UNIVERSITEIT GENT

FACULTEIT WETENSCHAPPEN

Systeemprogrammeren

Academiejaar 2018–2019

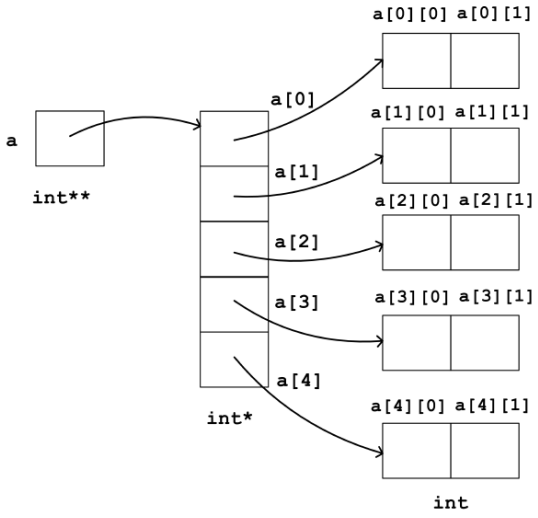
sysprog@lists.UGent.be

Oefeningenset 3
Omgaan met meerdimensionale arrays

Manipulatie van afbeeldingen

Situering

Deze oefening behandelt multidimensionale rijen in C. Figuur 1 toont de geheugenstructuur van een 2D-rij. Wanneer er geheugen gealloceerd dient te worden, wordt dit in twee stappen gedaan: eerst wordt een blok gealloceerd om de wijzers naar de rijen van de matrix (in dit voorbeeld vier wijzers) in op te slaan. Daarna wordt per rij een blok gealloceerd om de elementen in de rij (hier twee) op te slaan. Bij het vrijgeven van het geheugen dient in omgekeerde volgorde gewerkt te worden: eerst worden de rijelementen vrijgegeven, dan de rijen.



Figuur 1: geheugenstructuur van een 2D-rij van gehele getallen

De eerste oefening gaat over het aanmaken van en werken met 2D-matrices. De tweede oefening gaat over het bewerken van afbeeldingen via 3D-rijen.

Het bestand `main.c` bevat functies om je code te testen.

Bewerkingen

In deze oefening wordt een matrix van gehele getallen aangemaakt en worden enkele basis matrixbewerkingen ontwikkeld. In `matrix.h` vind je de volgende `struct`:

```
typedef struct {
        int num_rows;
        int num_cols;
        int** data;
} matrix;
```

Deze houdt het aantal rijen en kolommen van de matrix bij, alsmede een pointer naar matrix data in het geheugen.

Gevraagd wordt nu om volgende functies te implementeren in `matrix.c`:

```
void init_matrix(matrix* m, int num_rows, int num_cols);
```

⇨ initialiseert een matrix met het opgegeven aantal rijen en kolommen, reserveert geheugen voor de data en initialiseert alle elementen op 0.

```
void init_matrix_default(matrix* m);
```

⇨ doet hetzelfde als de vorig functie, maar gebruikt de default variabelen in `matrix.h` voor rij- en kolomgrootte.

```
void init_identity_matrix(matrix* m, int dimension);
```

⇨ intialiseert een eenheidsmatrix met grootte *dimension*. Een eenheidsmatrix is een vierkante matrix met enen op de hoofddiagonaal en de rest allemaal nullen.

```
void free_matrix(matrix* m);
```

⇨ geeft het geheugen ingenomen door de matrix vrij.

```
void print_matrix(matrix* m);
```

⇨ print de data van de matrix op een overzichtelijke manier (hou bij het printen rekening met het feit dat getallen soms meerdere karakters kunnen bevatten. Je mag hierbij veronderstellen dat getallen maximaal 3 karakters groot zijn).

```
void transpose_matrix_pa(matrix* m);
```

⇨ transponeert de matrix. Je mag er van uit gaan dat de matrix vierkant is. Gebruik in deze functie pointer-arithmetiek en de adresseringsmethode (`*(matrix+i)`) voor het adresseren van de matrixelementen, ipv de notatie `matrix[i]`.

```
void transpose_matrix_sn(matrix* m);
```

⇨ transponeert de matrix. Je mag er opnieuw van uit gaan dat de matrix vierkant is. Gebruik in deze functie de `matrix[i]` addresseringsmethode.

```
void multiply_matrices(matrix* a, matrix* b, matrix* result);
```

⇨ initialiseert matrix `result` met het juiste aantal rijen en kolommen en slaat het resultaat van de vermenigvuldiging van matrix a en b erin op.

```
void dynamic_expand(matrix* m, int new_num_rows, int new_num_cols);
```

⇨ breidt de matrix dynamisch uit (de matrix kan enkel groter worden, veronderstel dat `new_num_rows` ≥ huidige aantal rijen en `new_num_cols` ≥ huidig aantal kolommen). Het resultaat (dat na uitvoeren van de functie in `m` terecht moet komen) is een matrix waarbij alle elementen van de oorspronkelijke matrix in de linkerbovenhoek staan. De extra rijen en/of kolommen worden opgevuld met nullen.

Als alle functies zijn geïmplementeerd, kan je deze testen door de main-functie uit te voeren. Deze voert volgende matrix bewerkingen uit:

$$A = \begin{pmatrix} 1 & 0 & 2 \\ 0 & 3 & 1 \end{pmatrix}$$

$$B = \begin{pmatrix} 0 & 3 \\ 2 & 1 \\ 0 & 4 \end{pmatrix}$$

$$C = A \times B$$

$$C = C^T$$

$$A = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$B = C \times A$$

$$B = \begin{pmatrix} (\ B\ ) & 0 & 0 \\ & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

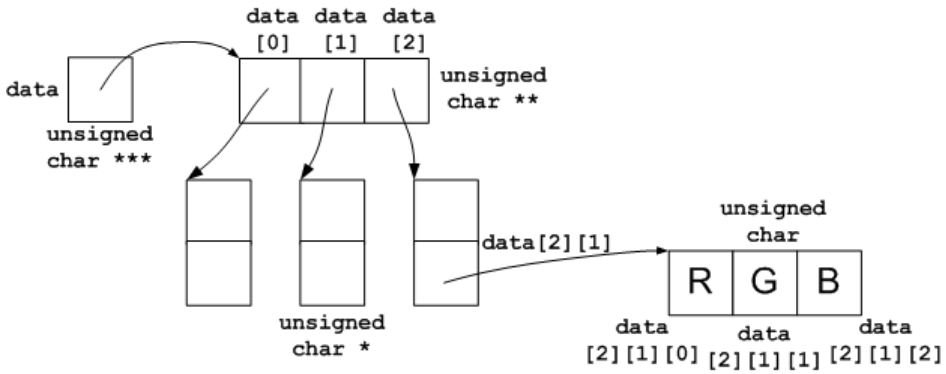
Image processing

Opmerking: tijdens deze oefening gebruiken we het *ppm* bestandsformaat. Windows heeft geen standaard viewer voor dit formaat. Om het resultaat te bekijken hebben we een extra tool nodig zoals IrfanViewer (via www.irfanview.com of uw favoriete zoekmachine kan deze tool online teruggevonden worden).

Bitmap-afbeeldingen zijn een 2-dimensionaal raster van punten (of pixels). Per pixel worden 3 kleurcomponenten bijgehouden, de zogenaamde RGB waarden (RGB staat voor rood, groen en blauw). Een R, G of B waarde is voor 24-bits kleuren een byte groot (dus een getal tussen 0 en 255). In deze oefening zullen we afbeeldingen inladen in een 3-dimensionale rij (denk ruimtelijk: drie 2-dimensionale matrices na elkaar, eentje per kleurcomponent R, G en B) en manipuleren. In `image.h` vind je de volgende `struct`:

```
typedef struct {
     int width;
     int height;
     unsigned char*** data;
} image;
```

Deze houdt de breedte en hoogte van de afbeelding bij. Merk op dat de grootte van de derde dimensie niet hoeft bijgehouden te worden, gezien deze steeds 3 is. Als datatype voor de kleurcomponenten werd gekozen voor `unsigned char` omdat die exact 1 byte groot zijn, en dus 256 waarden in het interval [0-255] kunnen bevatten.



Figuur 2: geheugenstructuur van een image

Figuur 2 toont de geheugenstructuur van het data-element van zo een afbeelding met breedte 3 en hoogte 2. Let op: bij afbeeldingen duidt de eerste coordinaat naar de kolom en de tweede naar de rij (dus omgekeerd in vergelijking met matrices). Dus `data[x][y]` wijst naar de R,G en B waarden van de pixel op de x-de kolom en y-de rij.

Het element `data` in de struct `image`, van het type `unsigned char***`, wijst naar een rij van type `unsigned char**` met lengte width. De elementen in die rij (`data[x]`) wijzen op hun beurt naar een rij van het type `unsigned char*` met lengte height. De elementen in deze rijen (`data[x][y]`) wijzen op hun beurt naar een RGB rij met lengte 3. Voor de eenvoud staat in de figuur maar 1 van de 6 RGB rijen afgebeeld.

Implementeer de volgende functies in `image.c`:

```
void init_image(image* im, int width, int height);
```

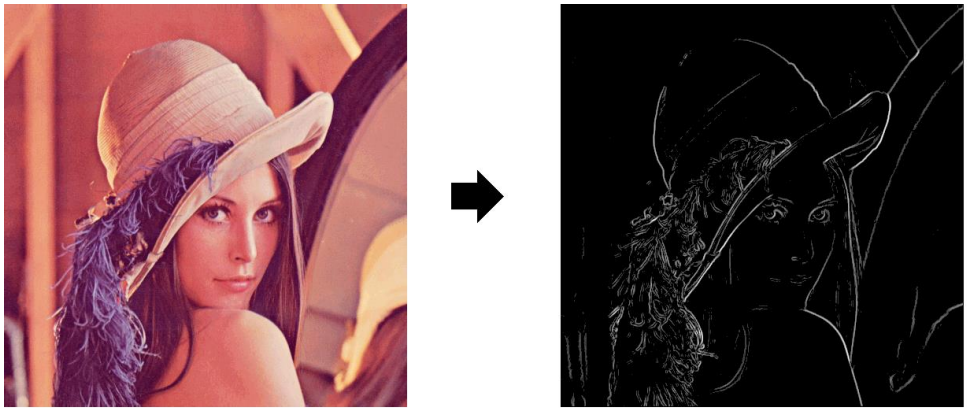
- ⇨ initialiseert een image met de opgegeven breedte en hoogte, reserveert geheugen voor de kleurcomponenten.

```
void free_image(image* im);
```

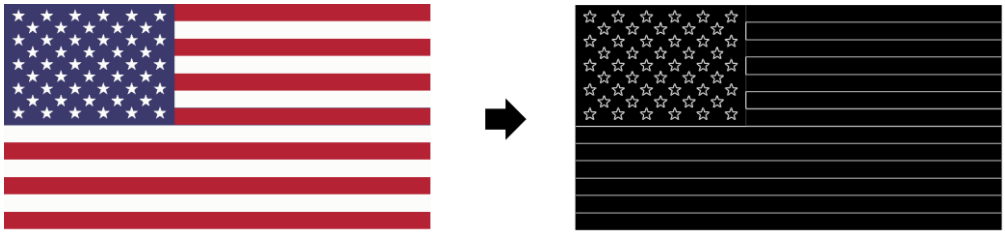
- ⇨ geeft het geheugen vrij van de afbeelding

Edge detection

Doelstelling: Het detecteren van randen in een afbeelding, zoals getoond in Figuur 3 en Figuur 4.

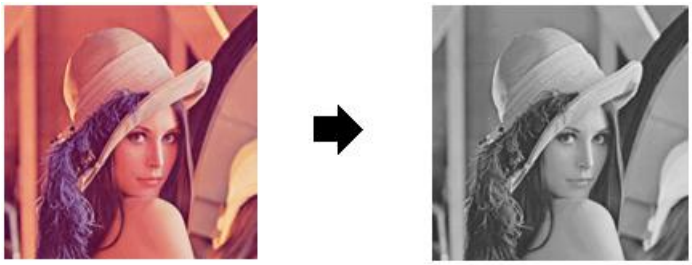


Figuur 3: edge detection Lena



Figuur 4: edge detection vlag

Werkwijze: Om het detecteren van randen in de afbeelding te realiseren, wordt de afbeelding eerst geconverteerd naar grijswaarden (zie Figuur 5). Hierbij kan er gebruik gemaakt worden van de gegeven functie `void RGB_to_gray(image* im)`, welke een gelijke waarde voor elk kleurkanaal (RGB) berekent via een empirische formule. Het detecteren van randen gebeurt op basis van de zogenaamde *Sobel-operator*. Deze Sobel-operator bestaat uit twee zogenaamde kernels (dit zijn twee 3x3 matrices S_x en S_y, weergegeven in Figuur 6). Door deze twee kernels te laten inwerken op elke pixel van de afbeelding, worden de afgeleiden van de intensiteitovergangen berekend zowel in de verticale als de horizontale richting.



Figuur 5: conversie van RGB naar grijswaarden

-1	0	+1
-2	0	+2
-1	0	+1

+1	+2	+1
0	0	0
-1	-2	-1

Figuur 6: Sobel operator: links S_x en rechts S_y

De berekening gaat als volgt: rondom elke pixel (de uiterste randen van de afbeelding dienen zwart ingekleurd te worden) wordt een 3x3 raster gedefinieerd. Om randen in de horizontale richting te detecteren gaan we eerst de linkse matrix in Figuur 6 elementsgewijs vermenigvuldigen met dit 3x3 raster. De waarde G_x van de intensiteit van de overgang in de horizontale richting wordt bekomen door deze vermenigvuldigde elementen op te tellen. Aldus wordt element (x, y) vermenigvuldigd met element (x, y) van de Sobel-operator S_x voor de horizontale richting (Figuur 6 links). Een klein voorbeeld: voor de pixel $P(1,1)$ op positie $(1,1)$ wordt G_x op positie $(1,1)$ bekomen door de volgende berekening:

$$G_x = P(0,0) \cdot S_x(0,0) + P(0,1) \cdot S_x(0,1) + \cdots + P(2,2) \cdot S_x(2,2).$$

Voor het berekenen van de intensiteitovergangen in de verticale richting wordt G_y op een gelijkaardige manier berekend. Hierna wordt de totale afgeleide G als volgt berekend:

$$G = \frac{1}{8}\sqrt{G_x^2 + G_y^2}$$

De originele grijswaarden van de pixels van de afbeelding worden dan vervangen door deze G-waarden. De laatste stap bestaat eruit alle G-waarden die lager zijn dan een bepaalde drempelwaarde (vb. 50) op zwart te zetten. Het uiteindelijke resultaat dient eruit te zien als Figuur 3 en Figuur 4.

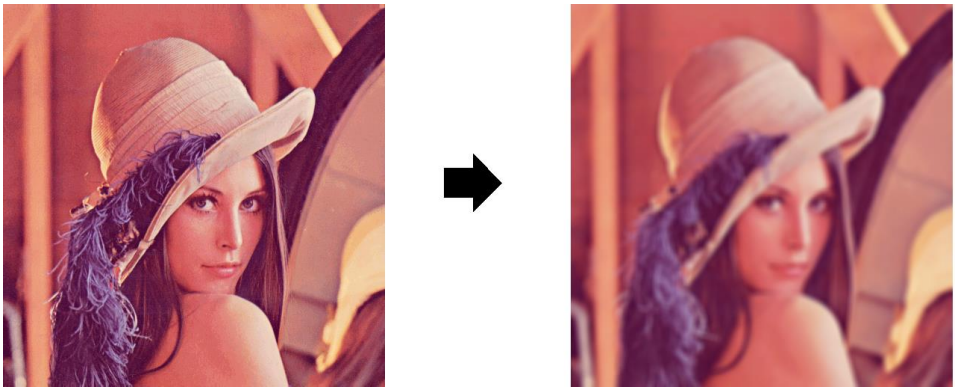
Deze verwerkingsstap wordt geïmplementeerd via de volgende functie:

```
void sobel_operator(image* im);
```

- ⇨ Detecteer lijnen in de afbeelding op basis van grijswaarden met de Sobel operator zoals hierboven beschreven. **Indien je bij het kopïeren van de afbeelding nood hebt aan extra datastructuren, moeten deze correct gealloceerd en vrijgegeven worden in het geheugen.** Verder mogen de randen van de afbeelding zwart ingekleurd worden. Het resultaat wordt teruggegeven via `image* im`.

Gaussische blur

Doelstelling: Het vertroebelen van een afbeelding zoals te zien in Figuur 7. Dit kan onder meer toegepast worden in beeldverwerking om de voorgrond van een afbeelding meer te benadrukken door de achtergrond te vertroebelen.



Figuur 3: vertroebelen van de afbeelding (radius = 5)

Werkwijze: In deze deelopdracht wordt de originele afbeelding vertroebeld op basis van Gaussische blur. Het toepassen van Gaussische blur gebeurt via de convolutie van de afbeelding met de Gaussische functie. Dit betekent dat we voor elk kleurkanaal (RGB) van elke pixel van de originele afbeelding de integraal nemen over de originele afbeelding (hier als functie van de pixels) vermenigvuldigd met een tweede functie (de convolutiefunctie f), welke telkens verschoven worden. Verder dient deze integraal genormaliseerd te worden door het resultaat te delen door de som van alle toegepaste vermenigvuldigingen met f. Daar onze afbeelding uit pixels bestaat, herleidt deze integraal zich tot een discrete som. Om de berekening te versnellen gaan we niet de volledige Gaussische functie toepassen, maar beperken we ons tot een vierkant rond de pixel (behalve in de rand waar dit vierkant aangepast moet worden), dit is een zogenaamde kernel zoals reeds gebruikt in de vorige deelopdracht. Dit laat toe de berekening van de convolutie voor elke pixel (x, y) te beperken tot $x \pm r$ en $y \pm r$ (r noemen we de radius van de blur) aangezien de waarde van de Gaussische functie toch verwaarloosbaar wordt naarmate we verder van het gemiddelde afwijken. De toe te passen blur-functie kan dus als volgt uitgeschreven worden (dit voor elk kleurkanaal (RGB)):

$$deel\_resultaat[x, y] = \sum_{i=-r}^{r} \sum_{j=-r}^{r} afbeelding[x + i, y + j] \cdot f(i, j)\,.$$

In ons geval is de functie f een tweedimensionale Gaussische functie

$$f(i, j) = \frac{1}{r^2\sqrt{2\pi}} e^{\frac{-(i^2+j^2)}{2r^2}}.$$

Hierna dient het resultaat nog genormaliseerd te worden door $deel\_resultaat[x, y]$ te delen door de bijhorende som

$$\sum_{i=-r}^{r} \sum_{j=-r}^{r} f(i,j).$$

Hierna wordt elk kleurkanaal van elke pixel vervangen door de waarde van dit resultaat, voor het bijhorende kleurkanaal. Als dit correct uitgevoerd is, dien je de afbeelding in Figuur 7 te verkrijgen.

```
void blur_picture(image* im, int radius);
```

- ⇨ Vertroebel de afbeelding via Gaussische blur zoals hierboven beschreven. **Indien je bij het kopïeren van de afbeelding nood hebt aan extra datastructuren, moeten deze correct gealloceerd en vrijgegeven worden in het geheugen.** Het resultaat wordt teruggegeven via `image* im`. Evalueer vervolgens de functie voor enkele r waarden (`radius = 2, 5, 10`).