<table>
<tr><td colspan="4">Politechnika Poznańska
Wydział Elektryczny</td></tr>
<tr><td colspan="4">Kierunek: Informatyka</td></tr>
<tr><td colspan="4">Przedmiot: Podstawy Teleinformatyki</td></tr>
<tr><td>Data:</td><td>Temat projektu:</td><td>Grupa:</td><td>Ocena:</td></tr>
<tr><td rowspan="3">27.01.2018</td><td rowspan="3">Malowanie obrazów biorąc pod uwagę ruch ciała</td><td>Bartosz Nędzewicz</td><td rowspan="3"></td></tr>
<tr><td>Paweł Pluta</td></tr>
<tr><td></td></tr>
</table>

Spis treści

1.Założenia projektu....3
2.Wybór tematu projektu....3
3.Platformy i język programowania....3
4.Narzędzia....4
5.Ramy projektowo – czasowe....4
6.Ogólna budowa programu....5
6.1.Zastosowanie architektury MVC: Model....6
6.2.Zastosowanie architektury MVC: Widok....8
6.3.Zastosowanie architektury MVC: Kontroler....9
7.Instrukcja użytkownika aplikacji....10
8.Interesujące problemy oraz ich rozwiązania....14

1. Założenia projektu

Głównym celem projektu było napisanie programu pozwalającego na rysowanie dowolnych kształtów na ekranie komputera przy pomocy ruchu dłoni. Do sterowania położeniem kolejnych punktów rysunku użyto obiektu o wybranym kolorze, w omawianym przypadku był to kolor niebieski (wartość odcienia światła pomiędzy 105 a 135, nasycenie koloru 100 – 255 i mocy światła białego 100 – 255).

Zadanie zostało wykonane w dwuosobowym zespole. Współpraca odbywała się poprzez korzystanie ze wspólnego repozytorium umieszczonego na serwerze GitHub. Finalna wersja projektu wraz z historią poszczególnych zmian oraz niniejszą dokumentacją dostępna jest pod adresem URL: www.github.com/novir/air-painter.

2. Wybór tematu projektu

Temat projektu został wybrany w oparciu o dużą liczbę dostępnych materiałów oraz samouczków. Większość z w/w źródeł prezentowała implementacje poszczególnych rozwiązań w takich językach programowania jak C/C++ i Python. Składnia języka Java oparta jest na składni rodziny języków C/C++, co ułatwia jej zrozumienie programistom tego języka, z kolei język Python, mimo że odmienny składniowo, został zaprojektowany z myślą o jego czytelności. Z tego powodu również on nie nastręcza zbyt wielu problemów jeśli chodzi o zrozumienie przykładów zaimplementowanych przy jego użyciu.

3. Platformy i język programowania

Program dedykowany jest na dwie platformy: **Microsoft Windows** oraz systemy **GNU/Linux**. Wybór dwóch platform podyktowany był różnymi preferencjami poszczególnych autorów, co do używanego środowiska programistycznego oraz systemu operacyjnego. Z uwagi na powyższe, w celu uzyskania przenośności pomiędzy poszczególnymi systemami operacyjnymi zastosowany został język programowania ***Java (JDK 8)***. Został on wybrany po obustronnej aprobacie autorów tworzących w/w oprogramowanie.

Kolejnym istotnym powodem wyboru Javy była dostatecznie dobra znajomość tego języka programowania, ze uwagi na mnogość zadań wykonywanych przy jego użyciu w przeszłości. Pomimo dużej dostępności w internecie samouczków, których autorzy wykorzystywali w swoich rozwiązaniach takie języki programowania jak C/C++ czy Python, możliwość zastosowania jednego z tych języków ograniczała ich niedostateczna znajomość, w przypadku C/C++, lub całkowity brak znajomości – Python.

Aplikacja powstała przy wykorzystaniu technologii JavaFX. Zastosowano ją ze względu na walory estetyczne, nowoczesny wygląd, a także łatwość projektowania przy użyciu środowiska Scene Builder opartego na znacznikach XML.

4. Narzędzia

- Java JDK 8,
- NetBeans IDE 8.2 (Bartosz),
- IntelliJ IDEA 2017.1.5 (Paweł),
- Biblioteka: OpenCV 3.3.0 (wersja dedykowana dla języka Java),
- JavaFX Scene Builder 9.0.1,
- Git wersja 2.11.0;

5. Ramy projektowo – czasowe

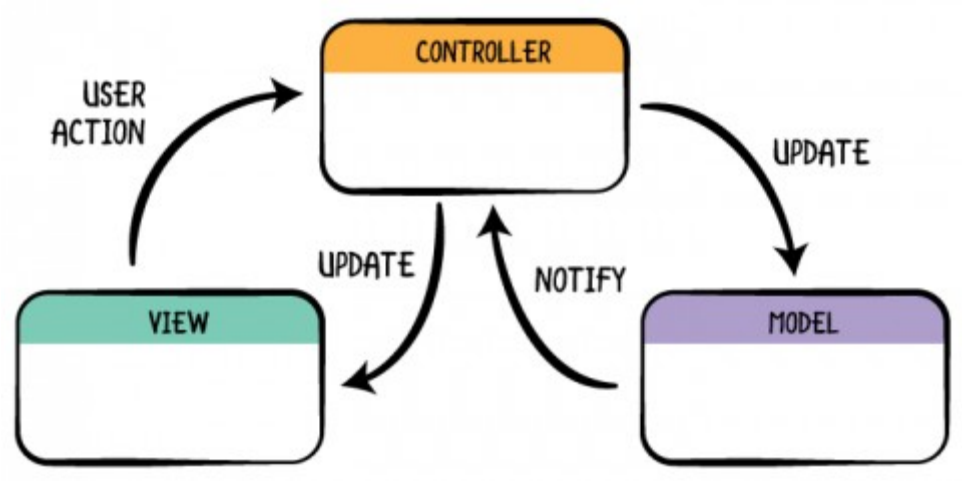
Projekt podzielono na kilka faz, będącymi krokami milowymi do ukończenia zadania.

- *Październik ’17* – wstępne zapoznanie się z tematem, wybór języka programowania i środowiska,
- *Listopad ’17* – nauka podstaw JavaFX, stworzenie szaty graficznej,
- *Grudzień ’17* – analiza zastosowanych technologii (OpenCV), uruchomienie kamery,
- *Styczeń ’18* – analiza przechwytywanego obrazu, wykrycie ruchu,
- *Luty ’18* – oddanie gotowego projektu, finalizacja dokumentacji;

6. Ogólna budowa programu

Program powstał z myślą o implementacji wzorca projektowania Model-View-Controller (MVC), który jest jednym z głównych wzorców służących do organizowania struktury aplikacji posiadających graficzny interfejs użytkownika. Model ten zakłada podział prac deweloperskich na trzy główne sekcje:

- Model – będący reprezentacją problemu/logiki aplikacji,
- Widok – opisujący metodę wyświetlania interfejsu użytkownika oraz definiujący zasadę jego obsługi,
- Kontroler – obsługuje dane podawane przez użytkownika i reaguje na wykonywane przez niego akcje;



Rysunek 1 Ogólny zarys architektury MVC. Źródło: internet/Google Images

Na rysunku powyżej zaprezentowano działanie przykładowej aplikacji opartej na wzorcu MVC. Widać wyraźnie, iż nie ma bezpośredniego połączenia pomiędzy klasami modelu a widokiem aplikacji. Wszelka interakcja tychże warstw odbywa się za pośrednictwem warstwy kontrolera. To właśnie kontroler zarządza zmianami widoku, w odpowiedzi na akcje wykonywane przez użytkownika, lub z uwagi na zmianę stanu sygnalizowaną przez model. Za jego pośrednictwem aktualizowany jest także stan samego modelu.

6.1. Zastosowanie architektury MVC: Model

Odzwierciedleniem struktury Model-View-Controller w projekcie jest struktura i podział plików, które składają się na realizację omawianego zagadnienia:

Model – pierwszy i zarazem najobszerniejszy element reprezentowany jest przez klasy:

- FrameGrabber.java – klasa odpowiedzialna za interakcję programu z kamerą. W tym celu użyto w niej klasy VideoCapture z biblioteki OpenCV. Klasa umożliwia:
  - utworzenie nowego obiektu reprezentującego kamerę,
  - ponowne uruchomienie wcześniej zwolnionej kamery,
  - zwolnienie nieużywanej kamery,
  - sprawdzenie czy kamera działa poprawnie,
  - pobranie następnej klatki obrazu z kamery,
  - dostosowanie poziomu jasności kamery,
  - ustawienie parametrów obrazu.

- FrameConverter.java – klasa odpowiedzialna za konwertowanie obrazu z kamery, zwracanego w formie macierzy (klasa Mat), do klas java.io.InputStream oraz javafx.scene.image.Image.

- ObjectTracker.java – klasa odpowiedzialna za odszukiwanie na obrazie niebieskiego obiektu. Implementuje ona rozwiązania umożliwiające:
  - konwersję kolorów obrazu z przestrzeni BGR do HSV,
  - konwersję kolorów obrazu do odcieni szarości,
  - progowanie obrazu, wyłapując na nim obiekt o określonym kolorze,
  - progowanie adaptacyjne, zależne od wartości oświetlenia na obrazie,
  - progowanie statycznego otoczenia, odróżnianie go od dynamicznego elementu pierwszoplanowego,
  - zastosowanie na obrazie, uprzednio progowanym jedną z powyższych metod, morfologicznego otwarcia, zamknięcia, a także dylacji,
  - znajdowanie współrzędnych śledzonego obiektu,
  - znajdowanie konturów śledzonego obiektu zależnych od podanej wielkości minimalnej,
  - wyliczanie centralnego punktu śledzonego obiektu.

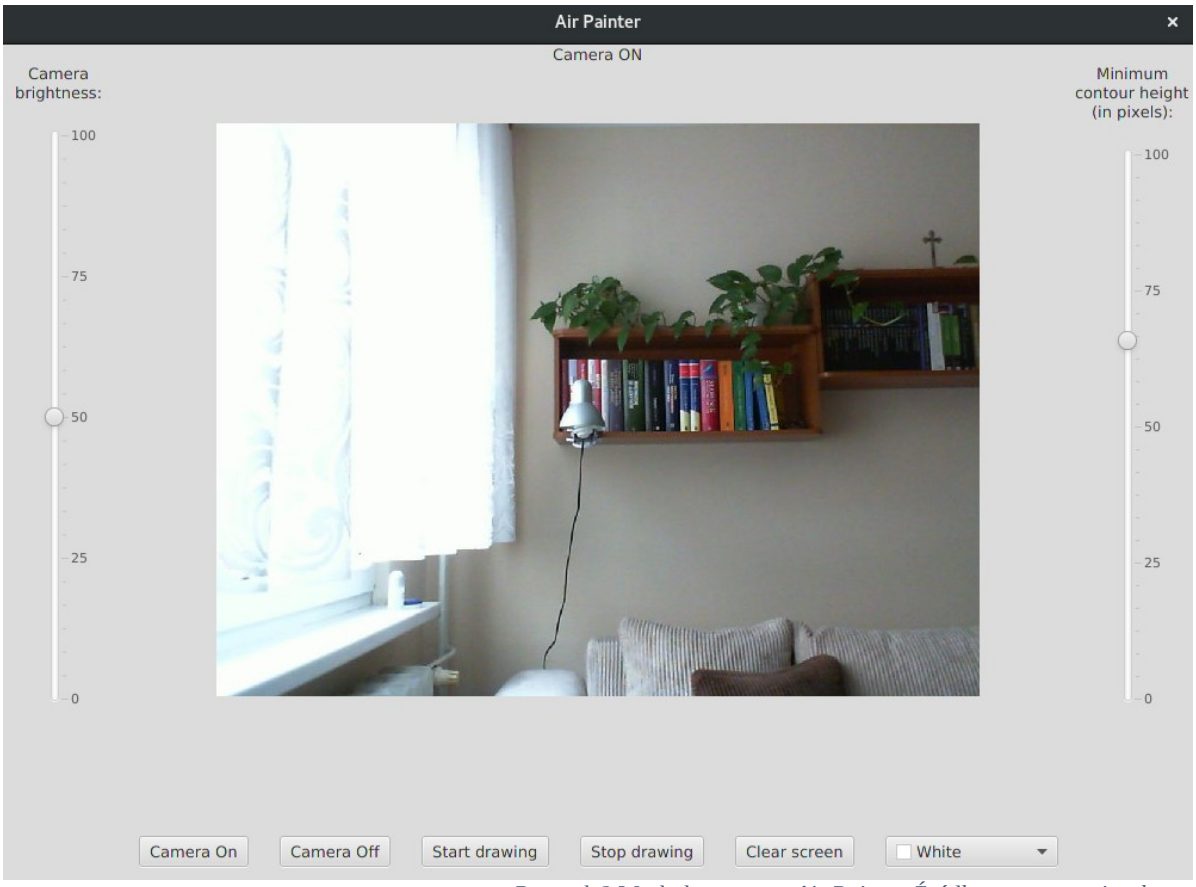
Podstawowa metoda powyższej klasy, odpowiedzialna za wyszukiwanie obiektów przyjmuje postać:

```
public Mat findBlueObject(Mat source) {
    Mat processed = convertBGRToHSB(source);
    processed = performBlueObjectThreshold(processed);
    processed = subtractBackground(processed);
    processed = performAdaptiveThreshold(processed);
    processed = applyMorphologicalOpening(processed, 5);
    processed = applyMorphologicalClosing(processed, 5);
    processed = applyDilation(processed, 10);
    return processed;
}
```

- FramePainter.java – klasa odpowiedzialna za rysowanie różnych kształtów na podanym jako argument obrazie. W omawianej implementacji obiekt klasy ma możliwość:
  - dodawania czerwonych okręgów wokół wskazanego punktu, umożliwia to rysowania okręgu wskazującego aktualne położenie śledzonego obiektu,
  - gromadzenia w kolekcji, będącej zbiorem punktów, współrzędnych poszczególnych punktów malowanego obrazu,
  - rysowania na podstawie zgromadzonego zbioru punktów rysunku na obrazie z kamery,
  - dostarcza także metodę usuwającą wszystkie zgromadzone wcześniej punkty, co powoduje usuniecie rysunku z obrazu.

6.2. Zastosowanie architektury MVC: Widok

Widok – drugi element, reprezentowany jest przez dwa pliki: gui_elements.fxml i settings.css. Interfejs graficzny został wykonany przy użyciu technologii JavaFX z użyciem notacji FXML. Do zaprojektowania interfejsu posłużono się oprogramowaniem Scene Builder. W celu uzupełnienia szaty graficznej wykorzystano także technologię CSS. Ostatecznie aplikacja przyjęła następujący wygląd:



Rysunek 2 Wygląd programu Air Painter. Źródło: opracowanie własne

6.3. Zastosowanie architektury MVC: Kontroler

Kontroler – trzeci element, który definiuje zachowania przycisków na scenie, a także odpowiada za zarządzanie poszczególnymi obiektami klas modelu. Opisywana warstwa aplikacji reprezentowana jest przez następujące klasy:

- UIController.java – obiekt tej klasy przechowuje w swoich polach poszczególne elementy widoku takie jak:
  - okno wyświetlające obraz z kamery,
  - przyciski,
  - suwaki,
  - przypisane do nich etykiety.

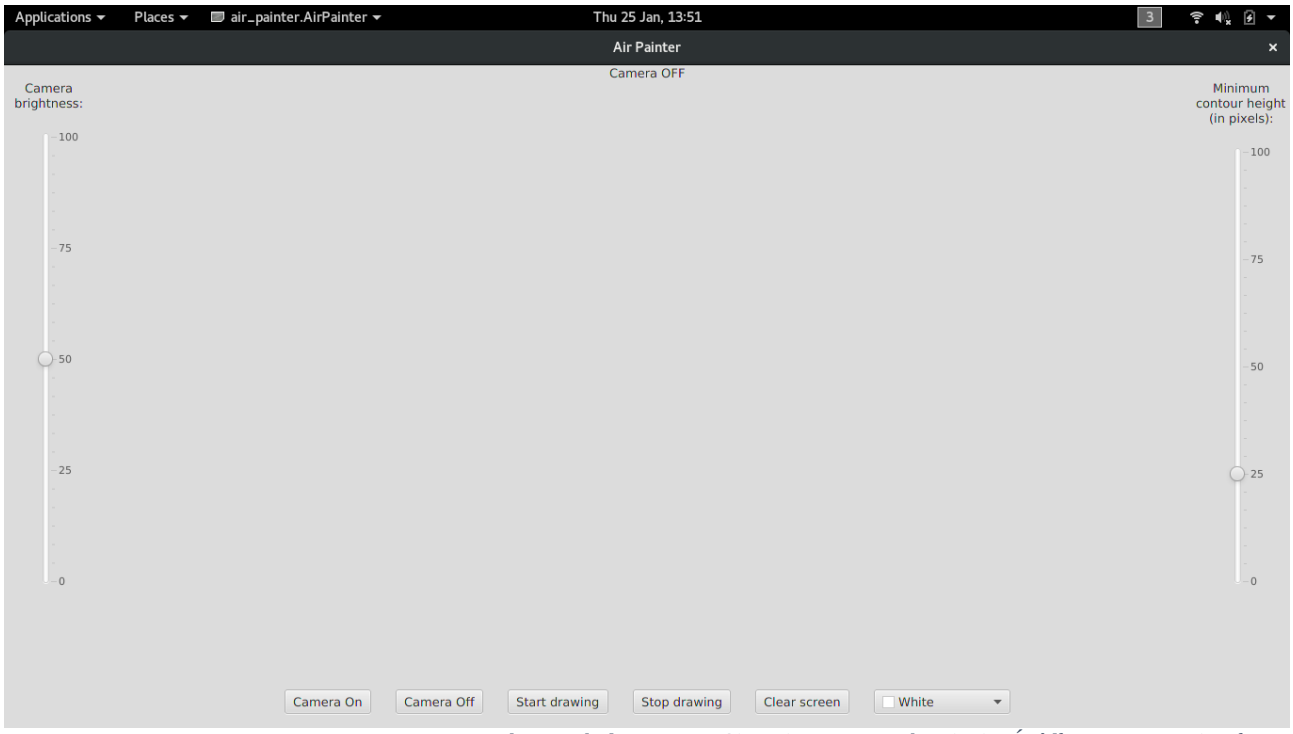
  Także w tej klasie znajdują się metody wywoływane w odpowiedzi na akcje związane z kliknięciem poszczególnych przycisków panelu głównego, takie jak:
  - włączenie kamery,
  - wyłączenie kamery,
  - rozpoczęcie rysowania na ekranie,
  - zakończenie rysowania na ekranie,
  - usunięcie rysunku.

- VideoController.java – obiekt tej klasy zarządza poszczególnymi obiektami wchodzącymi w skład modelu programu. Stanowi on także swego rodzaju pomost pomiędzy kontrolerem widoku aplikacji a obiektami modelu. Zaimplementowano w nim takie funkcjonalności jak:
  - uruchamianie pobierania obrazu z kamery w osobnym wątku,
  - pobieranie współrzędnych śledzonego obiektu na podstawie obrazu z kamery,
  - malowanie na obrazie z kamery, w oparciu o uzyskane wcześniej współrzędne obiektu,
  - przesyłanie przetworzonego obraz z kamery do kontrolera widoku aplikacji,
  - zabezpieczenie działania aplikacji na wypadek nie podłączenia kamery przez użytkownika,
  - kończenie, uruchomionego wcześniej, dodatkowego wątku programu,
  - zwalnianie kamery,
  - przekazywanie parametrów do obiektu reprezentującego kamerę oraz do obiektu klasy śledzącej artefakt,
  - zarządzanie rysowaniem na ekranie.

Nadrzędną klasą programu jest klasa AirPainter.java, która odpowiada za wygenerowanie interfejsu użytkownika, na podstawie podanego pliku FXML. Dane w formacie XML, zawarte w pliku gui_elements.fxml, służą do załadowania całej hierarchii elementów interfejsu użytkownika. Wykorzystano do tego klasę javafx.fxml.FXMLLoader, podając jej jako element nadrzędny (root) układ komponentów reprezentowany przez klasę javafx.scene.layout.BorderPane. Dodatkowo w klasie AirPainter.java wiązane są między sobą poszczególne kontrolery aplikacji.

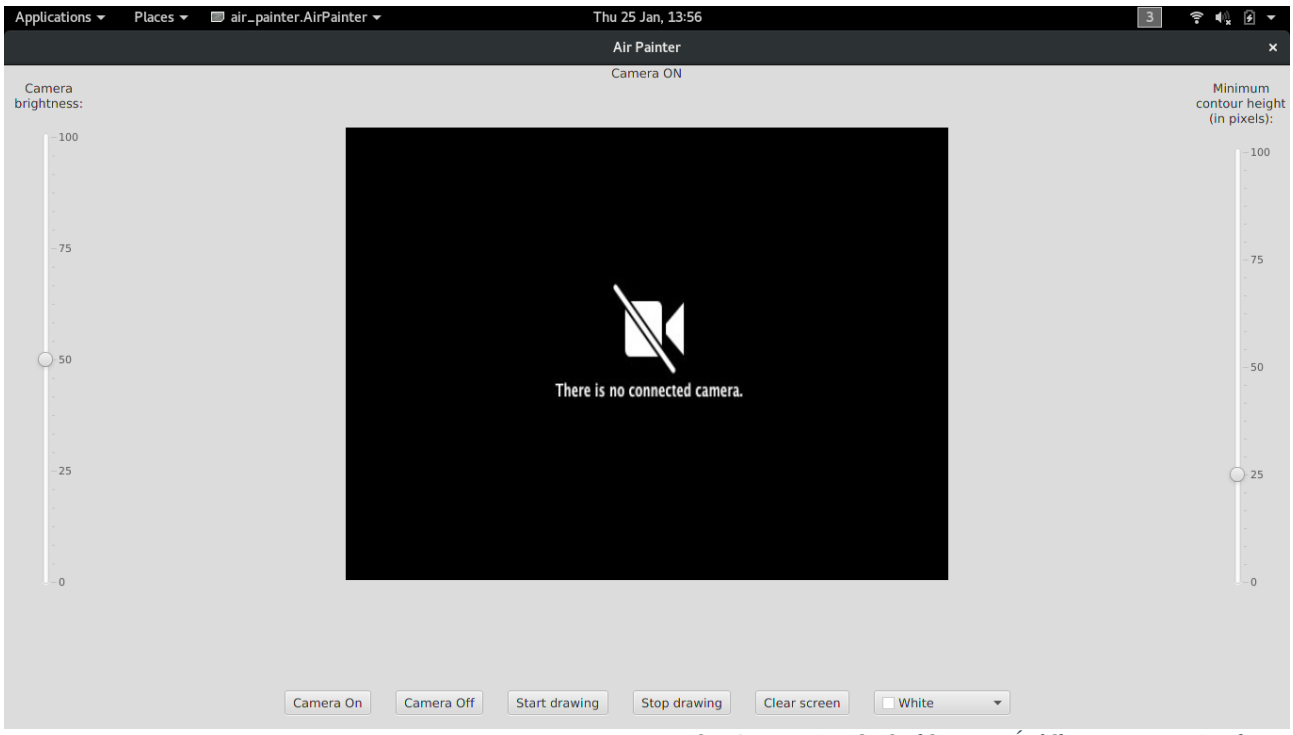
7. Instrukcja użytkownika aplikacji

Po uruchomieniu ekran aplikacji wygląda następująco:



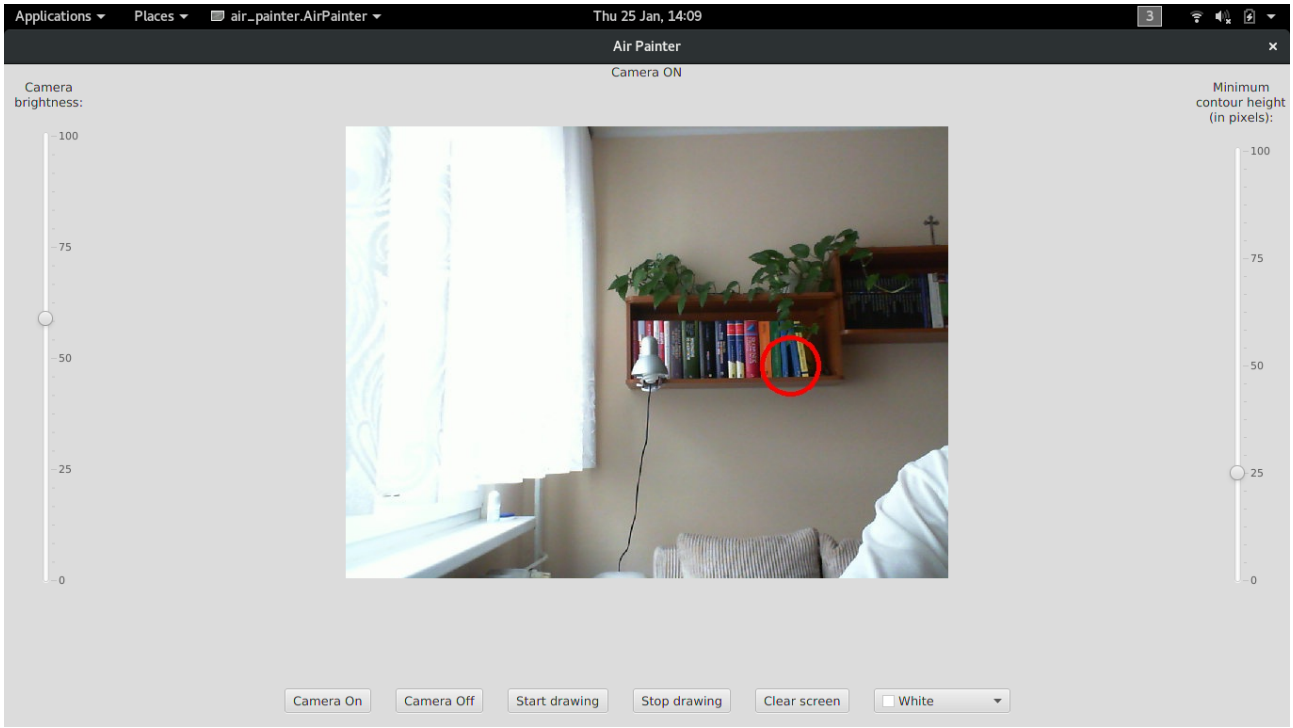
Rysunek 3 Wygląd programu Air Painter po uruchomieniu. Źródło: opracowanie własne

Na rysunku nr 3 pokazano okno aplikacji przed uruchomieniem kamery. W celu uruchomienia kamery użytkownik powinien nacisnąć przycisk „Camera On”. W przypadku gdy kamera nie została podłączona lub nastąpił inny problem z odnalezieniem przez system operacyjny domyślnej kamery aplikacja wyświetla informację o braku kamery. Przykład takiej sytuacji pokazano na rysunku nr 4.



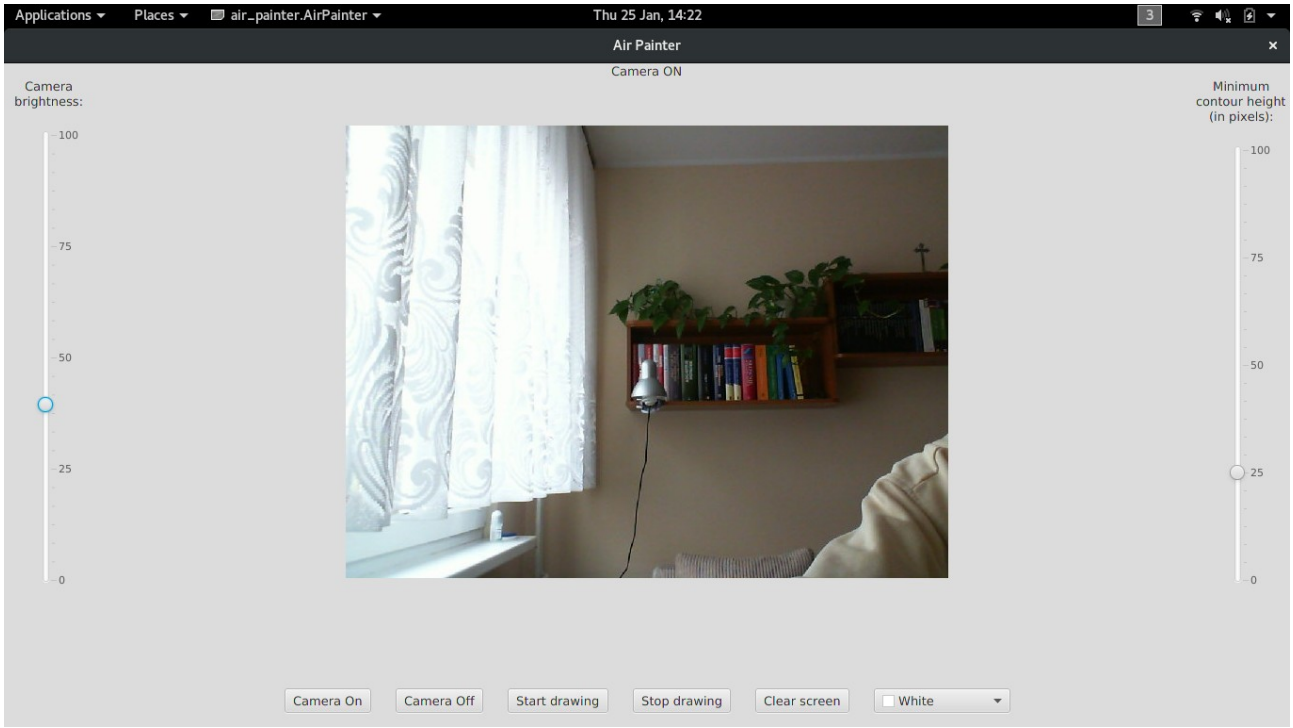
Rysunek 4 System nie odnalazł kamery. Źródło: opracowanie własne

Do zatrzymania pracy kamery służy z kolei przycisk „Camera Off”. Po uruchomieniu kamery aplikacja automatycznie przystępuje do wykrywania niebieskich obiektów. Wykryty obiekt zaznaczany jest czerwonym okręgiem. Pokazuje to rysunek nr 5.



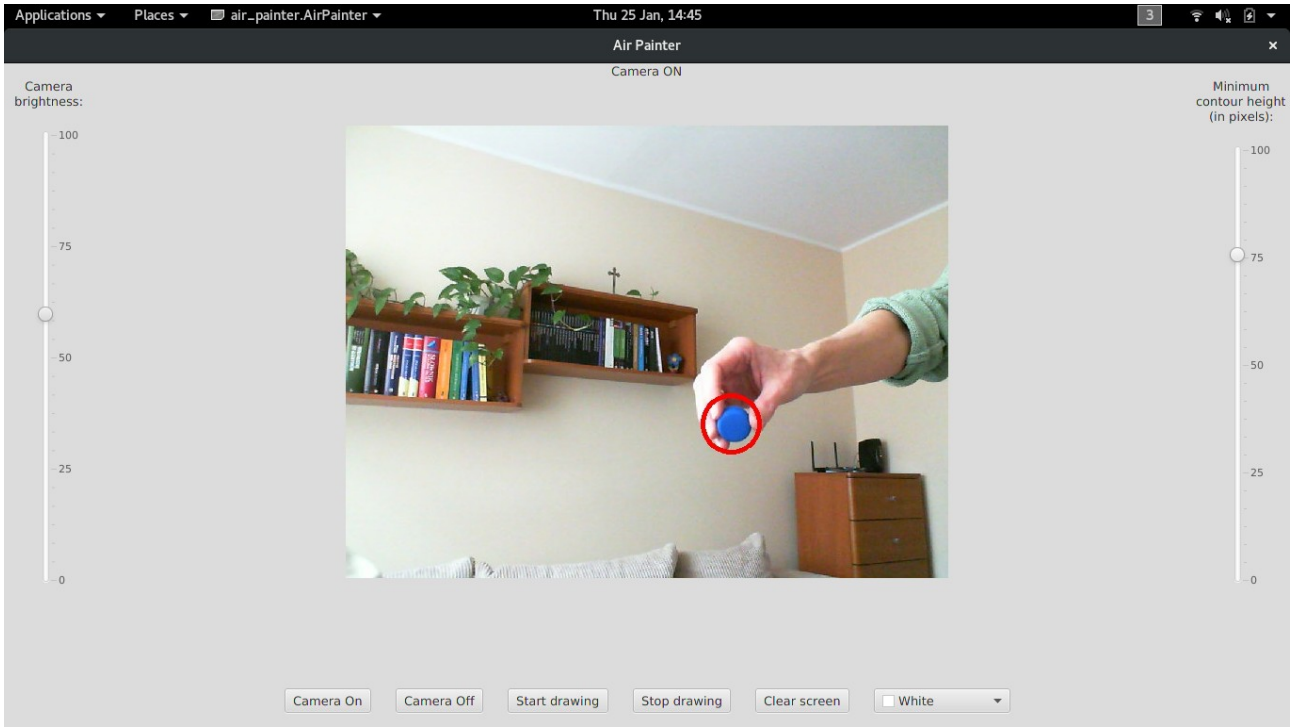
Rysunek 5 Wykrycie niebieskiej książki w tle. Źródło: opracowanie własne

Na powyższym rysunku wyraźnie widać, że aplikacja rozpoznała niebieską książkę umieszczoną w tle. W celu eliminacji tego typu niepotrzebnych elementów, użytkownik ma możliwość zmiany jasności obrazu kamery, przy użyciu suwaka umieszczonego po lewej stronie. Zastosowanie omawianego suwaka pokazano na rysunku nr 6.



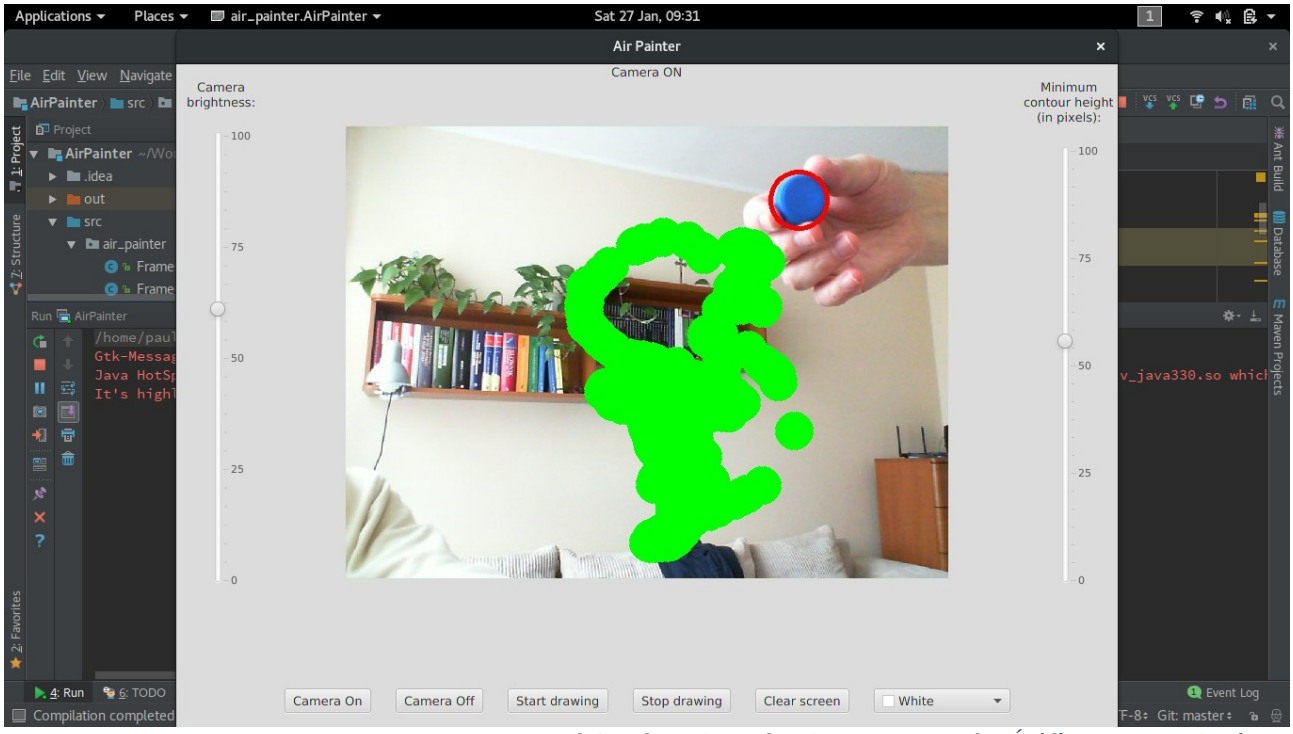
Rysunek 6 Zmiana jasności obrazu z kamery. Źródło: opracowanie własne

Aplikacja umożliwia także zmianę wielkości konturów wykrywanego obiektu wyrażonej w pikselach. Domyślnie aplikacja odrzuca kontury o wysokości mniejszej lub równej 25 pikseli. Suwak przy pomocy którego należy wskazać żądaną wielkość pikseli znajduje się po prawej stronie okna programu.



Rysunek 7 Zwiększenie minimalnego konturu obiektu. Źródło: opracowanie własne

Działanie suwaka zademonstrowano na rysunku nr 7. Zwiększenie minimalnej wysokości konturów śledzonego obiektu pozwoliło na wyeliminowanie wykrywanych w tle niebieskich książek.



Rysunek 8 Malowanie na ekranie za pomocą ruchu. Źródło: opracowanie własne

Na rysunku nr 8 pokazano efekt malowania na ekranie za pomocą ruchu niebieskiego obiektu. W obecnej wersji aplikacja umożliwia malowanie przy użyciu tylko jednego koloru. Mimo, iż na panelu osadzono przybornik służący do wyboru koloru, ta funkcjonalność nie jest jeszcze zaimplementowana. Aby uruchomić proces malowania na ekranie należy wcisnąć przycisk opatrzony etykietą „Start drawing”, z kolei w celu wyłączenia tegoż procesu należy nacisnąć przycisk „Stop drawing”. Aplikacja umożliwia także usuwanie rysunku z ekranu. Służy do tego przycisk „Clear screen”.

8. Interesujące problemy oraz ich rozwiązania

Niniejszy program używa do pobierania i przetwarzania obrazu z kamery dodatkowy wątek procesora. W celu zarządzania omawianym wątkiem użyto klasy implementującej interfejs java.util.concurrent.ScheduledExecutorService. Metoda inicjalizująca jego działanie wygląda następująco:

```
private void startDisplayThread(long initialDelay, long delay, TimeUnit unit) {
   threadExecutor = Executors.newSingleThreadScheduledExecutor();
   Runnable VideoCapture = () -> {
      try {
         Mat frame = frameGrabber.getNextFrame();
         Point newCoordinates = objectTracker.getCoordinates(frame);
         Image image = getImageWithDrawing(frame, newCoordinates);
         uiController.setImageToDisplay(image);
      } catch (IOException e) {
         System.err.println(e.getMessage());
      }
   };
   threadExecutor.scheduleWithFixedDelay(VideoCapture, initialDelay, delay, unit);
}
```

Jak widać w powyższym kodzie źródłowym wątek wykonuje cztery zasadnicze instrukcje: pobiera następną klatkę obrazu z kamery, przekazuje ją do obiektu klasy śledzącej artefakt, otrzymane współrzędne położenia artefaktu nanosi, w formie rysunku, na klatkę z kamery i na koniec przekazuje ją kontrolerowi okna aplikacji do wyświetlenia. Rysowanie na obrazie z kamery polega głównie na wywołaniu metody:

```
private Mat drawOnFrame(@NotNull Mat frame, @NotNull Point coordinates) {
   Mat result = framePainter.drawAllPoints(frame);
   if (requestedPictureDrawing) {
      framePainter.addNextPoint(coordinates);
   }
   return framePainter.drawCircle(result, coordinates);
}
```

W prezentowanej wyżej metodzie rysowanie polega na wywołaniu trzech metod na rzecz obiektu klasy malującej po obrazie. W rozpatrywanej klasie – FramePainter.java – koordynaty poszczególnych punktów obrazu przechowywane są w obiekcie klasy implementującej interfejs Map, czyli java.util.concurrent.ConcurrentHashMap. Jest to kolekcja, na której wykonywane operacje są synchronizowane pomiędzy poszczególnymi wątkami. Wcześniej, w celu przechowywania punktów rysunku, używano klasy implementującej interfejs Set, a dokładniej java.util.HashSet. Takie zastosowanie niesynchronizowanej kolekcji powodowało zawieszanie programu w momencie naciśnięcia przez użytkownika przycisku „Clear screen”. Równoczesny dostęp dwóch wątków do jednej kolekcji implikował wpadanie jednego z nich w nieskończoną pętlę, jak również brak możliwości dalszego zarządzania tymże wątkiem. Skutkowało to koniecznością siłowego przerywania jego działania z poziomu środowiska deweloperskiego.

Dodatkowo specyfika powtarzalności omawianego błędu, pozorny niedeterminizm jego występowania, nastręczała dodatkowych trudności w jego naprawie.

Po rozpoznaniu omawianego błędu naturalnym wyborem wydawało się zastosowanie implementacji interfejsu Set oferującej synchronizację podczas wielowątkowego dostępu do zawartości kolekcji. Jednak okazało się, iż standardowa biblioteka języka Java nie oferuje oddzielnej klasy spełniającej powyższe kryteria (https://stackoverflow.com/questions/6992608/why-there-is-no-concurrenthashset-against-concurrenthashmap). Z uwagi na omawiane niedogodności jako kolekcję przechowującą poszczególne punkty rysunku wykorzystano synchronizowaną wersję mapy haszującej, jako klucze wybierając typ Integer (opakowujący wartość typu int zwracaną przez metodę hashCode()), a jako wartość – obiekty typu Point.