Agnieszka Rusin
Bartosz Mikulski
Michał Włodarczyk
Szymon Janowski

12 czerwca 2019

cARds

Dokumentacja projektu

Spis treści

1. Wstęp

Projekt cARds jest aplikacją implementującą mechanizmy rozszerzonej rzeczywistości (*ang. Augmented Reality*). Odpowiednio oznaczone karty układane są w zasięgu kamery. Aplikacja wykrywa znaczniki umieszczone na kartach oraz nakłada na nie animowane modele 3D. Efekt ten widoczny jest na ekranie komputera, który rejestruje obraz z kamery. Cały proces odbywa się w czasie rzeczywistym. Karty można dowolnie podnosić czy przesuwać, a model nadal wyświetlany będzie na powierzchni karty.

Jako tematykę modeli umieszczanych na kartach wybraliśmy popularną grę Pokémon. Seria Pokémon posiada różne wydania własnych gier karcianych (Pokémon TCG). Poprzez proste nałożenie odpowiednich znaczników na karty możemy urozmaicić chociażby turnieje rozgrywane pomiędzy pasjonatami przed szerszą publicznością. Dzięki zastosowaniu architektury klient-serwer poprzez odpowiednie przygotowanie aplikacji możemy udostępniać usługę, gdzie każdy może cieszyć się rozszerzoną rzeczywistością w grze Pokémon na soim urządzeniu.

Uzasadnienie podjęcia tematu

Rozszerzona rzeczywistość jest nadal stosunkowo nowym i ciekawym zagadnieniem. Nie tak dawno temu cały świat opanowała gra Pokémon GO. Gra ta bazowała na właśnie takim systemie i została bardzo ciepło przyjęta. Dodatkowo odczuliśmy, że może być zapotrzebowanie na aplikację podobne do naszej. Ciągły rozwój rynku gier karcianych zwłaszcza tak popularnej marki jak Pokémon, która po sukcesie Pokémon GO przeżywa swoją drugą młodość oraz coraz popularniejszy streaming różnych wydarzeń dla szerokiej publiczności przekonały nas, że takie rozwiązanie ma rację bytu.

2. Cel i zakres prac

Przeznaczenie i zadania systemu

Główne założenie projektu to stworzenie aplikacji internetowej współpracującej z kamerą wideo urządzenia w celu wygenerowania obrazu wzbogaconego o animowane modele 3D. Kamera skierowana jest na wzbogacone o markery Aruco karty do gry Pokémon.

Rys. 1. Przykładowa karta Pikachu.

System rozpoznaje kartę po markerze (marker Aruco reprezentuje zawsze pewną liczbę) oraz wyświetla na niej odpowiedni model 3D. Model pozycjonowany jest dzięki macierzy translacji oraz rotacji uzyskanych po interpretacji położenia markera Aruco.

Aplikacja posiada także serwer plików, który dostarcza modele 3D dla agenta po stronie przeglądarki internetowej.

Modele zostały przygotowane w aplikacji Blender. Niezbędne prace przy modelach to odpowiednie skalowanie modeli oraz przystosowanie modeli pod

silnik BabylonJs. Niezbędnym krokiem było stworzenie szkieletu oraz mapy wag dla wierzchołków, które posłużyły do wykonania animacji. Modele zostały eksportowane do formatu .babylonjs, w celu konsumpcji przez aplikację.

Udział poszczególnych członków zespołu w realizacji zadania

Agnieszka Rusin	● Utworzenie kart do gry oraz wygenerowanie markerów Aruco. ● Implementacja moduł cards_recognition ○ odczytywanie obrazu z kamery ○ rozpoznawanie kart detektorem oraz wyznaczanie współrzędnych karty ○ rozpoznawanie kart po markerze Aruco ○ wyznaczanie macierzy translacji oraz rotacji
Szymon Janowski	● Przesyłanie parametrów modelu jak jego identyfikator, macierz rotacji i macierz translacji przez websockety do frontendu z modułu cards_recognition. ● Serwowanie plików statycznych z odpowiednimi uprawnieniami na serwerze. ● Strumieniowanie obrazu z modułu cards_recognition. ● Wyświetlanie tła dla sceny w aplikacji.
Bartosz Mikulski	● Dostosowanie modeli 3D do aplikację. ● Animowanie obiektu za pomocą szkieletu na scenie. ● Serwowanie plików statycznych z odpowiednimi uprawnieniami na serwerze. ● Eksport modeli do formatu .babylon. ● Hosting strony w serwisie GitHub Pages

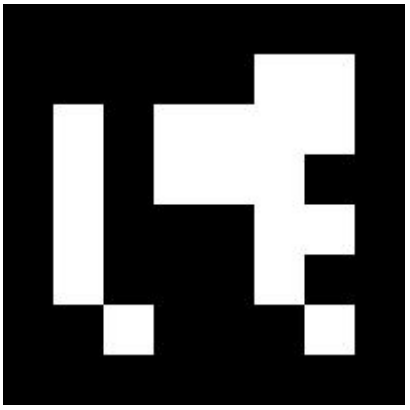
Michał Włodarczyk	● Tworzenie sceny 3D. ● Pobieranie współrzędnych i aktualizowanie pozycji modelu na scenie na ich podstawie. ● Translacja otrzymywanych współrzędnych po stronie klienta. ● Wyświetlanie obrazu z kamery jako tło.

3. Funkcjonalność aplikacji

a. Używanie wybranej przez użytkownika kamerki dostępnej na urządzeniu, na którym uruchamiana jest aplikacja.
b. Wyświetlanie obrazu z kamery w aplikacji internetowej.
c. Podstrony aplikacji internetowej zawierające markery Aruco poszczególnych modeli.
d. Po wprowadzeniu markera Aruco, który może znajdować się na dowolnej powierzchni (jak na specjalnie przez nas przygotowanych kartach) lub na ekranie telefonu, po przejściu do odpowiedniej podstrony aplikacji internetowej, pojawi się na nim model 3D jednego z 3 dostępnych Pokemonów wraz z animacją.
e. Zależnie od położenia markera jeżeli tylko jest on widoczny cały na kamerce obiekt umiejscowiony będzie zgodnie ze wszystkimi zmianami tego położenia.
f. Obiekty nakładane są na markery i wyświetlane w aplikacji użytkownika w czasie rzeczywistym bez zbędnych opóźnień.
g. Każdy z obiektów posiada swoje unikatowe animacje jak otwieranie otworu gębowego czy poruszanie odwłokiem modelu Bulbasaura czy poruszanie ogonem oraz głową modelu Pikachu.
h. Aplikacja ze względu na swoją budowę (jest aplikacją internetową/webową) uruchamiana może być na każdym urządzeniu z przeglądarką internetową, dostępem do internetu oraz kamerką (z pewnymi ograniczeniami).

4. Wybrane technologie

Aruco to małe markery 2D, które używane są w rozszerzonej rzeczywistości oraz w robotyce. Każdy marker przechowuje w sobie informacje, dzięki którym algorytm dekodowania jest w stanie zlokalizować, zdekodować i oszacować pozycję (położenie i orientację w przestrzeni) każdego z nich jeśli znajduje się on w polu widzenia kamery.



Rys. 2. Przykładowy marker Aruco

Informacje jakie w sobie przechowuje marker Aruco:

- marker ID
- x,y,z lub wektory - koordynaty 3D w milimetrach środka markera lub narożniki
- w,h,d - rozmiar markera(szerokość, wysokość i głębokość, która zawsze jest 1mm)

OpenCV - to najpopularniejsza darmowa biblioteka funkcji, która służy do zaawansowanego przetwarzania obrazów, czy też video. Wspiera platformy: Windows, Android, Linux, iOS, Mac OS. Przeznaczona jest ona do programowania w następujących językach: C++, Java i Python. W bibliotece tej są zawarte moduły między innymi związane z algorytmami przetwarzania obrazów, algorytmami przetwarzania video, czy też video 3D. Posiada także

metody związane z detekcją cech i deskryptorów oraz detekcją obiektów. Ma także funkcje z interfejsem przechwytywania video, akceleracją GPU i wiele innych. Wybraliśmy tą bibliotekę, ponieważ była darmowa, oferowała bardzo duży wachlarz metod do odczytywania obrazy z kamery, przetwarzania go, czy nawet funkcje związane z kalibracją kamery. Skorzystaliśmy z tej biblioteki używając języka Python, gdyż implementacja w tym języku była dla nas bardzo czytelna, przejrzysta i prosta.

Babylon.js - silnik 3D wykorzystujący bibliotekę JavaScript do wyświetlania i modelowania obiektów 3D w przeglądarce z wykorzystaniem HTML5. Pierwsza wersja Babylon.js została opublikowana w 2013 roku przez programistów Microsoft. Kod źródłowy jest napisany w języku TypeScript i następnie jest kompilowany on do JavaScriptu, który jest natywnie wspierany przez wszystkie współczesne przeglądarki internetowe obsługujące HTML5 i WebGL, wykorzystywany przy renderowaniu modeli 3D. Bibliotekę Babylon.js wykorzystaliśmy ze względu na to, że jest ona jedną z najpopularniejszych tego typu bibliotek dostępnych aktualnie i posiada ona bardzo rozbudowaną dokumentację, a także forum użytkowników Babylon.js jest bardzo rozbudowaną platformą pomocną w użytkowaniu bez wcześniejszego doświadczenia z silnikami 3D. Dodatkowym atutem, który przemawia za użyciem BabylonJS jest dobrze udokumentowana współpraca biblioteki z programem Blender.

Python - interpretowany język programowania wysokiego poziomu posiadający rozbudowany pakiet bibliotek standardowych. Charakteryzuje się prostotą tworzenia aplikacji. Używaliśmy język Python w wersji 3.6. Języka tego używaliśmy ze względu na bibliotekę OpenCV, która posiada API w Pythonie, a “pod spodem” wywołuje funkcje tej biblioteki napisane w C++ (co powoduje minimalny narzut wydajnościowy).

Flask - biblioteka języka Python służąca do rozwijania aplikacji internetowych. Flask upraszcza projektowanie. Zapewnia przejrzysty schemat łączenia adresów URL, źródła danych, widoków i szablonów. Domyślnie zawiera również

deweloperski serwer WWW. Używaliśmy go ze względu na lekkość i prostotę. Alternatywne frameworki były znacznie bardziej rozbudowane i zawierały wiele niepotrzebnych nam narzędzi.

Flask-SocketIO - rozszerzenie biblioteki Flask. Pozwala aplikacjom napisanym w technologii Flask na dostęp do dwustronnej komunikacji klient-serwer z niskim opóźnieniem. Część kliencka może używać dowolnego klienta używającego bibliotekę SocketIO, której implementacja jest w językach Javascript, C++, Java czy Swift. Użycie tego narzędzia było dla nas niezbędne podczas połączenia pomiędzy częścią serwerową napisaną w języku python, a napisaną w języku Javascript częścią kliencką.

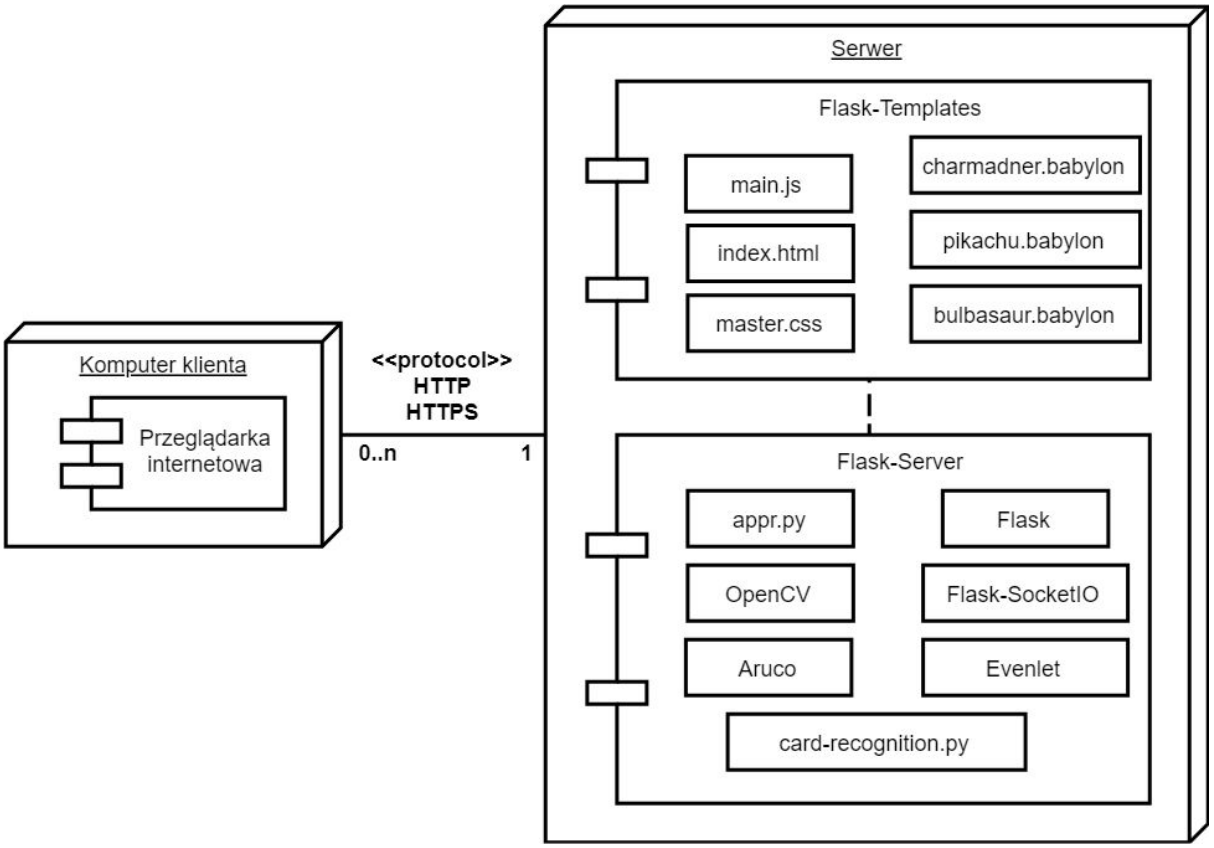
SocketIO - biblioteka używana przy aplikacjach czasu rzeczywistego. Umożliwia dwukierunkową komunikację pomiędzy klientem i serwerem. Implementowana była w naszej aplikacji po stronie klienta. Ze względu na użycie biblioteki Flask-SocketIO w części serwerowej.

Eventlet - biblioteka języka Python pozwalająca na używanie innego niż standardowy model wątków w środku interpretera Pythona. Dla kernela interpreter Pythona widoczny jest jako pojedynczy proces z wieloma otwartymi połączeniami. Jednak w środku interpretera osobne wątki są dostępne. Wykonanie każdego z wątków jest zaimplementowane w jak “najlżejszy” sposób. Dodatkowo interpreter nie wspiera szybkiego przełączania pomiędzy wątkami. Wykonanie wątku zostanie wstrzymane tylko w przypadku oczekiwania przez wątek na operację IO. Dzięki bibliotece kiedy wątek wywołuje funkcję do wykonania wejścia lub wyjścia wywoływana jest także funkcja Eventlet. Jeśli połączenie wątku czyta albo zapisuje to wszystko przebiega normalnie. W przeciwnym przypadku biblioteka przełącza się na inny wątek i uruchamia go. Ten proces powtarza się w nieskończoność. To pozwala aplikacji być pisanej w standardowy sekwencyjny sposób i nadal używać architektury zdarzeniowej(*ang. event-driven architecture*).

Base64 - biblioteka języka Python pozwalająca na kodowanie i dekodowanie danych zgodnie z standardem RFC 3548. Moduł został wykorzystany do kodowania obrazu w formacie Base64 i przesyłanie go używając websockets.

Blender - aplikacja do tworzenia trójwymiarowej grafiki komputerowej na licencji open-source. Blender pozwala na import oraz eksport modeli 3D w wielu formatach. Po załadowaniu modelu mamy gigantyczne możliwości edycji siatki obiektu, tekstur, materiałów, map UV, dodawania kości, map wypukłości, wag wierzchołków, modyfikatorów, symulacji oraz animowania modeli. Jest to prawdziwy kombajn multimedialny, który dodatkowo można rozszerzyć za pomocą palety dodatków i pluginów.
Głównymi funkcjonalnościami, które wykorzystaliśmy do edycja siatki obiektów oraz możliwość animowania modeli. Animacje zostały wykonane w oparciu o szkielet, który modyfikuje pozyjce przypisanych (z pewną wagą) wierzchołków wraz z przemieszczeniem (rotacją, skalowaniem) pojedynczej kości. Wszystkie animacje trwają domyślnie 250 klatek animacji w programie Blender. Przygotowane animacje były wypieczone (z angielskiego terminu *bake*), co oznacza, że ostateczny model zawierał informacje o położeniu każdego wierzchołka w każdej klatce animacji. Jest to wymagane przez moduł animacji w BabylonJS.
Kolejną funkcjonalnością użytą przez nas był oficjalny plugin wydany przez BabylonJS do eksportu modeli, tekstur i animacji z formatu .blend do formatu .babylon.
Wybór padł na Blendera, ponieważ jeden z członków naszego zespołu miał z nim już duże doświadczenie (około 500 godzin). Dodatkowo narzędzie jest darmowe, ma dużą społeczność, która chętnie pomaga użytkownikom oraz posiada plugin współpracujący z BabylonJS.
Wybrana przez nas wersja Blendera to 2.79.x, głównie ze względu, że to jest (w chwili pisania dokumentacji) najnowsza stabilna wersja Blednera. Plugin BabylonJS, który współpracuje z tą wersją Blendera jest w wersji 5.6.x

5. Architektura pierwotnej wersji systemu



Rys 3. Diagram wdrożenia aplikacji cARds

Diagram wdrożenia prezentuje budowę naszego systemu. Konsumentem naszych treści jest użytkownik, który korzysta z przeglądarki internetowej. Architektura zakłada serwer napisany w Pythonie, który wykrywa markery z obrazu. Z tych danych korzysta strona internetowa napisana w postaci szkieletów (Templates) frameworku Flask.

Pierwotna wersja systemu zakładała uruchomienie serwera napisanego w języku Python z wykorzystaniem biblioteki Flask. Serwer zajmował się odczytywaniem obrazu z podłączonej do niej kamery oraz wykrywał na takim obrazie markery. Wykryte markery były przekazywane do klienta za pomocą WebSocketów w postaci trójki (ID, T, R), gdzie ID to identyfikator wykrytego markera Aruco, T to wektor translacji, a R to wektor rotacji. Kolejny strumień danych służył do przesyłania obrazu odczytanego z makery.

W tym modelu klient łączył się do obu strumieni emitowanych przez serwer i odczytywał zawarte w nich dane. W razie potrzeby pobierał modele 3D z tego samego serwera używając plików statycznych. Klient używając przesyłanych danych o markerach, obrazu wideo i dostępnych modeli miał za zadanie renderowanie obrazu 3D nakładającego się na obraz wideo, co w efekcie dawało efekt rozszerzonej rzeczywistości.

Opis poszczególnych modułów

Card_recognition

Moduł do odczytu obrazu pochodzącego z kamery, rozpoznawania kart znajdujących się w polu widzenia kamery oraz wyznaczania ich dokładnych współrzędnych w przestrzeni.

Komponent posiada główną klasę `CardRecognition`, która przyjmuje jeden parametr do inicjalizacji - numer kamery, który oznacza z której będzie odczytywany obraz. Przypisywana jest także ścieżka, do zdjęć, z których będzie odbywać się kalibracja kamery. Moduł składa się z następujących metod:

- `create_cam_sream` - bezparametrowa metoda, która odczytuje z kamery jedną klatkę, którą zwraca.
- `show_frame` - bezparametrowa metoda, która w pętli `True` wywołuje metodę `create_cam_sream` w celu wyświetlania cały czas klatek z kamery, dopóki nie zostanie naciśnięty klawisz “q”.
- `calibration_cam_1` – bezparametrowa metoda, która przygotowuje parametry do kalibracji kamery. Głównym zadaniem jest odnajdywanie narożników szachownicy we wszystkich zdjęciach, przygotowanych do kalibracji. Metoda zwraca następujące parametry:
  - `objpoints` - wektor wektorów punktów wzorca kalibracji w przestrzeni rzeczywistej 3D ze wszystkich zdjęć szachownicy.
  - `imgpoints` - wektor wektorów projekcji punktów kalibracji na zdjęciach szachownicy, ze wszystkich zdjęć planszy
  - `gray` – obraz w szarości szachownicy, w dalszych częściach będzie wykorzystywany jedynie do wyznaczenia rozmiaru

obrazka, który wymagany jest do inicjalizacji wewnętrznej matrycy kamery

- `get_cam_matrix` – jako parametry wejściowe przyjmuje `objpoints`, `imgpoints`, `gray`, dzięki którym możliwa jest kalibracja kamery z użyciem metody z biblioteki `opencv`, czyli `calibrateCamera`, która zwraca pięć parametrów:
  - `ret`- średni błąd reprojekcji
  - `camera_matrix` – wyjściowa macierz zmiennoprzecinkowa o rozmiarze 3x3 oznaczająca orientację kamery. Zawiera w sobie długości ogniskowej fx i fy oraz współrzędne tak zwanego punktu głównego cx i cy. Metoda
  - `dist-coeffs` – wektor współczynników zniekształceń.
  - `rvecs` – wektor rotacji
  - `tvecs` – wektor translacji

  Metoda ta zwraca jedynie `camera_matrix oraz dist-coeffs`.
- `write_cam_parameters_to_file`- bezparametrowa metoda, która wywołuje metodę `calibration_cam_1` w celu otrzymania parametrów, które przekazuje do metody `get_cam_matrix`, wynikowe parametry zapisuje do pliku JSON. `Metoda nic nie zwraca`.
- `get_camera_parameters_from_file` – bezparametrowa metoda, która odczytuje wcześniej zapisane parametry kamery z pliku JSON, które następnie zwraca.
- `detect_aruco` – jako parametr wejściowy przyjmuje jeden parametr `image`, który jest jedną klatką kamery. Metoda tworzy z klatki szary obraz, następnie przy pomocy funkcji biblioteki opencv.`aruco`, czyli `Dictionary_get`, uzyskuje słownik aruco, następnie tworzy parametry przy pomocy metody `DetectorParameters_create()`. Na samym końcu wyznacza współrzędne rozpoznanych markerów Aruco oraz ich id, przy pomocy funkcji `detectMarkers()`. Metoda zwraca te współrzędne oraz id.

- `get_rotation_and_translation` – metoda, która przyjmuje następujące parametry:
    - `corners` – współrzędne rozpoznanych markerów Arcuo
    - `camera_matrix` – macierz orientacyjna kamery
    - `dist_coeffs` – wektor współczynników zniekształceń
    - `markerLength` – długość markera, domyślnie ustawiona jako 25, gdyż takie rozmiary markerów są na kartach.

  Metoda przy pomocy funkcji `estimatePoseSingleMarkers` z biblioteki opencv.`aruco` wyznacza wektor rotacji i translacji dla wykrytych markerów Aruco, które są zwracane.
- `get_camera_parameters` – bezparametrowa metoda, która odczytuje parametry z pliku, jeśli plik jest pusty to wywołuje metodę zapisu parametrów do pliku i ponownie odczytuje plik. Zwraca dwa parametry : `camera_matrix` oraz `dist_coeffs`, które zostały już wyżej dokładnie opisane.
- `get_corners_and_roration_and_translation` – funkcja, która jako parametr wejściowy przyjmuje jedną klatkę z kamery. Następnie przy użyciu metody `get_camera_parameters()` otrzymuje dwa parametry `camera_matrix` oraz `dist_coeffs`. Następnie wywołuje funkcję `detect_aruco()`, która zwraca współrzędne aruco oraz id. Na sam koniec przy użyciu metody `get_rotation_and_translation()`, do której przekazywane są wcześniej otrzymane parametry, otrzymuje wektor rotacji i translacji. Funkcja zwraca te wektory oraz współrzędne i identyfikatory.
- `show_asix` – funkcja, która w czasie rzeczywistym wyświetla na wykrytych markerach Aruco osie współrzędnych x, y oraz z. W metodzie na samym początku wywołana jest metoda `get_camera_parameters`, a następnie wykonuje się pętla `True`, w której odczytywana jest klatka kamery, wykrywane są markery Aruco, obliczane są wektory rotacji i translacji dla tych markerów i na sam koniec przy pomocy metody `drawAxis` z biblioteki

`opencv.aruco`, rysowane są osie współrzędnych. Działania te wykonywane są dopóki, nie zostanie wciśnięty klawisz “q”.

Websockets

Moduł służy do odczytywania poszczególnych klatek oraz parametrów odzwierciedlających położenie obiektu w przestrzeni z modułu Card_recognition.

Wartości parametrów zapisywane są w formacie JSON (*ang. JavaScript Object Notation*). Obiekt JSON zawiera parametry odzwierciedlające położenie modelu 3D, czyli wartość x, y, z na osiach oraz rotacje rx, ry, rz według poszczególnych osi. Do tego samego obiektu JSON dodawany jest numer ID. Numer ID informuje jaki rodzaj kodu Aruco został wykryty (ponieważ każdy kod Aruco zawiera w sobie jakiś identyfikator). Identyfikator ten informuje nas także o wybranym modelu. Drugi websocket przesyła zakodowaną w Base64 pojedynczą klatkę. Przetwarzane i wysyłane jest około 40 klatek na sekundę. Z tą samą częstotliwością wysyłane są parametry dotyczące położenia modelu oraz jego identyfikator. Pojedyncza klatka kodowana jest za pomocą Base64 ze względu na to, że musi być w tym formacie po stronie klienckiej gdzie jest wyświetlana użytkownikowi.

Moduł składa się z następujących funkcji:

- `index()` - funkcja, która z pomocą wbudowanej w bibliotekę Flask *render_template* renderuje stronę HTML o nazwie *modelShowcase*. Plik HTML musi znajdować się w katalogu templates. Do funkcji *render_template* przekazywany jest także parametr *title* z tytułem strony.
- `stream_video()` - funkcja kodująca klatkę obrazu z pomocą Base64 oraz emitująca ją na websocket *image*. Dodatkowo funkcja wysyła jako obiekt JSON parametry odzwierciedlające położenie modelu 3D czyli rozbite na poszczególne elementy wektory rotacji i translacji oraz

identyfikator modelu. W funkcji używana jest metoda sleep biblioteki Eventlet, która pozwala ograniczyć ilość wysyłanych parametrów, co zmniejsza obciążenie serwera. Żeby móc używać tej metody po deklaracji funkcji *stream_video* wymagane jest użycie metody spawn także pochodzącej z biblioteki Eventlet.

Frontend

Moduł służy do prezentacji obrazu i modeli klientowi oraz służy jako interfejs użytkownika.

Moduł posiada jedną główną klasę - `ModelPreview` odpowiedzialną za renderowanie obiektów 3D i która w swoim konstruktorze tworzy nową scenę 3D. Klasa składa się z następujących metod:

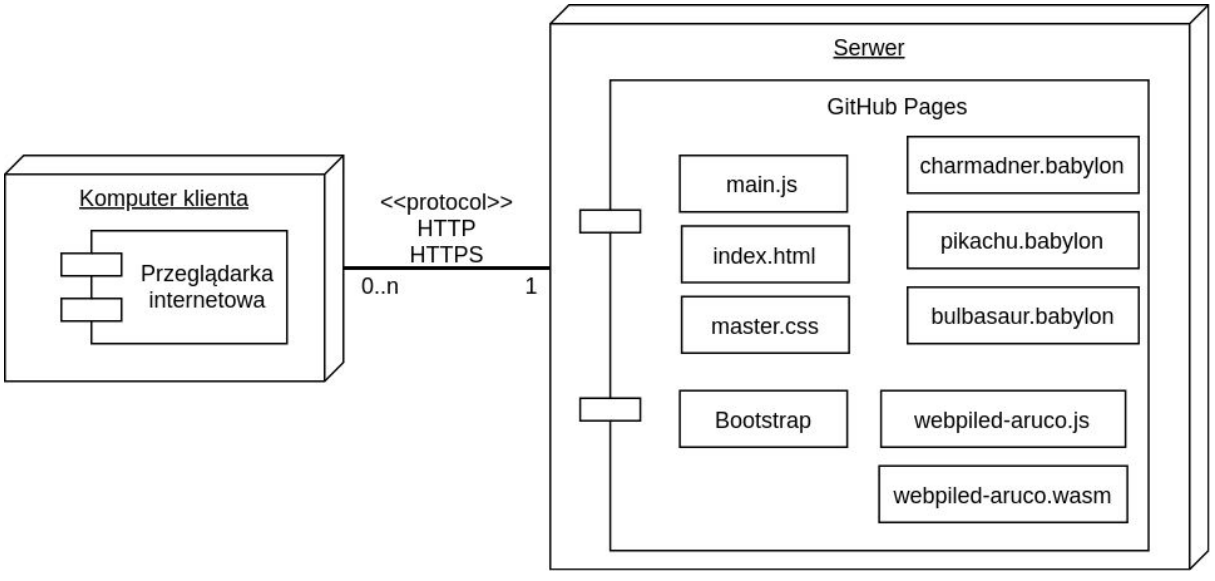
- `fillScene()` – bezargumentowa metoda wykorzystywana w konstruktorze klasy `ModelPreview` i jej zadaniem jest przygotowanie przygotowanie sceny 3D poprzez zainicjowanie nowej kamery - `BABYLON.UniversalCamera` oraz ustawienie odpowiedniego oświetlenia sceny - `BABYLON.HemisphericLight`. Następnie po ustawieniu powyższych elementów tworzona jest dwuwymiarowa płaszczyzna na której wyświetlana jest tekstura wideo pobierana z kamerki internetowej komputera za pomocą funkcji `BABYLON.VideoTexture.CreateFromWebCam` przyjmującej w parametrach wyrażenie lambda, wewnątrz którego wywoływana jest metoda `findMarkersInImage` znajdująca markery Aruco w obrazie rejestrowanym przez kamerę.
- `prepareMesh(meshID)` - metoda przygotowująca obiek o podanym `meshID` do wyświetlenia na scenie. Podstawową jednostką rotacji w bibliotece Babylon.js są kwaterniony. Dla każdego obiektu należy ustawić początkową wartość kwaternionu, używamy do tego wbudowanej metody - `BABYLON.Quaternion.Identity()`, oprócz tego widoczność początkowa jest ustawiana na 0 (obiekt jest niewidoczny).

- `loadMesh(path)` - metoda importująca wszystkie struktury obiektów 3D na podstawie podanej ścieżki - parametr `path`. Następnie wszystkie obiekty są przygotowywane do wyświetlenia za pomocą metody `prepareMesh` oraz rozpoczynane są ich sekwencje animacji - `this.scene.beginAnimation`.
- `loadMeshes()` - metoda importująca wszystkie stworzone obiekty pokemonów za pomocą `loadMesh`.

Oprócz klasy `ModelPreview` moduł Frontend składa się również z pliku `index.html` odpowiedzialnego za umożliwienie prezentacje wszystkich funkcjonalności systemu po stronie klienta za pomocą przeglądarki internetowej. Plik ten składa się z 2 głównych elementów: pierwszy z nich jest elementem nawigacyjnym, który pozwala na płynne przechodzenie pomiędzy widokiem kamery a widokiem wygenerowanych markerów Aruco dla poszczególnych obiektów. Drugim elementem jest kontener renderujący obraz rejestrowany i obsługiwany przez klasę `ModelPreview`.

6. Architektura aktualnej wersji systemu

Ogólna architektura systemu



Rys 4. Diagram wdrożenia aplikacji cARds

Diagram przedstawia nową wersję systemu. W porównaniu do starej usunięty został serwer napisany w Pythonie. Do naszych celów wystarczy zwykły serwer HTTP serwujący statyczne pliki. W tym przypadku jest to GitHub Pages.

Nowa architektura systemu zakłada model Fat-Client, co oznacza, że większość obliczeń wykonywana jest po stronie klienta. Serwer (w tym wypadku GitHub Pages) serwuje statyczne pliki do przeglądarki. Domyślnie punktem wyjścia jest index.html. Całość pracy sprowadzona jest na główny skrypt w JavaScript, który jest odpowiedzialny za cały cykl życia sceny 3D.

Wybraliśmy GitHub Pages jako serwer dla nasze aplikacji z kilku powodów:

- Jest darmowy - korzystanie z Pages jest darmowe, dzięki czemu nie musieliśmy przejmować się kosztami projektu.
- Jest zintegrowany z GitHub - dzięki temu nie trzeba było tworzyć nowych kont na stronach oferujących hosting. Mogliśmy skorzystać ze znanego nam serwisu, co przyspieszyło prace.
- Aplikacja jest repozytorium gita - dzięki temu uruchomienie najnowszej wersji kodu było równoważne z wykonaniem komendy *git push*.
- Jest prosty w obsłudze - aby strona działała, należy odpowiednio nazwać projekt, a także posiadać plik index.html w głównym folderze aplikacji.

Skrypt na samym początku tworzy scenę 3D, ustawia na niej kamerę oraz oświetlenie, a także “wyświetlacz”. W praktyce “wyświetlacz” jest zwykłą płaszczyzną z prostym materiałem oraz wideo teksturą. Strumień wideo pochodzi z dowolnej kamery internetowej podłączonej do klienta. Po utworzeniu sceny, skrypt sprowadza do pamięci dostępne po stronie serwera modele i ustawia je na scenie ze stuprocentową przezroczystością.

W tym momencie skrypt przechodzi do pętli renderującej. Przy każdym obiegu pobierany jest aktualna klatka obrazu z kamery. Dla pobranego obrazu

znajdowane są wszystkie markery Aruco dzięki skompilowanemu modułowi Aruco.

Dla każdego markera przetwarzany jest powiązany z nim model. Jeśli marker pojawił się, to przezroczystość z modelu jest usuwana. W przeciwnym razie dla każdego modelu pamiętany jest licznik nieobecnych klatek. Jest to słownik mapujący ID modelu/markera do liczby określającej liczbę następujących po sobie klatek w których marker powiązany z modelem był niewidoczny. Jeśli przy danej klatce nie ma odpowiadającego markera, to licznik jest zwiększany. Jeśli marker się pojawi, to licznik jest zerowany. Jeśli przez 10 klatek marker będzie niewidoczny, to powiązany z nim model jest ukrywany.

Każdy marker po przetworzeniu poza identyfikatorem zwraca także informację w postaci wektora translacji oraz wektora rotacji. Aby móc skorzystać z nich w BabylonJS to przesunięcie w osi X trzeba odwrócić - pomnożyć -1. Jest to podyktowane faktem, że model który chcemy wyświetlić musi być wyświetlony poprawnie. Bez tej poprawki, modele byłyby odbite względem osi X.

W celu wykonywania rotacji obiektem BabylonJS używa systemu quaternionów. Kąty otrzymane z modułu Aruco są w formacie Euler-Rodriguez i należy jest przetworzyć na reprezentację w postaci quaternionu.

Zastosowanie WebAssembly [5]

Aby wykryć pozycję markera na obrazie po stronie klienta mieliśmy do wyboru albo zastosowanie biblioteki przepisanej z C do JavaScriptu ręcznie, albo użyć metody kompilacji C/C++ do WebAssembly.

Cały proces jest całkowicie zautomatyzowany z użyciem skryptu powłoki bash, który wykonuje wszystkie niezbędne operacje pobierania danych kompilacji oraz tworzenia skryptu uruchomieniowego dla przeglądarki.

WebAssembly możemy potraktować jak język maszynowy do procesorów x86, ARM lub innych. Dlatego kompilacja do WebAssembly jest operacją podobną do operacji kompilacji kodu w języku C++ do asemblera konkretnego procesora. Wynikiem operacji kompilacji do WebAssembly jest plik .wasm, który można wywoływać ze skryptu napisanego w JavaScript. Największe zalety zastosowania tego podejścia to automatyczne generowanie modułu na podstawie kodu źródłowego oraz wysoka wydajność kody WebAssembly. Dzięki temu podejściu aktualizacja biblioteki źródłowej nie jest problemem, bo można wykonać kompilację ponownie i otrzymany moduł jest w 100% zgodny z nową wersją kodu bazowego.

Narzędzie które umożliwia nam przekonwertowanie plików źródłowych modułu do formatu .wasm jest Emscripten. Emscripten jest tak naprawdę zbiorem narzędzi służących do kompilacji. Pod spodem korzysta z LLVM (Low Level Virtual Machine). Emscripten w celu kompilacji modyfikuje zachowanie CMake, w taki sposób, aby docelowym formatem kompilacji było .wasm. Ponieważ CMake oraz make chodzą w parach Emscripten udostępnia też komendę emmake, która jest specjalną otoczką do make, której docelowym formatem wyjściowych plików jest .wasm.

Dodatkowym krokiem jest stworzenie specjalnego wrappera w języku JavaScript, który z modułu Aruco wyciągnie tylko interesujące nas funkcjonalności. Ten skrypt jest defacto plikiem, który używamy do wykrywania pozycji markerów na obrazie.

Wszystkie powyższe kroki są zawarte w gotowym skrypcie powłoki bash, który produkuje wszystkie niezbędne pliki dla aplikacji webowej [4]. Wynikowy plik JavaScript jest biblioteką, z której korzystamy. Dla uproszczenia wykorzystaliśmy już skompilowany moduł Aruco.

Frontend

W porównaniu do poprzedniej wersji obecna wersja jest w większości identyczna do poprzedniej. Poza refactoringiem , który dodał do kodu czytelność to sama struktura kodu jest praktycznie identyczna.

Główne zmiany nastąpiły w funkcjach które odpowiadały za pobieranie obrazu oraz wykrywanie znaczników. Poprzednia wersja pobierała obraz oraz informacje o znacznikach z dwóch niezależnych gniazd sieciowych (WebSocketów), a następnie używała ich do pozycjonowania modeli.
Obecne podejście czyta obraz z kamerki internetowej w postaci tekstury wideo, a następnie dla każdej klatki wykrywane są markery Aruco.
Dlatego właśnie zmodyfikowane oraz dodane zostały funkcje:

- `fillScene()` – metoda została rozszerzona tworzenie dwuwymiarowej płaszczyzny na której wyświetlana jest tekstura wideo pobierana z kamerki internetowej komputera za pomocą funkcji `BABYLON.VideoTexture.CreateFromWebCam` przyjmującej w parametrach wyrażenie lambda, wewnątrz którego wywoływana jest metoda `findMarkersInImage` znajdująca markery Aruco w obrazie rejestrowanym przez kamerę.
- `findMarkersInImage()` – funkcja odpowiedzialna za wykrywanie markerów na zarejestrowanym obrazie. Przyjmuje ona cztery parametry: `videoTexture`, `meshes`, `frameSinceSeen`, `callback`. Pierwsze trzy parametry zawierają odpowiednio: teksturę wideo w formacie `Babylon.VideoTexture`, słownik wszystkich dostępnych struktur obiektów i liczbę klatek od ostatniego zarejestrowania danego markera Aruco. Natomiast w parametrze `callback` przekazywana jest funkcja `translateRotateMesh`. Funkcja ta na podstawie znalezionych markerów uwidacznia poszczególne obiekty lub przy zniknięciu danego markera Aruco zmienia status odpowiedniego modelu na niewidoczny.

- `findMarkersInImage()` - funkcja odpowiedzialna za aktualizowanie pozycji modelu oraz jego rotacji. Przyjmuje ona 5 argumentów: `id`, `framesSinceSeen`, `mesh`, `t`, `r`. Argumenty odpowiadają odpowiednio za: identyfikator modelu, liczbę klatek od ostatniego zarejestrowania markera Aruco, model 3D i macierze rotacji oraz translacji.

7. Napotkane problemy

Problem 1 *- rozpoznawanie kart*

Początkowym założeniem była identyfikacja poszczególnych kart przy pomocy kodów QR. Każdy z kodów zawierał informacje o karcie. Niestety prosta aplikacja potrafiła rozpoznać tylko jeden kod, co było dla nas niewystarczające, gdyż zakładaliśmy możliwość rozpoznania wielu kart w tym samym czasie.

Rozwiązanie problemu 1

Do identyfikacji każdej karty użyliśmy kodów Aruco, dla których algorytm dekodowania informacji jest w nich zawartych. Jest możliwość zidentyfikować wszystkie, które są w polu widzenia kamery. To rozwiązanie zdecydowanie nas usatysfakcjonowało.

Problem 2 *- Dokładna lokalizacja karty - jej położenie i orientacja*

Pierwszym założeniem było lokalizowanie karty przy pomocy detektora SIFT (Scale Invariant Feature Transform).

Początkowe etapy znajdowania pozycji karty:

- Do detekcji oraz opisu cech użyliśmy detektora SIFT, który charakteryzuje się odporonością na zmiany i przekształcenia, dlatego wybór padł na niego.
- Do dalszych dopasowań pomiędzy wykrytymi cechami, wykorzystaliśmy algorytm BFMatcher (Brute-Force descriptor matcher).
- Kolejnym etapem było znalezienie perspektywiczną transformację między dwiema płaszczyznami.

- Dzięki wynikom uzyznamym z detekcji oraz dopasowań uzyskaliśmy parametry 2D, lecz naszym celem był 3D, czyli współrzędne x,y,z, których nie było można w łatwy sposób uzyskać.

Rozwiązanie problemu 2

Do uzyskania pozycji karty użyliśmy markerów Aruco. Przechowują w sobie informacje, dzięki którym algorytm dekodowania ich zwraca współrzędne 2D markera. Dzięki wcześniejszej kalibracji kamery i otrzymanych z niej parametrów oraz współrzędnych Aruco możliwe jest wyliczenie wektora rotacji i translacji. Dzięki nim uzyskaliśmy to czego chcieliśmy, czyli pozycję karty w przestrzeni.

Problem 3 *- Inne parametry położenia występujące pomiędzy środowiskiem Python, a Babylon.js.*

Problem wystąpił z wielu powodów. Pierwszym z nich był inny rozmiar obrazu (rozdzielczość), na którym były wykrywane markery, a rozmiarem (rozdzielczością) kamery umieszczonej na scenie. Problem nie był trywialny, bo różnica rozdzielczości nie była liniowa. Każdy obraz miał inne proporcje co przy jakimkolwiek normalizowaniu zdeformowało by jeden z obrazów.

Kolejnym problemem było zaadaptowanie danych uzyskiwanych w języku Python do sceny 3D. Scena operuje w innym układzie współrzędnych niż moduł wykrywający translację i rotację. Sama translacja była liczona w dziwnych jednostkach rzędu setek wartości, z kolei w scenie 3D kamera miała jednostki do kilku wartości. Samo odsunięcie kamery, aby objęła większy zakres jednostek sceny, nic nie dało, bo oddalenie kamery skutkowało znacznym zmniejszeniem wyświetlanych modeli. Powiększenie tych modeli skutkowało ponowieniem problemu.

Próbowaliśmy także dane normalizować, aby z formatu modułu Python uzyskać koordynaty w BabylonJS. Niestety okazało się to ślepą uliczką. Same modele sprawiały wrażenie, jakby latały nad znacznikiem, dodatkowo trzymały się tylko przy drobnych ruchach kartą. Przemieszczenie karty o centymetr w dowolną stronę sprawiało, że model wychodził znacznie poza obszar obserwowany przez kamerę.

Rozwiązanie problemu 3

Rozwiązaniem tego problemu było całkowite przeprojektowanie systemu. Zmieniliśmy model aplikacji na typ Fat-Client, co oznacza, że wszystkie obliczenia są przeprowadzane bezpośrednio po stronie klienta. Teraz operacja wyznaczenia pozycji markera Aruco oraz macierzy translacji i rotacji odbywa się w odniesieniu do sceny 3D zawierającej wyświetlane modele. Efekt ten uzyskaliśmy dzięki zastosowaniu Aruco bezpośrednio po stronie frontendu, a dane które ten moduł przetwarza pochodzą z tekstury wideo ładownej bezpośrednio do sceny 3D.

Problem 4 - *Format FBX 6100 nie jest wspierany przez Blender*

Problem pojawił się w przypadku importu modeli 3D pokemonów pobranych ze strony Free3D. Okazało się, że wszystkie modele były w starym standardzie .fbx, który przestał być wspierany przez Blender wiele wersji temu. Było de-facto jedyny format, który mogliśmy pobrać i użyć w Blenderze.

Rozwiązanie problemu 4

Rozwiązaniem okazał się stary i nie wspierany już program Autodesk FBX Converter 2013 w wersji 3. Jest to program dostępny na komputery z systemem Windows oraz Mac. Jedyna jego funkcjonalność to aktualizowanie formatu FBX 6100 do nowszego formatu FBX 7100 (z roku 2011), wspieranego już przez Blendera.

Problem 5 - *Przetwarzanie obrazu po stronie serwera wymaga od niego dużej mocy obliczeniowej przy wielu użytkownikach.*

Ponieważ początkowo obraz był strumieniowany z serwera wymagało to coraz większej mocy obliczeniowej dla większej ilości klientów.

Rozwiązanie problemu 5

Ostatecznie obliczenia odbywają się po stronie klienta, a serwer służy tylko do serwowania plików statycznych. Pozwala to korzystać wielu użytkownikom bez spadku wydajności aplikacji.

Problem 6 - *Aplikacja internetowa hostowana na Github Pages nie działa na wszystkich urządzeniach.*

Cechą aplikacji internetowych jest to, że każda z przeglądarek obsługuje kod w trochę inny sposób. To co działa na jednej, na innej przeglądarce może być problematyczne. W tym przypadku aplikacja nie działała na Operze, głównie z tego powodu, że Opera nie wykrywa podłączonej kamery internetowej.

Innym napotkanym błędem było to, że na jednym komputerze strona nie działała na wszystkich przeglądarkach, a na innym (dodam dokładnie tym samym modelu komputera i z tym samym systemem operacyjnym - Ubuntu 18.10)

Rozwiązanie problemu 6

Nie znaleźliśmy rozwiązania tego problemu. Głównie bierze się to z faktu, że technologie które zastosowaliśmy mogą nie być w pełni wspierane przez wszystkie przeglądarki. Inny sprawa jest taka, że możliwe, że jeden z komputerów miał uszkodzone przeglądarki internetowe, bo już wcześniej były na nim problemy z aplikacjami pobranymi przez manager *snap*.

Problem 7 - *Problem z wersją Pythona przy używaniu detektora SIFT*

Naszym początkowym zamysłem, było wykrywanie kart przy pomocy detektora SIFT. Zakładaliśmy także, że będziemy wszystko implementować w języku Python w wersji 3.7. Niestety wystąpiły pewne komplikacje. Metoda `cv2.xfeatures2d.SIFT_create()` była już niedostępna w tej wersji Pythona.

Rozwiązanie problemu 7

Chwilowo zrezygnowaliśmy z wersji 3.7 na rzecz Pythona w wersji 3.6, dzięki temu nie było żadnych problemów z detektorem SIFT, działał poprawnie. Ostatecznie jednak zrezygnowaliśmy z tego rozwiązania, gdyż użyliśmy

markerów Aruco, które nie wymagają implementacji w Python 3.6. Dzięki temu cała aplikacja powróciła do implementacji w języku Python 3.7.

Problem 8 - *Problem z animacją modelu Charmandera*

Model Charmandera zachowywał się w dziwny sposób po operacji wypieku animacji (operacji bake) i eksporcie do formatu .babylon. Mianowicie animacje się usuwały, a cały model został wykręcony w losowych kierunkach. Było to dziwne, ponieważ operacja eksportu nie powinna mieć żadnego wpływu na obecne w Blenderze klatki animacji. Dodatkową pracę utrudniał fakt, że ta operacja mieszała nie tylko z animacjami, ale ze samymi kośćmi przestawiając je dziwne pozycje.

Rozwiązanie problemu 8

Rozwiązaniem problemu okazało się uproszczenie animacji. Całkiem możliwym źródłem błędów było przypisanie zbyt dużej liczby wierzchołków do pojedynczej kości, przez co system eksportujący mógł wariować. Inne próby rozwiązania problemu opierały się na ręczne malowanie wag wierzchołków, automatyczne równoważenie obciążenia kości, aplikowanie obecnej skali, pozycji i rotacji modelu jako wartości podstawowe, a także całkowite usunięcie starego i dodanie nowego systemu armatury do modelu. Jednakże owocnym działaniem było ograniczenie animacji w tym modelu.

8. Instrukcja użytkowania aplikacji

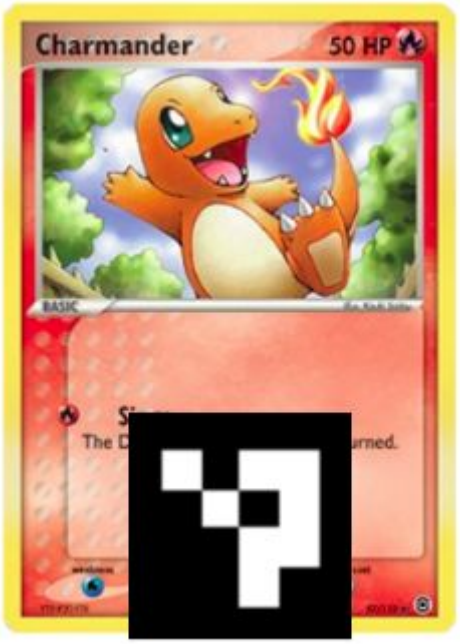
Aplikacja dostępna jest pod linkiem: https://axolotlofconflagration.github.io/.

Karty używane w naszej aplikacji zostały wzbogacone o kody Aruco, które zostały wygenerowanie przez generator online http://chev.me/arucogen/.

Użytkownik po uruchomieniu aplikacji znajduje się na stronie głównej, na której widzi rejestrowany obraz z wybranej przez niego kamerki w czasie rzeczywistym. Następnie po przesunięciu karty z wygenerowanym markerem Aruco w pole widzenia kamery na ekranie użytkownika renderowane i animowane są odpowiednie modele, zgodne z kartą która widoczna jest na ekranie.

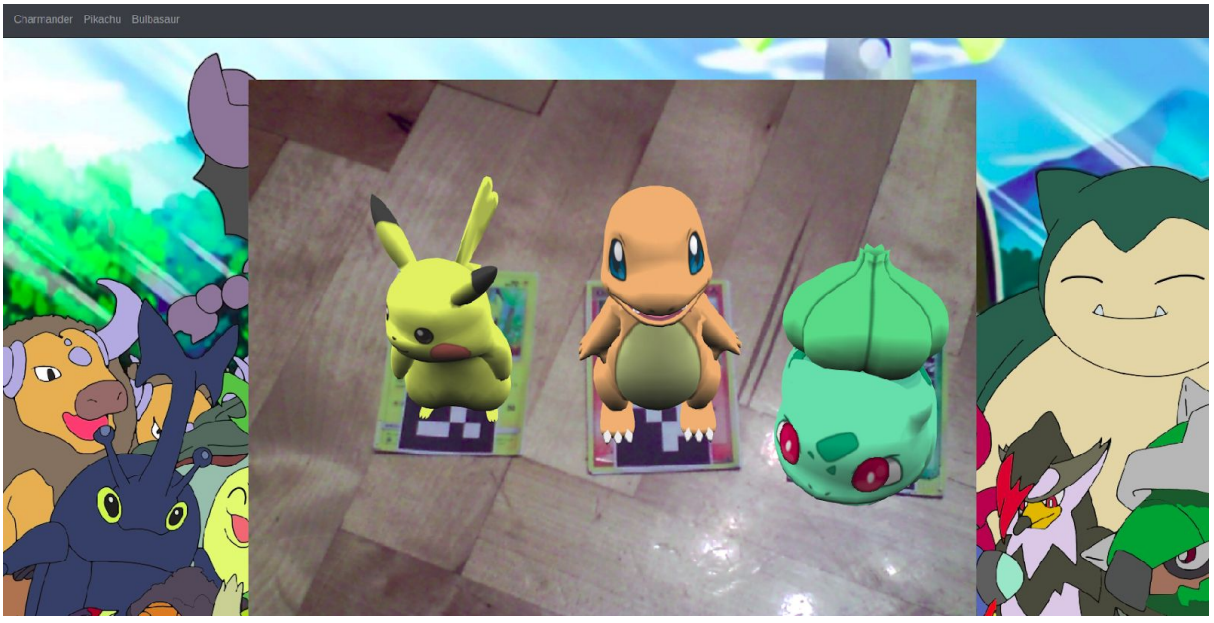
Dodatkowo użytkownik ma możliwość, za pomocą panelu nawigacyjnego w pasku u góry ekranu, wyświetlenia na ekranie odpowiednich markerów Aruco, które może np. wyświetlić na ekranie urządzenia mobilnego, aby móc przetestować działanie aplikacji w przypadku nieposiadania odpowiednio przygotowanych kart.

Rys 5. Karta Bulbasaur.

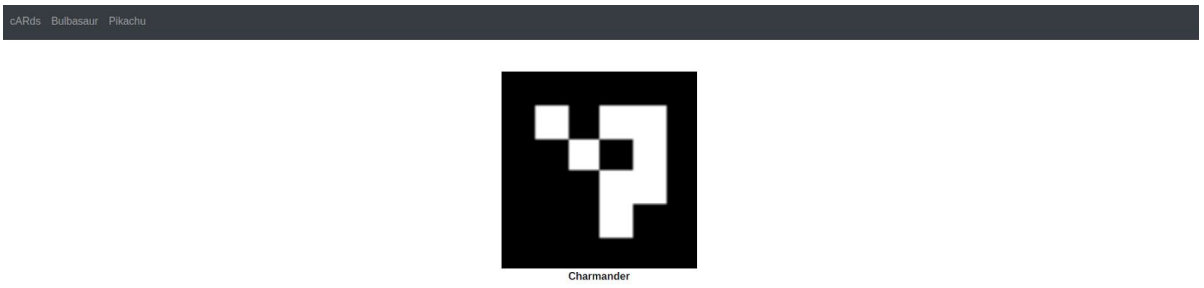


Rys 6. Karta Charmander.

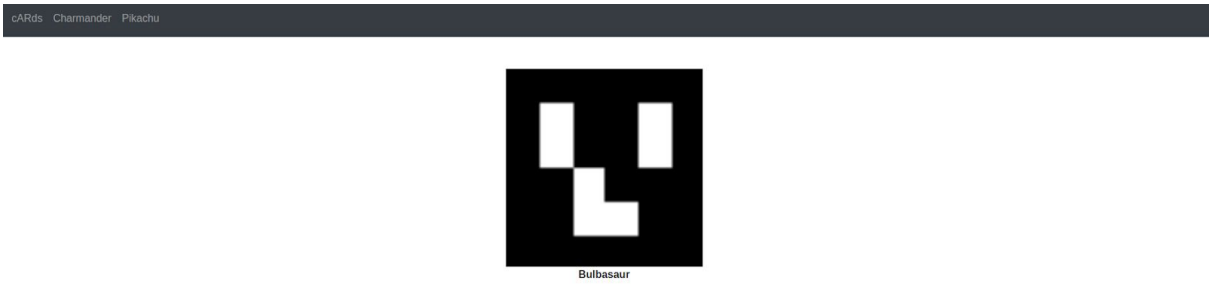
Rys 7. Karta Pikachu.



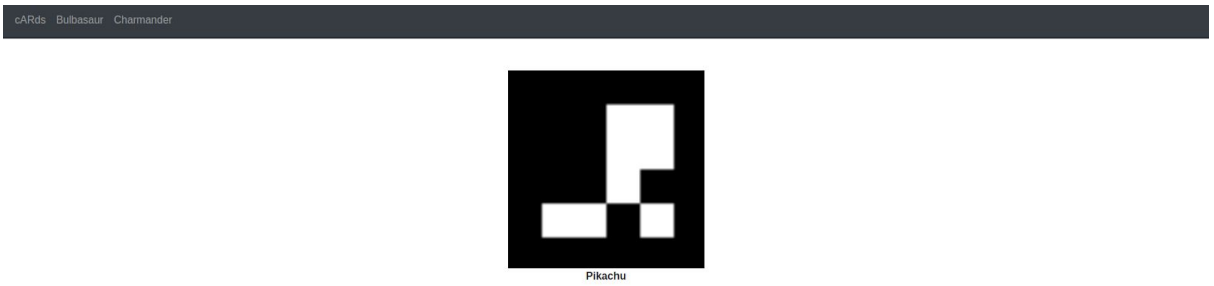
Rys 8. Strona główna wyświetlająca działanie aplikacji.



Rys 9. Strona z kodem aruco dla modelu Charmander.



Rys 10. Strona z kodem aruco dla modelu Bulbasaur.



Rys 11. Strona z kodem aruco dla modelu Pikachu.

9. Repozytoria i powiązane linki

Aplikacja udostępniona jest publicznie dzięki Github Pages pod następującym linkiem ⇒ http://axolotlofconflagration.github.io/

Repozytorium dla Github Pages ⇒ https://github.com/AxolotlOfConflagration/AxolotlOfConflagration.github.io

Repozytorium projektu ⇒
https://github.com/AxolotlOfConflagration/cARds

Filmiki prezentujące działanie aplikacji ⇒
https://youtu.be/849jNxuCtvg (prezentuje wszystkie trzy modele wraz z ich animacjami)

10. Podsumowanie

Kierunek rozbudowy systemu

Uważamy, że nasz aplikacja idelanie nadawałaby się na systemy mobilne. Testowaliśmy projekt łącząc się z telefonu przez przeglądarkę Google Chrome na system Android. Aplikacja działała jednak w zależności od modelu telefonu często jedyną kamerą z której można było korzystać byłą kamera przednia co było niepraktyczne. Dodatkowo widok z kamery na ekranie jest nieodpowiednio przeskalowany. Chcielibyśmy przystosować aplikację na systemy mobilne i rozwijać ją w tym kierunku.

Następnym naszym krokiem, który chcielibyśmy zrealizować to dodanie znacznie większej ilości Pokemonów. Aktualnie oferujemy wyświetlanie trzech modeli 3D. Sama pierwsza generacja Pokemon oferuje 151 pokemonów. Wymagałoby to od nas znacznie więcej pracy nad modelami, ale wprowadziłoby też różnorodność.

Kolejną funkcjonalnością, którą chcielibyśmy dodać to interakcje pomiędzy modelami 3D reprezentującymi konkretne Pokemony. Ponieważ aplikacja początkowo tylko miała służyć tylko do urozmaicenia gier karcianych poprzez pokazywanie na kartach animowanych modeli 3D, a używana miała być głównie na wydarzeniach gdzie takie rozgrywki się pokazuje to ciekawą dodatkową opcją byłyby dodatkowe interakcje. Pozwoliłoby to widowni na jeszcze lepszą

interakcję z graczami, a trwająca rozgrywka na pewno byłaby bardziej atrakcyjna.

11. Literatura

[1] https://flask-socketio.readthedocs.io/en/latest/

[2] http://www.hydrogen18.com/blog/fast-async-python-eventlet.html

[3] https://python101.readthedocs.io/pl/latest/webflask/

[4] https://github.com/syntheticmagus/demo-webpiling-end-to-end

[5] https://medium.com/@babylonjs/marker-tracking-in-babylon-js-ce99490be1dd