Programowanie Obiektowe i Graficzne
Projekt zespołowy
FilterStudio

Skład zespołu projektowego:
Paweł Chłąd
Daniel Jambor
Bartek Meller

7 lipca 2020

Imię Nazwisko	Odpowiedzialny za	Zrealizował zadania
Paweł Chłąd	Architekurę aplikacji	1A. Rozplanowanie architektury 1B. Implementacja architektury aplikacji 1C. Implementacja filtrów konwolucyjnych
Daniel Jambor	UI/UX	2A. Projektowanie UI/UX 2B. Wdrożenie założeń UI/UX
Bartek Meller	Walidacja danych i dodatkowe elementy UI	3A. Walidacja wprowadzanych danych 3B. Dodatkowe kontrolki użytkownika

1 Opis projektu

FilterStudio to aplikacja pozwalająca na budowanie systemów filtrów do zdjęć. Najczęściej są to filtry konwolucyjne, ale architektura aplikacji pozwala na dołączanie innych rodzajów filtrów. Użytkownik może wprowadzić zdjęcie i zmodyfikować je za pomocą filtrów.

2 Wymagania

Projekt spełnia następujące założenia funkcjonalne:

- Użytkownik może budować filtry konwolucyjne
- Użytkownik może składać wiele filtrów w jeden *projekt*
- Użytkownik może zapisać projekt, wraz z konfiguracjami filtrów
- Użytkownik może wprowadzić obraz do programu aby poddać go obróbce filtrom
- Użytkownik może zapisać przefiltrowany obraz

Założenia niefunkcjonalne są następujące:

- Solidna architektura aplikacji pozwalająca na łatwe wprowadzanie nowych filtrów do programu
- Szybkie wykonywanie filtrów konwolucyjnych poprzez użycie surowego dostępu do danych bitmapy

3 Przebieg realizacji

3.1 Architektura

Aplikacja została z góry zaprojektowana tak, aby współgrała ze wzorcem MVVM. Ważnym było też zapewnienie rozszerzalności projektu o dodatkowe rodzaje filtrów i możliwości budowania owych filtrów. Aby wspierać różne rodzaje filtrów, posłużyliśmy się wzorcem **strategii**, zamknęliśmy filtry za jednolitym interfejsem, dzięki czemu mogliśmy napisać obsługę wielu rodzajów filtrów. Później w trakcie prac okazało się iż potrzebujemy sposobu na dostarczanie *różnych* danych do *różnych* typów filtrów. Problem rozwiązaliśmy poprzez wprowadzenie **fabryki abstrakcyjnej**, dzięki czemu mogliśmy, tak samo jak w przypadku strategii, posługiwać się interfejsem do różnych fabryk, zamiast używać konkretnych obiektów. Następnym problemem było dostarczanie użytkownikom sposobu zmiany istniejących już filtrów (fabryka rozwiązuje tylko problem kreowania ich), zostało rozwiązane to poprzez użycie klasy abstrakcyjnej **FilterDataProviderVM**, która służy jako baza dla klas ViewModel, mających obsługiwać interakcję użytkownika z danymi filtrów. Każdy FilterDataProviderVM jest pewnym sposobem dzięki któremu dane mogą trafić do filtra. Do jednego typu filtra może nawet być przypisane wiele sposobów dostarczania danych, np. filtr gaussowski nadal jest filtrem konwolucyjnym, nie miałoby sensu tworzyć osobnej klasy filtra, który robi to samo co zwykły filtr, ale filtr gaussowski tworzymy inaczej niż filtr zwykły (poprzez określenie parametrów takich jak odchylenie standardowe). Naturalnym więc było, dostarczenie klas odpowiedzialnych tylko za tworzenie i przekazywanie danych do filtrów.

3.2 Filtry Konwolucyjne

Filtr konwolucyjny jest podstawą programu. Jego budowa i działanie są następujące:

Filtr składa się z macierzy 2D, która reprezentuje *maskę* filtru:

$$F = \begin{vmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \\ f_{31} & f_{32} & f_{33} \end{vmatrix} \tag{1}$$

Obraz to tak naprawdę macierz pixeli, którą oznaczymy jako I. Proces nakładania filtru wygląda następująco:

$$F * I = \sum_{dx=-a}^{a} \sum_{dy=-b}^{b} f_{dxdy} \cdot i_{x+dx,y+dy} \tag{2}$$

Gdzie x i y to pixel obrazu który będzie zmieniony po tej operacji. Cała operacja $F * I$ zwróci nam obraz po przejściu przez filtr. Jak wiemy obrazy nie składają się tylko z jednego koloru (u nas RGB) więc tą operację powtarzamy dla wszystkich kanałów osobno.

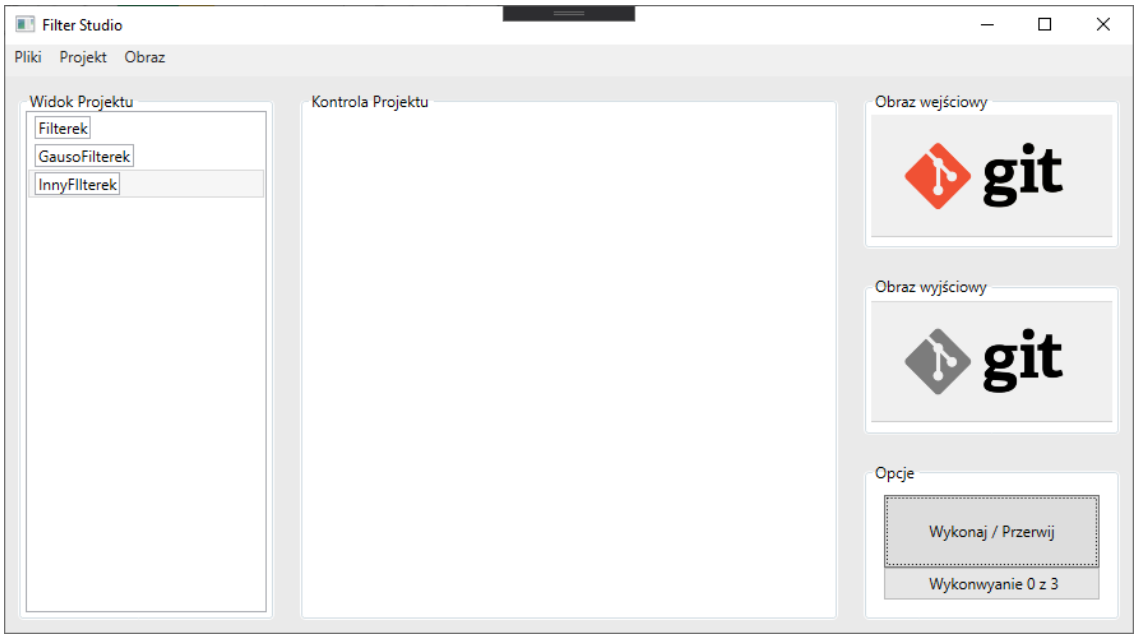
W naszej implementacji zastosowaliśmy także normalizację która opiera się na prostym dzieleniu przez sumę wszystkich elementów filtra. Dzięki temu z filtrem, który nie jest zbalansowany (czyli jego suma nie jest równa 0) nie otrzymujemy prześwietlonych/przyciemnionych obrazów.

Przy używaniu filtrów konwolucyjnych pojawia się problem brzegów. W naszej implementacji pomijamy piksele które wychodzą poza obszar obrazu.

4 Instrukcja użytkownika

Ta sekcja dostarcza informacji na temat użycia programu ***FilterStudio***

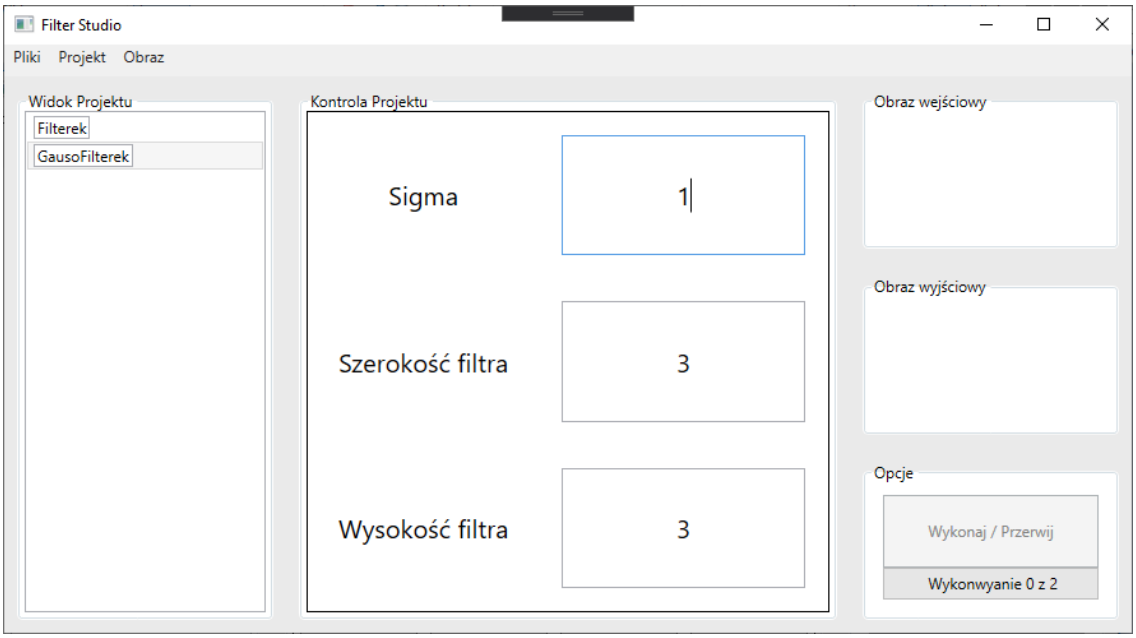
4.1 Overview



- *Widok Projektu* – Wyświetla wszystkie aktywne filtry i umożliwia użytkownikowi wybór filtra do edycji
- *Kontrola Projektu* – Umożliwia edycję filtrów, zobacz sekcje "Filtr Gaussa" oraz "Filtr Podstawowy"
- *Obraz Wejściowy* – Wyświetla miniaturę załadowanego obrazu
- *Obraz Wyjściowy* – Wyświetla miniaturę przetworzonego obrazu
- *Opcje* – Ta sekcja zawiera przycisk umożliwiający renderowanie oraz wskaźnik postępu

4.2 Filtr Gaussa

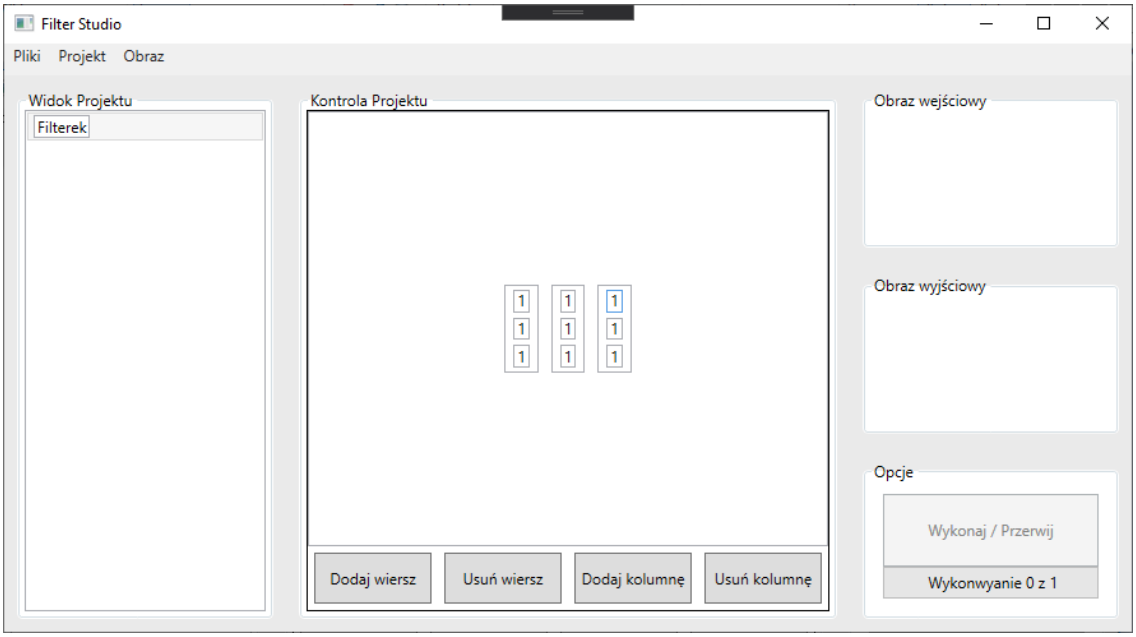
Aby dodać Filtr Gaussa wybierz **Projekt → Dodaj Filtr → Filtr Gaussa**,



i wpisz parametry w pola tekstowe

4.3 Filtr Podstawowy

Aby dodać Filtr Podstawowy wybierz **Projekt → Dodaj filtr → Podstawowy Filtr**,



dostosuj wymiary macierzy za pomocą przycisków w dolnej części panelu **Kontroli Projektu** i wpisz wartości w komórki.

4.4 Filtr czarno-biały

Aby dodać Filtr czarno-biały wybierz **Projekt** → **Dodaj filtr** → **Filtr czarno-biały**

4.5 Wczytywanie, zapisywanie, oraz tworzenie nowych projektów

Wszystkie te funkcje dostępne są w menu **Plik** i działają w ustandaryzowany sposób.

4.6 Wczytywanie i zapisywanie obrazów

Obie te funkcje dostępne są w menu **Obraz** i działają w ustandaryzowany sposób.

5 Podsumowanie i wnioski

Zrealizowaliśmy pełnoprawny projekt w technologii WPF razem z wdrożeniem wielu wzorców projektowych oraz pomyślnie wykorzystaliśmy system kontroli wersji *git*.

5.1 Problemy

Dużym problemem na początku tworzenia aplikacji było oddzielenie wątku wykonywania filtrów od głównego wątku UI i aplikacji. Kod wielowątkowy wielokrotnie wyrzucał wyjątki dot. synchronizacji zapisu/odczytu danych ze zmiennych. Poradziliśmy sobie z problemem poprzez użycie derektyw **lock** oraz poprzez kopiowanie obiektów bitmap do filtrów (tworzyliśmy kopię oryginału znajdującego się w VM), dzięki czemu filtr mógł na osobności operować na bitmapie.

Następnym problemem była optymalizacja filtru konwolucyjnego, problemem były metody GetPixel i SetPixel z klasy Bitmap, te metody, za każdym razem kiedy zostaną wywołane, wywołują tak naprawdę funkcję w gdiplus.dll poprzez mechanizm P/Invoke. Przy bitmapie 1000x1000 wywołujemy P/Invoke milion razy!!! (w sumie to dwa bo jeszcze SetPixel) Problem obeszliśmy poprzez skopiowanie surowych danych bitmapy a następnie operowaliśmy na surowych danych filtrem. Na koniec kopiujemy wszystkie bajty z powrotem do obiektu bitmapy za pomocą klasy Marshall. Wszystko by było dobrze, gdyby nie fakt iż długość jednej linii bitmapy jest zawsze wielokrotnością 4 w bajtach. Więc musieliśmy także uwzględnić padding bitmapy (na szczęście Bitmap udostępnia nam taką informację w właściwości stride) Nasze rozwiązanie jest około dziesięcio-krotnie razy szybsze od Get/SetPixel. Najlepszym sposobem przyspieszenia filtrów wszelkiego rodzaju, byłoby jednak użycie GPU, ale to zostawiamy na przyszłość.

5.2 Dalszy rozwój

Opcji na rozwój programu jest dużo:

- Przeniesienie operacji filtrów na GPU
- Dodanie nowych filtrów
- Dodanie nowych sposobów na tworzenie filtrów
- Rozdzielenie kanałów obrazu, możliwość wpływania na dowolny kanał, drzewo filtrów
- Możliwość kompilacji drzewa do programu standalone (masowe przetwarzanie zdjęć)
- Przetwarzanie filmów (film to seria zdjęć)
- Lepsze UI, przeniesienie projektu na bardziej przyszłościową bibliotekę UI

6 Dodatek - udokumentowanie wykorzystania systemu kontroli wersji

Repozytorium znajduje się pod adresem: https://github.com/Madoxen/FilterStudio