Tema 2 - Paint Simulator 2016

Responsabili

- Călin Cruceru
- Radu Iacob

Termen de predare: **09.12.2016, ora 23:00**

Pentru fiecare zi (24 de ore) de întârziere, se vor scădea 10 puncte din nota acordată. Temele trimise după 7 de zile de întârziere vor putea fi notate cu maxim 30 de puncte. În consecință, deadline-ul hard este **16.12.2016, ora 23:00**.

Întrebări

Dacă aveți nelămuriri, puteți să ne contactați pe forumul dedicat.
La orice întrebare vom răspunde în maxim 24 de ore.
Nu se acceptă întrebări în ultimele 24 de ore înainte de deadline.

Actualizări

- **[Călin; 25.11.2016 - 23:00]** Publicare enunț.
- **[Radu; 26.11.2016 - 17:44]** Update public checker (outputul de referință pentru testul 16 era incomplet)
- **[Radu; 03.12.2016 - 13:30]** Update public checker (checker-ul nu se mai opreste din evaluare pentru primele teste daca solutia intoarce un cod de eroare)
- **[Radu; 04.12.2016 - 00:00]** Update public checker (adaugat un timeout de 15s, entru a evita situatia in care o solutie hanguie pe un test si blocheaza in felul acesta verificarea celorlalte teste)
- **[Radu; 04.12.2016 - 30:00]** Adăugat o sugestie referitoare la apelarea comenzii **fflush (stdout)** inainte de a afișa un mesaj la stderr.
- **[Radu; 04.12.2016 - 14:40]** Update public checker (checkerul intoarce mesaje mai explicite pentru cazurile frecvente de eroare (ex: segfault, sigabrt))
- **[Radu; 06.12.2016 - 18.00]** Update public checker (am relaxat putin verificarea condiției de egalitate între fișierul obținut și fișierul de referință; checkerul ignoră eventuale spații redundante la finalul unei linii respectiv între valorile unor elemente)

Obiective Temă

- să se utilizeze corect funcţiile de alocare şi eliberare a memoriei
- să verifice daca instrucțiunile primite respectă proprietățile specificate
- să se însușească cunoștințele din primele opt laboratoare
- familiarizarea cu modul de reprezentare a imaginilor ca matrice de pixeli

Cerință

Pentru aceasta temă veți implementa funcționalitățile de bază ale unui editor simplu de imagini (ex: Microsoft Paint, Pinta, etc.). Concret, programul vostru va citi de la tastatură o serie de comenzi de

prelucrare de imagini. La final, programul va afișa imaginea rezultată.

Imaginea va fi reprezentă printr-o matrice de pixeli, de dimensiune **height** * **width**. Un pixel este reprezentat prin 3 bytes (numere cu valori între 0 și 255), conform modelului RGB. Canalele pentru fiecare pixel sunt salvate in ordinea RGB (**R**ed, **G**reen, **B**lue).

Pe fiecare linie citită de la tastatură va fi compusă dintr-un cod de operație și parametrii corespunzători.

```
opCode [p1 p2 p3..]
```

Operațiile pe care trebuie să le implementați sunt:

Inițializare pornind de la o imagine existentă

Format:

```
1 width height
...
[height linii cu 3 * width bytes]
```

Pe prima linie este se află codul de operație (1), împreuna cu numărul de linii, respectiv coloane ale imaginii. Pe urmatoarele **height** linii, sunt 3 * **width** valori, reprezentând valorile **r**, **g**, **b** pentru fiecare pixel din imagine.

Exemplu

```
1 2 3
0 0 255 0 255 0
255 0 0 128 0 128
255 255 255 0 0 0
```

Acest apel înlocuiește imaginea curentă cu o imagine nouă având 3 linii si 2 coloane.

Pixelul (0, 0) are valoarea rgb (0, 0, 255), prin urmare este albastru.

Pixelul (0, 1) are valoarea rgb (0, 255, 0), prin urmare este verde, etc.

Properietăți argumente

$1 \leq height \leq 1024$

$1 \leq width \leq 1024$

$0 \leq r, g, b \leq 255$

Nu este obligatoriu, dar v-ar ușura implementarea dacă ați folosi structuri pentru a reține imaginea. (vezi laboratorul 10) De exemplu, ați putea stoca valorile corespunzătoare unui pixel in felul următor:

```
struct pixel {
   unsigned char r, g, b;
}
```

Nu alocați static o matrice cu dimensiunile maxim posibile dacă nu este necesar. S-ar putea ca utilizatorii să folosească în practică aplicația doar pentru imagini mult mai mici.

Crop

Format:

```
2 start_col start_line end_col end_line
```

În urma operației, imaginea curentă este suprascrisă cu imaginea delimitată de drepunghiul având colțul stânga sus cu coordonatele **(start_line, start_col)** și colțul dreapta-jos cu coordonatele **(end_line, end_col)**.

Exemplu

Imaginea Originală:

```
(0, 0, 255) (0, 255, 0) (255, 0, 0)
(0, 128, 0) (0, 255, 0) (0, 128, 0)
```

Instrucțiunea:

```
2 0 0 1 0
```

Imaginea Obținută:

```
(0, 0, 255) (0, 255, 0)
```

Properietăți argumente

0 ≤ start_line ≤ end_line < Înaltimea curentă a imaginii

0 ≤ start_col ≤ end_col < Lățimea curentă a imaginii

Resize

Format:

```
3 width height
```

Modifică dimensiunea imaginii curente. Dacă una din dimensiuni este mai mare, noile linii și/sau coloane vor fi completate cu pixeli albi. Dacă noile dimensiuni sunt mai mici, apelul este echivalent cu o instrucțiune de tip **crop**:

```
2 0 0 (width-1) (height-1)
```

Exemplu

Imaginea Originală:

```
(0, 0, 255) (0, 255, 0) (255, 0, 0)
(0, 128, 0) (0, 255, 0) (0, 128, 0)
```

Instrucțiunea:

```
3 1 3
```

Imaginea Obținută:

```
(0, 0, 255)
(0, 128, 0)
(255, 255, 255)
```

Properietăți argumente

1 ≤ height ≤ 1024

1 ≤ width ≤ 1024

Colorează regiunea

Format:

```
4 start_col start_line end_col end_line r g b
```

În urma operației, pixelii din dreptunghiul având colțul stânga sus cu coordonatele **(start_line, start_col)** și colțul dreapta-jos cu coordonatele **(end_line, end_col)** vor fi suprascriși cu pixeli de valorea **r**, **g**, **b**.

Exemplu

Imaginea Originală:

```
(0, 0, 255) (0, 255, 0) (255, 0, 0)
(0, 128, 0) (0, 255, 0) (0, 128, 0)
```

Instrucțiunea:

```
4 1 0 2 1 100 150 200
```

Imaginea Obținută:

```
(0, 0, 255) (100, 150, 200) (100, 150, 200)
(0, 128, 0) (100, 150, 200) (100, 150, 200)
```

Properietăți argumente

0 ≤ start_line ≤ end_line < Înaltimea curentă a imaginii

0 ≤ start_col ≤ end_col < Lățimea curentă a imaginii

0 ≤ r, g, b ≤ 255

Blur

Format:

```
5 num_iter
```

Efectul de blur presupune înlocuirea fiecărui pixel cu media aritmetică a vecinilor săi (stânga, dreapta, jos, sus). Calculul se va executa separat pentru fiecare canal de culoare, iar rezultatul va fi trunchiat la un număr întreg. Pentru pixelii aflați la marginea imaginii, luați in considerare doar pozițiile din imagine.

Aceasta operație se va repeta de **num_iter** ori.

Exemplu

Imaginea Originală:

```
(0, 0, 255) (0, 255, 0) (255, 0, 0)
(0, 128, 0) (0, 255, 0) (0, 128, 0)
```

Instrucțiune:

```
5 1
```

Imaginea Obținută:

```
(0, 191, 0) (85, 85, 85) (0, 191, 0)
(0, 127, 127) (0, 170, 0) (127, 127, 0)
```

Properietăți argumente

0 ≤ num_iter ≤ 2000

Rotație

Format:

```
6 num_rot
```

În urma operației, imaginea va fi rotită de **num_rot** ori la 90 grade, în sensul acelor de ceasornic.

Exemplu

Imaginea Originală:

```
(0, 0, 255) (0, 255, 0) (255, 0, 0)
(0, 128, 0) (0, 255, 0) (0, 128, 0)
```

Instrucțiunea:

```
6 1
```

Imaginea Obținută:

```
(0, 128, 0) (0, 0, 255)
(0, 255, 0) (0, 255, 0)
(0, 128, 0) (255, 0, 0)
```

Properietăți argumente

1 ≤ num_rot ≤ 3

Fill (bonus)

Format:

```
7 start_col start_line r g b
```

În urma operației, pixelul de la pozitia **(start_line, start_col)** va fi suprascris cu pixeli de valoarea **r**, **g**, **b**. Operația va fi repetată recursiv pentru toți vecinii acestuia (stânga, dreapta, sus, jos) care aveau aceeași culoare cu pixelul original din **(start_line, start_col)**.

Dacă pixelul original avea tot culoarea (r, g, b), operația nu produce nici un efect.

Exemplu

Imaginea Originală:

```
(0, 0, 255) (10, 10, 10) (255, 0, 0)    (10, 10, 10)
(10, 10, 10) (0, 255, 0) (10, 10, 10)  (10, 10, 10)
(10, 10, 10) (10, 10, 10) (10, 10, 10) (10, 15, 10)
```

Instrucțiunea:

```
7 2 1 42 42 42
```

În urma aplicării operației, toți vecinii accesibili din poziția (1, 2), având culoarea (10, 10, 10), au fost suprascriși cu un pixel de valoarea (42, 42, 42). Pixelul de pe poziția (0, 1) a rămas neschimbat.

Imaginea Obținută:

```
(0, 0, 255)  (10, 10, 10)  (255, 0, 0)  (42, 42, 42)
(42, 42, 42) (0, 255, 0)  (42, 42, 42) (42, 42, 42)
(42, 42, 42) (42, 42, 42) (42, 42, 42) (10, 15, 10)
```

Properietăți argumente

0 ≤ start_line < Înaltimea curentă a imaginii

0 ≤ start_col < Lățimea curentă a imaginii

0 ≤ r, g, b ≤ 255

Print

Format:

```
8
```

În urma operației, veți afișa la **stdout** înalțimea și lățimea imaginii curente, respectiv imaginea propriu-zisă. Imaginea va fi afișată în același format ca la pasul inițializare.

Exemplu

Imaginea Originală:

```
(0, 0, 255) (0, 255, 0) (255, 0, 0)
(0, 128, 0) (0, 255, 0) (0, 128, 0)
```

Mesajul afișat la stdout:

```
3 2
0 0 255 0 255 0 255 0 0
0 128 0 0 255 0 0 128 0
```

Exit

Format:

```
0
```

Dacă primește instrucțiunea **exit**, programul se oprește cu succes.

Mesaje de eroare

În situația în care este necesar să semnalați un caz de eroare, afișați la stream-ul dedicat (**stderr**) codul de eroare, eliberați memoria alocată dinamic și apelați funcția **exit** (din **stdlib.h**) cu parametrul EXIT_FAILURE (acest macro este definit tot in **stdlib.h**).

Codurile de eroare sugerate:

- **EPERM** : cod 1 : dacă ați primit o instrucțiune de modificare/afișare a imaginii curente dar aceasta nu a fost inițializată sau o instrucțiune cu un cod de operație nespecificat
- **EINVAL** : cod 2 : dacă ați primit un parametru invalid
- **ENOMEM** : cod 3 : în urma unui apel eșuat de alocare de memorie

Exemplu

```
1 100 0
```

O instrucțiune de inițializare de acest tip este invalidă deoarece parametrii nu respectă proprietățile specificate ($1 \leq$ width). Puteți semnala acest lucru in felul următor:

```
fprintf(stderr, "%d\n", EINVAL);
...
exit(EXIT_FAILURE)
```

Testare

Pentru testarea implementării se va folosi checkerul de aici. Acesta cuprinde toate testele care se află și pe vmchecker, deci testele pe baza cărora tema va fi punctată.

Pentru a facilita debugging-ul, această arhivă cuprinde câteva lucruri ajutătoare, care nu sunt folosite în mod direct în testare. Pentru a vizualiza imaginile pe care se lucreaza, cele generate de soluția fiecăruia, precum și outputurile de referință, arhiva include un mic program în C care citește un fișier de output sub forma cerută în temă și creează pentru fiecare comandă de tip **print** o nouă imagine. Acesta este folosit de către scriptul de testare.

Prin urmare, directoarele **ref_ppm** și **output_ppm** (generat în urma testării), nu sunt direct folosite în testare, ci au scopul de a vedea și vizual, dacă e cazul, diferențele. În schimb, directorul input_ppm este folosit deoarece conține imaginea din care este extrasă matricea de pixeli care reprezintă inputul comenzii de tip **inițializare**. Cu excepția testelor 11 și 20 (vezi mai jos) și a celor care testează mesajele de eroare, niciun test nu conține explicit această operație. Fișierul **real** de input care este redirectat la **standard input**-ul programului este creat *on the fly* de scriptul de testare, prin adăugarea la începutul operațiilor a unei operații de tip 1. Sunt două motive pentru această alegere:

1. matricile de pixeli sunt foarte mari și ar fi mai dificil de observat care sunt de fapt operațiile care se efectuează ulterior.
2. ținându-le separate, putem stoca matricea de pixeli într-un format text foarte simplu, numit PPM; asta permite vizualizarea lor cu orice program de vizualizare a imaginilor care suportă acest format.

Testul 20 este singurul test pe care se va testa dacă programul are *memory leaks*. Motivul pentru care nu se testează pe oricare dintre celelalte teste este acela că rularea cu **valgrind** încetinește foarte mult execuția și ar dura extrem de mult testarea, în mare parte de timp așteptându-se după operații de **I/O**.

Testul 11 pornește de la o imagine foarte mică și construiește o imagine foarte mare prin operații de **resize** și **color**. Rezultatul dorit se poate vedea în imaginile **ref/test11-0.ref.ppm** și **ref/test11-1.ref.ppm**.

Testele **12-19** testează tratarea corectă a erorilor.

Trimitere temă

Tema va fi trimisă folosind vmchecker, cursul **Programarea Calculatoarelor (CB & CD)**.

Formatul arhivei va fi următorul:

1. fișier(ele) sursa `.c`.
2. Un fișier **Makefile** (detalii aici) care să conțină următoarele reguli:
    a. **build**: creează executabilul aferent (numele executabilului: **paint_simulator**)
    b. **run**: rulează executabilul aferent
    c. **clean**: șterge fișierele obiect/executabile create.

3. Un fișier README în care vă descrieți rezolvarea fiecărui task.

1. Arhiva trebuie să fie de tipul **zip**.
2. Inputul se va fi citit de la **stdin (tastatura)**, iar output-ul va fi afișat la **stdout (ecran)**. Testarea se face cu ajutorul redirectării acestora din linia de comandă/bash.

Observații

- Nu folosiți variabile globale.
- Fiți consistenți în ceea ce privește coding style-ul.
- Modularizați pe cât de mult posibil codul.
- Verificați să nu aveți leak-uri sau erori de memorie folosind **valgrind**.
- Apelați fflush(stdout) înainte de a afișa un mesaj la stream-ul standard de eroare (stderr) pentru ca mesajele de la cele doua stream-uri sa nu se intercaleze (mai multe explicații găsiți la [4]).

Listă depunctări

Lista nu este exhaustivă.

- o temă care nu compilează și nu a rulat pe **vmchecker** nu va fi luată în considerare
- o temă care nu rezolvă cerința și trece testele prin alte mijloace nu va fi luată în considerare
- o temă care folosește semnificativ mai multa memorie decât este necesar
- este obligatoriu sa compilați cu opțiunea '-Wall'
- [-1.0]: numele variabilelor nu sunt sugestive
- [-1.0]: linii mai lungi de 80 de caractere
- [-5.0]: abordare ineficientă
    - în cadrul cursului de programare nu avem ca obiectiv rezolvarea în cel mai eficient mod posibil a programelor; totuși, ne dorim ca abordarea să nu fie una ineficientă, de genul să nu folosiți instrucțiuni repetitive acolo unde clar era cazul, etc.

Referințe

[0] https://en.wikipedia.org/wiki/RGB_color_model

[1] http://ocw.cs.pub.ro/courses/programare/laboratoare/lab10

[2] http://valgrind.org/

[3] http://stackoverflow.com/a/4201325