



Zachodniopomorski
Uniwersytet
Technologiczny
w Szczecinie



Wydział
Informatyki

Patryk Piotrowski

nr albumu: 32792

kierunek studiów: Informatyka

specjalność: inżynieria oprogramowania

forma studiów: *stacjonarne*

SYSTEM WIRTUALNEJ RZECZYWISTOŚCI Z INTERAKCJĄ WSPOMAGANĄ WZROKIEM

VIRTUAL REALITY SYSTEM WITH GAZE ASSISTED INTERACTION

praca dyplomowa magisterska

napisana pod kierunkiem:

dr inż. Adama Nowosielskiego

Katedra Systemów Multimedialnych

Data wydania tematu pracy: 14.11.2018

Data dopuszczenia pracy do egzaminu:
(uzupełnia pisemnie Dziekanat)

Szczecin, 2019

OŚWIADCZENIE AUTORA PRACY DYPLOMOWEJ

Oświadczam, że praca dyplomowa magisterska pn. „*System wirtualnej rzeczywistości z interakcją wspomaganą wzrokiem*” napisana pod kierunkiem dr inż. Adama Nowosielskiego 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.

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.

.....
podpis dyplomanta

Szczecin, dn.

Virtual reality system with gaze assisted interaction.

Patryk Piotrowski

Supervisor: PhD Adam Nowosielski

Abstract

This thesis pertains to the implementation of the computer game prototype that uses virtual reality headset and motion controllers. Its content includes an overview of the most popular solutions using these technologies with development environments compatible with virtual reality technology. The essence of this project is to provide the user with touchless interaction with virtual world, which in combination with the assumed virtual reality googles gives the user a sense of full involvement in the game. The effectiveness of prepared solutions was examined on a group of students with their collected opinions.

Streszczenie

Niniejsza praca inżynierska dotyczy implementacji prototypu gry komputerowej wykorzystującej google wirtualnej rzeczywistości i kontrolery ruchu. W jej treści zawarty jest przegląd najpopularniejszych rozwiązań korzystających z tych technologii wraz ze środowiskami programistycznymi zgodnymi z techniką wirtualnej rzeczywistości. Istotą tworzonego projektu jest zapewnienie użytkownikowi bezdotykowej interakcji ze światem gry, który w połączeniu z założonymi goglami wirtualnej rzeczywistości daje użytkownikowi poczucie pełnego zaangażowania się w rozgrywkę. Efektywność przygotowanych rozwiązań zbadano na grupie studentów, a na podstawie zebranych opinii oceniono skuteczność tych metod.

Spis treści

1 Przegląd istniejących rozwiązań	3
1.1 Systemy wirtualnej rzeczywistości	3
1.2 Metody komunikacji człowiek-komputer	3
1.3 Okulografia - śledzenie wzroku	3
1.4 Podsumowanie	3
2 Przegląd narzędzi programistycznych i środowiska badawczego	4
2.1 Opis projektu	4
2.2 Zestaw wirtualnej rzeczywistości HTC Vive	4
2.2.1 Przegląd zestawu	5
2.2.2 Konfiguracja stanowiska	7
2.2.3 Wymagania sprzętowe	8
2.3 Urządzenie do śledzenia wzroku Pupil	8
2.3.1 Dostarczone oprogramowanie	8
2.3.2 Struktura procesu	9
2.3.3 Umiejscowienie kamery	10
2.3.4 Oprogramowanie klienckie	10
2.4 Środowisko Unity	12
2.4.1 Test	14
2.5 Podsumowanie	26
3 Implementacja systemu	27
3.1 Opis środowiska	27
3.2 Format danych wejściowych i wyjściowych	27
3.3 Opis implementacji	27
3.3.1 Struktura projektu	27
3.4 Podsumowanie	27
4 Rezultaty i wnioski	28
4.1 Opis procedury testowej	28
4.2 Prezentacja rezultatów	28
4.3 Wnioski z przeprowadzonych badań	28
5 Podsumowanie	29
A Opis zawartości płyty DVD	31

Wprowadzenie

Niniejsza praca inżynierska dotyczy implementacji gry korzystającej z techniki wirtualnej rzeczywistości na silniku do gier Unity. Dynamiczny rozwój jak i wzrost mocy obliczeniowej kart graficznych przyczynił się do tego, że możliwe stało się tworzenie coraz to bardziej szczegółowych i realistycznych gier komputerowych. Idący w parze z rozwojem nasilający się trend związany z goglami wirtualnej rzeczywistości daje możliwości na tworzenie zupełnie nowych tytułów, które pozwalają użytkownikowi na bardziej realistyczne węglubienie się w wirtualny świat gry.

Główną motywacją pracy była chęć zgłębienia wiedzy w zakresie tworzenia gier w technologii wirtualnej rzeczywistości jak i również fascynacja nowinkami technicznymi. Projekt zakłada wyposażenie użytkownika w gogle wirtualnej rzeczywistości Oculus Rift w wersji DK2 oraz zamontowany na przedniej stronie obudowy sensor śledzenia ruchów dłoni Leap Motion HandTracer Sensor. W założeniach cała obsługa sterowania grą oraz interakcja ze światem będzie odbywała się za pomocą obsługi detekcji ruchu jak i gestów dłoni. Celem pracy jest realizacja prototypu gry komputerowej wykorzystującej gogle wirtualnej rzeczywistości i kontrolery ruchu. Na realizację celu składał się przegląd oraz analiza rozwiązań stosowanych w goglach wirtualnej rzeczywistości oraz interfejsach bezdotykowych, następnie do zadań należało sformułowanie celu gry oraz założeń funkcjonowania interfejsu. Kolejnym krokiem było przeprowadzenie analizy dostępnych bibliotek programistycznych oraz silników gier pod kątem wykorzystania w realizowanym projekcie. Po wybraniu odpowiedniego środowiska do tworzenia gier następuje implementacja prototypu oraz sposobu sterowania, a następnie ocena opracowanego rozwiązania.

Treść pracy zawiera się w pięciu rozdziałach. Rozdział pierwszy opisuje podstawowe pojęcia z dziedziny wirtualnej rzeczywistości, budowę oraz zasady działania gogli i zastosowanego kontrolera śledzenia ruchu dłoni. W rozdziale drugim opisano szczegółowo założenia implementowanej gry oraz metodykę realizowanego projektu. Rozdział trzeci zawiera szczegółowy opis implementacji najważniejszych elementów proponowanego rozwiązania. W rozdziale czwartym zawarte są rezultaty przeprowadzonych badań w zakresie efektywności oraz poziomu imersji użytkownika z grą. Rozdział piąty podsumowuje całą pracę, komentując jej efekty i napotkane problemy oraz o planach rozwinięcia projektu.

Rozdział 1

Przegląd istniejących rozwiązań

1.1 Systemy wirtualnej rzeczywistości

1.2 Metody komunikacji człowiek-komputer

1.3 Okulografia - śledzenie wzroku

1.4 Podsumowanie

Koncepcja wprowadzenia wirtualnej rzeczywistości z wykorzystaniem technologii komputerowej trwa od lat 60-tych XX wieku. W procesie ich ewolucji powstało wiele koncepcji i projektów, lecz w wyniku wysokich kosztów opracowania tych technologii oraz stosunkowo ograniczonej mocy obliczeniowej komputerów rozwiązania te nie zdobyły popularności. Za kolejną próbę inicjatywy rozpowszechnienia wirtualnej rzeczywistości w 2013 roku stała firma Oculus VR, później za jej śladami podążyły między innymi takie firmy jak HTC, Sony oraz Google prezentując swoje rozwiązania.

Obecny udział wirtualnej rzeczywistości na rynku jest znikomy (w wyniku przeprowadzonej ankiety na platformie Steam 0.23% użytkowników posiada gogle wirtualnej rzeczywistości[?]) nie tylko głównie ze względu na koszty takich rozwiązań, ale również z powodu wysokich wymagań sprzętowych wymaganych do płynnej rozgrywki oraz brak pokaźnej ilości tytułów wysokobudżetowych korzystających z tych technologii.

W dalszym rozdziale zostaną zaprezentowane założenia gry wraz z interakcją bezdotykową oraz przedstawiony będzie zestaw wirtualnej rzeczywistości wraz ze środowiskiem programistycznym, na którym będzie implementowany projekt gry.

Rozdział 2

Przegląd narzędzi programistycznych i środowiska badawczego

W niniejszym rozdziale zostaną przedstawione wybrane technologie, które posłużą do implementacji projektu wraz z przygotowaniem środowiska badawczego.

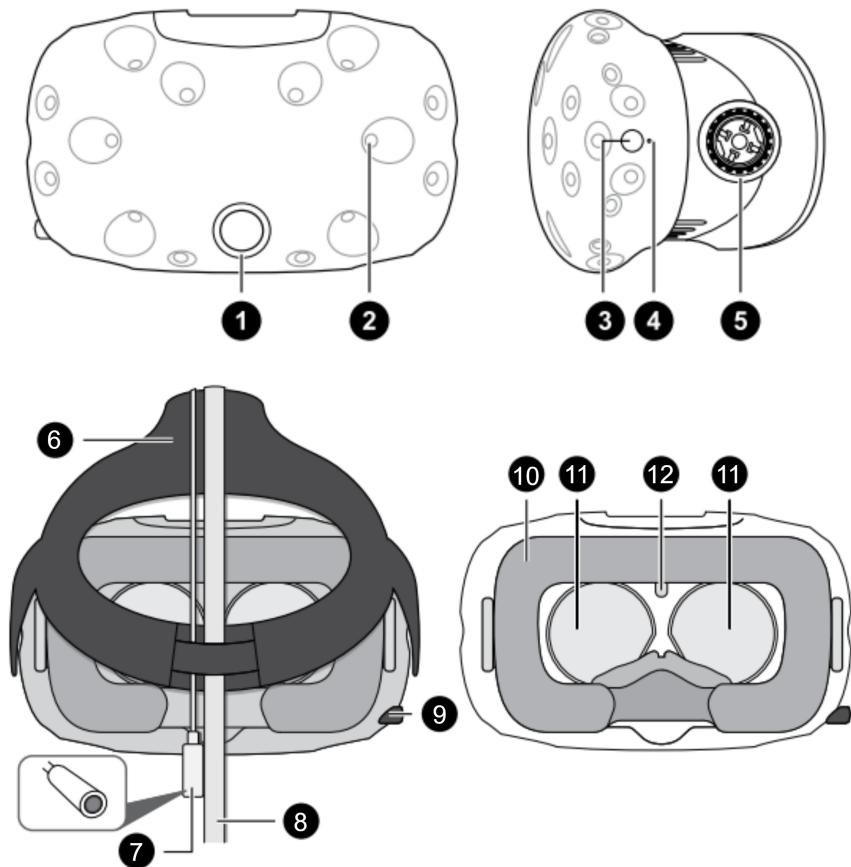
2.1 Opis projektu

Głównym celem projektu jest opracowanie systemu interakcji z wykorzystaniem wzroku w środowisku wirtualnej rzeczywistości. Opracowany system pełniłby rolę interfejsu pomiędzy oprogramowaniem wraz z okulografem dostarczonym przez PupilLabs, a środowiskiem interakcji dostępnym w oprogramowaniu Unity i zestawem wirtualnej rzeczywistości HTC Vive. Implementacja ma opierać się na wdrożeniu komponentów odpowiedzialnych za:

- mapowanie kierunku patrzenia na współrzędne ekranu oraz wygenerowanie promienia (*ang. raycast*) z punktu skupienia wzroku, a następnie przeprowadzenie analizy intersekcji z obiektem;
- obsługa interakcji obiektów, na których skupiony jest wzrok - elementy interfejsu użytkownika (przyciski akcji) oraz obiekty, z którymi użytkownik może przeprowadzić interakcję;
- rozszerzenie interakcji z obiektami wykorzystując systemu obsługi zdarzeń - *Invokowanie funkcji w Unity* - nie wiem jak to jeszcze nazwać.

2.2 Zestaw wirtualnej rzeczywistości HTC Vive

Vive to pierwszy system wirtualnej rzeczywistości opracowany wspólnie przez HTC oraz Valve[2]. Vive łączy najnowocześniejsze technologie w pełny system zawierający obraz, dźwięk oraz precyzyjne śledzenie ruchu[2].

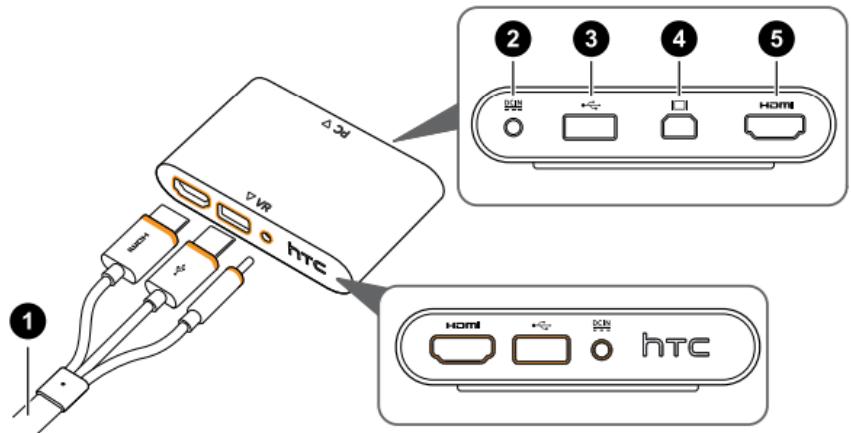


Rysunek 2.1: Schemat budowy gogli wirtualnej rzeczywistości HTC Vive[4]: 1. Obiektyw kamery. 2. Sensory śledzące ruch gogli. 3. Przycisk włączania/wyłączania headsetu. 4. Status light. 5. Regulator do zmiany odległości obiektywu. 6. Pasek usztywniający headset. 7. Złącze audio. 8. Pakiet złączy 3-in-1 do komputera. 9. Regulator IPD (interpupillary distance). 10. Amortyzator na głowę. 11. Soczewki. 12. Proximity sensor.

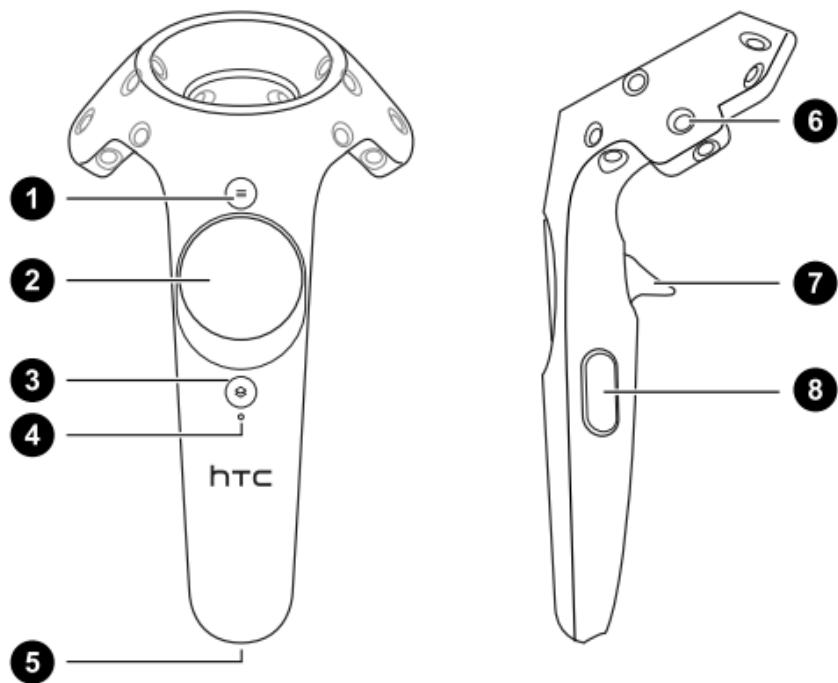
2.2.1 Przegląd zestawu

Zestaw HTC Vive składa się z następujących komponentów[4]:

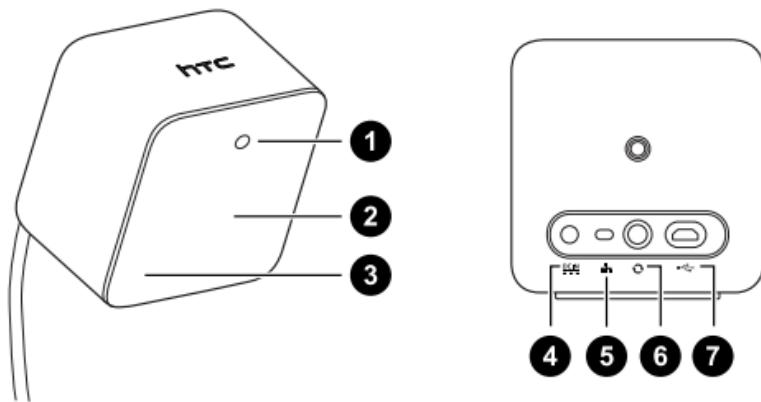
- headset wirtualnej rzeczywistości z ekranem OLED dysponującym rozdzielcząscią 2160x1200 pikseli (1080x1200 dla każdego oka), pracujący z częstotliwością odświeżania 90Hz (rysunek 2.1);
- urządzenie *Link Box* pozwalające na podłączenie gogli wirtualnej rzeczywistości z komputerem (rysunek 2.2);
- zestaw dwóch bezprzewodowych kontrolerów *Vive controllers* - pozwalające na interakcję użytkownika w wirtualnym środowisku (rysunek 2.3);
- dwie stacje bazowe *Base station* śledzące pozycję gogli i kontrolerów2.4.



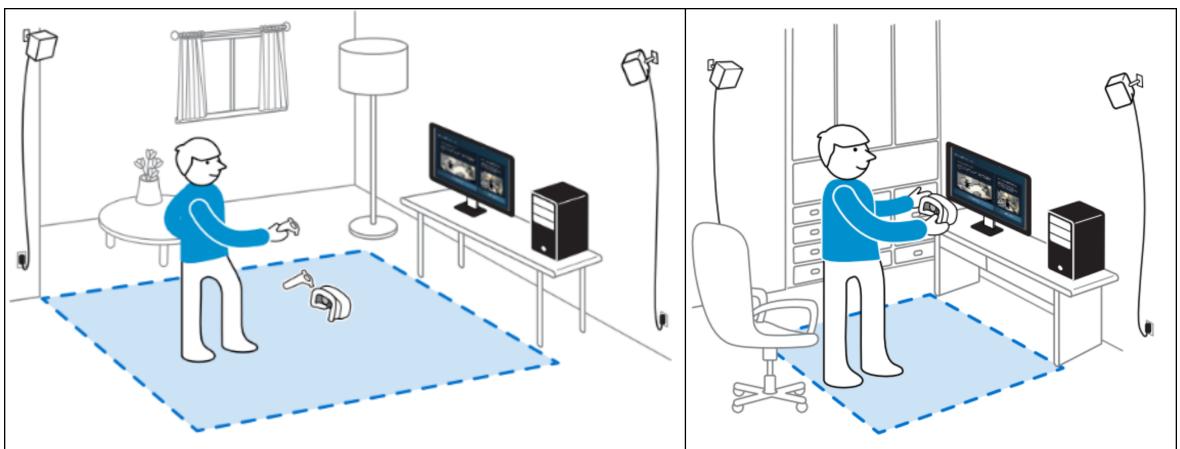
Rysunek 2.2: Schemat budowy urządzenia *Link Box*[4]: 1. Pakiet złączy 3-in-1 pochodzących z headsetu wirtualnej rzeczywistości. 2. Złącze zasilania. 3. Port USB. 4. Wejście Mini DisplayPort™. 5. Port HDMI.



Rysunek 3.3: Schemat budowy bezprzewodowego kontrolera wchodzącego w skład zestawu HTC Vive[4]: 1. Przycisk menu. 2. Trackpad. 3. Przycisk systemowy. 4. Sensor informujący o stanie urządzenia. 5. Port ładowania za pomocą złącza USB. 6. Sensory śledzące ruch kontrolera. 7. Przycisk trigger. 8. Przycisk grip.



Rysunek 2.4: Schemat budowy stacji bazowej (*Base Station*) śledzącej pozycję gogli i kontrolerów zestawu HTC Vive: 1. Sensor informujący o stanie urządzenia. 2. LED lens. 3. Informacje o. 4. Port zasilania. 5. Przycisk zmiany kanału (*channel button*). 6. Port na złącze synchronizujące. 7. Port micro USB.



Rysunek 2.5: Konfiguracja stanowiska dla HTC Vive: po lewej *tryb swobodny*, po prawej *tryb stojący*.[4]

2.2.2 Konfiguracja stanowiska

W celu korzystania z zestawu HTC Vive należy zainstalować sterowniki dostarczone przez producenta oraz aplikację Steam z nakładką *SteamVR*[4]. Po poprawnej instalacji należy przeprowadzić kalibrację obszaru korzystania z zestawu wirtualnej rzeczywistości. Oprogramowanie pozwala na wybór jednego z dwóch trybów korzystania z urządzenia (rysunek 2.5)[4]:

- tryb swobodny (*ang. room-scale setup*) - pozwala na swobodne poruszanie się po obszarze działania, wymagany jest minimalny obszar o wymiarach 1,5 x 2 metrów
- tryb stojący (*ang. standing-only play arena*) - brak wymogów dotyczących wymiarów, lecz ograniczona swoboda imersji użytkownika.

2.2.3 Wymagania sprzętowe

Aby w pełni wykorzystać możliwości urządzenia HTC Vive, komputer powinien spełniać następujące wymagania systemowe[2]:

- karta graficzna: NVIDIA GeForce® GTX 970, AMD Radeon™ R9 290, ich odpowiednik lub lepsza;
- procesor: Intel® i5-4590, AMD FX 8350, ich odpowiednik lub lepszy;
- pamięć RAM: 4GB lub więcej;
- wyjście wideo: HDMI 1.4, DisplayPort 1.2 lub nowsze;
- port USB: 1 port USB 2.0 lub lepszy;
- system operacyjny: Windows 7 SP1, Windows 8.1 lub Windows 10.

Ponadto wymagana jest instalacja klienta Steam oraz posiadane konto Steam z zaakceptowaną *Umową użytkownika Steam*[2].

2.3 Urządzenie do śledzenia wzroku Pupil

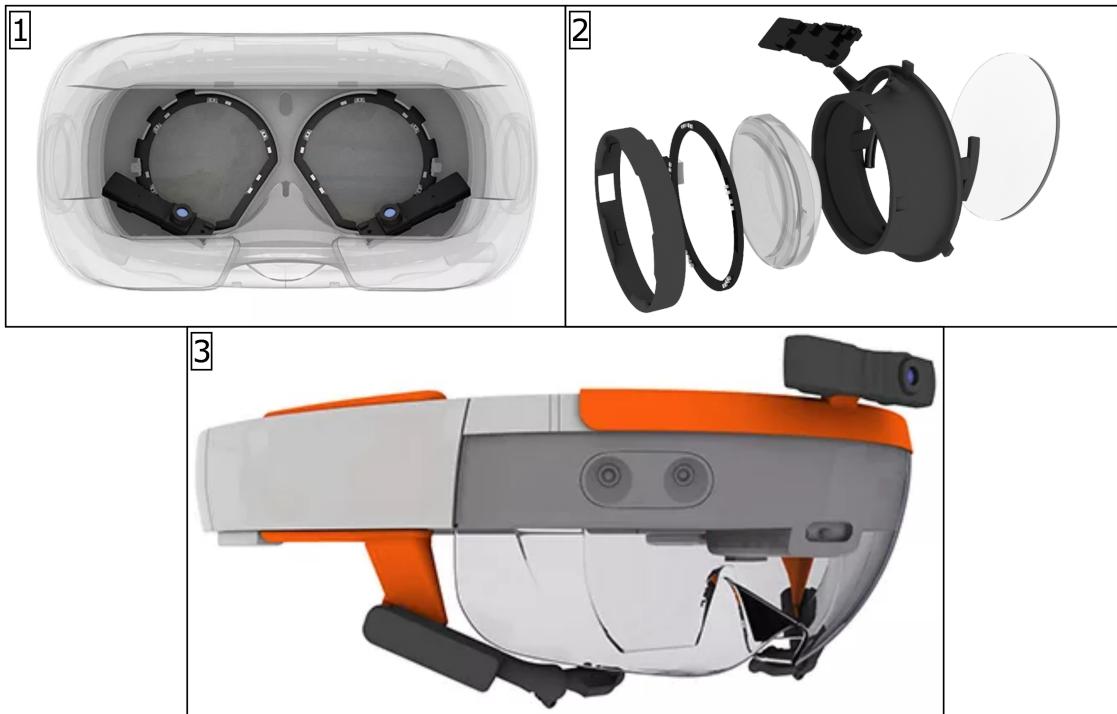
Firma Pupil Labs zajmuje się produkcją urządzeń oraz oprogramowania do śledzenia wzroku dla mobilnych headsetów, systemów wirtualnej i rozszerzonej rzeczywistości[1]. Dostarczane urządzenia są dedykowane między innymi dla:

- HTC Vive / HTC Vive PRO (rysunek 2.6.1),
- Oculus Rift DK2 / CV1 (rysunek 2.6.2),
- Microsoft HoloLens (rysunek 2.6.3).

2.3.1 Dostarczone oprogramowanie

Pupil jest oprogramowaniem typu Open Source, które zostało napisane w języku Python 3 wraz z wykorzystaniem komponentów odpowiedzialnych za widzenie maszynowe, kompresję danych i biblioteki graficzne opracowane w języku c/c++, które komunikują się za pośrednictwem rozszerzenia Cython, będący nadzbiorem języka Python cechującym się wsparciem dla wywoływania funkcji i deklaracji w języku C na zmiennych oraz atrybutów klas[1]. Oprogramowanie składa się z trzech głównych komponentów:

- Pupil Capture – jest to oprogramowanie wykorzystywane wraz z Headsetem Pupil. Program odczytuje strumień wideo z kamery rejestrującej otoczenie oraz kamery śledzącej wzrok. **Pupil Capture uses the video streams to detect your pupil, track your gaze, detect and track markers in your environment, record video and events, and stream data in realtime;**



Rysunek 2.6: Sposób montażu sensorów PupilLabs odpowiedzialnych za śledzenie wzroku: 1. HTC Vive 2. Oculus Rift DK2 3. Microsoft HoloLens

- Pupil Player – jest to oprogramowanie, które zamiast odczytywania strumienia wideo z kamery rejestrującej otoczenie wykorzystuje uprzednio przygotowanie nagranie przez użytkownika;
- Pupil Service – komponent ten w głównym założeniu skierowany jest dla systemów wirtualnej (Virtual Reality) oraz rozszerzonej rzeczywistości (Augment Reality). Komunikacja programu klienckiego z Pupil Service odbywa się z wykorzystaniem protokołu TCP.

2.3.2 Struktura procesu

W momencie uruchomienia przechwytywania i śledzenia wzroku uruchamiane są dwa procesy: Eye oraz World. Oba te procesy niezależnie od siebie przetwarzają dane z kamer dla każdej pojedynczej klatki obrazu[1].

Głównym celem dla **Eye Process** jest wykrycie źrenicy, a następnie udostępnienie jej pozycji. Proces detekcji prezentuje się następująco:

- pobranie pojedynczej klatki obrazu ze strumienia kamery śledzącej wzrok,
- detekcja pozycji źrenicy na obrazie,
- transmisja otrzymanej pozycji źrenicy.

World Process jest odpowiedzialny za:

- pobranie pojedynczej klatki obrazu ze strumienia kamery rejestrującej otoczenie znajdujące się na wprost od użytkownika;

- pobiera pozycję oka z procesu **Eye Process**;
- dokonuje kalibracji mapowania z pozycji źrenicy na koordynaty obszaru, na które użytkownik skupia wzrok;
- zajmuje się wczytywaniem i obsługą wtyczek – detect fixations, track surfaces, itp;
- nagrywaniem obrazu rejestrowanego z kamer i zapisywaniem danych (znaczniki czasu, pozycje skupienia wzroku, itp.).

2.3.3 Umiejscowienie kamery

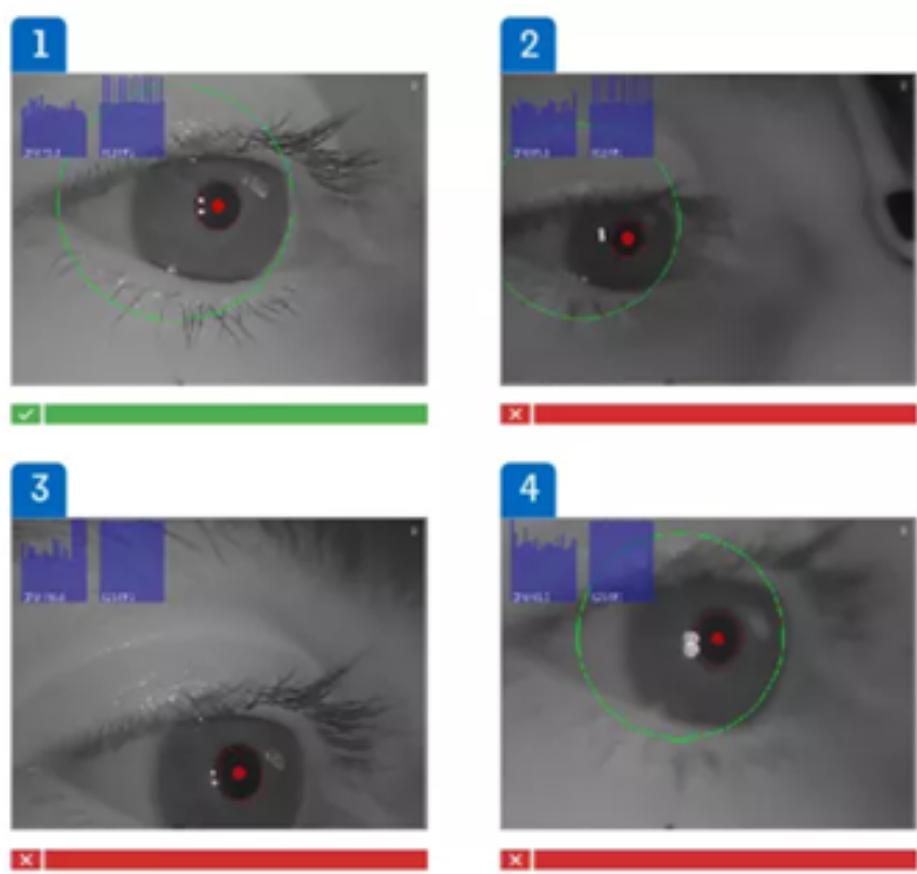
Istotnym czynnikiem przed rozpoczęciem kalibracji jest upewnieniem się, czy pozycje oczu są prawidłowo umiejscowione w celu uzyskania poprawnej detekcji oczu przez oprogramowanie PupilLabs (rysunek 2.7):

- **prawidłowo** - widoczny jest cały obszar śledzonego oka (rysunek 2.7.1),
- **nieprawidłowo** - wykrywana jest źrenica i częściowo oko (rysunek 2.7.2),
- **nieprawidłowo** - wykrywana jest tylko źrenica (rysunek 2.7.3),
- **nieprawidłowo** - obraz jest rozmyty i mogą wystąpić problemy z prawidłową detekcją (rysunek 2.7.4),

2.3.4 Oprogramowanie klienckie

Dostarczone oprogramowanie klienckie o nazwie **hmd-eyes** jest zbiorem wtyczek dedykowanych obsłudze śledzenia wzroku dla systemów korzystających z techniki wirtualnej rzeczywistości[3]. Pakiet składa się z zestawu assets'ów dedykowanych dla środowiska Unity, które służą do komunikacji z głównym komponentem Pupil Service[3]. W skład pakietu wchodzą takie komponenty jak:

- Calibration System – przed rozpoczęciem korzystania z oprogramowania należy przeprowadzić kalibrację systemu. Proces kalibracji polega na wyświetleniu w danym obszarze znacznika, na który użytkownik musi skupić wzrok, następnie otrzymane wyniki służą do mapowania kierunku patrzenia źrenic do współrzędnych spojrzenia (rysunek 2.8.1). Ponadto system kalibracji zapewnia dodatkowe ustawienia kalibracji – typ kalibracji (2d/3d), liczba próbek, wielkość znacznika, itp. (rysunek 2.8.2);
- Pupil Manager – jest to komponent, który zarządza połączeniem pomiędzy systemem Pupil Servicę a aplikacją kliencką (komunikacja z serwerem, przeprowadzenie procesu kalibracji). Komponent odpowiada za nasłuchiwanie i obsługę komunikatów przychodzących z aplikacji takich jak:
 - ustanowiono/przerwano połączenie,
 - informacje dotyczące kalibracji systemu (rozpoczęcie, ukończenie i niepowodzenie przeprowadzenia kalibracji),



Rysunek 2.7: Test

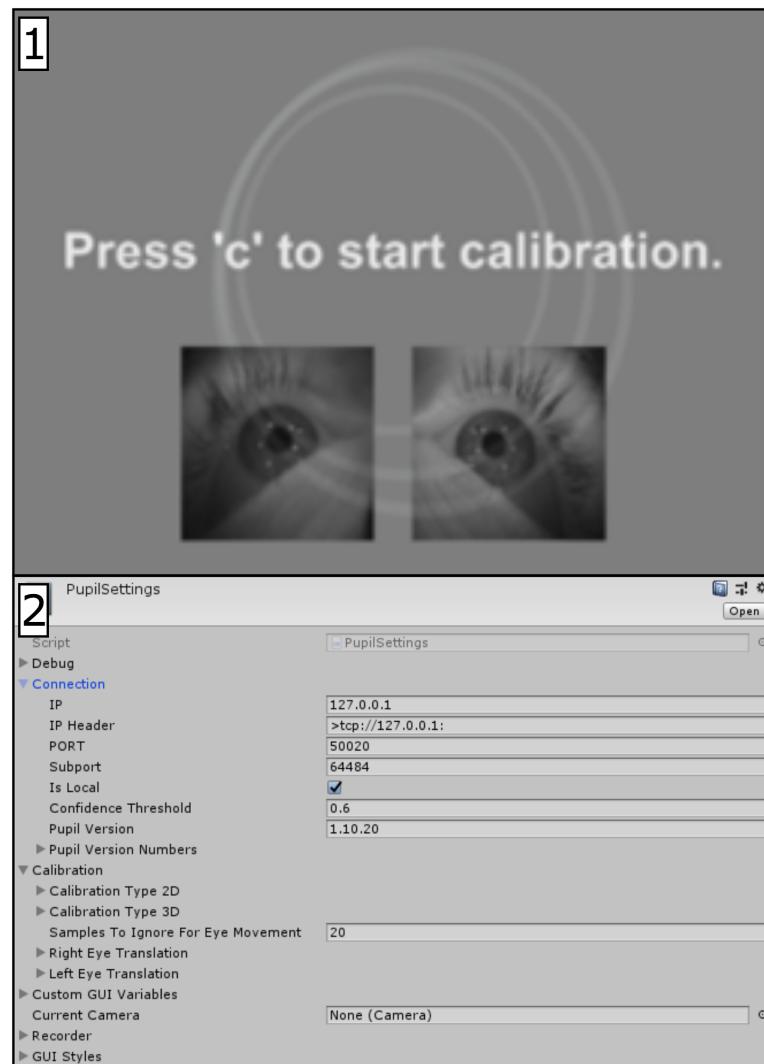
- dane informujące o kierunku patrzenia,
- dodatkowe komunikaty dostarczone przez wtyczki (*ang. plugin*).
- Accesing Data - komponenty, które służą do nasłuchiwanego zdarzeń,
- Recording Data;
- Example Scenes - przygotowane sceny, które mogą posłużyć jako wzorzec do przygotowania własnego środowiska:
 - kalibracja - scena służąca do przeprowadzenia operacji kalibracji systemu śledzenia wzroku (rysunek 2.8.1);
 - blink – zawiera komponenty prezentujące proces nasłuchiwanego zdarzeń przychodzących z systemu, w tym przypadku dotyczących mrugnięcia okiem przez użytkownika;
 - frame publishing - scena demonstrująca działanie Pupil ze strumieniem video z kamery okulografu;
 - VR Calibration Demo Scene - domyślna scena wynikowa uruchamiana po przeprowadzeniu procesu kalibracji.

2.4 Środowisko Unity

Unity jest jednym z narzędzi służących do tworzenia dwuwymiarowych oraz trójwymiarowych gier i aplikacji multimedialnych przeznaczonych między innymi dla komputerów osobistych, konsoli do gier jak i urządzeń mobilnych (rysunek 2.10). Ponadto zapewnione jest natywne wsparcie dla hełmów wirtualnej rzeczywistości, między innymi takich jak Oculus Rift oraz HTC Vive. Środowisko Unity pozwala na szybkie oraz efektywne wdrażanie iteracji gry poprzez cykle budowy prototypu i jego testowania. Unity umożliwia pisanie własnego kodu w popularnych językach programowania, takich jak C# oraz UnityScript mający składnię zaczerpniętą z języka JavaScript. Dzięki rozbudowanej dokumentacji technicznej oraz nieustannemu wsparciu ze strony deweloperów jak i społeczności za sprawą *Asset Store'a*, Unity jest jednym z najbardziej popularnych narzędzi wybieranym przez twórców gier.

Obiekty umieszczone na scenie składają się z **komponentów** (rysunek 2.9). Każdy z obiektów posiada komponent **Transform**, który zawiera informacje o *pozycji, obrocie i przeskalowaniu* danego obiektu w kartezjańskim układzie współrzędnych. Dodatkowe komponenty, które można przypisać do obiektów są podzielone na poszczególne kategorie:

- Analytics - komponenty związane z przekazywaniem zdarzeń do zewnętrznych usług Analytics Service w celach statystycznych bądź przeprowadzenia analizy danych;
- AR (*ang. Augment Reality*) - komponenty dedykowane obsłudze rozszerzonej rzeczywistości z wykorzystaniem narzędzi ARCore oraz ARKit;



Rysunek 2.8: 1. Okno widoku kalibracji. 2. Komponent PupilSettings odpowiedzialny za konfigurację połączenia z PupilService i zawierający ustawienia dotyczące kalibracji systemu.

- Audio - komponenty odpowiedzialne za elementy audiowizualne, takie jak *AudioSource*, *AudioListener*, *AudioFilter*, itp.;
- Effect - komponenty wchodzące w skład efektów wizualnych, systemy *ParticleSystems* oraz efektów PostProcessowych;
- Event - systemy obsługi i przekazywania zdarzeń - *Event Trigger*, *Raycaster*, itp.;
- Layout - elementy graficznego interfejsu użytkownika (GUI);
- Mesh - komponenty odpowiedzialne za rysowanie siatki modeli trójwymiarowych;
- Miscellaneous - komponenty, które nie zostały przypisane do żadnych z kategorii;
- Navigation - elementy sztucznej inteligencji w zakresie wyszukiwania ścieżek (*ang. Pathfinding*);
- Network - komponenty odpowiedzialne za obsługę protokołów sieciowych - wymiana pakietów i komunikacja pomiędzy klientami w przypadku gry sieciowej;
- Physics - elementy odpowiedzialne za obsługę fizyki w grze;
- Rendering - komponenty wchodzące w skład elementów świata - źródła światła, kamery, elementy PostProcessingu;
- Scripts - programy napisane w języku C# i UnityScript;
- Tilemap - komponenty dedykowane aplikacjom w widoku 2D;
- UI - elementy graficznego interfejsu użytkownika - *Canvas*, *Textbox*, *Buttons*, *Slider*, itp.

2.4.1 Test

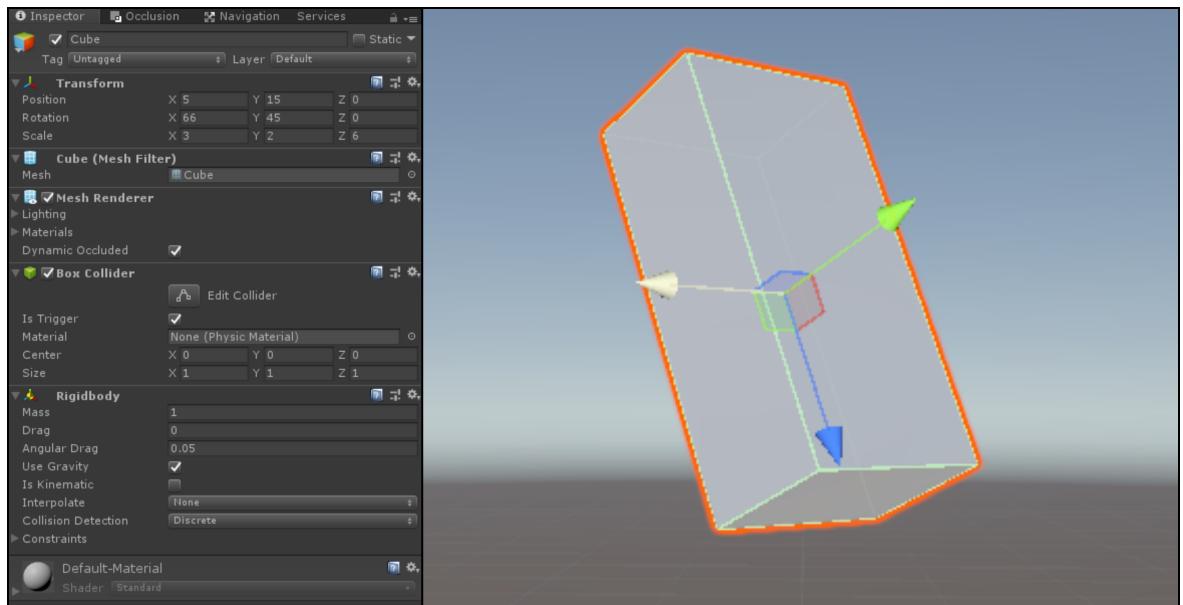
```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

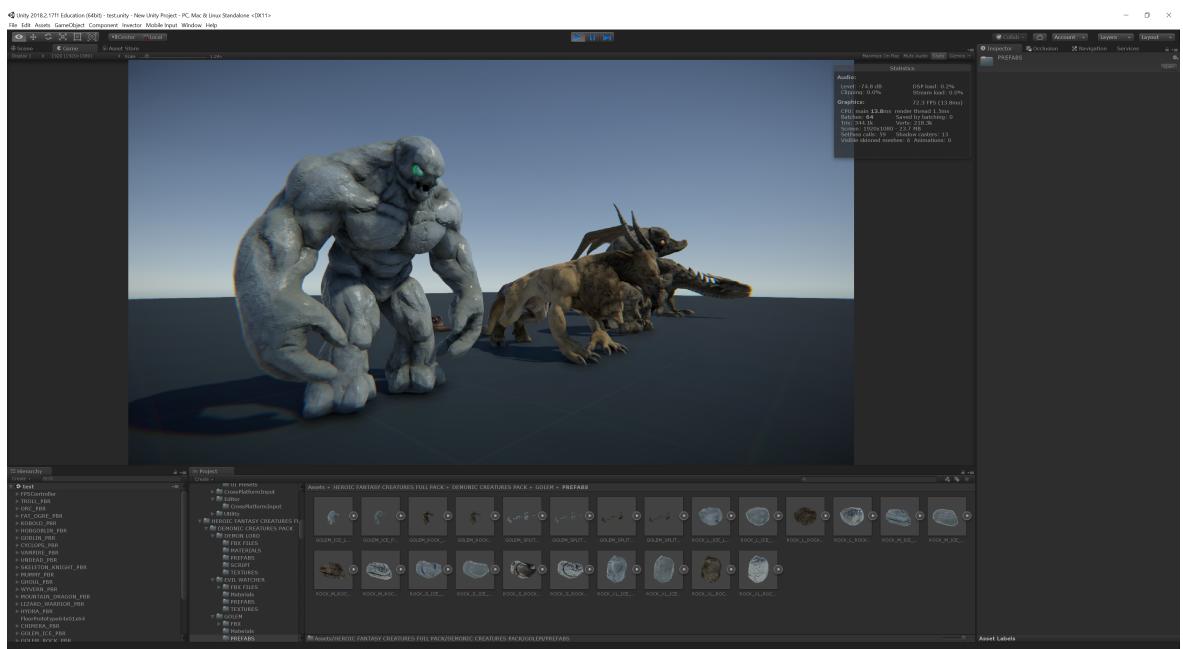
namespace GazeInteractionEngine
{
    public enum EVENT_TYPE
    {
        ON_EYE_INTERACTION,
        ON_EYE_STOP_INTERACTION,
    }

    public interface IGazeListener
    {

```



Rysunek 2.9: Przykład obiektu *Cube* składającego się z komponentów: *Transform*, *Mesh Filter*, *Mesh Renderer*, *Box Collider* oraz *Rigidbody*



Rysunek 2.10: Interfejs graficzny środowiska Unity 2018.2.17f

```
    void OnEvent(EVENT_TYPE eventType, Component Sender, Object param  
        = null);  
}  
}
```

```
using System.Collections;  
using System.Collections.Generic;  
using UnityEngine;  
using UnityHelpers;
```

```
namespace GazeInteractionEngine  
{
```

```
    public class GazeInteractionManager : MonoSingleton<GazeInteractionManager>  
    {  
        private Camera _sceneCamera;  
        private CalibrationDemo _calibrationDemo;  
  
        private LineRenderer _heading;  
  
        [ SerializeField ]  
        private Vector3 _standardViewportPoint = new Vector3(0.5f, 0.5f, 10);  
  
        private Vector2 _gazePointLeft;  
        private Vector2 _gazePointRight;  
        private Vector2 _gazePointCenter;  
        private Vector3 _viewportPoint;  
        private Vector3 _gazeWorldPosition;  
  
        private Vector3 _gazeEmulateNoise;  
        private Vector3 _gazeEmulatePosition;  
  
        public bool GazeEmulate = true;  
        private IGAzeListener _currentListener;
```

```
        public Vector2 ViewPortPoint  
        {  
            get { return _viewportPoint; }  
        }
```

```

    }

public Vector3 gazeWorldPosition
{
    get { return _gazeWorldPosition; }
}

void Start()
{
    PupilData.calculateMovingAverage = false;

    _sceneCamera = gameObject.GetComponent<Camera>();
    _calibrationDemo = gameObject.GetComponent<CalibrationDemo>();
    _heading = gameObject.GetComponent<LineRenderer>();

    if(GazeEmulate)
    {
        StartCoroutine(GazeEmulateNoise());
    }
}

IEnumerator GazeEmulateNoise()
{
    while(GazeEmulate)
    {
        _gazeEmulatePosition = new Vector3(Random.Range(0.0f, 1.0f),
            Random.Range(0.0f, 1.0f), 10);
        yield return new WaitForSeconds(Random.Range(0.3f,2.0f));

    }
}

void OnEnable()
{
    if (PupilTools.isConnected)
    {
        PupilTools.IsGazing = true;
        PupilTools.SubscribeTo("gaze");
    }
}

void Update()
{
    if (PupilTools.isConnected && PupilTools.IsGazing)
}

```

```

{
    _gazePointLeft = PupilData._2D.GetEyePosition(_sceneCamera,
        PupilData.leftEyeID);
    _gazePointRight = PupilData._2D.GetEyePosition(_sceneCamera,
        PupilData.rightEyeID);
    _gazePointCenter = PupilData._2D.GazePosition;

    Vector3 gazeTargetPos = new Vector3(_gazePointCenter.x,
        _gazePointCenter.y, 10.0f);
    _viewportPoint = Vector3.Lerp(_viewportPoint, gazeTargetPos, 10.0f
        * Time.deltaTime);

}
else
{
    Vector3 gazeNoise = new Vector3(Random.Range(-0.06f, 0.06f),
        Random.Range(-0.06f, 0.06f), 0);

    if (GazeEmulate)
        _viewportPoint = Vector3.Lerp(_viewportPoint,
            _gazeEmulatePosition + gazeNoise, 25.0f * Time.deltaTime);
    else
        _viewportPoint = Vector3.Lerp(_viewportPoint,
            _standardViewportPoint + gazeNoise, 25.0f * Time.
            deltaTime);

}

if (Input.GetKeyUp(KeyCode.L))
    _heading.enabled = !_heading.enabled;

_heading.SetPosition(0, _sceneCamera.transform.position -
    _sceneCamera.transform.up);

float thickness = .2f;

Ray ray = _sceneCamera.ViewportPointToRay(_viewportPoint);
RaycastHit hit;
if (Physics.SphereCast(ray, thickness, out hit))
{
    _gazeWorldPosition = hit.point;

    if (hit.transform.GetComponent<IGazeListener>() != null)
    {
        IGazeListener listener = hit.transform.GetComponent<

```

```

        IGazeListener>();
        listener .OnEvent(EVENT_TYPE.ON_EYE_INTERACTION,
        this, null);

        if( _currentListener != null && _currentListener != listener )
            _currentListener .OnEvent(EVENT_TYPE.
                ON_EYE_STOP_INTERACTION, this, null);

        _currentListener = listener ;

    }

    _heading.SetPosition(1, hit .point);
}
else
{
    _gazeWorldPosition = Vector3. positiveInfinity;

    _heading.SetPosition(1, ray. origin + ray.direction * 50f);

    if ( _currentListener != null )
    {
        _currentListener .OnEvent(EVENT_TYPE.
            ON_EYE_STOP_INTERACTION, this, null);
        _currentListener = null;
    }
}

}

}

```

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Events;
using UnityEngine.UI;

```

```

namespace GazeInteractionEngine
{

```

```

public class GazeInteraction_InteractableButton : MonoBehaviour,
    IGazeListener
{

```

```
#region Variables
```

```

    #region Parameters
public float reponseTime = 0.3f;
public float maxThresholdTime = 0.2f;
public float focusTime = 0.8f;
public float buttonTimeout = 1.0f;
public Image buttonBackgroundImage;
#endregion

```

```

    #region Private Variables
private float _currentTime = 0.0f;
private float _currentFocusTime = 0.0f;
private float _currentThresholdTime = 0.0f;
private bool _isInInteraction = false;
private bool _isInThreshold = false;
#endregion

```

```

    #region Events
public UnityEvent OnButtonPressed;
public UnityEvent OnButtonHover;
#endregion

```

```
#endregion
```

```
#region Functions
```

```

    #region IGAzeListener Interface

public void OnEvent(EVENT_TYPE eventType, Component Sender, Object
param = null)
{
    switch (eventType)
    {
        case EVENT_TYPE.ON_EYE_INTERACTION:
        {
            OnEyeInteraction();
            break;
        }
        case EVENT_TYPE.ON_EYE_STOP_INTERACTION:
        {
            OnEyeStopInteraction();
            break;
        }
    }
}

#endregion

```

```

#region Interactions
private void OnEyeInteraction()
{
    //Poczatek interakcji wzroku z obiektem
    if (_isInInteraction == false)
    {
        _currentTime += Time.deltaTime;

        if (_currentTime >= reponseTime)
        {
            _currentTime = 0.0f;
            _isInInteraction = true;
            if (OnButtonHover != null)
                OnButtonHover.Invoke();
        }
    }
    //Jak dluzej sie patrze na obiekt
    else
    {
        _isInThreshold = false;
        _currentFocusTime += Time.deltaTime;

        buttonBackgroundImage.fillAmount = (_currentFocusTime /
            focusTime);

        if (_currentFocusTime >= focusTime)
        {
            _currentFocusTime = 0.0f;
            StartCoroutine(PressedButton());

            if (OnButtonPressed != null)
                OnButtonPressed.Invoke();
        }
    }
}

private void OnEyeStopInteraction()
{
    _isInThreshold = true;
    _currentTime = 0.0f;
    StartCoroutine(ThresholdCounter());
}
#endregion

```

```

#region IEnumerators

private IEnumerator ThresholdCounter()
{
    _currentThresholdTime = maxThresholdTime;

    while (_isInThreshold)
    {
        _currentThresholdTime -= Time.deltaTime;

        if (_currentThresholdTime <= 0.0f)
        {
            _isInInteraction = false;
            currentTime = 0.0f;
            _isInThreshold = false;
            StartCoroutine(ThresholdButtonCounter());
            break;
        }

        yield return new WaitForEndOfFrame();
    }
}

private IEnumerator ThresholdButtonCounter()
{
    while (_currentFocusTime > 0.0f && !_isInInteraction)
    {
        _currentFocusTime -= Time.deltaTime;
        buttonBackgroundImage.fillAmount = (_currentFocusTime /
            focusTime);
        yield return new WaitForEndOfFrame();
    }
}

private IEnumerator PressedButton()
{
    Animator animator = buttonBackgroundImage.GetComponent<
        Animator>();

    animator.SetBool("animate", true);
    this.GetComponent<BoxCollider>().enabled = false;
    yield return new WaitForSeconds(buttonTimeout);
}

```

```

        this.GetComponent<BoxCollider>().enabled = true;
        animator.SetBool("animate", false);

        buttonBackgroundImage.fillAmount = 0;
    }
#endregion

#endregion

}

```

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Events;

```

```

namespace GazeInteractionEngine
{

```

```

    public class GazeInteraction_InteractableObject : MonoBehaviour,
        IGGazeListener
    {

```

```
#region Variables
```

```
#region Parameters
```

```
public float reponseTime = 0.5f;
public float maxThresholdTime = 0.25f;
#endregion
```

```
#region Private Variables
```

```
private float _currentTime = 0.0f;
private float _currentThresholdTime = 0.0f;
private bool _isInInteraction = false;
private bool _isInThreshold = false;
#endregion
```

```
#region Events
```

```
public UnityEvent OnStartInteraction;
public UnityEvent OnStayIteration;
public UnityEvent OnStopInteraction;
#endregion
```

```
#endregion
```

```
#region Functions
```

```
#region IGazeListener Interface
public void OnEvent(EVENT_TYPE eventType, Component Sender, Object
    param = null)
{
    switch (eventType)
    {
        case EVENT_TYPE.ON_EYE_INTERACTION:
        {
            OnEyeInteraction();
            break;
        }
        case EVENT_TYPE.ON_EYE_STOP_INTERACTION:
        {
            OnEyeStopInteraction();
            break;
        }
    }
}
```

```
#endregion
```

```
#region Interactions
private void OnEyeInteraction()
{
    //Poczatek interakcji wzroku z obiektem
    if( _isInInteraction == false)
    {
        _currentTime += Time.deltaTime;

        if ( _currentTime >= reponseTime)
        {
            _currentTime = 0.0f;
            _isInInteraction = true;
            if (OnStartInteraction != null)
                OnStartInteraction.Invoke();
        }
    }
    //Jak dluzej sie patrze na obiekt
    else
    {
        _isInThreshold = false;
```

```

        }

    }

private void OnEyeStopInteraction()
{
    _isInThreshold = true;
    _currentTime = 0.0f;
    StartCoroutine(ThresholdCounter());
}

#endregion

#region IEnumerators
private IEnumerator ThresholdCounter()
{
    _currentThresholdTime = maxThresholdTime;

    while(_isInThreshold)
    {
        _currentThresholdTime -= Time.deltaTime;

        if(_currentThresholdTime <= 0.0f)
        {
            if (OnStopInteraction != null)
            {
                OnStopInteraction.Invoke();
            }

            _isInInteraction = false;
            _currentTime = 0.0f;
            _isInThreshold = false;
        }

        yield return new WaitForEndOfFrame();
    }
}

#endregion

}

}

```

2.5 Podsumowanie

Rozdział 3

Implementacja systemu

3.1 Opis środowiska

3.2 Format danych wejściowych i wyjściowych

3.3 Opis implementacji

3.3.1 Struktura projektu

3.4 Podsumowanie

Przedstawiony rozdział opisuje jedynie niewielką część implementacji, która była głównie skupiona na elementach odpowiedzialnych za obsługę gry pod technikę wirtualnej rzeczywistości i kontrolerów ruchu. Cały przygotowany projekt gry składał się **88 klas** napisanych w języku C# zawierających łącznie **5283 linii kodu**, które są odpowiedzialne za całą logikę oraz przebieg gry. Istotnym zmianom uległ również proces przebudowania struktury sceny, graficznego interfejsu użytkownika oraz rozmieszczenia poszczególnych elementów gameplay'u.

Korzystając z zaimplementowanego projektu użytkownik może *wgląbić się* w wirtualny świat gry i bronić osady przed nacierającymi falami przeciwników poprzez umieszczanie obiektów obronnych wykorzystując system interakcji jak i obsługi gestów. Przygotowany projekt w pełni odpowiada na cel pracy. W jego ramach udało się zrealizować prototyp gry komputerowej wykorzystującej google wirtualnej rzeczywistości i kontrolery ruchu.

Rozdział 4

Rezultaty i wnioski

4.1 Opis procedury testowej

4.2 Prezentacja rezultatów

4.3 Wnioski z przeprowadzonych badań

Rozdział 5

Podsumowanie

Bibliografia

- [1] Dokumentacja techniczna pupillabs. <https://docs.pupil-labs.com>, Data dostępu: 10.03.2019.
- [2] Informacje o produkcie htc vive na stronie stream. https://store.steampowered.com/app/358040/HTC_Vive/, Data dostępu: 10.03.2019.
- [3] Repozytorium do oprogramowania pupillabs dla systemów vr. <https://github.com/pupil-labs/hmd-eyes>, Data dostępu: 10.03.2019.
- [4] Htc vive dokumentacja techniczna, Data dostępu: 14.03.2019.

Dodatek A

Opis zawartości płyty DVD

Na dołączonej do pracy płycie DVD znajdują się:

Pliki:

- ppotrowski2018.pdf - praca inżynierska w formacie PDF.

Foldery:

- Bin - wykonywalna wersja aplikacji opracowanej w ramach pracy;
- LaTeXSource - pliki źródłowe pracy inżynierskiej w postaci projektu typu LaTeX;
- Project - pliki projektu gry napisanego w Unity Engine;
- Scripts - wydzielone pliki skryptów odpowiedzialnych za cały przebieg gry.