Tactile Feedback in Virtual Realities

Technical Details

Johannes Lohmann {johannes.lohmann@uni-tuebingen.de}

July 24, 2017

Chair of Cognitive Modeling, Sand 14, 72076 Tübingen

Contents

1 Overview **1**

2 Hardware Implementation **2**

3 Software Interface **5**

3.1 Controller Program 5

3.2 Unity Interface 8

1 Overview

This document gives an overview how to realize a low-cost tactile stimulation device that can be applied to augment virtual object interactions with vibrotactile feedback. The device can be used in many different experimental setups of course, however, the use-case described here will focus on the application in a virtual reality (VR) setup. The first version of the device was realized by Jakob Gütschow during his master thesis. In his thesis, Jakob investigated whether multisensory integration is modulated by task demands and how multisensory information is maintained in working memory. In order to so, he applied a multisensory-conflict paradigm, where different sensory estimates regarding a single object did not match. To realize this mismatch it was necessary to provide haptic as well as visual information regarding the size of an object. The main purpose of the tactile stimulation device was to generate this kind of haptic information. In other studies we checked whether tactile feedback can be used to augment natural object interaction in VR. The following two sections provide a description of the hardware implementation and the software interface. The device is rather cheap, the core components can be obtained for less than 50€. Since the motors can be freely arranged, the device can be used in different setups, however, compared to commercial solutions (like for instance Gloveone), the device is rather inconvenient and preparing a participant takes considerably time.

In the next section, the hardware implementation is described. This is followed by a step-by-step description of the controller software and the integration of the device in a Unity® application. Please note that the final example requires a Leap motion controller. All code examples can be obtained from here.

2 Hardware Implementation

The circuit design is based on the project presented here. LearningaboutElectronics also hosts a youtube channel, with a lot of tutorials, the circuit described here is unfortunately not featured yet.

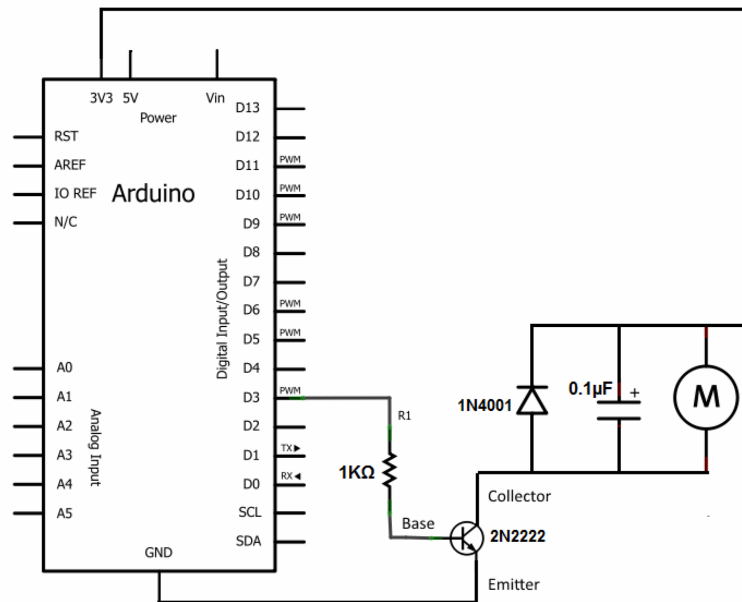

Figure 1: The wiring diagram for the protective circuit. It was obtained from LearningaboutElectronics. The circuit has to be realized for all motors that you want to attach to the microcontroller.

In a nutshell, the device consists of the following parts:

- an Arduino Uno microcontroller with a USB connector
- five shaftless vibration motors
- five 1N4001 diodes
- five 0.1 μF ceramic capacitors
- five 1KΩ resistors
- five 2N2222 NPN transistors

- one 10-pin connector (we used a shrouded 2×5 header)
- wiring, jumper cables and a breadboard

These components are rather cheap, the most expensive part is the microcontroller (about 24€). Depending on your vendor, the whole device costs between 40€to 50€. If you follow the approach proposed here, there are very few cases where you have to solder connections. However, there are at least 30 solder joints: The ten connections from the breadboard to the connector and four per motor. Setting up the whole circuit takes between two and three hours.

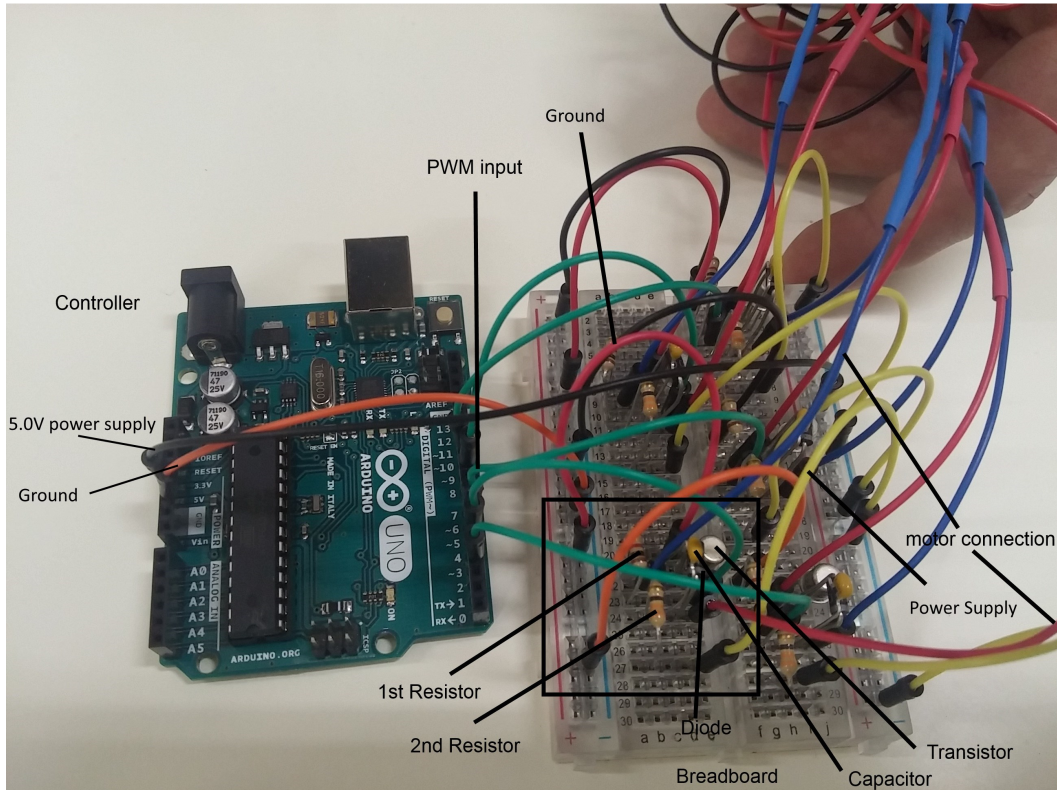

Figure 2: The five motor circuits arranged on a single breadboard. On the left side, you can see the microcontroller. Power supply is realized via the black cable, grounding is realized by the orange cable. Each of the motor circuits received input from a PWM connector (green cables). One circuit is marked with a black square and the components are annotated. The long yellow cable provides the power supply, while the short red cable serves as grounding. The blue and the red cable are the connections to the motor. The diode is hard to see but it is directly behind the capacitor (cf. Fig. 1). The whole arrangement can be set up without soldering connections.

The device is a combination of a microcontroller and five motors, which can for in-

stance placed under the fingertips of a participant. Here, we applied an Arduino Uno microcontroller (Arduino S.R.L., Scarmagno, Italy). Arduino boards were specifically designed to provide an easily accessible basis for programmable electronics. Both software and hardware are open source. For the purpose of the tactile stimulator, the most basic Arduino Uno board is sufficient. This board consists of a programmable microcontroller and a set of input and output connectors. Here, the most relevant outputs are the digital pulse-width modulation (PWM) connectors which convert 8-bit [0 - 255] inputs to simulated analog output for connected devices. This allows a convenient and continuous control over the vibration strength of the attached motors. The controller can be programmed with a simplified form of C++. Via a USB-port – which can also serve as power supply – a program can be stored in the internal 32KB flash memory. Once stored in the memory, the code runs continuously whenever the board is powered. The combination of a suitable program and the PWM outputs, allow precise control of the connected vibration motors.

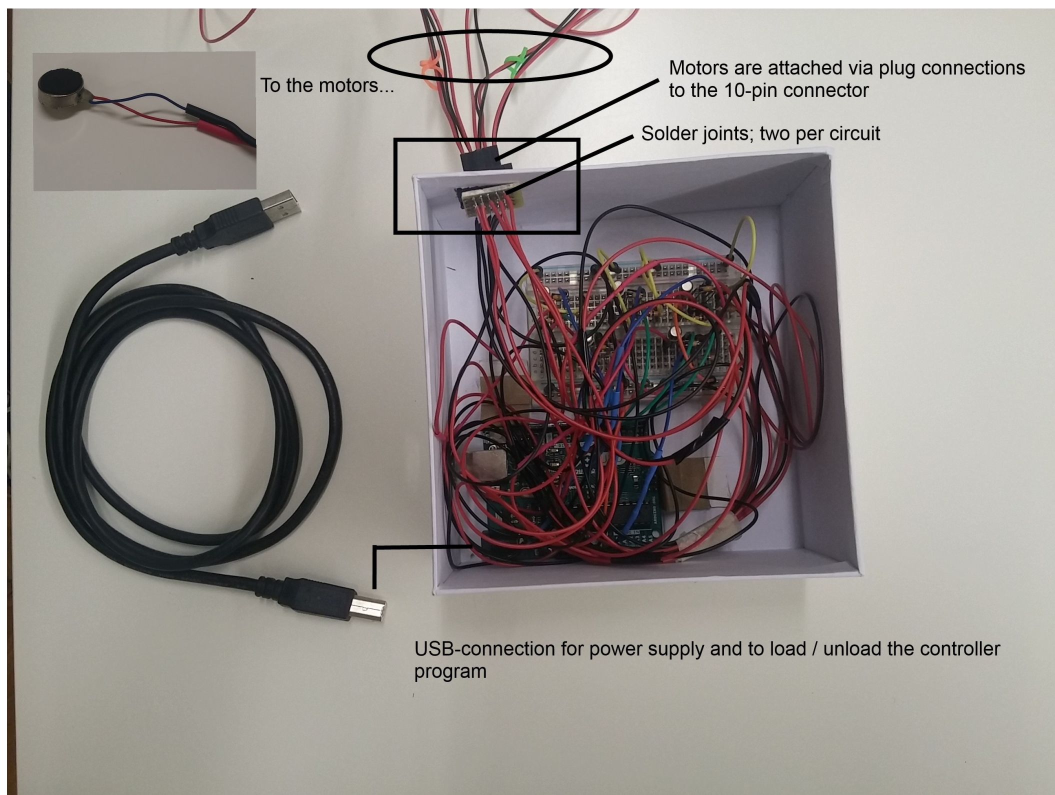

Figure 3: The two motor connections per circuit are attached to the 10-pin connector (marked with a square). These connections might be soldered, you can also use plug connections. The motors are connected to the 10-pin connector with plug connections. In the upper left, one of the motors is shown, which requires soldering connections as well.

We used shaftless vibration motors [1] with a diameter of 10 mm and a height of 3.4 mm in this project. Essentially, these motors are metal cases containing an eccentric disc, rotated by electric current, resulting in the vibration. At a current of 3.0 V, the motors produce a vibration with 200 rotations per second, the resulting vibration amplitude is about 0.75 g, which is well noticeable. While it is possible to directly attach the motors to the microcontroller, additional components are necessary to realize a stable setup (the wiring diagram is shown in Fig. 1). First, a diode have to be connected with the motor in parallel. The diode acts as a surge protector against voltage spikes which could damage the microcontroller. Second, a capacitor is connected in parallel to the motor, which absorbs voltage spikes produced by he motor. Third, a transistor is added to amplify the, comparatively weak, current output provided by the microcontroller. Fourth, a resistor is added to make sure that the applied current cannot damage the motor. The motor and all the protective elements connected in parallel, have to be connected to the collector of the transistor. The base of the transistor receives the PWM input, while the emitter is connected to the grounding (see Fig. 1).

It is possible to realize the whole circuit on a breadboard, thereby separating the protective circuit from the actual motors, which comes in handy if, for instance, one motor has to replaced. The breadboard wiring is displayed in Fig. 2 (sorry for the messy setup, but we tried to keep it small, so the whole electronics fit in a small box). In the figure, a single one of the five circuits (one per motor) is marked and annotated. Please note the second resistor included in our setup, this was necessary to keep the overall current flow low, while running the device with 5.0V instead of 3.3V. If you want to use a similar setup, we highly recommend this second resistor per circuit.

Each of the two motor connections per circuit has to be connected with a motor. We realized this via a 10-pin connector, as it is shown in Fig. 3. The connections between pins and circuit have to be soldered (of course you can use a plug connection as well). We used quite long wires (2m) in our setup, to connect the motors with the 10-pin connector, to allow unrestricted hand movements.

This setup allows to control five vibration motors, i.e. one per fingertip, via the microcontroller. The software interface is described in the next section.

3 Software Interface

After assembling the circuit, we need a software interface that allows to activate and deactivate the vibration motors on demand. The interface consists of two parts, the software that runs on the Arduino controller and a higher level interface that allows to send commands to the controller from an application.

3.1 Controller Program

The controller software can be written and uploaded via the Arduino IDE, which can be obtained here. A project file (.ino file) can be found in the downloaded archive in

[1] More precisely these ones: http://www.exp-tech.de/shaftless-vibration-motor-10x3-4mm

the `UnityArduinoBridge` directory. It can be opened with the IDE. You only need to connect your controller to a PC, specify the port and hit the upload button (see Fig. 4). Now the program will run every time you provide power to the controller.

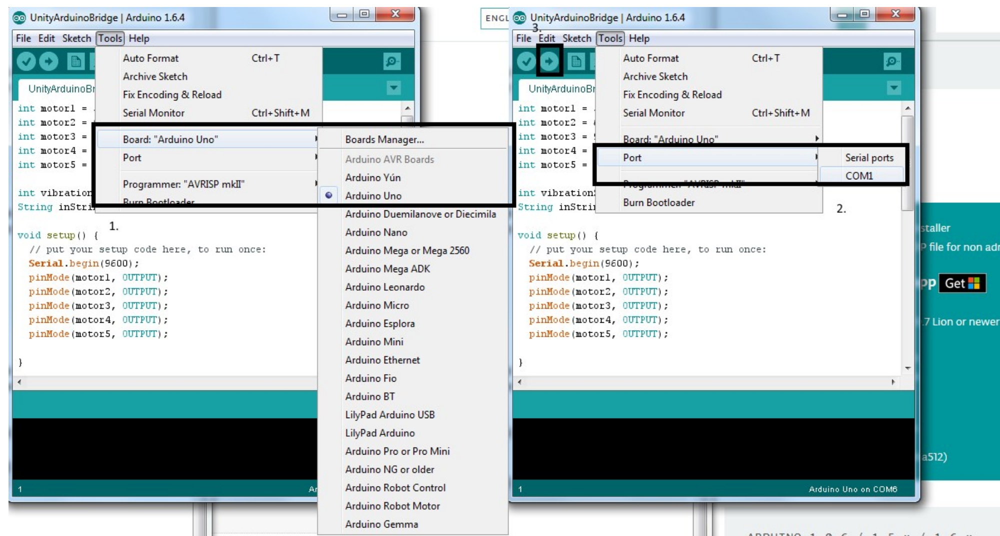

Figure 4: The controller program opened within the Arduino IDE. First, you need to specify your controller. Second, you have to indicate the port. If you are not sure to which port the Arduino controller is connected to, you can look it up in the device manager (Windows) or in the command line (`ls /dev/tty.*` on Mac OS, or `dmesg | grep tty` on Unix). After this, you can upload the code by hitting the respective button. Before uploading, you can verify the code using the respective button (the check-mark button in the upper left corner).

Lets have a closer look at the actual program code. It consists of three parts, the definition of the global variables, a setup method and a continuously running loop. The global variables are specified at the beginning of the file:

```
1 // these are the pwm pins; they provide the input to the motor circuits
2 int motor1 = 5;
3 int motor2 = 6;
4 int motor3 = 9;
5 int motor4 = 10;
6 int motor5 = 3;
```

The first five integers refer to the pin indices of the PWM outputs. PWM pins are marked on the board with a tilde, you can see the respective pins in the schematic as well (Fig. 1). Next, there is the `setup` method which is carried out when the controller is powered up:

```
// this is the initialization, the setup function is called once on start-up
void setup() {
 // we specify the data rate in bits per second (baud) that is how fast a single symbol is transmitted
 Serial.begin(9600);
 // we tell the controller to use the PWM pins as output
 pinMode(motor1, OUTPUT);
 pinMode(motor2, OUTPUT);
 pinMode(motor3, OUTPUT);
 pinMode(motor4, OUTPUT);
 pinMode(motor5, OUTPUT);
}
```

The data rate is specified in the fourth line, after this we define the PWM pins as outputs. The loop that handles the actual communication comes next:

```
// the loop function runs continuously, we can use it to check whether control commands have been send
void loop() {
 // is there some input available?
 while(Serial.available() > 0) {
  // if so, we try to obtain the control values in terms of integers between 0 and 255,
  // parseInt returns 0 if no valid input is available
  int inInt1 = Serial.parseInt();
  int inInt2 = Serial.parseInt();
  int inInt3 = Serial.parseInt();
  int inInt4 = Serial.parseInt();
  int inInt5 = Serial.parseInt();
  // the line break is our control sequence, if it was send along with some numbers, we apply them here
  if(Serial.read() == '\n') {
   // just to make sure the value is within the range
   inInt1 = constrain(inInt1,0,255);
   inInt2 = constrain(inInt2,0,255);
   inInt3 = constrain(inInt3,0,255);
   inInt4 = constrain(inInt4,0,255);
   inInt5 = constrain(inInt5,0,255);
   // now we send the vibration strength via the PWM pins to the motors
   analogWrite(motor1, inInt1);
   analogWrite(motor2, inInt2);
   analogWrite(motor3, inInt3);
   analogWrite(motor4, inInt4);
   analogWrite(motor5, inInt5);
  }
 }
}
```

The loop runs as long as the controller receives power. In every cycle, it looks whether data was send to the controller, if so and in case the command was terminated with a

line-break, the data is forwarded to the motors.

With this control structure, we can control the motors selectively.

3.2 Unity Interface

With the program described in the previous section, we can control the motors from every application that allows us to send data to a serial port. Now we will have a look how to do this from Unity®. The example projects have been build with Unity 5.5.0f3, you can obtain the engine here. There are two Unity® projects included in the archive, we will first have a look at the one called `BasicUnityInterface`.

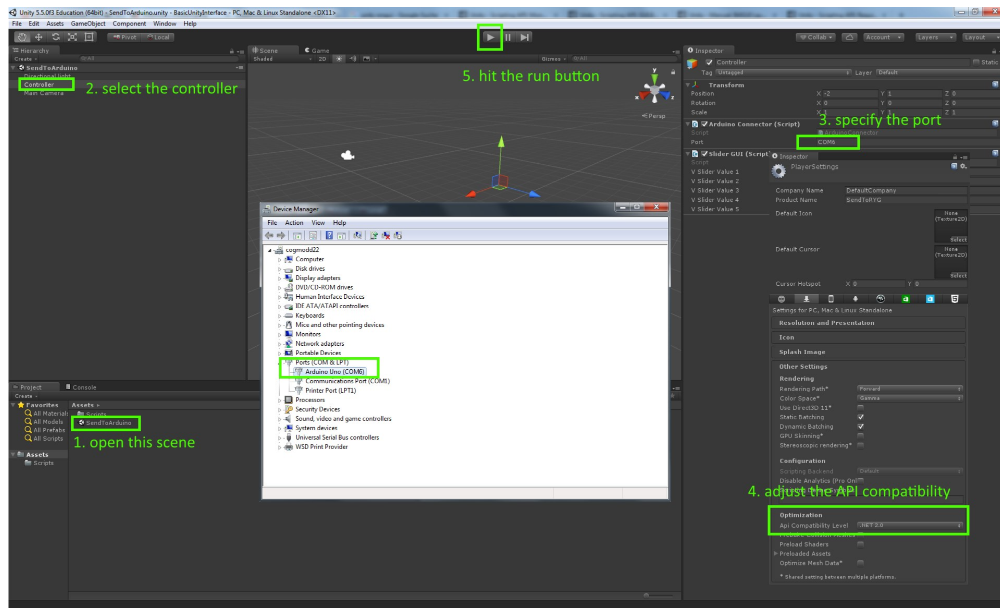

Figure 5: The basic Unity-to-Arduino interface. Just open the scene called `SendToArduino`. Open the object called `Controller` in the inspector view and enter the port the controller is connected to. You can look up the port for instance via the device manager in Windows. You also have to change the API compliance level in the **PlayerSettings** from `.Net 2.0 Subset` to `.Net 2.0`. Now you can run the scene. You will see five sliders, changes in the sliders will increase of decrease the vibration strength of the motors.

The project contains one scene and two scripts. Just open the `SendToArduino` and select the object called `Controller`. In the inspector view (usually on the right side), you will see a script called `ArduinoController` with a public variable called port. Here, you need to enter the name of the port the controller is connected to. Furthermore, you have to change the API compliance level. In order to do so, open the `PlayerSettings` via the `Edit` menu, select `Project Settings` and then `Player`. Within the `Other Settings` tab, you can change the compliance level from `.Net 2.0 Subset` to `.Net 2.0`. This is necessary, since the `ArduinoController` script relies on a wrapper class to enable

communication with a serial port. This class is not part of the libraries included in `.Net 2.0 Subset`. Now you can hit the run button. You will see five sliders, one per motor. Moving the slider will change the vibration strength of the respective motor. An overview of the different steps is shown in Fig. 5.

We now have the ability to control the vibration motors from a Unity program. Lets have a look how to combine this with motion tracking. For this example, you need a Leap motion controller and the Orion SDK. The example features the core and pinch modules from the Unity integration, which can be obtained here. The Leap sensor provides low-cost infrared hand tracking with a reasonable tracking range. Internally, the sensor software fits a handmodel to the infrared data and broadcasts the obtained joint positions and angles. Hence, it is possible to add a virtualization of your hand into a VR application. The following example shows how to augment virtual object interactions with tactile feedback. Here, we will activate the motors whenever a finger collides with a certain object.

Open the project called `LeapInteractionSample` and the scene with the same name. Plug in the Leap controller and hit the run button. Now move your right hand over the sensor and touch the cube (this should look like in Fig. 6). The motors will respond depending on how deep you put your fingers inside the cube. Sometimes the Leap sensor identifies you right hand as a left hand, if nothing happens when you touch the cube, just move your hand outside the tracking range and enter it again.

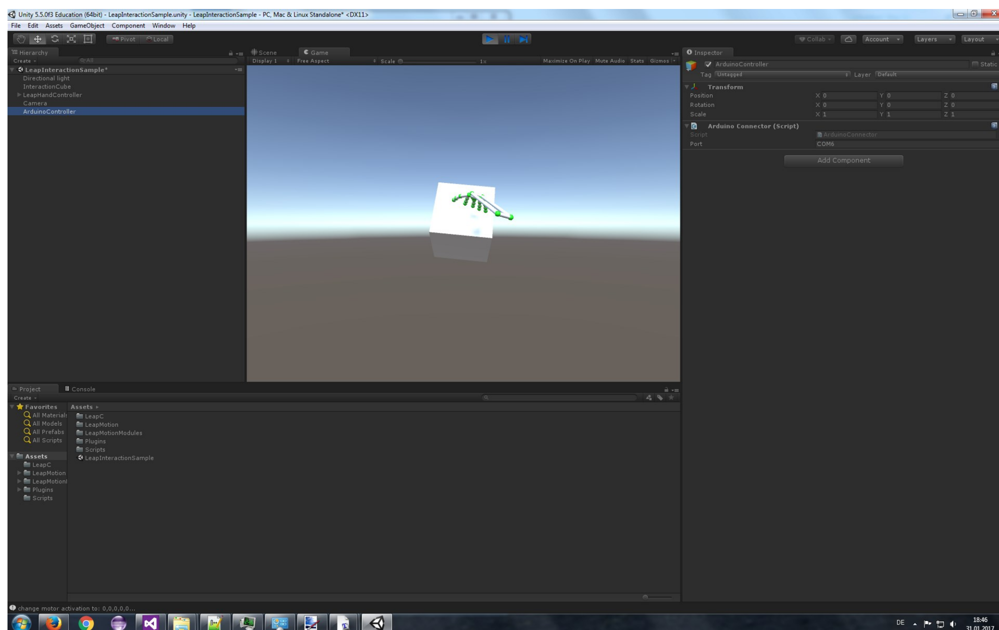

Figure 6: Like for the previous example, you have to change the API compliance level.

The basic communication logic is the same as in the previous example, the main difference is that the controller is only invoked if a collision between a right hand and the cube is detected. This is realized within the `InteractionCube` script, lets have a closer

look at this script. The `Update`, `LateUpdate`, `OnCollisionEnter`, and `OnCollisionExit` methods are provided by the Unity API. The collision methods are used to determine if a hand is currently colliding with the cube, if so, a reference to the hand is stored. In the late update method – which is carried out at the end of an update cycle – we calculate the vibration strengths and send them to the controller. This approach scales quite well, since the controller is only invoked if necessary. Within the late update we call the `processCollision` method, if the requirements are met:

```
// here, the fingertips of the colliding hand are retained, the distance to
    the
// object center is obtained and used to scale the vibration strength
private void processCollision()
{
 // we put in the vibration strengths here
 int[] values = new int[] {0, 0, 0, 0, 0};
 // these are the render bounds of the object, that is the visible volume
 Bounds bounds = this.gameObject.GetComponent<Renderer>().bounds;
 // we go through the fingers...
 foreach (FingerModel finger in this.hand.fingers)
 {
 // check if it is a defined one...
 if (finger.fingerType != Finger.FingerType.TYPE_UNKNOWN)
  {
   int index = (int) finger.fingerType;
   // we get the tip...
   Vector3 tip = finger.GetTipPosition();
   if (bounds.Contains(tip))
   {
    // calculate the distance...
    float distance = Vector3.Distance(this.transform.position, tip) /
   bounds.extents.magnitude;
    // and scale the vibration strength accordingly
    values[index] = 255 - (int) (distance * 255);
    if (values[index] < 0)
    {
     values[index] = 0;
    }
   }
  }
 }
 // here we send the current vibration strengths
 this.controller.sendData(values[0], values[1], values[2], values[3],
   values[4]);
}
```

As you can see, we go through the fingers, check if the respective tip is within the cube, that is if it is contained by the bounds, and apply the vibration accordingly.

To simulate haptic feedback, you can attach the motors to your fingers, for instance via velcro stripes. Usually tracking with the Leap sensor is less reliable when you put something on your fingertips, just put your hand in a glove (brighter material is better,

everything that absorbs infrared light is a bad choice) to keep the tracking stable. Of course the motors have to match the fingers. In the example above, the motor that is powered by PWM pin 5, corresponds to the thumb and so on. Depending on your setup, it might be useful to apply some color coding to the cables to match fingers and motors. In the top section of Fig. 3 you can see that we realized this by means of colored rubber bands attached to the cables.

That's it, with the described setup you have the possibility to augment virtual object interactions with vibrotactile feedback. Do not hesitate to write me in case you have any questions regarding the device, the controller software or the Unity integration.