POLITECHNIKA LUBELSKA

Wydział Elektrotechniki i Informatyki

Kierunek Informatyka

PRACA INŻYNIERSKA

Opracowanie aplikacji służącej do filtrowania szumów w obrazach wykonanych kamerką telefonu

Dyplomant:

Robert Korulczyk

Nr albumu:

061323

Promotor:

dr inż. Wojciech Surtel

Lublin 2012

Składam serdeczne podziękowania
dr inż. Wojciechowi Surtelowi
za poświęcony czas i pomoc
w przygotowaniu niniejszej pracy

Spis treści

1. Wstęp

Szybki rozwój techniki oraz postępująca miniaturyzacja sprawiły, że coraz powszechniejsze stało się posiadanie smartfonów. Miniaturowe urządzenia, mieszczące się w kieszeni, coraz bardziej przypominają swoją funkcjonalnością komputer. Popularyzacja tego typu urządzeń pociągnęła za sobą również rozwój systemów na platformy mobilne.

W niniejszej pracy poruszono dwa zagadnienia. Pierwszym z nich jest tworzenie oprogramowania na system Android. Wydany kilka lat temu system stworzony przez Google, bardzo szybko zdobył popularność i jest obecnie wiodącą platformą na nowoczesne telefony komórkowe. Udostępniony zestaw narzędzi dla programistów oraz obszerna dokumentacja znacznie ułatwiają tworzenie oprogramowania na nowy system. Dzięki udostępnionemu API (ang. Application Programming Interface) można korzystać z wielu gotowych funkcji, które ułatwiają na przykład korzystanie z wbudowanego w telefon sprzętu. Rozwój miniaturyzacji pozwolił również na umieszczenie w telefonie wielu urządzeń rozszerzających jego funkcjonalność, jak na przykład odbiornika GPS (ang. Global Positioning System), żyroskopu, modułu Wi-Fi czy miniaturowej kamery. Dostęp do szerokopasmowego internetu umożliwia korzystanie z wielu usług multimedialnych, takich jak VOD (ang. Video on Demand) czy też VoIP (ang. Voice over Internet Protocol).

Drugim zagadnieniem poruszonym w pracy jest usuwanie szumów z obrazów cyfrowych. Znanych jest wiele sposobów na redukcję zniekształceń towarzyszących cyfryzacji obrazów. Różnią się one zarówno trudnością w implementacji, złożonością obliczeniową, jak i również skutecznością w usuwaniu szumów i stopniem zniekształcenia przetwarzanego obrazu. Najskuteczniejsze pod tym względem są filtry wykorzystujące medianę, które dobrze radzą sobie z usuwaniem szumu, jednocześnie nie rozmywając krawędzi filtrowanych obiektów. Nie bez znaczenia jest też format, w jakim zapisywany jest przetworzony obraz. Wykorzystywany najczęściej do zapisu zdjęć format JPEG jest formatem stratnym, co znaczy, że podczas zapisu cześć informacji jest bezpowrotnie tracona, a więc sam zapis może wprowadzać drobne zniekształcenia do obrazu.

W pracy podjęto się opracowania aplikacji dedykowanej na system Android, która umożliwiałaby zrobienie zdjęcia, a następnie wykorzystując filtr medianowy, jego odszumienie. Jej celem jest zredukowanie zniekształceń, które są skutkiem niskiej jakości matryc umieszczanych zwykle w telefonach komórkowych.

2. Cel i zakres pracy

Celem niniejszej pracy jest opracowanie aplikacji na system Android umożliwiającej redukcję szumów z obrazów robionych kamerką telefonu. Zakres pracy obejmuje następujące zagadnienia:

- Przegląd literatury na temat architektury i sposobu tworzenia oprogramowania na system Android;
- Przegląd literatury na temat sposobów przetwarzania obrazów cyfrowych, a w szczególności dotyczącej sposobów usuwania szumów z obrazów;
- Wybór optymalnego algorytmu redukującego szumy w obrazach;
- Stworzenie aplikacji umożliwiającej zrobienie zdjęcia za pomocą aparatu w telefonie z systemem Android;
- Implementacja wybranego algorytmu do odszumiania zdjęć do wcześniej stworzonej aplikacji;
- Optymalizacja aplikacji pod kątem telefonów komórkowych;
- Testy aplikacji.

3. System Android

Android to otwarty system operacyjny dla urządzeń mobilnych (telefony komórkowe, smartfony, tablety PC i netbooki), rozwijany przez firmę Google. Android oparty jest na jądrze Linuksa oraz oprogramowaniu na licencji GNU. Jest obecnie najpopularniejszą platformą mobilną.

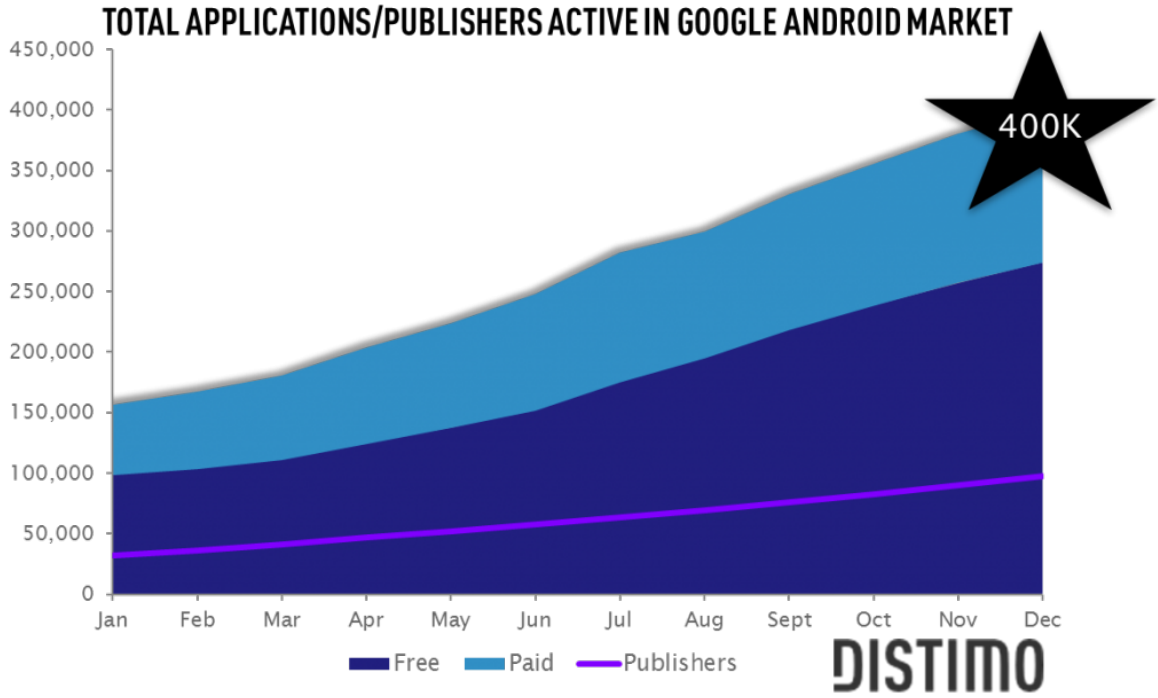
3.1. Historia systemu Android

Android był początkowo tworzony i rozwijany przez firmę Android Inc., która w 2005 roku została wykupiona przez Google. W listopadzie 2007 roku powstał Open Handset Alliance (OHA) – założony przez Google sojusz zrzeszający 34 firmy związane z branżą telekomunikacyjną, którego zadaniem było wspieranie otwartych standardów dla urządzeń mobilnych [VIII]. Wraz z założeniem OHA (który od tej pory zajmował się rozwojem Androida) została udostępniona wczesna wersja systemu Android. Kilka dni później udostępniono SDK (ang. Software Development Kit) – zestaw narzędzi dla programistów, dzięki którym każdy mógł napisać swoją aplikację na nowy system. Pierwsza komercyjna wersja Androida została udostępniona we wrześniu 2008 roku, w niespełna rok po opublikowaniu wersji beta. Pierwszym telefonem wyposażonym w system operacyjny stał się HTC Dream wyprodukowany przez firmę HTC.

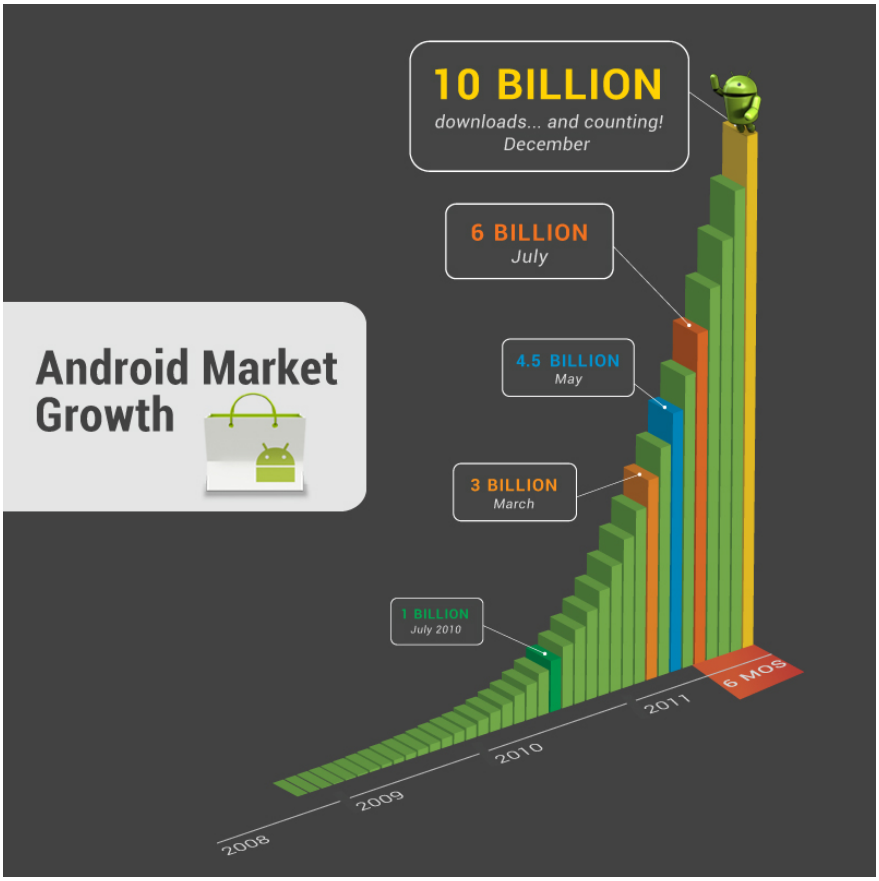
Nowy system w dość szybkim tempie zyskiwał nowych użytkowników. Znaczny wpływ na to miała duża ilość oprogramowania na nową platformę – praktycznie każdy mógł stworzyć swoją aplikację i umieścić ją na Android Market, skąd inny użytkownik Androida mógł ją ściągnąć i zainstalować na swoim telefonie. Google również zachęcało do tworzenia aplikacji na swój system – niedługo po opublikowaniu wersji beta firma ogłosiła konkurs na najbardziej innowacyjne aplikacje na Androida, z łączną pulą nagród sięgającą 10 milionów dolarów [IX]. Jak widać na rys. 3.1 i 3.2 zarówno ilość aplikacji na Android Market, jak i ich popularność wciąż rośnie. Dostępnych jest już ponad 400 tysięcy aplikacji [II], a ilość ich pobrań przekroczyła 10 miliardów [III].

Obecnie Android jest najpopularniejszym systemem na smartfonach, a jego popularność wciąż rośnie. 21 grudnia 2011 roku Andy Rubin, jeden z głównych współtwórców Androida, ogłosił, że ilość dziennie rejestrowanych urządzeń z systemem Android przekroczyła liczbę 700 tysięcy [VII]. Jak widać na rys. 3.3 zainteresowanie

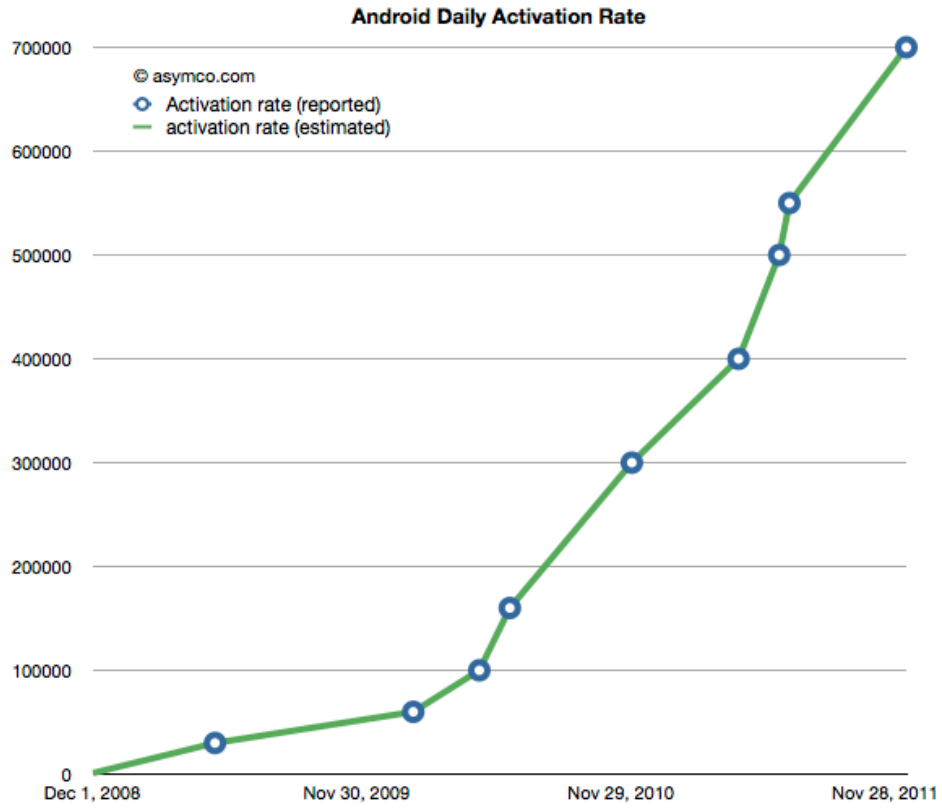
Androidem wciąż rośnie, a ilość urządzeń pracujących na Androidzie szacuje się na blisko 250 milionów, co czyni go numerem jeden jeśli chodzi o platformy smartfonowe [IV].



Rys. 3.1 Ilość aplikacji dostępnych na Android Market [II]



Rys. 3.2 Ilość pobrań aplikacji z Android Market [III]



Rys. 3.3 Ilość dziennie aktywowanych urządzeń z Androidem [IV]

3.2. Architektura systemu Android

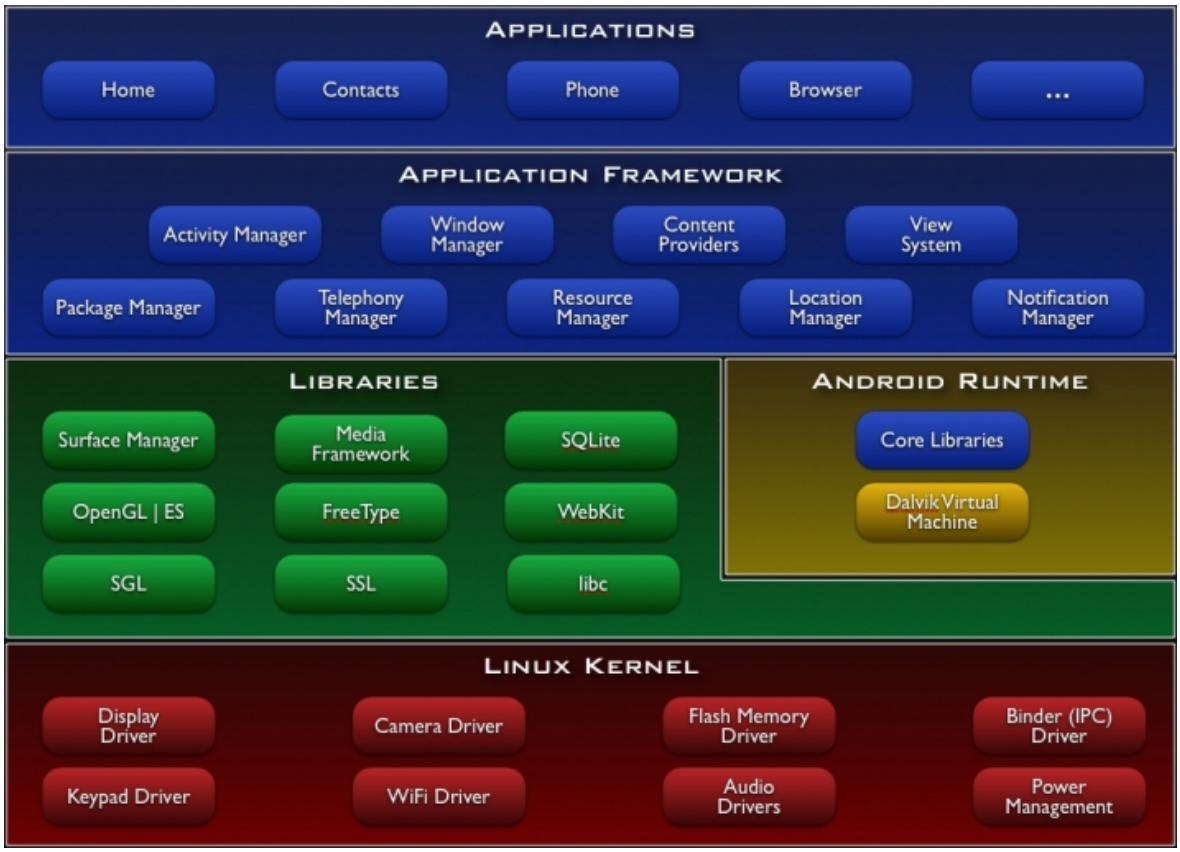
Android oparty jest na jądrze Linuksa, które wraz ze zbiorem bibliotek i specyficznym dla tej platformy środowiskiem uruchomieniowym tworzy podstawy dla działających w ramach systemu aplikacji. Jak można zauważyć na rys. 3.4, system można podzielić na 5 warstw:

- **Jądro Linuksa.** Jest to najniższa warstwa, która stanowi rdzeń systemu i pośredniczy pomiędzy sprzętem, a resztą systemu. Zawiera wszystkie sterowniki dla sprzętu, zapewnia zarządzanie pamięcią i procesami, bezpieczeństwo, obsługę sieci oraz zarządzanie zasilaniem. Od wersji Androida 4.0 wykorzystywane jest jądro w wersji 3.0, wcześniej było to 2.6;
- **Biblioteki.** Jest to warstwa działająca nad jądrem Linuksa, zawierająca biblioteki napisane w C lub C++, zarówno te podstawowe, jak na przykład libc, jak i biblioteki zapewniające obsługę podstawowych usług, takich jak:

- Obsługa przeglądarki internetowej i szyfrowanie przesyłanych danych – WebKit i SSL,
- Zarządzanie wyświetlaczem,
- Zaawansowane efekty 2D i 3D – SGL i OpenGL,
- Obsługę bazy danych – SQLite,
- Odtwarzanie audio i wideo;

- **Android Runtime.** Jest to przygotowane specjalnie dla Androida środowisko uruchomieniowe, które pośredniczy pomiędzy aplikacjami, a rdzeniem systemu. To ono odróżnia Androida od innych systemów, opartych na Linuksie. Składa się z dwóch podstawowych elementów:
  - **Podstawowe biblioteki.** Aplikacje na Androida są pisane w Javie, jednak nie są uruchamiane w maszynie wirtualnej Javy, a za pomocą Dalvik Virtual Machine, na której większość podstawowych bibliotek Javy nie jest dostępna. Lukę tę wypełniają specjalnie przygotowane biblioteki, które zapewniają większość funkcji dostępnych w podstawowych bibliotekach Javy, a także dodają nowe, specyficzne dla Androida,
  - **Dalvik Virtual Machine.** Jest to jeden z kluczowych elementów systemu Android. Zamiast korzystać z tradycyjnej maszyny wirtualnej Javy, Android wykorzystuje swoją własną maszynę wirtualną, przystosowaną specjalnie dla urządzeń o bardzo ograniczonych zasobach pamięci i procesora, jednocześnie umożliwiającej efektywne działanie wielu instancji na jednym urządzeniu. Dalvik VM wykorzystuje jądro Linuksa do obsługi niskopoziomowych funkcjonalności, na przykład zarządzanie procesami i pamięcią. Programy napisane w języku Java, zanim jeszcze zostaną zainstalowane, są kompilowane do kodu bajtowego, a następnie przekształcane z formatu .class kompatybilnego z JVM do formatu .hex, kompatybilnego z Dalvik Virtual Machine. Tak przygotowane pliki wykonywalne, są przystosowane do uruchamiania na sprzęcie o ograniczonych zasobach, dzięki czemu można zapewnić płynność działania systemu nawet na telefonach o bardzo niskich parametrach sprzętowych;
- **Framework Aplikacji.** Ta warstwa zawiera klasy służące do tworzenia aplikacji na Androida. Umożliwia także dostęp do sprzętu i zasobów aplikacji, oraz zarządzanie widokiem;

- **Warstwa aplikacji.** Wszystkie aplikacje, zarówno te standardowo wbudowane w system, jak i instalowane z zewnętrznych źródeł, działają właśnie w tej warstwie, i korzystają z tych samych bibliotek API. Aplikacje działające w tej warstwie wykorzystują klasy udostępniane przez framework dostępny w niższej warstwie [10].



Rys 3.4 Schemat architektury systemu Android [I]

3.3. Tworzenie aplikacji na system Android

Pomimo tego, iż istnieje możliwość portowania aplikacji napisanych na przykład w języku C++ do formatu zgodnego z Androidem, podstawowym językiem do tworzenia aplikacji na ten system jest Java. Choć język nie jest nowy, programowanie aplikacji na Androida znacząco różni się od programowania zwykłych aplikacji w Javie. Dzieje się tak, ponieważ większość podstawowych bibliotek Javy nie jest dostępna w Androidzie, więc mimo dobrej znajomości tego języka, można mieć spore problemy z tworzeniem aplikacji na ten system.

Aby rozpocząć tworzenie aplikacji na Androida potrzebne jest:

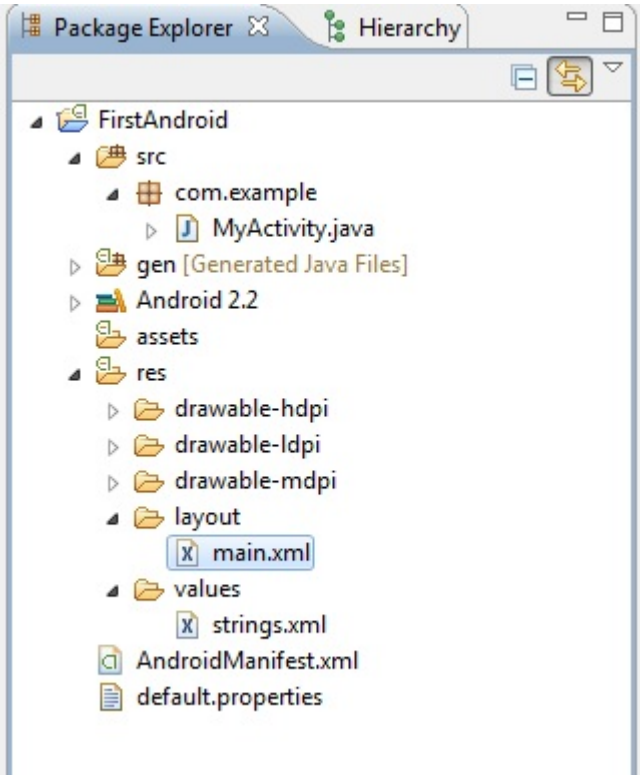
- Java SE Development Kit – zestaw do projektowania w środowisku Java;
- Android SDK – zestaw narzędzi specjalnie przygotowany dla programistów Androida. W jego skład wchodzą między innymi:
  - Android API, czyli wszystkie biblioteki wymagane przy tworzeniu aplikacji. Te same biblioteki są wykorzystywane przez sam system,
  - Emulator Androida umożliwiający testowanie aplikacji na komputerze bez konieczności posiadania telefonu z Androidem,
  - Narzędzia programistyczne, umożliwiające kompilowanie i debugowanie aplikacji,
  - Pełna dokumentacja,
  - Przykładowe fragmenty kodu [11];
- Środowisko programistyczne. Aplikacje można tworzyć w dowolnym edytorze, jednak zalecanym środowiskiem jest IDE Eclipse, wraz ze specjalnie dla niego przygotowaną wtyczką środowiskową ADT (ang. Android Development Tools) integrującym Android SDK z Eclipse [7].

Przed rozpoczęciem pisania pierwszych programów, warto zapoznać się ze szkieletem aplikacji, oraz jego podstawowymi składnikami. Są to:

- Widok (ang. View). Widoki są to elementy interfejsu użytkownika. Wykorzystują model hierarchiczny i posiadają zakodowane informacje, dzięki którym są poprawnie wyświetlane. Mogą przybierać formę przycisku, pola tekstowego, etykiety;
- Aktywność (ang. Activity). Jest to reprezentacja interfejsu danej aplikacji, zawierająca przeważnie co najmniej jeden widok. Za jego pomocą użytkownik może kontaktować się z aplikacją, wybierać co chce w danej chwili zrobić;
- Intencja (ang. Intent). Intencje definiują zamiar wykonania jakiegoś zadania. Umożliwiają uruchomienie usługi bądź aktywności, odebranie połączenia telefonicznego, wysłanie komunikatu czy też wyświetlenie strony internetowej. Intencje mogą być wysyłane zarówno przez aplikacje do systemu, jak i od systemu do aplikacji;

- Dostawca treści (ang. Content provider). Jest to standardowy mechanizm współdzielenia informacji pomiędzy aplikacjami działającymi na Androidzie. Dzięki niemu można korzystać z wybranych zasobów i danych innych aplikacji, bez udostępniania całych magazynów, struktury i ich implementacji;
- Usługi (ang. Service). Usługi, podobnie jak w innych systemach, są to procesy działające w tle, przygotowane do działania przez długi okres czasu i wykonujące konkretne zadanie. W Androidzie usługi dzieli się na dwa typy: usługi lokalne i zdalne. Usługi lokalne są dostępne tylko dla aplikacji, w ramach której działają, natomiast usługi zdalne umożliwiają zdalny dostęp do nich także dla innych aplikacji działających w systemie [8].

Struktura aplikacji Androida składa się z trzech głównych elementów: deskryptora aplikacji, kodu źródłowego oraz zasobów. Taki podział zapewnia oddzielenie zasobów od logiki aplikacji, co znacząco ułatwia tworzenie i rozwijanie oprogramowania.



Rys 3.5 Struktura przykładowej aplikacji Androida [V]

Na rys. 3.5 można zobaczyć strukturę przykładowej aplikacji stworzonej na system Android. Jej najważniejsze elementy to:

- *AndroidManifest.xml* – plik deskryptora aplikacji. Definiowane są w nim wszystkie intencje i aktywności, ustala wymagania oraz uprawnienia aplikacji, a także możliwość dostępu dla innych aplikacji, które korzystają z programu. Plik jest wymagany do prawidłowego działania aplikacji [6];
- *src* – folder, w którym przechowywany jest kod aplikacji. Znajdują się w nim wszystkie klasy, z jakich zbudowana jest aplikacja;
- *asset* – jest to luźny zbiór plików i folderów, nie jest wymagany do działania programu;
- *res* – folder, zawierający zasoby aplikacji. Zawiera podrzędne katalogi:
  - *drawable* – folder zawierający grafikę programu. Występuje w trzech wersjach, przystosowanych dla różnych wielkości ekranu, na którym działa aplikacja,
  - *layout* – folder przechowujący widoki aplikacji zapisane za pomocą plików XML (ang. Extensible Markup Language),
  - *values* – tutaj przechowywane są inne zasoby aplikacji, jak na przykład ciągi znaków, kolory. Zapisane są one za pomocą plików XML [8].

Jak można zauważyć, zasoby oraz elementy interfejsu są wyraźnie oddzielone od kodu aplikacji. Znacząco ułatwia to jej późniejsze rozwijanie i modyfikowanie, gdyż zmiany wyglądu aplikacji bądź treści komunikatów nie wymagają modyfikacji głównego kodu aplikacji – wystarczy zmienić wartości w odpowiednim pliku. Zastosowanie tutaj języka znaczników XML jest bardzo wygodne dla programistów, głównie dzięki jego przejrzystości. Nie ma też wpływu na wydajność, gdyż podczas instalowania programu, wartości są konwertowane i kompilowane do kodu binarnego. Dzięki temu nadmiarowość danych, jaka towarzyszy korzystaniu z plików XML nie ma wpływu na wydajność [7].

Android udostępnia również wiele gotowych klas, które ułatwiają obsługę i zarządzanie dodatkowymi urządzeniami, w jakie wyposażony jest telefon. Udostępniają one gotowe metody, dzięki którym możliwe jest zarządzanie na przykład kamerką, czy dyktafonem. W ten sposób programista nie musi przejmować się obsługą sprzętu. Wykorzystując gotowy interfejs w dużo prostszy sposób dodawać można dodatkowe funkcjonalności do swoich programów. Klasy, które odpowiadają za obsługę kamery to:

- Camera – klasa zajmującą się obsługą, oraz sterowaniem kamerą. Wykorzystywana jest przy robieniu zdjęć oraz nagrywaniu filmów wideo;
- SurfaceView – klasa wykorzystywana do prezentacji użytkownikowi aktualnego obrazu z kamerki telefonu;
- MediaRecorder – klasa wykorzystywana do nagrywania obrazu z kamerki. Wykorzystywana jest również przy rejestracji dźwięki (na przykład w dyktafonie).

Dodatkowo dzięki wykorzystaniu intencji, możliwe jest uruchamianie zewnętrznych aplikacji wykonujących wybrane czynności. Dzięki temu nie ma konieczności umieszczania obsługi aparatu w tworzonym programie – wystarczy za pomocą intencji uruchomić zewnętrzną aplikację obsługującą aparat, która zwróci gotowe zdjęcie. Jest to szczególnie przydatne przy tworzeniu aplikacji, której głównym celem nie jest robienie zdjęć, a jest to jedynie dodatek. W ten sposób niewielkim kosztem można dodać możliwość robienia zdjęć do swojej aplikacji.

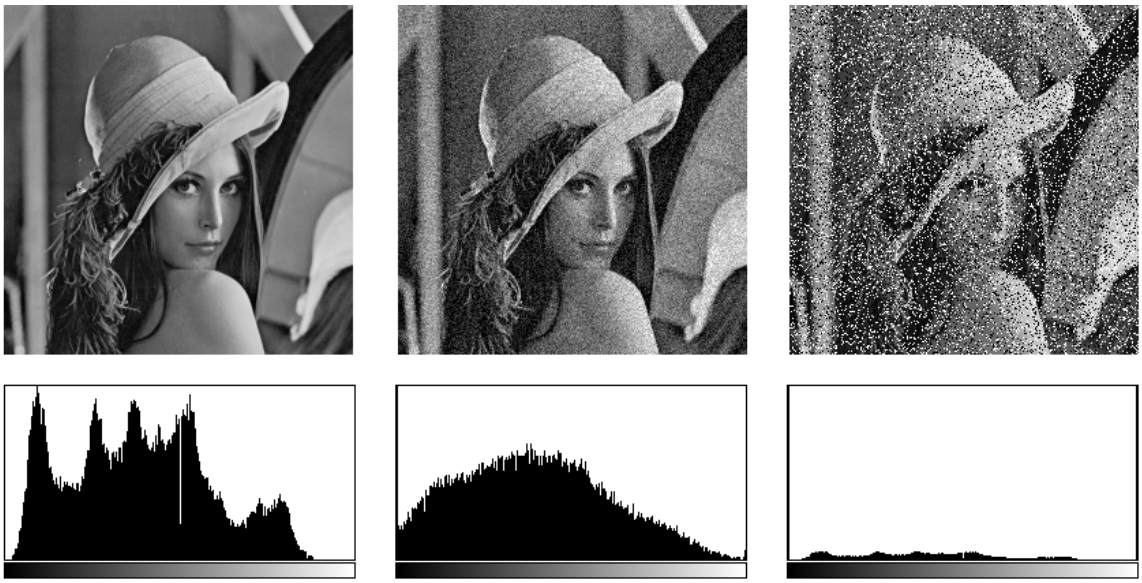
Android udostępnia również narzędzia ułatwiające zarządzanie multimediami. Dzięki wykorzystaniu klas takich jak *MediaPlayer* i *AudioManager* możliwe jest dodawanie do aplikacji możliwości odtwarzania obrazów i dźwięków, zarówno zapisanych w pamięci telefonu, jak i tych umieszczonych w sieci. Umożliwiają one również odtwarzanie plików stanowiących zasoby tworzonej aplikacji – w ten sposób możliwe jest na przykład dodanie motywu dźwiękowego działającego w tle aplikacji [XI].

4. Algorytmy usuwania szumów z obrazów

4.1. Szum w obrazach cyfrowych

Przy przetwarzaniu obrazów, szumem nazywa się piksele znacznie odbiegające kolorem od otoczenia. Są to zniekształcenia powstające wskutek zakłóceń oddziałujących na matrycę aparatu, przy zbyt dużej czułości ISO (ang. International Organization for Standardization). Szum jest bardziej widoczny przy matrycach o małej przekątnej, jakie występują zwykle w tańszych aparatach fotograficznych lub telefonach komórkowych.

Dwa najczęściej występujące przy cyfryzacji obrazów szumy, to szum Gaussa oraz szum impulsowy. Pierwszy z nich polega na dodaniu do każdego piksela obrazu sygnału szumu, opisanego zmienną losową o rozkładzie Gaussa, z zerową wartością oczekiwaną i stałym odchyleniem standardowym. W efekcie wszystkie piksele obrazu są lekko zniekształcone. Szum impulsowy z kolei polega na zmianie losowych pikseli na wartości znacznie odbiegające od rzeczywistej. Przykładem tego tupu szumu jest szum typu pieprz i sól [4]. Przykłady szumów można zobaczyć na rys. 4.1.



Rys. 4.1 Przykłady szumów oraz ich histogramy. Od lewej: oryginalny obraz, szum Gaussa, szum impulsowy typu pieprz i sól [X]

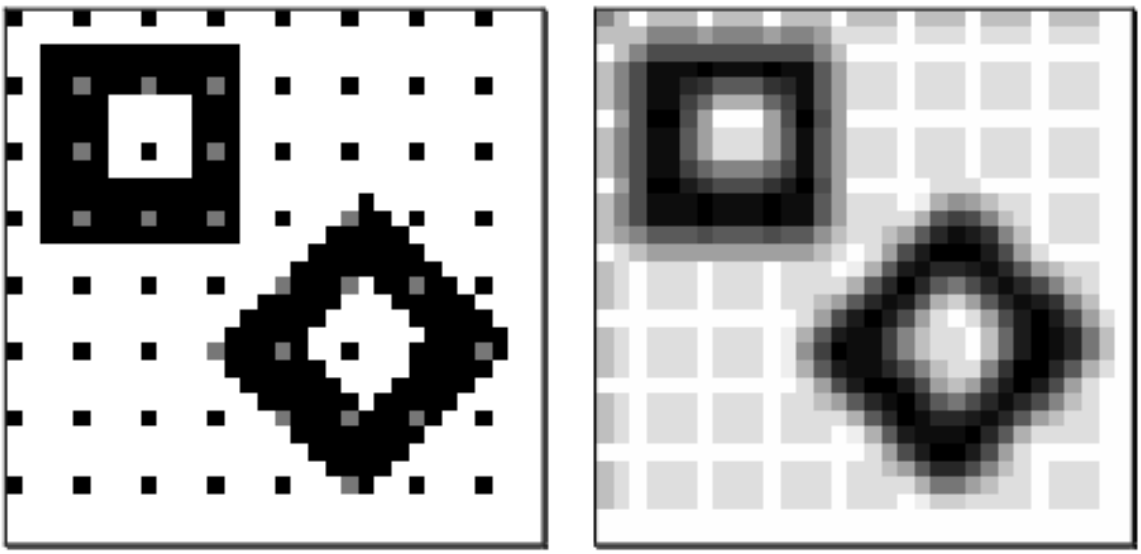
4.2. Klasyfikacja filtrów

W celu usunięcia z obrazu szumów stosuje się różne filtry, różniące się pomiędzy sobą zarówno skutecznością usuwania szumów, jak i stopniem zniekształcania obrazu.

Ponieważ z technicznego punktu widzenia nie można jednoznacznie stwierdzić co jest szumem i jaka była rzeczywista wartość badanego punktu, wszystkie filtry w mniejszym lub większym stopniu usuwają szczegóły z filtrowanego obrazu, zastępując piksel nową wartością obliczoną na podstawie wartości innych pikseli w jego otoczeniu. Zasadniczo filtry można podzielić na dwie grupy: filtry liniowe (splotowe) i nieliniowe.

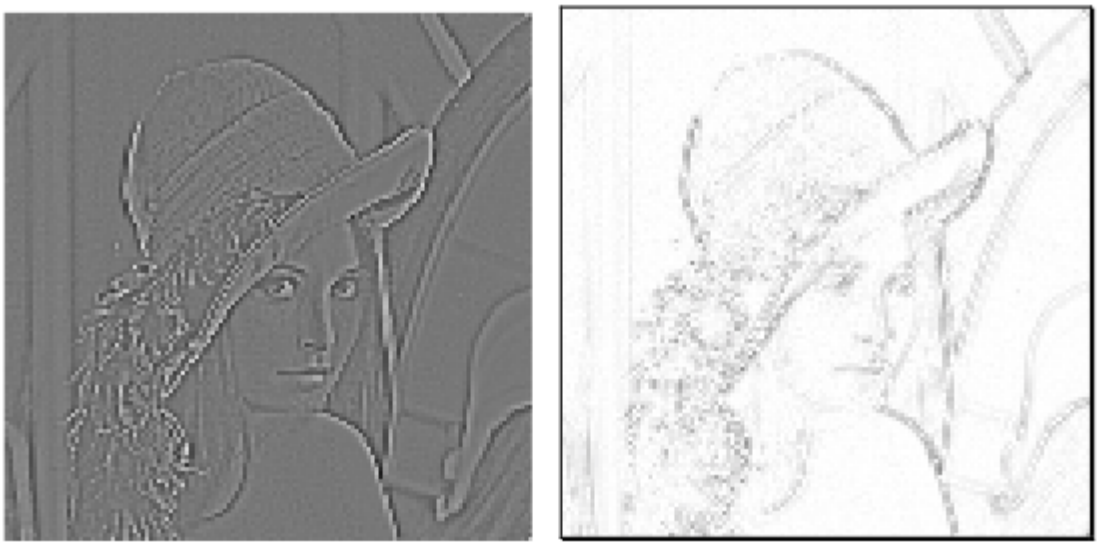
Filtry liniowe cechują się tym, że wszystkie punkty przetwarzanego obrazu przekształcane są w ten sam sposób. Ich dużą zaletą jest łatwa implementacja i niska złożoność obliczeniowa. Do usuwania szumów stosuje się filtry uśredniające, które obliczają średnią wartość z pikseli zadanej maski, i zastępują nią wartość badanego piksela. Filtry te bardzo dobrze radzą sobie z usuwaniem szumu Gaussa, ale cechują się bardzo dużą ingerencją w przetwarzany obraz (każdy piksel zastępowany jest nową wartością) poprzez rozmywanie krawędzi obrazu. Na rys. 4.2 i 4.3 można zauważyć przykładowe działanie filtru uśredniającego dla maski jednorodnej o wymiarach 3×3.

Jak widać na rys. 4.2 filtr rozmywa drobne zakłócenia w postaci pojedynczych czarnych punktów na białym tle, oraz jasnych na czarnym tle. Zakłócenia mają jednak duży wpływ na wartość pikseli w swoim otoczeniu – w miejscu czarnych punktów można zauważyć ciemniejsze obszary – zakłócenia wpłynęły na sąsiednie piksele obrazu. Filtr również bardzo niekorzystnie wpłynął na jakość obrazu, rozmazując kontury obiektów i pogarszając rozpoznawalność ich kształtów. Podobny efekt można również zauważyć na rys. 4.3, gdzie filtrowany był bardziej realistyczny obraz. Zniekształcenia wynikające z przefiltrowania obrazu można zobaczyć na rys. 4.4.



Rys. 4.2 Obraz z zakłóceniami (po lewej) i efekt jego filtracji filtrem uśredniającym [15]

Rys. 4.3 Efekt filtracji filtrem uśredniającym dla niezakłóconego obrazu naturalnego [15]



Rys. 4.4 Skutki filtracji filtrem uśredniającym: po lewej prosty obraz różnicowy (przeskalowany), po prawej moduł różnicy [15]

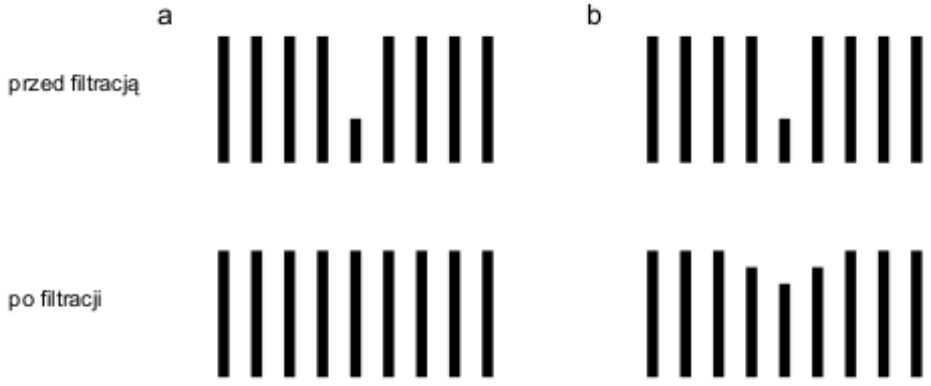
Rozmywanie obrazu można zredukować poprzez stosowanie wag. W powyższym przykładzie każdy piksel w masce miał taką samą wagę, przez co tak samo wpływał na wartość badanego piksela. Dzięki zróżnicowaniu wag możliwe jest zredukowanie rozmywania obrazu, poprzez zwiększenie wpływu centralnych pikseli, a osłabienie zewnętrznych. Przykłady masek można zobaczyć na rys. 4.5 [XII].

1	1	1
1	1	1
1	1	1

1	4	7	4	1
4	16	26	16	4
7	26	41	26	7
4	26	16	26	4
1	4	7	4	1

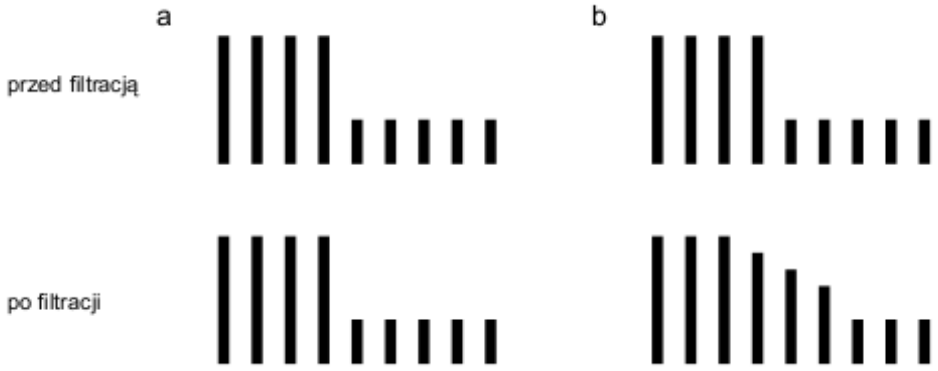
Rys. 4.5 Przykłady masek z uwzględnieniem wag pikseli [XII]

Innym typem filtrów są filtry nieliniowe. W przeciwieństwie do wcześniej omawianych, nie stosują one jednej reguły dla wszystkich pikseli, ale uwzględniając otoczenie badanego punktu wybierają najbardziej korzystną dla niego wartość. Najczęściej w tym celu stosuje się filtry wykorzystujące medianę. Mediana jest to wartość środkowa w uporządkowanym rosnąco ciągu wartości z przetwarzanego otoczenia badanego piksela. Filtr medianowy jest tak zwanym filtrem mocnym, to znaczy, że wartości znacznie odbiegające od średniej nie są brane pod uwagę i nie mają wpływu na wartość piksela po przetworzeniu. To sprawia, że filtr ten bardzo dobrze sprawdza się przy usuwaniu szumów lokalnych, jednocześnie nie powodując ich rozmazania na większej powierzchni, tak jak to było w przypadku filtrów uśredniających. Efekt ten można zauważyć na rys. 4.6.

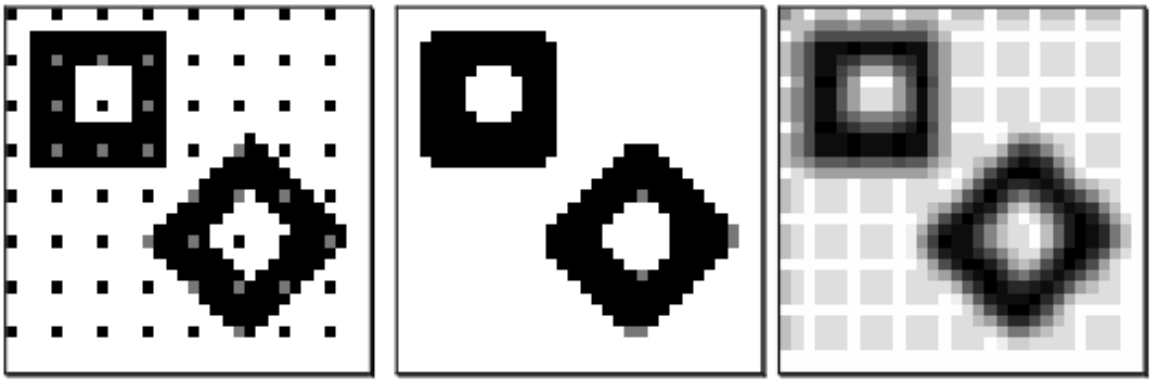


Rys 4.6 Usuwanie zakłóceń filtrem medianowym (a) i filtrem uśredniającym (b) [15]

Największą zaletą filtrów medianowych jest jednak to, że z reguły nie rozmywają krawędzi obiektów, co można zauważyć na rys. 4.7. Tradycyjny filtr uśredniający generuje sztuczne poziomy jasności pomiędzy rzeczywistymi wartościami, natomiast filtr medianowy nie powoduje pogorszenia ostrości krawędzi w badanym obrazie. Nie generuje też żadnych nowych wartości, więc wynikowy obraz nie wymaga żadnego skalowania. Różnicę w działaniu dwóch filtrów można zobaczyć na rys. 4.8.

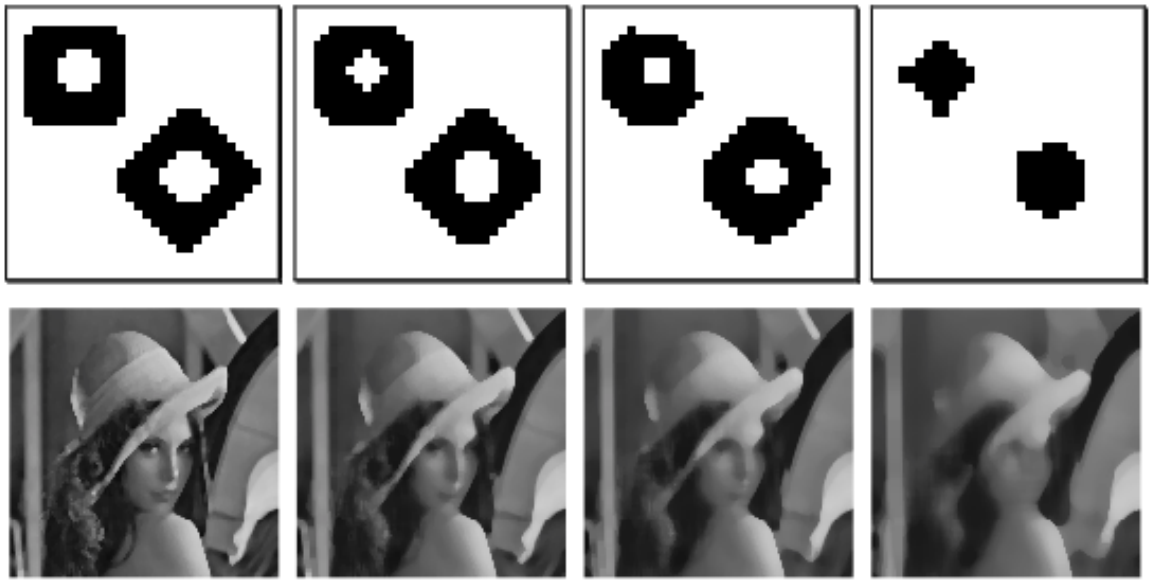


Rys. 4.7 Wpływ filtru medianowego (a) oraz filtru uśredniającego (b) na krawędzie obiektu [15]



Rys. 4.8 Sztuczny obraz (po lewej) po filtracji filtrem medianowym (pośrodku) oraz filtrem uśredniającym (po prawej) [15]

Jak można zauważyć, filtr medianowy posiada wiele zalet – bardzo dobrze eliminuje szumy z obrazów oraz nie rozmywa krawędzi obiektów. To czyni go jedną z najlepszych technik filtracji obrazów. Nie jest jednak pozbawiony wad, co widać już na rys. 4.8. Filtry medianowe mają tendencję do zniekształcania narożników filtrowanych obiektów. Poza tym również zniekształcają obraz, co można dokładniej zaobserwować na rys. 4.9, gdzie przedstawiono wyniki filtracji medianowej z różną wielkością okna, kolejno: 3×3, 5×5, 7×7, 9×9 [15].



Rys 4.9 Skutki filtracji medianowej dla różnych rozmiarów okna [15]

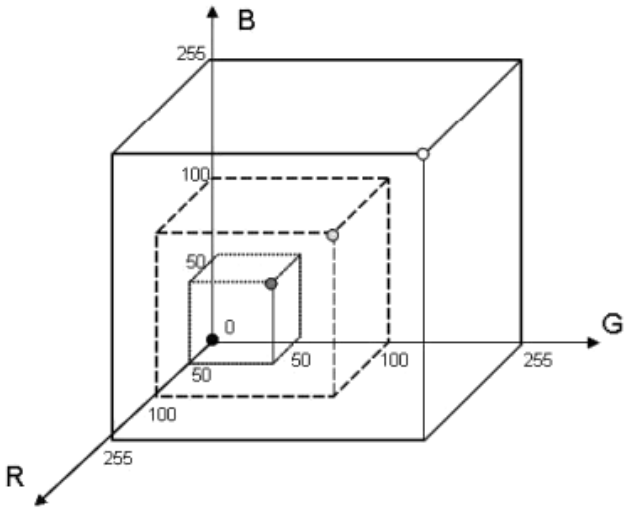
Kolejną wadą filtrów medianowych jest duża złożoność obliczeniowa. Najprostszy sposób obliczenia mediany polega na posortowaniu wartości wszystkich pikseli w badanej masce, a następnie wybranie środkowego. Istnieje jednak wiele znacznie wydajniejszych rozwiązań, gdyż w rzeczywistości nie ma potrzeby sortować wszystkich elementów, żeby poznać medianę. Jednym z nich jest algorytm Hoare'a, o działaniu podobnym do algorytmu QuickSort, czyli wybraniu jednej losowej wartości ze zbioru i podzieleniu go na wartości od niej mniejsze i większe [9]. Kolejną propozycją jest algorytm „magicznych piątek", bazujący na algorytmie Hoare'a, z tym że element dzielący nie jest wybierany losowo, ale tak, aby dzielił zbiór wartości na dwie stosunkowo równe części. Realizowane jest to poprzez podzielenie zbioru wartości na zbiory 5-elementowe, z których obliczane są mediany tych zbiorów, a następnie rekurencyjne wywoływanie tego algorytmu dla uzyskanego zbioru median. W ten sposób ostatecznie otrzymujemy liczbę s, która jest większa od przynajmniej $1/10$ zbioru oraz mniejsza od $1/10$ zbioru. Dzięki temu w następnym kroku rozpatrujemy już tylko $4/5$ zbioru, co zapewnia nam liniową złożoność obliczeniową algorytmu [5].

4.3. Vector Median Filter

Filtr medianowy łatwo zaimplementować jest w jednowymiarowej przestrzeni barw, czyli na przykład dla obrazów czarno-białych. Problem pojawia się w przypadku obrazów kolorowych, ponieważ nie ma prostej metody, która pozwalałaby nadać poszczególnym

kolorom jednowymiarowych wartości, które jednocześnie odwzorowywałyby różnice pomiędzy nimi. Nie ma więc możliwości obliczenia dla nich mediany w taki sam sposób, jak dla obrazów czarno-białych.

Do reprezentacji kolorów stosuje się specjalnie stworzone modele barw. W technice komputerowej najczęściej wykorzystywany jest model RGB (ang. Red, Green, Blue). Zakłada się w nim, że każdy kolor jest mieszanką trzech podstawowych barw: czerwonej, zielonej i niebieskiej, w wartościach od 0 do 255 (jeden bajt). Każdy kolor można więc określić jako wektor w trójwymiarowej przestrzeni, której osiami są trzy podstawowe barwy [4]. Schemat tego układu prezentuje rys. 4.10.



Rys 4.10 Bryła przestrzeni RGB [4]

Umieszczając kolory jako wektory w przestrzeni RGB, można za pomocą wzoru 4.1 obliczyć odległości pomiędzy nimi. W efekcie odległość ta będzie określała różnicę pomiędzy dwoma kolorami.

$$d_e(A,B)=\sqrt{(x_A-x_B)^2+(y_A-y_B)^2+(z_A-z_B)^2}, \qquad (4.1)$$

gdzie:

$d_e(A,B)$ – odległość punktu A od puntu B,

x, y, z – współrzędne punktu w trójwymiarowym układzie współrzędnych.

Odległość obliczana za pomocą wzoru 4.1 jest to odległość euklidesowa. Mimo iż dobrze odwzorowuje różnice pomiędzy dwoma kolorami, ten sposób jest dość kosztowny obliczeniowo. Bardzo podobne efekty przy niższym koszcie obliczeniowym można

uzyskać stosując metrykę miejską, gdzie odległość jest określana jako suma wartości bezwzględnych różnic ich współrzędnych (wzór 4.2).

$$d_m(A,B)=|x_A-x_B|+|y_A-y_B|+|z_A-z_B| \tag{4.2}$$

Jak można zauważyć, przy użyciu metryki miejskiej pomija się potęgowanie i pierwiastkowanie, dzięki czemu obliczenie odległości w ten sposób jest o wiele szybsze.

Mając możliwość obliczenia odległości pomiędzy punktami, można zastosować filtr medianowy stosując medianę wektorową. Pojęcie to nawiązuje do mediany skalarnej, którą opisuje wzór 4.3.

$$med=\arg\min_{F_i\in W}\sum_{j=0\ldots N-1}|F_i-F_j|, \tag{4.3}$$

gdzie:

W – zbiór wszystkich punktów maski (o indeksach 0..N-1),

F_i – punkt maski, dla którego obliczana jest odległość,

F_j – pozostałe punkty maski.

Jeśli zastąpi się odległości pomiędzy skalarami odległością pomiędzy wektorami, obliczoną według wybranej metryki, to otrzymamy medianę wektorową. Formalnie wyraża się ją wzorem 4.3 [14].

$$R=\min_{F_i\in W}\sum_{j=0}^{N-1} d(F_i-F_j), \tag{4.4}$$

gdzie:

R – mediana wektorowa,

W – zbiór zawierający wszystkie punkty maski,

N – liczba pikseli w masce,

F_i, F_j – punkty maski.

Filtr opierający się na medianie tego typu nazywany jest Vector Median Filter. Działa on w ten sposób, że dla każdego punktu zadanej maski obliczana jest suma odległości do wszystkich pozostałych punktów maski. Punkt, dla którego ta suma jest najmniejsza, jest medianą i zastępuje badany piksel.

4.4. Fast Modified Vector Median Filter

Smolka i inni w swojej pracy [13] zauważyli, że zaszumiona wartość badanego punktu może negatywnie wpływać na wybór sąsiada, który go zastąpi. Zaproponowali oni filtr FMVMF (Fast Modified Vector Median Filter), w którym pomijane w formule jest obliczanie odległości do badanego punktu. We wzorze wprowadzono współczynnik progujący β. Po danej modyfikacji wzór na sumę odległości punktu centralnego wygląda następująco:

$$R_0 = -\beta + \sum_{j=1}^{N-1} d(F_0 - F_j), \tag{4.5}$$

gdzie:

R_0 – odległości dla badanego punktu,

β – współczynnik progujący,

F_0 – badany punkt,

F_j – punkty maski.

Natomiast w pozostałych punktach nie uwzględnia się odległości do punktu centralnego, co widać na wzorze 4.6.

$$R_i = \sum_{j=1}^{N-1} d(F_i - F_j) \text{ dla } i = 1, \ldots, N-1, \tag{4.6}$$

gdzie:

R_i – odległości dla badanego punktu,

F_i, F_j – punkty maski.

Badany punkt F_0 zamieniany jest na punkt F_k gdy spełniony jest warunek zapisany we wzorze 4.7 [14].

$$\sum_{j=1}^{N-1} d(F_k - F_j) < -\beta + \sum_{j=1}^{N-1} d(F_0 - F_j) \tag{4.7}$$

Wartość parametru β przy założeniu normalizacji barw składowych do przedziału [0; 1] została przez autorów filtra [13] ustalona eksperymentalnie na 0,75, co w odniesieniu do przestrzeni barw na jakich operujemy ([0; 255]) daje wartość w okolicy 191. Jak można łatwo zauważyć, zaproponowany filtr jest szybszy od standardowego

VMF, gdyż tylko dla punktu środkowego uwzględniana jest odległość do wszystkich punktów – dla pozostałych liczymy o jedną odległość mniej niż w przypadku standardowego VMF. Filtr jest również bardziej odporny na zniekształcenia powodowane zaszumieniem badanego piksela [3].

5. Projekt aplikacji do redukcji szumów

5.1. Założenia aplikacji

Tworzona w ramach pracy aplikacja ma udostępniać dwie podstawowe funkcjonalności:

- Możliwość zrobienia zdjęcia oraz zapisania go do pamięci telefonu;
- Opcjonalna możliwość odszumienia zrobionego wcześniej zdjęcia. Operację tą można będzie powtarzać aż do uzyskania zadowalających efektów, a wynik każdego cyklu będzie zapisywany jako kolejne zdjęcie.

Zdjęcia w systemie Android zapisywane są zwykle w stratnym formacie JPEG (ang. Joint Photographic Experts Group). Wielokrotna kompresja do tego formatu, a następnie jego dekompresja, może powodować znaczne zniekształcenia w przetwarzanym obrazie [1]. Dlatego od momentu zrobienia zdjęcia, jego kopia przechowywana jest w pamięci telefonu jako mapa bitowa, dzięki czemu unika się zniekształceń towarzyszących wielokrotnemu zapisywaniu oraz odczytywaniu zdjęcia. Dodatkowo zapisywane zdjęcia są konwertowane do formatu JPEG przy ustawionej 100-procentowej jakości, dzięki czemu minimalizowane są zniekształcenia powodowane przez samą konwersję do danego formatu. Do zrobienia zdjęcia zostaje wykorzystywana zewnętrzna aplikacja, która umożliwia zrobienie zdjęcia, a następnie zwraca je do głównej aplikacji. Schemat działania programu można zobaczyć na rys. 5.1.

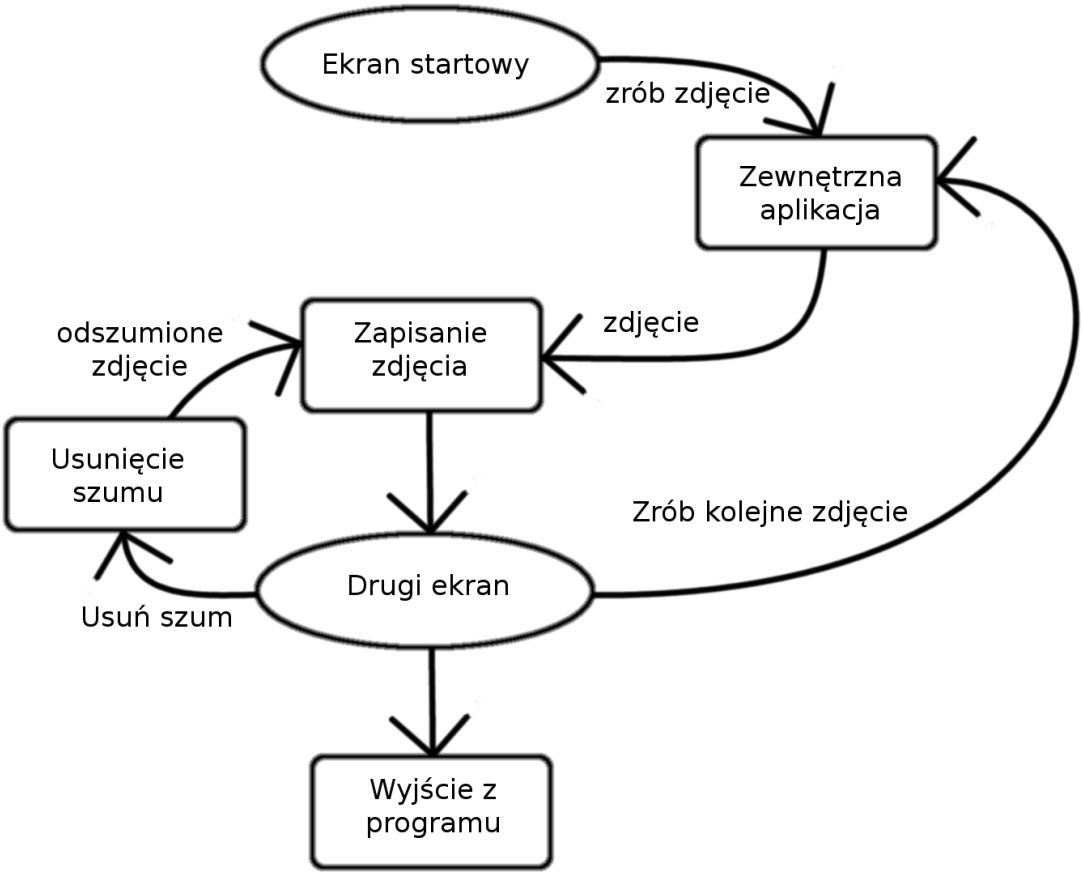
5.2. Wymagania aplikacji

Aplikacja do poprawnego działania wymaga telefonu z zainstalowanym systemem Android w wersji 2.2 lub wyższej, oraz z wbudowanym aparatem fotograficznym. Telefon musi mieć również zainstalowane podstawowe oprogramowanie do obsługi aparatu fotograficznego.

5.3. Obsługa aparatu

Aby uniknąć zniekształceń obrazu towarzyszących konwersji do formatu JPEG, aplikacja umożliwia robienie zdjęć, dzięki czemu uzyskany z aparatu obraz jest przekazywany bezpośrednio do aplikacji jako bitmapa i nie jest w żaden sposób konwertowany. Ponieważ jednak robienie zdjęć nie jest głównym zadaniem aplikacji,

zamiast komplikować program obsługą kamerki, wykorzystywana jest aplikacja zainstalowana już w systemie (zdecydowana większość telefonów z systemem Android ma już taką).



Rys. 5.1 Schemat działania aplikacji

Aby jednak móc korzystać z aparatu w telefonie, konieczne jest zdefiniowanie tego w pliku *AndroidManifest.xml*:

```
<uses-feature android:name="android.hardware.camera" />
```

W ten sposób zadeklarowano, że aplikacja do prawidłowego działania wymaga dostępu do kamerki telefonu. Po umieszczeniu aplikacji na Android Market, nie będzie ona widoczna dla telefonów, które nie posiadają kamerki.

Aby wywołać aplikację, wykorzystywana jest intencja (ang. Intent). Jak sama nazwa może wskazywać, intencja definiuje chęć wykonania jakiejś akcji, w tym wypadku zrobienie zdjęcia. Oto przykładowa funkcja, która umożliwia wywołanie aplikacji do robienia zdjęć:

```
private void dispatchTakePictureIntent(int actionCode) {
    Intent takePictureIntent = new
└Intent(MediaStore.ACTION_IMAGE_CAPTURE);
    startActivityForResult(takePictureIntent, actionCode);
}
```

Funkcja składa się z dwóch głównych elementów: stworzenia obiektu intencji, która ma uruchomić aplikację do robienia zdjęć, oraz uruchomienie wcześniej stworzonej intencji z podanym kodem akcji, który pozwoli ją zidentyfikować przy otrzymywaniu wyników. Nie jest to jednak wszystko, gdyż poza tym, że system uruchomił daną aplikację, nic się nie stało – zrobione zdjęcie nigdzie nie zostało zapisane.

Efekt działania intencji jest zwracany poprzez funkcję *onActivityResult()*. Ponieważ jednak uzyskany w ten sposób obraz jest bardzo mały, nie nadaje się jako materiał na zdjęcie. Aby uzyskać obraz w większych wymiarach, konieczne jest wskazanie intencji miejsca, gdzie należy zapisać fotografię. Wykorzystana została do tego wcześniej przygotowana funkcja *createTempImageFile()*, która tworzy plik tymczasowy, w którym przechowywane jest zdjęcie.

```
file = createTempImageFile();
takePictureIntent.putExtra(MediaStore.EXTRA_OUTPUT, Uri.fromFile(file));
```

W ten sposób po wykonaniu zdjęcia będzie możliwe jego odczytanie, a następnie przechowywanie w pamięci programu jako bitmapa.

```
cachebmOptions = new BitmapFactory.Options();
tempPhotoPath = file.getAbsolutePath();
cacheBitmap = BitmapFactory.decodeFile(tempPhotoPath, cachebmOptions);
```

Następnie konieczne jest zapisanie zdjęcia w pamięci telefonu. Do tego celu wykorzystywana jest funkcja *createImageFile()*, która działa bardzo podobnie do *createTempImageFile()*, z tym że tworzy plik, w którym ostatecznie będzie przechowywane zdjęcie. Następnie za pomocą strumienia bitmapa skompresowana jako JPEG zapisywana jest w utworzonym pliku.

```
File image = createImageFile();
mCurrentPhotoPath = image.getAbsolutePath();
FileOutputStream fos = new FileOutputStream(mCurrentPhotoPath);
cacheBitmap.compress(CompressFormat.JPEG, 100, fos);
fos.flush();
fos.close();
```

Aby obraz był widoczny w przeglądarce zdjęć, konieczne jest dodanie go do jednej z galerii. To również wykonywane jest za pomocą intencji, która przeskanuje wskazane zdjęcie.

```
Intent mediaScanIntent = new
Intent(Intent.ACTION_MEDIA_SCANNER_SCAN_FILE);
File file = new File(mCurrentPhotoPath);
Uri contentUri = Uri.fromFile(file);
mediaScanIntent.setData(contentUri);
this.sendBroadcast(mediaScanIntent);
```

Aby umożliwić użytkownikowi sterowanie aplikacją, konieczne jest podpięcie wcześniej stworzonych funkcji do interfejsu. W Androidzie interfejsy definiowane są za

pomocą plików XML. W tym wypadku potrzebne są dwa elementy: przycisk, który wywoła aplikację robiącą zdjęcia, oraz obrazek, w którym umieścimy efekt pracy aplikacji.

```
<Button android:text="@string/btnIntend" android:id="@+id/btnIntend"
        android:layout_height="wrap_content"
        android:layout_width="0dp"
        android:layout_weight="1" />
<ImageView      android:layout_height="wrap_content"
                android:layout_width="wrap_content"
                android:visibility="visible"
                android:id="@+id/imageView1" />
```

Ponieważ przy starcie aplikacji obiekt *ImageView* nie zawiera żadnej wartości, jest niewidoczny. Widoczny jest więc tylko przycisk. Aby jednak wciśnięcie go dawało jakiś efekt, koniecznie jest podpięcie do niego obsługi zdarzenia. Wykorzystywana jest do tego wcześniej przygotowana funkcja:

```
private void setBtnListenerOrDisable(
    Button btn,
    Button.OnClickListener onClickListener,
    String intentName
    ) {
          if (isIntentAvailable(this, intentName)) {
                btn.setOnClickListener(onClickListener);
          } else {
                btn.setText(getText(R.string.cannot).toString() + " " +
└btn.getText());
                btn.setClickable(false);
          }
    }
```

Powyższa metoda wykorzystując przygotowaną wcześniej funkcję *isIntentAvailable()* sprawdza, czy istnieje możliwość wywołania danej intencji. Jeśli jest to możliwe, podpina przygotowane wcześniej zdarzenie do przycisku, a w przeciwnym razie dezaktywuje go. Wywołanie przygotowanej funkcji można znaleźć poniżej – po jego wykonaniu kliknięcie na przygotowany przycisk będzie skutkowało uruchomieniem aplikacji do robienia zdjęcia.

```
Button picBtn = (Button) findViewById(R.id.btnIntend);
setBtnListenerOrDisable(
                picBtn,
                mTakePicOnClickListener,
                MediaStore.ACTION_IMAGE_CAPTURE
          );
```

Aby pokazać użytkownikowi miniaturę zrobionego zdjęcia, konieczne jest jego przeskalowanie, w celu zredukowania ilości zajmowanej pamięci. Aby ustalić skalę, konieczne jest porównanie wielkości naszej bitmapy do wielkości elementu interfejsu, który będzie przechowywał obrazek. Po przeskalowaniu, przygotowana miniaturka wstawiana jest do obiektu *ImageView*, a następnie ustawiana jako widoczna. Poniżej można zobaczyć przykładowy kod, który to robi.

```
/* Pobieramy wymiary obiektu ImageView */
int targetW = mImageView.getWidth();
int targetH = mImageView.getHeight();

/* Pobieramy rzeczywiste wymiary naszego zdjecia */
BitmapFactory.Options bmOptions = new BitmapFactory.Options();
bmOptions.inJustDecodeBounds = true;
BitmapFactory.decodeFile(mCurrentPhotoPath, bmOptions);
int photoW = bmOptions.outWidth;
int photoH = bmOptions.outHeight;

/* Ustalamy w jaki sposob nalezy przeskalowac obrazek */
int scaleFactor = 1;
if ((targetW > 0) || (targetH > 0)) {
    scaleFactor = Math.min(photoW/targetW, photoH/targetH);
}

/* Ustawiamy wybrana skale w opcjach dekodowania zdjecia */
bmOptions.inJustDecodeBounds = false;
bmOptions.inSampleSize = scaleFactor;
bmOptions.inPurgeable = true;

/* Dekodujemy obraz z JPEG do mapy bitowej */
Bitmap bitmap = BitmapFactory.decodeFile(mCurrentPhotoPath, bmOptions);

/* Wstawiamy przygotowana miniature do obiektu ImageView */
mImageView.setImageBitmap(bitmap);
mImageView.setVisibility(View.VISIBLE);
```

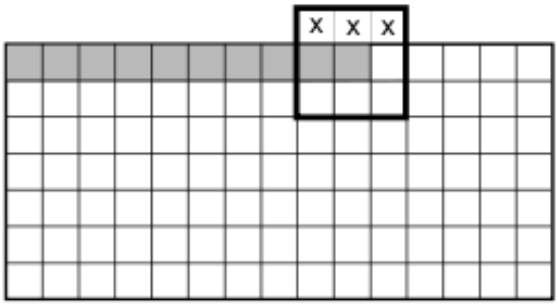
Powyższy przykład pobieżnie pokazuje jak wykorzystując zainstalowaną w telefonie aplikację, w prosty sposób można dodać do swojego programu możliwość robienia zdjęć. W przykładzie zostały wykorzystane fragmenty kodu zaczerpnięte z oficjalnej dokumentacji systemu Android [VI].

5.4. Implementacja algorytmu do redukcji szumów

Opierając się na informacjach zamieszczonych w przeglądzie literatury, oraz na badaniach Stolińskiego i Grabowskiego [14] wstępnie zostały wybrane dwa filtry: VMF oraz FMVMF. Filtry te okazały się być najbardziej uniwersalne oraz bardzo skuteczne w usuwanie szumu impulsowego, którego głównym źródłem są niedoskonałości matryc CCD [2]. Filtry zostały zaimplementowane dla pięciopunktowej maski krzyżowej. Po wstępnych testach wybrany został filtr FMVMF w metryce miejskiej. Główną jego zaletą była szybkość – filtr był zdecydowanie najszybszy spośród testowanych i jeśli chodzi o skuteczność osiągał porównywalne wyniki do pozostałych. Filtr FMVMF wymaga mniej obliczeń niż podstawowy VMF, a zastosowanie metryki miejskiej nie wymaga czasochłonnego podnoszenia do potęgi i pierwiastkowania. Ponieważ Android działa

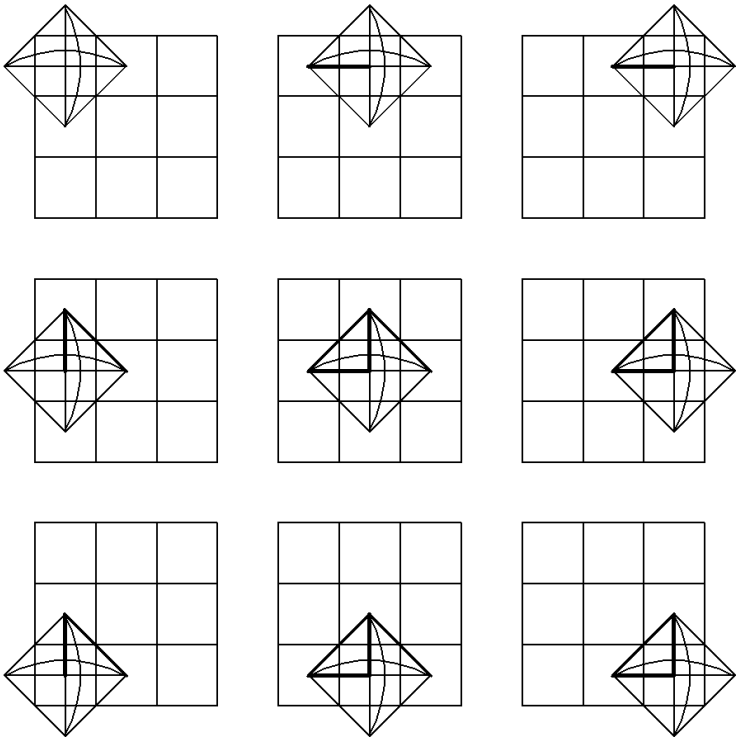
zwykle na platformach o ograniczonych zasobach, wydajność była jednym z kluczowych czynników.

Jednym z problemów jaki pojawia się przy kontekstowym przetwarzaniu obrazów jest brak niektórych pikseli w masce przy przetwarzaniu skrajnych pikseli. Obrazuje to rys. 5.2. Maski dobierane są zwykle w ten sposób, aby miały nieparzystą ilość pikseli, dzięki czemu zawsze istnieje punkt środkowy. Nie ma więc możliwości pominięcia brakujących pikseli. Jednym z rozwiązań tego problemu jest podstawienie pod brakujący piksel jakiejś wartości. Najczęściej spotykaną praktyką w tej sytuacji jest przyjmowanie wartości punktu centralnego. Ten sposób został wykorzystany w algorytmie tworzonej aplikacji.



Rys. 5.2 Problem brakujących pikseli przy kontekstowym przetwarzaniu skrajnych punktów obrazu [15]

Zauważono również, że nie zawsze istnieje konieczność liczenia wszystkich odległości – bardzo często można skorzystać z wcześniej obliczonych wartości. Do podstawowego algorytmu została wprowadzona modyfikacja, dzięki której możliwe będzie wykorzystywanie wcześniej obliczonych odległości. Wszystkie odległości jakie występują w badanej masce są najpierw obliczane i zapamiętywane, a przy obliczaniu sum wykorzystuje się już gotowe wartości. Dodatkowo zauważono, że istnieje możliwość wykorzystania odległości obliczonych przy przetwarzaniu poprzednich pikseli. Prezentuje to rys. 5.3, na którym przedstawiono 9 różnych przypadków wraz z już wcześniej obliczanymi odległościami zaznaczonymi pogrubioną linią.

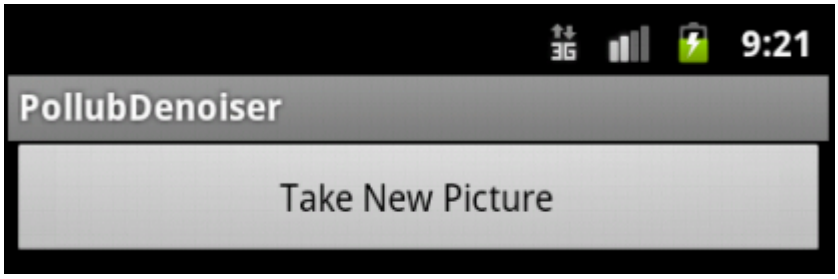


Rys. 5.3 Możliwości wykorzystywania odległości obliczonych przy przetwarzaniu wcześniejszych pikseli.

Jak widać dla wszystkich nieskrajnych pikseli można wykorzystać aż cztery wcześniej obliczone odległości. Opracowano więc algorytm przechowujący odległości, które mogą być później wykorzystane. W trójwymiarowej tablicy zapamiętywane są cztery odległości dla każdego piksela z dwóch ostatnio przetwarzanych wierszy – poprzedniego i obecnego. Przy obliczaniu odległości dla badanego piksela sprawdzane jest jakie wcześniej obliczone wartości mogą być wykorzystane, i pomija się dla nich obliczanie odległości po raz kolejny. Dzięki temu uzyskano ponad 30-procentowy wzrost wydajności.

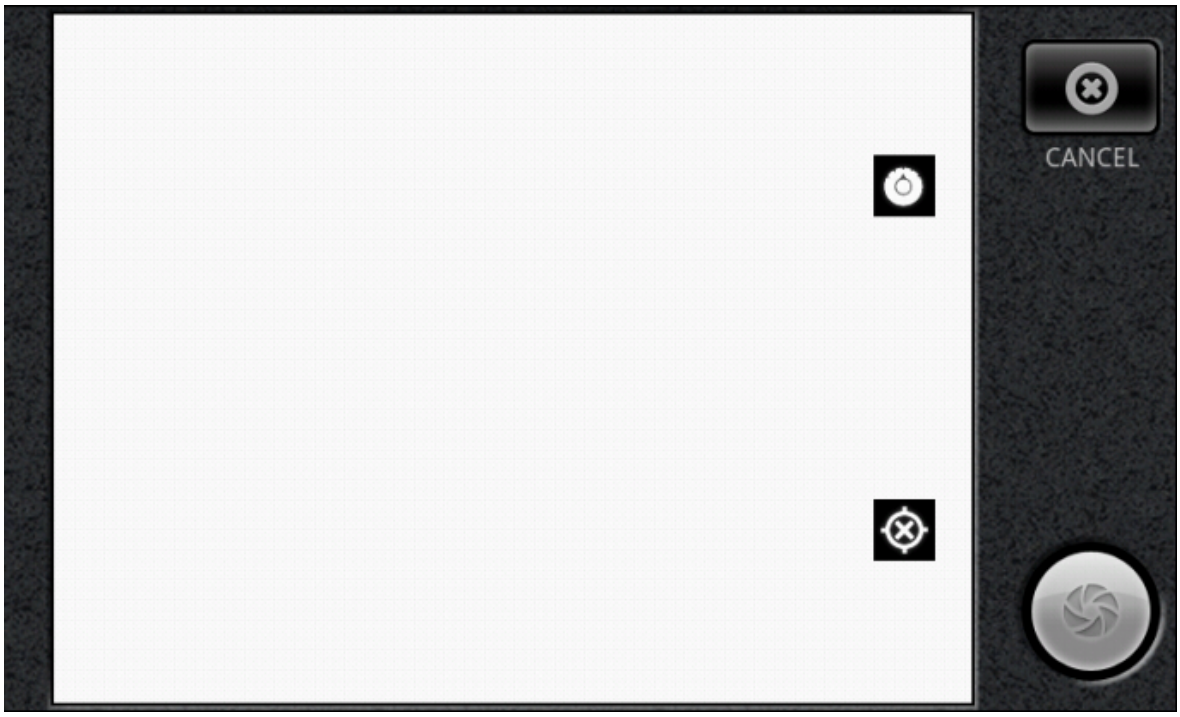
5.5. Opis aplikacji

Aplikacja udostępnia dwie podstawowe funkcje: robienie zdjęć oraz usuwanie z nich szumów. Po uruchomieniu aplikacji mamy do dyspozycji tylko jeden przycisk, widoczny na rys. 5.4.



Rys. 5.4 Ekran startowy aplikacji PollubDenoiser

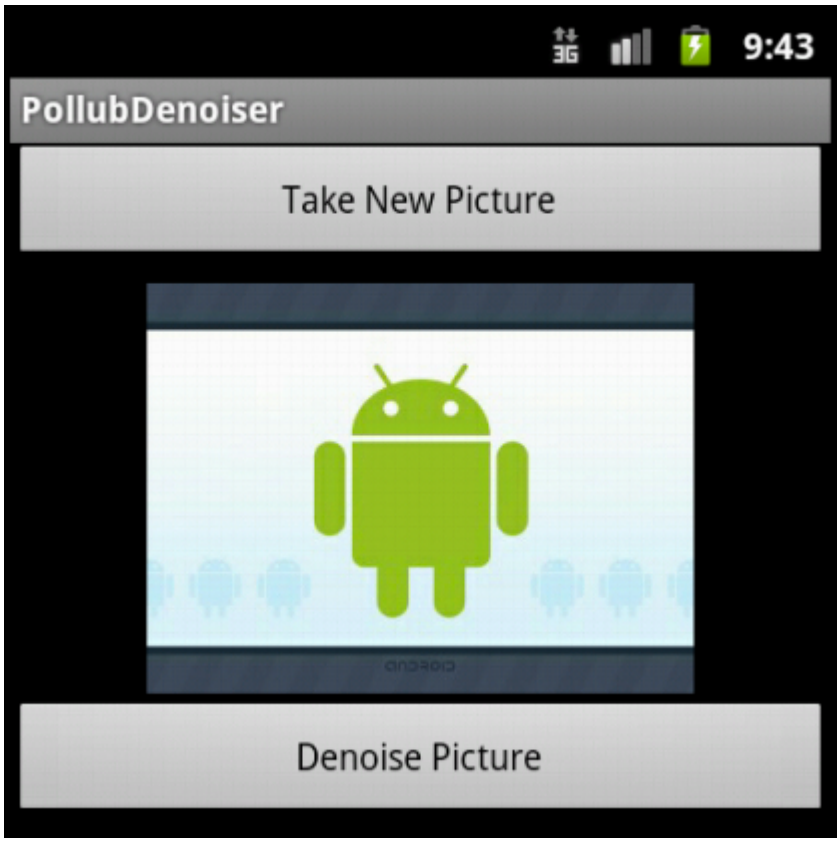
Po kliknięciu przycisku uruchamiana jest aplikacja do robienia zdjęć, widoczna na rys. 5.5.



Rys. 5.5 Przykładowy ekran aplikacji do robienia zdjęć

Uruchomiona aplikacja umożliwia zrobienie zdjęcia oraz jego podgląd, a także dostęp do podstawowych ustawień aparatu. Po zrobieniu zdjęcia zostaje ono zapisane i ponownie wyświetlany jest ekran aplikacji gdzie oprócz przycisku do zrobienia kolejnego zdjęcia, wyświetlana jest miniatura oraz kolejny przycisk umożliwiający odszumienie zdjęcia.

Po kliknięciu drugiego przycisku zmienia on swój kolor oraz następuje odszumienie zrobionego zdjęcia. W zależności od parametrów telefonu i wielkości zdjęcia, proces ten może trwać od kilku do nawet kilkunastu minut. Nie należy przerywać tego procesu, dopóki przycisk będzie miał zmieniony kolor, co widać na rys. 5.7. Po odszumieniu zdjęcie zostaje zapisane jako kolejny plik i wyświetla się ekran taki jak na rys. 5.6 z miniaturą już odszumionego zdjęcia. Z tego miejsca możemy ponownie odszumić aktualny obraz (proces można powtarzać aż do uzyskania zadowalającego efektu), lub zrobić kolejne zdjęcie. Efekty pracy programu zapisywane są jako kolejne pliki, dzięki czemu nie tracimy oryginalnego zdjęcia.

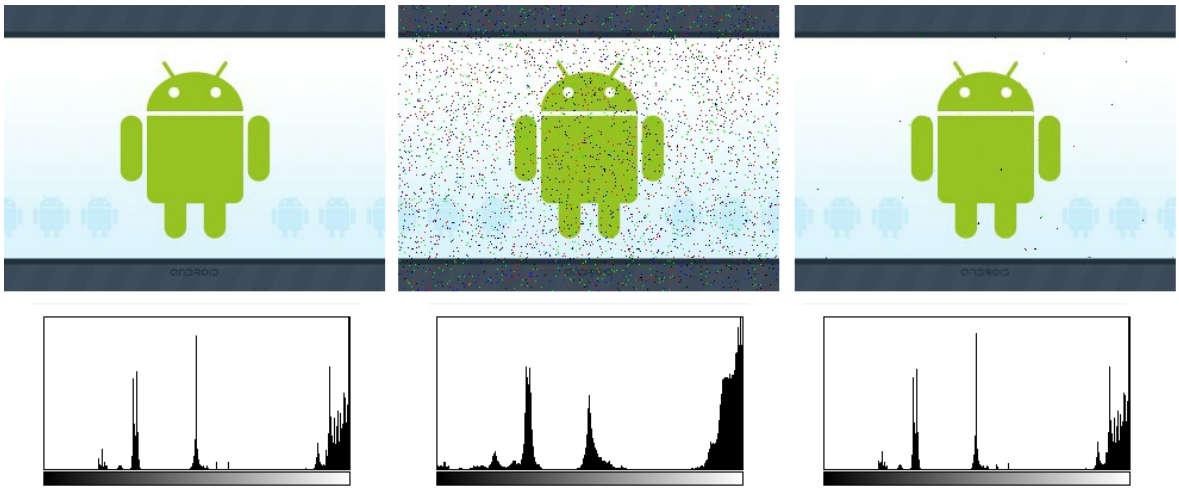


Rys. 5.6 Ekran aplikacji PollubDenoiser po zrobieniu zdjęcia.

Rys. 5.7 Przycisk do odszumiania zdjęcia w trakcie trwanie procesu odszumiania

5.6. Testy aplikacji

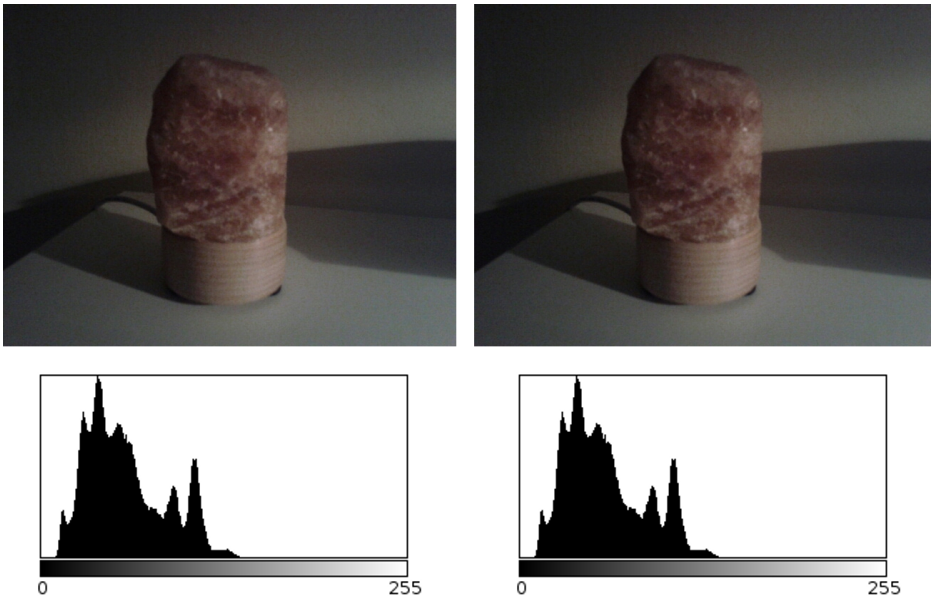
W celu sprawdzenia skuteczności stworzonej aplikacji, przeprowadzono dwuetapowe testy. Pierwszy typ testów odbył się na emulatorze. Komputer na jakim przeprowadzono testy został wyposażony w procesor Intel Core i3-370 oraz pracował pod systemem Ubuntu Linux 10.04 LTS. Na komputerze została uruchomiona maszyna wirtualna z ystemem Android w wersji 2.3.3. Ponieważ nie udało się zintegrować wbudowanej w omputer kamerki z emulatorem, testy zostały przeprowadzone na sztucznym obrazie, który został najpierw zakłócony 5-procentowym szumem impulsowym, a następnie odszumiony za pomocą aplikacji. Wyniki testów można zobaczyć na rys. 5.8.



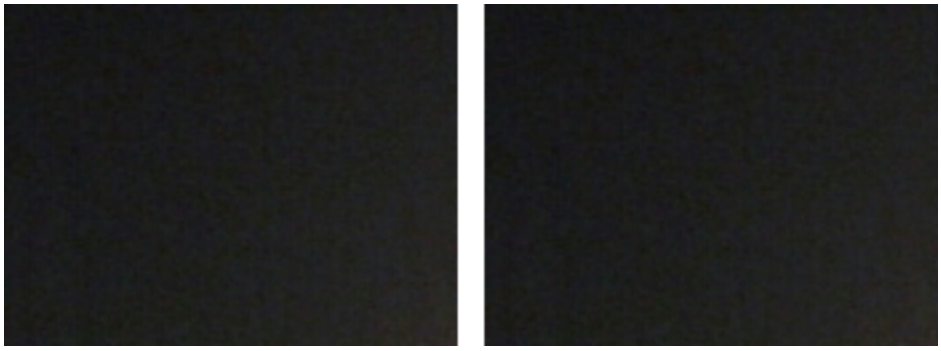
Rys. 5.8 Skutki filtracji oraz histogramy obrazów. Od lewej: obraz oryginalny, zaszumiony, odszumiony

Jak widać, szum został niemal całkowicie usunięty. Pogorszeniu nie uległa natomiast jakość zdjęcia. Problemem okazała się natomiast szybkość działania – dla zdjęcia o wymiarach 320×240 pikseli program potrzebował ponad pięć minut na pełne odszumienie obrazu.

Drugi etap testów polegał na sprawdzeniu aplikacji na prawdziwym telefonie – był to Nexus S z systemem Android w wersji 2.3.3. W tym wypadku aplikacja działała o wiele szybciej – dla obrazu o wymiarach 640×480 pikseli odszumianie trwało zaledwie 6 sekund. Problemem okazała się natomiast skuteczność algorytmu usuwającego szum. Wybrana 5-punktowa maska krzyżowa okazała się zbyt mała. Efekt jej działania można zobaczyć na rys. 5.9 i 5.10.

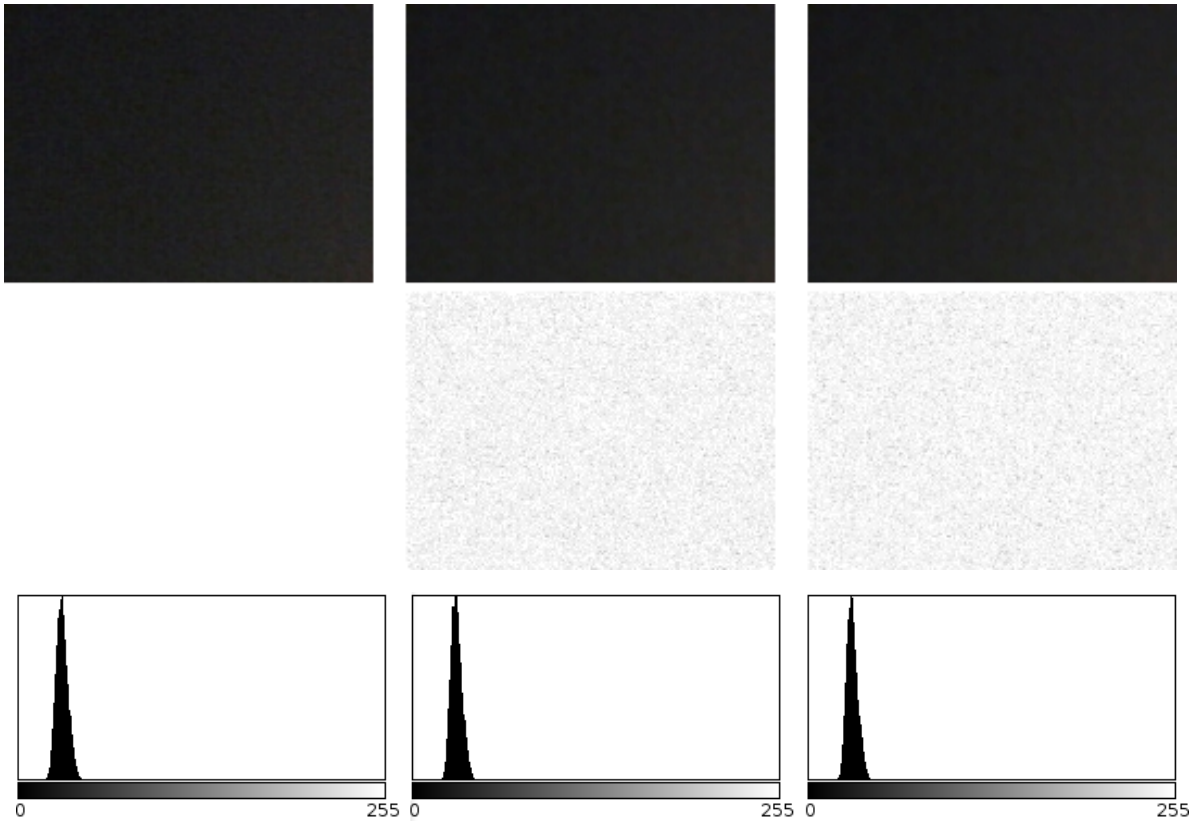


Rys. 5.9 Oryginalny obraz rzeczywisty (po lewej) i obraz po odszumieniu filtrem z 5-punktową maską krzyżową (po prawej)



Rys. 5.10 Powiększone fragmenty obrazu oryginalnego (po lewej) i odszumionego (po prawej)

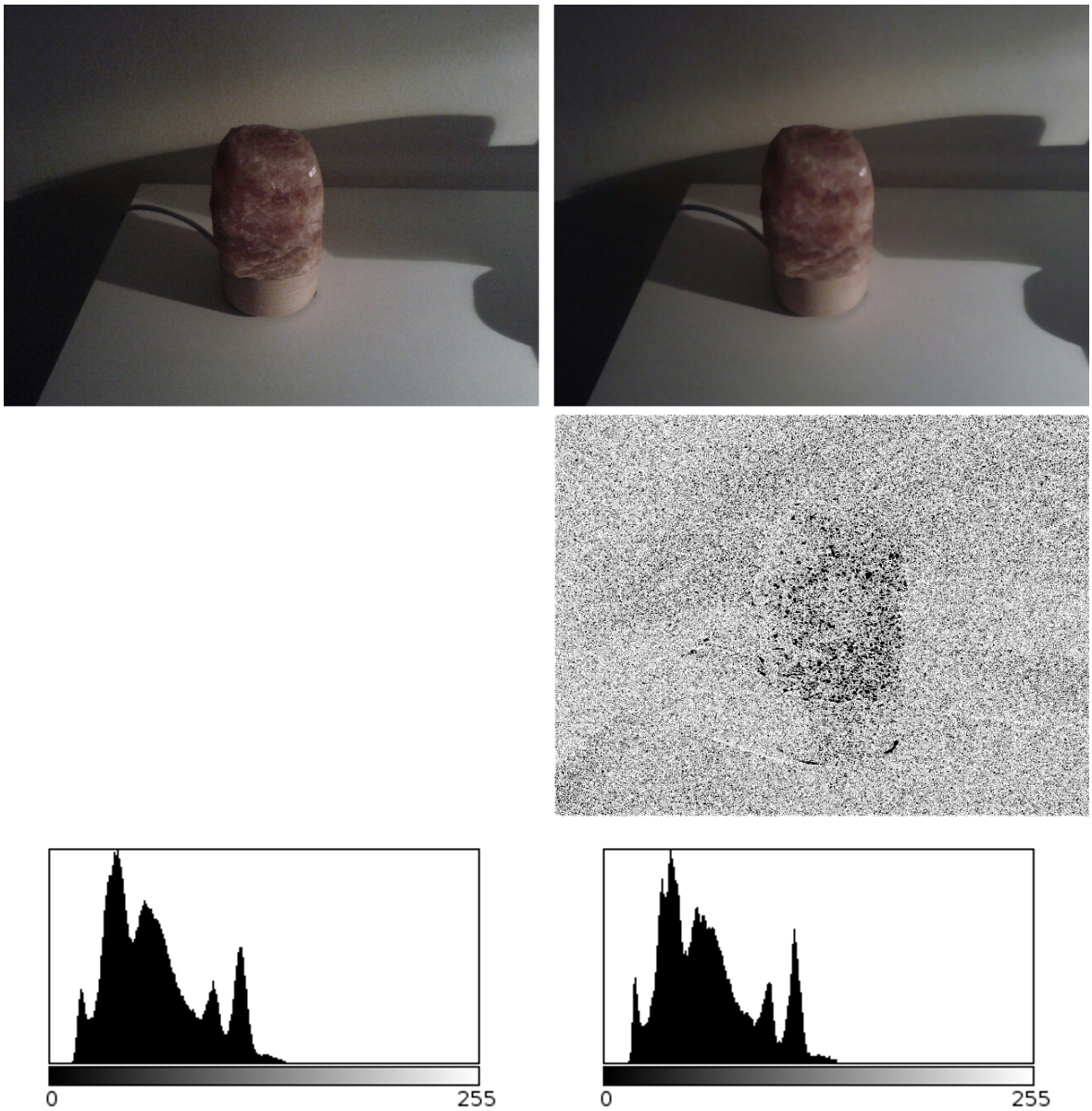
Jak widać, filtr niemal wcale nie wpłynął na zdjęcie. W celu zwiększenia jego skuteczności, maskę krzyżową zamieniono na maskę kwadratową o wymiarach 3×3. Oczywiście pociągnęło to za sobą konsekwencje w postaci wydłużenia czasu działania filtra. Dla zdjęcia o wymiarach 640×480 pikseli program potrzebował dwudziestu sekund na odszumienie. Efekt pracy filtra można zobaczyć na rys. 5.11.



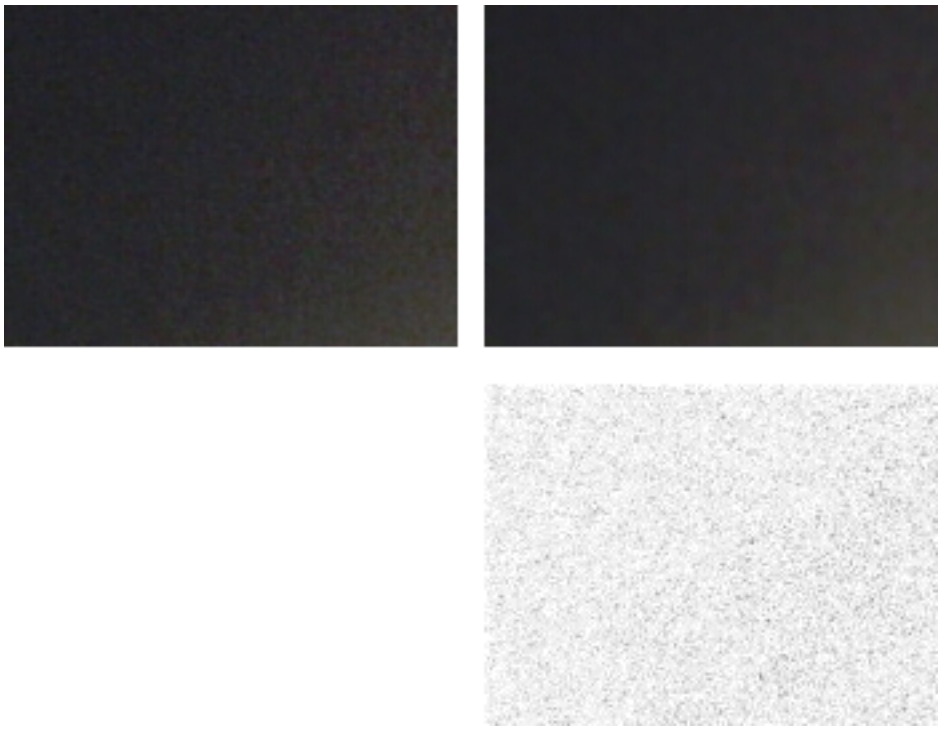
Rys. 5.11 Skutki działania filtra oraz uwypuklone różnice względem oryginalnego zdjęcia, od lewej: obraz oryginalny, po pierwszym przefiltrowaniu, po ponownym przefiltrowaniu.

Przy ustawionej masce z większą ilością punktów, filtr radził już sobie lepiej. Mimo to duża ilość szumów wciąż pozostawała na zdjęciu, nawet po dwukrotnym przefiltrowaniu. Rozmiar maski został więc zwiększony do wymiarów 5×5. Radykalnie

wydłużyło to czas działania skryptu. Dla zdjęcia o wymiarach 640×480 pikseli program potrzebował już ponad dwóch minut na przefiltrowanie obrazu. Efekty jednak wypadły dużo lepiej niż poprzednio, co można zobaczyć na rys. 5.12 i 5.13.



Rys. 5.12 Efekt działania filtra z maską 5×5: po lewej obraz oryginalny, po prawej po przefiltrowaniu, poniżej uwypuklona różnica pomiędzy obrazami



5.13 Efekt działania filtra z maską 5×5 w powiększeniu oraz uwypuklone różnice pomiędzy plikami: po lewej obraz oryginalny, po prawej po przefiltrowaniu

Jak widać filtr skutecznie zredukował wpływ szumów na obraz. Dużym problem okazał się jednak czas jego działania. Dla niedużego zdjęcia z około 300 tysiące pikseli były to aż dwie minuty, co już staje się uciążliwe dla użytkownika. Przy większych zdjęciach problem staje się jeszcze bardziej widoczny: dla zdjęcia z dwoma milionami pikseli potrzebne będzie około 14 minut na przefiltrowanie zdjęcia, natomiast dla pięciu milionów pikseli potrzeba ponad pół godziny.

Kolejnym problem okazały się być ograniczenia dotyczące ilości pamięci RAM, jaką może wykorzystywać aplikacja. Dla większości urządzeń wynosi to około 15 megabajtów. Jest to ilość zdecydowanie zbyt mała, jeśli w grę wchodzi przetwarzanie obrazów. W przypadku bitmapy każdy piksel zapisany jest na 4 bajtach. Przy zdjęciu posiadającym 4 miliony pikseli, nie ma możliwości wczytania go do pamięci urządzenia, gdyż jako bitmapa zajmuje 16 megabajtów pamięci. Uwzględniając fakt, że konieczne jest również przechowywanie dwóch wersji bitmap: oryginalnej i po przefiltrowaniu, problem pojawia się już przy zdjęciach z dwoma milionami pikseli. W rezultacie aplikacja sprawdza się tylko przy niewielkich zdjęciach, posiadających mniej niż 2 miliony pikseli. Biorąc pod uwagę fakt, że testowany telefon posiada aparat o rozdzielczości 5 megapikseli, jest to duże ograniczenie.

6. Wnioski

W powyższej pracy zostały przedstawione metody redukcji szumów w obrazach cyfrowych, opis systemu Android, a także wstęp do tworzenia aplikacji na niego. Efektem prac była aplikacja dedykowana na system Android, redukująca szumy w zdjęciach robionych kamerką telefonu.

W wyniku analizy literatury oraz wstępnych testów, do odszumienia obrazów został wybrany filtr FMVMF bazujący na medianie. Okazał się on najszybszy spośród testowanych filtrów, przy jednoczesnym zachowaniu dobrej skuteczności przy redukcji szumów. Do obliczania różnic pomiędzy kolorami zastosowano model RGB oraz metrykę miejską. Opracowana została również metoda umożliwiająca zwiększenie wydajności poprzez wykorzystywanie wcześniejszych obliczeń. Po testach aplikacji przy różnych rozmiarach maski ostatecznie została wybrana maska o wymiarach 5×5. Mimo dużego kosztu obliczeniowego, była ona bardzo skuteczna w usuwaniu szumu ze zdjęć.

W celu ograniczenia zniekształceń powodowanych kompresją zrobionego zdjęcia do stratnego formatu JPEG, do aplikacji dodano możliwość robienia zdjęć, dzięki czemu możliwe było bezpośrednie przekazanie zdjęcia z aparatu do aplikacji, z pominięciem kompresji. W tym celu zostało wykorzystane gotowe narządzie udostępniane przez system, dzięki czemu znacznie uproszczono logikę aplikacji, gdyż obsługą aparatu zajmowała się zewnętrzna aplikacja. Aplikacja została przetestowana zarówno na emulatorze udostępnianym razem z SDK Androida, jak i rzeczywistym telefonie, a wyniki testów umieszczone w pracy.

Biorąc pod uwagę ograniczenia dotyczące pamięci oraz szybkość procesora można wysnuć wniosek, że obecne telefony z systemem Android nie są gotowe na przetwarzanie obrazów cyfrowych. Problemem jest długi czas przetwarzania oraz ograniczenia co do wielkości zdjęcia wymuszane przez ilość pamięci, jaką może wykorzystywać pojedyncza aplikacja.

Literatura

1. Angulo J., Serra J.: *Morphological coding of color images by vector connected filters*, Seventh International Symposium on Signal Processing and Its Applications, Fontainebleau, 2003, pages: 69 – 72.
2. Bieniecki W., Grabowski S.: *Wybrane zagadnienia przetwarzania i analizy obrazów mikroskopowych w diagnostyce medycznej*, Zeszyt naukowy X Lat KIS, Łódź, 2005.
3. Bieniecki W.: *Nowoczesne algorytmy przetwarzania obrazów w wizyjnych systemach komputerowych wspomagających diagnostykę patomorfologiczną*, Praca doktorska, Politechnika Łódzka, Łódź, 2005.
4. Buczyński P.: *Optymalna reprezentacja kolorów w analizie i przetwarzaniu obrazów komputerowych*, Praca doktorska, Politechnika Warszawska, Warszawa, 2005.
5. Coetzee D.: *An efficient implementation of Blum, Floyd, Pratt, Rivest, and Tarjan's worst-case linear selection algorithm*, 2004, http://moonflare.com/code/select/select.pdf [Dostęp: 25.01.2011].
6. Haseman Ch.: *Android Essentials*, firstPress, New York, 2008.
7. Hashimi S., Komatineni S., MacLean D.: *Android 2. Tworzenie aplikacji*, Helion, Gliwice, 2010.
8. Hashimi S., Komatineni S., MacLean D.: *Pro Android 2,* Apress, New York, 2010.
9. Kirschenhofer P., Prodinger H., Martınez C.: *Analysis of Hoare's Find Algorithm with Median-of-three partition*, Random Structures & Algorithms vol. 10, Barcelona, 1996, pages: 143–156.
10. Meier R.: *Professional Android 2 Application Development*, Wrox, Indianapolis, 2010.
11. Meier R.: *Professional Android Application Development*, Wrox, Indianapolis 2009.
12. Murphy M.: *Beginning Android 2*, Apress, New York, 2010.

13. Smolka B. i inni: *Fast Modified Vector Median Filter*, II Konferencja „Komputerowe Systemy Rozpoznawania" KOSYR2001, Miłków k/Karpacza, 2001, str. 225–232.
14. Stoliński S., Grabowski S.: *Eksperymentalne porównanie filtrów medianowych do usuwania szumów impulsowych z obrazów barwnych*, Automatyka, Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie, rok 2005, tom 9, zeszyt 3, strony: 571—585.
15. Tadeusiewicz R., Korohoda P.: *Komputerowa analiza i przetwarzanie obrazów*, Wydawnictwo Fundacji Postępu Telekomunikacji, Kraków, 1997.

Materiały uzupełniające:

I. *What is Android?,* http://developer.android.com/guide/basics/what-is-android.html [Dostęp: 09.01.2012]
II. *Google Android Market Tops 400,000 Applications,* http://www.distimo.com/blog/2012_01_google-android-market-tops-400000-applications/ [Dostęp: 07.01.2012]
III. *10 Billion Android Market downloads and counting,* http://googleblog.blogspot.com/2011/12/10-billion-android-market-downloads-and.html [Dostęp: 07.01.2012]
IV. *How many Android phones have been activated?,* http://www.asymco.com/2011/12/21/how-many-android-phones-have-been-activated/ [Dostęp: 08.01.2012]
V. *First Interactive Android App,* http://www.annotate.com.au/2010/10/25/first-interactive-android-app/ [Dostęp: 12.01.2012]
VI. *Capturing Photos: Taking Photos Simply,* http://developer.android.com/training/camera/photobasics.html [Dostęp: 22.01.2012]
VII. Andy Rubin: *There are now over 700,000 Android devices activated every day,* http://twitter.com/#!/Arubin/status/149329329237667844 [Dostęp: 07.01.2012]
VIII. *Industry Leaders Announce Open Platform for Mobile Devices,* http://www.openhandsetalliance.com/press_110507.html [Dostęp: 07.01.2012]
IX. *Android Developer Challenge,* http://code.google.com/intl/pl/android/adc/ [Dostęp: 07.01.2012]

X. *Images for Noise Reduction*, http://cms.daegu.ac.kr/ytdo/Teaching/sensor/sensor.html [Dostęp: 26.01.2012]

XI. *Multimedia and Camera,* http://developer.android.com/guide/topics/media/index.html [Dostęp: 27.01.2012]

XII. Tomasz Lubiński: *Filtrowanie obrazów,* http://www.algorytm.org/przetwarzanie-obrazow/filtrowanie-obrazow.html [Dostęp: 28.01.2012]

OŚWIADCZENIE

Niniejszym oświadczam, że zapoznałem się z przepisami wynikającymi z **Ustawy o prawie autorskim i prawach pokrewnych" (Dz. U. Nr 24, poz 83 z dnia 23.02.1994 r. z późn. zm)**. Przedkładaną pracę dyplomową inżynierską pt. „*Opracowanie aplikacji służącej do filtrowania szumów w obrazach wykonanych kamerką telefonu*" napisałem samodzielnie.

.....

(imię i nazwisko – podpis)

.....

Lublin, dnia