

Imagy

Aplikacja do przetwarzania obrazów
Dokumentacja

Paweł Pozorski

March 18, 2025

Streszczenie

Aplikacja do przetwarzania obrazów, pozwalająca na zastosowanie różnych filtrów oraz analizę histogramu i rzutów. Zaimplementowana w SwiftUI, wspiera dowolne urządzenie z iOS w wersji 16.0+.

Spis treści

1	Wstęp	3
2	Filtryle	3
2.1	Averaging	3
2.2	Binary	4
2.3	Brightness	5
2.4	Contrast	6
2.5	CustomKernel	7
2.6	Emboss	8
2.7	FilmGrain	9
2.8	Gaussian	10
2.9	Glitch	11
2.10	Grayscale	12
2.11	Mosaic	13
2.12	Negative	13
2.13	Posterize	14
2.14	RobertsCross	16
2.15	Saturation	17
2.16	Sepia	18
2.17	Sharpening	19
2.18	Sobel	20
2.19	Vignett	21
3	Histogram	22
4	Rzuty	22
4.1	Pionowy	23
4.2	Poziomy	23
5	Pozostałe funkcjonalności	23
6	Podsumowanie	25
7	Ograniczenia	25
8	Dodatek	26

1 Wstęp

Poniżej opisany projekt jest aplikacją na iOS pozwalającą na zabawę z podstawowymi filtrami graficznymi. Pozwala na odczyt i zapis obrazów z galerii urządzenia, a następnie nakładanie na obraz jednego z kilkunastu wspieranych najpopularniej stosowanych filtrów. Prócz porównywania wizualnego zmian w obrazie, Imagy posiada wbudowany moduł serwujący histogram rozkładu pikseli obrazu oraz jego rzuty pionowe i poziome, poszerzając możliwości analizy zmian wprowadzonych przez transformacje o te 3 obszary.

Imagy jest aplikacją napisaną w Swift, ze wsparciem SwiftUI oraz pakietu SwiftImage. Wspiera wszystkie urządzenia iOS 16.0+ oraz niedawno wydany iPadOS. Została przetestowana na iPhone 16 oraz iPadzie Air w pionowym ustawieniu urządzenia. Aplikacja wymaga od użytkownika dostępu do galerii by poprawnie działać.

2 Filtry

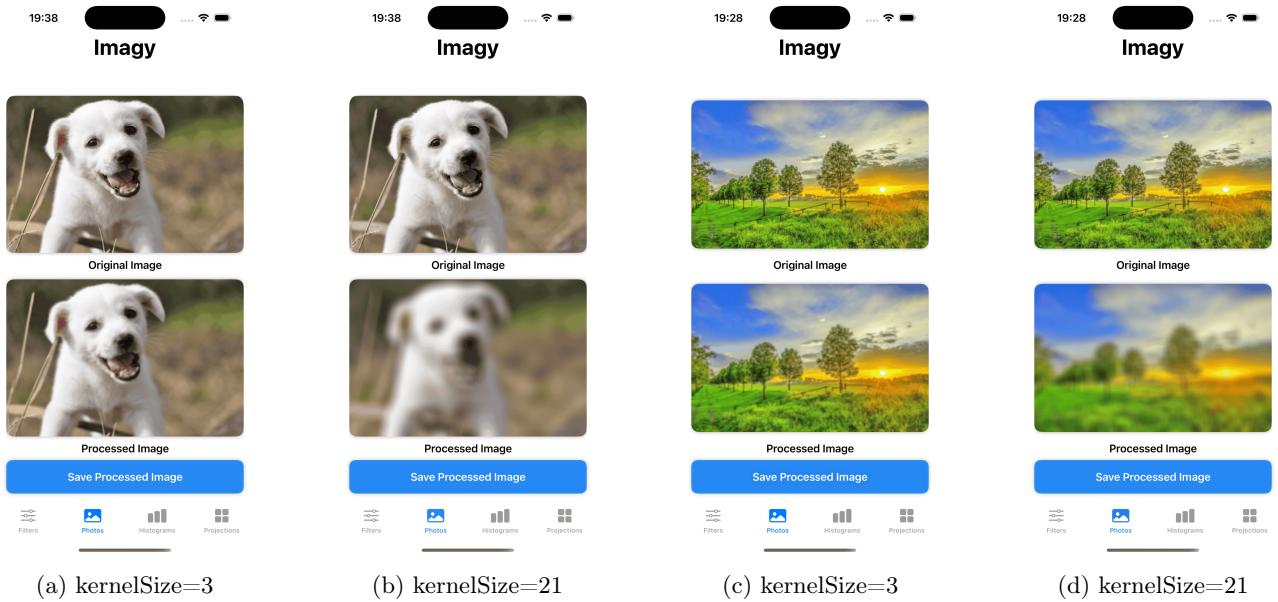
Dla jasności opisu pozostawiam nazwy filtrów w oryginalnym języku angielskim. Wszelkie filtry zostały zaimplementowane w oparciu o ten sam protocol `ImageFilterProtocol`, który zawiera następujące metody i pola:

- `static var name: String` - nazwa filtru.
- `func apply() -> Void` - przyjmuje oryginalny strumień obrazu i zwraca przetworzony.
- `func applyTransformation(
 to inputImage: SwiftImage.Image<SwiftImage.RGBA<UInt8>
) -> SwiftImage.Image<SwiftImage.RGBA<UInt8>` - przyjmuje obraz reprezentowany przez bibliotekę SwiftImage (tj macierz 2D struktury reprezentującej piksel w RGBA) i zwraca przetworzony. Jest to metoda nadpisywana w klasach implementujących ten protocol, zawierająca właściwą implementację filtru.
- `func applyKernel(
 kernel: [[Float]], to inputImage: SwiftImage.Image<SwiftImage.RGBA<UInt8>
) -> SwiftImage.Image<SwiftImage.RGBA<UInt8>` - przyjmuje macierz jądra filtru oraz obraz reprezentowany przez bibliotekę SwiftImage i zwraca przetworzony obraz. Jest to metoda nadpisywana w klasach implementujących ten protocol, zawierająca właściwą implementację filtru. Jej implementacja składa się z następujących kroków:
 - Kopia oryginalnego obrazu.
 - Jądro filtru jest synchronicznie nakładane na kolejne piksele obrazu.
 - Przypadki skrajne (tj piksele na krawędziach obrazu) - interpolacja najbliższymi sąsiednimi pikselami.

Implementacja wspiera filtry o parzystym wymiarze jądra, biorąc wtedy za środek filtra piksel prawy-dolny od środka macierzy jądra. Każdy filtr używający konwolucję nakłada ją na obraz za pomocą tej funkcji.

2.1 Averaging

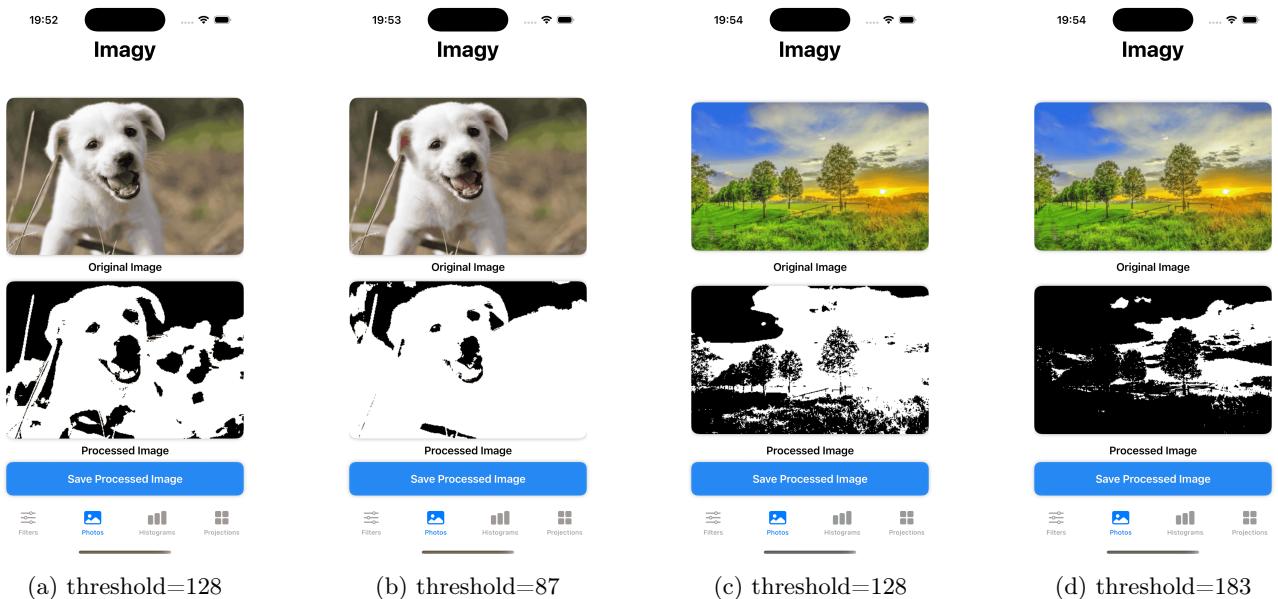
- `static var name: String = "Averaging"`
- `kernelSize: Int` - wymiar jądra filtru, domyślnie 3, pobierany z widoku GUI ograniczony do wartości nieparzystych z zakresu 3 – 21.
- `static func generateAveragingKernel(size: Int) -> [[Float]]` - generuje macierz jądra filtru o wymiarze `sizexsize` zawierającą wartości $1/(size * size)$.
- Celem filtru jest uśrednienie wartości pikseli w zadanym otoczeniu, używany do redukcji szumu i wygła- dzenia obrazu.



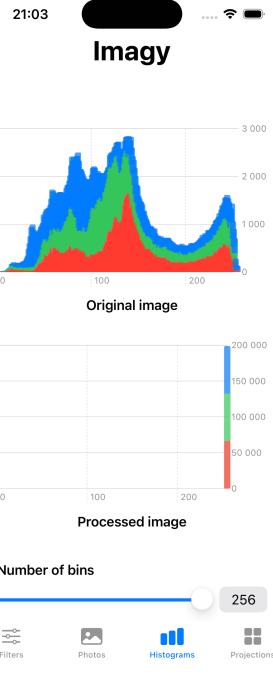
Rysunek 1: Averaging filter - demonstracja. Mniejszy rozmiar kernela zakrywa niektóre detale zdjęć, pozostawia je jednak w miarę czytelne. Duży kernel (21) produkuje kompletnie rozmazany obraz.

2.2 Binary

- static var name: String = "Binary"
 - threshold: Int - wartość progowa, domyślnie 128, pobierana z widoku GUI ograniczona do zakresu 0 – 255. Piksele o wartości mniejszej niż progowana są ustawiane na 0, a większej na 1.
 - Filtr jest używany do wykrywania krawędzi w obrazie, segmentacji obrazu, przetwarzania obrazów w Computer Vision (np wykrywanie tekstu OCR).



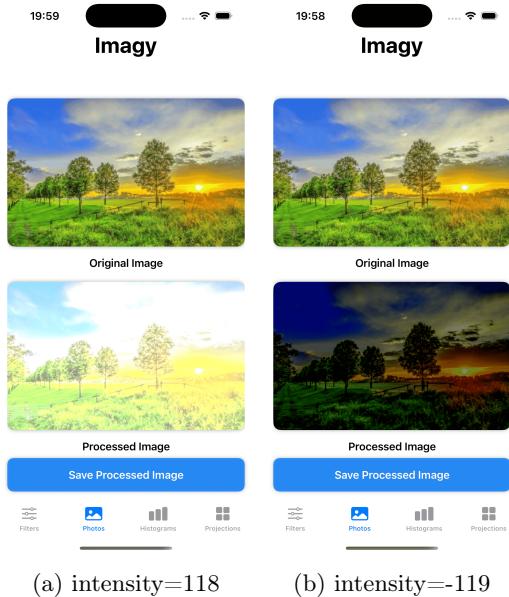
Rysunek 2: Binary filter - demonstracja. Po zdjęciu z psem widzimy że zmniejszenie thresholdu powoduje mapowanie coraz ciemniejszych obszarów na białe, w przeciwieństwie do demonstracji na prawo z drzewami - większy threshold powoduje że nawet część chmur jest mapowana na kolor czarni.



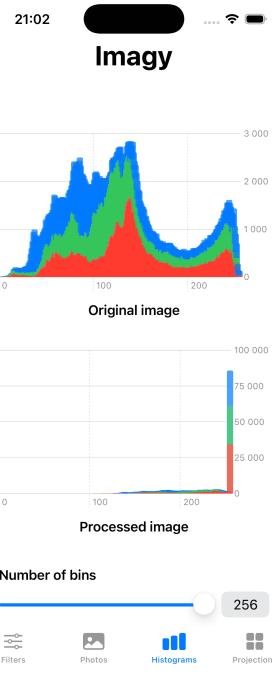
Rysunek 3: Histogram for dog's binary filter with threshold=128.

2.3 Brightness

- static var name: String = "Brightness Correction"
- intensity: Int - wartość jasności, domyślnie 50, pobierana z widoku GUI ograniczona do zakresu –255– 255. Do każdego piksela (kanały wszystkie prócz alfa) jest dodawana jego wartość (oczywiście potem piksel jest ograniczany do zakresu 0 – 255).
- Filtr jest używany do poprawy niedoświetlonych / prześwietlonych obrazów, poprawy widoczności szcze- gółów na ciemnych obszarach oraz w Computer Vision (np wykrywanie tekstu OCR).



Rysunek 4: Brightness filter - demonstracja. Intensyty znacząco dodatnie produkuje obrazek mocno rozjaśniony, wręcz wyblakły. W przeciwnieństwie do znacząco przyjemnionego rysunku na prawo z negatywną wartością intensyty.



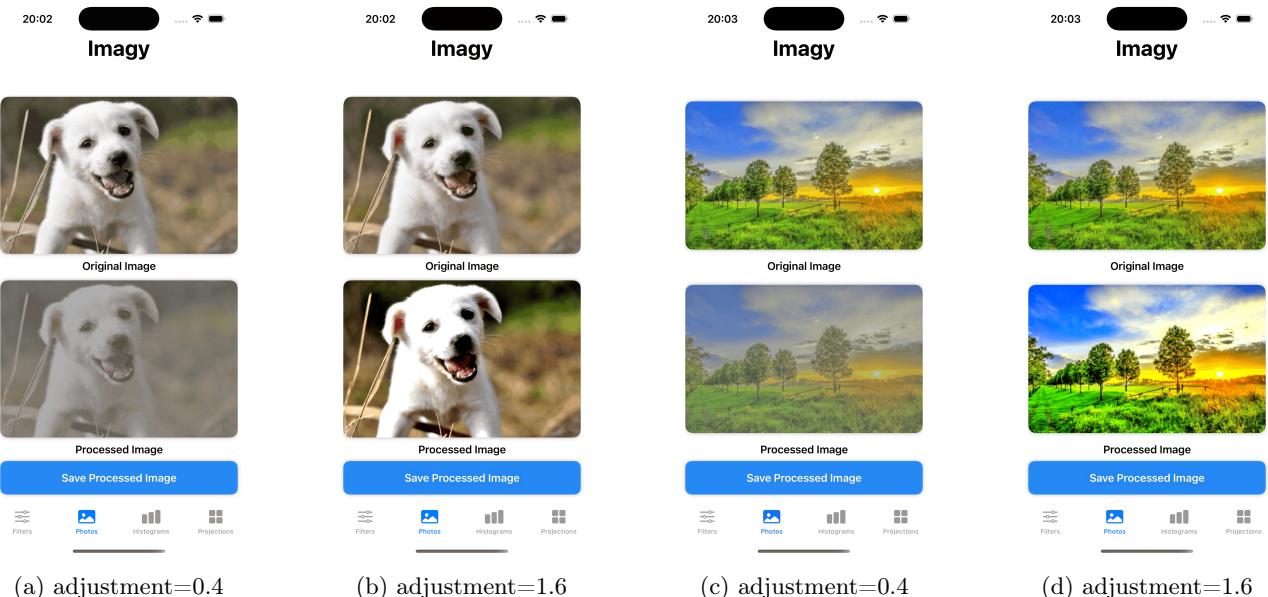
Rysunek 5: Histogram for dog's brightness filter with intensity=100.

2.4 Contrast

- static var name: String = "Contrast"
- adjustment: Double - defaultowo 1.2, ograniczony do zakresu 0.0 – 2.0, pobierany z widoku GUI ze stepem 0.1. Każdy piksel (kanały wszystkie prócz alfa) transformowany jest zgodnie z wzorem:

$$UInt8(\min(255, \max(0, Int((Double(pixel) - 128) * adjustment + 128))))$$

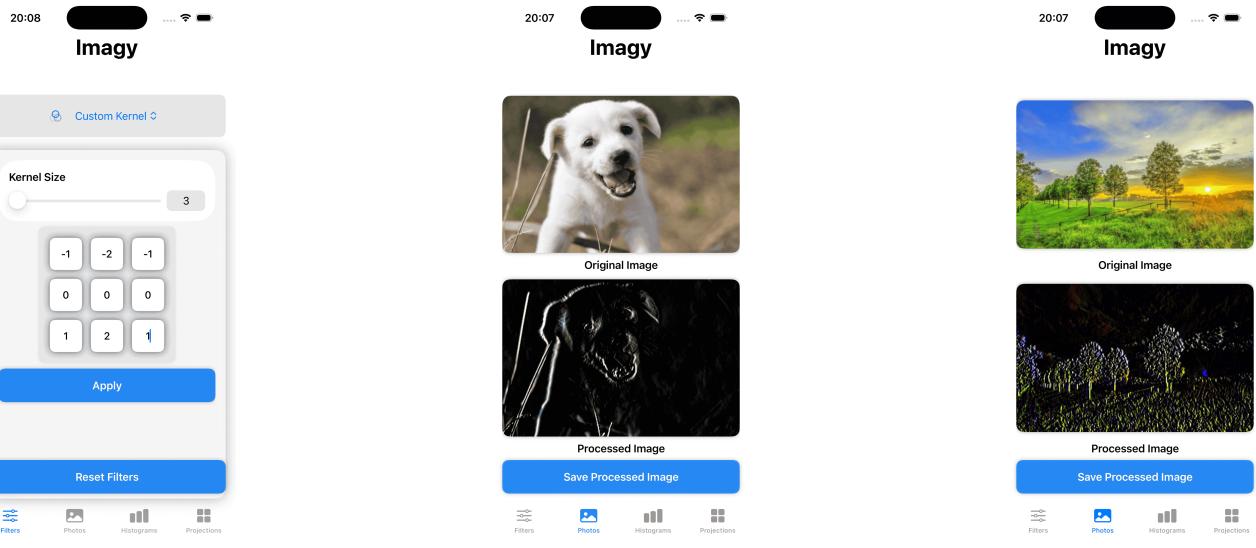
- Filtr zwiększa / zmniejsza różnicę między jasnymi a ciemnymi obszarami obrazu, wynikowo dając obraz o bardziej wyrazistych, stonowanych kolorach.
- Filtr jest używany do poprawy jakości (czytelności w 'złanych' obszarach), stylizacji obrazów (dramatyczny wygląd) czy poprawy ich czytelności.



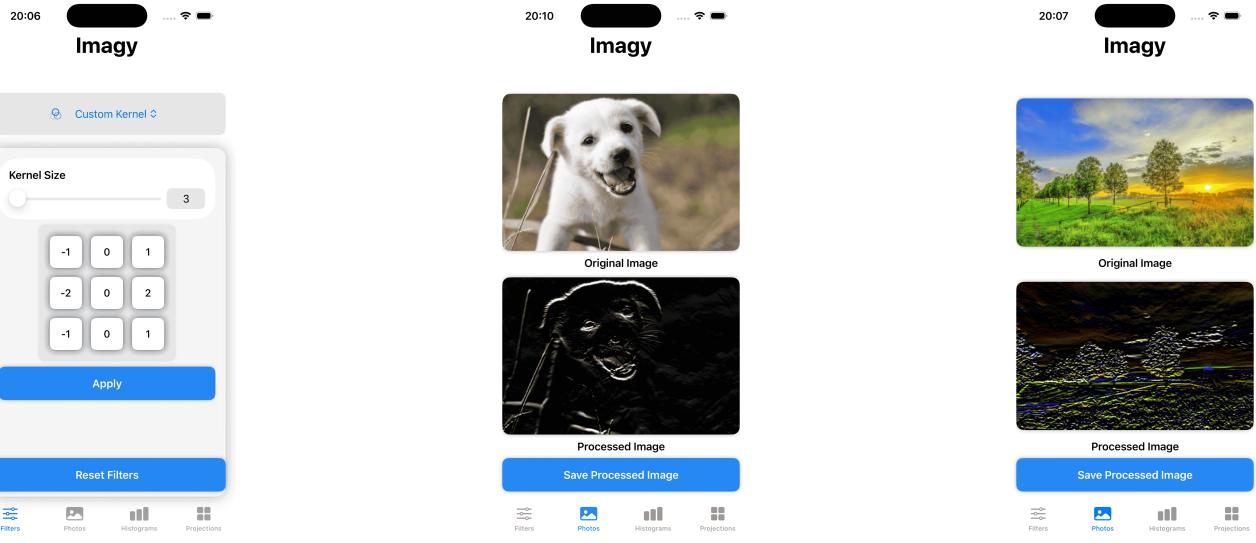
Rysunek 6: Contrast filter - demonstracja. Mały adjustment prowadzi do zdjęć przykrytych szarością, podczas gdy większa wartość 1.6 produkuje zdjęcia o wręcz nienaturalnie żywych kolorach.

2.5 CustomKernel

- static var name: String = "Custom Kernel"
- kernel: [[Float]] - kernel konwolucji pobierany z GUI. Ze względów ograniczeń wielkości interfejsu na telefonach i czytelności, możliwe rozmiary to jedynie 3x3 bądź 5x5. Wartości w gui nie mają ograniczeń prócz wymuszonego typu danych Float.



Rysunek 7: Custom kernel filter - demonstracja. Tutaj lewe zdjęcie pokazuje zastosowany kernel transformacji - Sobel vrtical. Na prawo wyniki.



Rysunek 8: Custom kernel filter - demonstracja. Tutaj lewe zdjęcie pokazuje zastosowany kernel transformacji - Sobel vrtical. Na prawo wyniki.

2.6 Emboss

- static var name: String = "Emboss"
- kernelType: EmbossKernelType - 3 opcje na kernele konwolucji:
 - default

$$\begin{bmatrix} -2 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 2 \end{bmatrix}$$

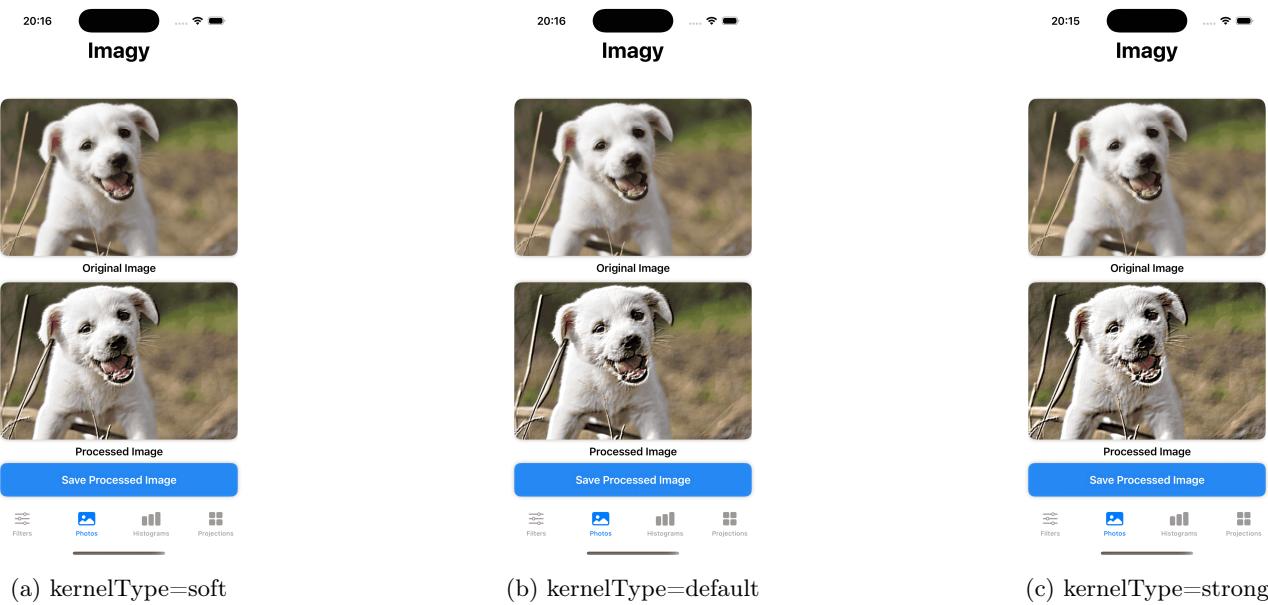
- strong

$$\begin{bmatrix} -1 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

- soft

$$\begin{bmatrix} -1 & -0.5 & 0 \\ -0.5 & 1 & 0.5 \\ 0 & 0.5 & 1 \end{bmatrix}$$

- Filtr ten służy do nadawania obrazom efektu trójwymiarowości przez uwypuklenie krawędzi, tworząc efekt światła i cienia. Używany do podkreślania krawędzi i tworzenia tekstur.



Rysunek 9: Emboss filter - demonstracja. Widzimy różnice pomiędzy siłami kernela, strong powoduje największą transformację, wręcz zbyt dostrzegalną. Wybór opcji soft w tym wypadku może jednak dawać złudzenie trójwymiarowości futra.

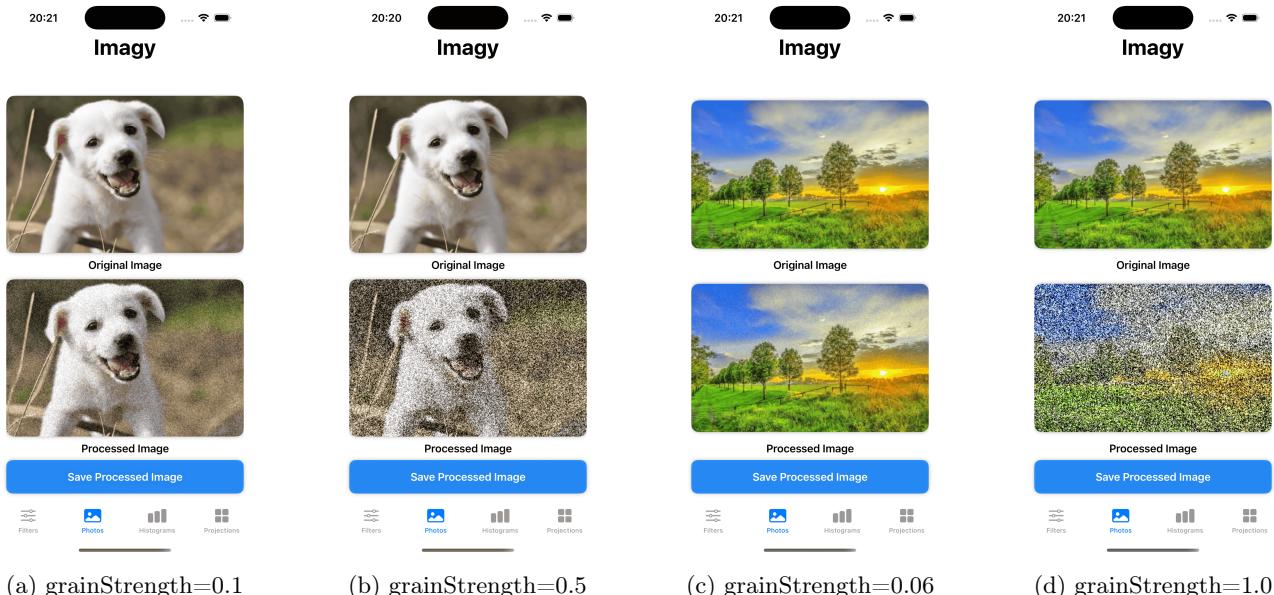
2.7 FilmGrain

- static var name: String = "Film Grain"
- grainStrength: Double - domyślnie 0.2, ograniczony do zakresu 0.0 – 1.0 pobierany z widoku GUI ze stepem 0.01. Dla każdego filtru losowany jest noise zgodnie z

$$Int.random(in : -Int(grainStrength * 255)...Int(grainStrength * 255))$$

a następnie dodawany (ta sama wartość) do każdego kanału (oprócz alpha, oczywiście ograniczając zakres do 0 – 255)

- Jest to podstawowa wersja tego filtru, można go w przyszłości rozszerzyć o rozmiar ziarna, jego zakres czy różne wartości dla poszczególnych kanałów.
- Zastosowanie - fotografia i filmografia artystyczna (styl vintage, atmosfera), augmentacja danych do Computer Vision.



(a) grainStrength=0.1

(b) grainStrength=0.5

(c) grainStrength=0.06

(d) grainStrength=1.0

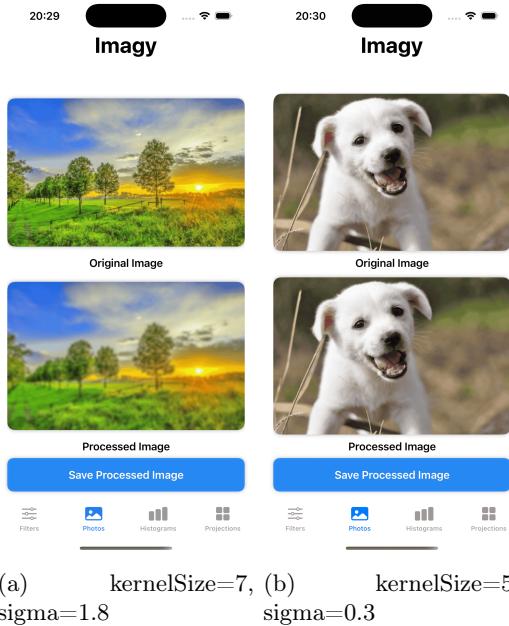
Rysunek 10: Film Grain filter - demonstracja. Wartość 1.0 produkuje obraz kompletnie nieczytelny, ledwo widać co na nim jest. Wartości 0.5 są konkretnie saszumione. Wartości poniżej 0.1 pozostają ledwo dostrzegalne, przypominając zakłuczenia znane ze starych teleskopowych telewizorów.

2.8 Gaussian

- static var name: String = "Averaging"
- kernelSize: Int - wymiar jądra filtru, domyślnie 3, pobierany z widoku GUI ograniczony do wartości nieparzystych z zakresu 3 – 21.
- sigma: Float - odchylenie standardowe rozkładu normalnego domyślnie 1.0, pobierany z widoku GUI ograniczony do wartości z zakresu 0.1 – 3.0 z krokiem 0.1.
- Przykładowe jądro generowane przez program:

$$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

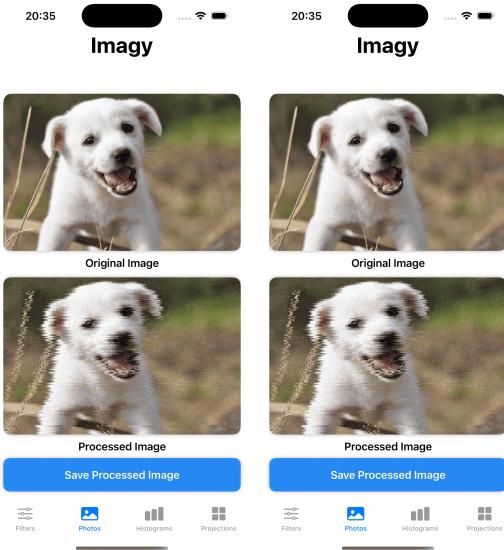
- static func generateGaussianKernel(size: Int, sigma: Float) -> [[Float]] - generuje macierz jądra filtru o wymiarze sizexsize zawierającą wartości losowane wartości z rozkładu normalnego z parametrami zależnymi od odległości od środka kernela, unormalnizowane by cały kernel element-wise się sumował do 1.
- Rozmycie obrazu (używane do odszumiania i zwiększenia jakości poprzez ukrywanie niedoskonałości), używany do przetwarzania obrazów w detekcji krawędzi.



Rysunek 11: Gaussian Blur filter - demonstracja. Na lewo zdjęcie z bardzo mocną transformacją powodującą znaczające rozmazanie się obrazka, jednak w przeciwieństwie do 1d rozmazanie to nie sprawia wrażenia że zdjęcie jest poruszone. Na prawo słabszy kernel, powodujący lekkie zlanie się detali futra psa (trzeba się przyjrzeć by zobaczyć zmienę, choć ludzkie oko od razu widzi że te zdjęcia nie są identyczne).

2.9 Glitch

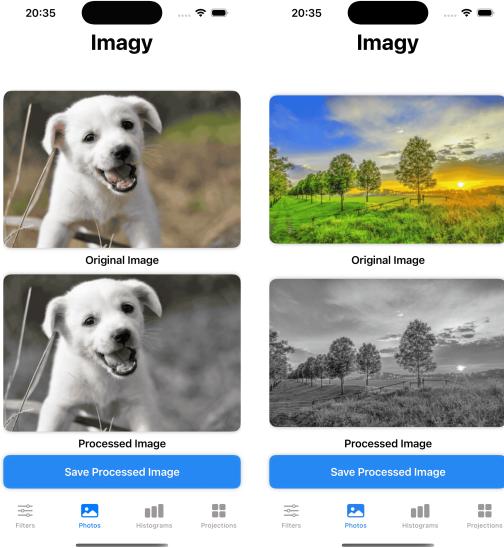
- static var name: String = "Glitch"
- Działanie:
 1. Dla każdego wiersza losujemy wartość z zakresu $-5\dots5$ i zamieniamy te wiersze pozycjami
 2. Dla każdego piksela losujemy (dla każdego z kanałów koloru oddzielnie) wartość z zakresu $-20\dots20$ i dodajemy do wartości piksela (oczywiście ograniczając wartości do $0\dots255$)
- Zastosowanie - tworzenie sztuki, gry video.



Rysunek 12: Glitch Blur filter - demonstracja. Tak jak wcześniej opisane, zdjęcia po przekształceniu sprawiają wrażenie jakby matryca je wyświetlająca była uszkodzona - stąd nazwa.

2.10 Grayscale

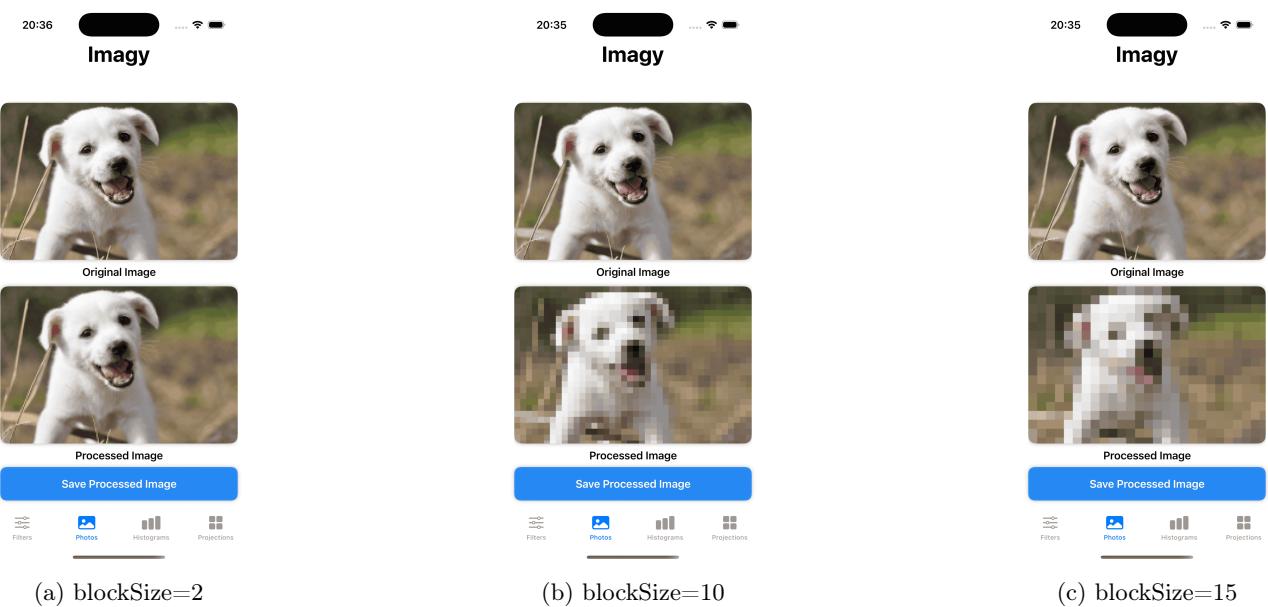
- static var name: String = "Grayscale"
- Działanie - dla każdego z pikseli dodajemy wartości kanałów koloru i dzielimy przez 3. Każdy kanał koloru ustawiamy na tą wartość.
- Zastosowanie - tworzenie monochromatycznych obrazów.



Rysunek 13: Grayscale Blur filter - demonstracja. Transformacja ze zdjęcia RGB robi zdjęcie czarno-białe, z zachowaniem jego jakości (choć w przypadku implementacji mojej aplikacji pod spodem dalej jest reprezentowane przez zdjęcie RGBA, po prostu każdy kanał kolory ma tą samą wartość dla konkretnego piksela.)

2.11 Mosaic

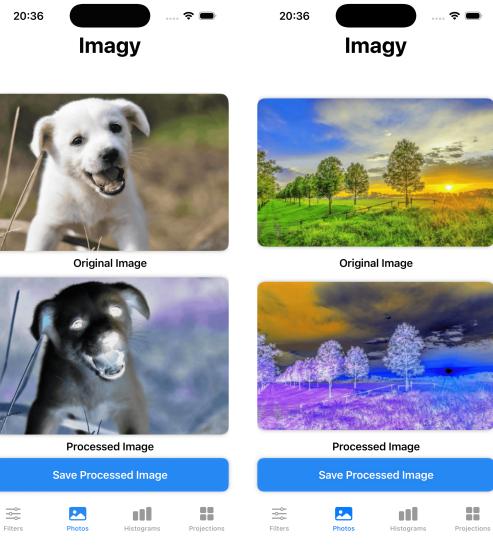
- static var name: String = "Mosaic"
 - blockSize: Int - rozmiar bloku. Defaultowo 10, pobierany z widoku GUI ograniczony do wartości z zakresu 2 – 15 z krokiem 1.
 - Działanie - przesuwamy ustalonej wartością blok (ze stride jako pełny blok, tj tworzymy z obrazu rozbicie na małe bloki ustalonej wielkości), piksele wewnętrz ka dego bloku ustawiamy (dla ka dego z kana ów oddzielnie) na srednia  warto  tego kana u wewn atrz bloku.
 - Zastosowanie - cenzura, efekty artystyczne, stylizacja.



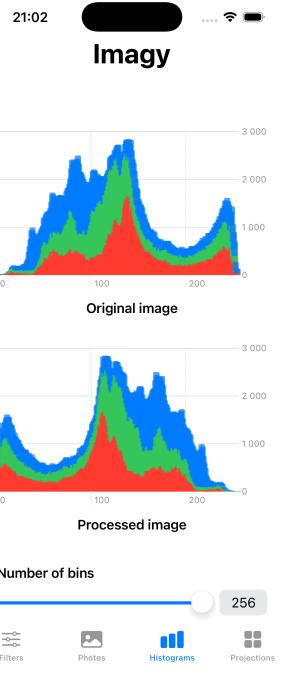
Rysunek 14: Mosaic filter - demonstracja. Większy blockSize powoduje coraz bardziej widoczne bloki tego samego koloru na zdjęciu. W pewnym sensie można go interpretować jako rzut na matrycę o gorszej rozdzielczości. blockSize=2 prezentuje to najmniej drastycznie, dopiero przyjrzenie się ździблom trawy ujawnia w sposób oczywisty, że mamy tu doczynienia z przefiltrowanym zdjęciem a nie wykonanym przez aparat o gorszej rozdzielczości.

2.12 Negative

- static var name: String = "Negative"
 - Działanie - dla każdego z pikseli (każdego z kanałów oddziennie) ustawiamy wartość $255 - x$ gdzie x to wartość poprzednia (oryginalna).
 - Zastosowanie - digitalizacja dokumentów, preprocessing do Computer Vision



Rysunek 15: Nagative filter - demonstracja. Odwrócone kolory prowadzą do zdjęć dla ludzi bardzo nienaturalnych, jednak potrafią podkreślić cechy normalnie ukryte, stąd jego popularność w niektórych algorytmach Computer Vision.



Rysunek 16: Histogram for dog's negative filter.

2.13 Posterize

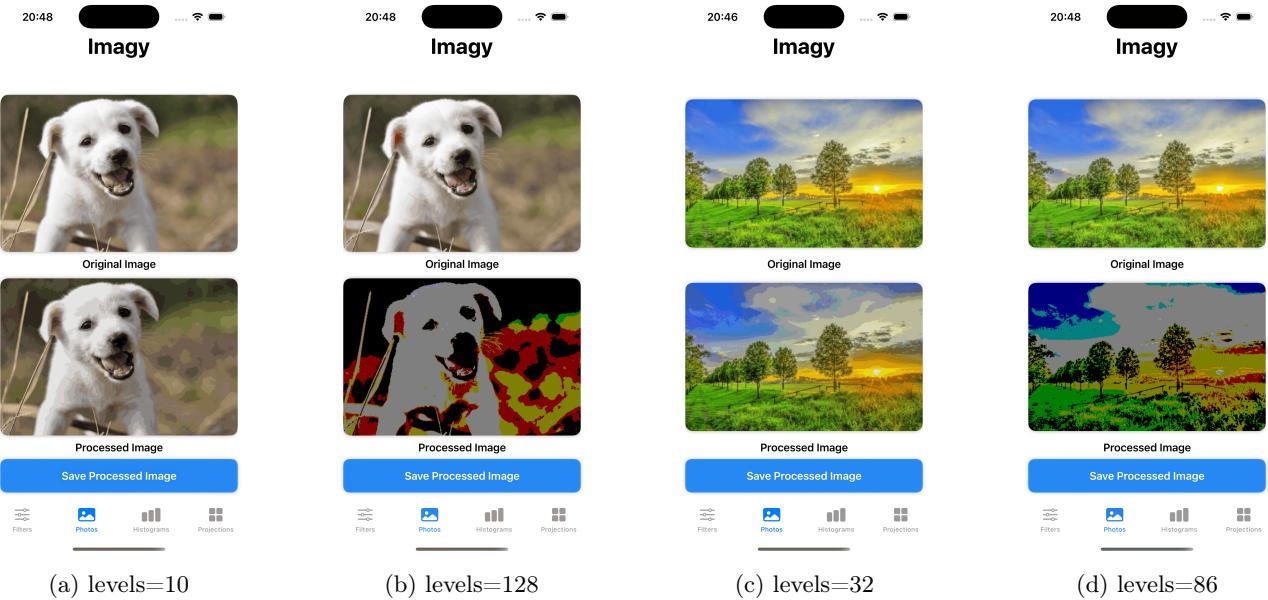
- static var name: String = "Posterize"
- levels: Int - rozmiar bloku. Defaultowo 4, pobierany z widoku GUI ograniczony do wartości z zakresu 2 – 128 z krokiem 2. Większość źródeł wspominających o filtrze nie zaleca więcej niż 32 poziomów, tutaj ograniczenie podniesione w celach demonstracyjnych.

- Każdy piksel, każdy kanał wyliczany zgodnie ze wzorem:

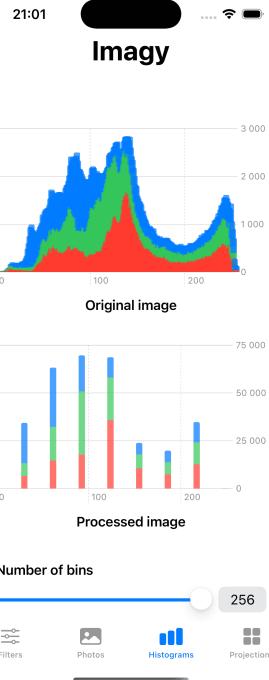
$$N = \frac{255}{P} \times \left\lfloor \frac{C \times P}{255} \right\rfloor$$

Gdzie N - nowa wartość, P - liczba poziomów, C - oryginalna wartość.

- Zastosowanie - efekty artystyczne, segmentacja, gry komputerowe - filtr nadaje obrazowi bardziej uproszczony styl, ponieważ redukuje liczbę dostępnych poziomów jasności.



Rysunek 17: Posterize filter - demonstracja. Więcej niż 32 poziomy prowadzą do zdjęć o mocno zniekształconej kolorystyce (i często kształtach), z pominiętymi detaliами. 32 poziomy na zdjęciu 17c sprawiają, że chmury są bardziej bajkowe. 10 poziomów na zdjęciu 17a również zniekształcają tło do sprawiającego wrażenie jakby było modelowane komputerowo przez nieudolnego grafika.



Rysunek 18: Histogram for dog's posterize filter with levels=32.

2.14 RobertsCross

- static var name: String = "Roberts Cross"
- Mając macierze

$$M_1 = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

oraz

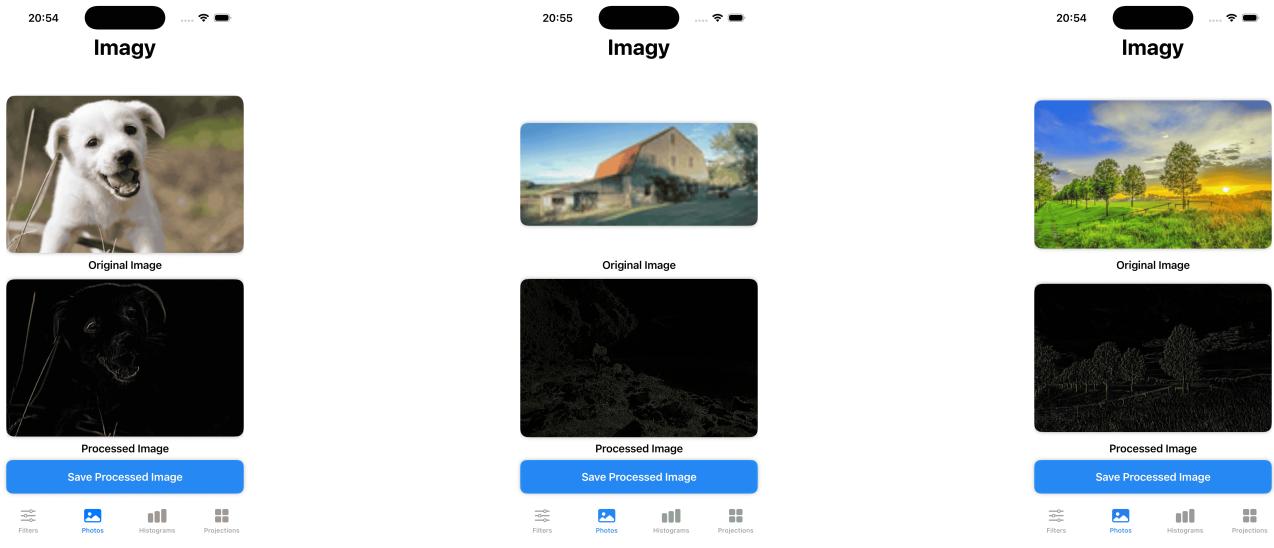
$$M_2 = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

najpierw przeprowadzamy oddzielnie konwolucję obu tych macierzy na całym obrazku otrzymując *gradientX* z macierzy M_1 oraz *gradientY* z macierzy M_2 . Następnie łączymy je (pixel, channel-wise) za pomocą wyliczenia magnitude:

$$N = \min(255, \sqrt{\text{gradXDouble} * \text{gradXDouble} + \text{gradYDouble} * \text{gradYDouble}})$$

gdzie *gradXDouble* jest pikselem z macierzy *gradientX*, *gradYDouble* jest pikselem z macierzy *gradientY*, a N jest ostateczną wartością po filtrowaniu. Pomaga on wykrywać nagłe zmiany jasności.

- Zastosowanie - detekcja krawędzi, segmentacja obrazów, OCR. Często jest używany w Computer Vision zastosowanym do danych medycznych.



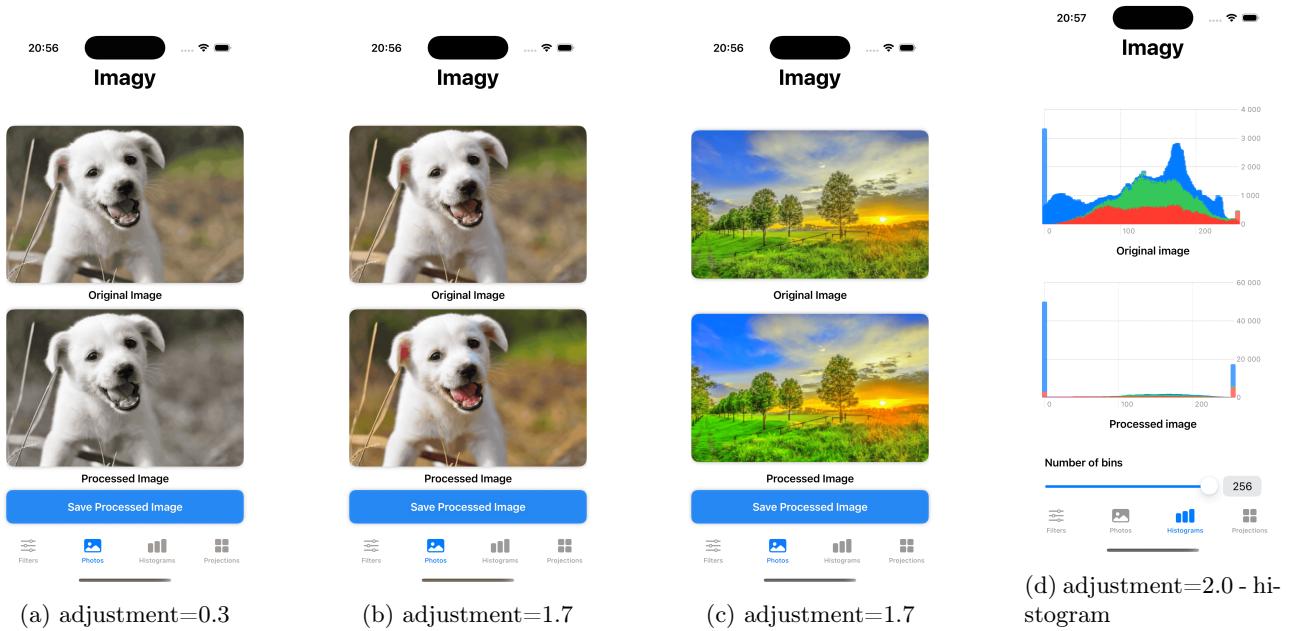
Rysunek 19: Robert Cross filter - demonstracja. Dla zdjęć wyższej rozdzielczości wykrywa poprawnie krawędzie i kontury mniejszych obiektów - w przypadku ?? konturów chmur nie widzimy po zastosowaniu tego filtru. Komplikacje pojawiają się dla zdjęć o bardzo małej rozdzielczości - ?? pokazuje że wtedy filtr ten może zwrócić krawędzie nieznaczącego dla nas szumu.

2.15 Saturation

- static var name: String = "Saturation"
- adjustment: Double - nasycenie koloru - wartość 0.0 prowadzi do koloru szarego, 2.0 to pełna intensywność. Defaultowo 1.2, pobierany z widoku GUI ograniczony do wartości z zakresu 0.0 – 2.0 z krokiem 0.1.
- Działanie - dla każdego pikselu przeprowadzane są następujące operacje:
 1. Wyliczany jest rzut piksela na odcień szarości zgodnie ze wzorem

$$letgrayValue = 0.299 * Double(pixel.red) + 0.587 * Double(pixel.green) + 0.114 * Double(pixel.blue)$$
 2. Wynikowa wartość po trasformacji wyliczana jest ze wzoru (tutaj dla kanału R, ale wzór analogiczny dla pozostałych):

$$newPx.red = UInt8(min(255, max(0, Int(grayValue+adjustment*(Double(oldPx.red)-grayValue)))))$$
- Zastosowanie - fotografia, stylizacja, kinematogradia, gry komputerowe. Zwiększa on intensywność kolorów, wpływając na żywość obrazu.



Rysunek 20: Saturation filter - demonstracja. Małe wartości adjustment prowadzą do zdjęć "oszarzonych", duże - do zdjęć o podkręconym kolorze, aczkolwiek porównując transformację żywego zdjęcia 20c do 20c widzimy w tym drugim bardziej drastyczną zmianę. Z histogramu można wywnioskować, że skrajnie wysoka wartość adjustment ustępuje skrajne wartości pikseli dla wielu z kanału B.

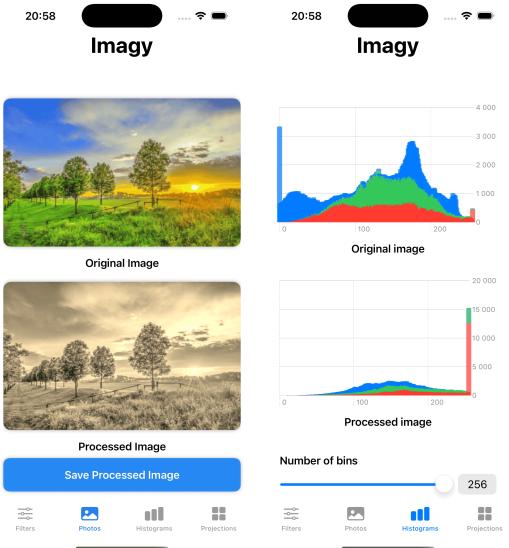
2.16 Sepia

- static var name: String = "Sepia"
- Działanie - nowe wartości kanałów dla kazdego piksela są wyliczane z następującego wzoru (kolejność RGB):

```
let tr = 0.393 * Float(pixel.red) + 0.769 * Float(pixel.green) + 0.189 * Float(pixel.blue)
let tg = 0.349 * Float(pixel.red) + 0.686 * Float(pixel.green) + 0.168 * Float(pixel.blue)
let tb = 0.272 * Float(pixel.red) + 0.534 * Float(pixel.green) + 0.131 * Float(pixel.blue)
```

gdzie tr , tg , tb to nowe wartości, przy czym są one potem ograniczane do zakresu 0..255. Daje to obraz o ciepłych kolorach, nadający klasyczny styl.

- Zastosowanie - digitalizacja dokumentów, preprocessing do Computer Vision



Rysunek 21: Sepia filter - demonstracja. Zdjęcie wynikowe ma ciepłe, vintage barwy. Potwierdza to histogram o kanale R w skrajnie wysokich wartościach i kanale B o rozkładzie mocno w lewą stronę.

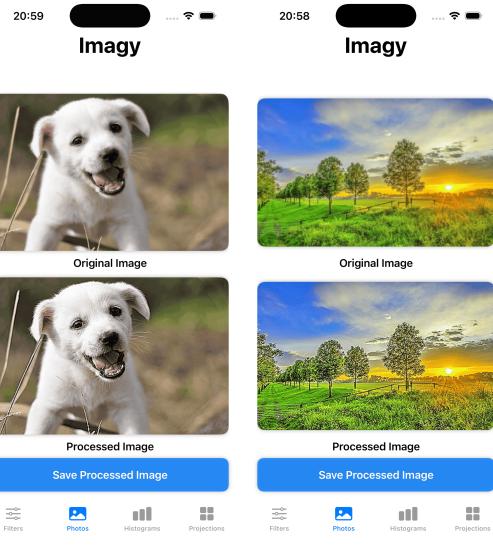
2.17 Sharpening

- static var name: String = "Sharpening"
- Działanie - konwolucja z filtrem

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

- środkowa wartość 5 zwiększa jasność piksela, otaczające go wartości -1 redukują wpływ sąsiednich pikseli. Prowadzi to do zwiększonego kontrastu w miejscach gdzie jasność się zmienia, podkreślając krawędzie.

- Zastosowanie - poprawa detali obrazu, OCR



Rysunek 22: Sharpening filter - demonstracja. Krawędzie są podkreślone. O ile w przypadku zdjęcia z drzewami generuje to nienaturalnie wyglądający obraz, zdjęcie psa wydaje się być po transformacji dużo bardziej szczegółowe.

2.18 Sobel

- static var name: String = "Sobel"
- Mając macierze

$$M_1 = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

oraz

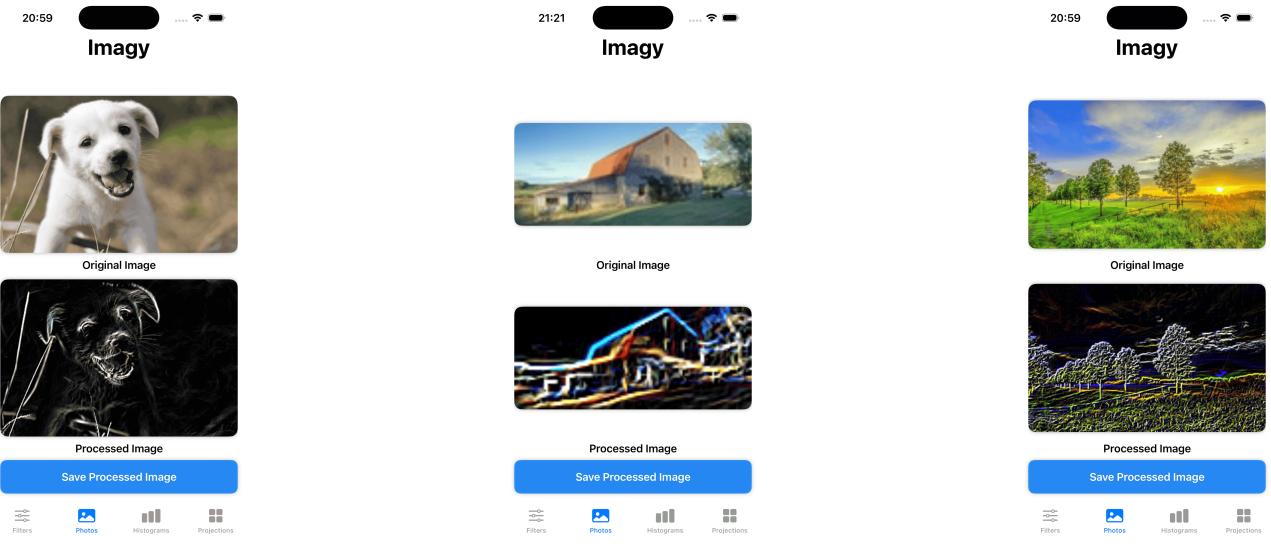
$$M_2 = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

najpierw przeprowadzamy oddzielnie konwolucję obu tych macierzy na całym obrazku otrzymując $gradientX$ z macierzy M_1 oraz $gradientY$ z macierzy M_2 . Następnie łączymy je (pixel, channel-wise) za pomocą wyliczenia magnitude:

$$N = \min(255, \sqrt{gradXDouble * gradXDouble + gradYDouble * gradYDouble})$$

gdzie $gradXDouble$ jest pikselem z macierzy $gradientX$, $gradYDouble$ jest pikselem z macierzy $gradientY$, a N jest ostateczną wartością po filtrowaniu. Jest to przybliżenie wartości gradientu jasności.

- Zastosowanie - detekcja krawędzi, podkreślanie konturów.



Rysunek 23: Sobel filter - demonstracja. W porównaniu z 19 wykrywane jest dużo więcej krawędzi, również dla zdjęć o małej rozdzielczości (co niewątpliwie można traktować jako zaletę względem tamtego). Krawędzie są jednak grupsze, w przypadku wysoce szczegółowych obszarów szybciej się zlewają.

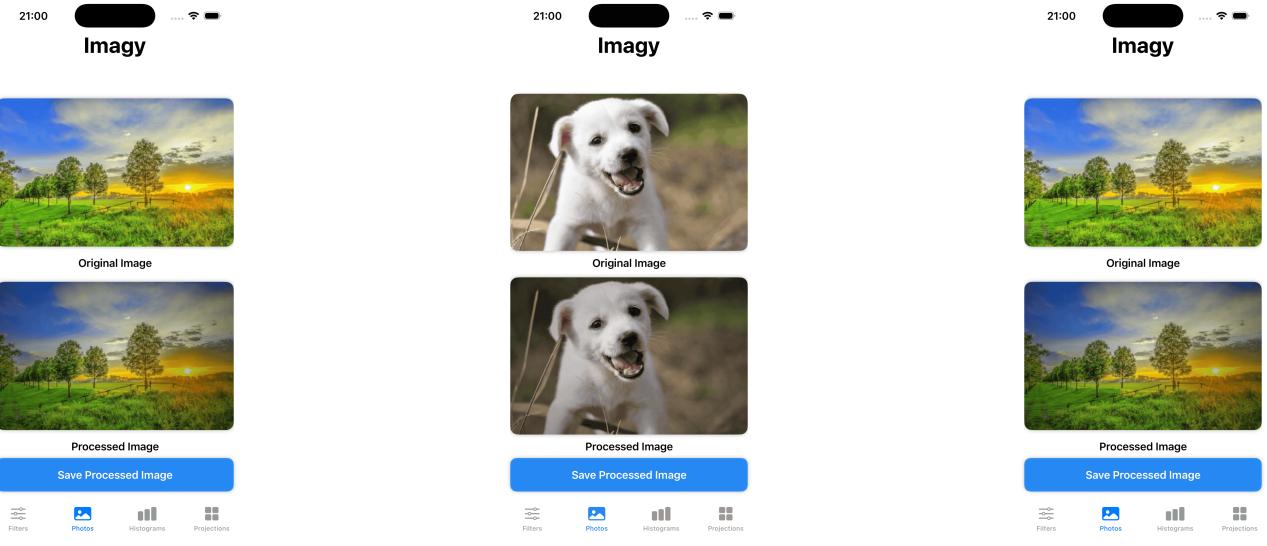
2.19 Vignett

- static var name: String = "Vignette"
- Każdy piksel, każdy kanał wyliczany zgodnie ze wzorem:
 1. Liczymy odległość piksela od środka obrazu.
 2. Każdą z wartości koloru mnożymy przez ten sam współczynnik obliczany ze wzoru

```
let intensityFactor = 1.0 - min(distance/maxDistance, 1.0) * intensity
```

i ograniczamy do zakresu 0...255.

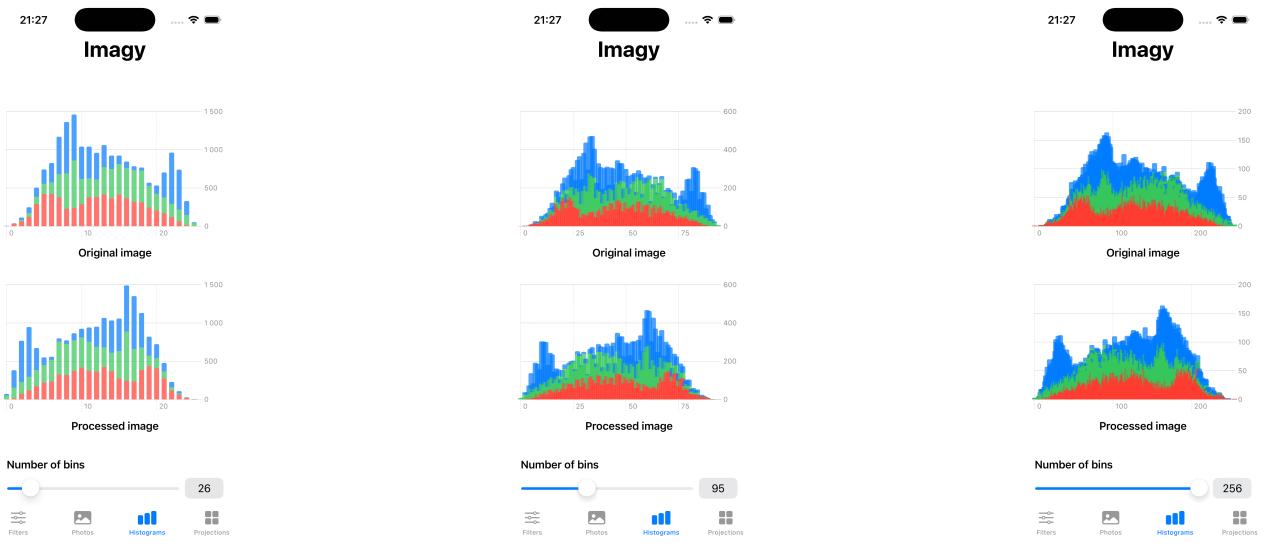
- Należy wspomnieć że jest to jedna z wielu alternatywnych (i raczej prostsza) implementacja tego filtra, np. transformacja sugerowana przez Chat GPT to wyliczanie intensywności z eksponenty z gęstości rozkładu normalnego (tutaj z dodatkowymi hiperparametrami).
- Filtr powoduje stopniowe (w rozumieniu odległości od środka obrazka) przyciemnianie.
- Zastosowanie - efekty artystyczne, kinematografia, rozpoznawanie wzorców (skupienie modelu na środku obrazka).



Rysunek 24: Vignett filter - demonstracja. Im dalej od środka obrazu, tym bardziej przyciemnione są zdjęcia. Potwierdza to histogram na prawo, reprezentujący rozkład pikseli przekształconego zdjęcia psa widocznego na środkowym rysunku.

3 Histogram

Dla każdego kanału, dla każdej wartości piksela (0...255), zliczamy liczbę pikseli w obrazku o tej wartości. Wynik przedstawiamy w postaci barplota.



Rysunek 25: Prezentacja działania histogramu dla tego samego zdjęcia (filtr negatyw na dole), różne ilości binów (widoczne na screenach).

4 Rzuty

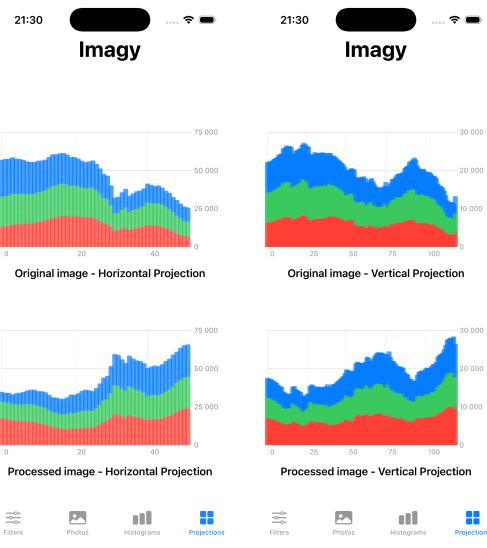
Niech obraz poddawany transformacji ma wymiar $x * y * 3$, gdzie ostatni wymiar reprezentuje 3 kanały koloru.

4.1 Pionowy

Jest on sumą wartości pikseli w każdym wierszu, z podziałem na kanały koloru. Prowadzi nas to do wektora jednowymiarowego rozmiaru $y * 3$

4.2 Poziomy

Jest on sumą wartości pikseli w każdej kolumnie, z podziałem na kanały koloru. Prowadzi nas to do wektora jednowymiarowego rozmiaru $x * 3$

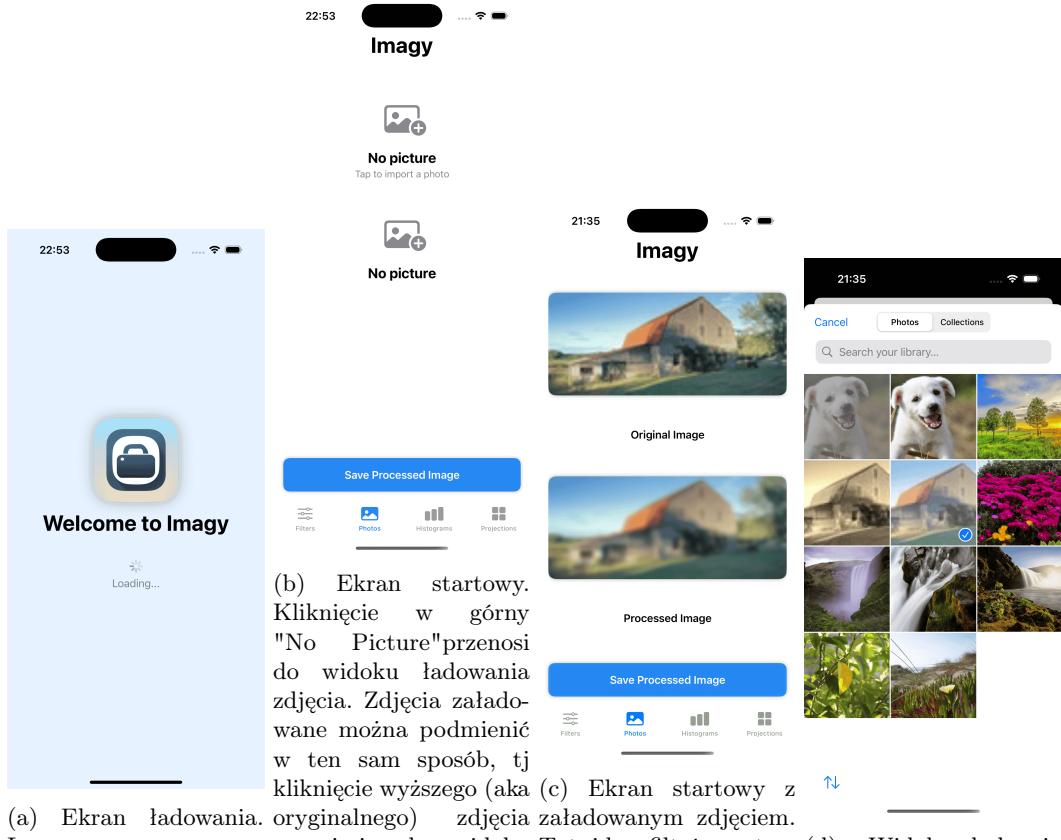


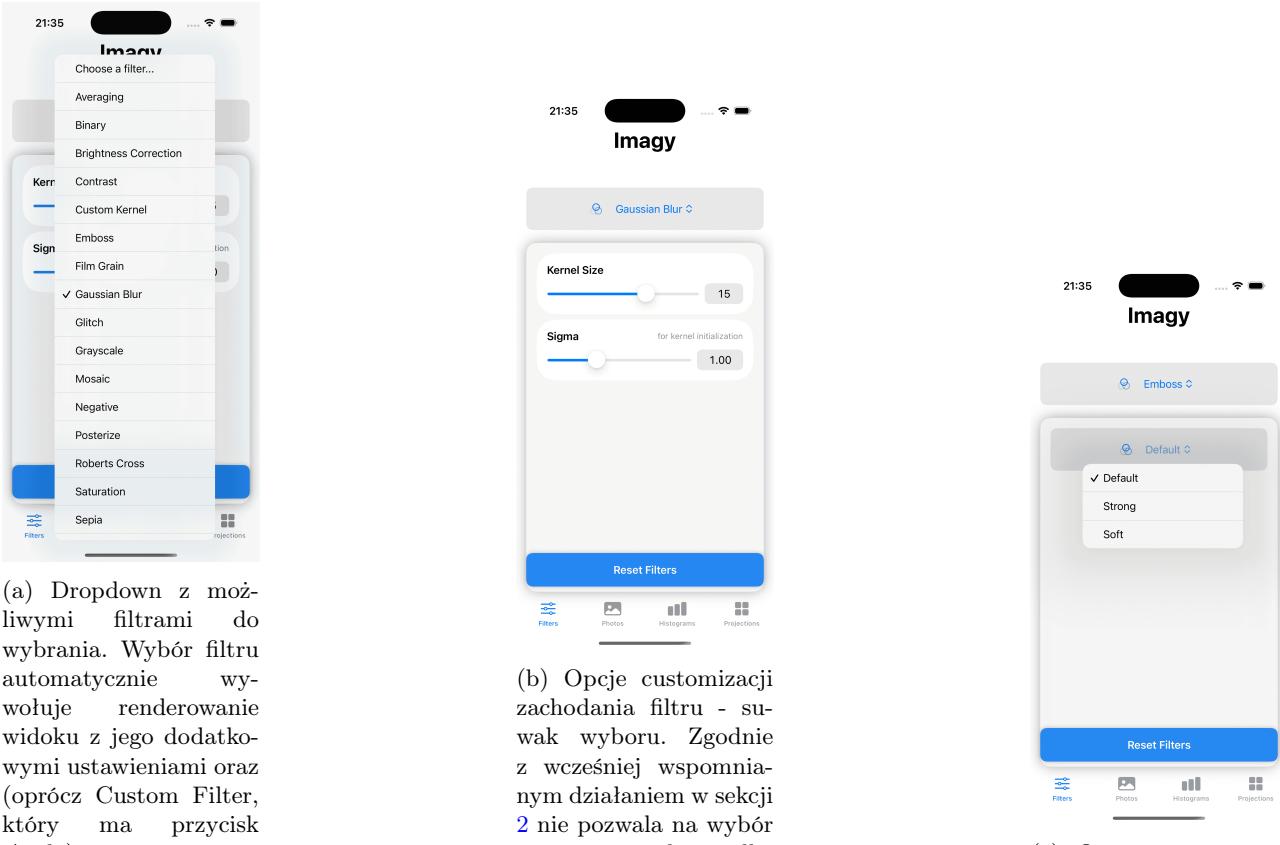
Rysunek 26: Rzuty pionowe i poziome demonstracja działania narzędzia. Liczba binów histogramu to liczba pikseli w danym wymiarze obrazka. Którego rzutu screen dotyczy da się bezpośrednio z niego wyczytać.

5 Pozostałe funkcjonalności

Detalne implementacyjne pozostały funkcjonalności udokumentowane są kodem, nie zostaną tutaj szczegółowo opisane jako iż Swift nie jest tematem przewodnim tego projektu. Prócz funkcjonalności przedstawionych w ramach opisu filtrów Imagy pozwala również na:

- Wczytywanie obrazku do przetworzenia bezpośrednio z galerii (applowe Photos).
- Zapisywanie przetworzonego obrazu do wcześniej wspomnianych Photos.





(a) Dropdown z możliwymi filtrami do wybrania. Wybór filtru automatycznie wywołuje renderowanie widoku z jego dodatkowymi ustawieniami oraz (oprócz Custom Filter, który ma przycisk *Apply*) automatyczne nakładanie filtru na obrazek w tle.

(b) Opcje customizacji zachodania filtru - suwak wyboru. Zgodnie z wcześniej wspomnianym działaniem w sekcji 2 nie pozwala na wybór z continuum lecz tylko na skok o wskazanej dokładności

(c) Opcje customizacji zachodania filtru - dropdown wyboru.

Rysunek 28: Wszelkie powyżej wspomniane funkcjonalności wyboru filtrów są zablokowane na czas przeliczania danych przez aplikację w tle, wtedy są wyświetlane z mniejszą alpha. Dotyczy to operacji aplikowania filtru na zdjęcie, liczenia histogramu oraz rzutów pionowych.

6 Podsumowanie

Imagy jest nowoczesną aplikacją na iOS, spełniającą wszelkie zalecenia Apple odnośnie pisania nowoczesnych, futurystycznych i skalowalnych aplikacji (z wyjątkiem poniżej wspomnianego mankamentu obliczeniowego wynikającego z ograniczeń narzuconych odgórnie na projekt). Implementuje ona kilkanaście podstawowych filtrów powszechnie używanych w fotografii, Computer Vision czy filmografii. Pozwala na zapis i odczyt zdjęć bezpośrednio z urządzenia oraz analizę zmian dokonanych przez daną transformację wewnątrz aplikacji zarówno przez wizualne porównanie zdjęć, jak i poprzez porównanie rozkładu ich pikseli z pomocą rzutu pionowego, poziomego i histogramu.

Aplikacja jest przeznaczona do celów edukacyjnych, powinna być urzywana jedynie ze zdjęciami niskiej rozdzielczości.

7 Ograniczenia

Z powodu ograniczeń czasowych następujące funkcjonalności nie zostały zaimplementowane:

- **Wsparcie dla zdjęć wysokiej jakości (np. 4K)** - istniejąca implementacja ze względu na potrzeby nakładania filtrów 'from scratch' jest zbyt wolna. Wszelkie transformacje są wykonywane na jednym wątku CPU, co sprawia że na urządzeniach mobilnych zdjęcia o rozdzielczości przekraczającej 480p wymagają długich czasów obliczeń, powodując zauważalne opóźnienia.

- **Możliwość nakładania wielu filtrów na jedno zdjęcie** - ze względu na wcześniej wspomniany problem optymalizacyjny oraz duży nakład czasowy wymagający dostosowania GUI do wielu filtrów, została zaimplementowana tylko możliwość zastosowania pojedynczego filtru.
- **Dostęp do większej liczby filtrów.** - z powodu ograniczeń czasowych i przejrzystości systemu, zostały zaimplementowane tylko podstawowe filtry (w liczbie 19).
- **Wyrównywanie histogramu.**

8 Dodatek

Opisany powyżej projekt ze względu na potrzebę zapakowania w GUI pochłonął masę czasu właśnie na ten jego aspekt, skutecznie zmniejszając pulę przeznaczoną na naukę i poszukiwania innych ciekawych metod (oraz ograniczając zaimplementowane do tych wpasowujących się w napisany User Interface). W ramach ciekawostki i rozwinięcia części filtrów tutaj opisanych przekieruję do (niekompletnej) pracy napisanej przezemnie jakiś czas temu - [Computer Vision - Processing](#).