User Manual

Tim Visée & Nathan Bakhuijzen

October 2018

1 Introduction

The purpose of this manual is twofold:

- To inform research how to use the platform to execute experiments and conduct research
- To inform future developers how to continue developing the platform

1.1 Problem Definition

What physical motions are natural, effortless and easy, in order to control a computer or other digital device?

Questions of this nature can be answered using the *Can't Touch This* platform. *Can't Touch This* aims to be a platform for researchers that would like to conduct research in the field of touchless computer systems. We believe that our platform allows researchers to build a strong foundation for the future of touchless control. Giving researchers the opportunity to conduct research improves the chance for touchless control of computers only seen in futuristic movies and tv shows.

1.2 Motivation

At the start of the KB-80 minor, students were given a choice in the subject of the research. Mister Al-Ers introduced us to a series of subjects, of which the LeapMotion project was the most interesting. The idea behind the LeapMotion project was to create or extend existing software to enable people to control a computer without touching any peripherals, like keyboards and mice. The idea for using such as solution in medical/sterile environments was mentioned. This project seemed cool to us because working with a sensor, processing its data and detecting gestures did sound challenging.

1.3 Background information

Research in the field of touchless computer systems is motivated by the desire for comparable systems in sterile environments. For example, surgeons often make use of computer systems to aid them during their surgeries by providing crucial information such as CT, MRI and X-ray scans. This is where touchless computer systems may offer a solution. Touchless computer systems allows surgeons to control a system without the need for physical peripherals.

2 Installation Guide

The *Can't Touch This* platform requires the following:

- A computer with the Windows (7+), OSX (10.7+, Lion+) or Linux (kernel 2.6.18+) operating system
- The Rust programming language must be installed, recommended to install with rustup. Rust nightly must be used (`rustup default nightly`)
- The Cmake toolchain must be installed
- The pkg-config tool must be installed
- The LeapMotion SDK V2 must be installed
- The physical LeapMotion device itself
- The *Can't Touch This* platform sources

2.1 Software Dependencies

The *Can't Touch This* platform is written using the Rust programming language. This means that the operating system that the platform will run on must also support the Rust programming language. Fortunately, Rust runs on all popular operating systems today, shown above in the list of requirements. An up-to-date list of all supported versions can be found on the Rust website. Additonally, the *Can't Touch This* platform requires the LeapMotion SDK to provide all necessary sensor data. Just like the Rust programming language, the LeapMotion SDK can be installed on all platforms.

2.1.1 Install Rust

Installing the Rust programming language and it's toolchain is quite simple. The rustup tool can be used for this. The rustup website shows how the tool is installed on your platform.
Check it out here: `https://rustup.rs`

To install rustup on Linux, first make sure you've `curl` installed. Then, initiate the installation by running the following command in a terminal:

```
curl https://sh.rustup.rs -sSf | sh
```

Listing 1: Install rustup and Rust

Follow the instructions shown by the installer, and make sure you select the `nightly` toolchain tuple. When installation is complete, you may check whether the Rust and carbo binaries are found by invoking:

```
cargo --version
rustc --version
```

Listing 2: Check whether Rust and Cargo are installed properly

Nightly Because this project currently requires a Rust nightly version due to the `Rocket` crate (library), it might break in future Rust versions. The last Rust version we've tried to be working successfully is `2018-10-30`. Generally speaking, if the project compiles, it works. If it doesn't, switch to this version using the following commands:

```
rustup toolchain install nightly-2018-10-30
rustup default nightly-2018-10-30
```

Listing 3: Switch to old Rust nightly

You can always check what the last successful Continuous Integration build was here, and use that Rust version.

2.1.2 Install Cmake and pkg-config

How to install `Cmake` and `pkg-config` falls outside the scope of this manual and varies greatly between platform and version.

It's recommended to check out the respective websites for installation instructions.

- https://cmake.org/install/
- https://www.freedesktop.org/wiki/Software/pkg-config/

2.1.3 Install LeapMotion SDK V2

The LeapMotion SDK V2 can be downloaded from this page. At the moment of writing this manual, an account is required thus you must register in order to to start the download. For developing this platform we've used version `v2.3.1`, although newer versions having the major version V2 should work fine.

Windows On Windows, using the LeapMotion SDK installer obtained from the ZIP file provided on their website should be sufficient. Be sure to check the included `README` for an up-to-date set of installation instructions. If you want to be able to build the platform from source, make sure you also install the SDK files.

Linux & macOS Installing the SDK on Linux or macOS requires a little bit more work. First install the LeapMotion daemon and control panel using the installer provided in the ZIP file from their website. Then the library files must be installed.

Because some manual work is required for this, we've created a simple installation script (that is also used in the Continuous Integration environment), and we recommend to use this script. To do this, first clone our project repository. Then navigate into the `ci` directory, and invoke the `install_sdk` script:

```
# Clone the repository, switch into it
git clone https://gitlab.com/timvisee/cant-touch-this.git
cd cant-touch-this

# Move to the script directory
cd ci

# Invoke the script as sudo
sudo ./install_sdk
```

Listing 4: Install the LeapMotion SDK library files with script

You may also inspect this script to do each step manually. It can also be found here.

Alternatively you can manually install these files. For this, you must move the library files into the respective location on your machine.

```
cp ./LeapSDK/include/Leap*.h /usr/include/
cp ./LeapSDK/lib/x64/libLeap.so /usr/lib/
```

Listing 5: Copy and install LeapMotion library files

The installed `pkg-config` tool is used to tell the compiler where the library installed. For this, a `pkg-config` configuration file must be created. An example can be found in the project repository here. Name it `libleap.pc` and place it at `/usr/lib/pkgconfig/libleap.pc`. If you've used the `install_sdk` script this is already done for you.

To validate that the `pkg-config` configuration file was installed correctly, run the command `pkg-config --libs libleap`. This should report `-lLeap`.

Again, be sure to read the included `README` for up-to-date instructions.

2.2 External resources

No additonal resources are required to run the *Can't Touch This* platform.

2.3 External development tools

Continuing development of the *Can't Touch This* platform requires basic tools like a text editor or IDE, and a terminal. It is highly recommended to use git, as this was used during development of the platform. Additionally, setting up an Continuous Integration server may prove useful. The current project repositories have a Continuous Integration environment configured for automated testing, but setting one up beyond the scope of this manual

2.4 Build & run platform

Now you've set up the sources for the platform, and build it from source.

To start, clone the project repository on your machine. See:

```
git clone https://gitlab.com/timvisee/cant-touch-this.git
cd cant-touch-this
```

Listing 6: Clone the platform sources

Once cloned and switched into the directory, you can start using Rust to build the project. It is recommended to check whether the project compiles first. For that, use the following command:

```
cargo check
```

Listing 7: Test compilation

On success, you can start building the project. There are two variants. By default a `debug` version is built, this includes information for debugging. You can also build a `release` version using the `--release` flag. These release builds are faster, however; at the moment of writing there seems to be a problem in Rust nightly that is causing `release` builds to crash (to segfault, to be exact). You should probably test a `release` build first, and fallback to a `debug` build if it crashes. Using a `debug` build doesn't matter too much in this case.

Using `cargo` you can `build` and `run` a Rust project. Build does only build the binary, which you can manually invoke later. `run` does build the project first, and runs it afterwards. This is probably what you want to use. Note that building the project for the first time (especially in `release` mode) might take some minutes. See the following example commands:

```
# Build in debug mode
cargo build

# Build in release mode
cargo build --release

# Build and run in debug or release mode
cargo run
cargo run --release

# View CLI options
cargo run -- --help
cargo run --release -- --help

# Automatically open the configuration panel
cargo run -- --open
cargo run --release -- --open
```

Listing 8: Test compilation

Congratulations! If everything works well you should now have the project up and running.

3 User Instructions

This chapter gives users instructions on how to use the *Can't Touch This* platform. It assumes that the user has followed the instructions found in the *Installation* chapter. The following instructions will detail how to setup the platform so that you can conduct the *experiments* found further into this manual.

3.1 Usage

- Attach the LeapMotion device using it's provided USB cable
- Start the LeapMotion daemon application provided by the LeapMotion SDK, optionally use the provided LeapMotion control panel for this
- Start the *Can't Touch This* platform by running the provided *Can't Touch This* executable, or run it manually in a terminal from the project directory (`cargo run`, or `cargo run --release` if supported). Rust nightly must be used (`rustup default nightly`)
- Start up an web browser (Firefox, Chrome, Safari, etc) and navigate to `http://localhost:8000` on the same machine
- Click on '*Start Recording*'
- Move your physical hand above the LeapMotion device to make a desired gesture
- Once you are done making the gesture, click on '*Stop Recording*'
- The recorded gesture you've just made should be visibly represented on the canvas. Trim it to cut off undesired parts of the recorded gesture
- Name the new gesture template, and save the gesture by clicking on '*Save Recording*'
- After recording, attempt to perform a recorded gesture above the sensor. The computer will give positive feedback by showing a notification in the web interface if the gesture is recognized
- The live visualizer can be toggled using the '*Visualize*' button

4 Requirements

This chapter details the list of open and finished requirements of the *Can't Touch This* platform. The prioritization of this list is not according to the MoSCoW method, because we had to write the entire platform from scratch. This took a long time, with lots of challenges and unexpected issues. We therefore chose to figure out what to do on the fly, rather than planning things out beforehand.

The following requirements have been satisfied:

- Operating System independant
- Web interface for user interaction
- Predefined gestures
- Record new gestures
- Gestures management
- Multi-finger recognition

By using the Rust programming language we inherently met the operating system requirement. This saved us a lot of work and allowed us to focus on the platform's functionality. Similarly, we chose to use a web interface for the user interaction. These decisions saved us a substantial workload.

The following requirements are left unsatisfied:

- Gesture bound actions
- Combining multiple sensors

Gesture bound actions are not yet implemented, because we did not believe this is an important feature. We instead worked on gesture recognition and template management. Instead of binding actions to gestures, we give users visual feedback when the gestures is recognized.

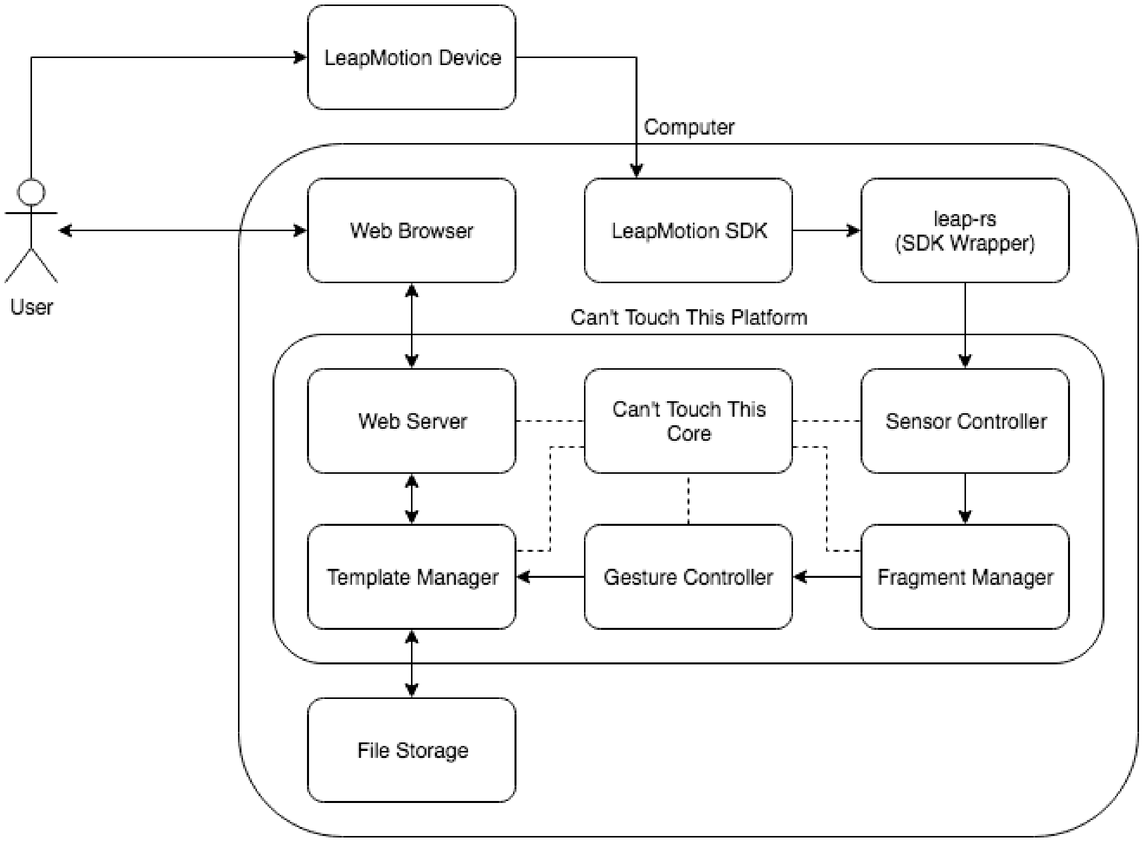
When we started out with the project, we wanted to see if it was possible to combine multiple LeapMotion sensors in order to achieve increased data accuracy. We looked into this, but we found that this was impossible with the current released SDK. Implementing it ourselves was not possible, due to the SDK being proprietary.

We contacted the LeapMotion team ourselves, but unfortunately they will not continue development on the LeapMotion, or at least not the version we're using. They're currently working on some new projects with Augmented Reality. New and more interesting projects keep the team occupied, and the LeapMotion itself is approaching obsolescence.

5 Architecture Diagrams

In this diagram, the *Can't Touch This* platform displayed globally. It shows three main objects:

- The user
- The LeapMotion device
- The Can't Touch This platform



Figuur 1: Context Diagram

The user attaches the LeapMotion device to the computer, installs the platform with it's required dependencies and moves it's hand above the sensor to conduct an experiment. The LeapMotion device shown at the top of figure 1 above acts similar to a webcam, and captures visual data which is send to the computer the device is attached to. The LeapMotion SDK (daemon) running on this machine uses this visual data to detect hands a user is hovering above the sensor.

5.1 LeapMotion SDK

This LeapMotion SDK consists of various components, as listed below:

- Daemon: *A piece of software running on the host computer, to manage and talk to physical LeapMotion sensors connected to the computer.*
- Library: *The software library that can be used to talk to the daemon through your own software which supports various programming languages, to obtain sensor data. The daemon must of course be running in order for it to work.*
- Control panel: *A tool included to manage LeapMotion sensors, the state of the daemon and other things.*

- Visualizer: *An application showing a 3D perspective of the detected hands, useful for visual sensor feedback.*

Along with the named components above, documentation and some examples are included as well.

5.2 Rust & Wrapper

We've decided to write our platform in the Rust programming language. There is no native LeapMotion library available for use with this language. Therefore we were required to use and develop a custom wrapper to translate parts of the provided library to something Rust supports, essentially being a Rust library. Our implementation is based on the open-source leap-rs project. This was the only Rust implementation of code that could talk to the LeapMotion library we could find.

Because it was quite limited, we did fork the project to extend it with new functionality required for our project. For example, we added support for obtaining finger data being part of a hand. This allowed us to check what fingers were in view, what fingers were extended and so on. We did push our improvements to the upstream project, which were accepted and merged!

5.3 Sensor Controller

Our software, the respective platform, talks to the LeapMotion daemon being part of the LeapMotion SDK to obtain data from the sensor. This is done in the `SensorController` shown in the figure above. Note that the figure clearly visualizes that all communication with the LeapMotion SDK is done through the `leap-rs` wrapper. This is also the start of what we call `the pipeline` within our platform. This pipeline is a virtual concept for how data flows through our program, from raw sensor data to gesture detection. Figure 2 (functional diagram) illustrates this in better detail.

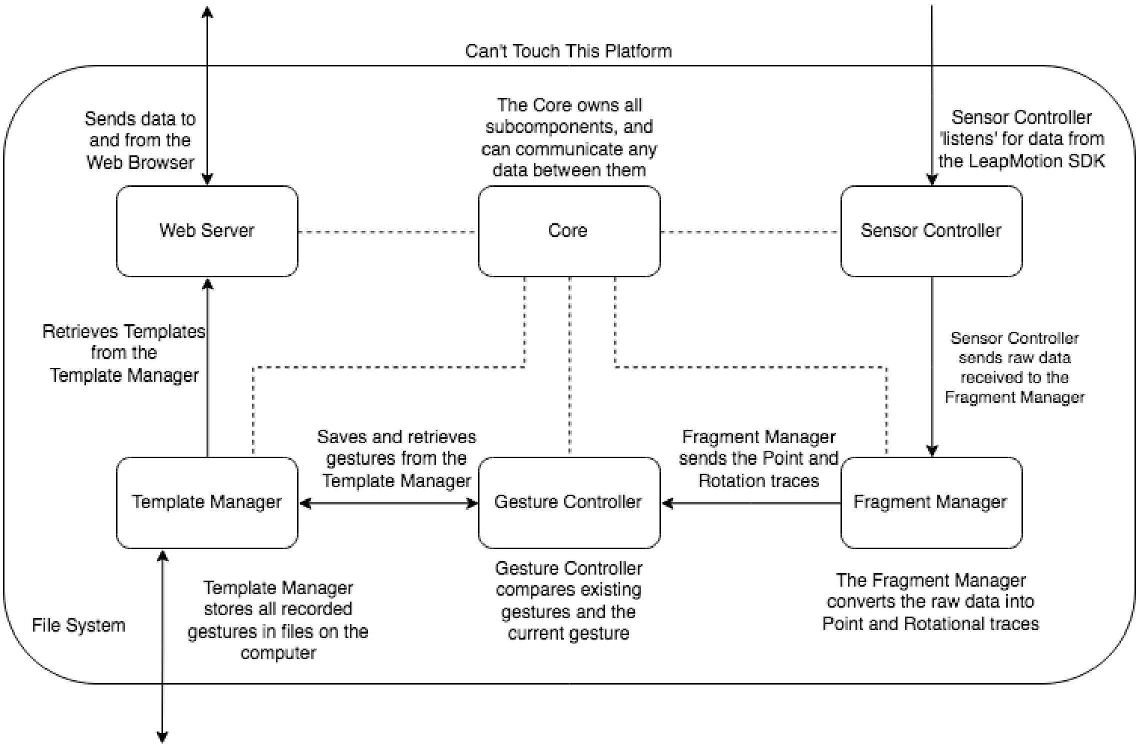
5.4 Fragment Manager

The `SensorController` controls any connected sensors and attempts to collect sensor data, which it passes on to the `FragmentManager`. This manager organizes and tracks sensor data. Raw incomming coordinate data is identified and grouped by hand and finger. Over a period of time the manager collects traces for detected fingers, which essentially define the path fingers have moved above the sensor until then. Along with collecting and organizing raw sensor data, the fragment manager processes the data into something we can easily use further down the `pipeline`. In our case, we resample, scale, filter and map 3D point data into rotational point data.

5.4.1 Resample

The LeapMotion sensor reports hand coordinate updates at varying rates, while the sensor is used. This causes collected 3D point data to have greatly varying distances between points, which makes further using the data quite a challenge. Therefore, the first step of processing the data is resampling 3D points with equal distance along the general curve the user had drawn above the sensor. The distance is small enough to represent the slightest curves and figures a user would intentionally perform, to ensure no important curve data is lost. Any exceptional disproportionate points are filtered from the stream of 3D points.

After resampling, the platform scales the drawn figure to a consistent size for easy comparison later on in the pipeline when attempting to detect gestures.



Figuur 2: Context Diagram

5.4.2 Rotational trace

Lastly, we map the new list of 3D points into rotational data. To achieve this, we calculate the angle in 2D space between all 3D points. For each of these angles we determine the difference to define a new list of points having a relative rotation and a constant distance to the next point. This rotational data along with the point distance is sufficient for our logic, thus we throw away the 3D coordinate data and are left with a trace of rotational points.

5.4.3 Example

To better illustrate the result data, here we'll go through two examples. Imagine drawing a perfectly straight line, this would map into a list of rotational points all having a relative rotation of 0° (degrees). The number of points would depend on how long the drawn line is, and what the constant distance is between the points when resampling.

When drawing a perfect circle, the sensor data would map into a list of rotational points all having a relative rotation of about 15°. The smaller the radius of the circle, the larger the relative rotation would become. When drawing the circle the other way around, the relative rotation for each point would be negated.

You can probably imagine that this rotational trace data is good enough to define an abstract variant of a figure a user has drawn with his finger, as this data defines the distance and curve a finger has been moved. This is exactly what we're using and are doing comparisons with while attempting to detect gestures, comparing the general figure of a curve being performed by a user.

5.5 Gesture Controller

For each new data point being collected by the `FragmentManager`, the then processed data is passed to the `GestureController`. Here is where the gesture detection magic happens. This controller connects to the `TemplateManager` which holds a database of known gestures that a user has defined.

The controller is constantly comparing incoming data against all defined templates until a match is found for the curve a user is currently performing. Simply said; this is done by walking through all rotational points for the current rotational trace and all rotational traces from all templates from the end (the newest point) to the beginning (the oldest point). If a template trace starts to differentiate too much from the current curve, that option is dropped. If walking through a template trace fully succeeds, a match is found.

Some constants and thresholds are used to determine whether a step is allowed when walking through, these are configurable in the source code of our platform itself. This allows testing of various threshold configurations for determining what produces the most accurate results. When a match is found, the user is notified with the name of the template that was detected.

We choose this approach with a custom gesture recognition system, because we were unable to find a Rust library that did have support for gesture recognition. Also, outside the scope of Rust the detection libraries are severely limited. Some are limited to a few hard-coded gestures and don't support shape matching. Implementations for touchpads on laptops are proprietary.

The LeapMotion SDK implements basic gesture recognition itself. This is however limited to a few static actions. Namely:

- A circle gesture
- A swipe gesture
- A tap gesture

The SDK doesn't support adding more gestures. And, because the SDK is proprietary we were not able to extract the detection logic from this to implement ourselves.

The Xbox Kinect had a few interesting gesture recognition libraries. One of these is KinectDTW. This library is also limited to some very basic actions, and wasn't viable for our purpose.

The OpenLeapKit project provided some awesome resources, but didn't really get into actual dynamic gesture recognition.

Two forum posts discussing hand gesture detection suggested to use an machine learning approach, for which you train a detection model. This wasn't a viable option for us though, as such models need to be trained with a lot of data. Doing this physically isn't really possible, thus we'd have to build a simulator and generate a lot of sample data to train. Building this would be a project on it's own.

In the end we came across HHM (Hidden Markov Models), and the term Circular Measurements, the two seemingly go-to methods for doing gesture recognition in our context.

After researching both we did decide to go with the latter. Not just because it seemed to be simpler to implement. But also because it is must more dynamic because it only compares and tries to match two gestures, and didn't require us to specify (and/or train) detection models. Recording a new gesture with the *Circular Measurement* method is as simple as converting the sensor data to data we normally use within the platform, and collecting it to a file to compare new gestures against later.

We thought implementing a system for this was an awesome challenge to use for gesture recognition.

5.6 Web interface

The platform provides a web interface for control. First of all, this interface provides a real-time visualiser for the processed sensor data. To achieve this, the web backend in our platform talks to various components in the `pipeline` of the application. The visualizer clearly shows the current curve a user has drawn, along with the resample points. This visual feedback should improve the quality of gestures users can perform as users can adapt to what they're seeing is being detected.

Templates are also managed through this interface as stored in the `TemplateManager`, for recording new templates and removing existing ones. When recording a new template, the visualizer shows the current state of the drawn figure until the user stops the recording. Recorded gestures can then be trimmed through the interface to cut off unwanted parts.

File storage is used to store user defined templates on the host machine, to properly remember templates across platform restarts. File storage has been chosen in contrast to a fully featured database, as it makes the platform much easier to set up.

When looking at the live visualizer for the first time, you might see that the visualized figure is drawn in a different orientation than your performed gesture above the sensor. This is a side effect of the data we're using. The shown gestures are based on data after processing in the `FragmentManager`. As noted, this only outputs relative rotation data along with a constant distance. Because the rotation is relative the visualizer has no concept of in what real-world orientation the gesture is performed. The visualizer always starts drawing the processed trace into the same direction (form the center of the visualizer, toward the left of the screen). This does not affect the actual detection at all. And in our opinion it isn't important enough to include the rotation relative to the world just to better visualize the performed gesture.

6 Test Report

6.1 Code Quality

In order to keep quality of of the codebase in this project high, we've done the following things:

We've clearly defined a `pipeline`, and modularized the project this way. Logic is localized to the corresponding module, and modules interconnect to pass data along as described in figure 2. First of all, this makes the complete project much easier to understand as it's usually immediately visible in what part of the program something happens. Secondly, this makes working together on the project much better as developers can focus on the module they're working on without interfering with someone else's changes. Although unsure at this time, it probably makes modifying the code base to test various configurations for processing sensor data much better doable.

We use code linting to enforce a consistent code style across the whole project. This is done using rustfmt, which is an additional tool that can be installed for Rust to format your whole project. This way developers work with a consistent code base. Along with that, another Rust tool called clippy is used for additional linting to suggest useful code logic improvements to achieve idiomatic Rust code.

Some unit tests have been implemented to ensure the most important parts of the project are working properly. Think of testing the logic to map a list of 3D points into a trace of rotations, and a test to confirm resampling works properly. The unit tests itself are quite limited due to time constraints, this is explained in more details later on.

Our code base is hosted on GitLab, on which we've enabled Continuous Integration support. For each commit we push to the repository a server is automatically spun up to build and test (unit tests) the project to verify it is working properly. Along with testing, the Continuous Integration environment verifies that the code is properly formatted following the `rustfmt` guidelines, and if it isnt́ the *build* fails. The latest Continuous Integration builds for the platform itself can be found here on GitLab.

Along with this, the repository has been configured to block direct pushes to the `master` branch. Therefore a development branch must be created for each feature, after which a merge request can be created to merge new changes into the `master` branch, only if all tests through Continuous Integration succeed.

6.2 Existing Tests

The platform currently has a few unit tests that cover basic operations, such as the conversion of a `Point` to a `PointTrace`, and more. We also have set up a few tests to cover the comparison of traces, such as straight lines and curves. Below you can see the a unit test for a straight line code:

```rust
#[test]
fn straight() {
    let points = PointTrace::new(vec![
        Point3::new(0.0, 0.0, 0.0),
        Point3::new(1.0, 1.0, 0.0),
        Point3::new(5.0, 5.0, 0.0),
    ]);

    let expected = RotTrace::new(vec![RotPoint::new(0f64, 2f64.sqrt())]);

    assert_eq!(points.to_rot_trace(false), expected);
```

```
12 }
```

Listing 9: Straight line unit test

This unit test creates `points`, a trace of `Point3`'s, and compares it to `expected`, a `RotTrace`.

The `PointTrace` on line 3 contains three points that travel the same distance at every step. The platform recognizes this as a straight line, as there is no change in the trajectory. The variable `expected` then gets assigned a `RotTrace` that contains only one `RotPoint` of 0° (degrees). This is correct, as the line drawn on line 3 is straight.

```
1  #[test]
2  fn corner() {
3      let points = PointTrace::new(vec![
4          Point3::new(0.0, 0.0, 0.0),
5          Point3::new(0.0, 5.0, 0.0),
6          Point3::new(5.0, 5.0, 0.0),
7          Point3::new(5.0, 0.0, 0.0),
8          Point3::new(0.0, 0.0, 0.0),
9      ]);
10
11     let expected = RotTrace::new(vec![RotPoint::from_degrees(-90.0, 5.0); 3]);
12
13     assert_eq!(points.to_rot_trace(false), expected);
14     assert_eq!(
15         points.to_last_rot_point(),
16         Some(RotPoint::from_degrees(-90.0, 5.0))
17     );
18 }
```

Listing 10: Corner unit test

The `corner` test is a little more complicated than the `straight` unit test. It creates a `PointTrace` with points that represent a 2D square. This motion first moves to a point, 5 units onto the `y` axis. After moving on this axis, it moves 5 units on the `x` axis. It then concludes the motion by moving back 5 units on the `y` and `x` axis, respectively.

The `RotTrace` on line 11 contains three `RotPoint`'s of `-90`°. This is because a `RotPoint` calculates itself based on three `Point`'s. In the case of this unit test, it takes three collections of three `Points`: `1-3`, `2-4` and `3-5`. For each collection, it calculates 2 angles: `1-2` and `2-3`. It then compares the difference between these angles, and that will be the `RotPoint` for those three points. This will produce three `RotPoint`'s of `-90` °.

Note that the project only includes unit tests for a few essential and easily testable parts. Due to time constraints we didn't have time left to implement more and more comprehensive tests.

6.3 Known Bugs

- At the moment of writing, there seems to be an issue in the latest few Rust nightly builds which might cause the *Can't Touch This* platform to crash when running in `--release` mode.
- On macOS, there seem to be yet unexplainable problems causing the sensor to stop reporting new data to our platform. We are unsure whether this is caused by a bad machine installation, or by macOS throttling unfocussed applications. A machine restart usually temporarily solves the issue.

7 Resources

The sources for this manual, along with all sources for all presentations given during this project, are available in the following public repository:

- `https://gitlab.com/timvisee/cant-touch-this-project.git`
- `https://github.com/timvisee/cant-touch-this-project.git` *(mirror)*

The source code for the *Can't Touch This* platform if available in the following public repository:

- `https://gitlab.com/timvisee/cant-touch-this.git`
- `https://github.com/timvisee/cant-touch-this.git` *(mirror)*