Zachodniopomorski Uniwersytet Technologiczny w Szczecinie

Wydział Informatyki

**Paulina Patrowicz**

nr albumu: 39397

Kierunek studiów: Informatyka

specjalność: Systemy komputerowe i oprogramowanie

forma studiów: stacjonarne

# Implementacja techniki identyfikacji złożonych ruchów wykonywanych podczas użytkowania gogli wirtualnej rzeczywistości

# Identification of complex movements during VR headset usage

Praca dyplomowa inżynierska
napisana pod kierunkiem:
**dr hab. inż. Radosława Mantiuka, prof. ZUT**
w Katedrze Systemów Multimedialnych

Data wydania tematu pracy: 23.09.2019

Data dopuszczenia pracy do egzaminu: .....
(uzupełnia pisemnie Dziekanat)

Szczecin, 2020

# Identification of complex movements during VR headset usage

Paulina Patrowicz
Supervisor: dr hab. inż. Radosław Mantiuk, prof. ZUT

**Abstract**

This engineering thesis concerns the identification, analysis and projection of complex movements, performed in virtual world while wearing VR glasses. The motivation behind this work is the desire to create a simulator of animal movements which does not require any additional controllers. The main assumption is to use only the basic devices included in the flagship VR sets.

The set chosen for the project, HTC Vive VR glasses, is by default equipped with two handheld controllers. The goal of the project is to create an interface between the player using VR glasses and the computer game environment. Body movement of the player are analyzed and reflected by head position sensors and hand-held controllers. The key is to develop an implementation of the algorithm that allows recognizing sequences of movements and classifying them into specific events. The obtained data is then translated into motion in the game.

The algorithm, analyzing in real time the movements of controllers held by the user, classifies them according to the adopted minimum criteria. The criteria can be adapted to each user to make them feel comfortable while making the move. Movements qualified as correct are then translated into simulated game movements, and the player may experience movement akin to flying.

The thesis is divided into four chapters. The first chapter provides an overview of available VR devices as well as applications that integrate them with the development environment. It also presents selected development environment, tools and their specifications. The second chapter presents a method of implementing identification of movements together with descriptions of used elements of the environment. The third chapter presents the final effect of the simulation created, along with the presentation of results and conclusions.

Keywords: virtual reality, VR controllers, 3d movement classification, movement projection, user interfaces, HCI, eye tracker

# OŚWIADCZENIE AUTORA PRACY DYPLOMOWEJ

Oświadczam, że praca dyplomowa inżynierska pn.

Implementacja techniki identyfikacji złożonych ruchów wykonywanych podczas użytkowania gogli wirtualnej rzeczywistości

napisana pod kierunkiem:

dr hab. inż. Radosława Mantiuka, prof. ZUT

jest w całości moim samodzielnym autorskim opracowaniem sporządzonym przy wykorzystaniu wykazanej w pracy literatury przedmiotu i materiałów źródłowych. Złożona w dziekanacie Wydziału Informatyki treść mojej pracy dyplomowej w formie elektronicznej jest zgodna z treścią w formie pisemnej/pisemnej i graficznej*.

Oświadczam ponadto, że złożona w dziekanacie praca dyplomowa ani jej fragmenty nie były wcześniej przedmiotem procedur procesu dyplomowania związanych z uzyskaniem tytułu zawodowego w uczelniach wyższych.

Szczecin, dn. 31.01.2020

Paulina Patrowicz

# Spis treści

**1 Przegląd technik interakcji ze środowiskiem wirtualnej rzeczywistości symulowanym przez gogle VR** **3**
1.1 Możliwe techniki interakcji . . . . . 3
1.2 Pakiety oprogramowania wspierające interakcję . . . . . 5
1.3 Przegląd wybranego zestawu i środowiska . . . . . 7
1.3.1 Zestaw HTC Vive . . . . . 7
1.3.2 Konfiguracja stanowiska ze sterownikiem SteamVR . . . . . 10
1.3.3 Środowisko Unity . . . . . 11

**2 Technika identyfikacji złożonych ruchów wykonywanych za pomocą kontrolerów VR** **13**
2.1 Identyfikacja ruchów wykonywanych przez kontrolery . . . . . 13
2.2 Identyfikacja ruchów rejestrowanych kamerami eye tracker . . . . . 16
2.3 Implementacja . . . . . 18
2.4 Podsumowanie . . . . . 21

**3 Testowanie opracowanej techniki na przykładzie gry komputerowej** **22**
3.1 Gra Pigeon Run . . . . . 22
3.2 Opis procedury testowej . . . . . 24
3.3 Dyskusja i wnioski . . . . . 25

**4 Podsumowanie** **28**

**A Poster z konferencji CGI 2019 (Conference on Game Innovations 2019)** **30**

**B Skrypty zaimplementowane w projekcie** **32**

# Wprowadzenie

Niniejsza praca inżynierska dotyczy identyfikacji, analizy oraz odwzorowania w wirtualnym świecie złożonych ruchów wykonywanych podczas użytkowania gogli wirtualnej rzeczywistości. Motywacją do stworzenia pracy stała się chęć stworzenia symulatora naśladującego poruszanie się zwierząt bez konieczności korzystania z dodatkowych kontrolerów. Głównym założeniem było wykorzystanie jedynie bazowych urządzeń wchodzących w skład flagowych zestawów do wirtualnej rzeczywistości.

Wybranym zestawem zostały gogle wirtualnej rzeczywistości HTC Vive, które domyślnie posiadają dwa kontrolery trzymane w dłoni. Celem projektu jest stworzenie interfejsu pomiędzy graczem używającym gogle VR a środowiskiem gry komputerowej. Gracz wykonywać będzie ruchy ciałem, które rejestrowane czujnikami położenia głowy oraz trzymanych w rękach kontrolery będą analizowane i odzwierciedlane. Kluczowym jest opracowanie implementacji algorytmu umożliwiającego rozpoznanie sekwencji ruchów i ich zidentyfikowanie jako określone zdarzenia, a następnie przełożenia uzyskanych danych na ruch w grze.

Algorytm analizując w czasie rzeczywistym ruchy kontrolerów trzymanych przez użytkownika, dokonuje ich klasyfikacji według przyjętych minimalnych kryteriów. Kryteria mogą być dostosowywane odpowiednio pod każdego użytkownika, aby czuł się komfortowo wykonując ruch. Ruchy zakwalifikowane jako prawidłowe przekładane są następnie na symulowany ruch w grze, a gracz może doznać efektu lotu.

Treść pracy zawiera się w czterech rozdziałach. Rozdział 1 stanowi przegląd dostępnych urządzeń VR a także zastosowań integrujących je ze środowiskiem deweloperskim. Prezentuje również wybrane środowisko deweloperskie, narzędzia oraz ich specyfikację. Rozdział 2 przedstawia metodę realizacji identyfikacji ruchów wraz z opisami wykorzystanych elementów środowiska. Rozdział 3 przedstawia finalny efekt stworzonej symulacji wraz z prezentacją rezultatów i wniosków. Rozdział 4 podsumowuje zrealizowaną pracę.

Słowa kluczowe: wirtualna rzeczywistość, kontrolery VR, identyfikacja ruchów 3d, odwzorowanie ruchów, interfejs użytkownika, HCI, eye tracker

# Rozdział 1

# Przegląd technik interakcji ze środowiskiem wirtualnej rzeczywistości symulowanym przez gogle VR

W tym rozdziale omówione zostaną możliwe techniki interakcji przy wykorzystaniu dostępnych na rynku urządzeń wspierających wirtualną rzeczywistość. Uwaga skupiona zostanie na kategoriach klasyfikacji urządzeń oraz ich rodzajach ze wskazaniem stosowanych w nich metodach detekcji ruchów. W dalszym etapie opisane będą pakiety oprogramowania umożliwiające bądź wspierające pracę z urządzeniami wirtualnej rzeczywistości, a także dokonane będzie uzasadnienie wyboru technologii wykorzystywanej dla dalszego przeprowadzania pracy.

## 1.1 Możliwe techniki interakcji

Z bazowym urządzeniem fizycznym niezbędnym do pracy z *wirtualną rzeczywistością* (ang. virtual reality - VR) najczęściej kojarzony jest hełm tzw. *headset* bądź gogle - okrojona wersja. Są one bardzo ważne, jednak do zapewnienia detekcji ruchów sam hełm nie wystarczy. Rozważmy wykorzystanie wirtualnej rzeczywistości jednie dla zastosowań wizyjnych czyli wyświetlania obrazów. Na rynku dostępnych jest wiele zastosowań takich jak:

- Cardboard - bardzo tanie gogle, najczęściej tekturowe, przeznaczone do samodzielnego złożenia po wydrukowaniu szablonu, bądź kupieniu gotowego zestawu z plastikowymi soczewkami. Złożony Cardboard do pełni funkcjonalności potrzebuje telefonu typu smartfon, ponieważ jest on jedynie możliwym do założenia na głowę stelażem, służącym do zamocowania wyświetlacza. Aby skorzystać z opcji jakie daje VR niezbędne jest pobranie na smartfon specjalnych aplikacji, które dostosują wyświetlany obraz do soczewek oraz wykorzystując żyroskop bądź akcelerometr pozwolą na obracanie obrazu o 360° zgodnie z ruchem głowy [Car19]. Dzięki prostocie oraz niskiemu kosztowi Cardboarda, dostęp do technologii VR w takiej formie posiada każdy zainteresowany użytkownik smartfonów.

- Dedykowane gogle do smartfonów (np. Samsung Gear VR) - solidnej konstrukcji gogle tworzone przez liderów rynku. Zasada ich działania jest identyczna jak w przypadku Cardboard'a, z małym wyjątkiem - dedykowne urządzenia bardzo często posiadają przyciski funkcyjne, pozwalające na wchodzenie w drobną interakcję z zamontowanym jako wyświetlacz urządzeniem. Dodatkowo producenci zapewniają specjalne oprogramowania do obsługi gogli, bardzo często zawierające biblioteki filmów oraz mini gier, w pełni sterowalne przyciskami funkcyjnymi [Sam19].

Wyżej wymienione urządzenia stworzone zostały do oglądania obrazów wyświetlanych z efektem 360°, wyposażone są najczęściej w akcelerometry, żyroskopy bądź czujniki zbliżeniowy, ale nie pozwalają one jednak na wejście w pełną interakcję ze względu na brak kontrolerów. Drugim rodzajem urządzeń są *sety* (ang. zestawy) zawierające oprócz hełmu również specjalnie kontrolery, pozwalające wejść w interakcję ze światem wirtualnym. W zależności od producenta, kontrolery mają różne możliwości i logikę działania. Najpopularniejsze urządzenia:

- PlayStation VR - zestaw VR stanowi w tym przypadku dodatek do konsoli gier, dlatego do interakcji ze światem wykorzystywany jest dedykowany konsolowy kontroler Dualshock, bądź wymyśle kontrolery Move, sprzedawane w różnych kształtach. Zarówno hełm jak i kontrolery mogą być sparowane kablem z konsolą, bądź podłączone bezprzewodowo wykorzystując *technologię Bluetooth*. Do badania pozycji hełmu i kontrolerów Move niezbędna jest specjalna kamera PlayStation Camera, śledząca przy pomocy podczerwieni czujniki znajdujące się w urządzeniach. Kamera wyposażona jest w dwa obiektywy co daje jej możliwość wykrywania głębi 3D, a co za tym idzie - potrafi śledzić pozycję i orientację gogli VR [Pla19].

- Oculus Rift - przewodowy hełm kompatybilny z komputerami, posiadający w zestawie dwa bezprzewodowe kontrolery Oculus Touch. Hełm cechuje się bardzo dobrą jakością *trackingu* (śledzenia) ponieważ wyposażony jest w wiele *sensorów podczerwieni*. Sensory te są zwykłymi kamerami z filtrami, które widzą tylko światło podczerwone. Aby wykryć położenie kontrolerów zbierają one z otoczenia dane, w tym emitujące fale diody LED kontrolerów, a następnie wysyłają je do komputera, który przetwarza sygnały i identyfikuje pozycję każdej diody, tym samym uzyskując względną pozycję każdego obiektu. Dla zwiększenia jakości trackingu możliwe jest dokupienie dodatkowego sensora Oculus Sensor. Kontrolery to dwa małe urządzenia - lewe i prawe, mieszczące się w dłoni, będące pod stałą detekcją gogli. Zawierają trzy przyciski, touchpad oraz joystick. Dzięki umieszczeniu wszelkich sensorów w hełmie, zestaw może być wykorzystywany w każdej przestrzeni [Ocu19].

- HTC Vive - przewodowy hełm kompatybilny z komputerami, zawierający w bazowym zestawie dwa bezprzewodowe kontrolery oraz stacje bazowe. Do pełni działającego trackingu hełm i kontrolery wykorzystują dwie stacje bazowe, które muszą być umiejscowione na stałe w wysokim miejscu i pod odpowiednim kątem. Urządzenia działają tylko w strefie detekcji czujników podczerwieni stacji bazowych. Nieruchome lasery stacji skanują obszar 100 razy na sekundę w

celu śledzenia czujników fotonicznych na hełmie i kontrolerach dzięki czemu zapewniają one największą dokładność oraz brak obaw o *okluzję* (przysłanianie się czujników) spośród wszystkich urządzeń dostępnych na rynku [Viv19b]. Kontrolery wyposażone są w przyciski, touchpad, trigger (przypominający spust broni) i grip (aktywowany ściśnięciem dłoni). Dodatkowo, niektóre wersje hełmu wyposażone zostały w kamery pozwalające na *eyetracking* - śledzenie ruchu gałek ocznych.

Ze względu na największy wachlarz możliwości jakie dają kontrolery oraz najlepszą precyzyjność detekcji ruchów, do zrealizowania pracy wykorzystałam zestaw HTC Vive z opcją eyetrackingu. Na Rys. 1.1 zaprezentowane są urządzenia wchodzące w skład zestawu HTC Vive: w centrum znajduje się hełm, po bokach na górze stacje bazowe, a na dole bezprzewodowe kontrolery.

Rysunek 1.1: Zestaw HTC Vive (źródło [Viv19b])

## 1.2 Pakiety oprogramowania wspierające interakcję

Posiadając wybrane urządzenie fizyczne kolejnym krokiem niezbędnym do rozpoczęcia prac jest zadbanie o sterowniki odpowiadające za prawidłowe funkcjonowanie setu, a także narzędzia deweloperskie umożliwiające tworzenie aplikacji. Bazowym oficjalnym sterownikiem wydanym przez producenta jest Vive Setup, aplikacja instalacyjna, konfigurująca poszczególne elementy sprzętu z oczekiwaniami użytkownika. Alternatywę do powyższego sterownika stanowi SteamVR, system do obsługi wirtualnej rzeczywistości stworzony przez firmę Valve i to on zostanie wykorzystany w dalszej pracy. SteamVR stanowi interfejs pozwalający na uruchomienie platformy VR czyli podłączenie, rozpoznanie i obsłużenie urządzeń takich jak hełm czy kontrolery, nie tylko firmy Vive. Dodatkowo jest również interaktywną platformą startową dla doświadczeń wirtualnej rzeczywistości, dzięki aplikacji SteamVR Home, zawierającej sporą bibliotekę darmowych mini gier, aplikacji i multimediów. Najważniejszym jednak aspektem jest fakt,

iż dzięki skorzystaniu ze SteamVR możliwym staje się prowadzenie dewelopmentu z wykorzystaniem VR, gdyż wymagany jest on w większości narzędzi deweloperskich.

Podstawowym narzędziem deweloperskim, które jest niezbędne do rozpoczęcia pracy to *silnik graficzny* (zestaw kodów odpowiedzialny m.in. za przetwarzanie i renderowanie obrazu w czasie rzeczywistym). Na rynku dostępnych jest wiele gotowych silników graficznych wyposażonych w rozbudowane edytory i możliwości rozszerzania, w związku z tym do stworzenia aplikacji wykorzystującej VR pod lupę wzięte zostały dwa darmowe silniki:

- Unity3d - silnik stworzony z myślą o małych firmach i twórcach niezależnych tworzących niskobudżetowe aplikacje. Posiada darmową licencję personalną z możliwością jej rozszerzenia w celu uzyskania dodatkowych funkcjonalności. Obsługa silnika odbywa się poprzez prosty edytor, bazujący na komponentach oraz kodach pisanych w języku C#. Środowisko jest bardzo łatwo rozszerzalne o nowe funkcjonalności poprzez dodatkowe biblioteki czy *pluginy* (małe programy rozszerzające funkcjonalność), udostępniane bardzo często za darmo przez innych twórców. Unity umożliwia tworzenie aplikacji na platformy desktopowe, mobilne, konsole a także różne urządzenia typu smart (np. telewizory). Dodatkowo, producenci silnika uruchomili specjalny sklep Asset Store, w którym każdy twórca może podzielić się z innymi swoimi rezultatami za darmo bądź odpłatnie. W celu uzyskania pomocy udostępniona została również przez twórców szczegółowo prowadzona i na bieżąco aktualizowana dokumentacja, która w porównaniu z ogromną społecznością użytkowników dzielących się wiedzą, jest tylko maleńkim wycinkiem[PON19].

- Unreal Engine - silnik stworzony na potrzeby gry "Unreal" a następnie "Unreal Tournament" przez Epic Games. Na początku dostępny tylko dla dużych twórców, od niedawna dostępny z pełną funkcjonalnością dla każdego za darmo. Podstawowa obsługa silnika odbywa się poprzez *Blueprinty* czyli wizualny system skryptowy, pozwalający na kodowanie "z bloczków", bez konieczności programowania, ale aby skorzystać z bardziej zaawansowanych opcji niezbędne jest pisanie kodów w języku C++. Niestety społeczność i dokumentacja Unreal nie są tak bogate jak w przypadku Unity3d, a za większość rozszerzeń trzeba zapłacić, jednak silnik ten pozwala osiągnąć dużo lepszą jakość grafiki i wydajność niż konkurent. Podobnie jak Unity3d jest multiplatformowy jednak ze względu na wysoki próg wejścia i małą ilość darmowych rozwiązań zalecany jest dla dużych wielko-budżetowych twórców [PON19].

Po wykonaniu przeglądu tych dwóch silników, do przeprowadzenia pracy wykorzystany zostanie silnik Unity3d, ze względu na niższy próg wejścia do dewelopmentu oraz preferencje autora co do języka programowania czyli C#. Dodatkowo, dla Unity udostępnione zostały darmowe pluginy do obsługi VR - VRTK, oraz eyetrackingu - Pupil Lab, rekomendowane i wspierane przez ogromną społeczność użytkowników. Pluginy te, które niżej zostały szczegółowiej zaprezentowane, również zostaną wykorzystane w pracy.

- VRTK (*Virtual Reality Toolkit* ang. Zestaw narzędzi do Wirtualnej Rzeczywistości) - to zbiór gotowych rozwiązań pozwalających na szybką i prostą integrację

urządzeń VR z silnikiem Unity. Dzięki komplecie skryptów i metod umożliwia budowanie aplikacji z obsługą wirtualnej rzeczywistości przy pomocy gotowych obiektów. Plugin zapewnia odzwierciedlenie kontrolerów fizycznych na wirtualnej scenie projektu, a także szereg implementacji obsługujących: lokomocję w przestrzeni wirtualnej, interakcje z obiektami na scenie oraz interfejsem (dotykanie, chwytanie, przesuwanie), fizykę ciał oraz dodatkowe predefiniowane prefabrykaty gotowe wchodzić w interakcję z kontrolerami. VR Toolkit dostępny jest w pełni za darmo jako paczka skryptów i prefabrykatów do zaimportowania w silniku Unity. W pracy wykorzystany zostanie do zinterpretowania w świecie wirtualnym kontrolerów trzymanych przez użytkownika w dłoniach.

- Pupil Lab - platforma dedykowana analizie i detekcji operacji śledzenia oczu oraz badań wizyjnych. Firma Pupil Labs wytwarza zarówno sprzęt jak i oprogramowanie realizujące wyżej wymienione założenia. Ze względu na wyposażenie wykorzystywanego do pracy setu, o rozszerzenie HTC Vive Add-On, czyli dwóch kamer Pupil Cam niezbędne będzie skorzystanie z oprogramowania dedykowanego przez producenta - hmd-eyes. Oprogramowanie jest pakietem pluginów, możliwych do zaimportowania w silniku Unity. Składa się z komponentu Pupil Service, czyli serwera sieciowego Pupil Lab, komunikującego się komponentem Pupil Capture śledzącego wzrok, komponentem Calibration System odpowiadającego za startową za kalibrację kamer, komponentem Pupil Manager zarządzającego połączeniem sieciowym i wiele innych.

## 1.3 Przegląd wybranego zestawu i środowiska

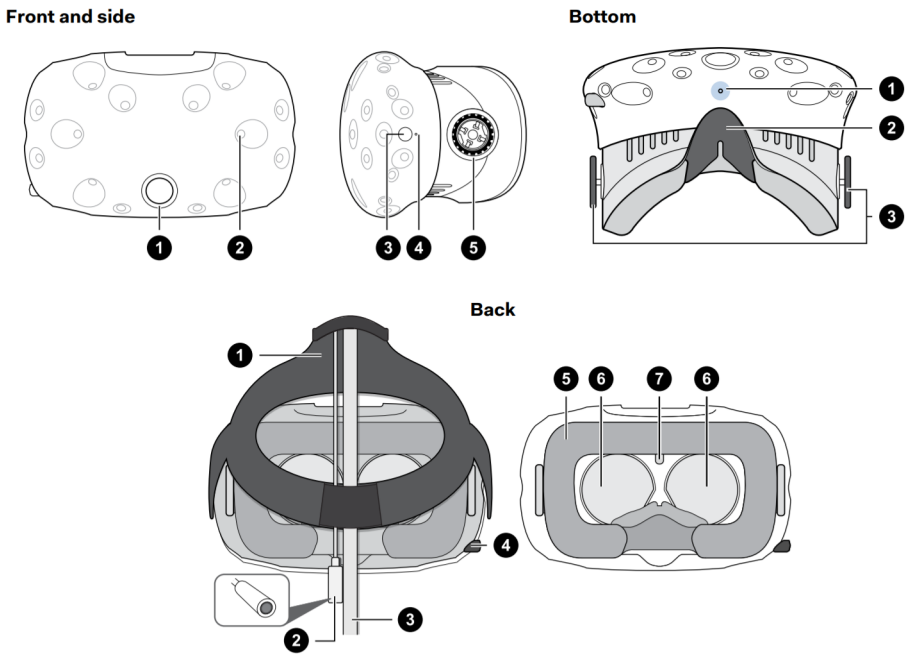
Do przeprowadzenia pracy wybrany został sprzęt HTC Vive z rozszerzeniem eyetracking firmy Pupil Lab. Do stworzenia implementacji wykorzystany zostanie silnik Unity z kodami pisanymi w języku C#.

### 1.3.1 Zestaw HTC Vive

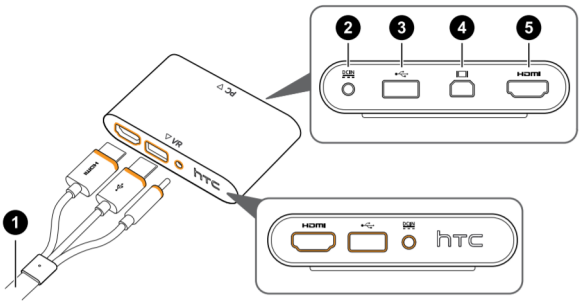
Na poniższych rysunkach Rys. 1.1-5 oraz tabelach Tab. 1.1-2. zaprezentowane zostały szczegółowe informacje dotyczące wybranego zestawu gogli VR. Tabela Tab. 1.1 prezentuje specyfikację zestawu HTC Vive natomiast Tab. 1.2 jego minimalne wymagania sprzętowe. Rysunki: Rys. 1.2, Rys. 1.3, Rys. 1.4 i Rys. 1.5 to kolejno schematy gogli, linkboxa, kontrolerów i stacji bazowych wchodzących w skład zestawu.

Tabela 1.1: Specyfikacja zestawu HTC Vive [Viv19b]

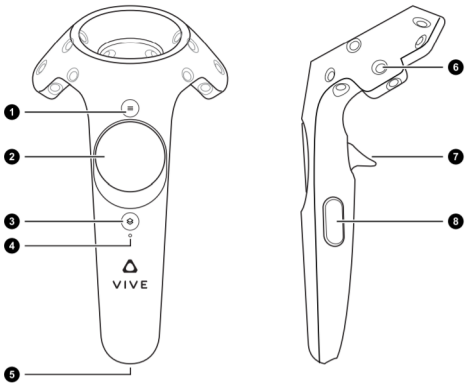
| | |
|---|---|
| **Headset** | |
| Wyświetlacz | dwa ekrany OLED, rozdzielczość 2160 x 1200 (1080 x 1200 na każde oko), pole widzenia 110°, częstotliwość odświeżania 90Hz |
| Sensory | SteamVR Tracking, G-sensor, żyroskop, czujnik zbliżeniowy |
| Porty I/O | HDMI, USB 2.0, stereo Jack, port ładowania, Bluetooth |
| **Kontrolery** | |
| Przyciski | wielofunkcyjny Touchpad, przycisk Grip aktywowany ściskiem, dwustopniowy, przycisk Trigger w kształcie spustu, przyciski systemowe, przycisk menu |
| Sensory | SteamVR Tracking |
| Porty I/O | port ładowania |
| **Stacje bazowe** | |
| Sensory | SteamVR Tracking, zasięg 7m, pole widzenia 160° x 115° |
| Porty I/O | port ładowania, złącze synchronizujące, microUSB |
| **Linkbox** | |
| Porty I/O | złącze ładowania, USB, mini DisplayPort, HDMI |



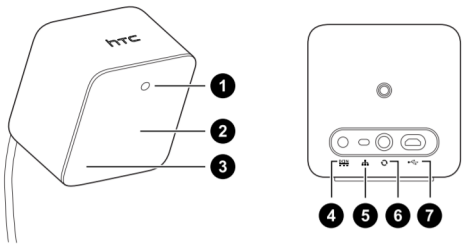

Rysunek 1.2: Schemat gogli HTC Vive [Viv19a]:
- Front and side, przód: 1 - kamera obiektywu , 2 - sensory śledzenia, 3 - przycisk hełmu, 4 - dioda statusu, 5 - pokrętło regulacji dystansu obiektywu.
- Bottom, dół: 1 - mikrofon, 2 - przestrzeń na nos, 3 - pokrętło odległości obiektywów.
- Back, tył: 1 - pasek, 2 - kabel audio, 3 - kabel 3 w 1, 4 - pokrętło regulujące rozstaw oczu, 5 - poduszka, 6 - obiektywy, 7 - czujnik zbliżeniowy.




Rysunek 1.3: Schemat łącznika LinkBox [Viv19a]: 1 - kabel 3 w 1, 2 - port zasilania, 3 - port USB, 4 - port Mini DisplayPort, 5 - port HDMI.




Rysunek 1.4: Schemat kontrolera [Viv19a]: 1 - przycisk Menu, 2 - touchpad, 3 - przycisk systemowy, 4 - dioda statusu, 5 - port micro-USB, 6 - sensor śledzenia, 7 - przycisk Trigger, 8 - przycisk Grip.




Rysunek 1.5: Schemat stacji bazowej [Viv19a]: 1 - dioda statusu, 2 - panel przedni, 3 - wskaźnik kanału, 4 - port zasilania, 5 - przycisk kanału, 6 - port kabla synchronizującego, 7 - port micro-USB.

Tabela 1.2: Zalecane wymagania sprzętowe do pracy z zestawem HTC Vive [Viv19a]

| Wymagania | |
|---|---|
| Grafika | NVIDIA® GeForce® GTX 1060 or AMD Radeon™ RX 480 lub lepsza |
| Procesor | Intel® Core™ i5-4590 or AMD FX™ 8350 lub lepsza |
| Pamięć | 4 GB RAM lub więcej |
| Wyjście wideo | HDMI 1.4, DisplayPort 1.2 lub nowszy |
| Porty USB | 1x USB 2.0 lub nowszy |
| System operacyjny | Windows® 7 SP1, Windows® 8.1, Windows® 10 |

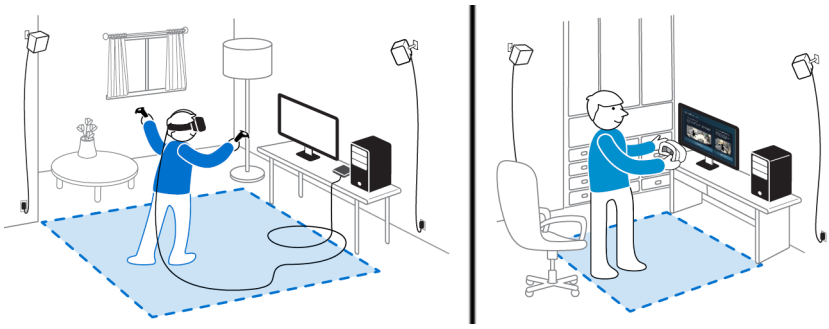
Komputer wykorzystywany do implementacji oraz uruchomienia środowiska VR spełnia powyższe wymagania.

### 1.3.2 Konfiguracja stanowiska ze sterownikiem SteamVR

Po podłączeniu zestawu i zainstalowaniu sterowników niezbędne jest przeprowadzenie konfiguracji stanowiska. Hełm może być użytkowany w dwóch trybach: statyczny (stojący / siedzący), dynamiczny (tryb pokoju). Wybór trybu zależy od ilości wolnej przestrzeni przeznaczonej na stanowisko do użytkowania wirtualnej rzeczywistości. W przypadku naszej pracy, nie będzie istniała potrzeba przemieszczania się po pokoju, dlatego też wybrany został tryb statyczny. Porównanie i prezentacja trybów została przestawiona poniżej na Tab. 1.3 i Rys. 1.6.

Tabela 1.3: Wymagania strefy śledzenia [Viv19a]

| Wymagania | |
|---|---|
| Tryb stojący / siedzący | Brak wymaganej przestrzeni |
| Tryb pokoju | Minimalna przestrzeń: 2m x 1.5m<br>Maksymalna przestrzeń: 3.5m x 3.5m |



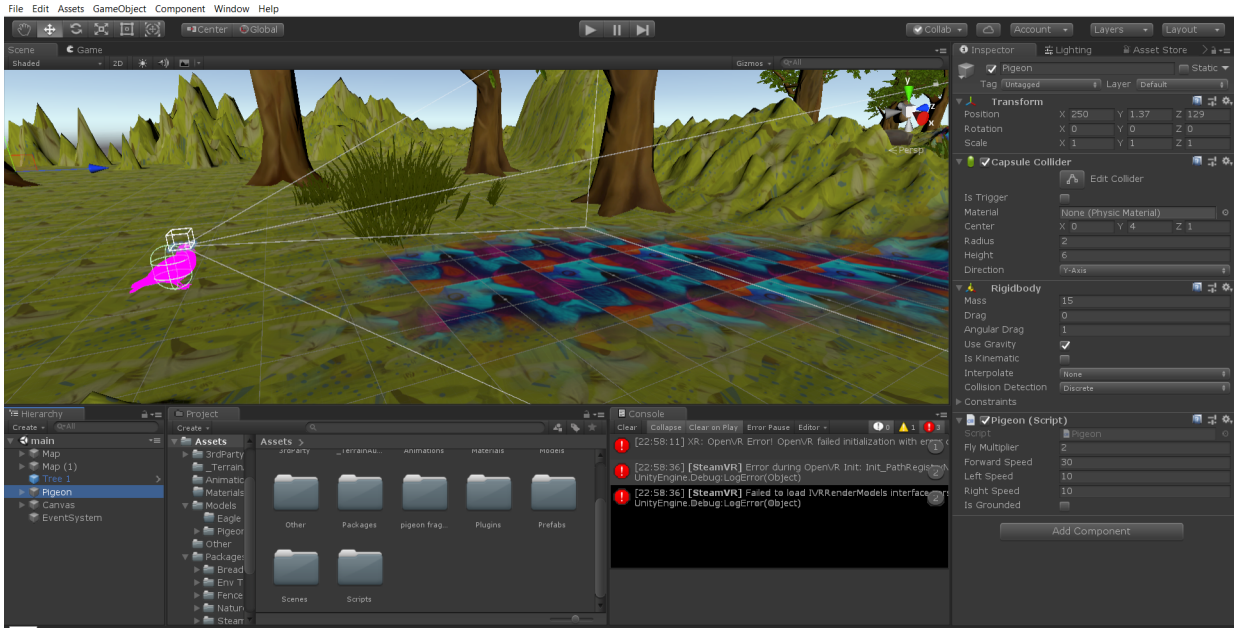
Rysunek 1.6: Przykład konfiguracji pokoju: z lewej strony tryb dynamiczny, z prawej strony tryb statyczny [Viv19a].

### 1.3.3 Środowisko Unity

Środowisko Unity obsługuje się przy pomocy Edytora (Rys. 1.7). Edytor składa się z wielu sekcji służących do zarządzania edytorem. Bazowymi sekcjami są:
- Scena - okno prezentujące widok całej sceny aplikacji włącznie z wszelkimi jej elementami, umożliwia poruszanie się po scenie a także manipulowanie obiektami.
- Gra - widok z aktualnie aktywnej kamery umieszczonej na scenie, który widoczny jest po uruchomieniu aplikacji.
- Hierarchia - lista obiektów znajdujących się na scenie, umożliwia ich dodawanie, usuwanie i manipulacje atrybutami.
- Inspektor - okno atrybutów obiektów, tzw. komponentów.
- Konsola - miejsce wypisywania informacji typu debug oraz komunikatów błędów.

W silniku Unity każdy element aplikacji jest obiektem. Każdy obiekt posiada określone atrybuty definiujące jego rodzaj, styl i zachowanie zwane komponentami. Z obiektów można tworzyć prefabrykaty będące kopiami obiektów ze zmodyfikowanymi wartościami komponentów. Bazowym komponentem, który zawiera każdy obiekt niebędący elementem UI (ang. User Interface, Interfejs Użytkownika) jest Transform. Komponent ten reprezentuje macierz transformacji, określającą położenie danego obiektu w scenie gry i będzie on kluczowym komponentem w implementacji rozwiązania problemu. Inne podstawowe komponenty wykorzystane w pracy:
- Camera - pozwala na interpretowanie obiektu jako kamery, z której będzie można pobierać obraz
- Mesh Renderer, Mesh Filter - tworzy bazową strukturę każdego obiektu posiadającego model
- Script - samodzielnie stworzony skrypt napisany w języku C#
- Collider - dodaje kolizję do obiektu o określonym kształcie obszaru kolizyjnego
- Rigidbody - moduł symulujący fizykę
- Animation - dodaje możliwość animowania obiektów
- Audio Listener, Audio Source - moduły obsługujące emisję dźwięku
- Rect Transform - wariacja komponentu Transform wykorzystywana w UI
- Canvas, Canvas Scaler, Canvas Renderer - moduł renderujący UI

Dodatkowo dzięki wykorzystaniu pluginu VR Toolkit, do projektu dostarczone zostały gotowe prefabrykaty obsługujące identyfikację, integrację oraz odwzorowanie kontrolerów trzymanych w dłoniach przez użytkownika a także hełmu w świecie gry.




Rysunek 1.7: Zrzut ekranu z edytora Unity

# Rozdział 2

# Technika identyfikacji złożonych ruchów wykonywanych za pomocą kontrolerów VR

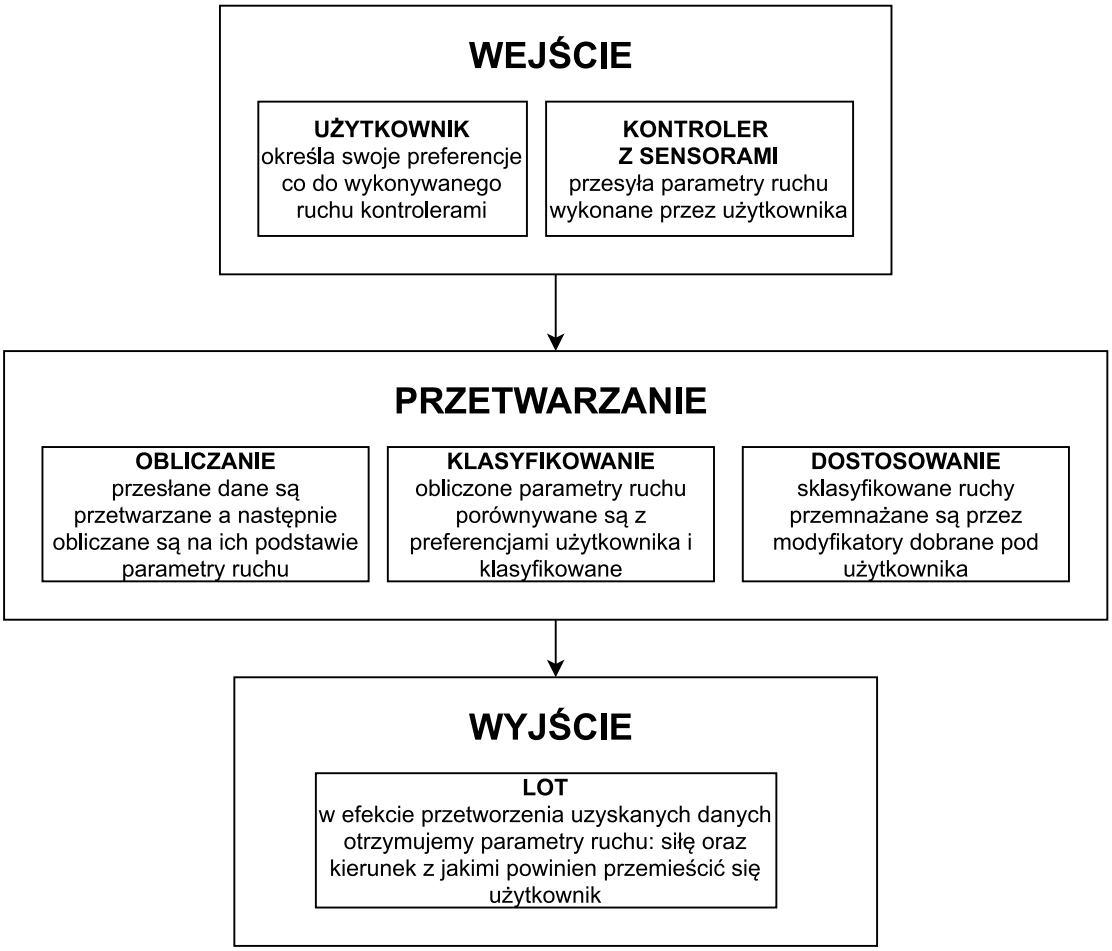
W tym rozdziale zaprezentowane zostaną schematy metod wykrywających ruch i sposób ich implementacji z wykorzystaniem silnika Unity oraz dodatkowych bibliotek wspierających wirtualną rzeczywistość.

## 2.1 Identyfikacja ruchów wykonywanych przez kontrolery

Celem projektu było opracowanie symulatora pozwalającego na wykonywanie w wirtualnym świecie ruchów naśladujących poruszanie się zwierząt, bez konieczności korzystania ze specjalnych kontrolerów. Pomysł ten stał się motywacją do powstania interfejsu między graczem używającym gogle VR a środowiskiem gry komputerowej. W celu uzyskania go, zaimplementowany został algorytm umożliwiający rozpoznanie sekwencji ruchów wykonywanych ciałem, rejestrowanych przez czujniki położenia głowy oraz trzymane w rękach kontrolery. W rezultacie przy wykorzystaniu silnika Unity3d powstała gra, wykorzystująca jako główną mechanikę detekcję ruchu machania rękami, odzwierciedlającą machanie skrzydłami ptaka w świecie gry. Dzięki zastosowaniu odpowiednich pluginów nasze fizyczne kontrolery widnieją w świecie gry jako obiekty na scenie, z określoną macierzą transformacji. Algorytm na jej podstawie analizuje zmianę pozycji Y kontrolerów i dopasowuje ją do wzorca wykonanego ruchu. Przy pomyślnym dopasowaniu ruchu, postaci w grze zostaje nadana siła o wektorze adekwatnym do wykonanego ruchu, dzięki czemu nasz ptak może się poruszać i założenie symulacji zostaje spełnione.

Głównym założeniem tej metody była możliwość zinterpretowania danych otrzymanych od kontrolerów a następnie przetworzenia ich w taki sposób by uzyskane wyniki wykorzystać do sterowania graczem. Ogólny schemat metody zaprezentowany został na Rys. 2.1 i prezentuje etapy działania metody. Na wejściu metody posiadamy użytkownika i kontrolery, które przekazują dane do programu, ten dokonując obliczeń, klasyfikacji i dostosowania wartości zwraca informację, z jaką ma zostać przemieszczony obiekt. W efekcie uzyskujemy efekt lotu gdy użytkownik wymachuje rękami, w których trzyma

kontrolery.




Rysunek 2.1: Schemat działania metody identyfikującej ruchy kontrolerów

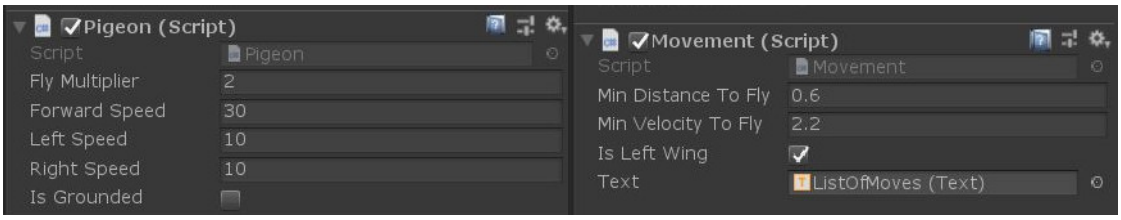
Sprzęt zastosowany przy realizacji projektu opisany został w Rozdziale 1, ciąg dalszy obecnego przedstawi szczegółowo implementację detekcji i interpretacji ruchów. Szczegółowy schemat obrazujący działanie projektu załączony został jako Dodatek A.

Bazowymi ruchami, na którego odwzorowaniu skupia się opisywany projekt jest machanie przez ptaka skrzydłami. Dla uproszczenia, machnięcie skrzydłem reprezentowane jest przez użytkownika jako uniesienie i opuszczenie ręki. Dzięki wykonywaniu tego ruchu w odpowiedniej czasowo sekwencji, użytkownik jest w stanie wznieść się w powietrze niczym ptak machający skrzydłami.

Głównym źródłem danych o wykonywanym przez użytkownika ruchu jest trzymany w dłoni kontroler, a także nałożony na głowę hełm. Użytkownik posiada dwa kontrolery, każdy trzymany w innej dłoni, które oznaczone będą później jako lewy i prawy kontroler. Te trzy czujniki odwzorowane są w środowisku Unity jako wirtualne obiekty, umiejscowione w trójwymiarowym świecie, opisywane macierzą transformacji. Dzięki analizie zmiany położenia obiektów z czujnikami, jesteśmy w stanie uzyskać wartości pozwalające na utworzenie odpowiedniej siły unoszącej główny obiekt gracza.

## Implementacja metody identyfikacji ruchów kontrolera

W przekazywaniu danych o wykonywanym ruchu uczestniczą użytkownik i kontroler. Użytkownik na starcie musi określić swoje preferencje co do prędkości i zasięgu z jakimi jest w stanie machać rękami, następnie te wartości przekazywane są do skryptu przetwarzającego (Rys. 2.2). W dłoniach użytkownika leżą dwa kontrolery, których zmiana położenia jest nieustannie śledzona przez stacje bazowe i przesyłana w postaci macierzy transformacji do Unity. Przesłana macierz transformacji przekładana jest na pozycję, rotację i skalę kontrolera w świecie wirtualnym. Dodatkowo, gdy posiadamy już skalibrowany ruch, możliwe jest dodanie do siły lotu modyfikatora dostosowanego pod preferencje użytkownika, który pozwoli na lepiej odczuwalny efekt lotu.




Rysunek 2.2: Fragment inspektora Unity z podglądem skryptów, do których przekazywane są parametry gracza.

Dane uzyskane od kontrolerów pozwalają na obliczenie takich parametrów ruchu jak: pozycja początkowa, pozycja końcowa, czas początkowy, czas końcowy. Wartości te pozwalają na określenie różnicy czasów, położeń a także prędkości i kierunku z jakimi ruch został wykonany. Kluczową informacją, jaką uzyskujemy od kontrolera jest również czy leży on w prawej czy lewej dłoni, ponieważ w zależności o strony tworzymy wektor siły z kierunkiem obróconym przeciwnie. Zastosowanie odwrócenia kierunku pozwala na bardziej realne odwzorowanie efektu machnięcia skrzydłem i przemieszczenia się na jego wskutek, ponieważ zgodnie z prawami fizyki machając skrzydłem lewym, napotkamy opór, który przesunie nas w prawą stronę. Analogicznie sytuacja ma się do prawego skrzydła.

Gdy posiadamy obliczone parametry ruchu niezbędnym jest wykonanie klasyfikacji i odrzucenia tych ruchów, których wartości nie mieszczą się w minimalnych kryteriach spersonalizowanych pod użytkownika: minimalny zasięg i minimalna prędkość machnięcia. Takie podejście pozwala nam na uniknięcie uznania przypadkowych ruchów za ruch lotu, a także dopasowanie wysiłku do możliwości kondycyjnych użytkownika. Sklasyfikowany ruch poddawany jest przemnożeniu przez modyfikatory, które również mają sprawić, że odwzorowany ruch będzie lepiej odebrany przez użytkownika.

Wynikowy wektor ruchu opisany jest wzorem:

$$\overrightarrow{F} = \overrightarrow{v}.\text{forward} * \text{forwardForce} + \overrightarrow{v}.\text{right} * \text{leftForce} + \overrightarrow{v}.\text{up} * \text{upForce} * \text{flyMultiplier}$$

gdzie:
$\overrightarrow{F}$ - wektor wynikowy,
$\overrightarrow{v}$ - wektor z punktem początkowym w puncie (0,0,0) gracza, a dopiski: forward, right, up to kierunki wektora kolejno znaczące: przód, prawo, góra,

$x$Force - skalary modyfikujące siłę w zależności od umieszczonego przedrostka $x$: forward, right, up znaczących kolejno: przód, prawo, góra
flyMultiplier - skalarny ogólny mnożlinik lotu

Wzór na siłę lotu przy zamachnięciu prawym skrzyłem posiada taką samą postać jednak odwrócić należy wektor skierowany w prawą stronę a także zmienić modyfikator siły leftF na right: ...$-\overrightarrow{v}$.right * rightForce...

Na wyjściu metody otrzymujemy wektor siły z kierunkiem, o który przesunąć ma się obiekt główny użytkownika. Wektor ten za każdym razem skierowany jest na wprost względem lokalnej osi obiektu z odchyłami na boki w zależności, którym skrzydłem został wykonany ruch. W efekcie, gdy użytkownik macha rękami z odpowiednią częstotliwością, wznosi się w górę, a gdy przestaje machać, opada zgodnie z zasymulowaną siłą grawitacji.

## 2.2 Identyfikacja ruchów rejestrowanych kamerami eye tracker

Hełm HTC Vive w posiadanej wersji wyposażony jest w kamery śledzące ruch gałek ocznych. Stąd dodatkowo w ramach projektu opracowany został system interakcji z wirtualnym światem poprzez skupianie wzroku na obiektach. Ogólny schemat metody zaprezentowany został na Rys. 2.3. Na wejściu do metody eyetracker przekazuje dane o zmianie położenia gałek ocznych użytkownika, następnie metoda przetwarza uzyskane wartości i wykrywa element otoczenia, na którym wzrok skupia użytkownik. Jeżeli pomyślnie uda się wykryć obiekt obserwowany metoda zwraca go, co umożliwia wejście z nim w interakcję.

Rysunek 2.3: Schemat działania metody identyfikującej ruchy gałek ocznych

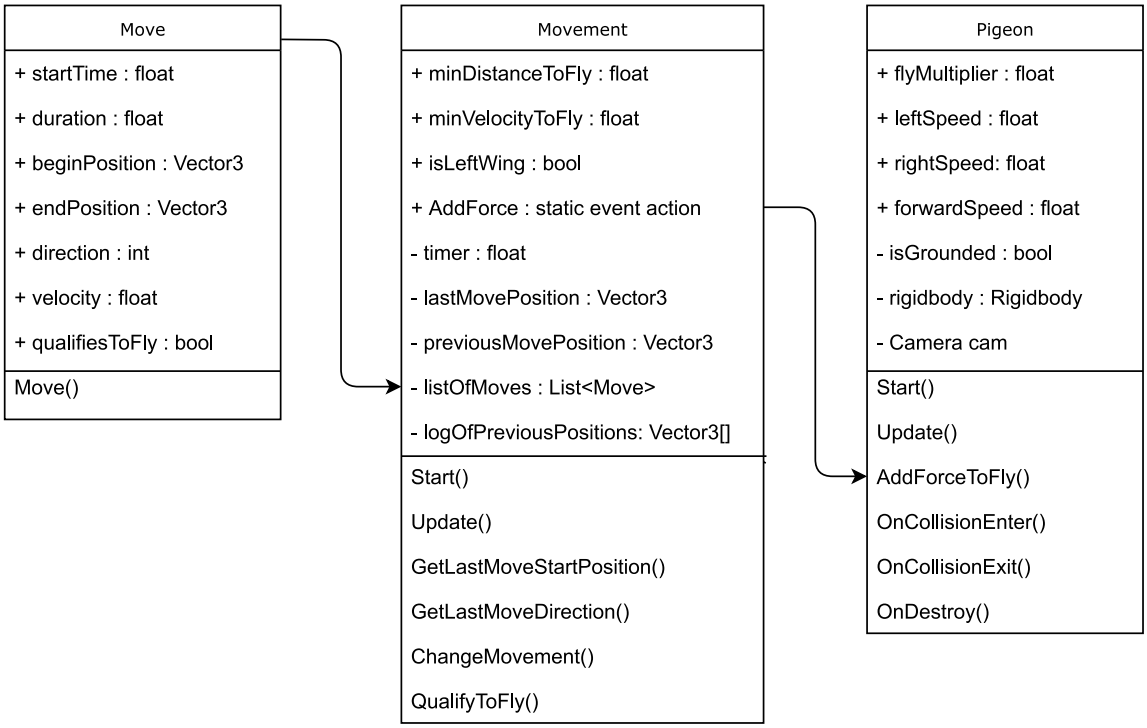
## Implementacja metody identyfikacji ruchów gałek ocznych

Kamery eye trackera stale rejestrują oczy użytkownika, uzyskując dane, które dzięki wykonaniu przez użytkownika kalibracji mogą być analizowane i interpretowane. Do Unity dzięki wykorzystaniu biblioteki Pupil Labs przekazywany jest punkt w przestrzeni 2d, który nakładamy na obraz kamery. Dzięki oznaczeniu punktu patrzenia na kamerze jesteśmy w stanie z tego miejsca wysłać promień wychodzący przed siebie od kamery do określonego maksymalnego punktu. Promień ten może wejść w kolizję z otoczeniem posiadającym odpowiednie właściwości, co pozwala na wykrycie na jaki element świata patrzy obecnie gracz. Jeżeli kolizja z obiektem jest wystarczająco długa, możemy uznać, że użytkownik skupia na tym obiekcie wzrok. W przypadku przeprowadzanej symulacji, detekcja punktu patrzenia posłużyła jako interakcja z otoczeniem podczas lotu. Użytkownik lecąc, może poprzez skupienie wzroku na modelach jedzenia zebrać je i zwiększyć swój pasek energii, niezbędnej do lotu. Mechanika ta została zaprezentowana na Rys. 2.4.

Rysunek 2.4: Zrzut z podglądu kamery gry. Na obrazku widać promień reprezentujący wzrok użytkownika i punkt jego skupienia.

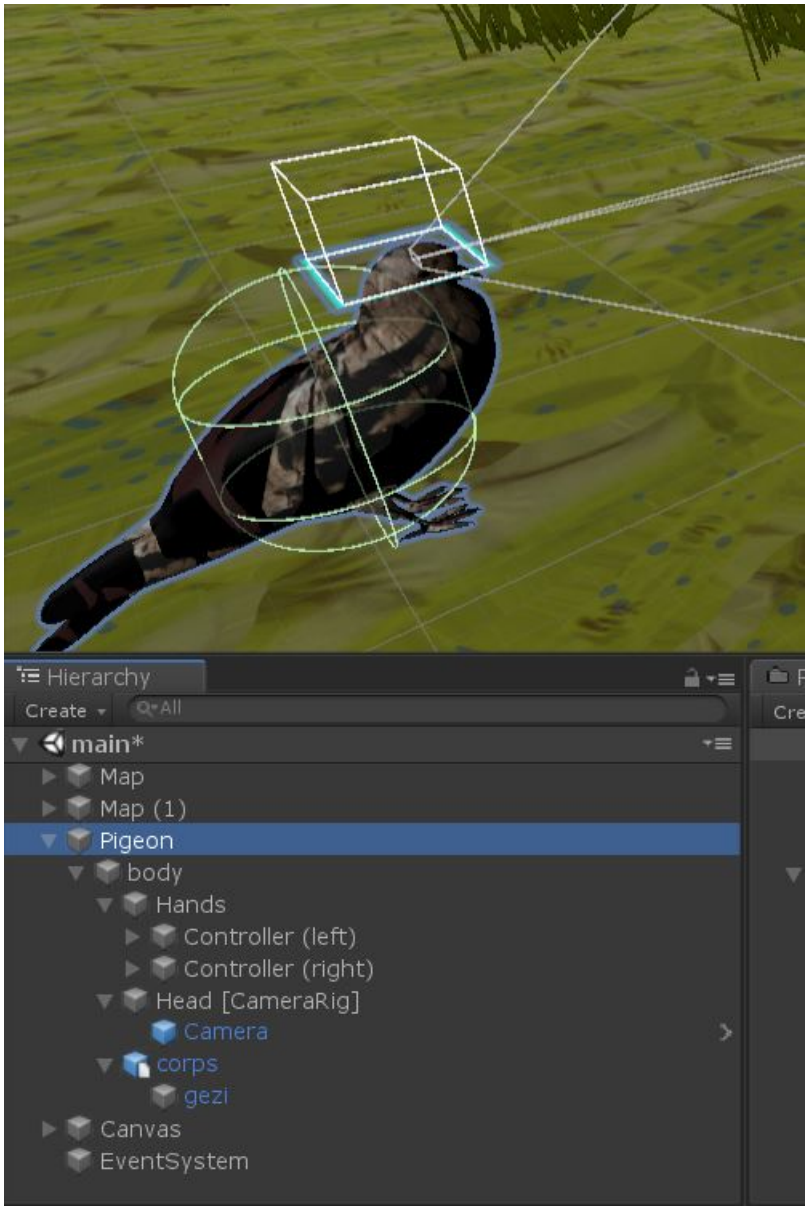
## 2.3 Implementacja

Implementacja metod bazuje na trzech klasach: Movement i Move (zawierających się w skrypcie Movement) oraz Pigeon, zaprezentowanych na Rys. 2.5 i dołączonych w całości jako Dodatek B. Klasa Movement jest pierwszym ogniwem, jeżeli chodzi o detekcję rozpoczętego przez użytkownika ruchu. Odpowiada ona za przyjęcie od kontrolerów informacji o zmianie ich położenia oraz przetwarzanie uzyskanych danych. Algorytm dąży do zebrania informacji o kompletnym pojedynczym ruchu i analizuje go tak długo, aż się nie zbierze wszelkich danych. Wykonany ruch reprezentowany jest przez obiekty klasy Move, specyfikującej właściwości ruchu, gdzie za początek i koniec ruchu uznawana jest dwukrotna zmiana kierunku wykonywanego machnięcia (metoda ChangeMovement()). Klasa Movment po uzyskaniu kompletnego obiektu typu Move dokonuje jego kwalifikacji w metodzie QualifyToFly() - sprawdza, czy zebrane o nim parametry spełniają minimalne wymogi dostosowane pod użytkownika. Jeżeli nie, to ruch pozostaje nieużyty, jeżeli tak to ruch przekazywany jest jako delegat do metody AddForceToFly() do Klasy Pigeon. Klasa Pigeon zarządza głównym obiektem gracza i reaguje na zgłaszane do niego akcje. Odpowiada za wchodzenie gracza w kolizję z otoczeniem jak i z obiektami wykrywanymi przez eye tracker (metody OnCollisionEnter() i OnCollisionExit()). Otrzymuje również informację o tym, ze został wykonany ruch uznany za prawidłowy i należy nadać graczowi siłę o odpowiednim wektorze.




Rysunek 2.5: Diagram klas zaimplementowanych skryptów.

W środowisku Unity spis wszystkich elementów Sceny znajduje się w Hierarchii, determinującej kolejność renderowania obiektów. Sposób ułożenia w niej obiektów, pozwala nie tylko na określenie sposobu renderowania, lecz również na manipulowanie punktami odniesienia obiektów względem siebie. Umożliwia to mechanizm dziedziczenia obiektów typu dziecko przez rodziców. W przypadku naszej sceny zaprezentowanej na Rys. 2.6 w hierarchii znajdują się obiekty: "Map" będące obiektami kontenerami zawierającymi jako dzieci konkretne elementy mapy gry (drzewa, krzewy, trawę, elementy środowiska); "Pigeon" czyli nasz główny gracz, który szczegółowo opisany zostanie niżej; "Canvas", kontener zawierający całe "UI", wyświetlające się w kamerze gracza; "EventSystem", obiekt niezawierający żadnych podobiektów, odpowiadający za komunikację elementów wysyłających i odbierających zdarzenia.




Rysunek 2.6: Podgląd hierarchii Sceny

Głównym obiektem gracza jest Pigeon, czyli widoczny na scenie ptak, w którego wciela się użytkownik. Pigeon składa się z "body", czyli ciała zawierającego niżej wymienione elementy:

- Hands - dłonie
  - Controller (left) - reprezentacja lewego kontrolera
  - Controller (right) - reprezentacja prawego kontrolera
- Head - reprezentacja hełmu
  - Camera - główna kamera z widokiem z oczu gracza
- corps.. - model obiektu, tekstury itp.

Każdy ruch ręką (Hands/Controller) interpretowany jest względem pozycji lokalnej, w której punktem (0,0,0) jest centrum głównego obiektu Pigeon (dla ułatwienia w tym punkcie umiejscowiony został hełm), dzięki czemu mamy pewność, że przesunięcie całego obiektu nie zniekształci danych sczytywanych z kontrolera. Główny obiekt natomiast interpretowany jest względem pozycji globalnej na całej scenie gry i jego punktem (0,0,0) jest punkt początkowy całej sceny, w tym przypadku to jedna z początkowych krawędzi mapy. Użytkownik wykonując ruch porusza całym obiektem gracza, zawierającym w sobie czujniki. Dzięki czemu, jeżeli zmieni się pozycja obiektu Pigeon, to pozycja kontrolera i głowy pozostanie nadal niezmienna.

## 2.4 Podsumowanie

W efekcie wykorzystania opisanych w powyższym rozdziale skryptów udało się uzyskać efekt lotu, który w zależności od prędkości i siły machania konkretną ręką daje inny rodzaj ruchu. Dokonano także wejścia w interakcję z elementami środowiska przy wykorzystaniu jako kontroler wzroku. Problemem okazało się jednak znalezienie odpowiednich wartości, odpowiadających większości użytkowników, które przełożą się na wygodę lotu. Koniecznym było znalezienie balansu między siłą potrzebną na machanie rękami, by wznieść ptaka, a faktycznym efektem lotu, tak aby użytkownik był na siłach spędzić nad naszą symulacją dłużej niż kilkadziesiąt sekund. Upublicznienie zmiennych opisujących wartości kwalifikujące ruch do lotu oraz modyfikatorów lotu ułatwiło to, gdyż możliwym stało się dostosowanie wartości pod każdego użytkownika podczas trwania rozgrywki. Wykorzystując opisywane w rozdziale metody udało się opracować grę będącą symulatorem lotu, której szczegółowy opis znajduje się w Rozdziale 3.

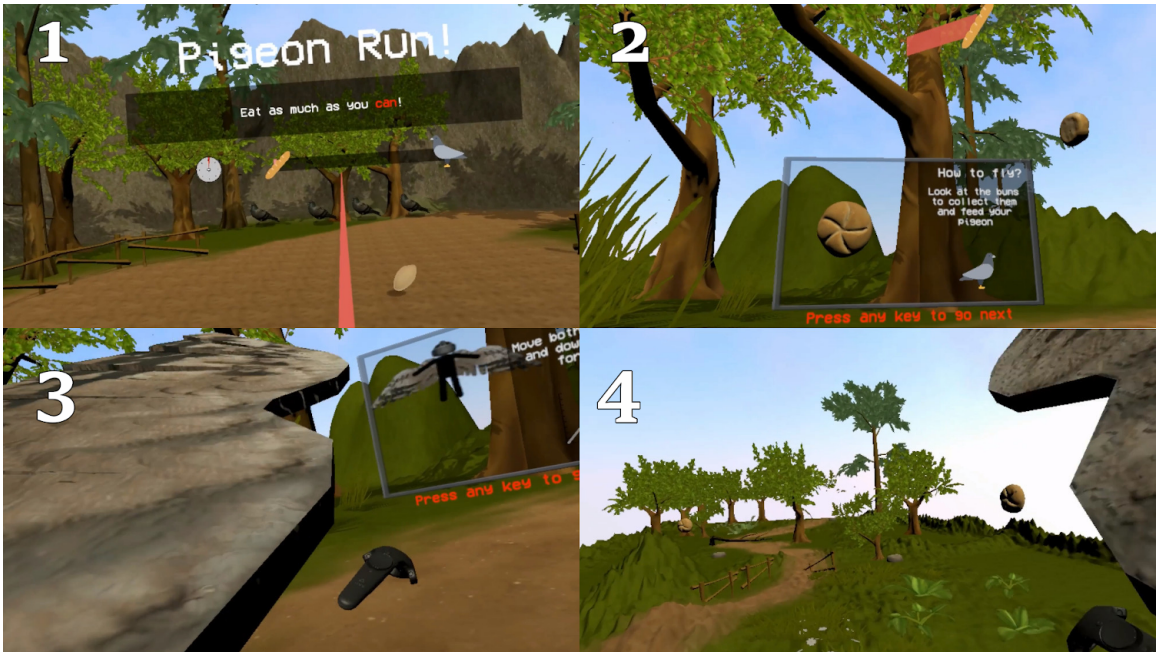
# Rozdział 3

# Testowanie opracowanej techniki na przykładzie gry komputerowej

W tym rozdziale zaprezentowana zostanie gra wykorzystująca detekcję, identyfikację i odwzorowanie ruchów machania rękami. Omówione zostaną problemy wynikające ze skorzystania z wirtualnej rzeczywistości, a także przestawione wnioski z przeprowadzonego projektu.

## 3.1 Gra Pigeon Run

W wykorzystaniu opracowanych implementacji metod, opisywanych w Rozdziale 2 powstała gra Pigeon Run, będąca symulatorem lotu. W grze użytkownik wciela się w gołębia i jego zadaniem jest przemieszczenie się do punktu końcowego mapy gry. Gracz wykorzystując mechanikę lotu musi poruszać się określoną ścieżką, unikając elementów środowiska gry: drzew, konarów, krzewów. Aby lecieć, konieczne jest posiadanie naładowanego paska z energią, która zużywa się z każdym machnięciem skrzydła. Energię zdobyć można poprzez zjadanie bułeczek lewitujących w powietrzu, w które należy wycelować wiązkę promienia idącego od oczu gracza. Pasek energii naładować można również w czasie mini gry rozgrywanej przed rozpoczęciem sceny lotu, polegającej na zjadaniu ziaren, zanim zrobią to inne ptaki. Gdy gracz spojrzy na bułeczkę bądź ziarno dłużej niż sekundę, zostaje ono zebrane i zamienione w energię lotu. Gracz rozpoczyna rozgrywkę z naładowanym częściowo paskiem energii. Jedna rozgrywka trwa średnio 3 minuty, im lepiej gracz jest wprawiony w sterowanie tym łatwiej i szybciej ona przebiega. Poniżej na Rys. 3.1-4 zaprezentowane zostały zrzuty z gry, prezentujące rozgrywkę. Na Rys. 3.1 widać przekrój opracowanej grywalności, na zrzucie pierwszym i drugim pokazany jest początkowy widok scen: mini gry z jedzeniem ziaren i tryb latania, natomiast na zrzucie trzecim i czwartym zaprezentowane są skrzydła i widok z oczu gracza podczas lotu. Na Rys. 3.2 zaprezentowano etapami funkcjonalność zebrania obiektu przy wykorzystaniu implementacji identyfikującej ruchy gałek ocznych. Na Rys. 3.3 i Rys. 3.4 widać poklatkowo kolejno prezentację poruszanych skrzydeł i efekt upadku na ziemię z zatrzęsieniem kamery.




Rysunek 3.1: Zrzut z gry Pigeon Run. Prezentacja grywalności.

Rysunek 3.2: Zrzut z gry Pigeon Run. Wykorzystanie eye trackera.

Rysunek 3.3: Zrzut z gry Pigeon Run. Prezentacja ruchu skrzydeł.

Rysunek 3.4: Zrzut z gry Pigeon Run. Prezentacja zetknięcia z podłożem.

## 3.2 Opis procedury testowej

Na pomyślność rozgrywki bardzo duży wpływ miała parametryzacja, która różni się w zależności od preferencji danego użytkownika. Parametryzacji poddane zostały trzy elementy:

1. Wysiłek wymagany do wykonania prawidłowego ruchu
   (a) minimalny zasięg machania ręką (różnica położeń między skrajnymi punktami podczas wykonywania ruchu)
   (b) minimalna szybkość machania ręką (wyliczony z różnicy położeń i czasu trwania ruchu)
2. Przełożenie ruchu na siłę lotu ptaka (skalary przez, które przemnażany jest cały wektor ruchu)
   (a) ogólny modyfikator siły lotu
   (b) siła lotu do przodu
   (c) siła lotu w lewo
   (d) siła lotu w prawo
3. Grawitacja (modyfikatory komponentu Rigidbody)
   (a) opór
   (b) masa ptaka

W zależności od siły i wydajności użytkownika, możliwym było dostosowanie wartości, tak aby odczucie lotu było jak najbardziej komfortowe. Testując grę na wielu użytkownika udało się wybrać uniwersalną parametryzację, która dodatkowo była zmieniana dla każdego użytkownika przed grą Rys. 3.5. Użytkownicy pytani byli o:
- Czy obecny zasięg poruszania rękami jest odpowiedni?
- Czy obecny prędkość machania rękami jest odpowiednia?
- Czy poruszasz się w swoim odczuciu obecnie za szybko / za wolno?

- Czy odpowiednia jest według Ciebie siła wznoszenia i opadania?
- Czy nie kręci Ci się w głowie i nie czujesz dyskomfortu podczas obracania głowy?
Po czym na bieżąco odpowiadania, w czasie rzeczywistym modyfikowane były wartości w celu znalezienia tych najlepszych. Poniżej w Tab 3.1 zaprezentowane zostały uśrednione wartości oraz przedziały wartości, które zostały wybrane przez użytkowników jako najbardziej komfortowe.

Tabela 3.1: Wartości wzorcowe wybrane przez użytkowników

| **Wartości wzorcowe [jednostki wirtualne]** | | |
|---|---|---|
| 1a) | min zasięg machania | (0.3, 0.8) |
| 1b) | min szybkość machania | (1.8, 2.4) |
| 2a) | ogólny modyfikator siły | 2 |
| 2b) | siła w przód | 30 |
| 2c) | siła w lewo | 10 |
| 2d) | siła w prawo | 10 |
| 3a) | opór | 0 |
| 3b) | masa | 15 |

Rysunek 3.5: Fotorelacja z konkursu Zespołowego Tworzenia Gier Komputerowych 2019 w Łodzi, na której zaprezentowana została gra Pigeon Run. Gra trafiła do finałowej trójki najlepszych prac w kategorii Virtual Environment.

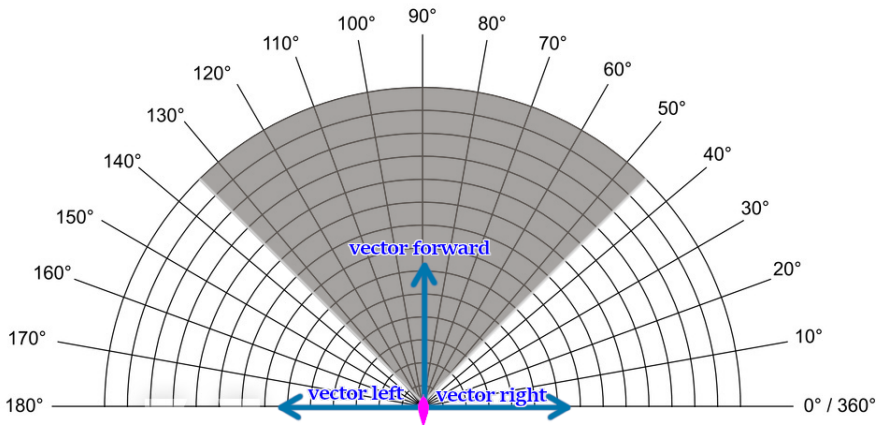
## 3.3 Dyskusja i wnioski

Analizując uzyskane od użytkowników odpowiedzi można było dostrzec pewne skłonności użytkowników. Użytkownicy preferowali szybsze machanie ale w krótszym zasięgu niż wykonywanie wolniejszych ale dłuższych ruchów, które okazywały się niekomfortowe. Podczas testowania trybu siedzącego i stojącego, gra okazała się dla użytkowników

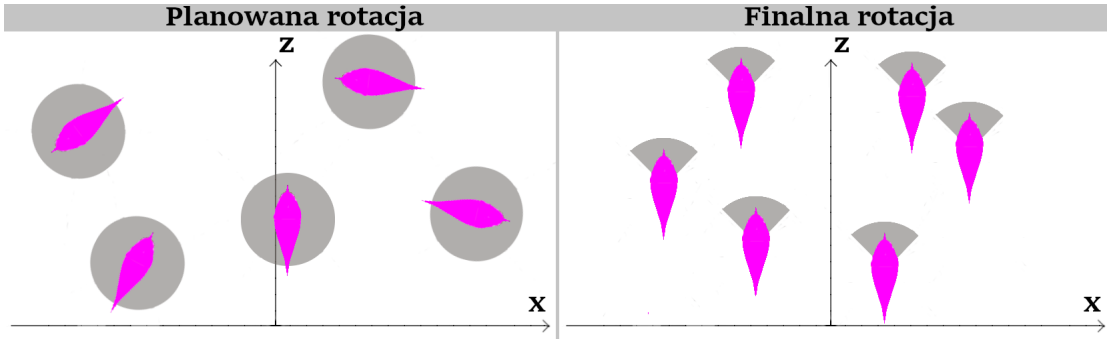
łatwiejsza siedząc, gdyż stojąc nieświadomie przemieszczali się i obracali, przez co łatwo było zgubić orientacje. Koniecznym było również zwiększenie masy gołębia przy zerowym oporze, ponieważ użytkownicy doznawali szoku podczas zbyt szybkiego wznoszenia się.

Efekt lotu okazał się dla użytkowników intuicyjny przy odpowiednim dostosowaniu parametrów identyfikacji, stąd założenia projektu można uznać za spełnione. Problemy jakie pojawiły się podczas implementacji wymusiły zmianę realizacji podstawowych założeń na mechanikę sterowania graczem.

Koniecznym okazało się zrezygnowanie z obrotu ciała głównego obiektu. Początkowo zaimplementowany został mechanizm obracania się gołębia o 360°, co miało umożliwić poruszanie się po mapie gry w dowolnym kierunku bez ograniczeń. Obrót zrealizowany został tak, aby rotacja obiektu była zawsze zgodna z kierunkiem patrzenia gracza. W praktyce gołąb miał lecieć tam, gdzie patrzy się użytkownik (Rys. 3.6 i Rys. 3.7). Po przygotowaniu prototypu, okazało się jednak, że użytkownicy nie są w stanie wytrzymać zmieniającej się jednoczenie rotacji głównego obiektu i rotacji kamery, a po kilkunastu sekundach zawroty głowy były zbyt duże by kontynuować rozgrywkę. Dodatkowym problem przy obracaniu okazał się przewodowy hełm, ograniczający ruchy przez plączące się kable. Odebranie możliwości swobodnego obrotu zmusiło do przebudowy bazowego pomysłu i ograniczenie do poruszania się tylko w przód z lekkimi odchyłami na boki w zasięgu 90°. Finalnie odebrana została możliwość obracania się a rzecz latania na boki, bez zmiany rotacji głównego obiektu. Aby móc przemieszczać się na boki, zaimplementowane zostało rozróżnienie kontrolerów na lewy i prawy. Następnie w zależności od rodzaju kontrolera wektor wynikowy ruchu jest przemnażany przed wektor kierunku przeciwnego do strony kontrolera. Dodatkowo, zmodyfikowana została mapa gry na podłużną ze ścieżką idącą chwiejnie na wprost.




Rysunek 3.6: Schemat zasięgu ruchu gołębia.




Rysunek 3.7: Porównanie zasięgów ruchu gołębia z założeniami i finalnym stanem. Wyszarzony obszar oznacza zasięg ruchu wynikający z sumowania wektorów.

Kolejnym problemem był brak poczucia uziemienia gołębia. W celu urzeczywistnienia upadku na ziemię, niezbędne było dodanie efektu wizualnego w postaci zatrzęsienia się ptaka przy zetknięciu z ziemią, a także dodanie lekkiego oporu przy wznoszeniu się. Gołąb nie mógł od razu wznosić się w powietrze i lecieć z maksymalną siłą, tylko musiało nastąpić lekkie opóźnienie związane z oderwaniem się i płynne zwiększanie prędkości. Udało się to rozwiązać poprzez sumowanie sił wykonanych dotychczas ruchów, które wygasały z czasem. Dzięki temu, gdy gracz dopiero rozpoczynał rozgrywkę i startował z sumą sił równą 0, to musiał włożyć więcej wysiłku na nadanie obiektowi prędkości, niż gdyby już od jakiegoś czasu leciał.

Podczas pracy nad projektem można było poznać możliwości jakie daje wirtualna rzeczywistość. Pozwala ona na przeniesienie się z perspektywy pierwszej osoby do całkowicie nowych zasymulowanych miejsc, a także doświadczyć uczuć, które często są nieosiągalne dla zwykłego człowieka. Opracowany projekt pozwolił na spojrzenie na świat oczami ptaka i spróbowania lotu przy wykorzystaniu siły własnych mięśni. Wykorzystanie kontrolerów odzwierciedlonych w grze, pozwoliło na głębsze wejście w interakcję z wirtualnym światem. Możliwość przeniesienia fizycznych urządzeń do wirtualnego świata daje blisko nieograniczone możliwości interakcji zbliżonej do świata realnego. Urządzenia takie mogą być analizowane pod względem zmiany położenia i rotacji, a dzięki dodatkowym przyciskom użytkownik może dopełniać interakcję.

Mając na uwadze ograniczenia w postaci przewodowości urządzenia, korzystając z VR nie możemy pozwolić sobie na wykonywanie zbyt zaawansowanych ruchów typu przewroty czy obroty. Problem stanowi również przemieszczanie się i obracanie kamerą, ponieważ bardzo często użytkownikom doskwiera choroba lokomocyjna, dlatego najlepszymi symulacjami okazują się te, które można wykonać siedząc bądź stojąc w miejscu. Spore ograniczenie stanowi również brak czujników dla całego ciała i możliwość wchodzenia w interakcję jedynie głową oraz dłońmi.

Reasumując, posiadając bazowe kontrolery dostępnych obecnie na rynku zestawów i wykorzystując uproszczenia mające poprawić komfort użytkownika, jesteśmy w stanie dokonać identyfikacji i odwzorowania złożonych ruchów wykonywanych podczas użytkowania gogli VR.

# Rozdział 4

# Podsumowanie

Celem pracy było stworzenie algorytmu identyfikującego ruchy wykonywane poprzez kontrolery wirtualnej rzeczywistości a także ruchy gałek ocznych rejestrowane kamerami eye tracker. Realizując projekt, ruchy udało się zidentyfikować i skalibrować ich identyfikację pod preferencję danego użytkownika. Wykorzystując zebrane o ruchu dane udało się stworzyć interfejs, który przekłada wykonywane przez użytkownika ruchy na ruchy w wirtualnym świecie. Na podstawie opracowanych algorytmów powstała gra symulacyjna, pozwalająca przenieść się użytkownikowi w świat wirtualny i wejść z nim w interakcję jako zwierzę.

W ramach pracy dokonano:

- Przeglądu rozwiązań służących do obsługiwania gestów wykonywanych kontrolerami gogli VR
- Opracowania autorskiego algorytmu identyfikacji złożonych gestów. Dane o ruchu pochodzą z czujnika ruchu głową oraz kontrolerów gogli VR.
- Implementacji algorytmu identyfikacji oraz odwzorowania ruchu w środowisku Unity
- Demonstracji oraz analiza działania opracowanego algorytmu
- Analizy rezultatów

Największą trudnością stało się komfortowe w odczuciu użytkownika odwzorowanie zidentyfikowanych ruchów i przełożenie zebranych danych na odpowiednie wartości. Jednak dzięki wykorzystaniu dodatkowych pluginów oraz własnej implementacji udało się osiągnąć zamierzony cel i stworzyć interfejs między użytkownikiem gogli a środowiskiem wirtualnej rzeczywistości.

Projekt pozostawił dużo możliwości dalszych prac zarówno pod względem dodania kolejnych metod identyfikacji jak i prób udoskonalenia odwzorowania w wirtualnym świecie ruchów wykonywanych przez użytkownika. Dużym problemem technologi wirtualnej rzeczywistości jest dyskomfort jaki odczuwa użytkownik w postaci choroby lokomocyjnej przy próbie przemieszania jego ciała w symulacji. Dlatego też opracowanie dobrych praktyk bądź rozwiązań minimalizujących ten efekt był by przełomowym odkryciem. Ponadto sama identyfikacja ruchów na większej ilości płaszczyzn niż zostało to zaimplementowane w pracy pozwoliłaby na zdobycie większej ilości danych gotowych do przetworzenia i wykorzystania.

# Bibliografia


[Car19] Oficjalna strona cardboard. https://arvr.google.com/intl/pl_pl/cardboard/, Data dostępu: 17.10.2019.

[Ocu19] Dokumentacja dla deweloperów urządzeń oculus. https://developer.oculus.com/documentation/pcsdk/latest/concepts/dg-input-touch-overview/, Data dostępu: 30.10.2019.

[Pla19] Oficjalna strona playstation vr. https://www.playstation.com/pl-pl/explore/playstation-vr/, Data dostępu: 17.10.2019.

[PON19] PONTYPANTS. Unity vs unreal – which engine should you choose?, Data dostępu: 11.11.2019.

[Sam19] Oficjalna strona samsung gear vr. https://www.samsung.com/pl/wearables/gear-vr-r324/, Data dostępu: 17.10.2019.

[Viv19a] Dokumentacja użytkownika htc vive. https://fast.ulmart.ru/manuals/4066906.pdf, Data dostępu: 9.11.2019.

[Viv19b] Oficjalna strona vive. https://www.vive.com/eu/product/, Data dostępu: 9.11.2019.

# Dodatek A

# Poster z konferencji CGI 2019 (Conference on Game Innovations 2019)

**Tytuł**: Classification of complex movements of VR headset controllers

**Autorzy**: Paulina Patrowicz and Radosław Mantiuk

**Abstrakt:** The VR headset users use controllers to interact with the virtual environment. The controllers are equipped with a number of sensor to determine their three dimensional location. This real time information is used to identify not only basic movements like pointing or grabbing objects but also to classify complex movements of the human hands. In this project we use the VR headset controllers to classify movements reminiscent of waving wings by a bird. The motivation behind this work is to develop a computer game, in which the player takes on a flying bird. The range and frequency of waving hands translate into bird's flight parameters. We implemented a prototype game based on the Unity3D engine. The controllers together with appropriate transformation matrix are assigned to the objects in the scene. These objects are given a speed proportional to the movements of the controllers. The game is an example of using the physical movement of the player to control objects on the scene, such a move requires not only precision, but also appropriate physical commitment.

**Data publikacji**: Jun 16 2019

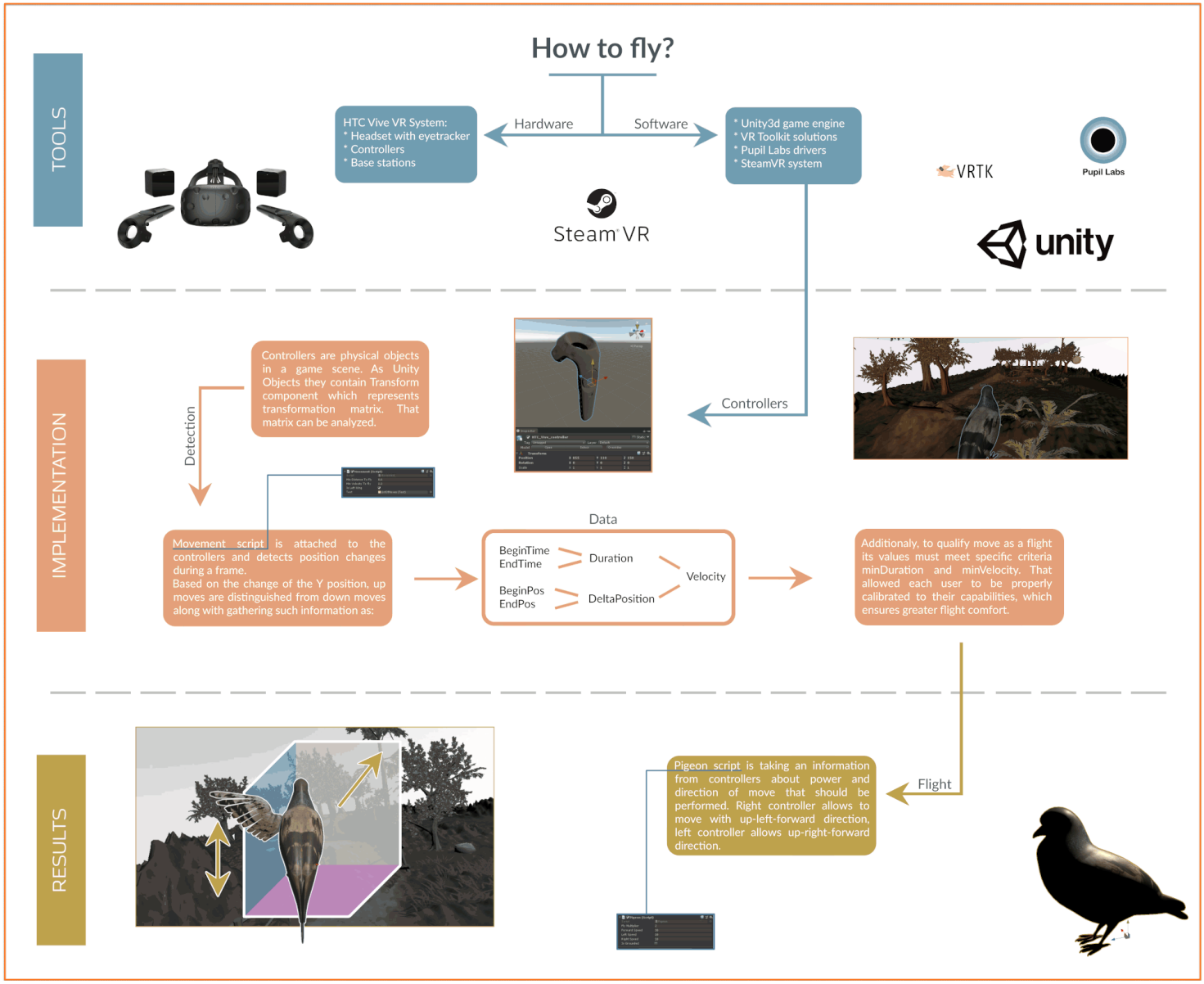
**Paulina Patrowicz and Radosław Mantiuk**
**West Pomeranian University of Technology, Szczecin**

# Classification of complex movements of VR headset controllers

**Abstract**

The VR headset users use controllers to interact with the virtual environment. The controllers are equipped with a number of sensor to determine their three dimensional location. This real time information is used to identify not only basic movements like pointing or grabbing objects but also to classify complex movements of the human hands. In this project we use the VR headset controllers to classify movements reminiscent of waving wings by a bird. The motivation behind this work is to develop a computer game, in which the player takes on a flying bird. The range and frequency of waving hands translate into bird's flight parameters.
We implemented a prototype game based on the Unity3D engine. The controllers together with appropriate transformation matrix are assigned to the objects in the scene. These objects are given a speed proportional to the movements of the controllers.
The game is an example of using the physical movement of the player to control objects on the scene, such a move requires not only precision, but also appropriate physical commitment.

Keywords: VR controllers, 3D movement classification, user interfaces, VR, HCI




Rysunek A.1: Wycinek posteru przedstawiający schemat oraz opis implementacji

# Dodatek B

# Skrypty zaimplementowane w projekcie

Listing B.1: Klasa analizująca ruch kontrolerów Movement.cs

```
public class Movement : MonoBehaviour
{
    /*klasa przechowująca ruch*/
    class Move
    {
        public float startTime; //czas rozpoczęcia ruchu
        public float duration; //czas trwania ruchu
        public Vector3 beginPosition; //pozycja początku ruchu
        public Vector3 endPosition; //pozycja końca ruchu
        public int direction; //kierunek ruchu
        public float velocity; //prędkość ruchu
        public float deltaPos; //różnica zmiany pozycji
        public bool qualifiesToFly; //kwalifikator poprawności ruchu

        /*konstruktor parametryczny, inicjalizujący zmienne*/
        public Move(float startTime, Vector3 beginPosition, int direction)
        {
            this.startTime = startTime;
            this.beginPosition = beginPosition;
            this.direction = direction;
            this.endPosition = Vector3.zero;
            velocity = 0;
            deltaPos = 0;
            qualifiesToFly = false;
        }
    }

    [SerializeField] private float minDistanceToFly; //klasyfikator minimalnego dystansu
    [SerializeField] private float minVelocityToFly; //klasyfikator minimalnej prędkości
    [SerializeField] private bool isLeftWing; /rozróżnienie rodzaju kontrolera (czy jest lewy)
    public static event Action<float, bool> OnWingForce; //delegat dodający siłę skrzydłu
    private float _timer; //licznik czasu
    private Vector3 _lastMovePosition; //ostatnia pozycja w ruchu
    private Vector3 _previousPosition; //poprzednia pozycja w ruchu
    private int _lastMoveDirection; //ostatni kierunek ruchu
    private List<Move> listOfMoves = new List<Move>(); //lista wszystkich wykonanych ruchów
    private Vector3[] logOfPreviousPositions = new Vector3[3] {
```

```
Vector3.zero, Vector3.zero, Vector3.zero }; //log zbierający informacje o poprzednich
    pozycjach

/*metoda inicjująca scenę*/
private void Start()
{
    _timer = 0;
    listOfMoves.Add(new Move(_timer, transform.localPosition, 0));
    logOfPreviousPositions = new Vector3[3] {
    transform.localPosition, transform.localPosition,
    transform.localPosition }};
    _lastMovePosition = GetLastMoveStartPostion();
    _lastMoveDirection = GetLastMoveDirection();

}
/*metoda odpalana co klatkę - główna pętla gry*/
private void Update()
{
    _timer += Time.deltaTime; //zliczanie czasu
    LogPosition(); //logowanie pozycji

    /*warunek sprawdzający zwrot wykonanego ruchu w kierunku osi Y*/
    if (logOfPreviousPositions[2].y > transform.localPosition.y)
    {
        /*warunek sprawdzający czy zwrot wykonywanego ruchu uległ zmianie*/
        if (_lastMoveDirection != -1)
        {
            ChangeMovement(-1); //zmiana zwrotu wykonywania ruchu
            Debug.Log("Down"); //log informacyjny, nowy zwrot ruchu to dół
        }
    }
    else if (logOfPreviousPositions[2].y < transform.localPosition.y)
    {
        /*warunek sprawdzający czy zwrot wykonywanego ruchu uległ zmianie*/
        if (_lastMoveDirection != 1)
        {
            ChangeMovement(1); //zmiana zwrotu wykonywania ruchu
            Debug.Log("Up"); //log informacyjny, nowy zwrot ruchu to góra
        }
    }
}
/*metoda zwracająca pozycję początkową ostatniego ruchu*/
private Vector3 GetLastMoveStartPostion()
{
    return listOfMoves[listOfMoves.Count - 1].beginPosition;
}
/*metoda zwracająca kierunek ostatniego ruchu*/
private int GetLastMoveDirection()
{
    return listOfMoves[listOfMoves.Count - 1].direction;
}
```

```
    /*metoda zmieniająca zwrot wykonywanego ruchu*/
    private void ChangeMovement(int direction)
    {
        Move moveToEnd = listOfMoves[listOfMoves.Count - 1]; //poprzedni
        ruch
        float deltaT = _timer - moveToEnd.startTime; //różnica czasów
        float deltaS = Vector3.Distance(moveToEnd.beginPosition,
        logOfPreviousPositions[2]); //różnica odegłości
        float velocity = deltaS / deltaT; //prędkość
        moveToEnd.velocity = velocity; //dodanie prędkości do poprzedniego ruchu
        moveToEnd.duration = deltaT; //dodanie czasu do poprzedniego ruchu
        moveToEnd.deltaPos = deltaS; //dodanie odległości do poprzedniego ruchu
        moveToEnd.endPosition = logOfPreviousPositions[2]; //dodanie
        pozycji końcowej do poprzedniego ruchu
        listOfMoves[listOfMoves.Count - 1] = moveToEnd; //dodanie poprzedniego ruchu do listy
            wszysktich ruchów

        listOfMoves.Add(new Move(_timer, transform.localPosition,
        direction));
        //dodanie obecnego ruchu do listy wszysktich ruchów
        _lastMoveDirection = direction; //przypisanie kierunku jako zeszły
        _lastMovePosition = transform.localPosition; //przypisanie pozycji jako zeszła

        QualifyToFly(); //weryfikacja obecnego ruchu jako prawidłowy

    }
    /*metoda klasyfikująca ruch jako do wykonania, wg spersonalizowanych wartości minimalnych*/
    private void QualifyToFly()
    {
        Move moveToQualify = listOfMoves[listOfMoves.Count - 2];
        //pobranie starego ruchu do porównania

        //zweryfikowanie, czy wykonany ruch różni się od starego o określone kryteria
        if (moveToQualify.deltaPos > minDistanceToFly &&
        moveToQualify.velocity > minVelocityToFly)
        {
            moveToQualify.qualifiesToFly = true; //zmiana wartości klasyfikatora
            if (OnWingForce != null) OnWingForce(moveToQualify.velocity,
            isLeftWing); //wyzwolenie delegatu odpowiadającego za lot z przekazaniem
                odpowiedniej siły
        }
    }
    /*metoda logująca ostatnie zmiany pozycji*/
    private void LogPosition()
    {
        logOfPreviousPositions[2] = logOfPreviousPositions[1];
        logOfPreviousPositions[1] = logOfPreviousPositions[0];
        logOfPreviousPositions[0] = transform.localPosition;

    }
}
```

Listing B.2: Klasa obiektu głównego Pigeon.cs

```
public class Pigeon : MonoBehaviour
{

    [ SerializeField ] float flyMultiplier ; //modyfktor lotu
    [ SerializeField ] float forwardSpeed, leftSpeed, rightSpeed; //prędkości w zależności od
        kierunku
    [ SerializeField ] bool isGrounded; //weryfikacja uziemienia postaci
    private Rigidbody rigidbody; //komponent fizyki
    Camera cam; //kamera główna gry

    /*metoda inicjująca scenę*/
    void Start()
    {
        /* inicjalizacje zmiennych*/
        rigidbody = GetComponent<Rigidbody>();
        cam = Camera.main;
        isGrounded = true;

        /*przypisanie delegatu wyzwalającego lot*/
        Movement.OnWingForce += AddForceToFly;
    }

    /*metoda wyliczająca siłę lotu*/
    void AddForceToFly(float force, bool isLeftWing)
    {
    /*warunek sprawdzający rodzaj kontrolera (lewy/prawy) i przekazujący
    odpowiednią siłę o kierunku oraz zwrocie zależnym od kontrolera*/
        if (isLeftWing)
            rigidbody.velocity = transform.forward * forwardSpeed +
            transform.right * leftSpeed + Vector3.up * force *
             flyMultiplier ;
        else
            rigidbody.velocity = transform.forward * forwardSpeed +
            (-transform.right * rightSpeed) + Vector3.up * force *
             flyMultiplier ;
    }

    /*metoda wyzwalana wraz z wejściem gracza w kolizję*/
    void OnCollisionEnter(Collision collision )
    {
        /*sprawdzenie czy obiekt, który wszedł w kolizję jest
        na warstwie 'gracz' a także nie jest uziemiony*/
        if ( collision .gameObject.layer == 8
            && !isGrounded)
        {
            isGrounded = true; //odnotowanie zmiany uziemienia gracza
            GetComponentInChildren<Animation>().Play("PigeonShakeAnim 1"); //włączenie
                animacji upadku
        }
    }

    /*metoda wyzwalana wraz z wyjściem gracza z kolizji*/
```

```
    void OnCollisionExit(Collision collision )
    {
        /*sprawdzenie czy obiekt, który wyszedł z kolizji jest na warstwie 'gracz' a także jest
            uziemiony*/
        if ( collision .gameObject.layer == 8
            && isGrounded)
        {
            isGrounded = false; //odnotowanie zmiany uziemienia gracza
        }
    }

    /*przeciążenie metody destruktora obiektu*/
    private void OnDestroy()
    {
        /*odłączenie delegatu wyzwalającego lot*/
        Movement.OnWingForce -= AddForceToFly;
    }
}
```