruger algoritmen istedet den nuværende pixels værdi, altså den pixel som algoritmen er igang med at udregne en ny værdi for.

3.5 Brugergrænsefladen

I vores main.c fil har vi oprettet to funktioner; start_ui(), og test(). Disse gør grundlæggende det samme, den ene bare med CUI (console user interface) og den anden automatisk. Hvad disse funktioner gør er at samle de forskellige dele af programmet og køre de 3 funktioner som kameraet skal kunne, samtidig med at de tester køretiden, nemlig at tage et billed (capture_image), filtrerer billedet, samt vise billedet på LCD (show_image).

Alle underlæggende processor testes for sig, f.eks. overførselstiden fra CCD til micro processoren osv.

4 Implementering

4.1 Komprimering

Komprimeringsalgoritmen implementeres ved at lave en funktion der tager en pointer til et array af pixelfarveværdier og gennemløber dette array. Dette gøres ved at løbe igennem to løkker, den anden indlejret i den første, som kører igennem henholdsvis hver række og hver kolonne et et arrayet. Da arrayet dog er 1-dimensionelt gøres det ved at tage den pågældende række og gange med bredden af billedet, og så lægge den pågældende kolonne til, for at få det pågældende index i arrayet. For hver værdi køre vi så vores komprimeringsalgoritme og gemmer resultatet i et nyt array. Til sidst overskrives det array der pointes til med de gemte værdier fra komprimeringen. På samme måde køres dekomprimeringsalgoritmen.

Vi har testet begge algoritmer, ved at komprimerer og dekomprimerer forskellige filer. Ved ukomprimerede filer får vi altid næsten samme filstørrelse cirka 260 kb, imens vi ved komprimerede filer får filstørrelse der stærkt varierer. Der er dog en tydelig sammenhæng imellem størrelsen af den komprimerede fil og så størrelsen af områder med ens farve. Således bliver de udleverede eksem-

pler fra CambusNet, example24.bmp og example8rle.bmp, efter komprimering henholdsvis 477 kb og 22,8 kb.

4.2 Filter

Vi har delt filter processen op i flere metoder. Først en metode som står for at strukturerer og køre de andre metoder (filter_image). Det er denne metode der bliver kaldt når et billed skal filtreres. Denne metode tager et filter id og en pointer til et pixel array som input.

Filter id'et sender den til en anden metode (create_filter) som opretter et filter udfra en filter struktur der er defineret i filter headeren, med de informationer der står i filter headeren under det givne id. I vores filter header har vi nemlig gemt et array som indeholder 18 filtre samt et andet array der indholder information om de 18 filtre. Disse informationer inkluderer, størrelse bias og factor (se under).

Når filteret er skabt returners en pointer til dette filter. Herefter sendes denne pointer samt pointeren til pixel arrayet til en tredje metode (filter_process) som køre selve filteralgoritmen med det givne filter, på det givne billed. Filter algoritmen er programmeret som fire indlejrede løkker. De første sørger for at vi går igennem vores billed pixel for pixel. Den første går igennem hver række (startende fra sidste række), og den anden igennem hver kolonne (startende fra første kolonne), og derved går vi fra nederste venstre hjørne og til højre og op. Inden i disse løkker ligger de to næste løkker som løber igennem de pixels der indgår i filteret.

Under hver gennemgang i den inderste løkke tilføres den pågældende værdi til den samlede sum, som til sidst (efter at blive normaliseret) bliver gem som den nye bit værdi, efter gennemkørslen af hele filteret. Herefter går vi så altså videre til næste pixel og køre filteret igen.

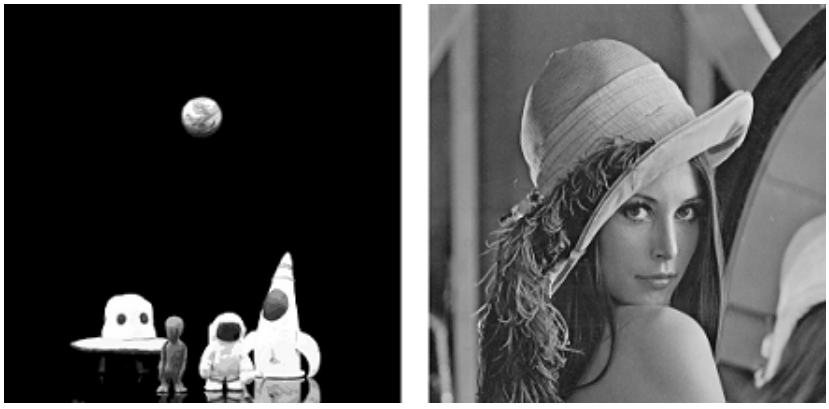
Prøver algoritmen at regne på pixels i filteret der ligger uden for billedet, altså ved kanterne, så tager den den pågældende pixel værdi istedet. Altså er vi ved pixel (0,100) med et 3x3 filter, vil filteret bruge denne værdi (altså værdien i (0,100)) istedet for de værdier der normalt ville ligge på (-1,99), (-1,100) og (-1,101).

Alle filtrerne er afprøvet og har givet det forventede resultat. F.eks. ses nedeunder et billed før og efter filtrering. Det pågældende filter er et såkaldt "emboss effekt" filter; [-1, -1, 0, -1, 0, 1, 0, 1, 1], der giver en 3D effekt som får billedet til at se ud som om det "går ind", ligesom hvis det var udhugget i en mur:

5 Resultater

5.1 Filstørrelse

Som beskrevet under implementering så variere resultatet af filstørrelsen efter komprimering, meget fra billed til billed. Således bliver billedet til venstre 22,8 kb stort efter komprimering, hvorimod billedet til højre bliver 477 kb stort.



Resultatet er ret godt for det første billed, hvorimod det andet billed faktisk bliver større af at blive komprimeret (ukomprimet som 8-bit, sort-hvid, fylder begge billeder 257 kb).

Det er muligt at der findes andre komprimeringsmetoder end RLE-komprimering der er mere effektiv. Ellers vil man kunne forbedre programmet ved at teste om komprimeringen formindsker billedet, og hvis den ikke gør, så gemme det ukomprimeret. Det ville kun bruge meget lidt cpu kræft (tage meget kort tid) at undersøge dette. Det bedste vil dog være hvis man kunne lave en vurdering

på forhånd inden billedet blev komprimeret. Jeg kan dog ikke se nogen løsning til at gøre dette.

5.2 Køretider

Følgende er de køretider vi fandt ved at køre vores program på ARM-processoren.

Funktion	average-case	worst-case
Time to capture	0.00 sec	0.00 sec
Transfer from ccd to mp	1.36 sec	1.42 sec
Filter	4.22 sec	4.31 sec
Transfer from mp to lcd	0.54 sec	0.58 sec
Show on lcd	0.00 sec	0.00 sec
Total cycles	3,342,490	3,342,661

Tabel 1: køretiderne ved anvendelse af 3x3 filter på small.bmp (32x32)

Funktion	average-case	worst-case
Time to capture	0.05 sec	0.05 sec
Transfer from ccd to mp	138.59 sec	146.91 sec
Filter	463.84 sec	472.87 sec
Transfer from mp to lcd	58.49 sec	60.08 sec
Show on lcd	0.027 sec	0.03 sec
Total cycles	922,455,980	922,456,747

Tabel 2: køretiderne ved anvendelse af 3x3 filter på example24.bmp (512x512)

Funktion	average-case	worst-case
Time to capture	0.07 sec	0.07 sec
Transfer from ccd to mp	138.21 sec	145.13 sec
Filter	609.24 sec	616.76 sec
Transfer from mp to lcd	59.52 sec	60.93 sec
Show on lcd	0.03 sec	0.03 sec
Total cycles	1,207,853,890	1,207,854,513

Tabel 3: køretiderne ved anvendelse af 5x5 filter på example24.bmp (512x512)

6 Diskussion

Capture Image og Show Image tager stort set ingen tid, idet de bare henter og gemmer en fil, så det er ikke så de er ikke så interessante. Derimod tager alle de tre andre funktioner - CCD to MP, Filtering og MP to LCD - rimelig langt tid, og de to transfer funktioner vokser meget kraftigt når billedet bliver større. Filter-funktionen tager naturligvis længere tid når man bruger et større filter. De indeholder alle tre bottlenecks for programmet, så hvordan kan vi søge at afhjælpe dem? Problemet med de to transfer funktioner er at de loader hver enkelt pixel for sig, så derfor er køretiden proportional med antallet af pixels. Den eneste oplagte måde at forbedre køretiden er ved at parallellisere overførslen, men det kræver dual-core teknologi. En måde at optimere filtreringsalgoritmen på kigger vi på i næste afsnit.

For at finde ud af hvilke dele af programmet, som ville være velegnet til at lave som hardwareløsning, laver vi running time tests af hver funktion. Da vi skal lave så lidt som muligt som hardware, finder vi den process som tager længst tid i forhold til de andre. Dette vil gøre den samlede køretid minimal med så lidt hardware som muligt. Der er tydeligt i køretiderne fra ARM-processoren at det er filtreringen der tager klart længst tid, og som man kan se senere i rapporten bliver forholdet endnu større på ARM-processoren, så det er her vi skal sætte ind. Nu skal vi så finde ud af hvilken del af filtreringen der tager tid, og her kan vi bare se på køretiderne fra Windows-testene at det er 2D-konvolution algoritmen der er tidssluger. De tre tider for filtrering var 0.574, 0.574, 0.585ms og de tilsvarende køretider kun for 2D-konvolution var 0.573, 0.573 og 0.584ms. Det er alle for-løkkerne og summeringen af pixels der tager så lang tid, så vi ville lave en specialiseret hardware-enhed til at køre for-løkker og summering. Denne relativt lille ændring ville sænke den samlede køretid dramatisk og er vores forslag til hardware/software partitioning. Hvis man ser

Funktion	**average-case**	**worst-case**
Time to capture	0.05 sec	0.05 sec
Transfer from ccd to mp	0.002 sec	0.002 sec
Filter	0.578 sec	0.585 sec
Transfer from mp to lcd	0.002 sec	0.002 sec
Show on lcd	0.03 sec	0.03 sec

Tabel 4: køretiderne ved anvendelse af 3x3 filter på example24.bmp (512x512)

på 2D-konvolution algoritmen er køretiden for de tre testkørsler hhv. 0.573, 0.573 og 0.584ms.

7 Konklusion

Vi har igennem denne rapport arbejdet med håndteringen af bitmap filer i et digitalt kamera. Vi har tilegnet viden om bitmap formatet, hvordan det skal håndteres og hvordan det kan manipuleres.

Det første problem vi stødte på i udviklingsprocessen var implementeringen af matrixfiltre, og problematikken der ligger i at få programmet til at ignorere kanten. Vi er yderst tilfredse med den løsning vi kom frem til og kan konkludere, at den fungerer uden større problemer. Dog må vi konstatere at køretiden på disse stiger voldsomt, i takt med at billedet det bruges på bliver større, såvel som når kompleksiteten i filret forøges.
Med den størrelse billeder moderne kameraer kan indfange i dag, er det yderst nødvendigt at optimere koden, så den kan arbejde effektivt sammen med nutidens komplicerede mikroprocessorer. En komplet optimering, ville omfatte at konvertere så meget som muligt, af vores nuværende kode til assembler - specielt den tunge filteralgoritme.

Overordnet er vi godt tilfredse med vores produkt. Vores program køre ufejlmæssigt, og algoritmerne er effektive (på den måde at de ikke, eller næsten ikke, kan optimeres softwaremæssigt), og strukturen er god. Vi har dog lidt problemer med en ordentlig user interface. Lige nu ligger der en ubrugt funktion, kaldt start_ui(), i main.c som starter et CUI. Denne CUI virker dog ikke i eclipse, kun i windows commando prompt. Da vi dog kun opfattede UI som hjælp til at vi selv kunne teste programmet, har vi dog valgt at køre og teste programmet automatisk istedet.