Assignment of bachelor's thesis

Title:	Unity module for MultiLeap library
Student:	Radoslav Kondáč
Supervisor:	Ing. Tomáš Nováček
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Computer Graphics
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2022/2023

Instructions

With the rise of virtual reality, hand tracking is becoming more critical for intuitive VR interaction. Use the MultiLeap library and Unity game engine to create VR scenes to show the possibilities of controlling virtual worlds with bare hands.

Goals of the thesis:
1) Analyse user interaction possibilities with the virtual environment using hand and finger movement detection, emphasising the Leap Motion sensor and MultiLeap library.
2) Analyse possibilities of Unity game engine for creating VR worlds.
3) Create a Unity module to connect the MultiLeap Library with the Unity game engine.
4) Create at least two Unity scenes to show the possibilities of interaction using the Unity module mentioned above.
5) Perform testing with users to verify the intuitiveness of the Unity scenes.

Electronically approved by Ing. Radek Richtr, Ph.D. on 23 May 2021 in Prague.

Bachelor’s thesis

Unity module for MultiLeap library

Radoslav Kondáč

Department of Software Engineering
Supervisor: Ing. Tomáš Nováček

June 21, 2022

Acknowledgements

I would like to thank my supervisor, Ing. Tomáš Nováček, for his guidance, advice, expertise and unending patience and my dearest friends Jozef Bugoš and Jan Peřina, for their thoughtful insights and support.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. I further declare that I have concluded an agreement with the Czech Technical University in Prague, on the basis of which the Czech Technical University in Prague has waived its right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act. This fact shall not affect the provisions of Article 47b of the Act No. 111/1998 Coll., the Higher Education Act, as amended.

In Prague on June 21, 2022 …………………

Czech Technical University in Prague
Faculty of Information Technology
© 2022 Radoslav Kondáč. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Kondáč, Radoslav. *Unity module for MultiLeap library.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Abstrakt

Táto bakalárska práca skúma možnosti interakcie užívateľa s virtuálnymi svetmi a scénami vytvorenými v hernom engine Unity, primárne za použitia Leap Motion senzorov a knižnice MultiLeap. Výsledkami práce sú plugin pre Unity, ktorý sprístupňuje možnosť použitia viacerých Leap Motion senzorov na presné snímanie rúk v reálnom priestore a ich následné premietnutie do priestoru virtuálneho, a dve testovacie scény v Unity slúžiace ako demo ukážka.

Klíčová slova virtuálna realita, rozšírená realita, Leap Motion, MultiLeap, Unity3D, C#

Abstract

This bachelor thesis explores the possibilities of interaction between user and virtual environment and scenes made in Unity game engine, primarily by using Leap Motion sensors and MultiLeap library. The results are Unity plugin, which enables the use of multiple Leap Motion sensors to accurately track hands in real space and subsequently project these in virtual space, and two testing Unity scenes used as a demo showcase.

Keywords virtual reality, augmented reality, Leap Motion, MultiLeap, Unity3D, C#

Contents

Introduction **1**
Motivation 1
Objectives 2

1 Analysis of current technologies **3**
1.1 Leap Motion 3
1.1.1 Hardware 3
1.1.2 Software 3
1.2 MultiLeap 4
1.2.1 Alignment of coordinate systems 5
1.2.2 Merging 6
1.2.3 Architecture 6

2 Unity Game Engine **9**
2.1 Overview of Unity 9
2.2 VR worlds 10
2.3 Ultraleap package for Unity 10

3 Implementation **17**
3.1 Proposed architecture 17
3.1.1 MultiLeap library integration 17
3.1.2 Physics 19
3.1.3 UI 20
3.1.4 Results 21

4 Testing **23**
4.1 Testing scenes 23
4.2 Sensor setup 25
4.3 Testing scenarios 25
4.4 Results 26

4.5 Problems 27

5 Future work **29**

Conclusion **31**

Bibliography **33**

A Acronyms **35**

B Contents of enclosed SD card **37**

List of Figures

1.1 Stereo IR 170 sensor[1] 4
1.2 Virtual hands from three non calibrated sensors 7
1.3 Virtual hands from three calibrated sensors 7
1.4 Merged virtual hand from three calibrated sensors 8

2.1 Unity application life cycle[2] 11
2.2 Example scene provided by Ultraleap 12
2.3 (1) Service provider, (2) Hands, (3) Interaction manager 14
2.4 Working setup with physics 14

3.1 New class structure 19
3.2 Merged hands with enabled physics 19
3.3 Project structure at runtime. (1) Before merging, (2) After merging 21
3.4 Simple UI 21

4.1 Sandbox 23
4.2 Castle 24
4.3 Sensor setup 25
4.4 Sensor setup with visible projection 26
4.5 User interacting with elements in Sandbox scene 26
4.6 Hand with incorrect handedness 27
4.7 Incorrectly evaluated left hand with flipped right hand superimposed on top 28

List of Code Examples

2.1 Polling loop 13
2.2 Start and Update 15

3.1 Frames 18
3.2 Controllers 20

4.1 Cubes 24

Introduction

Ever since the introduction of Virtual reality (VR), people tried to manifest their desire to interact with more than just what the real world had to offer. As a concept, it provided means to experience experiences not available or even not possible in reality, for example the thrill of flight, frolicking with dinosaurs on the surface of the sun, or being a bounty hunter in a dystopian future.

The technology was first used by militaries for wargame simulations [3]. Its availability to masses was hindered by high hardware requirements, and the concept was partially abandoned. Nevertheless, with the advent of personal computers and fast improvement of their computational capabilities, virtual reality was revisited and reintroduced to the public. What was once a sovereign domain of the wealthy is now widely available.

Virtual reality provides the ability to fully immerse oneself in a completely artificial world. However, while the concept is undeniably alluring, it has it's limitations. Virtual world has to be built from scratch, the necessity of using various peripherals, like hand-held controllers or VR headsets to interact with it degrades the overall experience and we are limited to senses of sight and hearing. These were among the reasons for the development of Augmented reality (AR) and Mixed reality (MR). Definition of augmented reality can be as follows: "Augmented reality (AR) is an enhanced version of the real physical world that is achieved through the use of digital visual elements, sound, or other sensory stimuli delivered via technology." [4]. Mixed reality is a mix of virtual reality and augmented reality. All of these are nowadays referred to as Extended reality (XR).

Motivation

With the aforementioned revival of extended reality, companies are trying to negate the shortcomings of already established technologies, be it by extending

the sensory feedback, like haptics [5], or reducing the number of peripherals or wearables needed to interact with the virtual world.

In this thesis, I explore the possibilities of the latter, that is the option to interact with computers without the need to wear any sensors on person. Applications of said technologies range from game industry, where it is already possible, for example, to visit virtual escape rooms or mazes (in this example, VR headsets are used for better immersion), through general industry, where XR technologies are used for the training of the workforce, to service industry, with contact-less kiosks being a fine example.

However, the use of contact-less sensors to track hand data in their current form is not without problems, namely limited range and inaccuracy in edge cases of hand tracking, for example when the palm is perpendicular to the sensor. These can be alleviated by using multiple sensors to track the hand, thereby improving accuracy and extending range. This solution was explored by MultiLeap library[6].

Objectives

My primary objectives are:

- Analyse the current state of preferred technologies.
- Explore usability of Unity Game Engine to create virtual worlds.
- Integrate the MultiLeap library (MLL) with Ultraleap's Unity plugin.
- Create demo environment and subsequently test the intuitiveness of the solution with users.

In the first chapter, I will provide analysis of current resources, namely Leap Motion software and hardware and MultiLeap library. In the second chapter, I will describe the possibilities of Unity Game Engine in regards to creating virtual environment. Third chapter contains the description of my solution, the next chapter encompasses my findings from subsequent testing and the final chapter is dedicated to future work.

Chapter 1

Analysis of current technologies

In this chapter, I will be looking into the technologies that I will primarily use in my implementation. I will describe available software and hardware and their capabilities.

1.1 Leap Motion

Leap Motion software and hardware, originally developed by the company Leap Motion, which after merging with Ultrahaptics became Ultraleap, provides complete set needed for real-time tracking of hands and their subsequent representation in virtual environment. The set consists of an optical sensor and software for receiving images from the sensor and creating hand tracking data from them. In my work, I used three Stereo IR 170 sensors[1].

1.1.1 Hardware

The sensor projects an infrared (IR) light invisible to human eye in a relatively wide cone of 170x170 degrees and up to effective distance of 80 centimetres using two IR LEDs, which is then detected by two monochrome IR cameras. Based on the reflected light, hand position and pose is computed. This data is then interpreted by Ultraleap Gemini hand tracking service[7], which provides pre-processed hand data via C-like LeapC API[8] for further use. Tracking software also sends information about the sensor itself, like its connection status or error messages. [6]

1.1.2 Software

Hand data is sent in structured format in the form of frames, with information about the hand models centre in palm, fingers, forearm bones and such. Compensation for ambient light and removal of unwanted elements, like the head, is also handled by the software, so the resulting outgoing information is purely

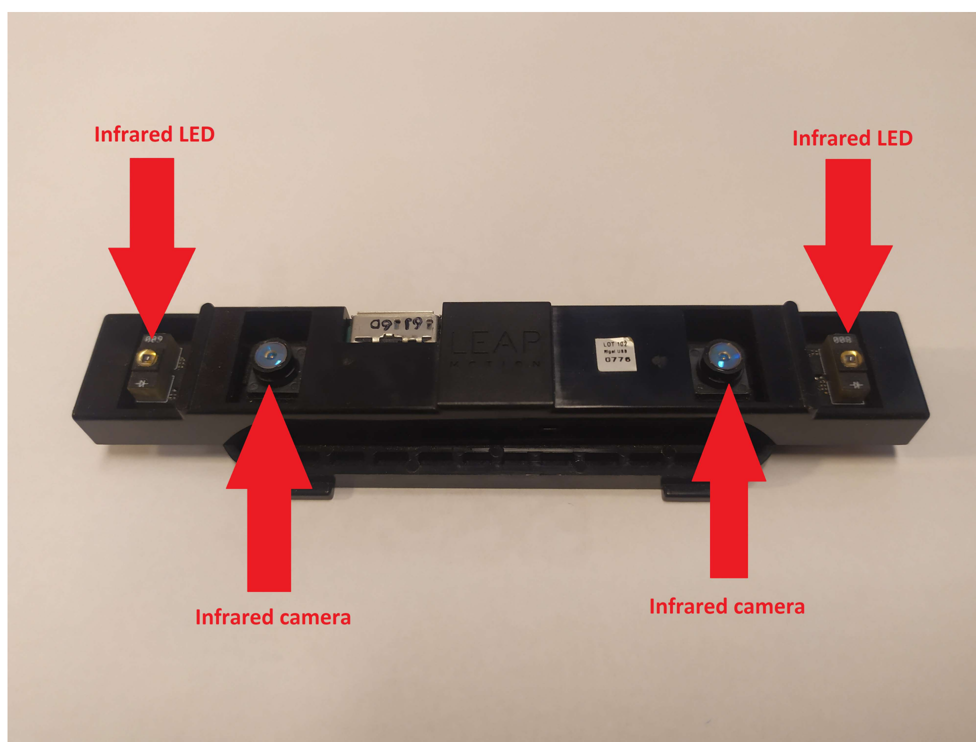

Figure 1.1: Stereo IR 170 sensor[1]

about hand-like objects. Hand models position is represented in right-handed Cartesian coordinate system, with the x- and z-axes lying in the cameras plane and y-axis representing depth. X- and y- axes are analog to Unity, with z-axis being mirrored. If the sensor can not see a part of the tracked hand, either due to it being occluded by another part of the hand or it being beyond the sensors radius, the provided software estimates the missing parts, using either interpolation from older frames or extrapolation for future ones.

Pre-processed tracking data obtained from LeapC API are stored in a queue. Frames from this queue must be efficiently processed further, otherwise information loss occurs.

Ultraleap also provides a simple visualizer and modules for both Unreal and Unity game engines. Unity module will be discussed in further detail in later chapters.

1.2 MultiLeap

By default, software provided by Ultraleap is capable of managing only one sensor at a time. However, in 2018, an experimental build was released, which supported multiple connected sensors at the same time[9]. This functionality was again revisited and made available in 2022 production release[7]. The

aforementioned iteration was able to send information and hand tracking data from each connected sensor separately. The ability to obtain information from multiple sensors at once opened the possibility of tracking the hands of one user by multiple sensors. MLL makes use of this, by adding the option to handle multiple sensors at the same time, and consequently merge their coordinate systems to one shared between all connected sensors and then fuse hands tracked by each sensor into one. MultiLeap library and encompassing software is based on functionality added in 2018 release, which at the time of implementation of my solution was not changed since its initial release.

Each stream of tracking data provided this way is based in separate coordinate system, with centre corresponding to sensor from which it was obtained. To merge this data so that it represents one pair of hands, two steps are necessary. First one is aligning the sensors coordinate systems, called calibration, and the second is fusing tracking data together. Calibration has several approaches available .

First approach provided by Leap Motion in the experimental build from 2018 was manual calibration, without merging implemented. This form of calibration required the sensors to be placed in virtual world accurately relative to each other, with distances and angles between them corresponding to the distances in real world. This can prove quite difficult, because of exactness needed to obtain data of required quality, but also beneficial in case when done correctly. Benefits are high accuracy of the result, and also the possibility of placing the sensors in a way so they can track different parts of space, for example placed back to back, which is not possible in further approaches.

Second is in the form of semi-manual calibration, partially implemented in Leap Motion Unity module. This approach required the user to save separate frames, and once enough data was provided, the hands should have aligned. MLL makes this approach more automated. After the user starts the calibration process, software automatically samples the frames from each sensor, and after enough information is provided, the sensors coordinate systems align, so they match the coordinate system of the first connected sensor. In earlier versions, the tracked hand had to be visible to all sensors to achieve this, but in more recent releases, the sensors just have to share some part of the tracked space. The resulting accuracy is slightly lower compared to fully manual approach, but the process is a lot easier and more comfortable to the user. Better results can be achieved by combining the approaches, by first doing manual calibration and then automatic.

The precision of the calibration can be increased if the user moves the hand in the tracked space.

1.2.1 Alignment of coordinate systems

The process of calibration requires computing the relative positions of sensors to each other. Both in solutions by Leap Motion and MultiLeap, this was

achieved by using Kabsch algorithm[10]. Using the first connected sensor as its origin point, position of each subsequent sensor to the first is calculated[6].

After sampling enough frames, the coordinate systems converge. This result can be visually observed in tracked hands visual representations overlapping, meaning multiple instances of hands occupying the same virtual space. As a simplified visual aid, the more the hands overlap, the more accurate was the calibration.

1.2.2 Merging

Next step is fusing the data from all sensors together. Confidence is a float value ranging from 0 to 1 provided by MultiLeap, which determines the credibility of each sensor. This credibility is based on the position of the hand to the sensor, with highest confidence achieved by hand with palm aimed directly at the sensor and lowest with the palm being perpendicular, since in that position, most of the tracked points are occluded by each other. Relative rotation of the hand is calculated from angle between the sensors y-axis and palms normal, ranging between 0 and 180 degrees. Any values outside of this interval are converted to fall into said range.

In the fusion process, the confidence value is used for real-time computing of the hands position, with tracking data from sensors with higher confidence having more prevalence in the result. This value is never lower than 0.3. As long as the sensor still tracks the hand, even with the lowest confidence value, the information provided is still useful.

The result of this operation is a single set of tracked hands, seemingly obtained from virtual sensor with origin based on the first sensor. Merging of the hands can be enabled and disabled at runtime.

1.2.3 Architecture

MLL is directly based on LeapC library (LCL), encasing it and obtaining data from it, which it later modifies, and adding additional information to it. Both are written in C++ programming language, providing adequate performance. MLL provides API similar to underlying original, thus easing the subsequent implementation processes. Due to this, LCL can be completely omitted when using MLL. In addition, C# wrapper encasing MLL is available, which was vital in process of integration with Unity, since it supports scripting in the same language.

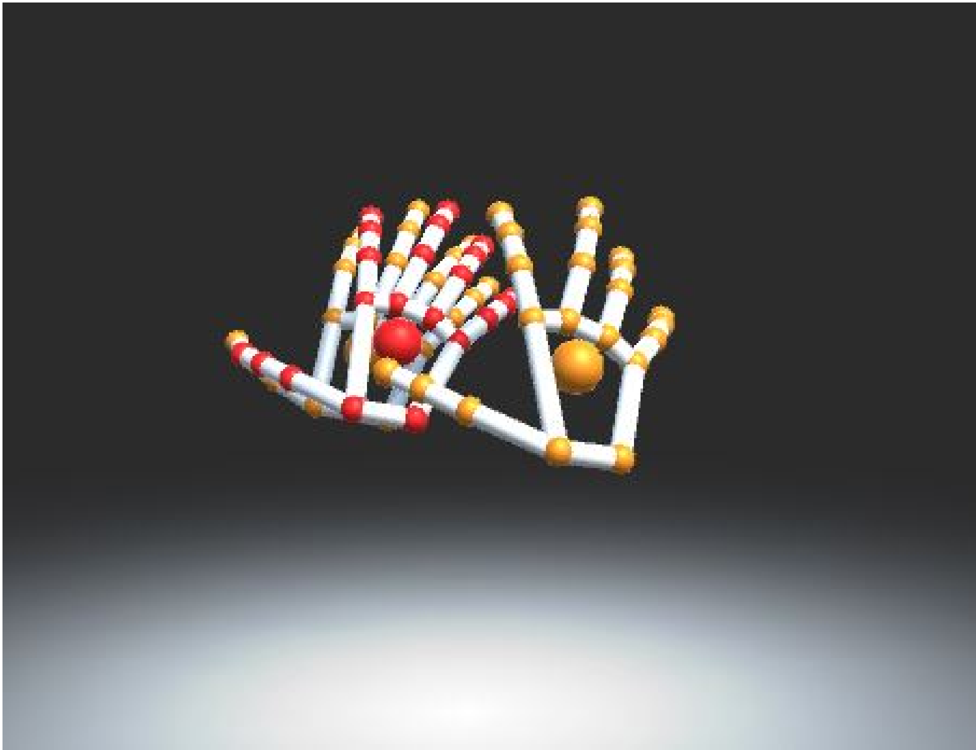

Figure 1.2: Virtual hands from three non calibrated sensors

Figure 1.3: Virtual hands from three calibrated sensors

Figure 1.4: Merged virtual hand from three calibrated sensors

Chapter 2

Unity Game Engine

In this chapter, I will briefly describe Unity Game Engine, it's capabilities and simplified process of creating virtual environments and lastly resources for said task provided by Ultraleap.

2.1 Overview of Unity

Unity can be briefly described as follows: "Unity is a cross-platform game engine developed by Unity Technologies, which is primarily used to develop video games and simulations for computers, consoles and mobile devices. First announced only for OS X, at Apple's Worldwide Developers Conference in 2005, it has since been extended to target 27 platforms." [11]. Unity is capable of creating 2D and 3D applications that can be deployed on more than 27 platforms, most commonly on PC, Android or iOS. Featured toolkit for designing games contains interfaces for graphics, audio and level-building.

The engine provides a visual editor, where user can manipulate game objects in 3D virtual space. The functionality of the world is facilitated by scripts, most commonly written in C#. For the script to run, it has to be attached to some object in the world and has to derive from predefined class MonoBehaviour. This is necessary, because the game engine expects all attached scripts to be controlled in a specified way, e.g. having defined methods for Unity events, like object creation, when it is clicked, pointed at and so on. These scripts extending MonoBehaviour can use other scripts not derived from the same class.

The life cycle of Unity script begins with initialization when scene loads, with Awake method called on object instantiation[1]. Since not all objects present in scene have to be active at initialization, OnEnable is called on active objects. If the object is inactive, execution waits until it is enabled.

[1] Creation of object instance

Call to Reset initializes scripts properties. Finally, Start method is called before the first frame update.

Next is the update cycle. This cycle repeats for every frame rendered by Unity. Cycle starts with FixedUpdate method that handles all physics calculations. All physics updates occur immediately after this method call. FixedUpdate can be called multiple times per frame update. Next, Update method is used to handle the majority of frame update operations. Lastly, LateUpdate occurs after the Update is finished. This can be used for example for third-person cameras, where player movement is calculated in Update and afterwards camera translation in LateUpdate, to ensure the player has stopped moving before the camera starts tracking his position. Update cycle is repeated for as long as the application is running or the object is active in the scene.

If the object is deactivated, OnDisable is invoked, and when destroyed, OnDestroy is called. These two run automatically if the application is closed.[2]

Additional functionality can be supplied by user made plugins. "A plugin is a software add-on that is installed on a program, enhancing its capabilities."[12]. Unity supports both native and managed plugins. Thanks to this, we can use precompiled Dynamic Link Libraries (DLL) written in other languages, for higher performance, reuse of code and access to content normally not available to Unity scripts, like information about drivers.

2.2 VR worlds

The basics of the process of creating a virtual world are quite similar to creating a standard 3D application. Unity, as one of the most popular game engines, provides various options for such tasks. Since most virtual spaces are based on the real world, meaning they observe similar if not the same physical laws, albeit simplified, the resulting world is intuitive for the user. To ease the process, many XR elements present in a virtual scene are available as premade collections, called prefabs. These can be for example objects like walls or collections of scripts attached to empty objects. A primitive scene can then be created by combining various of these prefabs.

In simplified terms, the main difference between XR world and a world present in standard 3D application is the users input and output source, with standard applications using common peripherals like mouse and keyboard for input and displays for output and the former using for example motion controllers as input and VR headsets as output.

2.3 Ultraleap package for Unity

Along with sensors used for tracking hands, Ultraleap offers free software for both Unity and Unreal for development on their hardware. Software for

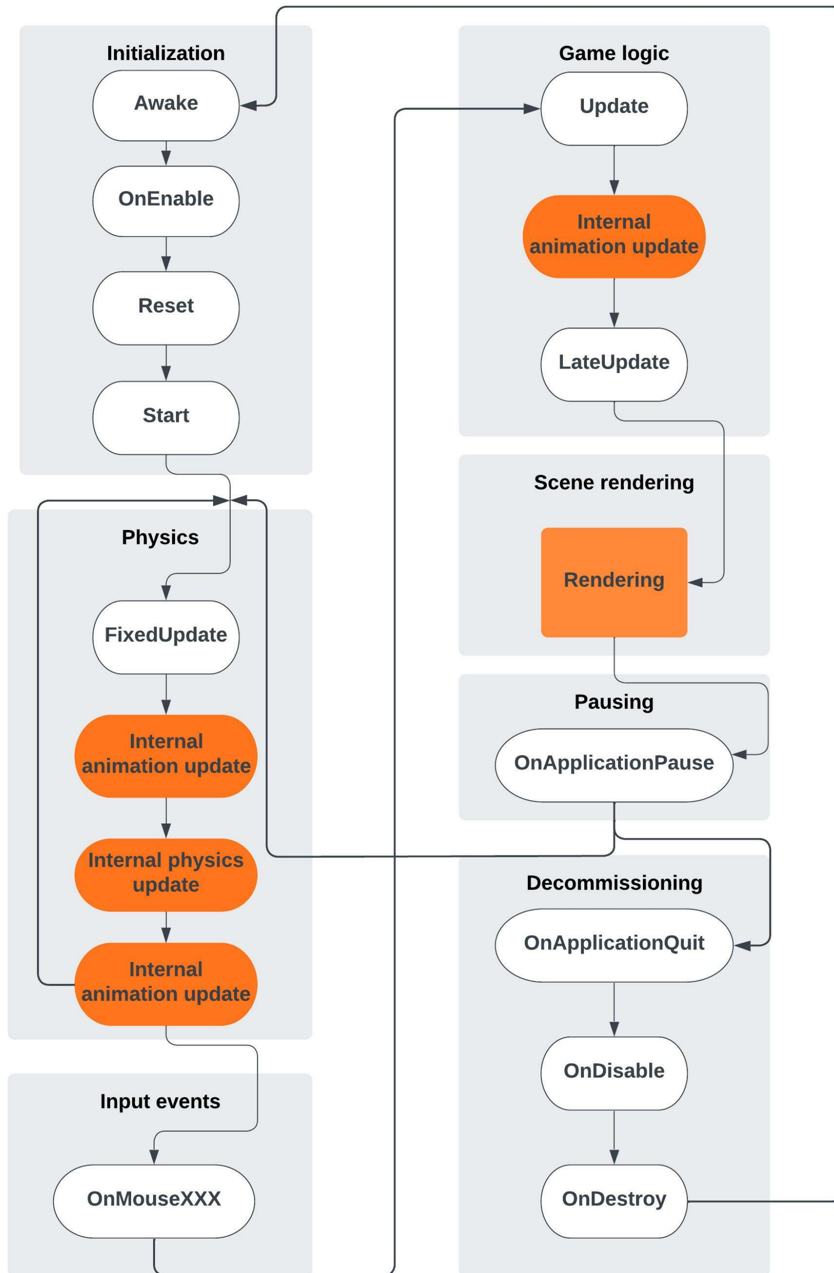

Figure 2.1: Unity application life cycle[2]

Unity is in a form of package[2], consisting of tracking core, hand models and

[2]Archive capable of installing itself into Unity project

interaction engine.

Tracking core contains LeapC library, along with C# wrapper exposing LeapC C++ code to Unity, and other prefabs for hands and controllers needed for basic visualization of virtual hands. Hand models contain various styles of hands usable by Leap Motion, with complete models, materials and bindings to the tracked bone elements. Lastly, interaction engine handles interactions between game objects and virtual hands, ranging from basic collisions to more advanced like grabbing an object or interaction with UI elements through hovering of the hand over them.

Additionaly, an examples package is available, containing prepared scenes with showcases of various features and functionalities of their solution.

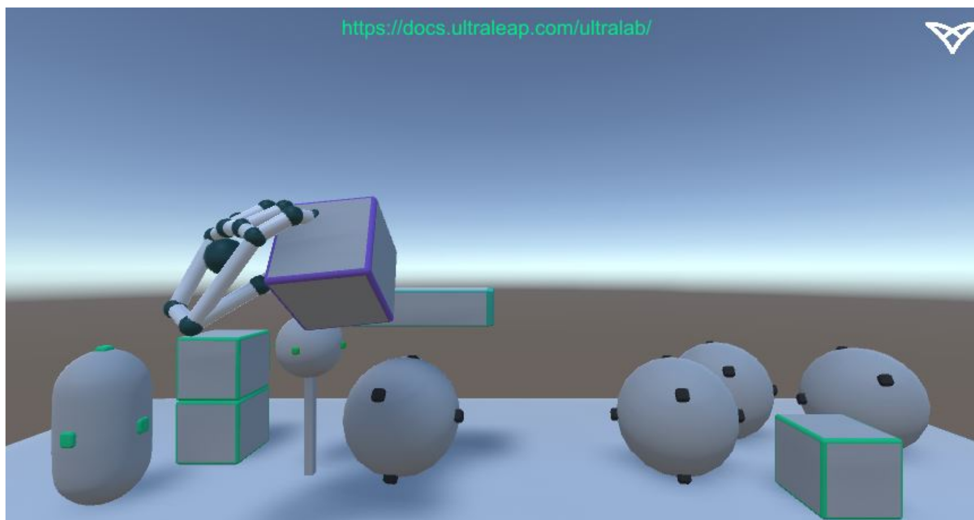

Figure 2.2: Example scene provided by Ultraleap

The sensor is represented in Unity by LeapServiceProvider (LSP), a child class of MonoBehaviour. This class serves as an interface to Unity, with any parts of the application using hand tracking data from the sensor needing a reference to this class to obtain it from it. This interface also provides additional information needed, like connection or device status. The sensor itself is run by class Controller, and the connection by class Connection.

Connection runs a polling loop on separate worker thread, handling incoming frames and sensor status messages, obtained from Ultraleap Gemini hand tracking service through calls to LCL C# wrapper. Obtained messages are then processed by the Controller and available through LSP. For every obtained frame, LSP dispatches an event. Various scripts listen to these events. These scripts then process the tracking data further, for example create hand representation in virtual space or calculate gestures like pinching or pointing.

Physics interactions are facilitated by a component called InteractionManager, with two child components InteractionHands, containing bones capable

```
private void processMessages()
{
    ...
    try
    {
        eLeapRS result;
        ...
        while (_isRunning)
        {
            ...
            LEAP_CONNECTION_MESSAGE _msg = new LEAP_CONNECTION_MESSAGE();
            ...
            switch (_msg.type)
            {
                case eLeapEventType.eLeapEventType_None:
                    break;

                case eLeapEventType.eLeapEventType_Connection:
                    LEAP_CONNECTION_EVENT connection_evt;
                    StructMarshal<LEAP_CONNECTION_EVENT>.PtrToStruct(
                        _msg.eventStructPtr, out connection_evt);
                        handleConnection(ref connection_evt);
                    break;
                ...
                case eLeapEventType.eLeapEventType_Tracking:
                    LEAP_TRACKING_EVENT tracking_evt;
                    StructMarshal<LEAP_TRACKING_EVENT>.PtrToStruct(
                        _msg.eventStructPtr, out tracking_evt);
                    handleTrackingMessage(ref tracking_evt, _msg.deviceID);
                    break;
                ...
            }
            ...
        }
    }
    catch (Exception e)
    {
        Logger.Log("Exception: " + e);
        _isRunning = false;
    }
    ...
}
```

Code 2.1: Polling loop processing messages from Leap Motion service

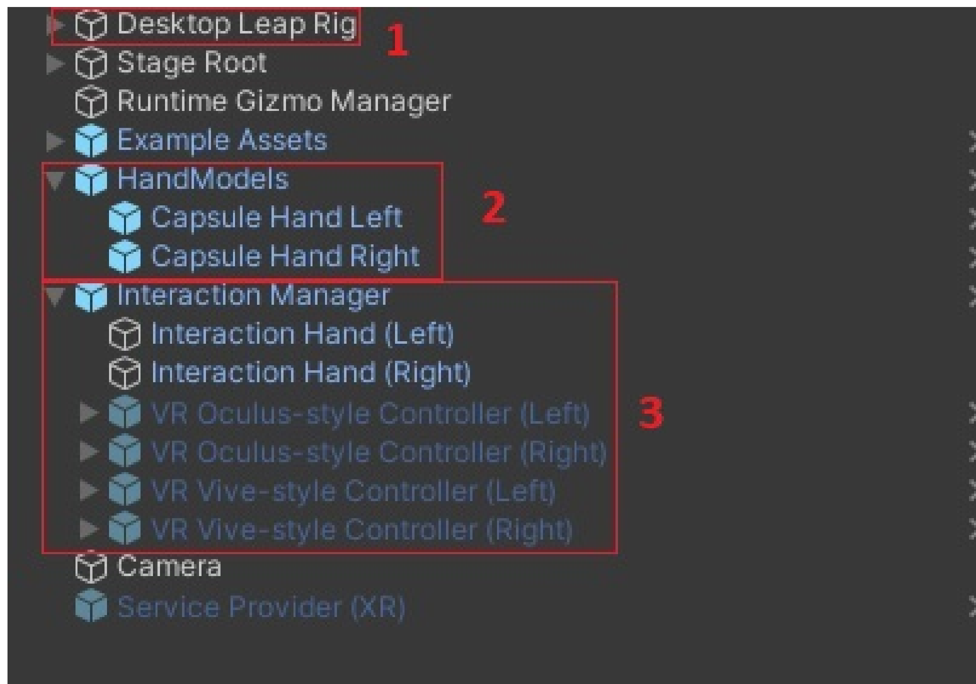

Figure 2.3: (1) Service provider, (2) Hands, (3) Interaction manager

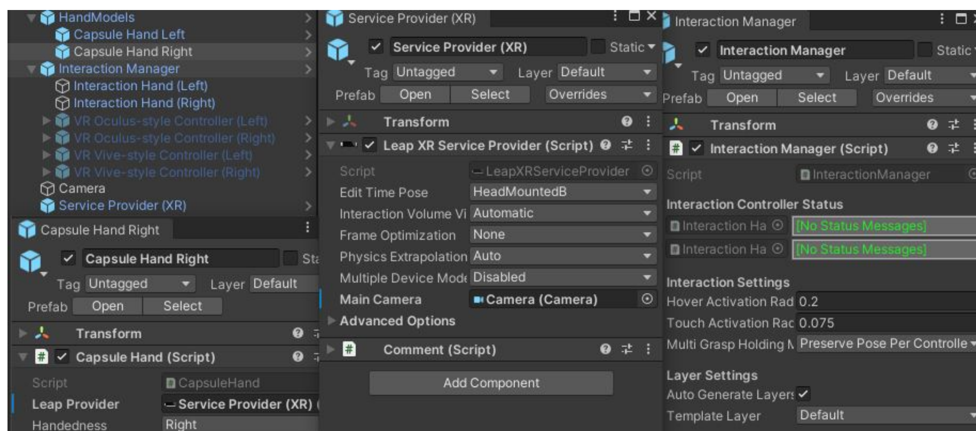

Figure 2.4: Working setup with physics

of interacting with objects, that subscribe to one LSP. These hands work in tandem with hand components used for visualization. Additionaly, all scene objects intended for interaction require InteractionBehaviour component. All these components are expected to be present in one scene only once, meaning InteractionManager can subscribe only to one LSP, which is expected to handle tracking data from one sensor.

Users can build a scene with tracked hands by adding three prefabs. One instance of Service provider out of three available is needed, with preset modes

```
protected virtual void Start()
{
    createController();
    _transformedUpdateFrame = new Frame();
    _transformedFixedFrame = new Frame();
    _untransformedUpdateFrame = new Frame();
    _untransformedFixedFrame = new Frame();
}

protected virtual void Update()
{
    ...
    if (!checkConnectionIntegrity()) { return; }
    ...
    if (_useInterpolation)
    {
    ...
        _leapController.GetInterpolatedFrameFromTime(_untransformedUpdateFrame,
            timestamp, CalculateInterpolationTime() - (BounceAmount * 1000));
    }
    else
    {
        _leapController.Frame(_untransformedUpdateFrame);
    }

    if (_untransformedUpdateFrame != null)
    {
        transformFrame(_untransformedUpdateFrame, _transformedUpdateFrame);

        DispatchUpdateFrameEvent(_transformedUpdateFrame);
    }
}
```

Code 2.2: Start and Update methods of LSP

for sensor placement, Desktop, Screentop or XR, meant for headset mounted sensors. Next, a prefab for hands, for example CapsuleHands. Lastly, one Interaction Manager prefab is needed for physics to work. Interaction Manager contains a set of Interaction Hand prefabs, along with prefabs for Oculus and Vive hand controllers.

Multiple device support can be set in the editor. This enables the plugin to work with multiple devices connected, but only by specifying one source sensor, from which tracking data are obtained. Other sensors are then ignored. All components dependent on each other resolve their references at initialization automatically. If no instance of required is present, component disables itself.

Benefits of this approach are a robust tracking core, able to be run with minimal configuration and user input. Its shortcomings become apparent when we try to add more sensor controllers to the scene, or create individual components not in intended order.

Chapter 3

Implementation

The chapter Implementation will describe my integration of MultiLeap library into Ultraleaps Unity plugin, along with some problems I encountered and had to resolve during the implementation.

3.1 Proposed architecture

As a base for my implementation, I used the plugin provided by Ultraleap and modified it to support MLL instead of LCL, to enhance the provided functionality and enable multi sensor support. For my implementation to work, I wanted to achieve several partial goals, namely:

- Modify the plugin so it uses MLL C# wrapper for communication with Ultraleap Gemini hand tracking service.
- Facilitate dynamic creation of sensor controllers, so any number of sensors can be used.
- Allow physics interaction between hands and scene objects.
- Create simple UI for configuring the resulting prefab at runtime.

The modifications to the plugin were intended to be minimal, for easier integration with future plugin releases.

3.1.1 MultiLeap library integration

Multiple device support offered by the plugin only enabled the option to specifically set the source sensor for a controller. This approach was not usable with how the MLL is implemented. MLL wrapper provides an interface similar to LCL, and is capable of handling the connection to Leap Motion Tracking service[3] by itself. It also receives data from all connected sensors. Due to this,

[3]Older version of Ultraleap Gemini hand tracking service

I created new scene object used to handle the connection through the wrapper. If included in the scene, the underlying script Control runs a polling loop on separate worker thread, obtaining tracking Frames. These are then sent directly to the controller, thus circumventing the Connection component.

Next step was creating controllers dynamically at runtime, so each connected sensor would have one controller receiving tracking frames from it. This part required some modification of LSP, Controller and Connection components. All of these were intended to be present in the scene since its initialization and contained various checks to avoid being created later. I had to reroute error checking back to Control class and C# wrapper. In comparison to original, MLL creates one extra virtual controller, to which it sends fused tracking data after calibration is complete and merging of the hands is enabled.

Due to us not knowing the exact position of sensors in real world, all controllers are created in the same place to better match reality where hands are usually close to each other. After the merging is enabled, virtual controller is the only one sending data to the application and is located at the same position as the first sensor in real world. For each controller to receive the data from correct sensor, each LSP had to be assigned unique ID, resulting in further modification of the source code. Modified controllers are loaded as prefabs, so any change on the original prefab affects all its instances, thus speeding up the development process in order to accommodate the growing need of the usage of XR in real world.

Since the tracking information provided by MLL is similar to the original ones, no further modification of components handling hand visualization was necessary, except the aforementioned changes. Physics is the only challenge remaining. I will tackle this problem in the next part of this chapter.

```
case eLeapEventType.eLeapEventType_Tracking:
    {
        StructMarshal<LEAP_TRACKING_EVENT>.PtrToStruct(
          message.leapMessage.eventStructPtr, out LEAP_TRACKING_EVENT tracking_evt);
        Frame frame =
        new Frame(message.leapMessage.deviceID).CopyFrom(ref tracking_evt);
        StructMarshal<MultiLeap_TrackingMessageMeta>.PtrToStruct(
            message.trackingMessageMeta,
            out MultiLeap_TrackingMessageMeta trackingMessageMeta);
        OnFrame(frame, trackingMessageMeta, wrapper);
        ctrl.mtx.WaitOne();

        c = FindConn(frame, ctrl.conns);
        if (c != null) c.Frames.Put(ref frame);

        ctrl.mtx.ReleaseMutex();
        break;
    }
```

Code 3.1: Sending frames directly to controller

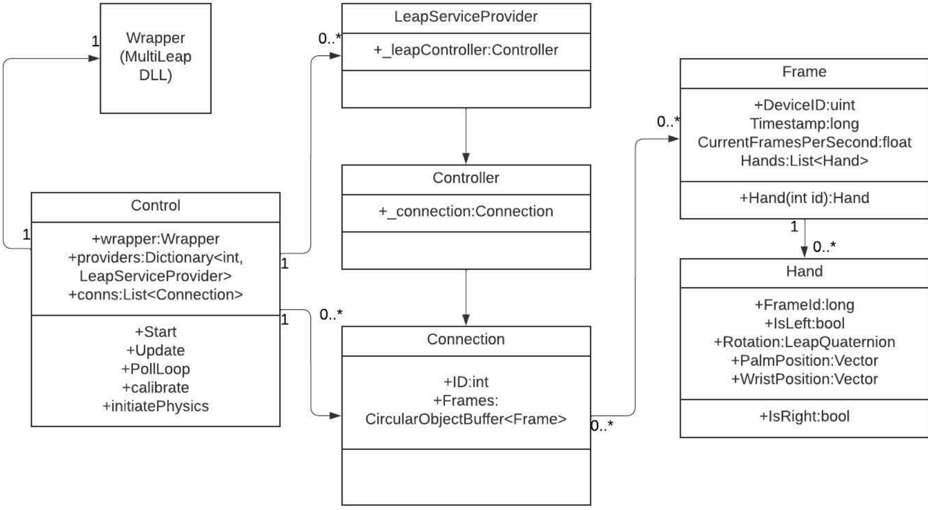

Figure 3.1: New class structure

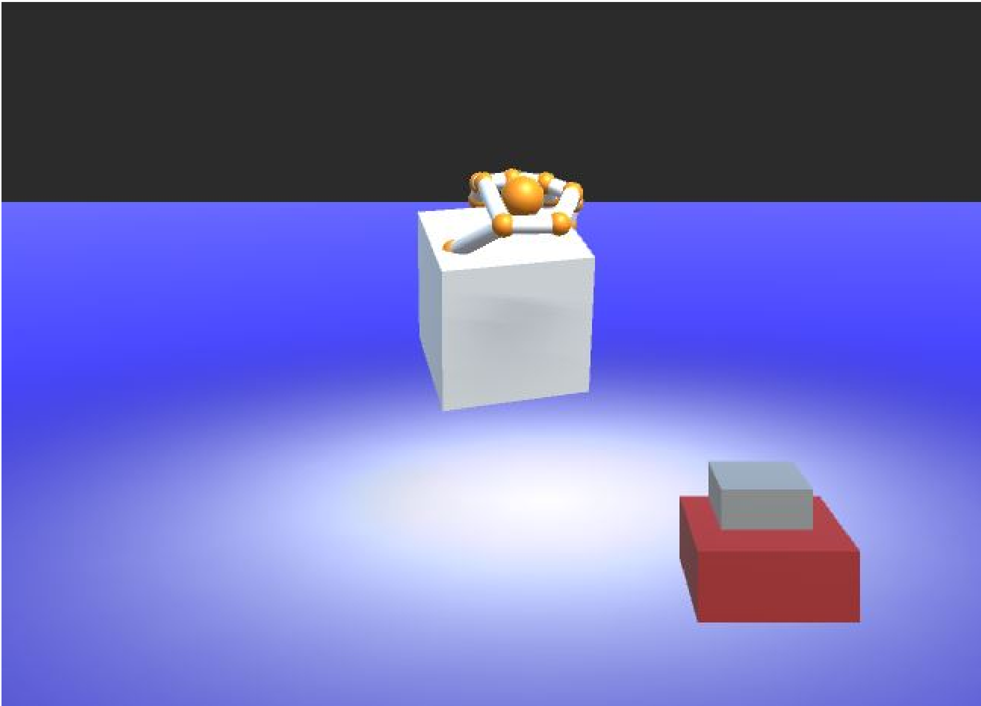

Figure 3.2: Merged hands with enabled physics

3.1.2 Physics

For physics to work, Ultraleap plugin requires a single instance of Interaction-Manager. If multiple instances are present, they become inactive and only

```
wrapper.GetDevices(ref devices, ref ids);
        Debug.Log("Found " + devices.Length + " devices.");
        text.text = "Found " + devices.Length + " devices.";
        for (int j = 0; j < devices.Length; j++)
        {
            Debug.Log("Device with id " + ids[j] + " has serial number " +
                devices[j].SerialNumber);
        }


        for (int i = 0; i < devices.Length; i++)
        {
            if (ids[i] != 42)
            {
                Vector3 posDefault = new Vector3(0, -0.15f, 0);
                Quaternion rot2 = Quaternion.identity;
                obj = (GameObject)Instantiate(Resources.Load("Wrap"),
                    posDefault, rot2);
                obj.transform.parent = this.transform;
                obj.name = "wrap" + ids[i];
                lss = obj.GetComponentInChildren<LeapServiceProvider>();
                lss.name = "rig" + ids[i];
                lss.wrapper = wrapper;
                lss.Start2(ids[i]);
                lss._leapController.wrapper = wrapper;

                lss._leapController.running = true;
                _conn = lss._leapController._connection;
                _conn.ID = ids[i];

                conns.Add(_conn);
                providers.Add(ids[i], lss);
                objects.Add(ids[i], obj);
            }
        }
```

Code 3.2: Creation of controllers at runtime

first remains running. Like in previous parts, it is required to be present in the scene at the time of initialization, otherwise InteractionBehaviour components on interactable objects become inactive. Due to this, it also had to be created at runtime, but since only one InteractionManager is allowed in a scene, a decision was made to enable interaction with objects only when using merged tracking data. Furthermore, all InteractionBehaviour components are assigned to objects only after first merging, thus enabling physics. All objects intended for interaction require to be assigned specific tag in Unity editor.

3.1.3 UI

For testing purposes, I created a basic User Interface (UI), with displayed status messages and explanation of controls used in testing scenes. After application starts, user is able to run calibration of devices, save this calibration or load a previous one, and enable or disable the merging process. Controls are bound to keyboard keys.

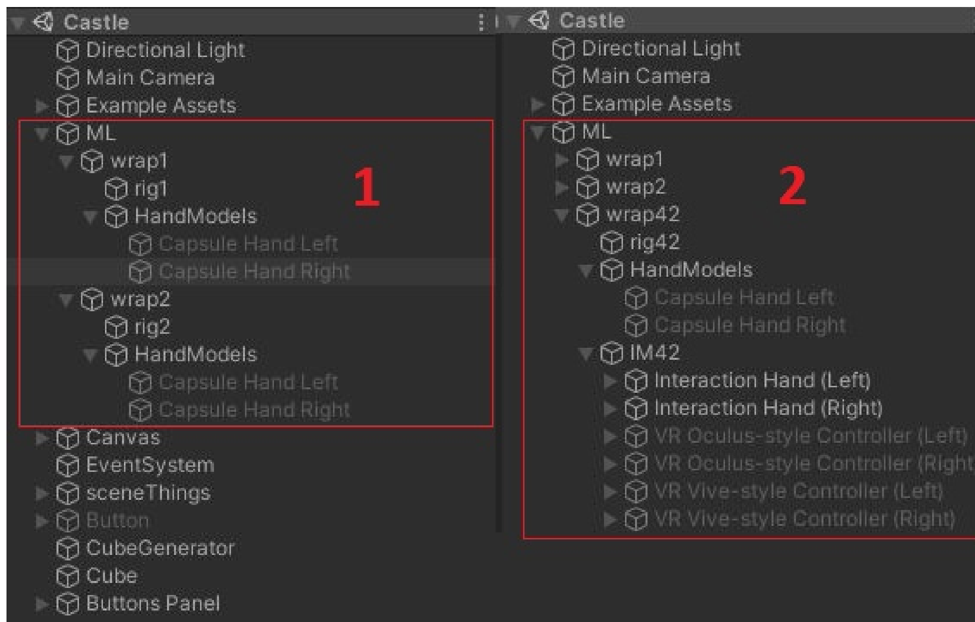

Figure 3.3: Project structure at runtime. (1) Before merging, (2) After merging

Figure 3.4: Simple UI

3.1.4 Results

The resulting solution can be obtained as Unity package, with all required prefabs. Unity package can be easily created through the editor GUI, where the user can select which parts of the project are to be included. Inside the package, ML prefab contains complete setup, which can be added to scene for the tracking to work. Various other prefabs used in my testing scenes are also provided.

Chapter 4

Testing

In this chapter, I will describe testing scenes I created, sensor setup, testing scenarios and lastly result obtained from users.

4.1 Testing scenes

I prepared two testing scenes for Unity, Castle and Sandbox. Scenes were created in Unity editor using primitive objects like spheres or cubes and custom made elements like interactable buttons.

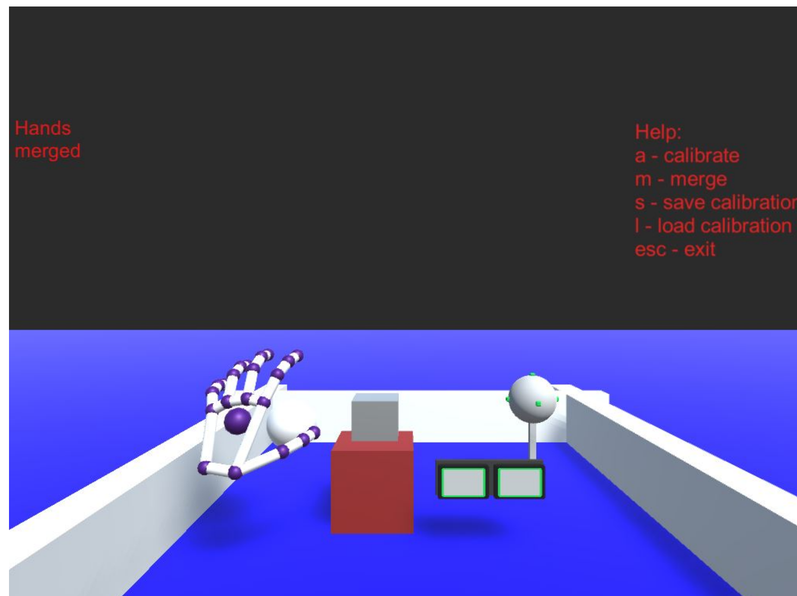

Figure 4.1: Sandbox

First scene Sandbox contained several elements for the user to interact with, like lever and a button producing sound when pressed.

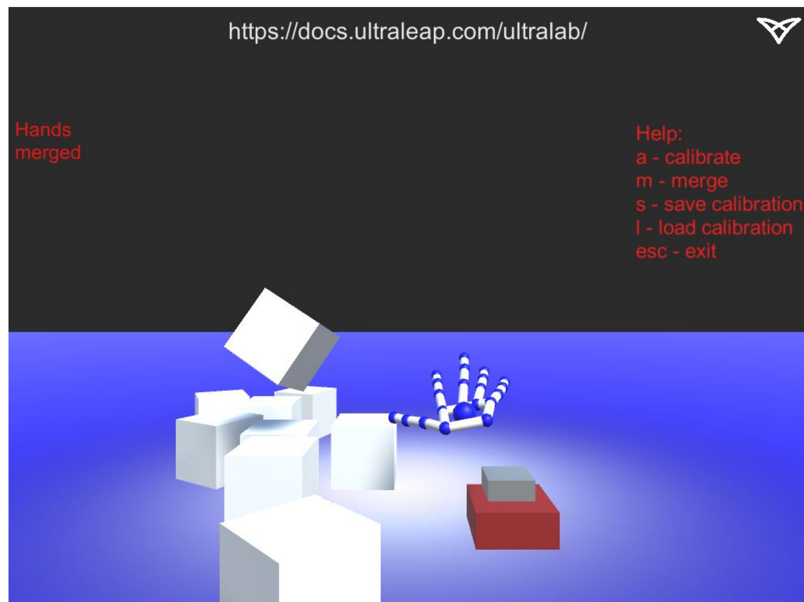

Figure 4.2: Castle

The second scene Castle comprised of button generating cubes when pressed. In addition to creating cubes in the scene, InteractionBehaviour is also added to each new cube, making it interactable.

```
public void createCube()
{
    GameObject go = GameObject.CreatePrimitive(PrimitiveType.Cube);
    go.AddComponent<MeshFilter>();
    go.AddComponent<InteractionBehaviour>();
    go.AddComponent<MeshRenderer>();
    go.transform.position = new Vector3(-0.2f, 0, 0);
    go.transform.localScale = new Vector3(0.1f, 0.1f, 0.1f);
    cubes.Add(go);
    cubesNumber++;
}
```

Code 4.1: Cube generation

4.2 Sensor setup

In my testing, I used three sensors fixed in place, with calibration done beforehand. Testing was done in a darkened room, to reduce IR interference from ambient light. Setup layout was similar to the one used by my supervisor, to obtain comparable results under near optimal conditions. Video projector was used to provide visual feedback.

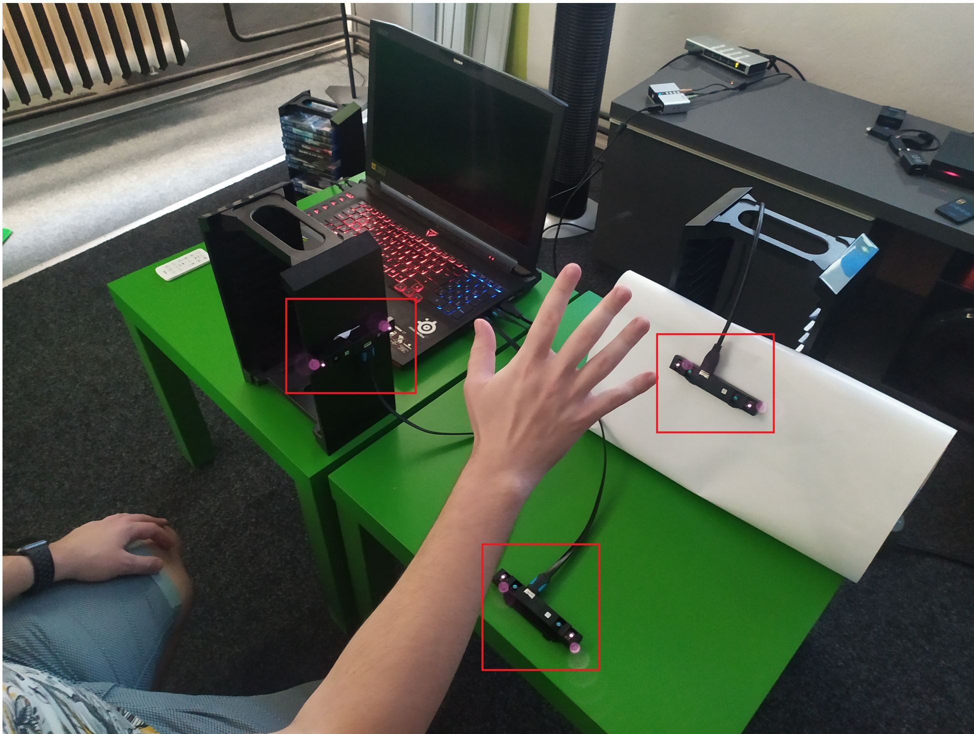

Figure 4.3: Sensor setup

4.3 Testing scenarios

I tested each scene with users twice, first time using only one sensor without merging, and the second time using complete setup with merged hands.

Aim of the first scene was for the users to try basic interaction with elements present, and then give feedback about intuitiveness of the controls and if they noticed any difference between using one and multiple sensors.

The objective of the second scene was to stack three cubes on top of each other, using the button to generate more cubes if necessary. Again, feedback was collected about intuitiveness, precision, and if the task was easier with one sensor or multiple.

Figure 4.4: Sensor setup with visible projection

Figure 4.5: User interacting with elements in Sandbox scene

4.4 Results

Testing was done with four people, each trying every scenario, first Sandbox, then Castle, both with limited and complete setup.

In the first scene, all users reported increased accuracy and easier interaction with multiple sensors. They also noted that larger tracking space,

provided thanks to multiple sensors tracking more real space, helped reach areas of the scene not accessible when using single one. They also noted decreased occurrence of the virtual hand vanishing. This phenomena happens when no sensor is able to track the hand, either to it being located outside of tracked space, or when it is not able to recognize the hand in the picture, possibly due to occlusion.

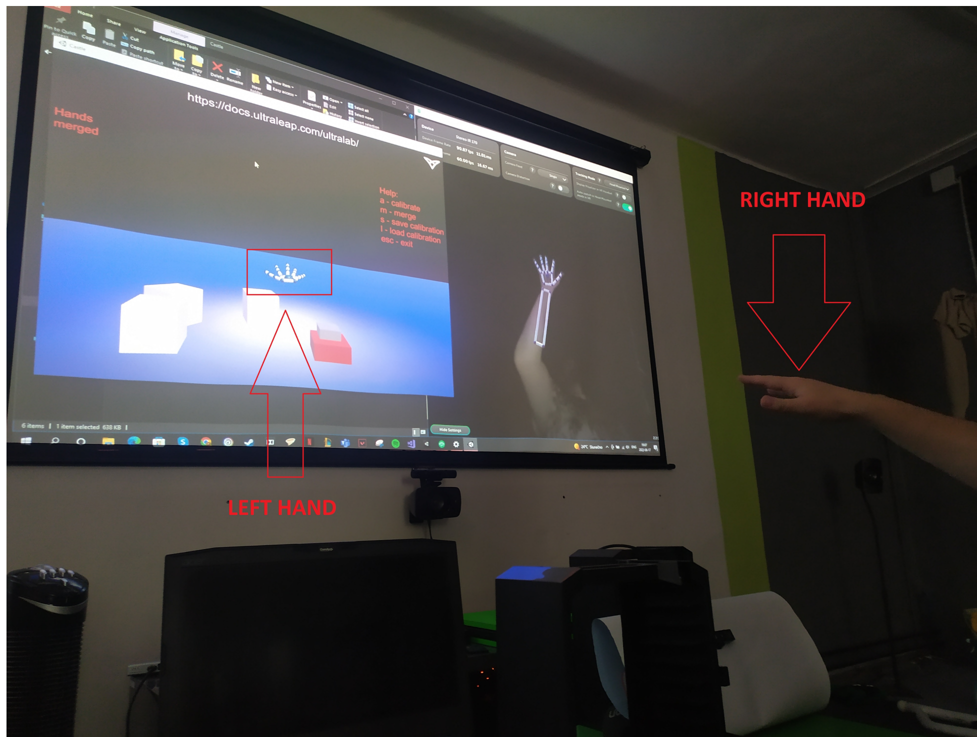

Figure 4.6: Hand with incorrect handedness

In the second scene, only two of the users were able to complete the given task when using single sensor, and again when using all three. Benefits of greater tracking area and increased precision when using full setup were diminished by increased occurrences of sensors incorrectly categorizing tracked hand as left or right.

Overall, all users reported scenes being intuitive for use, needing only minimal explanation of the scenarios. Also, perceived precision was higher when using full setup, allowing for more fluid movement.

4.5 Problems

During testing, several problems were observed. The most present problem was sensors assigning chirality[4] to hands incorrectly. This had the effect of

[4]Handedness, hand can be either left or right

the hand being flipped in the application, with all gestures being mirrored, increasing the difficulty of interaction. Due to implementation of physics by Ultraleap, if the hand interacting with an object stops being tracked, for example when grasping a cube suspended in air, the affected object is frozen in place, being activated again when acted upon by a hand. In some cases, MLL was able to correct the handedness of the hand. This resulted either in both correct and mirrored image of the hand being present at the same time, or by switching to the correct chirality. First outcome affected negatively mainly visual presentation, while the second resulted in sudden hand movement, often losing grasp on interacted object and in extreme cases launching several objects outside of the scene space at high velocities, due to internal collisions.

These problems were more pronounced with users with smaller hands. Due to the nature of the problems present, I was able to deduce the cause lies within sensors and software from the provider and not my implementation.

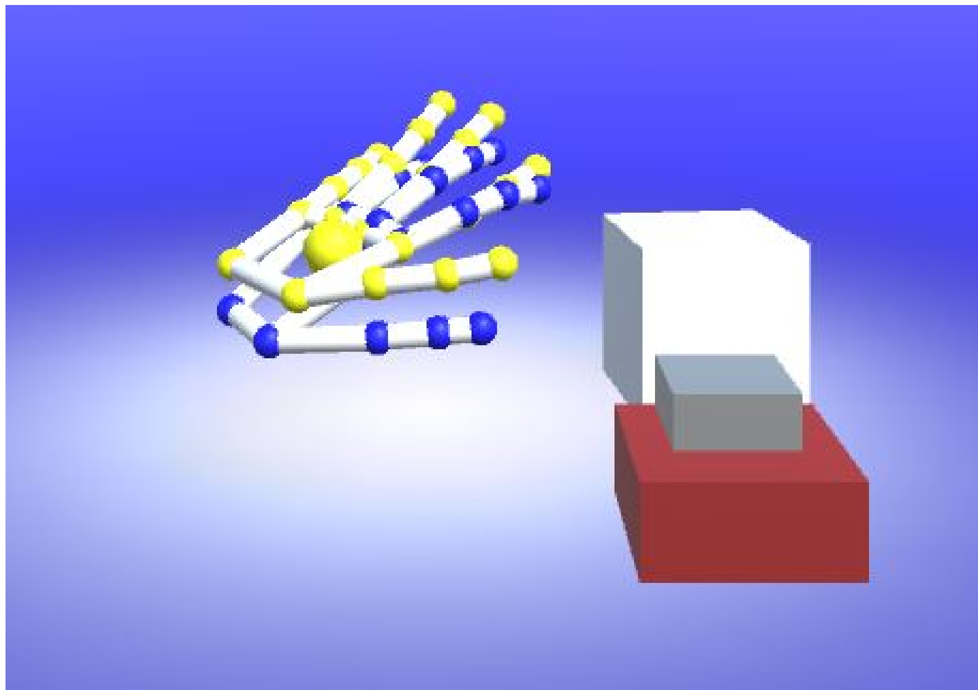

Figure 4.7: Incorrectly evaluated left hand with flipped right hand superimposed on top

Chapter 5

Future work

During development and subsequent testing, several additional requirements were identified for the package to be easily usable.

One of the main requirements is the ability to handle hot-loading of sensors. This means that the number of active sensors can be changed at runtime, either using more or fewer as needed, depending on the requirements.

Another is expanded interactable UI, allowing the user to easily configure solutions parameters, and facilitate the usage of all possible merge or calibration modes.

I intend to modify the InteractionManager component, so that physics will be enabled also on unmerged hands and interaction on objects, that can be set up during scene creation and not at runtime.

From the testing phase I identified a number of problems that can be addressed, such as hands being identified with wrong chirality, right hands as left hands, left hands being detected as right. More investigation is needed to determine the reason behind this behaviour, but I suspect the algorithms used in LCL are at fault. This problem is already being answered by my supervisor by additional algorithms for correcting the chirality directly in the MLL.

Another possible update is the identification of the gestures made by hands. An approach using machine learning algorithms is being explored in another thesis already. For example, we can imagine shooter game such as Doom to be mapped into the XR state of the game. We can imagine a game fully embedded into XR in which our movement will be tracked and mapped to every possible output. Maybe we will be shooting, or manipulating the game objects with our hands, holding a lamp or a flashlight with one of our hands. In the meantime, for example, the second hand will be holding a gun, or a chainsaw.

With this, the possible usage of this work is limited only by our imagination. The limits are set by the detection of our hands by the sensors. With better sensors, countless possibilities arise. Maybe the mapping of user interface to gestures, such as opening hands to open inventory, moving items from

shop to our gear and many more can be investigated.

This work can lead to number of possibilities, such as games based on complete capture of user's hands and their movement in real time. This can be used in most of current VR games, such as Superhot VR.

Furthermore, visual bounds of the space tracked during calibration can be beneficial to the calibration process by providing user with more visual feedback, thus increasing precision.

Conclusion

The purpose of this thesis was to analyze currently available software and hardware by Ultraleap used for hand tracking, analyze the capabilities of Unity Game Engine in creating virtual worlds with usage of aforementioned technologies, integrate MultiLeap library into Unity and lastly create demo environments for testing purposes.

I found the sensors provided by Ultraleap adequate for the proposed work, along with supplied software. With their Unity plugin, I was able to run their demo showcases with one sensor on an agreeable level. With these tools, user is capable of creating their own virtual environments quite easily. I was able to integrate MultiLeap library with Ultraleap plugin, making some changes to the original code. This somewhat prevents straightforward usage with future version, but this error will be amended by future work on the plugin. Lastly, I created two simple testing scenes and tested various setups with users.

In conclusion, I believe that Unity serves as an optimal engine for creating virtual worlds, resources that I worked with can be integrated together even better with sufficient research, and that users find interaction using touchless sensors very intuitive and entertaining, with better results obtained when using MultiLeap library.

Bibliography

[1] Ultraeap. Stereo IR 170 camera module evaluation kit - ultraleap. Nov 2020. Available from: https://www.ultraleap.com/datasheets/Stereo_IR_170_datasheet.pdf

[2] Unity Technologies. Order of execution for event functions. Jun 2022. Available from: https://docs.unity3d.com/Manual/ExecutionOrder.html

[3] Schnipper, M.; by Adi Robertson, C.; et al. The rise and fall and rise of virtual reality. Available from: https://www.theverge.com/a/virtual-reality/intro

[4] Hayes, A. Augmented reality definition. Sep 2021. Available from: https://www.investopedia.com/terms/a/augmented-reality.asp

[5] Blenkinsopp, R. What is haptics? definition of haptic feedback and Technology. Jun 2019. Available from: https://www.ultraleap.com/company/news/blog/what-is-haptics/

[6] Nováček, T.; Jiřina, M. Project MultiLeap: Making Multiple Hand Tracking Sensors to Act Like One. In *2021 IEEE International Conference on Artificial Intelligence and Virtual Reality (AIVR)*, IEEE, 2021, pp. 77–83.

[7] Leap Motion. Tracking software windows 5.3.0. Mar 2022. Available from: https://developer.leapmotion.com/releases/windows-gemini-5-3?fbclid=IwAR2c6FYX-1lpwKKdHGSC4OufBqwMLNsk9GSgpM_kZGMh3MfAvk82rebQkqw

[8] Ultraleap. LeapC API. 2021. Available from: https://docs.ultraleap.com/tracking-api/

[9] Leap Motion. Experimental Release #2: Multiple Device Support. Dec 2018. Available from: https://blog.leapmotion.com/

multiple-devices/?fbclid=IwAR2GKKJ12E4h8hEIdUL_Z-KMEeR9xgiHV7Woi5cqcfHyDkmaSP7m2pogYBc

[10] Kabsch, W. A solution for the best rotation to relate two sets of vectors. *Acta Crystallographica Section A*, volume 32, no. 5, 1976: p. 922–923, doi:10.1107/s0567739476001873.

[11] Free Code camp. Unity Game Engine Guide: How To Get Started with the most popular game engine out there. Apr 2021. Available from: https://www.freecodecamp.org/news/unity-game-engine-guide-how-to-get-started-with-the-most-popular-game-engine-out-there/

[12] Computer Hope. What is a plugin? Jun 2021. Available from: https://www.computerhope.com/jargon/p/plugin.htm

Appendix A

Acronyms

AR Augmented reality.

DLL Dynamic Link Libraries.

IR infrared.

LCL LeapC library.

LSP LeapServiceProvider.

MLL MultiLeap library.

MR Mixed reality.

UI User Interface.

VR Virtual reality.

XR Extended reality.

Appendix B

Contents of enclosed SD card

```
README.MD ..................... the file with SD card contents description
├── Ultraleap ................................ folder for Ultraleap scripts
│   ├── CircularObjectBuffer.cs ........................................
│   ├── Connection.cs ..................................................
│   ├── Controller.cs ..................................................
│   └── LeapServiceProvider.cs .........................................
├── Scenes ....................................... executable test scenes
│   ├── Castle .........................................................
│   └── Sandbox ........................................................
├── Control.cs ........................................ main control class
├── CubeGenerator.cs .............. simple class generating cubes on event
├── MultiLeap.unitypackage ................. MultiLeap package for Unity
└── Thesis ................................... LaTeX codes of the thesis
    └── thesis.pdf ................................................ thesis
```