DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics: Games Engineering

Smartphone-assisted Virtual Reality using Ubi-Interact

Michael Lohr

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics: Games Engineering

Smartphone-assisted Virtual Reality using Ubi-Interact

Smartphone-gestützte Virtuelle Realität mit Ubi-Interact

Author:	Michael Lohr
Supervisor:	Prof. Gudrun Johanna Klinker, Ph.D.
Advisor:	Sandro Weber, M.Sc.
Submission Date:	October 15, 2019

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, October 15, 2019 Michael Lohr

Acknowledgments

Abstract

Contents

Abbreviations

API Application Programming Interface

Protobuf Google Protocol Buffers

IMU Inertial Measurement Unit

JS JavaScript

UBII UBI-Interact

UID Unique Identifier

VR Virtual Reality

2D Two-dimensional

3D Three-dimensional

1 Introduction

Hi, this is my thesis, and I'm going to be the introduction.

2 Implementation

2.1 Ubi-Interact

UBI-Interact (UBII)[1] is a framework for distributed applications, which enables to connect all kinds of different devices together. A centralized server is used to manage the system in a local network. The abstraction into devices, topics and interactions allows to decouple the implementation of a software from device specific environments.

2.1.1 Architecture

The main components of the UBII framework are:

Clients describe a basic network participant. For every client registered in the system, also exists one network socket adress. Clients are an abstraction of a physical network device. They are defined by an Unique Identifier (UID).

Devices can be registered by clients. A UBII-device groups different input and output devices together. It is defined by a UID and a list of components.

Components contain the topic name, message formats for input/output devices and wether it publishes input or receives output data. A data source for such an input device, could be any sensor for example a button or a camera. Data output examples for input devices are lamps and displays.

Message Formats define the format of data published to a topic. Even though it is possible to implement custom ones, ost common data types are available. For example `Vector4×4` (a four by four matrix), `Vector2` (a two-dimensional vector) or `boolean` (a binary value) are built-in.

Topics are data channels which are addressed by a name. Clients can publish messages to topics, which are registered by a device. Clients are able to receive messages, after subscribing to a topic. Such messages (also called "topic data") are formatted

[1]UBII is currently developed and maintained by Sandro Weber, who is also the advisor for this thesis.

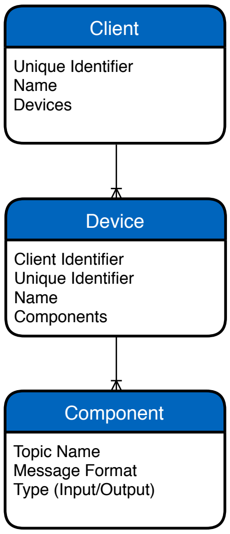
as JSON[1]-string, whose structure is defined by the device.

Sessions operate on the server, but can be specified by the client. They are defined by a UID as well as a list of interactions and **input/output mappings**. The mappings are defined by a message format and topic name.

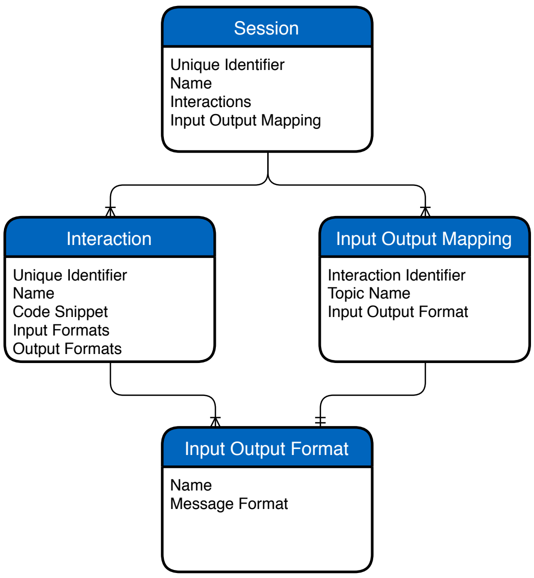
Interactions are reactive components. They operate on topics and are defined by a source code snippet[2]. Interactions are executed in a fixed interval on the UBII server. They can subscribe to topics and use the received topic data as input, given an input/output mapping description. The output of the interaction is published into another topic. It is also possible to keep data to use in future executions (persistent state).

Services are channels, used to send commands or requests to the UBII server. For example they are used to subscribe to a topic or list all available topics.

The Figure 2.1.1 visualizes the relationships of the different components.



(a) The client components.



(b) The session components.

Figure 2.1.1: Relationships of the core components in an entity relationship diagram.

[1]JSON is a standardized data exchange format, that uses human-readable text. It is often used for web-based data communication [ECM17, p. iii].

[2]Currently only JavaScript is supported as a script language.

2.1.2 Interactions

An powerful but optional core feature of UBII are interactions. As explained in the component overview (see 2.1.1), they are reactive components, which operate on topics and regularly execute given code snippets (processing functions) on the UBII server. Interactions are isolated components, which just depend on topic data and nothing else. This abstraction introduces the possibility to reuse logic in other applications in similar context. The data flow from a device to the interaction is visualized in the Figure 2.1.2.

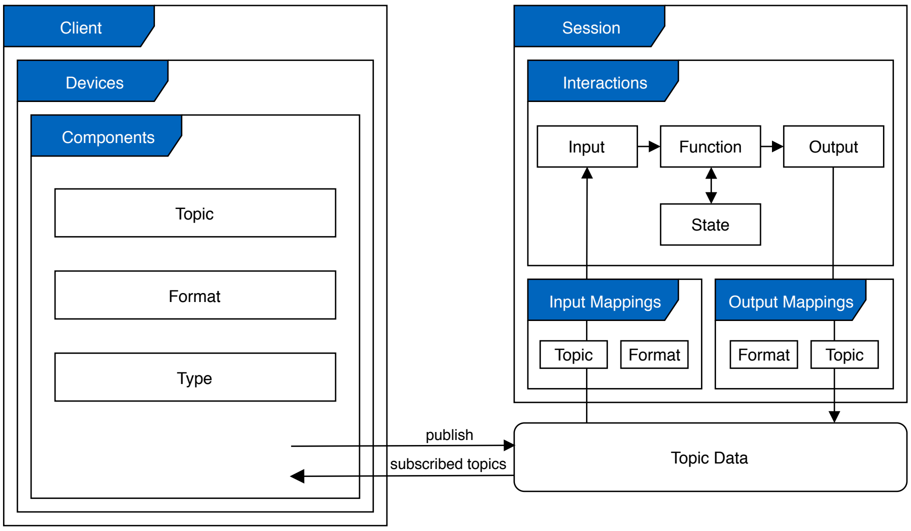


Figure 2.1.2: Interaction processing overview. This graphic gives a rough overview of the dataflow when using an interaction. Figure created with the help of Sandro Weber.

Interactions should be designed generalized, so that they are easy to reuse. They can be used to discretize data, converting data to other formats or just to outsource some logic from the application. Concrete examples include detecting button presses, transforming coordinates and evaluating data. An example implementation which detects position changes can be seen in Figure 2.1.3.

They are also useful, if two topics with different formats should be connected. An example for such a scenario could be an application, which consumes a rotation given in euler angles. But some input devices publish euler angles in degrees. An interaction, which takes euler angles in degrees from one topic and publishes euler angles in radians to another one could be implemented.

```
// detect intentional movement by comparing the current position with a previous one
function (inputs, outputs, state) {
  const threshold = 0.05;

  if (state.position) {
    const vector = {
      x: inputs.position.x - state.lastPosition.x,
      y: inputs.position.y - state.lastPosition.y,
    };

    const squaredDistance = Math.pow(vector.x, 2) + Math.pow(vector.y, 2);

    outputs.moved = squaredDistance < threshold;
  } else {
    outputs.moved = true;
  }

  state.lastPosition = inputs.position;
}
```

Figure 2.1.3: An example for an interaction written in JavaScript. This interaction calculates the squared distance of two points. One of the points is provided through the input, while the other one is stored in the state variable. To achieve this, the euclidean vector norm of the subtraction of both vectors without the square root is calculated and compared with a threshold constant. The result is then written into the output as a boolean data type. This is used to detect intended changes of the input position.

The code snippet has to define a function, which accepts three parameters: `inputs` is a collection of values, which contains values which were published into a topic. The topic which was used, is defined by the input mappings of the session. `outputs` is an empty collection, where values can be added. Those values are then published into a topic, defined by the output mappings of the session. `state` stores a persistent collection of values, which can be used in later executions of the same interaction.

2.2 Technology Stack

Since most of the existing software for UBII was written in JavaScript (JS)[1] using a web-based architecture, I decided to adapt this approach for my application. This has the major advantage of platform independence. Most modern devices can run web-based software, which means they can also run my application. Also the application is served by a web server, which means the user does not have to install the software onto his device.

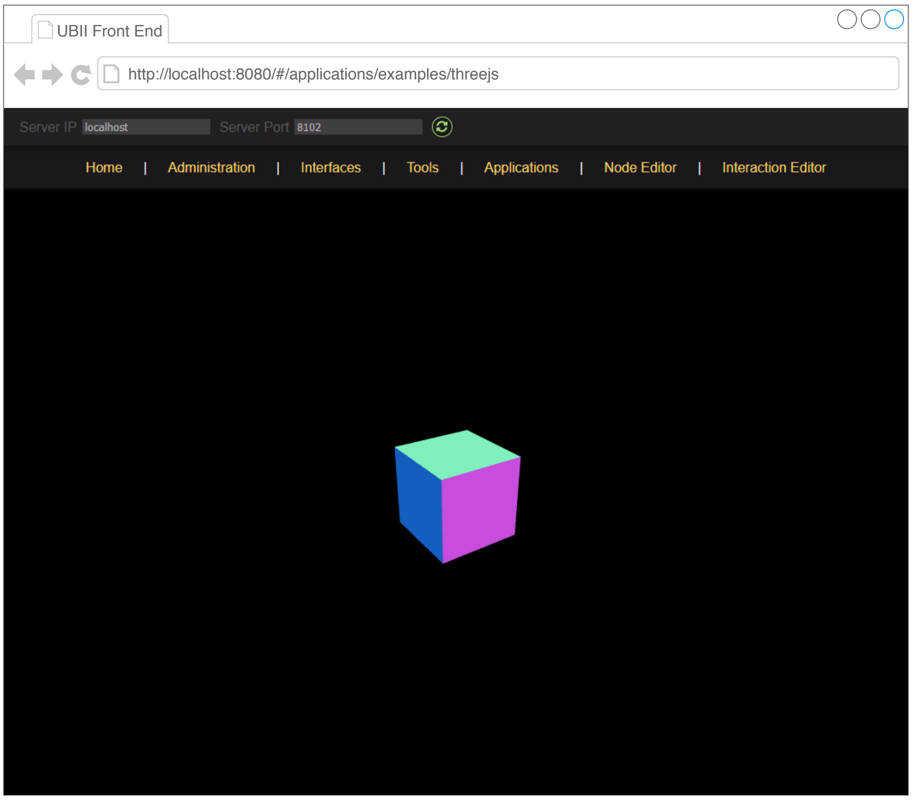


Figure 2.2.1: A screenshot of the UBII front end rendering a 3D cube.

A web interface with some UBII content (the UBII front end), demos and debugging tools was already written[2], so I included my experiments in this application as well. The technology stack of the front end was built with the following technologies:

[1]JS is a just-in-time compiled scripting language, widely used in web technology. It is a dynamic prototype-based language, which supports object-orientated programming [ECM18, pp. 43, 47].

[2]The front end was developed by Sandro Weber, Daniel Dyrda and me.

Web APIs are Application Programming Interfaces (APIs) available in modern web browsers to provide access to functionality or data outside the browser. The WebAPI is an additional layer of abstraction of the APIs of an operating system. While this has the advantage that the API is the same on every device, this also prevents the access to the raw sensor data[1]. In this thesis the WebVR API and the device orientation API were used. The former enables to render to external Virtual Reality (VR) headsets. The latter gives access to the data of the Inertial Measurement Unit (IMU)[2].

Vue.js [3] is a modern open source JavaScript web framework[4] [You19]. Being released in 2014 and developed by Evan You, it is a relatively young framework [Koe16, p. 17]. But it quickly gained traction and is quite popular now [Koe16, pp. 12 sq.]. Packages like Vue.js itself, Vue.js plugins and other JavaScript libraries are managed using the package manager npm [5].

Three.js [6] is a lightweight open source library which utilizes WebGL to render three-dimensional (3D) computer graphics [Cab19]. It can be used to render scenes to the display as well as to a VR headset using WebVR. This high-level library comes with a lot of features, similar to a game engine, like scenes, effects, lights, animation, geometrie and much more.

UBII Client is an JavaScript client for the UBII system. It abstracts the protocol and provides high-level functions to register devices as well as send and receive topic data. The UBII system uses Google Protocol Buffers (Protobuf)[7] to serialize the data.

Figure 2.2.1 displays an test view, which uses Vue.js to manage the views and Three.js to render a cube.

[1] The specification is available on www.w3c.github.io/deviceorientation

[2] An IMU is an electronic component which is part of most smartphones and allows to measure force, angular rate and magnetic field.

[3] Vue.js: Website: www.vuejs.org; Source code: www.github.com/vuejs/vue

[4] A web framework is a software framework which provides a standard way to build web applications. It comes with tools and libraries to automate and make the development of web applications easier.

[5] "NPM" stands for "Node Package Manager" and is also used in the UBII server itself. Website: www.npmjs.com

[6] Three.js: Website: www.threejs.org, Source code: www.github.com/mrdoob/three.js

[7] Protobuf is a mechanism to serialize data. The data is defined in a platform-neutral language, which compiles as library to all commonly used programming languages [Goo19b]. Website: www.developers.google.com/protocol-buffers/

2.3 Smart Device

The "Smart Device" is a part of the UBII front end. Because it is web-based, only data which is available through the **Web APIs** can be obtained. Since it was not designed for a specific use case, it is thought as general purpose or testing device. Only touch positions, touch events, orientation and acceleration is sent to different topics using the **UBII Client**. For more specific scenarios, the smart device can not be used and a custom interface has to be implemented. For the experiments in this thesis though, the smart device client was sufficient, after implementing some improvements.

The data which is published, is also displayed on the screen for debugging purposes. It is possible to set the view to full screen mode, to prevent unintentional interactions with control elements of the web browser or the operating system. Since the reference system for the orientation is fixed to the earth [Dev19, Chapter 4.1], a calibration system was implemented. With the press of the "Calibrate" button, the device is calibrated to the new orientation.

2.3.1 Topic Data

The orientation is provided by the **Web API** through the `DeviceOrientation` event. It is defined by three euler angles named `alpha`, `beta` and `gamma`, as seen in Figure 2.3.1. While `alpha` returns values in the range $[0, 360)$, `beta` only returns the range $[-180, 180)$ and `gamma` $[-90, 90)$ [Dev19, Chapter 4.1]. This limitation entails that no full orientation tracking is possible with this event.

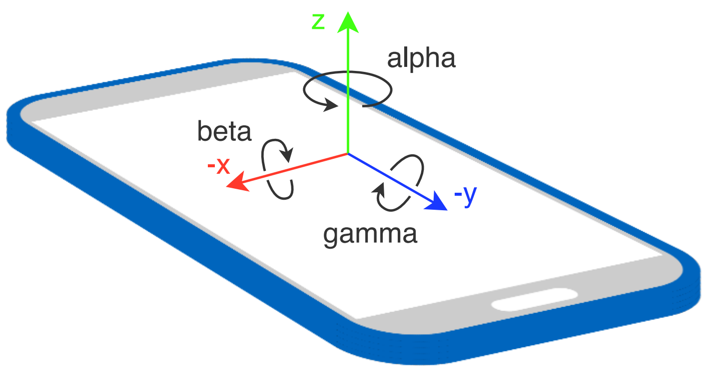


Figure 2.3.1: The specification of the orientation values visualized. The x and y axes are inverted for the sake of clarity in this graphic.

The **Web API** also provides the `MotionEvent`, which returns multiple vectors, one being the acceleration with the gravity (`accelerationIncludingGravity`). Since the gravity vector always points down, this vector can be used as a reference vector. Together with the values from the `DeviceOrientation` event, the full orientation can be derived. The resulting orientation, then has to be smoothed, because the acceleration vector uses the raw IMU acceleration output.

The data from the `DeviceOrientation` event already provides all three euler angles and is smoothed. Implementing the same for the data from the `MotionEvent`, would be outside the scope of this thesis. Because of this consideration, the `DeviceOrientation` event data is used in the experiments.

The touch position on the display, is normalized to a range from zero to one. This removes the influence of the display resolution and size. Events for start and stop touching are sent on different topics. The acceleration of the smartphone is also sent to a topic, but is not used in the experiments of this thesis.

2.3.2 UBII Device Definition

The smart device is registered as a device in the UBII network. The definition in JS can be seen in Figure 2.3.2. The structure of a UBII device was described in Chapter 2.1.1.

```
const ubiiDevice = {
  name: 'web-interface-smart-device',
  components: [
    {
      topic: clientId + '/web-interface-smart-device/touch_position',
      messageFormat: 'ubii.dataStructure.Vector2',
      ioType: ProtobufLibrary.ubii.devices.Component.IOType.INPUT
    },
    {
      topic: clientId + '/web-interface-smart-device/orientation',
      messageFormat: 'ubii.dataStructure.Vector3',
      ioType: ProtobufLibrary.ubii.devices.Component.IOType.INPUT
    },
    {
      topic: clientId + '/web-interface-smart-device/linear_acceleration',
      messageFormat: 'ubii.dataStructure.Vector3',
      ioType: ProtobufLibrary.ubii.devices.Component.IOType.INPUT
    },
    {
      topic: clientId + '/web-interface-smart-device/touch_events',
      messageFormat: 'ubii.dataStructure.TouchEvent',
      ioType: ProtobufLibrary.ubii.devices.Component.IOType.INPUT
    }
  ]
};
```

Figure 2.3.2: The smart device definition in JavaScript. It is defined by a name and a list of components. The structure of a UBII device is further described in Chapter 2.1.1.

A UBII device and all topics must be registered with an individual id for each client, because it should be possible to read the data from different devices. This allows for using multiple devices at the same time, so that they can be differentiated in interactions. If the topic names would not include the `clientId`, each connected device would publish to the same topic, which would make the data unusable.

At the time of creating the experiments presented later, the component for the touch events was not implemented yet. The touch position is published multiple times per second, but only sends the current position of the first touch on the smartphone display. Using this data, it is not possible to detect wether the display was just touched or released. A new topic using the `Boolean`-type could have been used, but the position has to be obtained from the touch position topic. To remove this dependency on the other topic, the new type `TouchEvent` was implemented. The Protobuf definition can be seen in Figure 2.3.3. It contains the two-dimensional position and the binary type `ButtonEventType`, which can be reused in other events, too. `ButtonEventType` is an enumerated type which defines wether the touch interface was just touched or released.

```
syntax = "proto3";
package ubii.dataStructure;

import "proto/topicData/topicDataRecord/dataStructure/vector2.proto";

enum ButtonEventType {
  UP = 0;
  DOWN = 1;
}

message TouchEvent {
  ButtonEventType type = 1;
  ubii.dataStructure.Vector2 position = 2;
}
```

Figure 2.3.3: The definition of the touch event, sent by the smart device client when a user touches or releases the touch screen. It is defined by a position and wether the touch pad was touched or released.

2.4 Experiments

Three main experiments were implemented to show how the smartphone can help with common interactions when using VR software. To achieve consistency amongst all experiments in terms of look and functionality, a parent class was implemented. The parent class implements multiple utilities and helpers, which are required by each experiment. It also sets up a basic scene, which contains a sky, a floor and lights. Also the connection to the UBII server is handled.

2.4.1 Model Viewer

Virtual Reality offers a new way of experiencing 3D content. It is more convenient to view a model from different angles and gives a feel of a real presence of the object. Model viewers like Sketchfab[1] have implemented VR support a while ago [Den16]. But

[1]Sketchfab is an online platform to publish and view 3D content. Website: www.sketchfab.com

this experience can be enhanced with a smartphone. Models can be rotated without changing the position of the headset.

Katzakis and Hori implemented this without VR. His approach uses a smartphone to rotate a model, which is displayed on a conventional display. He uses a similar setup, where the phone is wireless connected to a computer, where the model is rendered. The orientation data comes from the magnetometer and, once calibrated to the screen position, is directly mapped to the model [KH10, p. 139]. In the comparison against a mouse and a touch pen, the smartphone clearly wins in terms of the time it takes to rotate the model to a certain pose [KH10, p. 140]. Since this approach turned out to be very successful, it was used in this experiment as well.

To feature how easy it is to view a complex model using VR and the smartphone as a manipulator, a skeleton model is used. This experiment is the only one, supporting more than one smartphone client at the moment. For every client that connects, a new skeleton is created. The position is fixed and arranged around the position of the VR headset. The scene is shown in Figure 2.4.1.

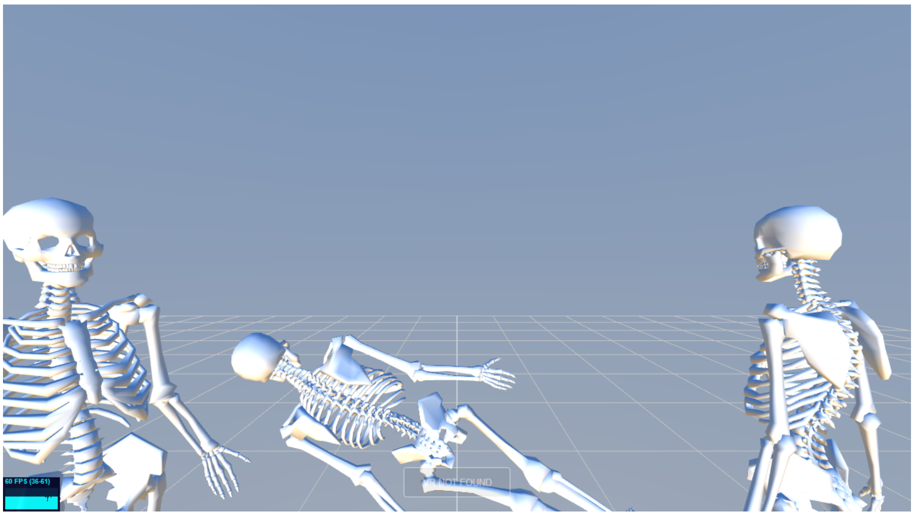


Figure 2.4.1: A screenshot of three devices being connected and controlling the rotation of the models.

The implementation of this experiment listens for new clients. As soon as one connects, a new interaction is published and the resulting topic is subscribed. Since the smart device publishes the orientation data in a different format than ThreeJS needs for rendering, a reusable interaction was created. The interaction converts the angles from radian to degrees, changes the coordinate system and publishes them to the `[client id]/SAVRLaserPointer/orientation`-topic. The code for the interaction is shown in

Figure 2.4.2.

```
function (input, output, state) {
  if (!input) {
    return;
  }

  const deg2Rad = function(v) {
    return v * Math.PI / 180;
  };

  output.orientation = {
    x: deg2Rad(input.orientation.y),
    y: deg2Rad(input.orientation.x),
    z: deg2Rad(-input.orientation.z)
  };
}
```

Figure 2.4.2: This interaction is used to convert the orientation data sent by the smart device to the format ThreeJS needs for rendering. The values are converted by multiplying with an approximate of the number Π ("PI") and dividing by 180.

2.4.2 Laser Pointer

Selecting elements in a virtual world is a basic interaction most VR applications use. The selection of elements in a two-dimensional (2D) environment with standard input devices like a mouse or touch screen is trivial. But the selection of elements in a 3D environment is problematic, because the element might be too far away from the the user or the cursor. Ray casting[1] is used to solve this problem: A ray, with the tracked device as origin, is created. Then, the element first hit by the ray is selected. Implementations without a tracked device, often use the position and orientation of the headset. The ray is fixed to the head of the user and casted along his viewing direction [Kam18, p. 23]. This forces the user to keep the head still and look at a certain object to select it, until a button is pressed or a certain time has passed.

A better solution is the use of handheld controllers, where the position of the controller is used as origin for the ray. Since the smartphone provides orientation data, it can be used for this task, too. But most handheld controllers have also positional tracking,

[1]Ray casting describes a technique to determine the objects which intersect with a ray, cast from a given point into a given direction.

which allows them to represent the hand of a user by displaying a virtual phone. This emulates the use of a laser pointer in the real world. Since a smartphone does not have positional tracking, the origin has to be somewhere else. Again, the head could be used, but then the user would have no visual representation of the rotation of the phone. To give the user a better feel for direction he is pointing, a visual representation is needed. The user has to see where the ray is going, even when rotating it in the opposite direction of the view direction.

Argelaguet and Andujar evaluated more than 30 different object selection techniques for virtual environments, but there are no technique that uses the orientation but not the position of the pointing device [AA13, Table 1]. To work around the missing position data of the device, the ray origin is set to a fixed location relative to the users head where the phone could be in the real world. The ray origin is represented by a 3D phone model, which orientation is synchronized with the one from the last connected smart device client, similar to the first experiment (2.4.1). To keep the virtual phone inside the view frustum of the user, it rotates relative to the user on the y-axis. A line is attached to the front of the phone (the "laser") to indicate the direction of the ray.

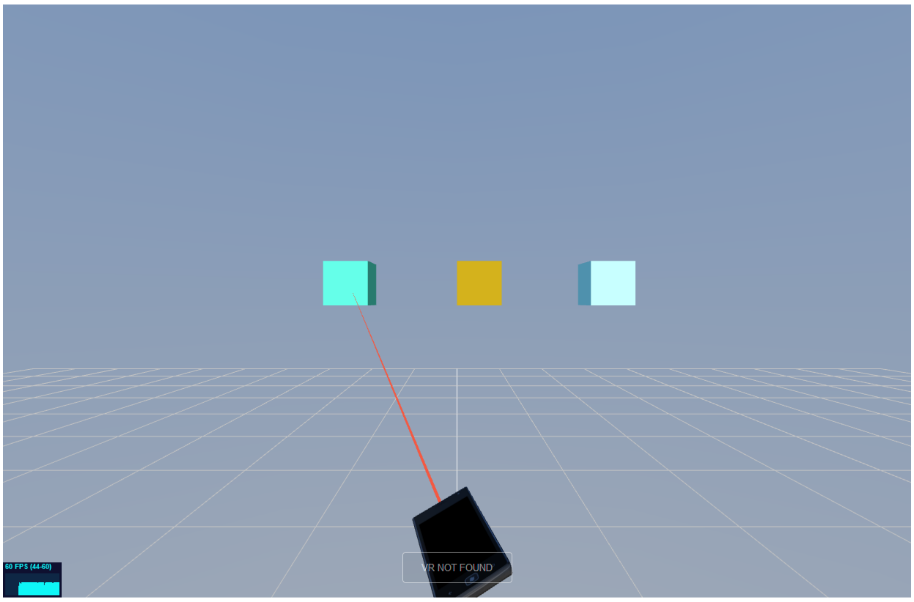


Figure 2.4.3: A screenshot of the virtual laser pointer and selectable cubes.

In addition to the orientation topic, this implementation subscribes to the touch events topic. The button down event is needed to trigger the actual selection. In this experiment cubes float in front of the user. If he points the laser at one and touches the display, the cube will change it's color. This works not only with cubes, but with any mesh. Also the system can trigger any kind of event or action. A screenshot of this setup can be seen in Figure 2.4.3.

2.4.3 Virtual Keyboard

Text input is not an easy task to perform in VR. But it is often required for labeling, annotation, entering filenames for saving operations and setting parameters in visualizations [RBP02, p. 2154]. This is why a lot of applications try to avoid it. Tilt Brush[1] solves this by identifying their scenes with a screenshot of the scene rather than a filename. To save a scene, the users gets a virtual camera attached to his hand motion controller, which he then uses to make a thumbnail of the current scene [Goo19a]. Others use a laser pointer either attached to a hand controller or to the headset to select keys on a 2D image of a keyboard [Wee17]. Another more recent approach is the frequently called “drum keyboard”, which attaches drum sticks to the hand controllers which are then used to hit the keys [Wei17]. There are also approaches which do not use hand controllers, like hand gloves [ESV99; RBP02], a real keyboard [McG+15; Wal+17] or other peripherals [Gon+09, pp. 111, 112]. Other experimental methods are speech recognition [RBP02, pp. 2154, 2156] or handwritten character recognition [Gon+09, p. 113].

Similar to the approach by Dias, Afonso, Eliseu, and Santos to implement user interfaces using a real smartphone and a virtual representation visible in the VR headset [Dia+18, p. 5], this experiment displays a virtual keyboard in VR as seen in Figure 2.4.4.

[1] Tilt Brush by Google is a tool for three-dimensional painting in VR. Website: www.tiltbrush.com

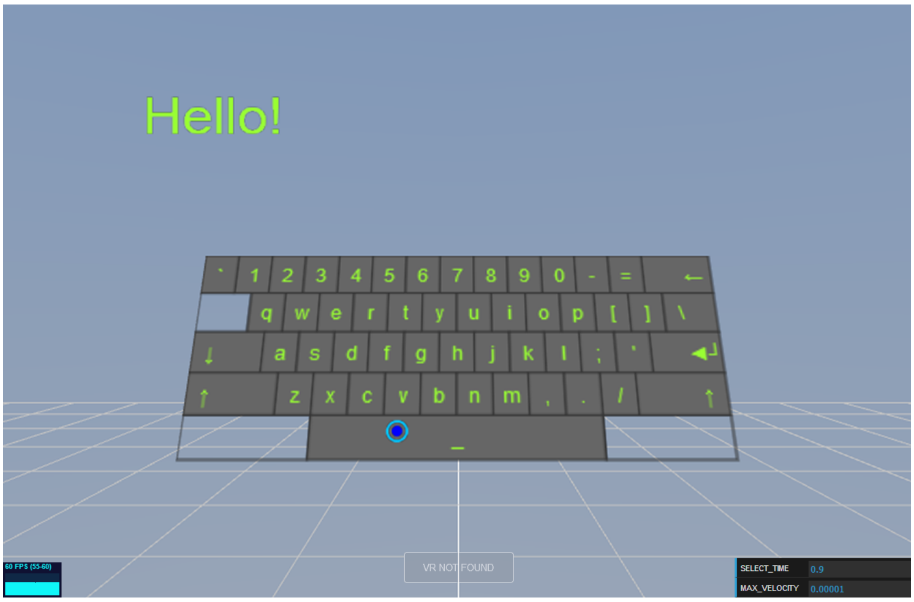


Figure 2.4.4: A screenshot of the virtual keyboard with the blue cursor and the previously typed text.

List of Figures

List of Tables

Bibliography

[AA13] F. Argelaguet and C. Andujar. "A survey of 3D object selection techniques for virtual environments." In: *Computers & Graphics* 37.3 (2013), pp. 121–136. ISSN: 00978493. DOI: `10.1016/j.cag.2012.12.003`.

[Cab19] R. Cabello. *Three.js: JavaScript 3D library*. 2019. URL: `https://github.com/mrdoob/three.js/` (visited on 06/17/2019).

[Den16] A. Denoyel. *Virtual Reality evolved: Sketchfab VR apps and WebVR support: New on Sketchfab - 16 May 2016*. 2016. URL: `https://sketchfab.com/blogs/community/announcing-sketchfab-vr-apps-webvr-support/` (visited on 06/23/2019).

[Dev19] Devices and Sensors Working Group. *DeviceOrientation Event Specification: Editor's Draft, 15 April 2019*. Ed. by Devices and Sensors Working Group. 2019. URL: `https://w3c.github.io/deviceorientation/#dom-deviceorientationevent-alpha` (visited on 06/17/2019).

[Dia+18] P. Dias, L. Afonso, S. Eliseu, and B. S. Santos. "Mobile devices for interaction in immersive virtual environments." In: *AVI '18: Proceedings of the 2018 International Conference on Advanced Visual Interfaces*. Ed. by T. Catarci, F. Leotta, A. Marrella, and M. Mecella. New York, NY, USA: ACM, 2018. ISBN: 978-1-4503-5616-9. DOI: `10.1145/3206505.3206526`.

[ECM17] ECMA International. *Standard ECMA-404: The JSON Data Interchange Syntax*. 2nd ed. 2017.

[ECM18] ECMA International. *Standard ECMA-262: ECMAScript 2018 Language Specification*. 9th ed. 2018.

[ESV99] F. Evans, S. Skiena, and A. Varshney. "VType: Entering Text in a Virtual World." In: (1999).

[Gon+09] G. González, J. P. Molina, A. S. García, D. Martínez, and P. González. "Evaluation of Text Input Techniques in Immersive Virtual Environments." In: *New Trends on Human-Computer Interaction*. Ed. by P. M. Latorre, A. Granollers Saltiveri, and J. A. Macías. London: Springer-Verlag London, 2009, pp. 109–118. ISBN: 978-1-84882-351-8. DOI: 10.1007/978-1-84882-352-5_11.

[Goo19a] Google LLC. *Protocol Buffers*. 2019. URL: https://developers.google.com/protocol-buffers/ (visited on 06/19/2019).

[Goo19b] Google LLC. *Tilt Brush Help: Saving and sharing your Tilt Brush sketches*. 2019. URL: https://support.google.com/tiltbrush/answer/6389651?hl=en (visited on 06/26/2019).

[Kam18] C. Kamm. "Precision of Pointing with Myo: A Comparison of Controller- and Gesture-Based Selection in Virtual Reality." Master's Thesis. Munich: Technische Universität München, 2018.

[KH10] N. Katzakis and M. Hori. "Mobile devices as multi-DOF controllers." In: *IEEE Symposium on 3D User Interfaces (3DUI), 2010 ; Waltham, Massachusetts, USA, 20 - 21 March 2010*. Ed. by M. Hachet. Piscataway, NJ: IEEE, 2010, pp. 139–140. ISBN: 978-1-4244-6846-1. DOI: 10.1109/3DUI.2010.5444700.

[Koe16] J. Koetsier. "Evaluation of JavaScript frame-works for the development of a web-based user interface for Vampires." PhD thesis. 2016.

[McG+15] M. McGill, D. Boland, R. Murray-Smith, and S. Brewster. "A Dose of Reality: Overcoming Usability Challenges in VR Head-Mounted Displays." In: *CHI 2015 crossings*. Ed. by J. Kim. New York, NY: ACM, 2015, pp. 2143–2152. ISBN: 9781450331456. DOI: 10.1145/2702123.2702382.

[RBP02] C. J. Rhoton, D. A. Bowman, and M. S. Pinho. "Text Input Techniques for Immersive Virtual Environments: an Empirical Comparison." In: *Proceedings of the Human Factors and Ergonomics Society: 46th Annual Meeting, Baltimore, Maryland, September 30 - October 4, 2002 : Bridging Fundamentals & New Opportunities*. Ed. by Human Factors and Ergonomics Society. Annual meeting. Santa Monica, Calif.: SAGE Publications, 2002, pp. 2154–2158.

[Wal+17] J. Walker, B. Li, K. Vertanen, and S. Kuhl. "Efficient Typing on a Visually Occluded Physical Keyboard." In: *Explore, innovate, inspire*. Ed. by G. Mark, S. Fussell, C. Lampe, m. schraefel m.c, J. P. Hourcade, C. Appert, and D. Wigdor. New York, NY: Association for Computing Machinery Inc. (ACM), 2017, pp. 5457–5461. ISBN: 9781450346559. DOI: 10.1145/3025453.3025783.

[Wee17] Weelco Inc. *Unity Asset Store: Keyboard VR Pro*. Ed. by Weelco Inc. 2017. URL: https://assetstore.unity.com/packages/tools/input-management/keyboard-vr-pro-83708 (visited on 06/26/2019).

[Wei17] M. Weisel. *An open-source keyboard to make your own – Normal*. 2017. URL: `http://www.normalvr.com/blog/an-open-source-keyboard-to-make-your-own/` (visited on 06/26/2019).

[You19] E. You. *Vue.js*. 2019. URL: `https://vuejs.org/` (visited on 06/17/2019).