Wrocław, 29-maja-2019

Kamil Paczos, 218377
PT-N-12

Prowadzący: Janusz Biernat

Laboratorium Architektury Komputerów

(5) Przetwarzanie obrazów

5.1 Treść ćwiczenia

Zakres i program ćwiczenia

- Zapoznanie z biblioteką SDL.
- Poznanie struktury plików BMP i metod ich przetwarzania.
- Wykonanie operacji na plikach graficznych i wyświetlenie efektów na ekranie.

Zrealizowane zadania

- Utworzenie programu wyświetlającego wskazany obraz na ekranie.
- Utworzenie programów wykonujących obrót obrazu wokół osi X i Y (odbicie pionowe i poziome).
- Wykonanie programów modyfikujących parametry obrazu: utworzenie negatywu, rozmycie, zmiana nasycenia barw.

5.2 Przebieg ćwiczenia

5.2.1 Program wyświetlający obraz na ekranie

Program wyświetlający wskazany plik graficzny na ekranie zrealizowany został w języku C z wykorzystaniem biblioteki SDL. Na samym początku kodu programu wywoływana była funkcja biblioteczna *SDL_SetVideoMode* z parametrami: rozdzielczość 640x480, głębia 16-bit, SDL_HWSURFACE. Parametry te nie były istotne, gdyż w dalszej części programu funkcja ta była wywoływana już z właściwymi dla danego pliku wartościami. Wywołanie jej na początku było konieczne, by dalej można było użyć funkcji *SDL_VideoModeOK,* służącej do sprawdzenia poprawności parametrów trybu wideo. Gdy funkcja *SDL_SetVideoMode* nie była wywoływana na początku sprawdzenie poprawności powodowało błąd “Segmentation fault”.

Pierwszym etapem wykonanym w celu wyświetlenia obrazu było załadowanie pliku graficznego. W tym celu utworzona została funkcja *Load_image*. Wykorzystując funkcje biblioteki SDL wczytywała ona plik z podanej ścieżki, sprawdzała poprawność jego otwarcia i w przypadku braku błędów zwracała go jako powierzchnię *SDL_Surface.*

Następnym etapem było wywołanie funkcji *SDL_VideoModeOK*. Jej parametrami była rozdzielczość wczytanego obrazu, głębia 32-bit oraz flaga *SDL_SWSURFACE,* wskazująca jako miejsce przechowywania obrazu pamięć systemową zamiast pamięci wideo, jak ma to miejsce w przypadku flagi *SDL_HWSURFACE.* Jeżeli wartość zwrócona przez funkcję wynosiła 0 oznaczało to, że wystąpił błąd i program był przerywany. Sytuacja ta nie miała jednak nigdy miejsca, zwracana zawsze była poprawna wartość 32 oznaczająca zgodność z trybem głębi 32-bit.

Po sprawdzeniu wartości parametrów były one przekazywane do funkcji *SDL_SetVideoMode.* Następnie, jeżeli w pliku graficznym zapisana była paleta barw, była ona ustawiana dla ekranu za pomocą funkcji *SDL_SetColors* z parametrami pobranymi z obiektu powierzchni obrazu.

```
//ustawia paletę, jeśli istnieje
if( image->format->palette && screen->format->palette)
SDL_SetColors(screen, image->format->palette->colors, 0,
        image->format->palette->ncolors);
```

W kolejnym kroku obraz rysowany był na ekranie za pomocą utworzonej funkcji Paint. Zawierała ona wywołanie funkcji *SDL_BlitSurface* służącej do szybkiego kopiowania powierzchni źródłowej na powierzchnię docelową. W tym przypadku obrazek kopiowany był na ekran. Następnie ekran odświeżany był funkcją *SDL_UpdateRect*.

Wykonany program poprawnie wyświetlał zadany plik BMP na ekranie monitora. Stworzony kod został w całości wykorzystany w pozostałych zrealizowanych na tych laboratoriach ćwiczeniach, dlatego jego opis jest pomijany w kolejnych podpunktach.

5.2.2 Program wyświetlający odbicie pionowe obrazu

W programie tym utworzona została dodatkowa powierzchnia zawierająca przetworzony obraz. Po wczytaniu obrazka zarówno powierzchnia źródłowa, jak i przetworzona zostały zablokowane funkcją *SDL_LockSurface.* Następnie wywołana została utworzona funkcja flip.

Funkcja flip składała się z dwóch zagnieżdżonych pętli for. Pętla nadrzędna iterowała od 0 do pionowej liczby pikseli w obrazku. Pętla wewnętrzna miała zakres działania od 0 do poziomej liczby pikseli obrazka. Wewnątrz pętli następowała zamiana miejsc pikseli pomiędzy obrazkiem źródłowym a obrazkiem docelowym. Piksel będący na pozycji pionowej *h* przemieszczany był na pozycję *max.h - h*. Tak więc dla obrazka o wysokości 500 pikseli, kwadrat znajdujący się na pozycji 30 przemieszczany był na pozycję 499 – 30, zatem na miejsce 469 w tablicy pionowych pikseli obrazka docelowego. Struktura pętli wyglądała następująco:

```
for(int h=0; h < original->h; h++){
        for(int w=0; w < original->w; w++){
            pixel = getpixel(original, w, original->h-h); //pobiera piksel
            putpixel(result, w,h,pixel); //umieszcza piksel we właściwym
            miejscu
        }
}
```

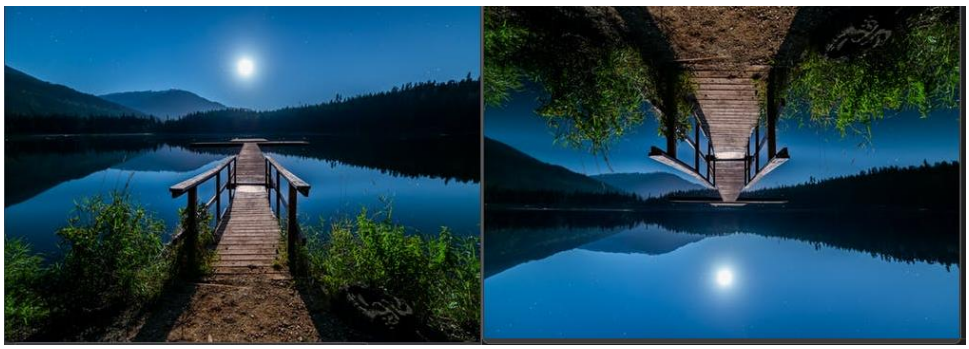
Jak widać na powyższym listingu kodu, do pobierania i umieszczania pikseli w tablicy wykorzystane zostały utworzone funkcje *getpixel* oraz *putpixel*. Pierwsza z nich zwracała piksel znajdujący się na zadanej pozycji x i y w tablicy pikseli powierzchni przekazanej do funkcji. W zależności od ilości bajtów opisujących piksel w danej powierzchni zwracana była odpowiednio przetworzona wartość w postaci liczby całkowitej bez znaku. W przypadku obrazów przetwarzanych w czasie laboratoriów piksele

zawsze opisywane były przez 3 bajty. Kod piksela zwracany był z uwzględnieniem sposobu kodowania składowych koloru:

```
case 3:
    if(SDL_BYTEORDER == SDL_BIG_ENDIAN)
        return p[0] << 16 | p[1] << 8 | p[2];
//składowe koloru (BIG ENDIAN)
    else
        return p[0] | p[1] << 8 | p[2] << 16;
//składowe koloru (LITTLE ENDIAN)
```

Funkcja *putpixel* po ustaleniu ilości bajtów opisujących piksel dokonywała przekodowania liczby całkowitej podanej jako jeden z argumentów na składowe piksela: niebieską, zieloną i czerwoną. Zakodowany piksel umieszczany był w podanej jako argument funkcji powierzchni.

Obie omówione powyżej funkcje zostały wykonane zgodnie ze wzorcem przedstawionym w instrukcji laboratoryjnej. Zostały one wyłączone do oddzielnego pliku .C i wykorzystane w kolejnych opisanych w tym sprawozdaniu programach. W czasie testów wykonanego programu wykryty został błąd, w wyniku którego program kończył swoje działania po przetworzeniu 128 pionowej linii pikseli z komunikatem błędu “Naruszenie ochrony pamięci”. Po analizie debuggerem GDB oraz sprawdzeniu różnych parametrów wykryto, że przyczyną błędu był uszkodzony plik BMP pobrany z wyszukiwarki obrazów Google. Po podmienieniu pliku obrazka na inny program wykonywał się poprawnie, jednak wykrycie przyczyny tego błędu wymagało około 30 minut pracy.



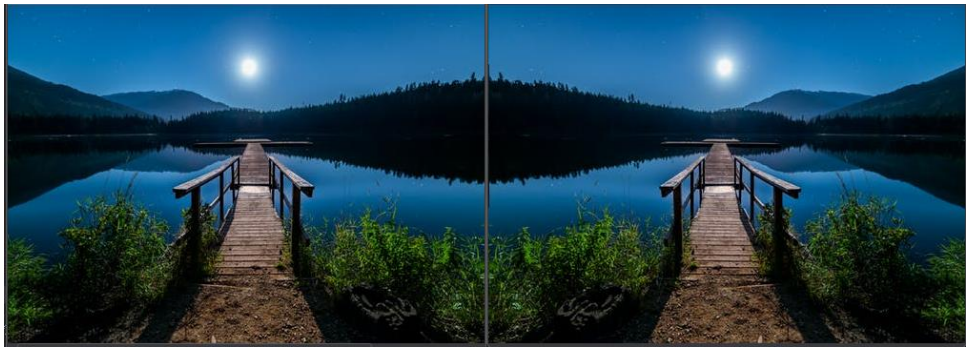
Rysunek 1 Obraz oryginalny oraz odbity w pionie

5.2.3 Program wykonujący odbicie względem osi pionowej (poziome)

Kod tego programu bliźniaczy jest do opisanej wyżej aplikacji wykonującej odbicie w pionie. Jedyna różnica pojawia się w momencie przeniesienia pikseli z obrazu źródłowego do obrazu docelowego. Na pozycji (w,h) umieszczany jest piksel będący na pozycji (w_max – w, h) w obrazie źródłowym.

```
pixel = getpixel(original, original->w-w, h);
putpixel(result, w,h,pixel);
```

Po testach program wykazał prawidłowe działanie.

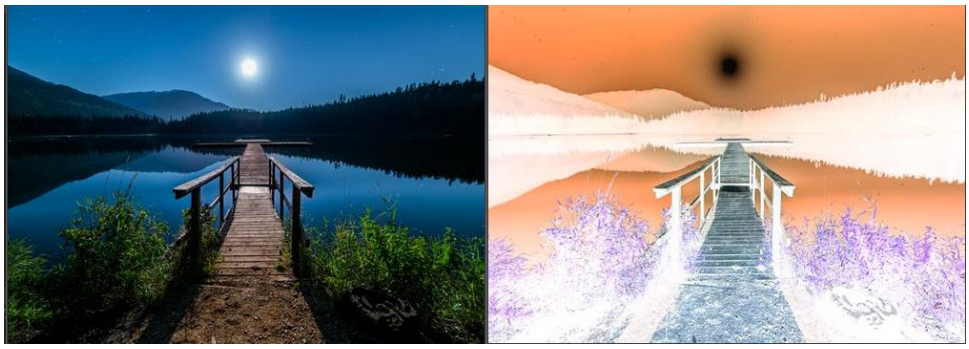


Rysunek 2 Obraz oryginalny oraz odbity w poziomie

5.2.4 Program tworzący negatyw obrazu

Szkielet tego programu identyczny był z przedstawionymi wcześniej programami wykonującymi operację odbicia obrazu. Funkcja *flip* została jednak zastąpiona nową funkcją *negative.* Jej parametrami były dwie powierzchnie: źródłowa - original oraz docelowa – result. Tak jak poprzednio w funkcji zagnieżdżone zostały dwie pętle for iterujące po wysokości i szerokości obrazka. W każdym obiegu pętli wewnętrznej pobierany był jeden piksel o współrzędnych w i h. Następnie jego kod w postaci liczby całkowitej przekazywany był do wstawki języka Assembler wykorzystującej operacje MMX. W pierwszej instrukcja wartość przekazana do wstawki w rejestrze rax przesuwana jest do rejestru mm0. Następnie przygotwywana jest wartość w rejestrze mm1. Jako że uzyskanie negatywu polega na uzupełnieniu kodu piksela do samych jedynek, wykonana jest operacja porównania mm1 z samym sobą, co w rezultacie daje oczywiście same jedynki.

Następnie wykonywana jest instrukcja *PANDN mm1, mm0.* Służy ona do zanegowania bitów w rejestrze mm0, a następnie wykonania operacji logicznej AND na podanych rejestrach. W rezultacie uzyskujemy kod piksela w negatywie, który przesuwany jest do rejestru rax w celu zwrócenia z funkcji. Uzyskany kod funkcją *putpixel* umieszczany jest w powierzchni docelowej result.



Rysunek 3 Obraz oryginalny oraz jego negatyw

5.2.5 Program wykonujący rozmycie obrazu

Program został wykonany po zajęciach. Kod aplikacji opierał się na tym samym szkielecie, który opisany został wcześniej. Funkcja tworząca negatyw zastąpiona została funkcją *blur*, która również przyjmowała dwie powierzchnie jako parametry. Również tutaj zastosowano zagnieżdżone pętle pozwalające na przetworzenie każdego piksela. W tym programie jednak poza bieżącym pikselem

pobierane były piksele na lewo i prawo od niego. Jeżeli piksel był pierwszym lub ostatnim w danym wierszu to odpowiednio jako poprzedni lub kolejny zwracany był również aktualny piksel:

```
pixel = getpixel(original, w, h);
before = w > 0 ? getpixel(original, w-1, h) : pixel;
after = w < original->w - 1 ? getpixel(original, w+1, h) : pixel;
```

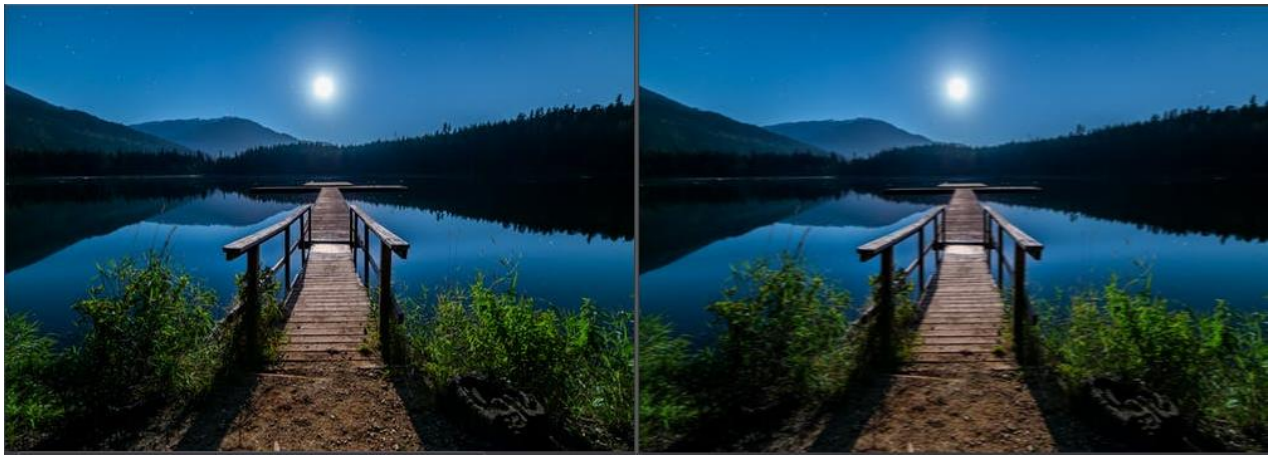
Następnie każdy z pobranych pikseli rozkładany był na części składowe RGB za pomocą funkcji *SDL_GetRGB:*

```
SDL_GetRGB(pixel, original->format, &ri, &gi, &bi);
```

Każda składowa z trzech pobranych pikseli była sumowana, a następnie uśredniana przez podzielenie na 3. W takiej postaci składowe były z powrotem kodowane do liczby całkowitej:

```
pixel = SDL_MapRGB(result->format, r/3, g/3, b/3);
```

Uzyskany w ten sposób piksel umieszczany był w obrazie docelowym, który po ukończeniu przetwarzania wyświetlany był na ekranie.



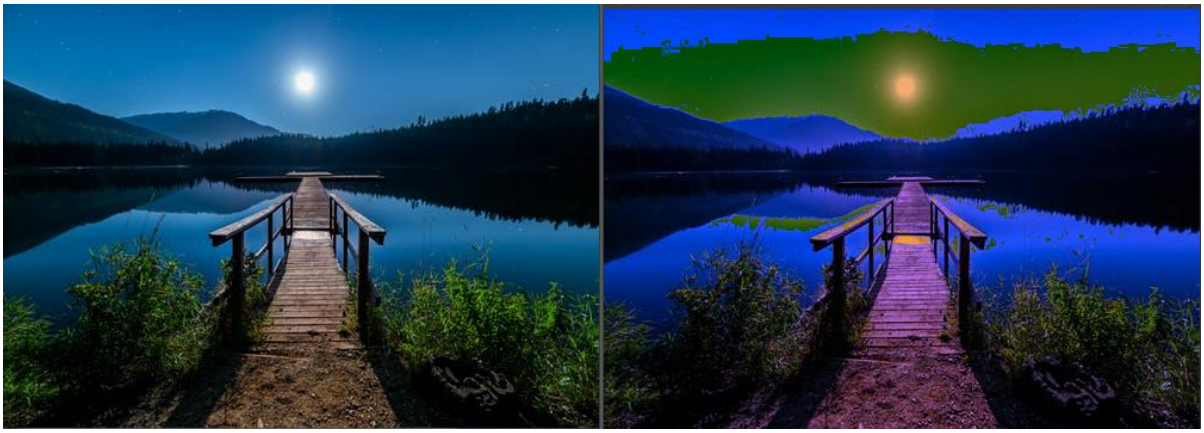
Rysunek 4 Obraz oryginalny oraz rozmyty

5.2.6 Program zmieniający balans kolorów

Program wykonany po zajęciach. Aplikacja ta była modyfikacją tej opisanej w podpunkcie 5.2.5. Funkcja odpowiadająca za rozmycie została zmieniona w celu uzyskania zmiany balansu barw. Ponownie pobierany był tylko aktualny piksel, który rozkładany był na części składowe RGB. Po tym zmieniany był ich balans:

```
r = ri * 1.0; //czerwony - bez zmian
g = gi * 0.7; //zielony - osłabiony
b = bi * 1.5; //niebieski - wzmocniony
```

Tak zmodyfikowany piksel był z powrotem mapowany do kodu całkowitego i umieszczany w obrazie docelowym.



Rysunek 5 Obraz oryginalny oraz zmodyfikowany

5.3 Wnioski

Wykonane zadań wyznaczonych na te laboratoria było bardzo utrudnione ze względu na problemy z serwerem w laboratorium. W ich wyniku około 20% czasu przeznaczonego na wykonanie zadań stanowiło oczekiwanie na “odwieszenie” się komputera laboratoryjnego. Dodatkowym utrudnieniem były błędy w instrukcji dotyczącej laboratoriów. Dla przykładu we wstawce Assembler’a używanej do tworzenia negatywów należało zmienić przekazanie argumentów z użyciem placeholderów na przekazanie bezpośrednio do rejestru rax. Ostatecznie udało się wypełnić wszystkie założenia i zrealizować cele postawione na te laboratoria.

Literatura

References

[1]"SDL Wiki", Wiki.libsdl.org. [Online]. Available: https://wiki.libsdl.org/. [Accessed: 29- May- 2019].

[2]J. Biernat, "Laboratorium AK –ATT asembler (LINUX)", Zak.ict.pwr.wroc.pl, 2019. [Online]. Available: http://zak.ict.pwr.wroc.pl/materials/architektura/laboratorium%20AK2/Linux-AK2-lab-2019%20May.pdf. [Accessed: 29- May- 2019].

[3]R. Blum, Professional assembly language. Indianapolis, IN: Wiley, 2005.