



POLSKO-JAPOŃSKA AKADEMIA
TECHNIK KOMPUTEROWYCH

WYDZIAŁ INFORMATYKI
KATEDRA SIECI KOMPUTEROWYCH
SIECI URZĄDZEŃ MOBILNYCH

Kamil Kacprzak
s14004

**Rozwój technologii wirtualnej
rzeczywistości na przykładzie
rękawicy-kontrolera**

Praca inżynierska
dr inż. Michał Tomaszewski

Warszawa, Czerwiec, 2020

Spis treści

1 Wstęp	1
2 Zagadnienia	3
2.1 Czym jest wirtualna rzeczywistość?	3
2.2 Działanie kontrolerów w VR	5
2.3 Wykorzystanie dloni jako kontrolera	7
3 Komponenty rękawicy-kontrolera	9
3.1 IMU - Inercyjna jednostka pomiarowa	10
3.1.1 Sensory	10
3.1.2 Stopnie swobody	11
3.2 Śledzenie położenia palców	14
3.3 Łączność przy użyciu Bluetooth niskiego poboru mocy	14
4 Projekt rękawicy	16
4.1 Przegląd podzespołów użytych w projekcie	16
4.1.1 Mikrokontroler	17
4.1.2 Czujnik wygięcia	19
4.1.3 Rezystor	21
4.2 Budowa rękawicy	23
4.3 Oprogramowanie mikrokontrolera	27
4.4 Podsumowanie	36
5 Dedykowana aplikacja w systemie Android	37
5.1 Interfejs	38
5.2 Komunikacja	41
5.2.1 Obsługa połączenia Bluetooth Low Energy	44
5.2.2 Pobieranie danych	46
5.3 Google Sceneform SDK	50
5.3.1 Model ręki	51
5.3.2 Rozpoznanie i animacja modelu	55

5.3.3	Rotacja	58
5.3.4	Przesunięcie	61
5.3.5	Kalibracja	62
6	Dalszy rozwój projektu	64
6.1	Problemy mikroprocesora	64
6.2	Problemy budowy i czujników wygięcia	65
6.3	Animacja modelu	67
6.4	Błąd rotacji	67
6.5	Problem obliczania przesunięcia	68
7	Podsumowanie	69

Spis rysunków

2.1	Kontrolery VR	6
3.1	Model prezentujący sześć stopni swobody	12
3.2	Problem blokady gimbala (Przed i po obrocie)	13
4.1	Opis wejść oraz wyjść Arduino Nano 33 BLE	18
4.2	Sensor wygięcia własnego wykonania	21
4.3	Oznaczenia rezystorów	22
4.4	Układ dzielnika napięcia	23
4.5	Poglądowy układ kontrolera przedstawiający sposób podłączenia dzielnika napięcia	24
4.6	Efekt końcowy rękawicy-kontrolera	25
5.1	Interfejs użytkownika	39
5.2	Modele animacji dloni.	52

Streszczenie

Tematem pracy jest “Rozwój technologii wirtualnej rzeczywistości na przykładzie rękawicy-kontrolera” w której pokazano czym jest świat wirtualny a także w jaki sposób kontroluje się w nim środowisko. Przedstawiono tematykę kontrolerów a także rękawicy-kontrolera, oraz ich podstawowych funkcjonalności. W rozdziale 3 została pokazane problemy oraz rozwiązania z jakimi się mierzą współczesne rozwiązania. Następnie pokazano własną implementację rękawicy-kontrolera spełniającej podstawowe funkcje. Rękawica ta przesyła dane bezprzewodowo. Przykładowy odbiornik tych danych, w postaci aplikacji na Androida został pokazany w rozdziale 5. Rozdział ten porusza takie tematy jako implementacja łączności bluetooth, ustalenie orientacji, położenia a także animacja modelu dłoni na podstawie otrzymanych danych z kontrolera. Na podstawie pracy z projektem zostały wyciągnięte wnioski oraz możliwe ulepszenia projektu pokazane w rozdziale 6, po czym zostało sformułowane podsumowanie przeprowadzonego projektu, pokazujące trudności na jakie się napotkano w trakcie pracy. Problemy te należą do dziedziny nawigacji inercyjnej i nie istnieje uniwersalne rozwiązanie które można by zastosować. Celem pracy jest przeprowadzanie próby stworzenia własnego rozwiązania realizującego zagadnienia związane z tematyką kontrolerów - w tym przypadku prezentowanego jako rękawica.

Słowa kluczowe: Wirtualna rzeczywistość, Kontroler, Rękawica, Nawigacja inercyjna, Akcelerometr, Żydroskop, Czujnik wygięcia

Abstract

The topic of this thesis is the "Development of virtual reality based on the example of a glove controller". This thesis presents (the concept of) virtual reality as well as ways of controlling and interacting within its universe. The topic of controller, especially in the context of glove controllers, was addressed and its functionality and usability cases were shown. Chapter 3 highlights the challenges encountered by the controller's designers. Next, the implementation of the glove controller is presented, with a focus on delivering basic functionality. Data is sent from the glove in a wireless manner. The receiver of this data, an Android application, is described in chapter 5. The mentioned chapter discusses topics of Bluetooth Low energy implementation, the orientation of the device, its displacement and the model animation based on data received from the glove. On the basis of the experiences acquired while working on this project, flaws and possible improvements were outlined in chapter 6, followed by a summary and multiple conclusions formed in relation to the project, detailing the variety of obstacles faced while creating the glove proposal. The issues encountered relate to the immersive navigation realm, where no universal solution could be applied. The goal of this paper is to attempt creating a homemade glove controller, which fulfils all of the basic requirements for a functional controller device.

Key words: Virtual reality, Controller, Glove, Immersive navigation, Accelerometer, Gyroscope, Bend sensors

Rozdział 1

Wstęp

Poniższa praca przedstawia ewolucję rozwiązań oraz problemów z którymi mierzą się współczesne firmy, w celu stworzenia nowych, wydajnych a także intuicyjnych rozwiązań, w tym kontrolerów dla użytkowników świata wirtualnego. Kontrolery w szczególności napotykają wiele problemów ze względu na przeszkody wynikające z natury nawigacji inercyjnej, czyli określenia położenia i orientacji zazwyczaj w środowisku bez punktu odniesienia. Problemy te zostaną opisane w dalszej części tej pracy, a także metody jakie zostały zastosowane aby rozwiązać wspomniane problemy przez liderów branży.

Z biegiem czasu dąży się do jak najbardziej realistycznych rozwiązań, tak aby w pełni oddać naturę świata rzeczywistego w środowisku programowalnym. W tym celu powstały pierwsze komercyjne modele rękawic, pozwalających na ruch w przestrzeni świata wirtualnego, co dało możliwość naturalnego interfejsu pomiędzy użytkownikiem a światem w którym się znajduje. Naturalność tego rozwiązania, oraz potencjał jaki w sobie kryje, sprawiło że wiele firm zaczęło tworzyć własne rozwiązania tego produktu, dodając unikalne właściwości takie jak wibracje symulujące dotyk, punktową imitację nacisku czy też szkielet blokujący palce dłoni przed zgięciem. Rozwój tego rynku oraz chęć inwestycji w tego typu projekty ze strony dużych firm i organizacji pokazuje zapotrzebowanie na

tego typu rozwiązanie. Potencjał który się w tym kryje, oraz brak uniwersalnego rozwiązania jest głównym powodem dla powstania tej pracy.

Celem tej pracy jest stworzenie własnego, uproszczonego kontrolera-rękawicy, w celu lepszego poznania znanych problemów dla tego typu urządzeń oraz właściwej implementacji ich rozwiązań. Kontroler ten powinien być prosty w wykonaniu, tak aby można było go odtworzyć w domowych warunkach, przy jednoczesnym spełnianiu podstawowych wymagań, które są od niego wymagane takie jak określenie orientacji, położenia oraz momentu zgięcia poszczególnych palców. Ostatecznie dążono do stworzenia aplikacji na system Android, która będzie łączyć się z kontrolerem oraz prezentować przesłane z rękawicy dane. Na podstawie tych informacji zostanie wyświetlony w czasie rzeczywistym model dłoni na ekranie smart-fona, który będzie reagował na zmiany w przesyłanych danych.

Rozdział 2

Zagadnienia

Zanim zostanie przedstawione wykonanie projektu, należy zapoznać się z podstawowymi zagadnieniami związanymi ze światem wirtualnej technologii oraz sposobem w jaki świat ten wchodzi w interakcję z użytkownikiem. W tym celu zostanie pokazane czym jest wirtualna rzeczywistość, w jaki sposób producenci zestawów VR (z ang. Virtual Reality) podchodzą do tematu kontrolerów oraz jak narodził się pomysł na rękawice-kontroler, która pewnego dnia może zastąpić wykorzystywane do tej pory rozwiązania.

2.1 Czym jest wirtualna rzeczywistość?

Przede wszystkim przed rozpoczęciem pracy nad kontrolerem należy zrozumieć czym jest wirtualna rzeczywistość oraz jak wygląda interakcja pomiędzy światem rzeczywistym a wirtualnym. Rzeczywistość wirtualna jest to wygenerowany komputerowo trójwymiarowy obraz przedstawiający pewną scenę w której mogą być umiejscowione różne obiekty imitujące przedmioty, ludzi, zdarzenia oraz interakcje pomiędzy nimi. W zależności od sposobu stworzenia takiej przestrzeni użytkownik może się w tej przestrzeni poruszać, bądź zostać ustawiony w jednym punkcie. Scena to może zawierać elementy zarówno prawdziwego jak i fikcyjnego świata. Niezależnie jednak od sposobu prezentacji, założeniem świata

wirtualnego jest stworzenie naturalnych i realnych doznań. Realność jest odbierana przez ludzki mózg jako seria sygnałów pochodzących z różnych zmysłów, dzięki którym możemy określić czy środowisko w jakim się znajdujemy jest tym "żalnym", które znamy z życia codziennego. W związku z tym aby w pełni oszuścić ludzki mózg, nawet najmniejsze detale muszą odwzorowywać dokładnie elementy świata rzeczywistego, które jako cywilizacja staramy się osiągnąć poprzez wprowadzanie ciągłych udoskonaleń. Najważniejszym elementem prezentowania wirtualnej rzeczywistości było wprowadzenie okularów VR, które całkowicie blokowały świat zewnętrzny, pozwalając jedynie na rozglądarkę się w świecie stworzonym przez twórców. Typowym elementem jest możliwość rozglądarki się w świecie wirtualnym dzięki poruszaniu głową, dzięki czemu obraz generowany i wyświetlany w okularach niejako pokazuje świat wirtualny w ten sam sposób jak odkrywamy świat rzeczywisty. Ważnym elementem w tworzeniu światów wirtualnych jest poznanie fizjonomii człowieka. Obraz który widzi człowiek jest jedynie częścią otoczenia - ważnym elementem jest również zmysł słuchu który potwierdza widziany przez użytkownika obraz w postaci sygnałów, których osoba się spodziewa widząc dany przedmiot. Jeżeli sygnały są sprzeczne, długotrwałe użytkowanie takiego sprzętu potrafi doprowadzić do znanej wielu osobom choroby lokomocyjnej. Mając do dyspozycji obraz oraz dźwięk twórcy mogą tworzyć różnego rodzaju pokazy oraz filmy 360°, dzięki czemu odbiorcy otrzymują informacje w nowy ciekawy sposób. Oczywiście w pewnym momencie prezentowanie obrazu nie było wystarczające - w celu podniesienia poziomu realności widzianego obrazu, użytkownik powinien móc wchodzić w interakcję z obiektami wokół niego się znajdującymi. Najprostszym rozwiązaniem jest dodanie przycisku oraz celownika na środku ekranu. Dzięki temu gdy wycelowano w dany obiekt, przycisk służył niejako mysz komputera pozwalając na wywołanie zaprogramowanej akcji obiektu. Rozwiązanie to w bardziej skomplikowanych przedsięwzięciach

jest jednak nie wystarczające w związku z czym zostały stworzone kontrolery z których pomocą można było wchodzić w interakcję w świecie wirtualnym [28].

2.2 Działanie kontrolerów w VR

Pomiędzy kontrolerami typu gamepad, używanych do gier wideo oraz tymi używanymi do interakcji z obiektami wirtualnej rzeczywistości jest kilka różnic zarówno z wewnętrz jak i zewnętrz, jednak podstawowe założenie obu jest takie samo, ponieważ muszą one spełnić te same funkcje niezależnie od sposobu prezentacji obrazu użytkownikowi. Często spotykaną praktyką jest umiejscowienie na kontrolerze pewnego rodzaju joysticka pozwalającego na ruch oraz przyciski dzięki którym można wykonywać akcje zależne od środowiska w jakim się znajduje użytkownik. Oczywiście sposób obsługi nie jest taki sam. Wyświetlając obraz na ekranie kontroler zazwyczaj jest obsługiwany oburącz w związku z czym w ten sposób też są one projektowane. Stąd pojawiają się pierwsze różnice wizualne. Kontrolery w świecie VR przede wszystkim są projektowane tak aby były obsługiwane jedną ręką. Oprócz tego zazwyczaj posiadają pewnego rodzaju system śledzenia wewnętrzny, zewnętrzny bądź łączący oba, aby można było określi pozycję kontrolera w przestrzeni, dzięki czemu pozycję kontrolerów można odwzorować w świecie wirtualnym np. jako dlonie. Sprawia to jednak że posiadają one często dodatkowe, nietypowo wyglądające elementy. Śledzenie kontrolerów w przestrzeni nie należy do problemów dających się łatwo rozwiązać w związku z czym wiele firm zdecydowało się na różne sposoby rozwiązania tego zadania. Pomimo braku śledzenia, gamepady takich firm jak Sony czy Microsoft współpracują z technologiami VR na konsole przez nich produkowane, jednak zagłębienie się nie jest tak łatwe jak w przypadku kontrolerów do tego przeznaczonych. Na zdjęciu 2.1 znajdują się przykładowe kontrolery dwóch zna-

nych producentów na rynku - HTC Vive oraz Oculus Rift. Zapewniają one system



(a) Kontroler firmy Oculus

Źródło: <https://www.oculus.com/rift-s/accessories/>

(b) Kontroler firmy Vive

Źródło: <https://www.x-kom.pl/p/381876-akcesorium-do-gli-vr-htc-vive-controller.html>

Rysunek 2.1: Kontrolery VR

śledzenia dloni poprzez specjalne systemy - w przypadku Oculusa jest to system ukrytych diod, które emitują podczerwone światło, nie widoczne dla ludzkiego oka. Sensory które obserwują użytkownika analizują układ diod, dzięki czemu mogą przeanalizować pozycję. Analiza obrazu jest usprawniona poprzez przechowywania danych dotyczących położenia urządzenia na podstawie odczytów z żyroskopu, natomiast kierunek przemieszczenia na podstawie akcelerometru. W przypadku HTC Vive problem położenia rozwiązany jest bez przesyłania żadnych danych do obliczenia. W przeciwnieństwie do latarni, które okresowo wysyłają wiązki światła podczerwonego, dzięki czemu na podstawie okresu pomiędzy odczytami z latarni, kontrolery są w stanie określić swoje położenie w przestrzeni. Problem rotacji jest rozwiązywany na podstawie żyroskopu tak jak jest to wykonywane w przypadku innych urządzeń. Oprócz tego nowsze urządzenia odchodzą od dodatkowych systemów śledzenia

na rzecz wbudowanych kamer w zestawy do VR, ze względu na niechęć użytkowników do dodatkowych urządzeń które należy montować w celu poprawnego działania. [16] [30].

2.3 Wykorzystanie dloni jako kontrolera

Jeżeli pomyśli się świecie wirtualnym oraz jego celu, czyli stworzeniu realnych i naturalnych doznań, naturalnym się wydaje używanie dloni w celu interakcji ze sceną - bez konieczności trzymania dodatkowych kontrolerów. I rzeczywiście podejście to było też zastosowane w praktyce. Użycie dloni, a konkretnie rękawicy na dloniach jako kontrolera w świecie wirtualnym po raz pierwszy wykorzystano w 1977 roku przez Dana Sandina. Dzięki wykorzystaniu światłoczułych czujników na każdym z palców, możliwe było otrzymanie pomiarów zgięcia palców, które głównie służyło do regulacji suwaków [31]. Współczesne rękawice znacznie się rozwinęły od tamtej pory, implementując różnorodność rozwiązań. W celu ustalenia orientacji urządzenia korzystają przynajmniej z jednej jednostki do pomiaru inercji, stosując dokładne filtry w celu jak największej dokładności. Wiele urządzeń korzysta z wielu takich jednostek pomiarowych, umieszczając po dwie bądź trzy na każdym palcu w celu dokładnego mierzenia pozycji palców. Oprócz tego urządzenia stosują systemy vibracji, zwiększając tym samym jak realistyczne jest doznanie, w szczególności gdy dochodzi do interakcji z przedmiotami. Rękawice są dobrze znane w branży wirtualnej rzeczywistości. Rozwiązania te jednak pozostają drogie i nie tak dokładne jak prezentowane w sekcji 2.2 przykładowe kontrolery, w szczególności braku uniwersalnego braku rozwiązania dla wszystkich użytkowników - nie każdy w końcu ma dłoń tej samej wielkości. Problemy takie jak ten próbowała rozwiązać firma *Plexus*, jednak produkt ten nie jest jeszcze dostępny na rynku [22]. Z pewnością można się spodziewać rozwoju tego i

wielu innych produktów, i jak to bywa z wieloma początkowo drogimi technologiami - spadku cen gdy zainteresowanie produktem stanie się większe. Na chwilę obecną widać duże zainteresowanie tego rodzaju rozwiązaniami i wiele osób już wykorzystuje tego rodzaju produkty w biznesie. W rozwój technologi inwestują takie firmy jak: Toyota, Netflix, Audi czy NASA. Biorąc to pod uwagę nie można zaniedbać potencjału jaki się kryje w tym segmencie rynku wirtualnej rzeczywistości [18].

Rozdział 3

Komponenty rękawicy-kontrolera

Wspomniane były do tej pory projekty komercyjnych rękawic-kontrolerów, jednak w celu dokładnego działania, wykorzystują one złożone techniki w celu osiągnięcia jak najlepszych rezultatów, jak i również dodatkowe moduły pozwalające na rozwiązanie problemów w bardziej wyszukany sposób - często poprzez analizę otoczenia, bądź gdy otoczenie analizuje gdzie znajduje się kontroler. W przypadku podstawowej wersji nie użyto żadnych dodatkowych systemów a jedynym zamiarem jest wykorzystanie podstawowych, łatwo dostępnych bądź możliwych do skonstruowania elementów w celu rozwiązania problemów nawigacji inercyjnej. W związku z tym najpierw zdefiniowano czym jest nawigacja inercyjna. Nawigacja inercyjna, znana również pod akronimem INS(z ang. Inertial Navigation System), jest to dziedzina problemowa która zawiera metody określania pozycji przy użyciu sensorów wykorzystujących zasady działania fizyki. Sensory te to akcelerometr oraz żyroskop. Akcelerometr dzięki wykorzystaniu drugiej zasady dynamiki Newtona, pozwala określić oddziaływanie siły na urządzenie na podstawie sił jakie oddziałują na niego samego. Żyroskop natomiast uzyskuje pomiar działania prędkości kątowych dzięki czemu jest w stanie określić orientację urządzenia. Dzięki wykorzystaniu tych sensorów, w teorii jesteśmy w stanie określić dokładne położenie i orientację bez konieczności użycia zewnętrznego źródła po-

miarów. Warto zauważyć że jest tak tylko w teorii ponieważ w praktyce sensory te podlegają wielu błędom przez które niemożliwym jest określenie położenia w przestrzeni z wystarczającą dokładnością aby oszukać zmysły człowieka w szczególności w systemach wirtualnej rzeczywistości [29].

3.1 IMU - Inercyjna jednostka pomiarowa

Aby można było stosować pomiar w ramach nawigacji inercyjnej zostały stworzone inercyjne jednostki pomiarowe, znane jako IMU(z ang. Inertial Measurement Unit), które w podstawowej wersji występują w postaci połączenia akcelerometru oraz żyroskopu. W wersji bardziej zaawansowanej dołączony jest również trzyosiowy magnetometr i w popularnych produktach rękawic to właśnie ten rodzaj IMU jest najczęściej spotykany. W celu lepszego zrozumienia działania tej jednostki zostaną przedstawione kolejno sensory wchodzące w jej skład oraz zostanie wyjaśnione pojęcie wieloosiowego pomiaru, które jest bezpośrednio powiązane z określeniem stopni swobody [1].

3.1.1 Sensory

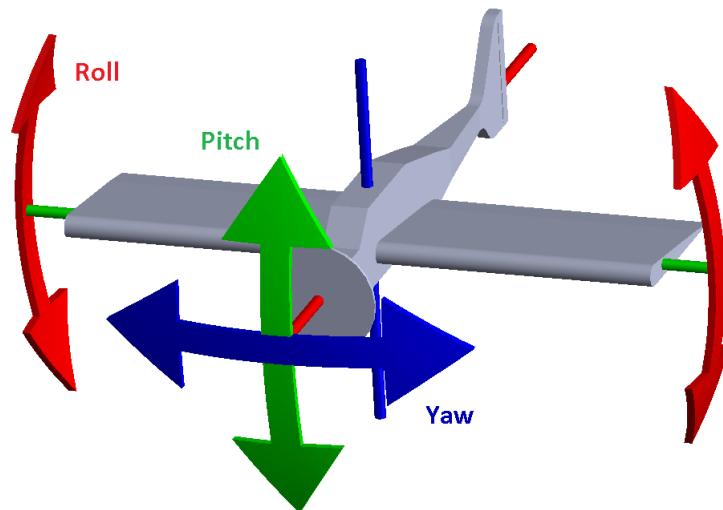
Akcelerometr jest prostym sensorem którego budowa opiera się o zamocowanie małej masy na sprężynach które ją utrzymują. W zależności od tego jak zadziała siła na akcelerometr sprężyna wyciągnie się bądź skurczy, wprawiając ciało w ruch. Masa ta jest częścią układu kondensatora pomiarowego, co oznacza że w wyniku poruszania zmienia się jego pojemność - czyli napięcie wyjściowe. Napięcie to dalej jest przekazywane do mikroprocesora bądź przetwornika analogowo-cyfrowego, a wynik podawany jest w $\frac{m}{s^2}$. Sensor ten głównie jest wykorzystywany do pomiaru przemieszczenia. Drugim równie ważnym sensorem którego celem jest pomiar prędkości kątowej jest żyroskop. Żyroskop wykorzystuje do działania zjawisko znane jako efekt Coriolisa. Polega ono na wytwarzaniu siły

prostopadłej do masy, umieszczonej na talerzu żyroskopu, która przemieszcza się względem zadanej osi. Przemieszczanie się tej masy jest mierzone przy użyciu kondensatorów. Oznacza to że siła ta mierzona jest tylko gdy sensor podlega rotacji, wprawiając masę znajdująca się w środku w ruch - w przypadku gdy sensor jest w pozycji stacjonarnej, niezależnie od aktualnej rotacji, odczyty wskazują zero na wszystkich osiach pomiaru. Jednostką pomiaru są $\frac{rad}{s}$, co oznacza że uzyskanie orientacji w postaci kąta obrotu jest możliwe poprzez całkowanie wartości uzyskanych z żyroskopu. Ostatnim czujnikiem wykorzystywanym w IMU jest magnetometr, dzięki któremu uzyskiwana jest wartości pola magnetycznego oraz jego kierunek. Istnieje kilka sposobów na odczyt tych danych w związku z czym metody budowy magnetometru nie zostaną w tej pracy opisane. Magnetometru używa się w celu określenia kierunku magnetycznego bieguna północnego ziemi. Aby osiągnąć geograficzny kierunek bieguna północnego należy do tej wartości dodać deklinację geograficzną która różni się w zależności od miejsca na ziemi w którym znajduje się magnetometr. Magnetometr zwraca wartości podane w mikro-Teslach zapisywanych jako μT . Tak jak pozostałe sensory również ten jest wrażliwy na zaburzenia odczytów. W przypadku magnetometru są to pola magnetyczne wokół których znajduje się sensor [1].

3.1.2 Stopnie swobody

Do tej pory powiedziano o działaniu sensorów przy których stwierdzono że działają one wokół pewnych osi. Świat trójwymiarowy jest reprezentowany jako układ trzech osi ustawionych do siebie prostopadle. Jeżeli sensor nazywany jest trójosiowym, oznacza to że dokonuje on pomiaru we wszystkich kierunkach dzięki czemu jesteśmy w stanie określić dane w przestrzeni trójwymiarowej. W zależności od sensora IMU, dane te pozwalają określić wartości takie jak położenie, orientację w przestrzeni względem punktu początkowego, a gdy ten jest znany

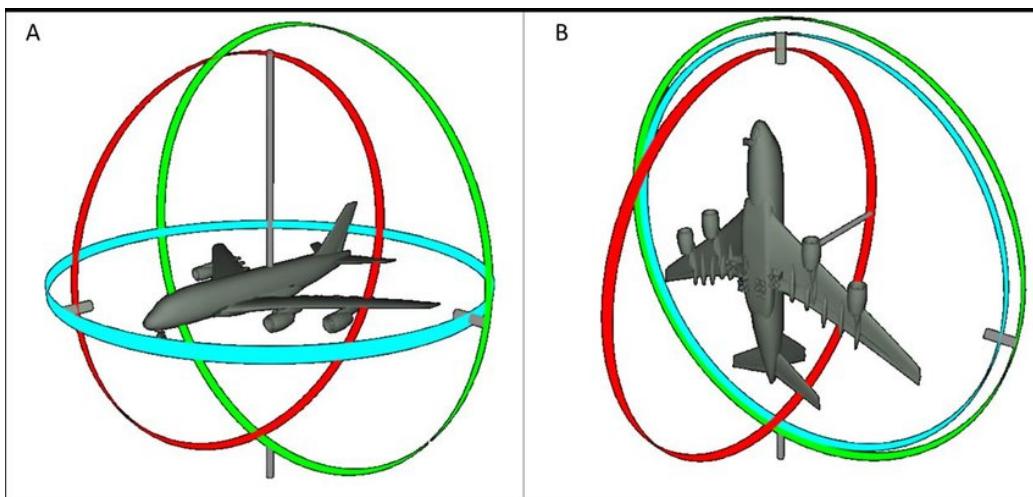
orientację bezwzględną, oraz kierunek względem biegunów ziemi. Stopień swobody oznacza możliwość pomiaru danego parametru wokół zadanej osi. W ten oto sposób jeżeli mamy do czynienia z 3-osiowym akcelerometrem oraz 3-osiowym żyroskopem mówimy o 6 stopniach swobody, oznaczanych jako 6DoF (z ang. 6 Degrees of Freedom), czyli położenie względem punktu zero, oznaczonego jako dystans na osi X,Y oraz Z, a także orientację wokół tych osi, opisywanych jako wartość przechylenia, nachylenia oraz zbaczania z trasy. W literaturze jednak poważnie stosowane są angielskie określenia roll, pitch oraz yaw. Roll jest określany jako obrót wokół osi poziomej wzdłużnej, pitch jest definiowany jako obrót wokół osi poprzecznej modelu, natomiast yaw jako obrót wokół osi pionowej. Rysunek 3.1 pokazuje umiejscowienie modelu samolotu w punkcie zero układu współrzędnych oraz rotację wokół tych osi reprezentując 6DoF. Ten rodzaj usta-



Rysunek 3.1: Model prezentujący sześć stopni swobody
Źródło: <http://www.rc.info.pl/slownik/roll-pitch-yaw-katy-rpy/>

wienie obrazujący kierunek oraz rotacje, w przestrzeni trójwymiarowej na podstawie trójosiowego układu współrzędnych jest nazywany katami Tait–Bryan'a. Kąty te jednak mają jedną poważną wadę. W skutek obrotu wokół osi, jest moż-

liwa utrata jednego stopnia swobody. Dzieje się to z powodu rotacji w której dwie z trzech osi żyroskopu zostaną ustawione do siebie równolegle. Spowodowane jest to tym że obrót jest odwzorowywany z danych żyroskopu według jednej osi jednocześnie, co sprawia że kolejność wykonywania rotacji ma znaczenie. Kolejność od pierwszej do ostatniej jest określana jako obrót zewnętrzny, środkowy oraz wewnętrzny. W przypadku kolejności obrotu wokół osi Y, następnie X a ostatecznie Z, jeżeli dokonamy obrotu wokół osi X o 90° , sprawi to że osią zewnętrzna Y a także wewnętrzna Z ustawią się w jednej płaszczyźnie, w związku z czym stracimy możliwość obrotu względem jednej osi. Problem ten obrazuje rysunek 3.2. Jeżeli w jednostce używany jest również magnetometr, określa się takie IMU jako 9DoF. Podsumowując - w zależności od użytych sensorów, oraz osi wokół których zostały ustawione sensory, mówimy o odpowiedniej ilości stopni swobody. Oznacza to że jeżeli użyjemy trzech akcelerometrów i żyroskopów, jednak wszystkie będą ustawione wokół tej samej osi mówimy o 2DoF, jeden dla przemieszczenia, drugi dla rotacji wokół tej osi [23] [8].



Rysunek 3.2: Problem blokady gimbala (Przed i po obrocie)

Źródło: https://www.researchgate.net/figure/Gimbal-lock-problem-for-Euler-angles-A-no-gimbal-lock-B-yaw-and-roll-angles-are_fig14_332394380

3.2 Śledzenie położenia palców

Aby mówić o kontrolerze w postaci rękawicy, wymaganym elementem jest możliwość śledzenia położenia palców dloni. Rozwiązania tego problemu zazwyczaj widuje się w postaci jednego z dwóch sposobów. Pierwszym sposobem jest wykorzystanie wspomnianych IMU i umieszczenie ich na dloni w cel uzyskania pozycji całej ręki oraz na palcach w celu ich śledzenia. W takim wypadku, w spotkanych rozwiązańach użyte zostały dwa bądź trzy IMU na każdym z palców. Następnie dane ze wszystkich sensorów, ze wszystkich IMU zostają złożone w całość w celu wygenerowania pozycji każdego z czujników na ekranie użytkownika. Rozwiązanie to jest znacznie bardziej skomplikowane lecz oprócz zginania palców pozwala na określenie ich odwodzenia co zapewnia dużo bardziej realne wrażenia. Rozwiązanie to jednak nie jest tanie, a także sposób integrowania danych jest bardziej skomplikowany. Alternatywą tego rozwiązania jest wykorzystanie czujników zgięcia. Czujnik taki umiejscowiony na palcu pozwala na określenie stopnia wygięcia. Rozwiązanie to jest proste w implementacji, jednak czujniki pozwalające na dokładne pomiary również nie są tanie [9] [18].

3.3 Łączność przy użyciu Bluetooth niskiego poboru mocy

Ostatnim elementem ważnym dla rękawicy jest możliwość przesyłania danych. Dane można przesyłać w tradycyjny sposób przewodem przy użyciu odpowiednich złącz, jednak w celu większego komfortu użytkowania i spełnienia realizmu wirtualnej rzeczywistości połączenia bezprzewodowe są tymi które dostarczają lepsze wrażenia. Wśród połączeń bezprzewodowych wykorzystywane jest połączenie przy użyciu Bluetooth a w szczególności BLE (z ang. Bluetooth Low Energy). Rozwiązanie to kosztem transferu osiągnęło długi czas pracy na mini-

malnym zasilaniu, przy czym w wersji 5.0 pozwala również na współpracę z wieloma urządzeniami jednocześnie, co może okazać się przydatne gdy korzystamy z dwóch rękawic jednocześnie.

Gdy mowa o połączeniu BLE warto znać kilka podstawowych terminów. Przede wszystkim trzeba wiedzieć czym jest GATT (z ang. Generic Attribute Profile). Jest to struktura pozwalająca na wykonywania podstawowych operacji przy użyciu Bluetooth, takich jak przesyłanie i przechowywanie danych. W ramach tej struktury mieszą się dodatkowe moduły. Przede wszystkim należy wyróżnić serwis - jest to moduł który zawiera w sobie kolekcję cech, oraz związki pomiędzy serwisami, które zawierają dane z urządzenia bądź jego części, na którym są uruchamiane. Oznacza to że jest to źródło danych do którego się podłączamy poprzez adapter bluetooth. Serwis jednak nie jest odpowiedzialny za bezpośrednie przechowywanie danych - jedynie za listę cech. Cecha jest to zdefiniowany atrybut który przechowuje dane zadeklarowanego typu. W wykorzystywanym urządzeniu zazwyczaj istnieje wiele cech które są przechowywane w serwisie, a ich wartości są pobierane w zależności od sposobu deklaracji cechy. Mogą one przechowywać wartości które są wysyłane do innego urządzenia, bądź przechowywać wartości przysłane przez adapter, co może zmienić sposób działania urządzenia na którym zadeklarowano GATT. Ostatnią ważny modułem jest deskryptor, którego zadaniem jest opisanie cechy do której jest przypisany. W szczególności jest to ważne gdy mamy do czynienia z wieloma cechami, bądź nawet serwisami. Deskryptory są zdefiniowane osobno w celu łatwego odczytu przez ludzi, natomiast podczas deklaracji cechy, jej niepowtarzalnym parametrem jest UUID (z ang. Universally Unique IDentifier). Jest to uniwersalny unikalny identyfikator, który jest wyrażony w postaci 128 bitowych ciągów znaków, w których skład mogą wchodzić zarówno cyfry, liczby jak i znaki specjalne. Istnieje więcej niż jedna wersja generatora, w zależności od tego na czym generator jest oparty [7].

Rozdział 4

Projekt rękawicy

Istotą poniższego rozdziału jest pokazanie użytych w projekcie podzespołów i technologii oraz lepsze zrozumienie powodów dla których to właśnie te produkty zostały wybrane. Po zapoznaniu się z motywem, zostanie szczegółowo opisana specyfikacja tych produktów a także sposób ich poskładania w spójną całość, co sprawiło pewne problemy względem oryginalnego szkicu - próby oraz efekty rozwiązywania tych problemów również zostaną opisane w tym rozdziale. Po nakreśleniu podstawowych założeń projektu zostanie zaprezentowana finałowa wersja, a także szczegółowo zostanie omówiony kod rękawicy-kontrolera, który jest obsługiwany przez mikrokontroler i stanowi najważniejszą część tego projektu.

4.1 Przegląd podzespołów użytych w projekcie

Jak dowiedzieliśmy się z rozdziału 3 dotyczącego podstawowych komponentów rękawic, kluczowym dla powodzenia projektu jest ustalenie następujących pozycji:

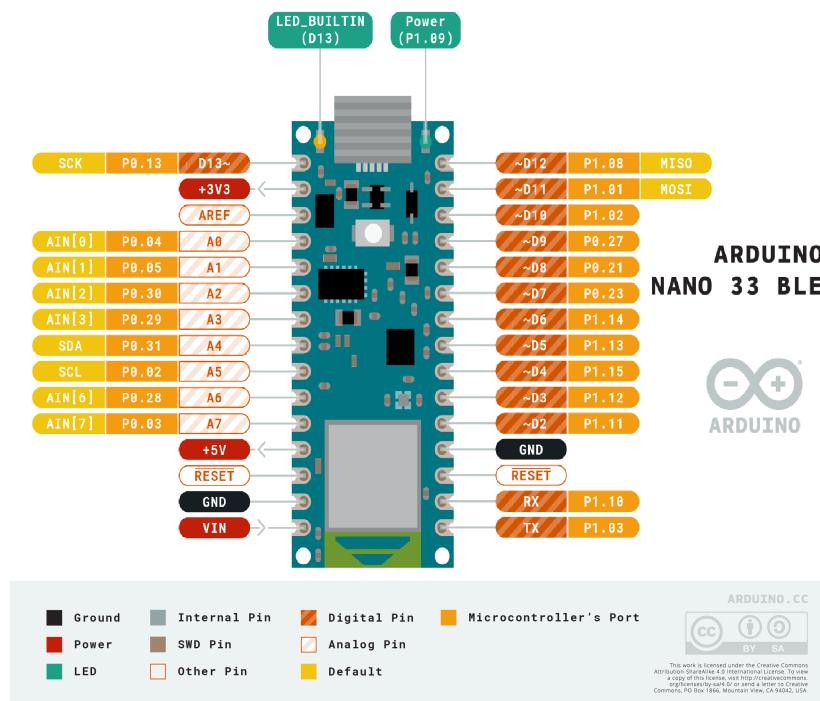
- orientacji dloni względem punktu początkowego
- położenia względem punktu początkowego
- moment i stopień zgięcia palców

W tym celu należy zebrać informacje z czujników, a następnie wszystkie te informacje należy przesyłać do pożądanego urządzenia. Elementem które pozwala to osiągnąć w tym projekcie jest mikrokontroler Arduino nano 33 BLE, który odpowiada za dostarczenie informacji z żyroskopu, akcelerometru a także czujników wygięcia. Zasady działania pierwszych dwóch zostały opisane z podrozdziałem 3.1.1. Natomiast w poniższych podrozdziałach zostanie opisane rozwiązanie zastosowane do odczytu położenia palców, zasada działania przy wykorzystaniu rezystorów oraz sposób połączenia wszystkich wspomnianych elementów w jeden finałowy kontroler.

4.1.1 Mikrokontroler

Jak przed chwilą wspomniano, w projekcie wykorzystywana jest płytka od Arduino, która nosi nazwę Nano 33 BLE. Jest to małych rozmiarów płytka o wymiarach 45 x 18 mm, pozwalająca na wysoką wydajność przy jednoczesnym małym poborze prądu, co zapewnia użyty mikrokontroler nRF52480 o taktowaniu 64 MHz. Do dyspozycji mamy również pamięć RAM o pojemności 256 kB oraz pamięć Flash o pojemności 1 MB. IMU które zostało zamontowane na płytce to LSM9DS1, które obsługuje akcelerometr, żyroskop oraz magnetometr w trzech osiach. Więcej informacji na temat IMU jest przedstawione w części 4.3. Warto na wstępie zauważyć że Nano 33 BLE pracuje domyślnie wyłącznie z napięciem 3,3 V, w związku z czym nie należy podłączać bezpośrednio zasilania o większym napięciu. W celu podłączenia zasilania 5 V należy zlutować zworkę znajdującą się pomiędzy pinami RDT oraz A7 - temat ten nie zostaje jednak poruszony w tej pracy, ponieważ na potrzeby projektu używane jest zasilanie poprzez złącze micro USB które to również jest obsługiwane. Płytki ta posiada wiele użytecznych sensorów, jednak na potrzeby tej pracy została wybrana z powodu wbudowanej inercyjnej jednostki pomiarowej, dzięki czemu można było uprościć konstrukcję

oraz zmniejszyć ilość połączeń na rękawicy, wbudowanego modułu Bluetooth - a w tym przypadku modułu Bluetooth Low Energy obsługiwany w standardzie 5.0, a to wszystko w przystępnej cenie co również było jednym z kryteriów przy tworzeniu tego projektu. Płytkę w momencie tworzenia tej pracy można kupić za 119 zł. Wartym uwagi jest fakt możliwości zakupu płytki bez wyprowadzonych złącz, co w przypadku opisywanego projektu pozwoli na zmniejszenie wymiarów oraz większą swobodę montażu [8].



Rysunek 4.1: Opis wejść oraz wyjść Arduino Nano 33 BLE

Źródło: https://content.arduino.cc/assets/Pinout-NANOble_latest.pdf

Z najważniejszych elementów układu płytki należy wiedzieć że posiada ona dwie diody po dwóch stronach portu micro USB - zielona indykuje podłączone zasilanie, natomiast pomarańczowa zaczyna mrugać gdy jest przesyłany kod do mikrokontrolera. Oprócz tego do dyspozycji są dwa piny wyjściowe zasilające o

napięciu 3,3 V oraz 5 V, jeden pin zasilający wejściowy, którego ograniczenia zostały wspomniane w poprzednim paragrafie, a także dwa piny uziemiające, po jednym z każdej strony płytki. Posiada ona piny zarówno analogowe jak i cyfrowe, jednak na potrzeby tego projektu zostały użyte jedynie piny analogowe, których do dyspozycji jest aż osiem umiejscowionych po jednej stronie, przy czym warto zwrócić uwagę że piny A4 oraz A5 używane są jako magistrala I2C w związkach z czym zalecane jest nie stosowanie tych wejść analogowych. Szczegółowy opis wejść/wyjść płytki przedstawiony jest na grafice 4.1. W tym projekcie wykorzystano zasilanie poprzez złącze micro USB, wyjście o napięciu 3,3 V w celu uzyskania odczytów z czujników wygięcia na pinach analogowych A0, A1, A3, A6 oraz A7, o których zostanie więcej powiedziane w punkcie 4.2, a także uziemienie znajdujące się po tej samej stronie płytki.

4.1.2 Czujnik wygięcia

Kluczowym dla działania kontrolera jest możliwość określenia pozycji palców względem dłoni. Ma to wiele zastosowań zarówno wizualnych jak i praktycznych. Ważne jest aby odwzorować świat rzeczywisty tak dokładnie jak to możliwe - im lepsze odwzorowanie tym bardziej zmysły użytkownika zostaną oszukane, podwyższając komfort użytkowania technologii wirtualnej rzeczywistości. Za stroną praktyczną natomiast przemawia możliwość śledzenia palców w celu dokładnego ich użycia w stworzonym środowisku np. do ściskania i podnoszenia obiektów czy też korzystania z klawiatury wirtualnej. Jak wspomniano w rozdziale 3 dotyczącym komponentów komercyjnych rękawic - wśród znanych producentów na rynku, decyduje się na użycie wielu inercyjnych jednostek pomiarowych, na podstawie których są w stanie dokładnie określić położenia każdej części palca, bądź też nie aż tak popularne rozwiązanie, które wykorzystujące specjalistyczne sensory służące do pomiaru stopnia wygięcia czujnika względem

pozycji prostej. Czujnik ten po podłączeniu do prądu zwiększa swój opór wraz ze zwiększym stopniem odchylenia. Oba te rozwiązania pomimo wysokiej dokładności pomiarów nie są rozwiązaniami tanimi. W związku z tym na potrzeby stworzenia taniego kontrolera, należało znaleźć rozwiązanie bardziej przystępna a jednocześnie pozwalające na osiągnięcie tego samego celu.

Aby osiągnąć postawione założenia zostały skonstruowane czujniki wygięcia w domowych warunkach. Rozwiązanie to jest często używane do osiągnięcia pomiaru stopnia wygięcia bez konieczności wydawania ponad 100 zł na jeden czujnik [9]. Jest ono stosunkowo proste w założeniach i wymaga jedynie dwóch klużowych elementów. Tkaniny przewodzącej o specjalnych właściwościach oraz dwóch przewodników po obu stronach materiału. Z jednej strony zostanie podłączone napięcie z drugiej natomiast uziemienie. Ważnym jest aby połączenia te się ze sobą nie stykały w żadnym punkcie a jedynie zostały nałożone na siebie, z tkaniną ściśniętą pomiędzy nimi - dzięki temu mamy pewność że odczyty które otrzymamy będą prawidłowe. Pozostaje odpowiedzieć na pytanie jakiego rodzaju materiał należy wykorzystać. Na rynku znajdziemy wiele rodzajów materiałów które zmieniają swój opór w zależności od spełnienia takich kryteriów jak nacisk, temperatura, rozciąganie czy też właśnie zgięcie materiału. Pomimo próby uzyskania materiału który zmienia swój opór w zależności od rozciągania, co pozwoliłoby na skonstruowanie części na palce rękawiczki z tego materiału, zapewniając dokładniejszy i bardziej estetyczny efekt końcowy, w momencie projektowania rękawicy był on jedynie możliwy do sprowadzenia ze stanów, co nie było najtańszym rozwiązaniem. W związku z tym zdecydowano się na użycie folii Velostat która jest czuła na nacisk or zginanie [5]. W roli przewodnika wybrano nić przewodzącą, która zapewniła potrzebną elastyczność oraz możliwość przy mocowania poszczególnych elementów przy jednoczesnym zapewnieniu funkcjonalności. Elementy te zostały sklejone na kawałku taśmy samoprzylepnej oraz do-

datkowo sklejone przy brzegu aby nie wyśliznęła się w trakcie korzystania z czujnika. Efekt końcowy jest widoczny na zdjęciu 4.2. W celu otrzymania pomiarów wszystkich palców zostało wykonanych pięć takich sensorów, o szerokości 15 mm; dwa o długości 8 cm, dwa o długości 10 cm a także jeden 11 cm, w celu jak najlepszego dopasowania względem miejsca na palce na zakupionej rękawicy do której sensory zostaną przymocowane, co można zobaczyć na zdjęciu 4.6.



Rysunek 4.2: Sensor wygięcia własnego wykonania

4.1.3 Rezystor

Rezystor, potocznie zwany opornikiem, jest to prosty element elektroniczny, posiadający jedynie wyjścia z dwóch stron elementu łączącego. Element ten tworzy opór, powodując ograniczenie przepływającego przez niego prądu gdy jest włączony do obwodu szeregowo. Opór ten jest mierzony w omach. Istotną informacją jest fakt że nadmiar prądu jest zamieniany przez opornik na energię cieplną, a także brak zdefiniowanego kierunku - co oznacza że działa on niezależnie od sposoby zanotowania go w układzie.

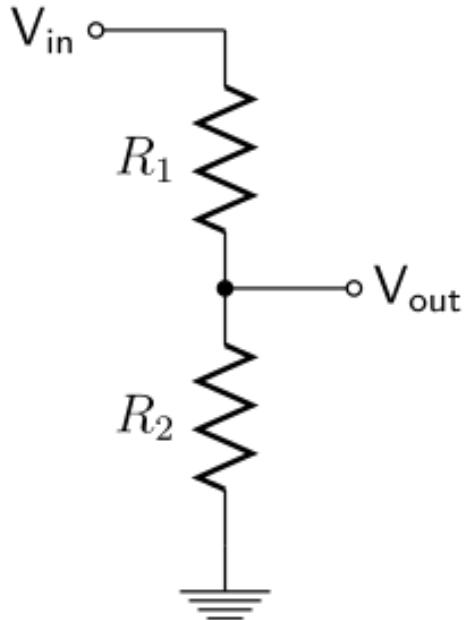
Pomimo swojej prostoty budowy i zastosowania, dla danego projektu ważne

jest aby wybrać odpowiednie rezystory. Podstawową wartością na którą należy zwrócić uwagę jest rezystancja. Rezystancję podaje się w omach i można spotkać na rynku zakres od milionów do megaomów. Spośród dostępnych rodzajów rezystorów w projekcie zostały użyte rezystory THT (z ang. Through-Hole Technology) - czyli tak zwane rezystory do montażu przewlekłanego. W tym rodzaju oporników rezystancja jest ilustrowana poprzez kolorowe paski umieszczone wokół oporu, co pozwala odczytać ich wartość według ilustracji 4.3. Alternatywą do tego sposobu jest podłączenie rezystora pod miernik elektryczny ustawiony w tryb pomiaru oporu [14]. Element ten jest kluczowy w celu ograniczenia przepływu prądu w obwodzie rękawicy, co pozwala na monitorowanie oporu wytwarzanego poprzez czujnik wygięcia.

Kolor	Wartość		Mnożnik	Tolerancja ± %	Współczynnik temp. ± ppm/K
	1 pasek	2 pasek			
czarny	0		x 1 Ω		250
brązowy	1	1	x 10 Ω	1	100
czerwony	2	2	x 100 Ω	2	50
pomarańczowy	3	3	x 1 kΩ		15
żółty	4	4	x 10 kΩ		25
zielony	5	5	x 100 kΩ	0,5	20
niebieski	6	6	x 1 MΩ	0,25	10
fioletowy	7	7	x 10 MΩ	0,1	5
szary	8	8	x 100 MΩ	0,05	1
biały	9	9	x 1 GΩ		
złoty			0,1 Ω	5	
srebrny			0,01 Ω	10	
brak				20	

Rysunek 4.3: Oznaczenia rezystorów

Źródło: <https://sites.google.com/site/informatykauni/jna/home/poszczegolne-czesci/rezystor>



Rysunek 4.4: Układ dzielnika napięcia
Źródło: https://www.wikiwand.com/en/Voltage_divider

Sposób działania układu, nazywanego dzielnicą napięcia jest pokazany na rysunku 4.4, oraz wyraża się wzorem

$$\nu_{out} = \nu_{in} \left[\frac{R_2}{R_1 + R_2} \right]$$

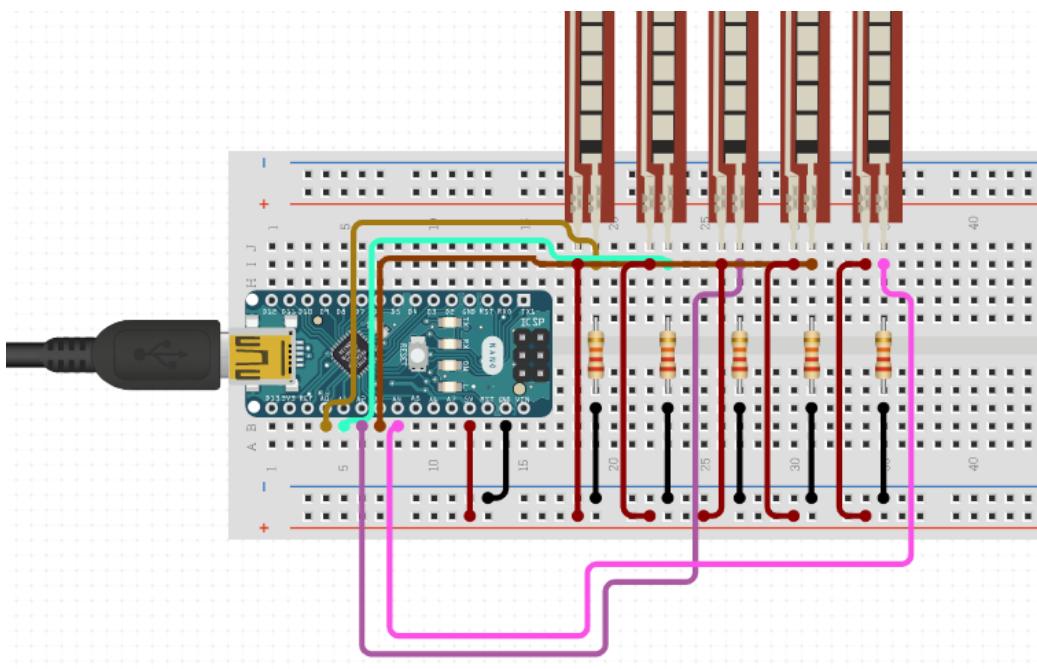
Wzór ten podaje napięcie wyjściowe ν_{out} , które równa się napięciu wejściowemu ν_{in} przeskalowanemu przez stosunek rezystorów. W opisywanym przypadku jest to stosunek zastosowanego rezystora $4.7k\Omega$ wyrazonego we wzorze poprzez R_2 , do sumy tego rezystora wraz z oporem wytwarzanym poprzez czujnik wygięcia R_1 - który

jak opisano w podrozdziale 4.1.2 jest zmienny. Oznacza to że im bardziej czujnik wygięcia jest zgięty, wytwarza on większy opór a co za tym idzie napięcie wyjściowe spada. Miara ta obrazuje jak bardzo palec jest zgięty i jest możliwa do uzyskania właśnie dzięki zastosowaniu układu dzielnika napięcia [4].

4.2 Budowa rękawicy

W poniższej sekcji zaprezentowano sposób w jaki zostały złączone wszystkie elementy rękawicy wspomniane w sekcji 4.1, aby była gotowa na oprogramowanie mikrokontrolera tworząc finałowy produkt. W tym celu została wybrana rękawica budowlana o grubych niciach ze ściągaczem wokół nadgarstka w celu zapewnienia komfortu, jak i precyzji położenia. Wybór ten również jest uzasadniony faktem początkowego planu dotyczącego wszycia materiału bezpośrednio w ręk-

wice jak i elastyczności które zapewnia grubsza rękawica. Tak jak wspomniano w podsekcji 4.1.2, do połączenia elementów została wykorzystana nić przewodząca. Dzięki grubym włóknom odstępu pomiędzy nićmi przewodzącymi prąd mogły być mniejsze, bez obawy przed spięciami w trakcie poruszania ręką. Dla tego projektu kontroler jest budowany dla lewej dłoni. Układ przewodów kontrolera jest zobrazowany na rysunku 4.5. Jest to jedynie obraz poglądowy, przedstawiony na płytce prototypowej a szczegółowy opis połączeń kontrolera zostanie opisany poniżej.



Rysunek 4.5: Poglądowy układ kontrolera przedstawiający sposób podłączenia dzielnika napięcia

Źródło: <https://www.circuito.io/app?components=514,8606,8606,8606,11022>

Mikroprocesor został umieszczony na wysokości centralnej części dłoni, przy kciuku, skierowany złączem USB na zewnątrz, pozwalając na łatwe podłączenie przewodu zasilającego jak i również ustawienie wejść analogowych w stronę



Rysunek 4.6: Efekt końcowy rękawicy-kontrolera

palców lewej dłoni. Z tej strony płytki Arduino będziemy też korzystać - zarówno przy wyprowadzaniu napięcia jak i uziemienia. Montaż samej płytki nie stanowi żadnego problemu ze względu na cztery otwory w rogach pozwalające na mocowanie, do którego została użyta zwykła nić do szycia. Aby stworzyć obwód wychodzący od pinu GND (z ang. Ground) czyli właśnie uziemienia, należy najpierw poprowadzić go przez rezystory. W tym celu wyjścia oporników zostały wygięte w pętle oraz zalutowane aby w łatwy sposób można było je przyszyć do rękawicy. Aby dodatkowo ułatwić sobie to zadanie - dodatkowo zostały one przyklejone bezpośrednio do materiału na bardzo małą ilość kleju. Umiej-

scowienie rezystorów jest po zewnętrznej wierzchniej stronie dloni, zaczynając się na wysokości kontrolera, zmierzając szeregowo w stronę nadgarstka. W ten oto sposób nić przewodząca została poprowadzona na zewnętrzną część dloni a następnie poprzez wszystkie pętle rezystorów, kończąc na ostatnim. Z drugiej strony rezystora dochodzi natomiast nić pochodząca z czujnika oporu. W tym celu podczas tworzenia czujników oporu pozostawiono 25 cm długości nici, co pozwoliło na przy mocowanie sensorów, a następnie połączenie obwodu. Sensory zostały zaopatrzone w krótkie, 1-2 cm długości kawałki taśmy elastycznej która została przyklejona na czubku każdego z sensorów pozwalając na elastyczny ruch sensora bez zerwania nici. Taśma ta została przyszyta w czubkach palców co dało punkt zaczepu dla sensorów. W tym momencie pozwoliło to na wygodne używanie nici wychodzących z sensorów w celu zamknięcia obwodu. Z powodu dużego zagęszczenia nici, które nie mogą się ze sobą stykać podczas ruchu ręki, ważnym dla projektu było naprzemienne używanie przestrzeni na rękawicy od spodu jak i od góry. Dzięki temu nici uziemienia oraz napięcia mogą się krzyżować nie zaburzając przy tym odczytów z sensorów. W celu jak najlepszego zagospodarowania przestrzenią wokół sensorów, uziemienie kciuka oraz małego palca zostały umiejscowione na nicie wychodzącej z prawej strony sensora, natomiast napięcie po lewej stronie. Dla pozostałych palców ustawienie to jest odwrotne. Mając to na uwadze została poprowadzona nić od kciuka poprzez wyjście 3.3 V na płytce a następnie wzduż krykci rękawicy, pozwalając nicią odpowiedzialnym za napięcie w odpowiednich sensorach na przyczepienie się w celu poboru napięcia tworząc tym samym dodatnią stronę układu. Nici wychodzące z sensorów które natomiast nie zostały do tej pory użyte, zostały przeszyte poprzez wyjścia analogowe a następnie podłączone kolejno do odpowiadających im rezystorów. Wyjścia odpowiadające każdemu z palców zostały opisane w tabeli poniżej.

Palec	Wyjście analogowe
Kciuk	A3
Wskazujący	A0
Środkowy	A1
Serdeczny	A6
Mały	A7

Sposób w jaki zostały dobrane wyjścia jest podyktowany zaleceniami o nie korzystaniu z wyjść analogowych A4 oraz A5 oraz pozostawieniu przestrzeni wokół wyjścia A3 przeznaczonego na kciuka, ponieważ jako jedynie połączenie musiało najpierw zmierzać w kierunku palców a następnie w dół w celu połączenia z rezystorem. Efekt końcowy pracy przedstawia zdjęcie 4.6.

4.3 Oprogramowanie mikrokontrolera

Do tej pory została opisana teoria omawianego kontrolera, elementy które zostały użyte w projekcie a także sposób ich połączenia. W niniejszej sekcji przedstawiony jest kod programu który został napisany w środowisku programistycznym oraz języku o tej samej nazwie - Arduino. Język ten poza drobnymi zmianami opiera się na języku C/C++, a jego pełny kod źródłowy jest dostępny na platformie Github [10]. Aby rozpocząć pracę z wybranym produktem od Arduino należy wykonać pewne czynności przygotowywacze, takie jak instalacja sterowników płytka dla danego systemu czy sposób obsługi w samym środowisku Arduino. Czynności te zostały opisane na stronie producenta, w związku z czym nie zostaną one szczegółowo opisane [27]. Po spełnieniu wszystkich wymagań, został przygotowany program który został napisany oraz wgrany do mikrokontrolera, który można podzielić na trzy części:

- Deklaracje
- Ustawienie mikrokontrolera

-
- Główna pętla programu

których cel oraz opis został przedstawiony poniżej.

W pierwszej kolejności zostanie przedstawiona część deklaracji, w której to dołączono wymagane biblioteki dla poprawnej obsługi wszystkich sensorów. Pierwszą biblioteką która została dołączona do programu jest *Arduino_LSM9DS1.h*. Biblioteka ta jest odpowiedzialna za obsługę IMU która przekazuje dane poprzez I2C do mikroprocesora. Biblioteka ta zajmuje się obsługą połączenia jak i kalibracja całego modułu. Inicjalizacja wartości poprzez bibliotekę wygląda w następujący sposób [25]

Sensor	Zakres	Częstotliwość
Akcelerometr	$[-4, +4]g - / + 0.122mg$	104Hz
Żyroskop	$[-2000, +2000]dps + / - 70mdps$	104Hz
Magnetometr	$[-400, +400]uT + / - 0.014uT$	20Hz

Tak jak wspomniano w sekcji 4.1.1 do połączenia bezprzewodowego Arduino Nano 33 BLE wykorzystuje moduł Bluetooth Low Energy, w związku z czym w części deklaracji dołączamy przeznaczoną do tego bibliotekę o nazwie *ArduinoBLE.h*. Biblioteka ta pozwala na dostęp oraz sterowanie modułem BLE (z ang. Bluetooth Low Energy), i na potrzeby projektu zostaną opisane używane elementy z poniższych klasy których znajomość jest wymagana w celu zrozumienia napisanego programu. Klasy te to

- **BLEService**
- **BLECharacteristic**
- **BLEDescriptor**
- **BLE**
- **BLEDevice**

Szczegóły na temat zasad działania BLE są podane w sekcji 3.3, w tej sekcji jedynie zostanie opisana struktura tych klas. *BLEService* umożliwia nawiązanie połączenia z innym urządzeniem obsługującym bluetooth i jako parametr przyjmuje UUID - jest to jedyny paramater jaki należy podać i w ten sposób zostaje stworzony nowy serwis BLE dostępny pod tym identyfikatorem. Klasa *BLECharacteristic* tworzy nową cechę którą należy przypisać do danego serwisu. Cechą ta może zostać zadeklarowana jako cecha wybranego z pośród dostępnych typów w Arduino bądź jako uniwersalna poprzez podstawową klasę *BLECharacteristic*. Klasa ta przyjmuje trzy wymagane parametry: UUID, właściwości cechy oraz wartość. Wartość może zostać zadeklarowana jako podany ciąg znaków, bądź też poprzez określenie rozmiaru danych i wartość początkową. Wartość początkowa nie musi być podana w trakcie deklaracji. Ważnym elementem tej klasy jest określenie jej właściwości. W zależności od przeznaczenia mamy do wybory następujące opcje, które mogą być ze sobą łączone: BLEBroadcast, BLERead, BLEWriteWithoutResponse, BLEWrite, BLENotify, BLEIndicate. W celu wysyłania danych podczas gdy dane te się zmienią cechy w prezentowanym programie przyjęły dwie wartości BLE: read (z ang. czytać) oraz notify (z ang. powiadomić). Klasa *BLEDescriptor* jest niejako klasą pomocniczą. Wartości tej klasy przypisuję się do danej cechy w celu lepszej obsługi serwisy oraz łatwiejszego rozpoznawania w przypadku pracy ze skomplikowanymi serwisami. Standardowo należy podać UUID danego deskryptora jako parametr, a także jego wartość jako ciąg znaków, bądź wartość w postaci tablicy bajtów oraz maksymalnego rozmiaru danych. Są to podstawowe klasy znajdujące się w części deklaracji których użycie zostały przedstawione na listingu 4.1 prezentującym wycinek części deklaracji [26]. Pozostałe dwie klasy zostaną opisane w części głównej pętli programu gdzie zostanie również pokazane ich zastosowanie.

```

1 #include <Arduino_LSM9DS1.h>
2 #include <ArduinoBLE.h>
3 BLEService imuService("1101");
4
5 BLECharacteristic imuAccChar("2101", BLERead | BLENotify,
6 , 12);
7 BLECharacteristic imuGyroChar("2102", BLERead | BLENotify,
8 , 12);
9 BLECharacteristic fingersChar("2103", BLERead | BLENotify,
10 , 20);
11 BLEDescriptor     imuAccDescriptor("3101", (byte*) "Acc
12 Descriptor", 15);
13 BLEDescriptor     imuGyroDescriptor("3102", (byte*) "Gyr
14 Descriptor", 15);
15 BLEDescriptor     fingersDescriptor("3103", (byte*) "Fin
16 Descriptor", 15);

```

Listing 4.1: Część deklaracji programu mikrokontrolera

Następną częścią programu jest ustawienie mikrokontrolera oraz sprawdzenia poprawności działania sensorów. Gdy proces ten zakończy się powodzeniem zostanie uruchomiona główna część programu - funkcja *loop()*, która wykonuje się nieprzerwania pozwalając kontrolować pracę naszego kontrolera. Za część związaną z inicjalizacją jest odpowiedzialna funkcja *setup()*. Funkcja ta jest wywoływana tylko raz za każdym razem gdy płytka jest podłączona do prądu bądź zostanie zresetowana [6]. W funkcji tej wywoływana jest statyczna metoda *begin(z ang. rozpoczęć)* na klasie *IMU* która należy do biblioteki *Arduino_LSM9DS1.h*, i to właśnie ta funkcja pozwala na inicjalizację wspominanych sensorów wchodzących w skład jednostki pomiarowej - program zapętli się w tym miejscu jeśli funkcja ta zwróci błąd, poprzez rozpoczęcie pętli z warunkiem zawsze prawdziwym, co przedstawia listing 4.2 [25].

```

12 if (!IMU.begin()) {
13     Serial.println("Failed to initialize IMU!");
14     while (1);
15 }

```

```
16  
17     if (!BLE.begin()) {  
18         Serial.println("Failed to start Bluetooth!");  
19         while (1);  
20     }
```

Listing 4.2: Inicjalizacja IMU oraz BLE

Jeżeli sensory działają poprawnie, sprawdzana jest możliwość połączenia poprzez moduł bezprzewodowy. W tym celu wykorzystywana jest statyczna klasa *BLE*, która odpowiada za włączenie modułu Bluetooth oraz jego ustawienia. Tak jak w przypadku klasy *IMU*, program zapętli się w tym miejscu zwracając błąd jeżeli połączenie nie będzie możliwe. W przypadku powodzenia program rozpoczyna łączenie zadeklarowanych elementów, takich jak serwis, cechy oraz deskryptory tych cech. Klasa *BLE* posiada funkcje pozwalającą ustawić nazwę dla naszego połączenia, jej adres a przede wszystkim gdy wszystkie elementy są już gotowe wywołuję metodę *advertise()*, która pozwala na połączenie się z naszym serwisem. Na listingu 4.3 pokazane jest szczegółowa konfiguracja klasy BLE oraz serwisu a także odpowiednie użycie funkcji dla jednej z cech, które należy powtórzyć dla każdej dodawanej cechy z odpowiednimi wartościami.

```
21     BLE.setLocalName ("VrGlove");  
22     BLE.setAdvertisedService(imuService);  
23  
24     imuService.addCharacteristic(imuGyroChar);  
25     imuGyroChar.addDescriptor(imuGyroDescriptor);  
26     imuGyroChar.writeValue((byte) 0x01);  
27  
28     BLE.addService(imuService);  
29     BLE.advertise();
```

Listing 4.3: Obsługa serwisu przy użyciu klasy BLE

W ten oto sposób kończy się funkcja *setup()* i zostaje wywołana kolejna funkcja która od tej pory będzie odpowiedzialna za pracę kontrolera - *loop()*. W funkcji tej zostaje zadeklarowana ostatnia z wymienionych klas do obsługi Bluetooth. Klasa

ta nazywa się *BLEDevice* i jest bezpośrednio powiązana z urządzeniem które jest aktualnie podłączone. Dopóki nie ma podłączonego do kontrolera urządzenia, żadne czynności nie zostają podjęte. W momencie uzyskania danych z modułu łączności o nawiązanym połączeniu zostaje uruchomiona pętla, funkcjonująca tak długo jak połączenie to nie zostanie przerwane. Fragment kodu znajduje się na listingu 4.4 [26]. Niestety w trakcie pracy z urządzeniem odkryto błąd związany z rozłączeniem się centrali. Problem pojawi się gdy urządzenie zostaje rozłączone z mikrokontrolerem - w tym momencie klasa *BLE* nie zawsze zgłasza informację o rozłączeniu, myśląc że urządzenia cały czas są podłączone. Problem został zauważony, jednak nie jest rozwiązany na stan z maja 2020 roku [24].

```
30     BLEDevice central = BLE.central();
31     if (central) {
32         Serial.print("Connected to central: ");
33         Serial.println(central.address());
34     }
35     while( central.connected()) {
36         [...]
37     }
```

Listing 4.4: Oczekiwanie na połączenie z urządzeniem przez mikrokontroler

Gdy wszystkie dotychczasowo opisane elementy programu nie zgłoszą problemów, następuje ostatnia faza którą jest zbieranie i przesyłanie danych. W pierwszej kolejności sprawdzany jest warunek czy jednostka pomiarowa ma dostęp do nowych danych. W przypadku tej aplikacji dane które zostały poddane analizie pochodzą z żyroskopu oraz z akcelerometru i są wyrażone jako zmienne *x,y* oraz *z* typu *float*. Z powodu naturalnych drgań dloni dane te ciągle się zmieniają w mikro skali która nie ma wpływu na efekty, niemniej jednak możemy założyć że gdy mikroprocesor nie jest przymocowany do stałego obiektu, dane te będą dostarczane z częstotliwością pracy sensorów. Za odczyt danych z żyroskopu i akcelerometru odpowiadają odpowiednio funkcje *readGyroscope(x,y,z)* oraz *re-*

adAcceleration(x,y,z) z klasy *IMU* które przypisuję odpowiednie wartości do podanych parametrów. Ważnym elementem podczas odczytu tych danych jest tak zwany szum który powstaje w trakcie przygotowania sensorów. W trakcie pierwszych odczytów szum ten został zmierzony oraz usunięty z pomiarów. Akcelerometr położony na płaskim obiekcie prawidłowo podawał odczyty, czyli zwracał wektor $[0, 0, g]$ gdzie g oznacza grawitację ziemską. Żyroskop natomiast wskazywał błąd rzędu $[2.80, 0.18, 0.18]$ w związku z czym od każdego odczytu właśnie taką wartość należy odjąć w celu uzyskania prawidłowych danych - czyli wektora zbliżonego do $[0, 0, 0]$ w pozycji w której został skalibrowany. Jak wspomniano w sekcji 3.1.1 dotyczącej żyroskopu, wartości zwarcane są podane w $\frac{rad}{s}$, również znane jako dps (z ang. Degrees per second) czyli w kątach na sekundę. W celu określenie kątów w jakim urządzenie się znajduje w danym momencie musimy te dane przemnożyć przez częstotliwość z jaką są one pobierane - czyli przemnożyć przez czas co zwróci nam miarę kątów. Osiągamy to poprzez zapisanie czasu w którym dane zostały pobrane a także poprzez zapisanie tej wartości przed następnym pobraniem danych jako czas ostatniego poboru. Różnica pomiędzy tymi wartościami daje czas jaki upłynął aby uzyskać nowe wartości. Zaczynając w pozycji kalibracyjnej, z idealnie usuniętym szumem, żyroskop zwróci wartość zero, a wraz ze zmianą wartości żyroskopu, suma tych zmian wskaże na orientację kontrolera względem pozycji wyjściowej. Więcej informacji na ten temat znajduje się w rozdziale 5 [12]. Ostatnią cechą którą chcemy przekazać są dane pobierane z sensorów wygięcia w celu określenia pozycji palców. Tak jak wspomniano w sekcji 4.1 dane te pobieramy przez mierzenie napięcia jakie znajduje się na poszczególny analogowych pinach wejściowych. Odczyty te uzyskujemy poprzez wywołanie metody *analogRead(pin)* [6]. Pobieranie danych oraz ich wstępna obróbka przedstawia listing 4.5.

```

38 if (IMU.accelerationAvailable() && IMU.gyroscopeAvailable
39   ()) {
40     previousTime = currentTime;
41     currentTime = millis();
42     elapsedTime = (currentTime - previousTime) / 1000;
43
44     IMU.readAcceleration(acc[0], acc[1], acc[2]);
45     IMU.readGyroscope(gyro[0], gyro[1], gyro[2]);
46
47     gyro[0] -= 2.80;
48     gyro[1] -= 0.18;
49     gyro[2] -= 0.18;
50
51     fingers[0] = analogRead(A3);
52     [...]
}

```

Listing 4.5: Wczytywanie danych z sensorów.

Na tym etapie mamy już dostęp do danych z akcelerometru w $\frac{m}{s^2}$ oraz dane z żyroskopu wyrażone w stopniach. Informacje te mogłyby zostać przesłane w takiej formie, jednak z racji znajomości projektu, dane można dodatkowo skorygować poprzez zastosowanie dodatkowych filtrów. Podstawowym zabiegiem który został zastosowany jest eliminacja danych nieznaczących, czyli szumu który powstaje poprzez naturalne drgania ciała. Aby to osiągnąć, w programie przechowywane są poprzednie wartości z poszczególnych sensorów. Jak dane znaczące dla akcelerometru przyjęto różnice ± 0.1 , a dla danych z sensorów wygięcia ± 15 . Dla żyroskopu natomiast przyjęto wartości różniące się o przynajmniej ± 0.2 , a także dodatkowo sprawdzane jest czy dane z żyroskopu nie są w pozycji przy kalibracyjnej. Oznacza to że jeżeli wartości zwracane z żyroskopu wynoszą 0 ± 0.1 , zostaną one zamienione na wartość równą 0. Filtrowanie wartości pokazane jest na listingu 4.6. W ten oto sposób otrzymano wartości z sensorów które były gotowe do wysłania na inne urządzenie. Wartości te jednak są wyrażone w postaci tablic typu *float*, natomiast jako wartości cechy bluetooth przyjmują tablice by-

tów. W związku z tym przed przypisaniem danych są przekazywane adresy tablic do nowych zmiennych, a następnie zmienne te zapisane w poszczególnej cęsie przy użyciu metody `setValue(value,valueSize)`. Rozmiar jest ten określony jako 12 ponieważ mamy do czynienia z tablicą 3 zmiennych typu `float` z których każda zajmuje 4 bajty pamięci. Zapisywanie danych jest pokazana na listingu 4.7 dla jednej cechy - proces ten należy powtórzyć dla wszystkich danych. Program następnie zapisuje obecne dane jako dane obecnej pętli, tak aby następne odczyty mogły zostać porównane z obecną iteracją, tym samym kończąc wykonywaną pętlę. Warto zauważyć że program pomimo swojego głównego działa w funkcji `loop()`, jest wykonywany w ramach jednej iteracji jak tylko dojdzie do połączenia kontrolera z odbiornikiem.

```
53 | for (int i = 0;i<3;i++) {
54 |     if(!(gyro[i] < oldGyroData[i]-0.2 || gyro[i] >
55 |         oldGyroData[i]+0.2)){
56 |         if( gyro[i] < -0.1 || gyro[i] > 0.1){
57 |             gyro[i] = oldGyroData[i];
58 |         }else{
59 |             gyro[i] = 0.0;
60 |         }
61 |
62 |         if(!(acc[i] < oldAccData[i]-0.02 || acc[i] > oldAccData
63 |             [i]+0.02)){
64 |             acc[i] = oldAccData[i];
65 |         }
66 |
67 |     for (int i = 0;i<5;i++){
68 |         if(!(fingers[i] < oldFingersData[i]-15.0 || fingers[i] >
69 |             oldFingersData[i]+15.0)){
70 |                 fingers[i] = oldFingersData[i];
71 |             }
72 |     }
```

Listing 4.6: Wstępne filtrowanie danych z sensorów.

```
73     byte *accChar = (byte*)&acc;
74     imuAccChar.setValue(accChar, 12);
```

Listing 4.7: Zapisywanie danych do cech w serwisie.

4.4 Podsumowanie

W tym rozdziale podano informacje na temat projektu rękawicy kontrolera a także elementów które są wymagane w celu jego funkcjonowania. Zostały podane specyfikacje podzespołów, sposoby pozyskiwania danych, metoda komunikacji z innymi urządzeniami a także rozwiązano problemy związane z kosztem projektu. Krok po kroku przedstawiono złożenie elementów w celu uzyskania końcowej wersji produktu i ostatecznie omówiono zasadę działania programu napisanego w Arduino wraz z bibliotekami które zostały użyte w ramach projektu, natomiast zasady ich praktycznego działania zostały pokazane na listingach. W ten oto sposób została zakończona główna część projektu, pozwalająca na wykorzystanie rękawicy kontrolera jako część większych przedsięwzięć. Aby jeszcze lepiej zobrazować sposób obsługi kontrolera, oraz metody wykorzystania danych, w rozdziale 5 zostanie pokazana przykładowa aplikacja wykorzystująca dane w celu analizy oraz prezentacji.

Rozdział 5

Dedykowana aplikacja w systemie Android

W tym rozdziale pokazana zostanie przykładowa aplikacja która współpracuje z rękawicą-kontrolerem. Aplikacja ta ma za zadanie wyświetlanie aktualnego stanu sensorów rękawicy oraz połączenia, a także prezentowanie tych danych. Pomimo możliwości wykorzystania wielu dostępnych platform do tworzenia animacji oraz środowisk wirtualnych, zdecydowano się na napisanie aplikacji na system Android wykorzystując do tego język Java. Aplikacja ta pozwala na wysoką mobilność, zagłębianie się w tematykę BLE, dzięki zaprogramowaniu własnego połączenia z serwisem udostępnianym przez rękawicę, a także łatwe zaznajomienie się z działaniem rękawicy nawet bez żadnej informacji dotyczącej jej obsługi. Rozdział ten przedstawia interfejs użytkownika, sposób komunikacji z rękawicą-kontrolerem a także opis używanego przez aplikację SDK o nazwie *Google Sceneform*. Dołączony projekt firmy Google pozwala na importowanie modeli dloni do aplikacji, prezentację ich a także odpowiada za animację na ekranie. W tym rozdziale zostaną pokazane zaimplementowane rozwiązania z którymi mierzą się twórcy kontrolerów, w szczególności przy wykorzystaniu nawigacji inercyjnej.

5.1 Interfejs

W niniejszej sekcji opisano interfejs wraz z jego elementami oraz ich funkcjonalność w ramach całej aplikacji. Aplikacja składa się z jednej aktywności o nazwie *MainActivity* w ramach której został zaimplementowany układ zakładek, a konkretnie dwóch zakładek - dane oraz animacja. Zakładki te odpowiednio są uzupełnianie przygotowanymi fragmentami poprzez klasę *SectionsPagerAdapter* oraz *ViewPager*. W celu odpowiedniej obsługi fragmentów, aktywność ta musi implementować metody interfejsu *OnFragmentInteractionListener*, które są zadeklarowane w klasach *ModelRenderer* oraz *GloveData*. Klasy te aby spełnić swoją funkcjonalność dziedziczą po klasie *Fragment* [13]. W ten oto sposób zadeklarowano trzy klasy na podstawie których wyświetlany jest interfejs użytkownika, przedstawiony na rysunku 5.1. Metodę *onCreate(Bundle)* przedstawia listing 5.1 na którym to widać krok po kroku sposób ładowania fragmentów w aktywności. Dwie ostatnie linie listingu pokazują wywołanie rejestracji odbiornika w celu otrzymywania informacji dotyczącego modułu Bluetooth wbudowanego w urządzenie na którym aplikacja jest uruchomiona. Oprócz wspominanych do tej pory klas, aplikacja korzysta również z dodatkowej klasy o nazwie *VrGlove*, służącej do przechowywania informacji na temat kontrolera.Więcej informacji o tej klasie oraz o odbiorniku modułu Bluetooth zostanie przedstawionych w sekcji 5.2.

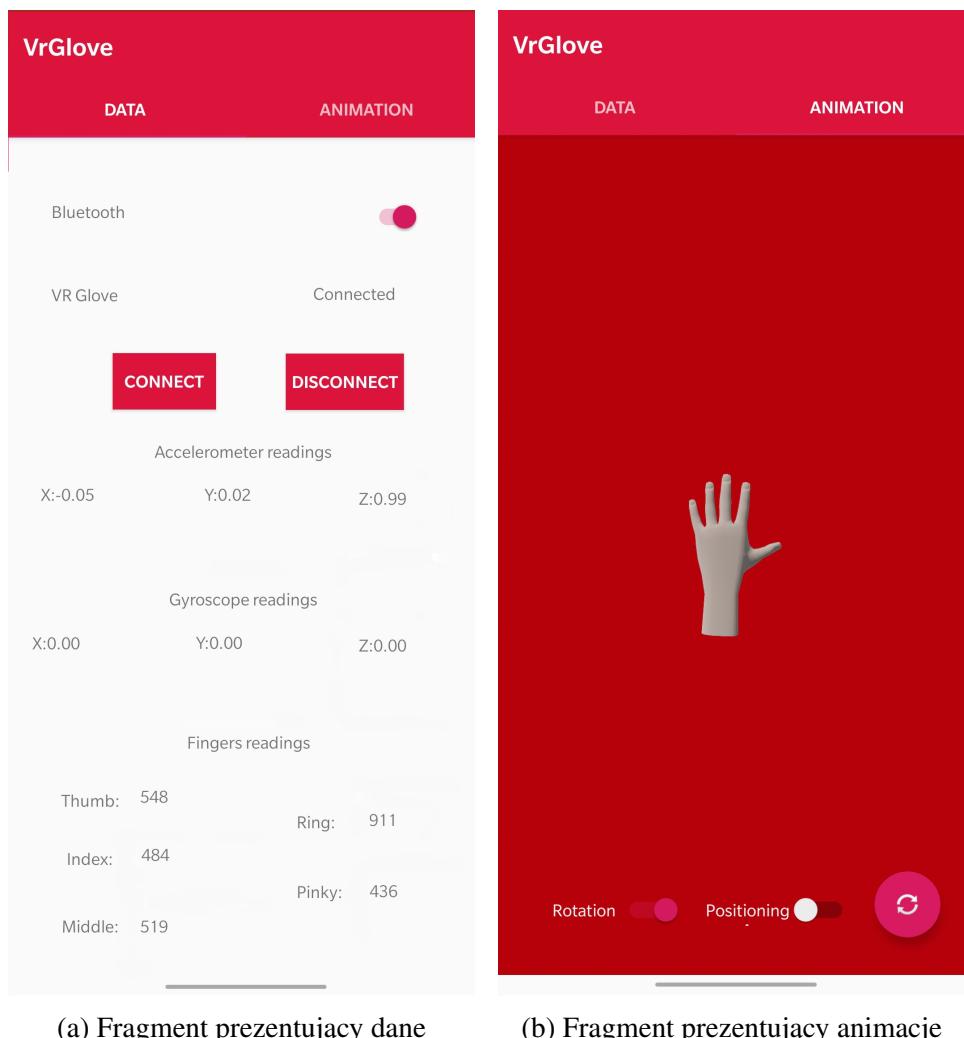
```
75  @Override
76  protected void onCreate(Bundle savedInstanceState) {
77      super.onCreate(savedInstanceState);
78      setContentView(R.layout.activity_main);
79      SectionsPagerAdapter sectionsPagerAdapter = new
80          SectionsPagerAdapter(this,
81              getSupportFragmentManager());
82      ViewPager viewPager = findViewById(R.id.view_pager)
83          ;
84      viewPager.setAdapter(sectionsPagerAdapter);
85      TabLayout tabs = findViewById(R.id.tabs);
86      tabs.setupWithViewPager(viewPager);
```

```

84
85     IntentFilter filter = new IntentFilter(
86         BluetoothAdapter.ACTION_STATE_CHANGED);
87     registerReceiver(mReceiver, filter);
87 }

```

Listing 5.1: Interfejs użytkownika.



Rysunek 5.1: Interfejs użytkownika

Jak widać na 5.1a rozpoczętając od góry widzimy informację dotyczącą obecnego stanu modułu Bluetooth w urządzeniu, wyrażanego poprzez przycisk prze-

łącznika, pozwalający na włączenie/wyłączenie modułu bezpośrednio z poziomu aplikacji. Pod spodem Widnieje informacja o możliwości połączenia/statusie połączenia z kontrolerem opisana etykietą VR Glove. Etykieta ta przyjmuje następujące wartości:

- Włącz Bluetooth (ang. Turn On Bluetooth)
- W trakcie włączania (ang. Turning On)
- W trakcie wyłączania (ang. Turning Off)
- Gotowy do połączenia (ang. Ready to connect)
- W trakcie łączenia (ang. Connecting)
- Połączony (ang. Connected)
- W trakcie rozłączania (ang. Disconnecting)
- Rozłączony (ang. Disconnected)

Pierwsze trzy stany możemy osiągnąć poprzez włączanie/wyłączanie modułu Bluetooth w urządzeniu. Stan gotowości do połączenia pokazuję się gdy moduł bluetooth został włączony i jest gotowy do połączenia z kontrolerem. Dwa przyciski znajdujące się poniżej etykiety, odpowiednio *połącz* (ang. *Connect*) i *rozłącz* (ang. *Disconnect*) pozwalają na połączenie z rękawicą-kontrolerem. Ostatnie cztery stany obrazują status połączenia z rękawicą, który możemy zmienić korzystając odpowiednio z przycisków. Poniżej znajdują się etykiety dotyczące sensorów kontrolera które są puste gdy aplikacja nie została jeszcze połączona z kontrolerem. Pola ta odpowiednio od góry reprezentują wartości zwarcane przez akcelerometr i żyroskop jako wartości X,Y i Z, oraz wartości sensorów znajdujących się na palcach, z każdym sensorem mającym własną etykietę. Drugą częścią

interfejsu prezentuje 5.1b, na której to widoczna jest lewa dłoń, znajdująca się na środku ekranu. Pozycja ta jest przyjmowana zanim kontroler zostanie podłączony. W dolnej części ekranu widzimy dwa przyciski przełączniki, służące do regulowania modelu, pozwalając decydować nam które elementy mają być brane pod uwagę podczas generowania animacji. Od lewej odpowiednio możliwe jest do zmiany branie pod uwagę orientacji kontrolera przy generowaniu modelu, następnie jego pozycji względem pozycji kalibracyjnej, oraz ostatni element tego interfejsu czyli FAB(z ang. Floating Action Button) - służący do ponownej kalibracji dłoni. Kalibracja ta jest również wywoływana przy każdej zmianie decyzji dotyczącej generowania rotacji bądź położenia. Pozycją kalibracyjną jest pozycja w której lewa dłoń na której znajduje się rękawica-kontroler, wraz ze wszystkimi palcami znajdują się w pozycji wyprostowanej, a kciuk wskazuje ciało użytkownika. Interfejs ten pozwala na szybkie połączenie się z kontrolerem a gdy tylko pierwsze dane zostaną przesłane do aplikacji, natychmiastowo obserwujemy pracę kontrolera na ekranie smart-fona. Warto zauważyć że po włączeniu aplikacji przełącznik orientacji jest włączony natomiast pozycjonowanie dłoni na ekranie wyłączone, w związku z czym animacja wyświetla się na środku ekranu. Do tej pory opisano wygląd i informacje elementów znajdujących się na ekranie, dalsza część tego rozdziału pokaże w jaki sposób wspomniane elementy działają od strony kodu źródłowego.

5.2 Komunikacja

Opis interfejsu aplikacji pokazuje elementy które są wymagane oraz zaprogramowane w aplikacji. Pokazuje jakie dane są używana oraz w jakim celu wykorzystywane. Żeby jednak skorzystać z tych danych najpierw aplikacja musi zostać połączona z kontrolerem. W sekcji 4.1.1 powiedziano o wykorzystywaniu

w tym celu połączenia BLE a o tym jak to jest obsługiwane przez prezentowaną aplikacje zostanie pokazane w części 5.2.1. Aby rozpocząć pracę z BLE przede wszystkim należy się upewnić że moduł Bluetooth w urządzeniu jest dostępny oraz włączony. W celu sprawdzenia dostępności modułu bluetooth w urządzeniu wykorzystywany jest plik *AndroidManifest.xml*, w którym to zdefiniowano dostęp. Wszystkie wymagane pozwolenia pokazuje listing 5.2.

```
88 <uses-permission android:name="android.permission.  
     BLUETOOTH" />  
89 <uses-permission android:name="android.permission.  
     BLUETOOTH_ADMIN" />  
90 <uses-permission android:name="android.permission.  
     READ_EXTERNAL_STORAGE" />  
91 <uses-permission android:name="android.permission.  
     CAMERA" />
```

Listing 5.2: Wymagane pozwolenia dla aplikacji.

Mając wymagany dostęp możemy sterować sensorem poprzez przycisk przełącznika. W kodzie programu wystarczy uzyskać do niego dostęp używając metody *findViewById(int)* a następnie ustawić nasłuchiwacza kliknięć. Przyciski *Połącz* oraz *Rozłącz* są zdefiniowane w ten sam sposób. Aby wprowadzić zmiany w Bluetooth należy użyć klasy *BluetoothAdapter*. Po pobraniu domyślnego adaptera jesteśmy w stanie określić jego stan. używając metod *enable()* oraz *disable()*. Sposób obsługi adaptera jest pokazany na listingu 5.3. Zmiana ta wywołuje funkcję znajdująca się w klasie *MainActivity* która reaguje na zmiany adaptera oraz ustawia jeden z pierwszych czterech statusów z listy 5.1 dla pola definiującego obecny stan połączenia. Skrócony listing 5.4 pokazuje wywołanie tej funkcji dla przykładowego stanu adaptera. Ostatnie cztery stany wypunktowane w 5.1 pochodzą ze zmiany połączenia wywoływane poprzez wspomniane przyciski *Połącz* oraz *Rozłącz* które zostaną opisane w sekcji 5.2.1.

```

92    switch (v.getId()) {
93        case R.id.switchBT:
94            Switch switchBT = v.findViewById(R.id.
95                switchBT);
96            BluetoothAdapter mBluetoothAdapter =
97                BluetoothAdapter.getDefaultAdapter();
98            if (switchBT.isChecked()) {
99                if (!mBluetoothAdapter.isEnabled()) {
100                    mBluetoothAdapter.enable();
101                }
102            } else{
103                if (mBluetoothAdapter.isEnabled()) {
104                    mBluetoothAdapter.disable();
105                }
106            }
107        [...]
108    }

```

Listing 5.3: Obsługa wbudowanego modułu Bluetooth.

```

105 private final BroadcastReceiver mReceiver = new
106     BroadcastReceiver() {
107         @Override
108         public void onReceive(Context context, Intent
109             intent) {
110             final String action = intent.getAction();
111             Switch tbBT= findViewById(R.id.switchBT);
112             TextView tvStatus = findViewById(R.id.
113                 textView_vrGlove_status);
114             if (action.equals(BluetoothAdapter.
115                 ACTION_STATE_CHANGED)) {
116                 final int state = intent.getIntExtra(
117                     BluetoothAdapter.EXTRA_STATE,
118                     BluetoothAdapter.ERROR);
119                 switch (state) {
120                     case BluetoothAdapter.STATE_OFF:
121                         tbBT.setChecked(false);
122                         tvStatus.setText("Turn on bluetooth
123                             ");
124                         break;
125                     [...]
126                 }
127             }

```

Listing 5.4: Zmiana statusu na podstawie adaptera bluetooth.

5.2.1 Obsługa połączenia Bluetooth Low Energy

Gdy poznano stan modułu Bluetooth, bez przeszkód można nawiązać połączenie. w tym celu wykorzystano przyciski sterujące połączeniem. Do przechowywania danych o połączeniu wykorzystywana jest klasa *VrGlove*, w której zdefiniowano statyczne zmienne klasy *BluetoothDevice* pozwalające na wybranie odpowiedniego urządzenia z puli dostępnych urządzeń w pobliżu poprzez jego adres oraz *BluetoothGatt* odpowiedzialnej za obsługę *GATT* (z ang. Generic Attribute Profile) w androidzie. Listę dostępnych serwisów otrzymano deklarując zmienną implementującą listę klasy *BluetoothGattService*. W ten sposób w dowolnym miejscu programu można odwołać się do klasy kontrolera, sprawdzając jego stan a także uzyskać dane o jego udostępnionych serwisach. W ten oto sposób na listingu 5.5 pokazano dostęp do klasy rękawicy z nasłuchiwacza przycisku *Rozłącz*. Fragment ten sprawdza czy istnieje aktualnie połączony serwis *GATT* oraz czy jest on w stanie *Połączony* wyrażony jako typ *int* - 2. To właśnie na podstawie tego statusu są określone ostatnie cztery stany połączenia z rękawicą z listy 5.1. Sposób zmiany statusu są bardzo zbliżone do listingu 5.4, różnica polega na wywołaniu odbiornika zmiany statusu połączenia z klasy *BluetoothGattCallback* w przeciwieństwie do *BroadcastReceiver* oraz zostaje wywołana metoda *onConnectionStateChange* zamiast metody *onReceive*. Jeżeli powyższe warunki są spełnione *GATT* zostaje rozłączony [13].

```
121 case R.id.buttonDisconnect:  
122     if(VrGlove.getGatt() != null && VrGlove.  
123         getGattState() == 2 ) {  
124         VrGlove.getGatt().disconnect();  
125     }  
126     break;
```

Listing 5.5: Obsługa przycisku rozłącz.

Ostatnia część którą opisano jest zarazem najważniejszą. Mowa o obsłudze przy-

cisku *Połącz*. Tak jak w przypadku przycisku *Rozłącz*, najpierw sprawdzany jest status urządzenia, czyli czy adapter jest włączony oraz w przeciwnieństwie do sprawdzania czy nasz serwis *GATT* jest połączony, kod przycisku wykona się tylko wtedy gdy nie jest aktualnie nawiązane połączenie. Gdy warunki te są spełnione, zostaje pobrany adapter bluetooth oraz zostaje podjęta próba połączenia z urządzeniem przy użyciu klasy *BluetoothDevice*, która otrzymuje zwracaną wartość metody *getRemoteDevice(String)* wywołaną na adapterze bluetooth. W naszym przypadku jako parametr typu *String*, zostaje podany adres *D0:6B:F2:A7:95:03*, który jest adresem rękawicy-kontrolera z którym zostanie podjęta próba połączenia. Następnie zostaje stworzona nowa instancja klasy *VrGlove*, której zostaje przekazana w parametrach wartość zmiennej typu *BluetoothDevice* oraz aktualny widok na którym pracuje fragment. Posiadając te informacje rozpoczyna się klawiszowy etap połączenia, mianowicie zostaje zainicjalizowana zmienna klasy *BluetoothLeScanner*, poprzez wywołanie metody *getBluetoothLeScanner()* na adapterze urządzenia. Metoda ta pozwala na wyszukiwanie urządzeń BLE znajdujących się w pobliżu, poprzez wywołanie metody *startScan(ScanCallback)*, gdzie jako parametr zostaje podany stworzony skaner, oczekujący na pojawienie się urządzenia z wcześniej podanym adresem. Bardzo ważnym elementem jest zatrzymanie skanera gdy zostaje odnalezione urządzenie - co można osiągnąć poprzez wywołanie metody *stopScan(ScanCallback)*. W ten oto sposób nawiązano połączenie pomiędzy urządzeniami a rezultat tego połączenia zostaje zapisany w postaci ustawienia serwisu *GATT* w klasie *VrGlove*. Opisane powyżej czynności są przedstawione na listingu 5.6 [13]. W ten oto sposób możemy kontrolować połączenie z kontrolerem z dedykowanej aplikacji. Ostatnim elementem poprawnego funkcjonowania jest przekazywanie danych w czasie rzeczywistym pomiędzy odbiornikiem a nadajnikiem, co zostanie pokazane w sekcji 5.2.2.

```
126 final BluetoothManager bluetoothManager =
127         (BluetoothManager) getActivity()
128             .getSystemService(Context.
129                 BLUETOOTH_SERVICE);
130 mBluetoothAdapter = bluetoothManager.getAdapter();
131 [...]
132 BluetoothDevice device = mBluetoothAdapter.getRemoteDevice(
133     "D0:6B:F2:A7:95:03");
134 new VrGlove(device, vw);
135 [...]
136 BluetoothLeScanner scanner = mBluetoothAdapter.
137     getBluetoothLeScanner();
138 scanner.startScan(scanCallback);
139 [...]
140 scanner.stopScan(scanCallback);
141 gatt = VrGlove.getDevice().connectGatt(getActivity(), false,
142     bluetoothGattCallback, TRANSPORT_LE);
143 VrGlove.setGatt(gatt);
```

Listing 5.6: Kluczowe elementy przycisku *Połącz* pozwalającego na połączenie z kontrolerem.

5.2.2 Pobieranie danych

Mając do dyspozycji informacja pozyskane w trakcie połączenia, które są przechowywane jako zmienne statyczne w klasie rękawicy, jesteśmy w stanie pozyskać dane które opisano w rozdziale 4. W tym celu, po stronie aplikacji należy sprawdzić czy aktualnie jest nawiązane połączenie, co jak już zostało powiedziane oznacza serwis *GATT* w stanie wyrażanym jako *int = 2*. Gdy potwierdzono połączenie, z klasy *BluetoothGatt* zostaje wywołana metoda *discoverServices()*, która jest wywoływana dopóki nie zostaną wykryte serwisy. Gdy tak się stanie, rezultat odnalezionych serwisów uzyskiwany jest poprzez metodę serwisu *GATT* *getServices()*. Mając dostęp do serwisów - w opisywanym przypadku wiemy z rozdziału dotyczącego rękawicy-kontrolera że jest to tylko jeden serwis, możemy pobrać cechy używając metody *getCharacteristic(*UUID*)*, które ten serwis

osiada. Cechy te zostały pokazane na listingu 4.1, oraz zostały przypisane im skrócone UUID wyrażone jako *int* od 0x2101 do 0x2103, odpowiednio jako dane akcelerometru, żyroskopu oraz sensorów umiejscowionych na palcach. Implementacje algorytmu z typu *int* do UUID pokazuje metoda *convertFromInteger(int i)* na listingu 5.7 [19].

```
139 |     private UUID convertFromInteger(int i) {  
140 |         final long MSB = 0x0000000000001000L;  
141 |         final long LSB = 0x800000805f9b34fbL;  
142 |         long value = i & 0xFFFFFFFF;  
143 |         return new UUID (MSB | (value << 32), LSB);  
144 |     }
```

Listing 5.7: Zamiana zmiennej int na UUID.

Dla każdej cechy zostaje wywołana metoda *readCharacteristic (BluetoothGattCharacteristic)*, która po sprawdzeniu warunków które są wymagane od każdej z cech dla prezentowanej aplikacji ustawia te cechy w trybie powiadomień - etap ten prezentuje listing 5.9. Tryb powiadomień dla cechy oznacza że dane zostaną pobrane za każdym razem gdy zajdzie w nich jakaś zmiana. Ostatnią częścią jest pobranie deskryptora danej cechy. W ten sposób oprócz nawiązanego połączenia uzyskano dostęp do serwisów oraz cech reprezentowanych przez urządzenie z którym się połączono. Etap ten zaprezentowano na wycinku kodu 5.8. Ważnym elementem tego procesu jest odczytywanie cech pojedynczo, z racji tego że serwis *GATT* obsługuje połączenie tylko z jedną cechą jednocześnie, co oznacza że gdyby spróbowało pobrać następną cechę, podczas gdy połączenie nie zostało zakończone z poprzednią cechą - połączenie to zostanie nadpisane. Obsługę połączenia z cechą zapewnia nadpisanie metody *onCharacteristicChange(BluetoothGatt, BluetoothGattCharacteristic)* wywoływaną z klasy *BluetoothGattCallback*. Kod został napisany właśnie w tej metodzie ze względu na tryb w jaki cechy zostały ustawione, czyli tryb powiadomień. Dzięki temu funkcja ta zostaje wywołana za każdym razem gdy obserwowane cechy zostaną w jakiś

sposób zmienione. W zależności od rozpoznanej cechy, wywoływana jest metoda zmieniające aktualny zestaw danych w klasie kontrolera, co pokazuje listing 5.10.

```
145 if (VrGlove.getGattState() == 2) {
146     VrGlove.getGatt().discoverServices();
147     [...]
148     VrGlove.setServices(VrGlove.getGatt().getServices());
149     for (int i = 0x2101;i<0x2104;i++) {
150         mCharacteristic = VrGlove.getServices().get(2).
151             getCharacteristic(convertFromInteger(i));
152         readCharacteristic((mCharacteristic));
153         BluetoothGattDescriptor descriptor =
154             mCharacteristic.getDescriptor(convertFromInteger
155             (0x2902));
156         [...]
157         VrGlove.getGatt().writeDescriptor(descriptor);
158     }
159 }
```

Listing 5.8: Uzyskanie dostępu do cech serwisu.

```
156 private boolean readCharacteristic(final
157     BluetoothGattCharacteristic characteristic) {
158     if (VrGlove.getGatt() == null) {
159         Log.e(TAG, "ERROR: Gatt is 'null', ignoring
160             read request");
161         return false;
162     }
163     if (characteristic == null) {
164         Log.e(TAG, "ERROR: Characteristic is 'null',
165             ignoring read request");
166         return false;
167     }
168     if ((characteristic.getProperties() & PROPERTY_READ)
169         == 0 ) {
170         Log.e(TAG, "ERROR: Characteristic cannot be
171             read");
172         return false;
173     }
174     VrGlove.getGatt().setCharacteristicNotification(
175         characteristic, true);
176     return true;
177 }
```

Listing 5.9: Ustawienie cechy w trybie powiadomień.

Następnie funkcje klasy *VrGlove* przypisują zmiennym odpowiadającym sensorom nowe dane oraz dokonują ich konwersji z tablicy typu *byte* do typu *float*. Dane te po obróbce są przypisywane odpowiednim polom w interfejsie użytkownika. Pobieranie danych polega na wycinaniu z tablicy informacji kolejnych 4 bajtów, co jest równoznaczne jednej zmiennej typu *float*. Ważne dla konwersji jest również sposób w jaki dane zostają zamienione - w tym przypadku używana jest notacja *LITTLE_ENDIAN*. Wykonane kroki pokazuje listing 5.11 dla odczytu pierwszej wartości z tablicy danych żyroskopu. Proces ten należy wykonać dla wszystkich wartości w danej cęsie oraz odpowiednio dla każdej z pobieranych cech [13].

```

172     if(characteristic.getUuid().equals(convertFromInteger(0
173         x2101))) {
174         VrGlove.setAccReadings(value);
175     } else if (characteristic.getUuid().equals(
176         convertFromInteger(0x2102))) {
177         VrGlove.setGyroReadings(value);
178     } else if (characteristic.getUuid().equals(convertFromInteger(
179         0x2103))) {
180         VrGlove.setFingersReadings(value);
181     } else{
182         Toast.makeText(getActivity(), "Unknown characteristic",
183             Toast.LENGTH_SHORT);
184     }

```

Listing 5.10: Odczytywanie danych cechy.

```

181     static void setGyroReadings(byte[] gyroReadings) {
182         VrGlove.gyroReadings = gyroReadings;
183         getGyroReadings();
184     }
185
186     private static void getGyroReadings() {
187         TextView x = vw.findViewById(R.id.textView_gyr_X);
188         Float[] data = new Float[3];
189
190         float f = ByteBuffer.wrap(gyroReadings, 0, 4).order(
191             ByteOrder.LITTLE_ENDIAN).getFloat();
192         data[0] = f;
193         x.setText(String.format("X:%.2f", f));

```

```
193     [ ... ]  
194  
195     dataSet.put("Gyro", data);  
196     setmIsStateChanged(true);  
197 }
```

Listing 5.11: Przypisywanie danych pozyskanych z cech, do zmiennych w aplikacji.

W tej sekcji został pokazany sposób połączenia aplikacji z kontrolerem, oraz sposób w jaki po ustanowieniu połączenia dane zostają przekazane i obrabiane. W ten sposób fragment *GloveData*, odpowiedzialny za prezentację interfejsu pokazanego na rysunku 5.1a kończy swoje działanie. Na podstawie tych danych fragment *ModelRenderer* jest w stanie wygenerować drugą część interfejsu która zostanie teraz opisana.

5.3 Google Sceneform SDK

Jak widać na zdjęciu interfejsu 5.1b, drugi fragment prezentuje dłoń, której pozycja, orientacja oraz kształt zmienia się w czasie rzeczywistym. Aby to osiągnąć zdecydowano się na wybór SDK udostępnionego od *Google*, służącego do renderowania realistycznych scen w aplikacjach na androida. Pomimo oryginalnego zastosowania służącego do budowania aplikacji AR (z ang. Augmented Reality), zestaw narzędzi został rozbudowany do obsługi aplikacji spoza tej dziedziny. W ten oto sposób, bez znajomości OpenGL otrzymano dostęp między innymi do narzędzi, obsługujących scenę na której renderowane są obrazy, a także wtyczki do Android Studio, pozwalającej na importowanie modeli 3D do aplikacji w prosty sposób. Wtyczkę dodano do środowiska programistycznego poprzez wyszukiwanie w menu *Wtyczki*, wtyczki o nazwie *Google Sceneform Tools (Beta)*. W ten oto sposób po kliknięciu prawym przyciskiem myszy na model znajdujący się w drzewie projektu, pojawia się opcja *zaimportuj model*, tworząca dwa pliki na

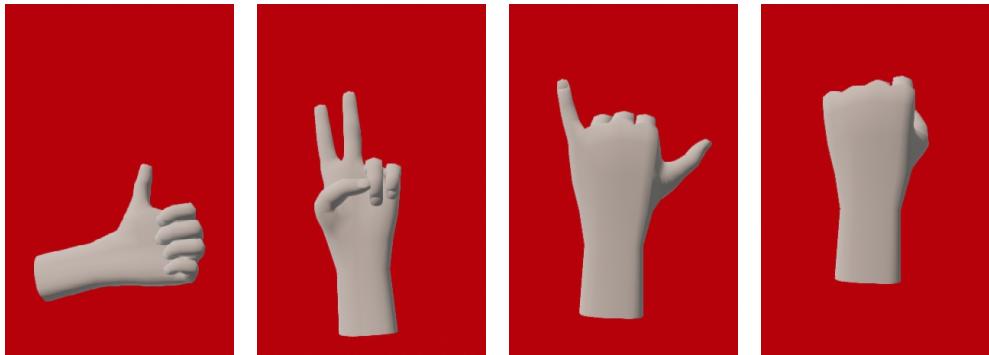
podstawie których *Sceneform* generuje model na ekranie: *.sfa* oraz *.sfb*. Gdy zainstalowano wtyczkę, należy pobrać z oficjalnego linku do strony projektu Github foldery zawierające SDK. Foldery te należy dodać do projektu, które od teraz są jego częścią. Szczegółowy opis jak przeprowadzić ten proces znajduje się w dokumentacji SDK. Ważną informacją dotyczącą tego projektu jest fakt że w Marcu 2020 roku, repozytorium zostało zarchiwizowane, a Google nie przewiduje dalszych prac nad projektem [15].

Aby wygenerować scenę, w momencie utworzenia fragmentu zostają przypisane wysłuchiwacze przycisków akcelerometru, żyroskopu i kalibracji, o czym zostanie więcej powiedziane w dalszych sekcjach tego rozdziału, a także zostaje wywołana funkcja *generateSceneView()*. Funkcja ta jest odpowiedzialna za utworzenie sceny, zadeklarowanej w układzie fragmentu a także przypisaniu do tej sceny początkowego węzła, co osiąga poprzez wywołanie funkcji *renderView()*. Węzeł ten jest wykorzystywany jako punkt zaczepienia dla modelu. Lecz aby opisać etap obsługi węzłów poprzez aplikację, zostanie najpierw pokazane jak węzeł ten reprezentuje dłoń na ekranie.

5.3.1 Model ręki

Model ręki zastosowany w projekcie został stworzony przez 3DHaupt, i jest udostępniony do użytkowanie za darmo w celach edukacyjnych, oraz do celów niekomercyjnych. Model został wykonany w programie *Blender*, i posiada w pełni użytkowy szkielet, dzięki któremu można ustawić dlonie w dowolnej pozycji [3]. Aplikacja obsługuje jedynie rękawice-kontroler przeznaczoną dla lewej dłoni, w związku z czym tylko ta część modelu została zachowana. Korzystając z programu blender, zmodyfikowany szkic został zapisany w formacie *.obj*, a następnie umieszczony w folderze aplikacji o nazwie *smpledata*. Oprócz modelu dłoni z wyprostowanymi palcami, pokazanego na rysunku 5.1b, zostały przygotowane

dodatkowe cztery modele. Modele te zostały pokazane na rysunku 5.2. Tak jak



(a) Model: OK

(b) Model: Pokój

(c) Model: Mahalo

(d) Model: Pięść

Rysunek 5.2: Modele animacji dłoni.

powiedziano wcześniej, do importowania modeli używana jest wtyczka stworzona przez *Google*. Jednak od tego etapu do wyświetlenia modelu na ekranie jest jeszcze kilka kroków które należy wykonać. Przede wszystkim nazwy modeli w aplikacji przechowywane są w tablicy typu *String*. Niestety ten etap nie jest automatyczny, w związku z czym trzeba ręcznie podać nazwę modelu w aplikacji, aby był brany pod uwagę. Po wywołaniu funkcji *renderView()*, zostaje wywołana funkcja *assignModelNames()* której jedynym zadaniem jest przypisanie kolejnym pozycją w tablicy kolejnych nazw modeli z których chcemy korzystać. Nazwy te to pliki wygenerowane przez wtyczkę z rozszerzeniem *.sfb*. Następnie wywoływana jest pętla w której dla każdej nie pustej nazwy w tablicy zostaje wywoływany wzorzec projektowy *builder*, na klasie *ModelRenderable* celu stworzenia finałowej wersji modelu. Modele te po zbudowaniu wykorzystują klasie *CompletableFuture<>* która pozwala wywołać metodę która zostanie wywołana na danym modelu gdy tylko ten zostanie ukończony. Metodą tą jest *onRenderableLoades(ModelRenderable, String)* w której to modele kolejno są dodawane do zbioru przechowywanych modeli w aplikacji. Kod aplikacji 5.12 prezentuje omówione zagadnienia.

```

198     private void generateSceneView() {
199         renderView();
200         assignModelNames();
201
202         for (String s : modelNames) {
203             if (s != null && !s.equals("")) {
204                 ModelRenderable.builder()
205                     .setSource(getContext(), Uri.parse(
206                         s))
207                     .build()
208                     .thenAccept(renderable ->
209                         onRenderableLoades(renderable,s)
210                     )
211                     .exceptionally(
212                         throwable -> {
213                             Log.e("Model", "Unable
214                                 to load Renderable."
215                                 , throwable);
216                             return null;
217                         });
218             }
219         }
220
221         loadStartingModel();
222
223         new Thread (this::startDataListener).start();
224     }
225
226     private void renderView() {
227         sceneView = vw.findViewById(R.id.scene_view);
228         [...]
229         coreNode = new Node();
230         [...]
231         sceneView.getScene().addChild(coreNode);
232     }

```

Listing 5.12: Generowanie sceny i modeli użytych w aplikacji.

Gdy omawiany proces zakończy się, zostaje wywołana funkcja *loadStartingModel()* która jest odpowiedzialna za przygotowanie sceny, zanim dojdzie do połączenia z rękawicą kontrolerem. W funkcji tej zostaje wybrany model który zo-

staje przypisany do sceny jako pierwszy, a dzieje się to poprzez sprawdzanie w nowym wątku dziesięć razy na sekundę czy wybrany model został już wygenerowany przez wzorzec projektowy oraz czy został dodany do zbioru gotowych modeli. Gdy warunek ten zostanie spełniony wywoływana jest metoda *setrenderable(Renderable)* która pozwala na przypisanie modelu do węzła już znajdującego się na scenie. Ostatnim elementem jest wywołanie nowego wątku w którym to ustawiany jest wysłuchiwacz zmian. Funkcja ta pokazana jest na listingu 5.13. Sposób jej działania polega na uruchomieniu nieskończonej pętli w której to są sprawdzane wartości flag zmiany danych. Flagi te są ustawiane jako prawdziwe w przypadku otrzymania nowych danych z kontrolera, co widać w ostatniej linii listingu 5.11. W ten oto sposób są sprawdzane flagi *isCalibrating* co zostanie opisane w sekcji 5.3.5, *ismIsStateChanged()* dla zmian danych pochodzących z IMU z zagnieżdżonymi flagami *renderRotation* oraz *renderPosition*, sprawdzającymi czy użytkownik chce aby brano odpowiednio rotację i pozycję pod uwagę podczas generowania modelu. Działanie aplikacji w przypadku gdy te warunki są spełnione zostaną opisane odpowiednio w sekcjach 5.3.3 oraz 5.3.4. Ostatnią flagą jest flaga pochodząca z klasy *VrGlove* - *ismIsFingersReadings()*, mówiąca czy dane z sensorów palców się zmieniły.

```
228 private void startDataListener() {  
229     while(true){  
230         if(!isCalibrating){  
231             if(VrGlove.ismIsStateChanged()) {  
232                 if(renderRotation){  
233                     [...]  
234                     if(renderPosition){  
235                         [...]  
236                     }  
237                     if(VrGlove.ismIsFingersReadings()) {  
238                         [...]  
239                     }  
240             }  
241         }  
242     }  
243 }
```

Listing 5.13: Wysłuchiwacz zmian w danych rękawicy.

5.3.2 Rozpoznanie i animacja modelu

Jeżeli dane się zmieniły od ostatniego wywołania tej funkcji wartość

`VrGlove.ismIsFingersReadings()` zwróci prawdę co oznacza że wykona się kod rozpoznania i ewentualnej zmiany modelu. Kod ten jest obsługiwany poprzez wywołanie funkcji `replaceModel()` i nie zależy od jej działania, zawsze po jej zakończeniu wywoływana jest linia kodu `VrGlove.setmIsFingersReadings(false);`, dzięki której uzyskano pewność że ten sam zestaw danych nie będzie rozpatrywany wielokrotnie - flaga ta zostanie zmieniona na prawdziwą dopiero gdy nowe dane trafią do aplikacji. Najważniejsze kroki funkcji `replaceModel()` zostaną opisane poniżej.

Powiedziano wcześniej że w celu dodania nowego modelu potrzebna jest jego model, który następnie jest importowany przez plugin a nazwa zostaje dodane do tablicy w której przechowywane są nazwy modeli z rozszerzeniem `.sfb`. Rzeczywiście dzięki temu model znajduje się w ramach aplikacji jednak z perspektywy użytkownika nigdy nie zostanie on zaobserwowany. Dzieje się tak z powodu ostatniego elementu który należy dodać w celu dodania nowego modelu - dla odpowiadającej mu pozycji w tablicy należy dodać tablicę pięciu liczb `-1/0/1`. Tablica ta odpowiednio reprezentuje pozycje palców na modelu który jest dodawany, gdzie 0 oznacza palec zgięty, 1 palec wyprostowany natomiast -1 pozycję pomiędzy tymi wygięciami. Z powodów opisanych w rozdziale 6 nie zaleca się stosowania wartości -1. W ten oto sposób możemy rozpatrzyć nowe dane w kontekście istniejących modeli.

W aplikacji zostały zdefiniowane wartości minimalne oraz maksymalne dla każdego sensorów. Wartości te zostały zczytane dla każdego palca gdy wszystkie palce były wyprostowane, a także gdy każdy z palców po kolejno był zginany tak bardzo jak to było możliwe. Wartości te zostały pobrane eliminując wartości

skrajne, czekając aż odczyty z palców się unormują wokół pewnego zakresu. W trakcie porównywania jest brany pod uwagę margines błędu rzędu ± 30 więc są to jedynie przybliżone wartości. Rezultaty tych wartości prezentuje tabela poniżej:

Palec	Minimalna wartość	Maksymalna wartość
Kciuk	520	680
Wskazujący	400	580
Środkowy	520	670
Serdeczny	500	750
Mały	480	620

Mając do dyspozycji pokazane dane, pierwszym etapem analizy sensorów palców jest sprawdzenie jak odnoszą się one względem tych danych. Dla każdego palca następuje porównanie do wartości minimalnej oraz maksymalnej. Jeżeli wartość sensora odpowiada wartości minimalnej ± 30 zostaje przypisane dla tego palca 0, w przypadku wartości maksymalnej 1 a w pozostałych przypadkach -1. Mając do dyspozycji wartości w jakiej znajduje się obecnie dłoń, oraz wymagania położenia każdego z palców w danym modelu, dokonano porównania tych stanów. Jeżeli rozpoznano model, i model ten nie jest obecnie prezentowanym modelem - funkcja zwraca nowy model który należy wygenerować. Opisany proces pokazuje listing 5.14.

```
241 | for (int i =0; i < fingersReadings.length; i++) {
242 |     if(fingersReadings[i] < sensorsBoundarySettings
243 |         [i][0]+30.0f) {
244 |             pattern[i] = 0;
245 |         }else if(fingersReadings[i] >
246 |             sensorsBoundarySettings[i][1]-30.0f) {
247 |                 pattern[i] = 1;
248 |             }else{
249 |                 pattern[i] = -1;
250 |             }
251 |         for(int[] i : modelRequirements) {
252 |             for (int j = 0; j < i.length; j++) {
253 |                 if(i[j] != pattern[j]){
break;
```

```
254         }
255         m = models[j];
256     }
257     if(m != null) {
258         break;
259     }
260 }
261 if (m != null) {
262     if(m != currentModel) {
263         return m;
264     }
265 }
266 return null;
```

Listing 5.14: Rozpoznanie danych sensorów i wzoru modelu.

Generowanie nowego modelu odbywa się poprzez wywołanie funkcji *assignModelToNode(ModelRenderable)*, rozpoczynając od sprawdzenia czy przekazany model posiada wartość a następnie wywołuję sekwencje zadań na wątku interfejsu. Tak jak powiedziano wcześniej model jest wyświetlany na scenie poprzez połączenie go z węzłem. Jednak aby uniknąć powtarzania tych samych sekwencji oraz zachować płynność obrazu pomiędzy głównym węzłem a modelem został dodany węzeł pośredni. Główny węzeł-rodzic odpowiada za położenie na ekranie a także rotację względem punktu początkowego. Węzeł-dziecko natomiast jest zmieniany w zależności od modelu który należy zaprezentować na ekranie. Sekwencja uruchamiana w interfejsie rozpoczyna się od utworzenia nowego węzła i przypisania mu nowego modelu, który został wybrany do prezentacji użytkownikowi. Następnie dochodzi do usunięcia obecnego modelu poprzez usunięcie ze sceny węzła z który model ten był powiązany a następnie dodanie węzła z nowym modelem. Cały ten proces pokazuje listing 5.15, jednocześnie podsumowując proces obsługi modeli w projekcie.

```
267 Objects.requireNonNull(getActivity()).runOnUiThread(() -> {
268     Node render = new Node();
269     render.setRenderable(modelRenderable);
270     coreNode.removeChild(renderedNow);
271     renderedNow = render;
272     coreNode.addChild(render);
273     replacingInProgress = false;
274 }) ;
```

Listing 5.15: Zamiana węzłów z powiązanymi modelami w interfejsie użytkownika.

5.3.3 Rotacja

W sekcji 5.3.2 pokazano w jaki sposób umiejscowiono główny węzeł na scenie prezentowanej użytkownikowi, a także w jaki sposób modele są przypisywane do węzłów pomocniczych. Tak jak wspomniano, aby model mógł się poruszać dokonywane są zmiany na węźle głównym, które jako efekt końcowy dotyczą również węzłów pomocniczych, czyli z perspektywy użytkownika - modelu dloni. W pierwszej kolejności zostanie pokazane jak dane z IMU zostają zamienione na rotację modelu. Aby rotacja była brana pod uwagę przede wszystkim w interfejsie użytkownika przycisk typu przełącznik musi być w pozycji włączonej - oznacza to że flaga *renderRotation* jest prawdziwa i aplikacja będzie dokonywała obliczeń. W tym celu wykorzystywane są dane zarówno z żyroskopu jak i akcelerometru. W pierwszej kolejności wywoływana jest funkcja *calculateRotation()*, która pobiera aktualnie dostępne dane i o ile dane istnieją dla obu sensorów, zostają wywołane funkcje *calculateAccAngles* oraz *calculateGyroAngles*. Pierwsza z nich odpowiedzialna jest za wyliczenie kątów kontrolera na podstawie akcelerometru. W przypadku akcelerometru dla odczytów X,Y,Z, kąty te mogą zostać wyliczone dla przechyłu bocznego (ang. roll) oraz nachylenia (ang. pitch), przy

użyciu następujących wzorów:

$$Roll = \arctan\left(\frac{Y}{\sqrt{X^2 + Z^2}}\right)$$

$$Pitch = \arctan\left(\frac{-1 * X}{\sqrt{Y^2 + Z^2}}\right)$$

W ten sposób otrzymujemy wartości kątów w radianach. Aby uzyskać kąty w stopniach należy wynik przemnożyć przez $\frac{180}{\pi}$. W ten oto sposób otrzymane kąty z akcelerometru trafiają do funkcji *normalizeAngles*, która upewnia się że kąt zwracany przez *Roll* nie przekroczy ± 85 stopni, aby uniknąć blokady gimbala, a także w przypadku ciągłej rotacji, gdy kontroler wykona pełen obrót, czyli 360° , stopnie te wracają do pozycji początkowej czyli 0° . W ten oto sposób otrzymujemy końcowy efekt dla akcelerometru i wykonuje się funkcja obliczania kątów z danych żyroskopu. Jak stwierdzono w dziale 4.3, zadaniem żyroskopu w tym projekcie jest jedynie wyliczanie kątów, w związku z czym dane żyroskopy zostały zamienione na stopnie już po stronie kontrolera. Oznacza to że funkcja *calculateGyroAngles* jedynie dodaje do aktualnego stanu kątów urządzenia, które w stanie początkowym wynoszą zero, nowo przesłane wyniki z żyroskopu. Kąty te następnie są normowane tak jak w przypadku akcelerometru. Końcową częścią obliczania kątów urządzenia jest zastosowanie filtra komplementarnego na podstawie obliczonych kątów z sensorów. Filtr komplementarny służy jako mechanizm kontrolowania błędu odchylenia gromadzonego przez żyroskop w długim okresie czasu, poprzez zastosowania małej korekty z bardziej dokładnego lecz wolniejszego akcelerometru. W celu skorygowania skrętu (ang. Yaw) używa się magnetometru, jednak w tym projekcie nie wykorzystano tego sensora w związku z tym jedynie dane z żyroskopu są brane pod uwagę. Poprzez ustawienie parametru filtra, można zmienić stopień w jakim akcelerometr koryguje dane - w aplikacji filtr jest ustawiony na 94%, co oznacza że akcelerometr koryguje dane w 6%. Implementacja filtra jest pokazana na listingu 5.16 [12].

```

275 float filterValue = 0.94f;
276 result[0] = filterValue * gyroAngles[0] + (1 - filterValue)
277     * tmp[0];
277 result[1] = filterValue * gyroAngles[1] + (1 - filterValue)
278     * tmp[1];
278 result[2] = gyroAngles[2];

```

Listing 5.16: Implementacja filtru komplementarnego.

W ten sposób kończy się obliczanie rotacji kontrolera według każdej z osi. Zostaje one dopasowana do orientacji dłoni na ekranie poprzez wykorzystanie zamiany kątów podanych w stopniach na kwaternion z wykorzystaniem metody *axisAngle(Vector3, float)*, a następnie w celu określenia rotacji końcowej przemnożenie przez siebie poszczególnych rotacji. W mnożeniu tym ważna jest kolejność wykonywania, dlatego też jako element środkowy musi wystąpić kąt określony według osi X, który został zablokowany w przedziale $\pm 85^\circ$, co nie pozwala na powstanie blokady gimbala. Wynik mnożenia poszczególnych rotacji jest obrotem który należy wykonać, poprzez funkcję *setLocalRotation(Quaternion)*. W ten oto sposób aplikacja obsługuje rotację modelu. Proces ten pokazuje listing 5.17.

```

279 Quaternion[] quat = new Quaternion[3];
280 calculateRotation();
281 quat[0] = Quaternion.axisAngle(new Vector3(0.0f, 0.0f, -1.0f),
282     , modelAngles[0]);
282 quat[1] = Quaternion.axisAngle(new Vector3(1.0f, 0.0f, 0.0f),
283     , modelAngles[1]);
283 quat[2] = Quaternion.axisAngle(new Vector3(0.0f, 1.0f, 0.0f),
284     , modelAngles[2]);
284 Quaternion resultOrientation = Quaternion.multiply(
285     Quaternion.multiply(quat[1], quat[0]), quat[2]);
285 this.coreNode.setLocalRotation(resultOrientation);

```

Listing 5.17: Obrót węzła na podstawie obliczonych kątów.

5.3.4 Przesunięcie

Ostatnim elementem zmieniającym model jest ustalenie zmiany w jego położeniu, tak aby móc odwzorować ten sam ruch na ekranie. Opcja ta gdy jest włączona, ustawia flagę *renderPosition* na prawdziwą pozwalając na obliczanie pozycji. W prezentowanej aplikacji, do tego celu wykorzystywany jest tylko jeden sensor - akcelerometr. Na podstawie tych danych wylicza pozycję kontrolera funkcja *parseAccDataToDisplacement()*, a następnie podobnie jak w przypadku rotacji, na głównym węźle wywoływana jest funkcja która ustawia nową pozycję. Funkcja ta to *setLocalPosition(Vecotr3)*. Aby osiągnąć pozycję kontrolera na podstawie danych z akcelerometru, które są podawane w $\frac{m}{s^2}$, należy wykonać podwójne całkowanie. Dzięki temu najpierw osiągniemy prędkość w $\frac{m}{s}$, a następnie wartość w m . W tym celu podczas otrzymania nowych danych zapisywany jest czas w którym te dane pozyskano, i tak jak w przypadku danych z żyroskopu, dane te są mnożone przez upływ czasu pomiędzy pomiarami. Oczywiście wyniki z akcelerometru podlegają działaniu grawitacji, w związku z czym nie jest to jedynie prędkość poruszania się kontrolera. Wektor grawitacji jest pozyskiwany w trakcie kalibracji urządzenia co będzie opisana w sekcji 5.3.5. Oprócz tego w prezentowanym projekcie mamy do czynienia z kontrolerem który zmienia swoją rotację, co oznacza że również wektor grawitacji zmienia się w zależności od orientacji w przestrzeni względem układu orientacji ziemi. W związku z tym aby osiągnąć prawidłowe pomiary przed całkowaniem danych, należy przywrócić pozyskane dane z orientacji w której się aktualnie znajdują, do orientacji początkowej, usunąć wektor grawitacji który w takiej pozycji pozyskano a następnie przywrócić orientację urządzenia. Dzięki temu dane które zostaną podwójnie scałkowane zwrócią wynik samego przemieszczenia bez dodatkowych sił oddziałujących na akcelerometr. W aplikacji proces ten jest osiągnięty poprzez stworzenie macierzy

rotacji na podstawie obecnej orientacji modelu, dzięki której przemnożono wektor danych pozyskany z akcelerometru. Od wektora wynikowego została odjęta przechowywana grawitacja pozyskana przy kalibracji, a następnie wektor został przemnożony przez odwróconą macierz rotacji, dzięki czemu przywrócono oryginalną rotację. W ten sposób dane zostały scałkowane uzyskując pozycję. Omawiane metody prezentuje listing 5.18 [2] [21] [11].

```
286 final float dT = (currentTime - lastTimestamp) *  
287     dtNanoToSec;  
288 [...]  
289 float[] rotationM = getRotationMatrixFromAngles(tmpAngles);  
290 [...]  
291 float[] accDataPostRotation = removeRotation(accSet,  
292     rotationM);  
293 for (int i = 0; i < accDataPostRotation.length; i++) {  
294     accDataPostRotation[i] -= gravityV[i];  
295 }  
296 float[] invMatrix = invertMatrix(rotationM);  
297 [...]  
298 accDataWithoutG = multiplyMatrix3xVector(invMatrix,  
299     accDataPostRotation);  
300 [...]  
301 for (int i = 0; i < velocity.length; i++) {  
302     velocity[i] += accDataWithoutG[i]*dT;  
303     pos[i] += velocity[i]*dT;  
304 }  
305 [...]  
306 lastTimestamp = currentTime;
```

Listing 5.18: Listing obrazujący obliczanie przemieszczenia kontrolera.

5.3.5 Kalibracja

Do tej pory zostały opisane elementy sterujące animacją modelu, jego pozycją oraz orientacją, a także flagi pozwalające na wyłączenie orientacji oraz położenia. Ostatnim elementem odpowiedzialnym za zmiany na ekranie jest kalibracja modelu z kontrolerem. Dzieje się to poprzez wciśnięcie przycisku FAB na

ekranie. Zmienia on wartość flagi *isCalibrating*, nie pozwalając tym samym na wykonywania nowych akcji na modelu dłoni. Oprócz tego kalibracja jest wykonywana gdy nawiązano połaczenie z kontrolerem oraz gdy użytkownik zmieni pozycję przycisków przełączników odpowiedzialnych za orientację i położenie. W wysłuchiwanym przycisku FAB oprócz zatrzymania zmian na obecnie działającym modelu, resetują się wszystkie zmienne jakie do tej pory zostały pozyskane. Oznacza to że kąty modelu znów ustawione są jako zero wokół każdej osi, zmiana położenie jest liczona od nowa, ustawiony zostaje ponownie model początkowy a także jego pozycja i orientacja zostaje przywrócona do pozycji początkowej. Aby dać na dostosowanie się do tych zmian użytkownikowi, aplikacja wyświetla nowy model po upływie trzech sekund, ciągle wyświetlając przy tym wiadomość na ekranie informującą użytkownika ile czasu mu się pozostało. Podczas kalibracji oczekuje się od użytkownika ustawienia lewej dłoni z wyprostowanymi palcami w taki sposób aby kciuk wskazywał ciało użytkownika a także aby utrzymać tą pozycję bez dodatkowych ruchów. Pozycja ta jest pozycją początkową modelu i to użytkownik musi się do niej zastosować. Gdy tylko kalibracja jest zakończona, pierwszym etapem przed wznowieniem pracy na ekranie jest pobranie aktualnej wartości akcelerometru - wartość ta jest pobierana przy założeniu że użytkownik nie wykonuje żadnych dodatkowych ruchów i jest zapisywana jako wektor grawitacji oddziałujący na model w pozycji kalibracyjnej. W tym momencie pozycja w jakiej się znajduje rękawica-kontrolera, jest uznawana za pozycję wyjściową, od której będą mierzone kąty obrotów wokół osi a także przemieszczenie, a modele będą się zmieniać w zależności od danych z sensorów na palcach. W ten oto sposób zaprezentowano wszystkie elementy i zasady działania aplikacji wykorzystującej do działania dane z rękawicy-kontrolera.

Rozdział 6

Dalszy rozwój projektu

W rozdziałach 4 i 5 pokazano budowę kontrolera oraz aplikacji która wykorzystuje zbudowany kontroler w celu obsługi podstawowych funkcji rękawicy-kontrolera. Projekt ten powstał z myślą ograniczonego budżetu, prostoty wykonania oraz możliwości replikacji. Założenia te spowodowały że zdecydowano się na pewne rozwiązania które w końcowej wersji projektu pokazały swoje wady. W tym rozdziale zostanie poruszony temat błędów popełnionych w pierwszej wersji tego projektu oraz przykładowe sposoby na ich rozwiązanie w przyszłości.

6.1 Problemy mikroprocesora

Przede wszystkim szukano małego mikrokontrolera tak aby nie był on przeszkodą podczas użytkowania kontrolera. O ile założenie to było dobrym pomysłem, okazało się że umiejscowienie przy brzegu sprawiło że ruch palców, w szczególności kciuka, może zmienić położenie jednostki IMU na rękawicy. Oznacza to że nawet jeśli nasza ręka znajduje się w stałej pozycji, samo poruszanie palcami wprowadza błąd w odczycie. Niestety wybór tego produktu od Arduino również przysporzył wiele kłopotów z racji błędnego rozłączania się adaptera Bluetooth. Z dotychczasowego użytkowania można stwierdzić iż kontroler poprawnie łączy się i rozłącza dwa razy, zaś w większości testowanych przypadków dochodzi do błędu połącze-

nia przy trzeciej próbie. Aby usunąć ten błąd należy odłączyć płytę od zasilania i podłączyć ponownie, resetując tym samym moduł. Samo oprogramowanie rękawicy skupia się na odczytach z dwóch sensorów. Praktyka pokazała że dane te nie są wystarczająco dokładne, i jeżeli to możliwe powinny być pobierane również dane magnetometru w celu dodatkowego korygowania odczytów z żyroskopu.

6.2 Problemy budowy i czujników wygięcia

Problem z użytkowaniem rękawicy pojawił się dość szybko od jej zbudowania. Mianowicie wybrana do projektu rękawica była zbyt gruba, powodując dyskomfort w użytkowaniu w szczególności przez dłuższy okres czasu. Początkowe kryterium elastyczności, przesądziło o wybraniu tej rękawicy, jednak cienka rękawica również spełniła by wymagania końcowego produktu. Rozwiążanie zastosowane w celu odczytu wygięcia palców z założenia wyglądało na idealnie pasujące do wymagań projektu. Pomimo swojej prostoty wykonania oraz braku pomiaru takich cech jak odwodzenie palców czy też wygięcie poszczególnych stawów, spełnia ono swoją podstawową funkcję. Problemem tego rozwiązania jest natomiast brak elastyczności sensorów. W momencie zgięcia palców droga od knykci do paznokci się wydłuża sprawiając że sensor jest poddawany sile nacisku od strony palca która jest tym spowodowana. Sensory te pomimo braku elastyczności są zbudowane z materiału wytrzymałego na rozciąganie dzięki czemu nie pękają podczas zgięcia palców, jednak w celu zapewnienia lepszego mocowania i większej ochrony, z jednej strony została przymocowana elastyczna guma która trzyma sensor przy czubkach palców, z drugiej natomiast sensor został wszyty w rękawicę. Problem który się pojawił w trakcie użytkowania pochodził ze sposobu wszycia sensora. Została do tego użyta nić przewodząca która z powodu rozciągliwości rękawicy nie mogła zostać wszyta na sztywno, w związku z czym stawała

ona mniejszy opór podczas zginania palców i niejako została wyciągnięta przez sensor, powodując tym brak dokładności odczytów. Nić ta oprócz niskiej elastyczności okazała się być nietrwała. W trakcie korzystania z rękawicy doszło do kilku pęknięć, które zostały ponownie związane, jednak została przerwana w ten sposób ciągłość obwodu. Przez dodanie dodatkowych wiązań odczyty z sensorów się pogorszyły, sprawiając że wygięcie palca wskazującego ma większy wpływ na odczyty z kciuka, niż zgięcie kciuka samo w sobie. Podobna sytuacja przytrafiła się z sensorem małego palca oraz serdecznego. Mała powierzchnia na dłoni wokół której należało poprawić wiele połączeń, sprawiła że część nici była blisko siebie, powodując momentami odczuwalne mrowienie na dłoni. Problem ten został rozwiązyany poprzez zastosowanie izolacji od wewnętrznej strony rękawicy, jednak nie gwarantuje to przeciwdziałaniu zwarć w obwodzie. Konkludując, nić przewodząca nie jest najlepszym rozwiązaniem w celu połączenia elementów dla tego projektu i powinno zostać zastąpiona trwałszym połączeniem. Gdyby jednak została ona użyta, element przewodzący powinien znajdować się w środku oplotu, bądź powinny zostać zastosowane inne sposoby izolacji, a sama wytrzymałość nici powinna być znacznie większa. Mocowanie sensorów wygięcia powinno być bardziej trwałe oraz statyczne, nie pozwalając na przemieszczenie sensora na palcu. Alternatywą dla tego rozwiązania jest wykorzystanie czujników pomiaru wygięcia opartych o światło nadawane z jednej strony plastikowej tuby oraz miernika natężenia światła z drugiej. W ten sposób wiadomo że im mniejszy pomiar otrzymanywanego światła, tym większe wygięcie tuby, której załamanie blokuje bezproblemowy dopływ światła. Rozwiązanie to również zapewnia pomiary niezniesztalcone poprzez zachowanie innych sensorów a także wygląda na bardziej dokładne [20].

6.3 Animacja modelu

Ostatnim elementem aplikacji dla projektu jest zapewnienie animacji dloni. W tym celu został wykorzystany *Google Sceneform*, dzięki któremu zimportowano modele, ustawiono scenę, przypisano model a także obsługiwano przemieszczenie i orientację. Ostatnim brakującym elementem jest animacja modelu. Według dokumentacji starszej wersji projektu osiągnąć to można poprzez klasę *Skeleton-Node*, pozwalającą na dostęp do kości modelu, bez wykorzystania zewnętrznego programu graficznego. Jednak z niejasnych przyczyn klasa ta została usunięta w ostatniej wersji SDK, powodując brak możliwości wprowadzania zmian w modelu który został zimportowany przy użyciu wtyczki. Problem ten rozwiązano poprzez wykorzystanie programu *Blender*, dzięki któremu można było wyeksportować modele w wyznaczonej pozycji. Aby osiągnąć jednak animację modelu w czasie rzeczywistym, na podstawie dostępnych danych z sensorów wygięcia - cała klasa renderująca fragment 5.1b musi zostać napisana od nowa z wykorzystaniem innej technologii, ponieważ na oficjalnej stronie dystrybucji *Sceneform*, jest napisane iż projekt został zarchiwizowany, w związku z czym taka opcja nie zostanie dodana [15].

6.4 Błąd rotacji

W przypadku rotacji jest wiele sposobów na polepszenie rezultatów. W prezentowanym projekcie wybrano podstawową metodę która wykorzystuje jedynie żyroskop oraz akcelerometr i przy ich użyciu wykorzystuje filtr komplementarny. Tak jak wcześniej wspomniano, aby dokładnie skorygować żyroskop na wszystkich trzech osiach, należy wykorzystać również magnetometr. Oprócz tego istnieje wiele filtrów takich jak Kalmana czy Madgwick'a które skutecznie usuwają szum, a także algorytmy wykorzystujące nowe pomiary w połączeniu z tymi ze-

branymi przed nimi. Możliwości łączenia technik udoskonalania odczytu rotacji z jednostek IMU sprawia, że nie ma jednego najlepszego rozwiązania, a ich wybór jest uzależniony od rodzaju projektu nad którym się pracuje [17].

6.5 Problem obliczania przesunięcia

Obliczenie położenia kontrolera w przestrzeni, niewątpliwie należy do najtrudniejszego problemu w tym projekcie. Sedno problemu tkwi w niedokładności danych. Z powodu wykorzystania metody podwójnego całkowania, błąd użytkowany w cm przy pojedynczej całce, rośnie do m przy całkowaniu podwójnym. Ekran aplikacji jest mierzony w m, a ruch dloni z kontrolerem ma ograniczony zakres długości ramienia. Pomimo tego w niewielkim czasie błąd rośnie do poziomu w którym model znika z ekranu użytkownika. Niestety nie istnieje łatwy sposób na skorygowanie błędów powstałych w wyniku tego algorytmu. Firmy zajmujące się tym problemem dodatkowo umieszczały czujniki pozwalające określić odbiornik urządzenia i ustalić pozycję względem niego, kamery zewnętrzne obserwujące ruch w przestrzeni a także dodatkowe czujniki optyczne. W przypadku rękawicy-kontrolera który może poruszać się we wszystkich kierunkach dodatkowy problem stanowi rotacja, przez którą błąd staje się coraz większy. W przypadku prostej aplikacji nie wykorzystującej zaawansowanych jednostek pomiarowych oraz algorytmów filtrujących, często efekt jaki można osiągnąć tą techniką nie sprawdza się w zastosowaniu, dlatego też dla tej aplikacji domyślnie funkcja ta jest wyłączona [17].

Rozdział 7

Podsumowanie

Przedmiotem działań podjętych w celu stworzenia tej pracy było stworzenie podstawowej wersji kontrolera którego można używać w sposób intuicyjny poprzez samo poruszanie dlonią, w szczególności przeznaczonych do środowisk wirtualnej rzeczywistości. Projekt miał za zadanie przygotowanie podłoża dla dalszych prac, usprawnień a także poznania przykładowych rozwiązań znajdujących się na rynku. Podjęto działanie w celu zrozumienia problemów oraz rozwiązań nawigacji inercyjnej przy użyciu IMU, poznając zasady działania sensorów które jednostka ta obsługuje oraz standardu przesyłania tych danych poprzez adapter Bluetooth Low Energy. Został również zaadresowany problem pomiaru wygięcia palców przy niewielkim budżecie. Wszystkie te elementy zostały złożone w całość tworząc w pełni sprawny kontroler, zdolny do realizacji przedstawionych celów. Kontroler ten niestety ma też wiele wad które sprawiają że wymaga on wielu usprawnień w przyszłości. W celu poznania metod obsługi kontrolera została napisana aplikacja na system Android, pozwalająca wykorzystać stworzone urządzenie w praktyce. Wykorzystując SDK *Sceneform* została przedstawiony model dłoni, reagujący na działanie kontrolera. Dedykowana aplikacja prezentuje wizualną wersję problemów nawigacji inercyjnej, pokazując skalę problemów które trzeba rozwiązać aby osiągnąć satysfakcjonujące efekty. Dzięki zebranym danym,

ustalono błędy wykonane w projekcie oraz przedstawiono przykładowe sposoby na ich rozwiążanie bądź usprawnienie.

Projekt ten pozwolił na wyciągnięcie wielu cennych lekcji dzięki zrozumieniu wszystkich jego elementów. Dzięki temu można stwierdzić że jest on niejako prototypem docelowego projektu, któremu należy poświęcić dużo więcej pracy. Mając jednak do dyspozycji odpowiednie narzędzia i budżet można osiągnąć satysfakcjonujące rozwiązanie nawet w domowych warunkach w szczególności jeżeli poruszanym tematem jest orientacja a także kształt dloni w czasie rzeczywistym. W celu osiągnięcia dokładnego położenia, najlepszym rozwiązaniem pozostaje jak dotąd użycie zewnętrznego systemu do śledzenia położenia. Jeżeli chcemy skorzystać z tego projektu przy użyciu dostępnych na rynku okularów VR, które posiadają w zestawie system namierzania kontrolerów, należałyby jako dodatkowy element zapewnić synchronizację, tworząc tym samym kompletny produkt w domowych warunkach. Jest to niewątpliwie projekt o dużym potencjale, z wciąż rosnącym rynkiem oraz zapotrzebowaniem i zainteresowaniem na tego typu produkty pokazywanym przez wielkie koncerny samochodowe a nawet agencje kosmiczne. Na zakończenie warto dodać że wciąż nie odkryto pełnych możliwości wirtualnej rzeczywistości, co sprawia że praca nad tym projektem jest tak ekscytująca, a na przykładzie tego projektu pokazano że nie jest to ekskluzywny rynek i każdy może w nim wziąć udział w celu zbudowania lepszej i być może wirtualnej przyszłości.

Bibliografia

- [1] *Professional Android Sensor Programming(Chapter 5 and 6)*. wrox, 2012.
- [2] *Dynamics: Theory and Application of Kane's Method (Appendix 1)*. Cambridge University Press, 2016.
- [3] 3DHaupt. My rigged and animated 3d hands. <https://3dhaupt.com/3d-model-anatomy-rigged-hands-low-poly-vr-ar-game-ready-blender/>, 3 2012.
- [4] K. Academy. Voltage divider. <https://www.khanacademy.org/science/electrical-engineering/ee-circuit-analysis-topic/ee-resistor-circuits/a/ee-voltage-divider>, 7 2020.
- [5] Adafruit. Pressure-sensitive conductive sheet. <https://www.adafruit.com/product/1361>, 7 2020.
- [6] Arduino. Language reference. <https://www.arduino.cc/reference/en/>, 7 2020.
- [7] Bluetooth. Specyfikacja. <https://www.bluetooth.com/specifications/>, 7 2020.
- [8] Botland.com. Arduino nano 33 ble - abx00030. <https://botland.com.pl/pl/arduino-seria-nano-oryginalne-ptytki/14758>

-arduino-nano-33-ble-abx00030-7630049201491.html, 7
2020.

- [9] Botland.com. Czujnik ugięcia 112 x 6,3 mm - sparkfun sen-08606. <https://botland.com.pl/pl/czujniki-nacisku/2406-czujnik-ugiecia-112-x-63-mm-sparkfun-sen-08606.html>, 7 2020.
- [10] P. Bugalski. (arduino, co w środku... – część 3 – źródło wbudowanych funkcji). <https://forbot.pl/blog/arduino-co-w-srodku-3-zrodlo-wbudowanych-funkcji-id17291>, 7 2020.
- [11] ChRobotics. Using accelerometers to estimate position and velocity. <http://www.chrobotics.com/library/accel-position-velocity>, 7 2020.
- [12] Dejan. Arduino and mpu6050 accelerometer and gyroscope tutorial. <http://howtomechatronics.com/tutorials/arduino/arduino-and-mpu6050-accelerometer-and-gyroscope-tutorial/>, 7 2020.
- [13] A. Developers. Build anything on android. <https://developer.android.com/>, 7 2020.
- [14] Forbot. Rezystor - co warto wiedzieć. <https://forbot.pl/blog/1eksykon/rezystor>, 7 2020.
- [15] Google. Sceneform overview. <https://developers.google.com/sceneform/develop>, 3 2020.
- [16] D. Heaney. How vr positional tracking systems work. <https://uploadvr.com/how-vr-tracking-works/>, 4 2019.

-
- [17] R. Labs. Um7 content library. <https://redshiftlabs.com.au/support-services/um7-content-library/>, 7 2020.
 - [18] Manus. Manus glove. <https://www.manus-vr.com/>, 7 2020.
 - [19] A. Mouratidis. Writing characteristic property check is always false android bluetoothgatt class. <https://stackoverflow.com/questions/57474710/writing-characteristic-property-check-is-always-false-android-bluetoothgatt-clas>, 10 2019.
 - [20] A. D. Murray. Oprical flex sensors. <https://www.instructables.com/id/Optical-Flex-Sensor/>, 7 2017.
 - [21] M. OpenCourseWare. Multiplication and inverse matrices. https://ocw.mit.edu/courses/mathematics/18-06sc-linear-algebra-fall-2011/ax-b-and-the-four-subspaces/multiplication-and-inverse-matrices/MIT18_06SCF11_Ses1.3sum.pdf, 9 2011.
 - [22] Plexus. Plexus. <https://www.plexus.com/en-us/>, 7 2020.
 - [23] D. Rose. Rotations in three-dimensions: Euler angles and rotation matrices. http://danceswithcode.net/engineeringnotes/rotations_in_3d/rotations_in_3d_part1.html, 2 2015.
 - [24] Russell108. Ble nano 33 does not report or disconnect from central. <https://github.com/arduino-libraries/ArduinoBLE/issues/33>, 5 2020.
 - [25] SM. Arduino lsm9ds3 library. <https://www.arduino.cc/en/Reference/ArduinoLSM9DS1>, 7 2020.

-
- [26] SM. Arduinoble library. <https://www.arduino.cc/en/Reference/ArduinoBLE>, 7 2020.
 - [27] SM. Getting started with the arduino nano 33 ble. <https://www.arduino.cc/en/Guide/NANO33BLE>, 7 2020.
 - [28] Systel. Virtual reality, czyli czym jest wirtualna rzeczywistość? <https://systel.pl/virtual-reality/>, 7 2020.
 - [29] I. Szczecin. Podstawy nawigacji inercyjnej. http://irm.am.szczecin.pl/images/instrukcje/PUN/wyklady/nawigacja_inercyjna.pdf, 7 2020.
 - [30] J. Velasco. Vr controllers: the good, the bad, and the ugly. <https://vrsource.com/vr-controllers-6794/>, 11 20160.
 - [31] E. visualization laboratory. Sayre glove. <https://www.evl.uic.edu/entry.php?id=2162>, 7 2020.