14.06.19

1. Wojciech Niedbała
2. Patryk Romaniak
3. Albert Millert
4. Michał Mirończuk

Projekt zespołowy - dokumentacja

https://github.com/mirooon/fingero

Malowanie obrazów biorąc pod uwagę ruch ciała

1. Opis projektu

Nasz projekt *Malowanie obrazów biorąc pod uwagę ruch ciała* polegał na stworzeniu aplikacji internetowej, bazującej na wykrywaniu, a także śledzeniu ruchu ręki użytkownika i rysowaniu przy jej pomocy linii. Efektem odwzorowania ścieżki, po której przemieszcza się dłoń jest obraz na płótnie.

Projekt udostępnia możliwość rysowania zwykłym ołówkiem lub pędzlem. Ponadto, do dyspozycji użytkownika udostępnione są narzędzia, pozwalające dostosować styl rysowania, np. paleta kolorów oraz przyciski do zmiany szerokości końcówki, przy pomocy której powstaje rysunek.

Użytkownik na każdym etapie powstawania dzieła ma również możliwość dokonywania korekty. Do tego zadania można wykorzystać białą gumkę.

W sytuacji, gdy użytkownik chce przemyśleć dalszy rozwój pracy, jednocześnie nie chce, by system śledził jego ruchy, ma do dyspozycji przycisk, umożliwiający chwilowe wstrzymanie. Jeśli rysunek nie spełnia oczekiwań autora, z powodzeniem można skorzystać z możliwości jakie oferuje przycisk resetowania płótna; odświeża widok i rozpoczyna śledzenie dłoni na nowo. Po stworzeniu obrazu można go spokojnie zapisać na komputerze, wykorzystując odpowiedni przycisk *Save*.

Dokonaliśmy również rozeznanie, czy istnieją podobne aplikacje, korzystające wyłącznie z kamerek internetowych, a nie bardziej zaawansowanych urządzeń (na przykład *Kinect*) i znaleźliśmy dużo przykładów zaimplementowanych w *Pythonie*, jednak były one tworzone typowo jako aplikacje testowe, bez żadnego interfejsu, działające bezpośrednio w okienkach dostępnych z poziomu *OpenCV*. Po wypróbowaniu kilku z tych aplikacji, stwierdziliśmy, że działają one słabo i dość mocno obciążają komputer (prawdopodobnie twórcy przygotowując filmiki pokazujące możliwości ich programów wybierali jedynie najlepsze próby, przez co większość artykułów i poradników wygląda jakby

działała bardzo dobrze, jednak w rzeczywistości wyniki okazywały się bardzo zróżnicowane Już sam fakt, że każdy laptop posiada kamerkę internetową o zróżnicowanych parametrach, powoduje zmiany w wynikach.

2. Uzasadnienie wybrania tematu

Temat zainteresował nas, ponieważ nie mieliśmy wcześniej styczności z rozpoznawaniem i śledzeniem ruchu rąk. Poprzednie projekty z jakimi mieliśmy styczność w dziedzinie przetwarzania obrazów dotyczyły głównie wykrywania twarzy, bardziej statycznych obiektów oraz zdjęć. Śledzenie ręki wydało nam się zatem odpowiednim wyzwaniem, do którego możemy wymyślić rozwiązanie na kilka sposobów. Każdy z nas miał trochę inną wizję i pomysł na aplikację i wymiana tej wiedzy była interesująca i pozwoliła wybrać najlepszy wspólny pomysł. Z uwagi na nieregularny kształt ręki, wiedzieliśmy, że jej wykrycie i śledzenie może być problematyczne, a także gorzej opisane w internecie.

Z racji na fakt, że projekt jest na zajęcia z *Teleinformatyki*, chcieliśmy również wypróbować alternatywne sposoby na przesyłanie informacji od *REST*, bardziej odpowiadające aplikacjom z potrzebami aktualizacji danych i widoków na żywo, tak żeby każda przetwarzana informacja między stroną a serwerem była dostępna możliwie szybko oraz asynchronicznie. Zdawaliśmy sobie sprawę, że jest to dodatkowe utrudnienie i prościej by było podążać za gotowymi tutorialami i stworzyć kolejną aplikację w *Pythonie*, która by działała jedynie na okienkach dostarczanych przez *OpenCV*, jednak nas interesował projekt bardziej z punktu widzenia produktu, który miałby sens w użytkowaniu. W końcu nie każdy kto chce korzystać z takiej aplikacji będzie sobie instalował na komputerze *Pythona*, *OpenCV* i wszystkie inne zależności. Dlatego u nas w projekcie założyliśmy istnienie serwera, który by zawierał już wszystkie niezbędne programy, odciążając przy tym komputer klienta.

3. Podział prac

Wojciech Niedbała:

- Przesyłanie klatek z kamerki internetowej do serwera
- Widoki w aplikacji
- Obsługa WebSockets
- Przyciski pauzowania i czyszczenia

Patryk Romaniak:

- Możliwość podglądu ścieżki podczas rysowania
- Zapisywanie utworzonego obrazu

Michał Mirończuk:

- Możliwość zmazywania rysunku
- Możliwość wyboru trybu rysowania
- Zmiana stylu, koloru, grubości markera

Albert Millert:

- Przetwarzanie otrzymanego obrazu
- Rozpoznawanie ręki
- Śledzenie ruchu

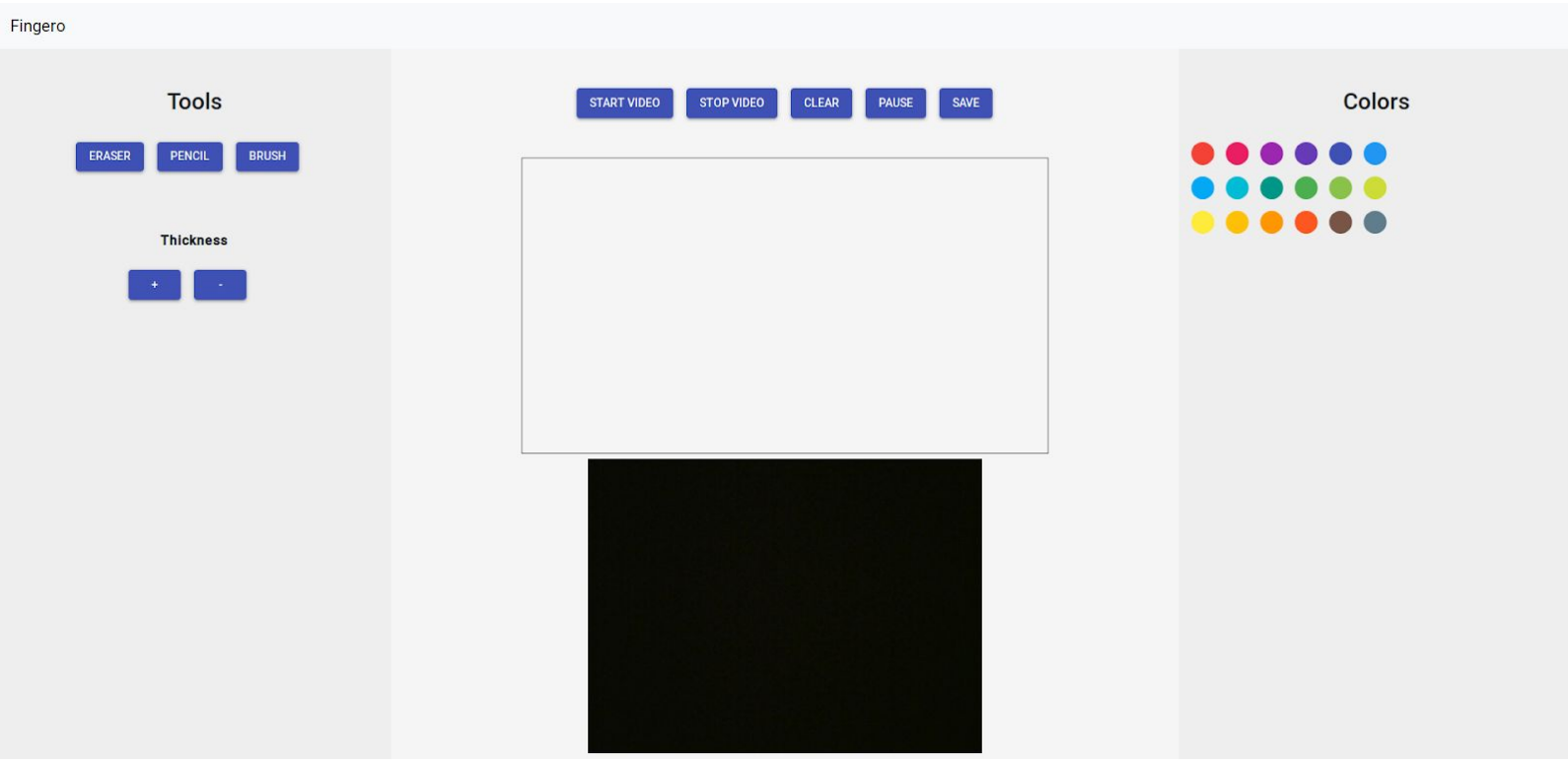
Pierwotnie (w pierwszej prezentacji) podział wyglądał nieco inaczej, gdyż część zadań musiała zostać zmieniona i dostosowana względem decyzji podejmowanych wraz z kolejnymi iteracjami implementacji projektu. Częścią tych zadań końcowo się podzieliliśmy, gdyż okazały się trudniejsze niż początkowo zakładaliśmy. Przykładowo, sposób wykrywania dłoni zmieniał się praktycznie za każdym razem i jedna osoba mogłaby nie poradzić sobie bez pomocy innych. Nie uważamy tego za minus, wręcz przeciwnie - jest to zaleta pracy zespołowej i niektóre problemy można rozwiązać znacznie szybciej w grupie niż pojedynczo. W trakcie pracy pojawiły się także pomniejsze zadania, których nie uwzględniliśmy przy podziale prac.

4. Wylistowanie i opisanie funkcjonalności oferowanej przez aplikację

Holistyczny opis aplikacji

Strona główna aplikacji jest utrzymana w minimalistycznej stylistyce i jest wzorowana na kolorystyce zgodnej z *Material Design* (https://material.io/). Wstępnie próbowaliśmy również stworzyć aplikację mając na uwadze urządzenia mobilne, niestety zbyt wiele elementów różniło się; począwszy od domyślnej, pionowej orientacji telefonu, po sposób działania kamerki internetowej, która odpowiednio działała tylko na niektórych modelach, a na innych nie (co prawdopodobnie było ściśle związane z pozwoleniami udostępnionymi dla przeglądarki na dostęp do przedniej kamery telefonu).

Kwestie designu nie stanowiły dla nas kwestii priorytetowej przy tworzeniu aplikacji, dlatego nie poświęciliśmy jej nadmiernej ilości czasu. Znacznie ważniejsze od samego designu było dla nas przykładowo to, żeby zmiana koloru odbywała się natychmiast po jego wybraniu. Z tej przyczyny oczywiste jest, że w kwestii wyglądu aplikacji można dostrzec sporo niedociągnięć, szczególnie, gdy strona zostanie otwarta w nietypowej rozdzielczości.



Rys 1. Widok aplikacji

Menu

Informatywne menu przedstawia nazwę aplikacji - *Fingero*.

Trzy panele, stanowiące główną część aplikacji

Pierwszy panel (patrząc od lewej strony) oferuje szereg narzędzi, umożliwiających użytkownikowi wybór trybów, którymi chce wprowadzać zmiany na płótnie (panel środkowy). Do wyboru są gumka (*eraser*), ołówek (*pencil*), pędzel (*brush*). Poniżej można zmieniać grubość pędzla (*thickness*).

Wspomniany panel środkowy udostępnia najważniejsze operacje w aplikacji dostępne za pomocą przycisków:

- *Start Video* - odpowiada za uruchomienie wykrywania i śledzenia ręki oraz rysowania. Dodatkowo łączy się przez *WebSocket* z serwerem i uzyskuje od niego liczbę pokazywanych palców, dzięki czemu można w trakcie rysowania za pomocą gestów zmieniać tryb rysowania,
- *Stop Video* - wstrzymuje całkowicie wykrywanie, śledzenie i rysowanie oraz zamyka połączenie z serwerem. Po wykonaniu tej operacji możliwe jest jedynie uruchomienia śledzenia od nowa (poprzez *Start Video*) lub pobranie narysowanego obrazu,
- *Clear* - czyści całkowicie *canvas,*
- *Pause* - umożliwia chwilowe zatrzymanie śledzenia palca i rysowania, na przykład w celu przeniesienia ręki w inne miejsce.
- *Save* pozwala na zapisanie obrazu do pliku **.png*.

Za funkcjonalność zapisywania odpowiada komponent *Save*. Obiekt klasy dziedziczącej po komponencie jest pozwala wyrenderować widok; w tym przypadku przycisk, któremu nadajemy funkcjonalność do zapisu aktualnie rysowanego obrazka.

```
class Save extends Component{
 saveCanvas() {
   const canvasSave = document.getElementById('myCanvas');
   const d =
canvasSave.toDataURL('image/png').replace("image/png",
"image/octet-stream");
   window.location.href = d;
 }

 render(){
   return(
      <Button variant="contained" color="primary"  onClick={
this.saveCanvas }>Save</Button>)}}
```

Tę sekcję można rozszerzać o nowe funkcjonalności, takie jak:

- *ClearLast* - czyszczenie ostatniej narysowanej kreski,
- *Manual* - przełączenie się na tryb rysowania manualnego - myszką lub gładzikiem, w celu dokonania bardziej szczegółowych poprawek,
- *Random* - umożliwiające wybór losowego koloru.

Poniżej opisanych przycisków znajduje się *canvas* (płótno), na które zostają nanoszone linie rysowane przez użytkownika. Na samym dole panelu widnieje pole, w którym po włączeniu trybu rysowania, zostaje umieszczony obraz z kamery na żywo. Docelowo oba obrazy miały być nałożone na siebie, jednak w zależności od warunków oświetleniowych samo płótno bywało bardzo słabo widoczne, co ostatecznie sprawiło, że porzuciliśmy ten pomysł.

Ostatni panel, wysunięty najbardziej na prawo, umożliwia wybór jednego spośród całej gamy przedstawionych kolorów. Wybrany kolor zmienia się od razu, w przypadku gumki nic nie zmienia.

5. Wybrane technologie wraz z uzasadnieniem dlaczego

Back-end

Logika serwera została zaimplementowana w języku *Golang*. Wybraliśmy ten język z powodu jego prostoty oraz podobieństwa do wcześniej nam znanych języków takich jak *Python* czy *JavaScript*. Ponadto *Go* oferuje bogatą obsługę współbieżności za pomocą wbudowanych mechanizmów takich jak *goroutines* czy *kanałów*.

Dzięki temu, że *Go* nie zawiera klasy *Threads*, która umożliwiałaby zarządzanie wątkami, *Go* wyróżnia się na tle tradycyjnych języków. Aby skorzystać z goroutines wystarczy dodać słówko kluczowe go przed wywołaniem funkcji. W dowolnym momencie wykonywania programu jeden wątek wykonuje jeden goroutine; gdy ulegnie zablokowaniu - zostanie zamieniony na inny goroutine, który następnie wykona się w tym wątku.

Podejście to jest podobne do działania planisty, ale po pierwsze jest obsługiwane przez środowisko wykonawcze *Go*, a po drugie - jest to znacznie szybsze. Dzięki takiemu mechanizmowi można stworzyć współbieżny serwer bez potrzeby opierania się na eventach.

Tabela poniżej przedstawia podstawowe różnice między podejściem wątkowym a wykorzystującym goroutines:

Wątek	Goroutine
Wątki są zarządzane przez system operacyjny.	Goroutines są zarządzane przez środowisko wykonawcze go i nie mają zależności sprzętowych.
Wielkość stosu wątku wynosi około 1MB	Wielkość stosu goroutine wynosi 2KB
Wielkość stosu jest stałą wartością której nie możemy zmienić.	Wielkość stosu Go jest zarządzana w czasie wykonywania i może wynieść nawet 1 GB, co jest możliwe dzięki zarządzaniu stertą
Komunikacja nie jest prosta wymagana jest odpowiednia synchronizacja.	Wbudowany system bezpiecznej komunikacji kanały, przy ich użyciu o wiele trudniej doprowadzić do zakleszczenia.
Utworzenie wątku kosztuje dość dużo tak samo jak proces jego zamknięcia, ponieważ wątek potrzebuje sporo zasobów które musi zarezerwować bądź oddać systemowi.	Goroutines są tworzone i niszczone przez środowisko wykonawcze Go. Operacje te są bardzo tanie w porównaniu z wątkami, ponieważ środowisko wykonawcze utrzymuje już pulę wątków dla goroutines. W tym przypadku system operacyjny nie wie o istnieniu goroutines.

W naszej aplikacji używamy *goroutines* do obsługi każdego klienta WebSockets w osobnym wątku, dzięki czemu z naszej aplikacji może korzystać wielu klientów na raz. Kod odpowiadający za tę logikę wygląda następująco:

```
func main() {
    log.SetOutput(os.Stdout)
    log.SetFlags(log.LstdFlags | log.Lshortfile)

    fmt.Println("Serving on localhost:8080")
    http.HandleFunc("/ws", serveWs)
    http.ListenAndServe(":8080", nil)
}

func serveWs(w http.ResponseWriter, r *http.Request) {
    fmt.Println("Connected with client")

    ws, err := upgrader.Upgrade(w, r, nil)
    if err != nil {
        log.Println(err)
    }
    reader(ws)
}
```

Warto zaznaczyć, że funkcja *http.HandleFunc* sama w sobie odpowiada za logikę obsługi klientów na różnych goroutynach, co oznacza, że wielowątkowy serwer w *Go* można uzyskać bez większego nakładu pracy - bez pisania zbędnego kodu; sam moduł *http* należy do bibliotek systemowych.

Z uwagi na fakt, że aplikacja w głównej mierze skoncentrowana jest na przetwarzaniu obrazu oraz rozpoznawaniu dłoni, zdecydowaliśmy się na zrealizowanie tej funkcjonalności z wykorzystaniem biblioteki *OpenCV* (*Open Source Computer Vision*). Narzędzie jest bezpłatne, open-sourcowe oraz posiada bogatą społeczność ulepszającą produkt. Za pomocą kamerki internetowej jesteśmy w stanie filtrować obraz, segmentować go oraz wykrywać poszczególne krawędzie, bez implementowania na nowo kodu

rozwiązującego powtarzające się zagadnienia, a także unikając uczenia maszynowego, które wymagałoby sporej ilości danych treningowych (w tym przypadku zdjęć rąk).

Do obsługi *OpenCV* w języku *Go* posłużyliśmy się biblioteką *GOCV*, umożliwiającą prostą komunikację z zainstalowanym systemowo *OpenCV* w wersji 4.

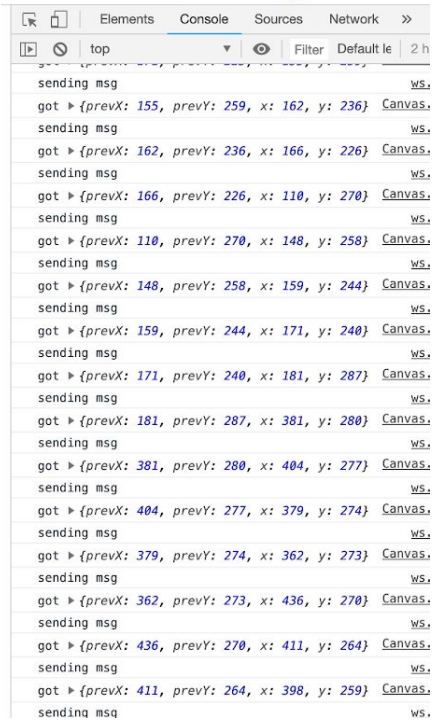
Front-end

Po stronie front-endu posłużyliśmy się bardzo wygodną i nowoczesną technologią *React* - biblioteką, wykorzystywaną do tworzenia klienckich aplikacji internetowych. Pozwala ona tworzyć aplikacje oparte o komponenty, które ustawione są względem siebie w strukturze drzewiastej, gdzie korzeń najczęściej stanowi komponent *App.js*. Za pośrednictwem komponentów mamy możliwość dodawania nowych funkcjonalności w sposób niezależny od siebie, co pozwala na równoległą pracę oraz na łatwiejszą organizację kodu.

Dodatkową zaletą *Reacta* jest jego prostota, bariera wejścia dla osoby z minimalną wiedzą z zakresu *JavaScript* jest niska i pozwala bez większego wysiłku zacząć tworzyć ciekawe projekty. Dzięki udogodnieniom oferowanym przez tę bibliotekę programista ma możliwość przekazywania parametrów do utworzonych komponentów. Poniższy kod stanowi przykład wykorzystania języka *JSX* w celu zastosowania dynamicznych zmiennych bezpośrednio w znaczniku *Canvas*, w języku *HTML*.

```
<Canvas
 width={this.state.width}
 height={this.state.height}
 point={this.state.point}
 mode={this.state.mode}
 thickness={this.state.thickness}
 color={this.state.color}
 reset={this.state.reset}
/>
```

Dzięki takiemu przekazywaniu danych (*props*) do komponentu *Canvas,* który w dużym uproszczeniu stanowi wrapper na element *canvas* w *HTMLu*. Każda ze zmiennych, ulegając zmianie powoduje. że komponent *Canvas* sam się aktualizuje oraz odpowiednio wyrenderuje na podstawie zaistniałych zmianach w danych. Nie ma potrzeby na manualne wywoływanie aktualizacji strony, co sprawia, że kod jest znacznie bardziej przejrzysty, a cała generyczna logika aktualizacji strony jest obsłużona przez *React*.



Rys 2. Przykładowe parametry wyświetlające się w konsoli

Jak widać na powyższym zrzucie ekranu, *React* rzeczywiście samodzielnie renderuje ponownie komponent *Canvas,* mimo faktu, że dane są zmieniane w innym komponencie - *App.js,* który jest głównym komponentem naszej aplikacji (a także domyślnie głównym komponentem każdej aplikacji w *React*).

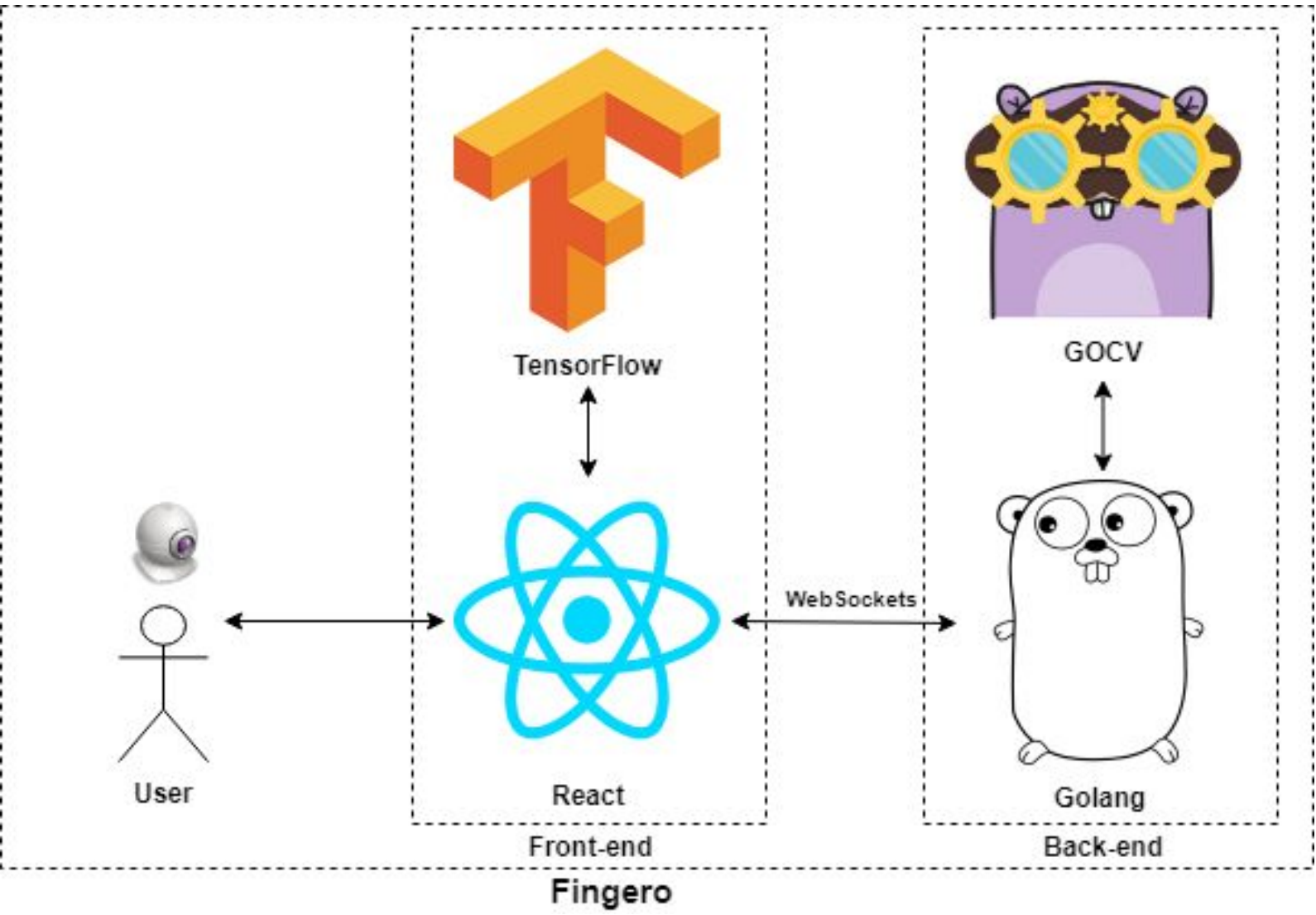
Użyliśmy również wbudowanej w *JavaScript* obsługi protokołu *WebSockets* do wysyłania i odbierania danych z serwerem. Inicjalizacja połączenia następuje, gdy użytkownik zaczyna rysować (poprzez naciśnięcie przycisku *Start Video*):

```
this.setState(
{ socket: new WebSocket("ws://localhost:8080/ws") },
this.addSocketMethods
);
```

Kod ten oprócz inicjowania połączenia z serwerem, również ustawia stan zmiennej *socket* na obiekt z połączenia protokołem *WebSockets*. Do zamknięcia takiego połączenia wystarczy wyciągnąć ten obiekt z całego stanu komponentu głównego, a następnie wywołać funkcję zamykającą połączenie:

```
  const { socket } = this.state;
  socket.close();
```

6. Architektura rozwiązania



Rys 3. Architektura systemu Fingero

Po stronie serwerowej wykorzystujemy *Golang* wraz z zainstalowaną biblioteką *OpenCV* z którego korzysta biblioteka *GOCV*. Jest on używany do rozpoznawania gestów (ilości pokazanych palców), a następnie odsyłania *JSONa* z polem count zawierających ilość. Moduł serwerowy składa się z następujących funkcji:

- *reader* - przyjmuje jako argument wskaźnik na *Websocket*, weryfikuje dane przesłane przez klienta,
- *serveWs* - przyjmuje dwa parametry *http.ResponseWriter* oraz *http.Request*, pozwala na obsługę *Websocketu* przez serwer,
- *toRGB8* - na wejściu przyjmuje obraz który konwertuje do *RGB8,*
- *countFingers* - przyjmuje obraz, przekształca go na obraz w skali szarości, następnie proguje w celu znalezienia największego kontur, z którego wyciągana jest liczba palców,
- *getBiggestContour* - przyjmuje tablicę punktów jako argument, jest to funkcja pomocnicza która pozwala na znalezienie największego konturu.

W celu komunikacji z front-endem wykorzystujemy *WebSocket*. Obraz z kamery internetowej użytkownika jest przesyłany formacie *png*, zakodowany w *Base64*. *WebSockets* umożliwiają serwerowi i klientowi wysyłanie wiadomości do siebie w dowolnym momencie, po ustanowieniu połączenia, bez wyraźnego żądania jednego lub drugiego. Kontrastuje to z *HTTP*, który tradycyjnie wiąże się z zasadą wezwanie-odpowiedź - gdzie, aby uzyskać dane, należy je wyraźnie zażądać. Pozwala to na przetwarzanie obrazu w czasie rzeczywistym.

Tensorflow.js używamy do wykrywania i śledzenia ręki bezpośrednio na aplikacji klienckiej w celu odciążenia serwera, i zmniejszenia opóźnienia związanego z przesyłaniem danych w obie strony, które zajmowało więcej czasu niż rzeczywiste wykrywanie. Działa on bardzo dobrze z typem *canvas,* dzięki czemu wdrożenie tej biblioteki nie wymagało sporej ilości zmian, wystarczyło jedynie wywołanie funkcji getCanvas na obrazie z kamerki internetowej. Korzystamy dokładnie z *handtrack.js* oraz z *COCO-SSD models*.

Prezentacja aplikacji jest realizowana za pomocą aplikacji internetowej wykorzystującą *React*. Biblioteka ta pomogła nam w bardzo prosty sposób użyć websocketów które są podstawą komunikacji w *Fingero*. Po stronie klienta wykrywana jest dłoń, jak i również renderowany jest jej ruch. Na kliencką stronę składa się kilka modułów, plików css oraz html.

Komponent *App* stanowi główny moduł naszej aplikacji, obsługuje m.in zmiany przyboru do rysowania oraz grubości kreski. Wszelkie zmiany są zapisywane w stanie obiektu. Sam moduł składa się z następujących składowych:

- *addSocketMethods* - metoda odpowiedzialna za otwarcie socketów i połączenie z serwerem,
- *startVideo* - metoda wywoływana poprzez przycisk *Start Video*; rozpoczyna pracę kamerki oraz proces wykrywania dłoni, ustala także punkty położenia dłoni w stanie obiektu,
- *stopVideo* - metoda, która kończy połączenie z serwerem oraz pracę kamerki,
- *Settery* dla przyborów - każdy z nich zmienia aktualnie używane narzędzie do rysowania; wywoływane są po kliknięciu w odpowiedni przycisk lub pokazanie odpowiedniego gestu; wszelkie zmiany są odwzorowywane w stanie obiektu; do wyboru - gumka, pisak, długopis.

Komponent *Canvas* - obsługuje płótno do rysowania, w swoim stanie ma zapisaną poprzednią wartość x, y dzięki czemu możemy określić czy nowe wartości są poprawne i powinny zostać uwzględnione na płótnie. Komponent składa się z metod:

- *updateCanvas* - metoda która służy do rysowania, waliduje nowy ruch i jeżeli spełnia on założenia to rysuję odpowiednią ścieżkę na płótnie, metoda ta przyjmuje jako argumenty właściwości przyrządu do rysowania,
- *clearCanvas* - metoda służąca do wyczyszczenia płótna, wywoływana przez naciśnięcie przycisku.

7. Interesujące problemy i rozwiązania ich na jakie się natknęliście

Przesyłanie kolejnych klatek pomiędzy serwerem a stroną

W celu wykonywania jakichkolwiek operacji na obrazie po stronie serwera, potrzebowaliśmy przesyłać obraz z kamerki internetowej na serwer. W tym celu po uruchomieniu rysowania, nawiązywaliśmy połączenie z serwerem poprzez *WebSocket* i w regularnych odstępach czasowych wysyłaliśmy zakodowany obraz **.png* w *Base64*. Kod po stronie aplikacji wygląda następująco:

```
let socketID = setInterval(() => {
  const { socket } = this.state;
  if (socket.readyState === WebSocket.OPEN) {
    socket.send(this.webcam.getScreenshot());
  }
}, 100);
```

Funkcja *getScreenshot,* sama koduje obraz do *Base64*, jednak format pliku (**.png*) był podawany przy inicjalizacji kamery internetowej. Po stronie serwera, kodu jest znacznie więcej, chociażby dlatego, że trzeba się pozbyć prefixu dodawanego przez wspomnianą wcześniej bibliotekę. Utrzymanie połączenia z *WebSocketem* po stronie serwera w nieskończonej pętli (aż do rozłączenia), wygląda następująco:

```
for {
    messageType, p, err := conn.ReadMessage()
    if err != nil {
        log.Println(err)
        return
    }
    p = bytes.Replace(p, "", []byte{},
1)
    unbased := make([]byte, len(p))
    _, err = base64.StdEncoding.Decode(unbased, p)
    if err != nil {
        log.Println("Cannot decode b64")
        return
    }
    img, err := png.Decode(bytes.NewReader(unbased))
    if err != nil {
        log.Println(err)
        continue
    }
    imgMat, err := toRGB8(img)
    if err != nil {
        log.Println(err)
        continue
    }
    fingersCount := countFingers(imgMat) // ignore error here
because marshalling this map can not fail
    body, _ := json.Marshal(map[string]int{
        "count": fingersCount,
    })
    err = conn.WriteMessage(messageType, body)
    if err != nil {
        log.Println(err)
        continue}}
```

Wykrywanie dłoni na podstawie zakresu HSV obrazu

Wczytywany obraz zostaje przekonwertowany z modelu przestrzeni barw *BGR* na *HSV*, co pozwala na rozdzielenie go na składowe:

- odcień światła,
- nasycenie koloru,
- jasność (wartość światła białego).

Rys 4. Obraz przekonwertowany z BGR na HSV

Takie podejście pozwala w prosty sposób zastosować operację progowania obrazu (funkcja *gocv.InRangeWithScalar*), używając wcześniej zmierzonych i ustalonego zakresu wartości zgodnych z kolorami skóry. Po zastosowaniu operacji, wartości w obrazie, które nie mieszczą się w podanym zakresie, zostają usunięte, a na otrzymanym obrazie możemy wykonać funkcje *gocv.Blur* i *gocv.Threshold,* w celu usunięcia jak największej ilości szumów (na obrazku widoczne głównie w lewym górnym rogu oraz na oknach na środku).

Efekt operacji można obserwować na następującym obrazie:

Rys 5. Obraz uzyskany po szeregu operacji progowania oraz odszumiania

Następnie, na obrazie wykonujemy operacją wykrywania konturów - *gocv.FindContours,* która pozwala oddzielić wszystkie znalezione obiekty (o ile nie są ze sobą ściśle złączone) i zwróci listę list punktów - a konkretnie struktur z polami *int* - *X* oraz *Y* (współrzędnymi).

Kolejnym krokiem jest wybranie jednej listy, odpowiadającej z możliwie największym prawdopodobieństwem palcowi, którym posługuje się użytkownik do rysowania, a nie na przykład głowie, czy innym częściom ciała ludzkiego. Zdecydowaliśmy się na jedno z najbardziej intuicyjnych i prostych rozwiązań - wybór listy, odpowiadającej kształtowi o największym polu wierząc, że będzie ona przechowywać informacje o oczekiwanym kształcie.

Następnym założeniem, które przyjęliśmy w tej metodzie jest uznanie najwyższego punktu w wybranym kształcie za czubek palca, czyli ręka zawsze musi być wyprostowana. Miało to być rozwiązanie przejściowe w tej metodzie, używane tylko na początku, jednak nigdy nie zostało ulepszone ze względu na to, że całkowicie porzuciliśmy ten pomysł. O decyzji zadecydowało wiele powodów:

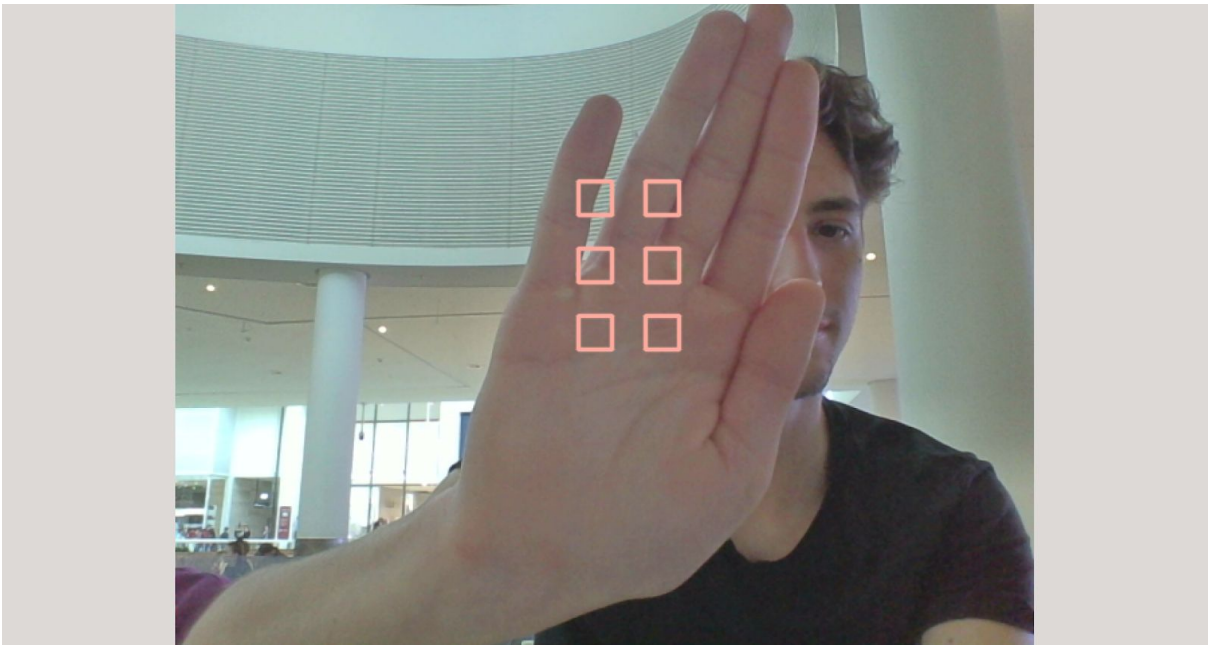
- Wartości *HSV* mające odpowiadać kolorowi skóry były wpisane w programie, a nie modyfikowane lub kalibrowane, przez co aplikacja działała tylko w dziennych warunkach oświetleniowych.
- Jako, że zawsze wybieraliśmy listę punktów o największym polu, musieliśmy za każdym razem to pole obliczyć dla każdego obiektu z podanego przedziału *HSV* (np. głowa, ramię, jasna ściana). Gdy takich obiektów pojawiało się wiele, serwer zauważalnie spowalniał i występowały opóźnienia w renderowaniu rysowanego obrazu na stronie.
- W przypadku gorszych warunków oświetleniowych występowało dużo szumów i części z nich nie dało się pozbyć, gdyż nachodziły na przykład na rękę, przez co algorytm minimalnie zmieniał co klatkę najwyższy punkt ręki co powodowało skoki oraz ostre linie na rysowanym obrazie.
- Większość czasu w tym podejściu serwer tracił na kodowanie i dekodowanie obrazu **.png* do *base64*.
- Przy próbach rozwoju algorytmu natrafiliśmy na braki w funkcjach w *GOCV*, które po prostu nie były jeszcze zaimplementowane w momencie pisania aplikacji. Niewykluczone, że jest to sytuacja przejściowa i biblioteka ta stanie się w końcu kompletna.
- Algorytm czasem wybierał głowę (podobny kolor skóry), mimo że ta była oddalona i wyraźnie mniejsza od ręki (prawdopobnie pole z głowy dało się dokładniej policzyć - lista punktów była większa z uwagi na bardziej regularny kształt głowy, przez co i pole mogło być większe).

- Zarówno operacja wykrywania ręki, jak i wykrywania gestów odbywały się na serwerze, przez co był on mocno obciążony, przez co już kilku klientów generowało spore obciążenie i były zauważalne obciążenia - wymiana danych przez WebSockets nie odbywała się w regularnych odstępach czasowych. Był to główny powód późniejszych pomysłów na przeniesienie logiki wykrywania i śledzenia ręki do klienta, żeby serwer mógł się skupić jedynie na wykrywaniu gestów.

Wykrywanie dłoni z wykorzystaniem histogramów koloru skóry

Metoda, która w teorii miała umożliwić rozpoznawanie dłoni niezależnie od warunków otoczenia - zmieniającego się oświetlenia, koloru skóry osoby testującej oprogramowanie. Histogramy kolorów dostarczają znacznie więcej zróżnicowanych informacji w przeciwieństwie do pojedynczego zakresu wartości jaki oferowała poprzednia metoda.

Na ekranie w początkowej fazie ukazywane są prostokąty, wskazujące użytkownikowi miejsca, z których pobrane zostaną wartości, na podstawie których następnie liczone są histogramy kolorów.



Rys 6. Przykładowa kalibracja

Na tym etapie pojawił się główny problem z tym podejściem - brakująca implementacja funkcji *cv2.calcBackProject*, mającej umożliwić wyodrębnienie fragmentu obrazu od reszty, wybierając jedynie wartości z zakresów wyznaczonych przez wyliczone histogramy.

Kolejnym krokiem było wyznaczenie wszystkich konturów z wyodrębnionego obrazu oraz wybranie jednego - o największej powierzchni. Dla danego konturu obliczane zostają momenty, dzięki którym dzieląc $moment_{10}$ przez $moment_{oo}$ otrzymujemy współrzędną x centroidu; w sposób analogiczny można wyliczyć punkt współrzędną y - $moment_{01}$ / $moment_{00}$. Przez centroid rozumiemy punkt środkowy kształtu. Dodatkowo, na podstawie największego konturu zostaje wyznaczona otoczka wypukła, która “uwypukla wklęsłe” części konturu, niejako rozszerzając go. Na podstawie otrzymanego kształtu wystarczy wybrać najbardziej odległy od centrum punkt i przyjąć go za czubek palca. Zakładamy, że palec, wykorzystywany do rysowania jest najbardziej wyróżniającym się.

Etap końcowy stanowi stworzenie listy, przechowującej ostatnie pozycje czubka palca, która zostaje wykorzystana do rysowania.

Wykrywanie dłoni w oparciu o przetrenowany model sieci neuronowej

Ostatni z zaimplementowanych i przetestowanych pomysłów na wykrywanie i śledzenie ręki. Daje najbardziej satysfakcjonujące rezultaty, dlatego to on został wybrany.

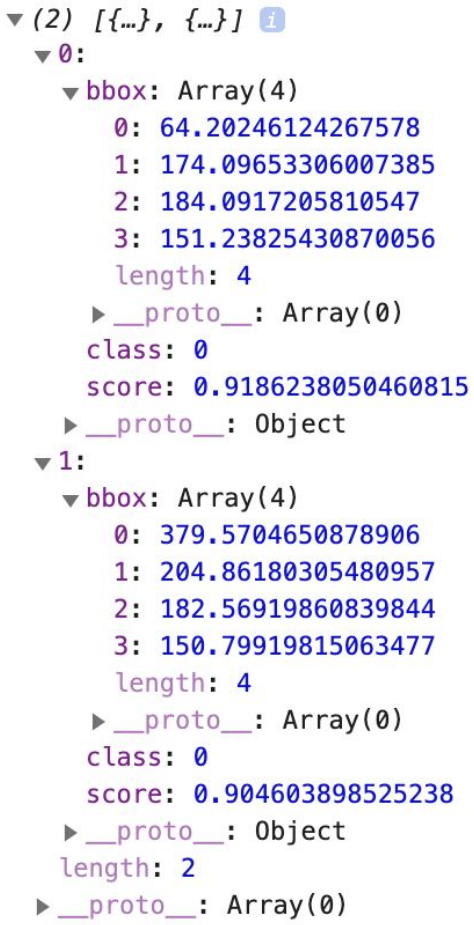
Za wykrywanie ręki odpowiada biblioteka *TensorFlow.js*, która została napisana i jest nadal rozwijana poprzez *Google* i jest zaadaptowana do wykorzystania w *JavaScript*, co jest szczególnie przydatne w użyciu w aplikacji klienckiej, dzięki czemu można znacznie odciążyć serwer. Wygodne jest też wsparcie dla rozszerzeń adaptujących gotowe modele i tworzących predykcje na ich podstawie.

Rozpoczęcie śledzenia dłoni rozpoczyna się tuż po kliknięciu przycisku *Start Video*. Wtedy wykonuje się następujący kod:

```
let id = setInterval(() => {
  if (this.state.pause) return;
  this.state.model.detect(this.webcam.getCanvas()).then(pred
=> {
    if (Array.isArray(pred) && pred.length) {
      let pos = pred[0].bbox;
      this.setState({
        point: {
          prevX: this.state.point.x,
          prevY: this.state.point.y,
          x: pos[0],
          y: pos[1],
        }
      });
    }
  });
}, 1000 / 10);
```

Odpowiednio wybrany model rozpoznaje wszystkie obiekty przypominające

ręce i zwraca listę tych obiektów wraz z prawdopodobieństwem w formacie zgodnym ze standardami *Tensorflow*:



Rys 7. Przykładowe wyniki predykcji modelu

Jak widać na powyższym zrzucie ekranu z konsoli przeglądarki, *Tensorflow.js* zwraca listę z potencjalnymi kandydatami. Próg ustawiony przez nas na pole *score* to 0.9, jest stosunkowo wysoki, jednak nie na tyle, żeby wyniki z obrazów ze słabszych kamerek internetowych były ignorowane. Pozostaje teraz w jakiś sposób wybrać jeden z tych wyników, przy prawdopodobieństwie powyżej 90% szansa na pomylenie jednego z obiektów z ręką jest bardzo mała, dlatego nie sortowaliśmy już tych wyników. Ponieważ nie chcieliśmy pogarszać wydajności aplikacji, po prostu wzięliśmy pierwszy element z listy, co jest widoczne w powyższym kodzie.

Dzięki temu podejściu żadna inna część ciała o podobnym odcieniu skóry nie

będzie przeszkadzać w rysowaniu, a ręka może być ustawiona pod dowolnym kątem, z dowolną ilością wyprostowanych palców, wykorzystana może zostać nawet pięść. Znaczna część logiki wykrywania i śledzenia została przeniesiona do klienta, dzięki czemu serwer jest odciążony i może służyć już wyłącznie do analizy i wykrywania gestów. Mimo faktu, że klient na tym etapie został znacznie bardziej obciążony, aplikacja z punktu widzenia użytkownika działa w dalszym ciągu stosunkowo płynnie, a rysowanie rzeczywiście nadążało za ruchem ręki. Żeby to osiągnąć, zastosowaliśmy również kilka prostych *sztuczek*, ulepszających wrażenie z rysowania. Oto podstawowe z nich:

```
const ctx = this.refs.canvas.getContext('2d');
ctx.lineCap = 'round';
if (Math.abs(point.prevX - point.x) > 75 ||
    Math.abs(point.prevY - point.y) > 75) {
    point.x = point.prevX - 1;
    point.y = point.prevY - 1;
}
```

Jak widać, odległość jest liczona w bardzo prosty sposób, znacznie bardziej skomplikowane obliczenia (np. pierwiastek sumy kwadratów różnic poprzednich i następnych współrzędnych w celu ustalenia ich odległości) powodowały wzrost zapotrzebowania na zasoby po stronie klienta. Z tego też powodu sprawdzanie zostało uproszczone, ale działa zadowalająco - gdy ruszymy nagle ręką lub zrobimy pauzę, przesuniemy daleko rękę i wznowimy śledzenie, kreska nie zostanie narysowana, a rysowanie zacznie się od nowego punktu. Można by tutaj również polegać na czasie oddzielającym poszczególne ruchy (gdy trwa za długo kreska nie zostałaby narysowana, a rysowanie zaczynałoby się od nowego punktu), jednak nie jest to łatwe do zaimplementowania, gdyż język *JavaScript* jest jednowątkowy, co powoduje, że liczenie czasu jest problematyczne (choć wykonalne).

Zdecydowanym minusem podejścia jest waga predefiniowanych modeli,

wynosząca ~15MB. Jednak nie przejęliśmy się tym faktem, gdyż cały czas są prowadzone prace, mające na celu znaczne zredukowanie ich rozmiaru.

Wykrywanie gestów

Sam algorytm wykrywania palców pochodzi w dużej mierze z przykładowych aplikacji napisanych w *GOCV*. Polega na stosowaniu progowania oraz rozmycia Gaussa w celu przygotowania obrazu do znalezienia wszystkich konturów, a następnie obliczeniu liczby defektów w konturze poprzez wywołanie metod:

```
gocv.ConvexHull(c, &hull, true, false)
gocv.ConvexityDefects(c, hull, &defect
```

Końcowym etapem jest obliczenie odległości między kolejnymi wykrytymi punktami i zastosowaniu nazwanej przez twórców tego przykładu *zasady cosinusa.* Kod który stosuje tę zasadę, wygląda następująco:

```
// apply cosine rule here
angle = math.Acos((
math.Pow(b, 2) +
math.Pow(c, 2) -
math.Pow(a, 2))/ (2*b*c)) * 57

// ignore angles > 90 and highlight rest with dots
if angle <= 90 {
    defectCount++
    gocv.Circle(&img, far, 10, green, 2)
}
```

Z tą różnicą, że nas nie interesuje rysowanie kółek na wykrytych palcach, a jedynie zwracana wartość *defectCount*, którą możemy następnie przekazać z powrotem do front-endu tym samym *WebSocketem*, którym otrzymaliśmy zakodowany obraz. Algorytm ten przez większość czasu działa dobrze - wystarczająco na nasze potrzeby, czasem się myli o jeden palec, ale uwzględniliśmy to przy wyborze trybów.
Liczba palców oraz przypisany tryb za jaki odpowiada:

- liczba palców mniejsza od dwóch przypisana została do wyboru pędzla; z uwagi na fakt, że zerowa liczba palców jest uwzględniona w tym przypadku, również defaultowy tryb to pędzel do malowania;
- dwa lub trzy palce pozwalają zwiększyć grubość pędzla;
- dowolna liczba palcy, większa od trzech włącza tryb gumki, umożliwiając zmazywanie obrazu przygotowanego przez użytkownika, w celu zresetowania *canvas*.

Renderowanie danych w czasie rzeczywistym

Poniższy kod przedstawia działanie rysowania w czasie rzeczywistym oraz aktualnie wykorzystywane narzędzie. W celu nałożenia nowego skrawka tworzonego rysunku, pod uwagę jest brana obecna oraz ostatnia pozycja palca. Na ich podstawie rysowana jest następnie linia. Ponieważ opóźnienia są bardzo małe, użytkownik nie widzi najmniejszego opóźnienia związanego z nakładaniem dodatkowej warstwy na obraz. Dodatkowymi parametrami funkcji jest kolor oraz grubość rysowanej linii.

```
updateCanvas = (point, mode, thickness, color) => {
        const ctx = this.refs.canvas.getContext('2d');
        ctx.lineCap = 'round';
        if (Math.abs(point.prevX - point.x) > 75 ||
Math.abs(point.prevY - point.y) > 75) {
            point.x = point.prevX - 1;
            point.y = point.prevY - 1;
        }
        if (mode === "pen") {
            ctx.globalCompositeOperation = "source-over";
            ctx.beginPath();
            ctx.moveTo(point.prevX, point.prevY);
            ctx.lineTo(point.x, point.y);
            ctx.strokeStyle = color;
            ctx.stroke();
        } else if (mode === "brush") {
            ctx.globalCompositeOperation = "source-over";
            ctx.lineWidth = thickness;
            ctx.beginPath();
```

```
            ctx.moveTo(point.prevX, point.prevY);
            ctx.lineTo(point.x, point.y);
            ctx.strokeStyle = color;
            ctx.stroke();
        } else if (mode === "eraser") {
            ctx.globalCompositeOperation = "destination-out";
            ctx.arc(point.x, point.y, thickness, 0, Math.PI *
2, false);
            ctx.fill();}}
```

Nie każda linia może być narysowana zgodnie z zamierzeniem autora dlatego do naszej aplikacji wprowadziliśmy gumkę, którą użytkownik może użyć w celu poprawienia niechcianego elementu.

8. Instrukcja użytkowania aplikacji

Do uruchomienia aplikacji wymagane są:

Golang 1.12

- GoCV 0.19.0
- Gorilla Websocket 1.4.0

OpenCV 4.1.0

npm 6.9.0

W celu uruchomienia aplikacji należy:

Uruchomić serwer

>> go run api/main.go

Uruchomić aplikację

>> cd ui

>> npm install

>> npm start

Otworzyć aplikację, domyślnie pod adresem http://localhost:3000/

W celu zbudowania aplikacji należy:

Zbudować serwer

>> go build api/main.go

Zbudować stronę

>> cd ui

>> npm install

>> npm build

Sprawdzony sposób wdrożenia aplikacji w chmurze Google:

- Serwer umieszczony w usłudze *App Engine*
- Strona umieszczona w usłudze *Firebase*