# Implementing Gesture Control for a Quadcopter

**Student Research Project**

At Course of Studies Applied Computer Science

At the Cooperative State University

Stuttgart

by

Andreas Degele

andreasdegele (at) outlook.com

08.06.2015

| | |
|---|---|
| Time of Project | 13.10.2014 – 08.06.2015 |
| Supervisor | Sebastian Bejga<br>bejga (at) lehre.dhbw-stuttgart.de |

DHBW
Duale Hochschule
Baden-Württemberg


## Abstract

The Myo is a brand new device for capturing gestures by monitoring muscle activity. For this, it uses electrodes built into an armband. At the CES trade show, the manufacturer Thalmic Labs demoed the device by flying an AR.Drone quadcopter with the Myo. However, the source code of this program was not released. Therefore, the scope of this project is to implement an application which should enable controlling the AR.Drone with the Myo armband.

At first, the paper looks into providing the necessary software infrastructure for implementing gestures. Necessary functionalities are transferring the video stream from the AR.Drone and rendering it, sending commands to the quadcopter using the keyboard, altering the configuration, and prepare the extension with different input devices, especially with the Myo. The next step was to design a gesture model for the Myo, which maps gestures and orientation data to commands for the AR.Drone. For this, the Xbox controller, which was also implemented, was used as inspiration on how to handle analogue data. In the end, the program implements the gestures and enables users to control the AR.Drone with the Myo armband.

DHBW
Duale Hochschule
Baden-Württemberg

## Table of Contents

## List of Abbreviations

| | |
|---|---|
| API | Application Programming Interface |
| CES | Consumer Electronics Show |
| FPS | Frames per Second |
| GUI | Graphical User Interface |
| Hz | Hertz |
| LAN | Local Area Network |
| ms | Milliseconds |
| PC | Personal Computer |
| SDK | Software Development Kit |
| UAV | Unmanned Aerial Vehicle |

# 1 Introduction

During the last years, human-computer interactions discovered the field of gestures as new way of interaction. More and more products aim to record movements of the human body and transform them to data usable by a computer. Early in 2015, Thalmic Labs released the Myo, which is a wearable armband designed to detect hand gestures. One year before, they presented a demo at the CES trade show, where they used the Myo armband to control an AR.Drone quadcopter[1]. However, an application which combines Myo and AR.Drone was not yet officially released.

Therefore, this student project looks into the capabilities of the Myo in order to design a mapping from gestures to actual commands for the drone. This information is developed further to a program for Windows named MyoPilot[2], which allows piloting the AR.Drone with the Myo.

# 2 State of the Art

## 2.1 Predecessor project LeapPilot

LeapPilot is the predecessor project of MyoPilot. It aimed to control the drone with the Leap motion sensor and speech recognition. Leap motion has infrared cameras to track hand gestures. The application stack is based on the node.js webserver and uses the chrome browser to display the GUI[3].

---

[1] Lai, 2014

[2] Source Code is available on https://github.com/it12052/MyoPilot

[3] Wolff & Pece, 2014, pp. 7, 22

LeapPilot uses the position of the hand relative to the leap's field of view to control the drone. This approach has two problems. First, there is no neutral zone, therefore the drone starts moving in three directions as soon as the leap detects a hand. Second, users cannot see the position of their hand relative to the field. The only feedback available is how fast the drone moves. There are arrows indicating movement on the GUI, but they do not display the strength of the movement. Combined with the first problem, the feedback of the GUI is hardly useful.

The project was not expanded with a Myo module. This is because of the architecture. As already mentioned, LeapPilot uses a two tier architecture with node.js in the backend. This approach seems to be popular among programmers for the drone. However, there is no real use case justifying the separation of backend and GUI. In most cases, the solution will run outdoors on a laptop. Wireless LAN is required to communicate with the drone and even if the laptop can handle a wired secondary connection, there is no reason to run GUI and backend on different machines. Instead of implementing an additional interface to pass data between both components, it was deemed easier to combine them into a new desktop application.

## 2.2 Myo by Thalmic Labs

The Myo is an armband reading muscle activity to detect gestures. Eight EMG muscle sensors placed around the arm detect movements of the hand. Therefore, the Myo requires direct contact with the skin for the electrodes to work properly. For best results, the armband is positioned below the elbow at the thickest part of the forearm where the signals are strongest[4]. After a short warmup period, the

[4] Thalmic Labs, 2015a

user performs a sync gesture. This allows the Myo to detect its orientation on the arm and calibrate gesture detection[5].

Figure 1: White Myo Armband (Thalmic Labs, 2015c)

Figure 2: Myo Gestures (Thalmic Labs, 2015c)

Currently, a Myo can detect five gestures out of the box: double tap with thumb and middle finger, spread fingers, wave right, wave left, and fist. These poses can be combined with motion data. To gather information about movement, the Myo is equipped with a gyroscope, an accelerometer, and a magnetometer[6]. This enables gestures like “make a fist and rotate”, which is used by most media applications in the Myo market to control the volume. Orientation data is represented as quaternion and can be accessed through roll, pitch, and yaw angles (Figure 5, p 6). Values from the gyroscope and accelerometer are three-dimensional vectors.

[5] Thalmic Labs, 2015b

[6] Thalmic Labs, 2015c

The Myo has a small rumble motor built in. Therefore, devices can trigger haptic feedback to the user. It allows a program to respond to an action with a short, medium, or long vibration.

The Myo armband transmits the data via Bluetooth to a compatible device. For this reason, the device needs to be Bluetooth 4.0 LE compatible. Modern smartphones already fulfil this requirement, PCs can use an USB-Adapter instead[7].

Additionally, PCs require Myo Connect to be installed and running on the PC. This is a software for Windows and Mac to set up the Myo armband. Besides, it controls applications (e. g. presentation software or music players), and mediates software access to the armband[8]. Furthermore, it displays recognized gestures in a corner of the screen.

## 2.3 AR. Drone by Parrot

As already mentioned, the gestures should control a quadcopter. For this project, the AR.Drone 2.0 is used, which is a small, civil UAV designed for augmented reality gaming. It is composed of a body housing for electronics which is the centre of an X-frame. The four rotors are mounted at the tips of the frame[9].

[7] Thalmic Labs, 2015c

[8] Thalmic Labs, 2014a

[9] Parrot, 2012, p. 5


Figure 3: AR.Drone 2.0 (Parrot, 2012, p. 5)

Two different shaped polystyrene hulls can be attached to the drone. One is the outdoor hull which only protects the body housing. This design aims for a low profile in order to reduce influence by winds. The other on is the indoor hull with focuses on protecting body housing and rotors. The latter is shown in Figure 3[10].

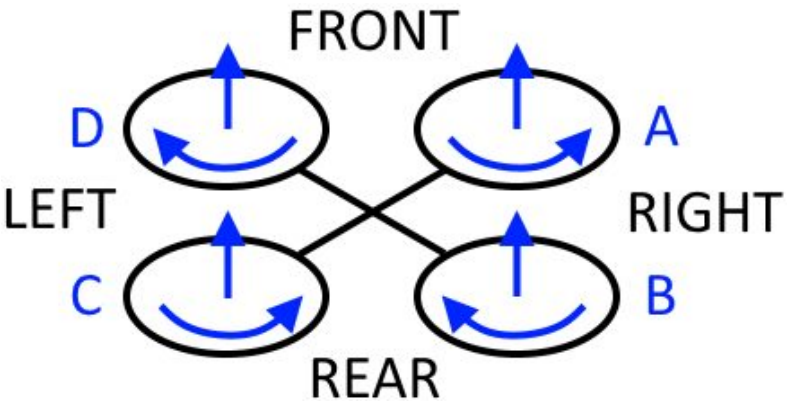


Figure 4: Drone rotors (Parrot, 2012, p. 5); modified to fit layout of AR.Drone 2.0

The rotors are grouped into pairs which turn in opposite directions. For the AR.Drone 2.0, rotor A and C turn counter clockwise, B and D turn clockwise accordingly. To achieve movement, the drone manipulates the spinning speed of the rotors. One set of rotors increases its speed whereas the other set decreases

[10] Parrot, 2012, p. 7

its speed to perform manoeuvres. The connection between manoeuvres and sets is listed in the table below[11].

| Direction | Speeding up | Slowing down | Orientation |
|---|---|---|---|
| Up | A, B, C, D | - | Gaz > 0 |
| Down | - | A, B, C, D | Gaz < 0 |
| Left | A, B | C, D | Roll < 0 |
| Right | C, D | A, B | Roll > 0 |
| Forward | B, C | A, D | Pitch < 0 |
| Backward | A, D | B, C | Pitch > 0 |
| Clockwise | A, C | B, D | Yaw >0 |
| Counter | B, D | A, C | Yaw < 0 |

Table 1: Drone movements controlled by rotor speed

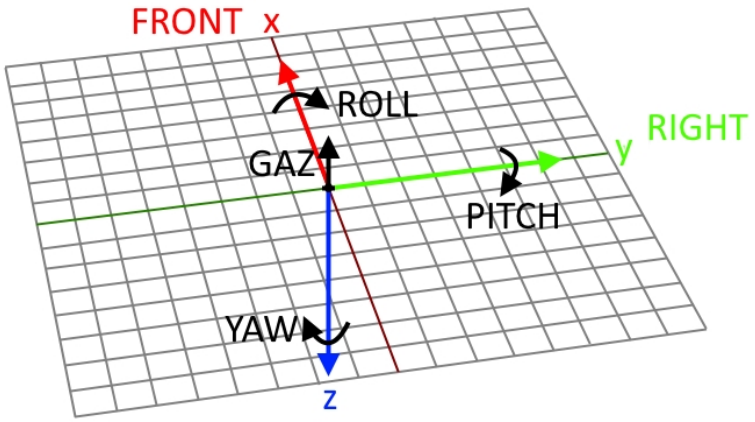


Figure 5: Orientation in 3D Space

The AR.Drone has many sensors, which help to stabilize the flight. An inertial measurement unit provides the current roll, pitch, and yaw orientation angles to the drone. The measurements are compared to the desired orientation, in order

[11] Parrot, 2012, pp. 5f, 36f

to stabilize the drone. Likewise, an ultrasound telemeter measures the current height and assists while the drone moves vertically. Because the ultrasound sensor does not work above 6 meters over the ground, a pressure sensor supports the height measurement. Additionally, the drone can measure its speed with a bottom facing camera. Since version 2.0 of the drone, it is also equipped with a magnetometer and can be extended with a GPS sensor[12].

The measurement data from the sensors allows the drone to assist the user with basic flying manoeuvres. Take-off, landing, and hovering are completely automated. Moving the drone is as simple as giving it a direction via the orientation angles, which are transformed to motor speeds by the on-board chip[13].

On start-up, the drone creates a WIFI network. Data from the drone and commands from the client are transmitted over this WIFI network. There are four different channels[14]:

- UDP port 5556: So called AT commands are sent to the drone for controlling and configuration. These commands are sent periodically, usually 30 times per second.
- UDP port 5554: All data about the drone is periodically sent to the client, including information about sensor measurements and drone state.
- TCP port 5555: This port is used for video streaming, either from the front- or bottom-facing camera.
- TCP port 5559: Critical data like configuration and acknowledging information is transmitted via this control port.

[12] Parrot, 2012, pp. 8, 11

[13] Parrot, 2012, p. 8f

[14] Parrot, 2012, p. 11

In other words, it is easy to communicate with the drone with any WIFI enabled device. The commands are on a high level, so it is possible to execute complex tasks like take-off or control movement via simple instructions.

# 3 Architectural overview about MyoPilot

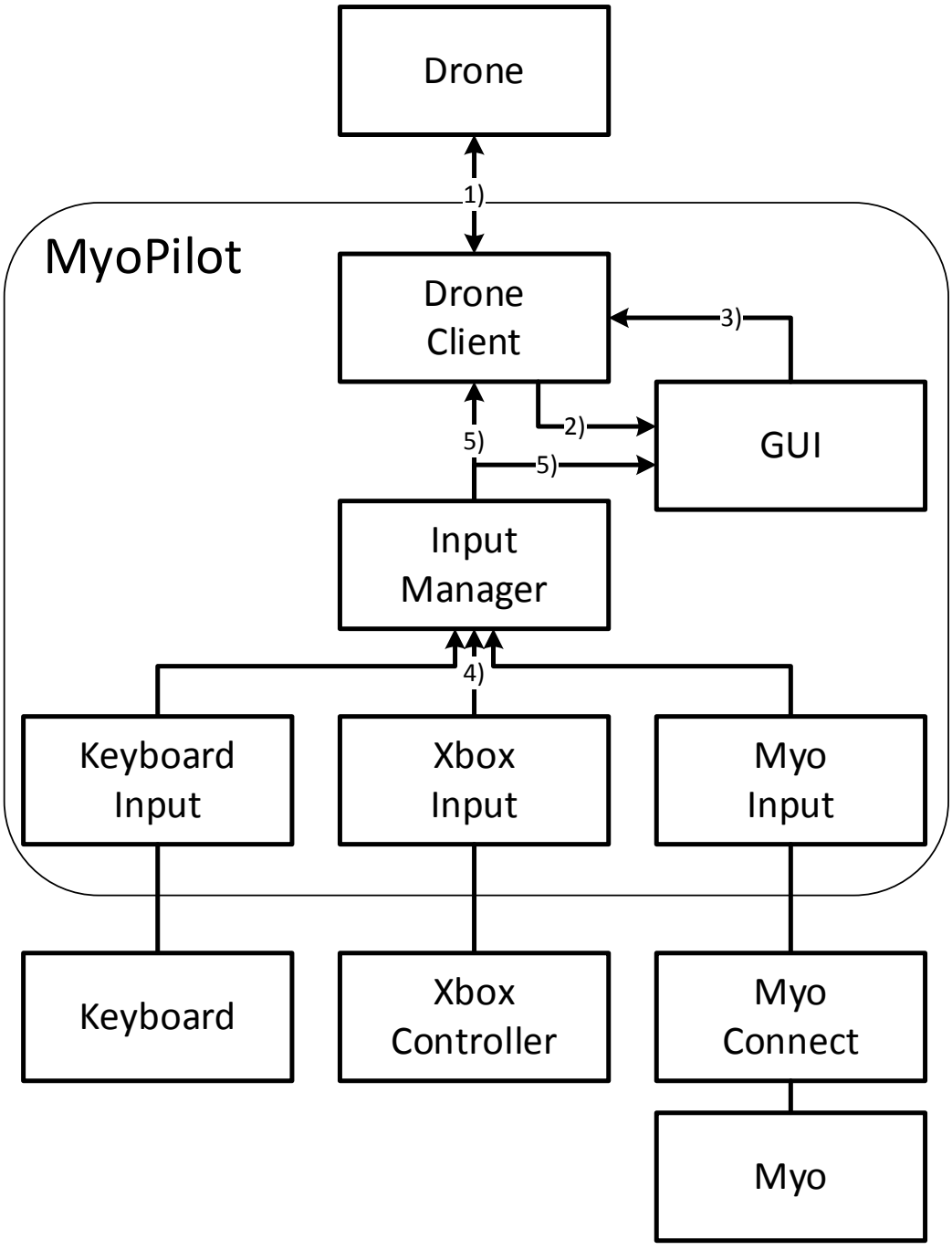


Figure 6: Architecture of MyoPilot

MyoPilot consists of several different components and has to interact with external modules. Figure 6 shows the components and how they belong together. More details about implementation of these components can be found in chapter 4.

Starting at the top, the drone component corresponds to the AR.Drone described in Chapter 2.3. To be exact, the component shown in the diagram represents the software on the drone. It is responsible to execute commands and calculate rotor movement speed, as well as stabilizing the flight. Additionally, it collects data from the sensors of the quadcopter and streams them to a client.

Closely related to the drone is the drone client. This is the communication interface inside MyoPilot, which is able to talk to the drone. They transfer data between each other via wireless LAN, over the four channels described in Chapter 2.3. The client can send commands and configurations, besides retrieving the video stream and information about the drone and its sensors. Implementation of this component was done by Ruslan Balanukhin who shared the code on github[15].

Next is the GUI, which is the primary component of MyoPilot. This is because it controls the interaction between the components according to user input. The GUI needs access to most of the other components for this reason. A majority of interactions can be setup during program start, therefore the GUI triggers and coordinates initialization. Afterwards, it sets up the connections between all components and registers itself as a listener for information about the drone, and for user input controlling the drone. Both kinds of data are rendered to the screen. Additionally, the user can open a settings window to configure the drone and MyoPilot.

[15] https://github.com/Ruslan-B/AR.Drone, LGPL license

The InputManager coordinates the components responsible for handling user input (through keyboard, Xbox controller and Myo). It aggregates the events raised by the controls so that a listener has to subscribe only to one set of events, instead of one set per control. Additionally the InputManager issues the hover-command if no other module triggers a command during that update-cycle.

Users can issue commands to the drone with either a standard PC keyboard, an Xbox 360 Controller for Windows, or the Myo gesture armband. Each of these devices has an input module which transforms a key press, a movement or a gesture to a command for the drone. However, the Myo cannot be accessed directly through the SDK. It requires Myo Connect to be installed and running on the PC, because it mediates software access to the armband[16]. Additionally, it displays identified gestures on the screen. Hence, the user knows how the Myo sees and recognizes the pose of his arm. This provides an excellent way to give feedback to the user without the need to implement it in MyoPilot, because it is already present in Myo Connect.

The components pass kinds of information between each other. Figure 6 shows the important data flows as numbered arrows.

1. The drone streams video and status information to the drone client. Status includes all information about orientation, movement, battery, and wireless link quality. Similarly, the drone client can configure the drone over this link and send commands to the drone, which executes them.
2. Video stream and status is forwarded to the GUI, which processes the data and renders it to the screen.
3. Via the settings dialog of the GUI, the user can change the configuration of the drone. These changes are transmitted by the drone client.

[16] Thalmic Labs, 2014a

4. Each of the input controllers produces commands for the drone. First, these commands are sent to the input manager for accumulation.
5. Afterwards the commands are sent to subscribers inside the application: The GUI, which provides visual feedback to the user, and the drone client for sending them to the drone.

# 4 Implementation details of MyoPilot

## 4.1 Main window

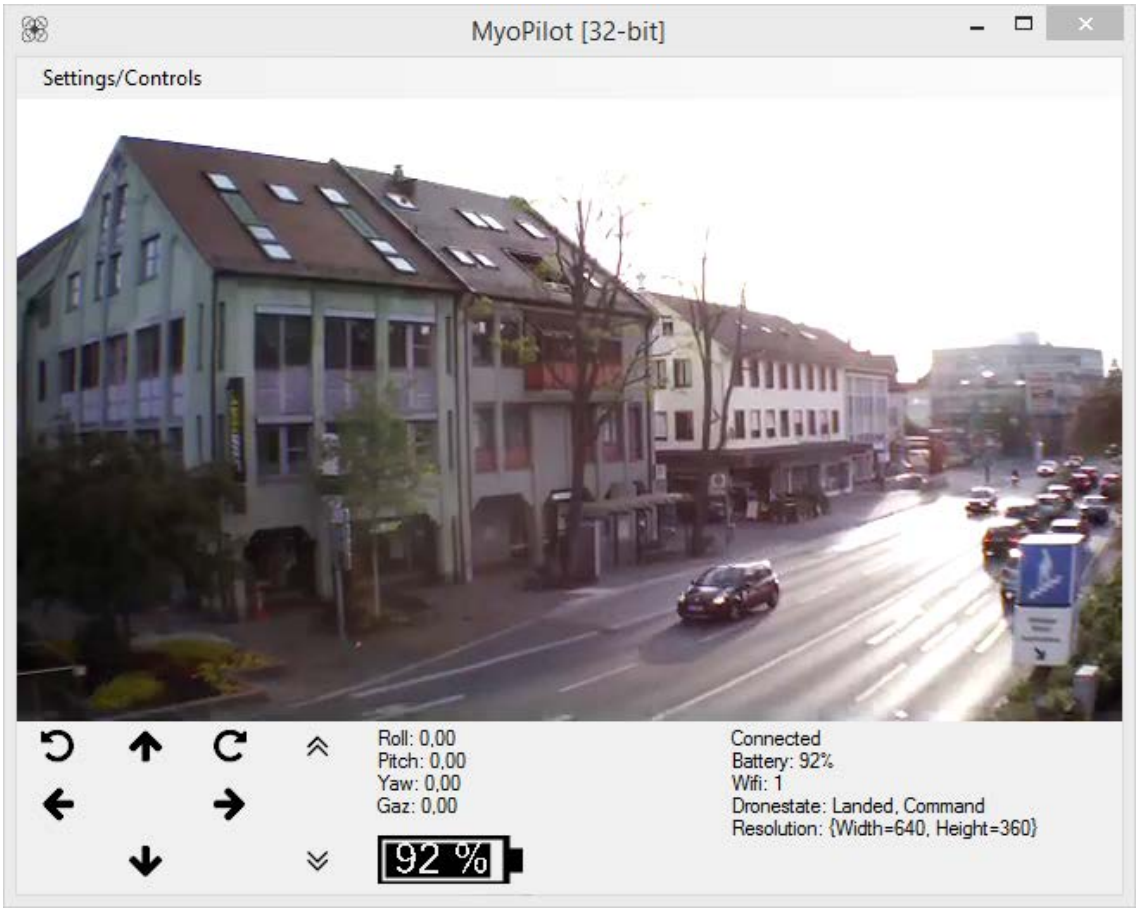


**Figure 7: Main form of MyoPilot**

MyoPilot is implemented in C# with a GUI made with Windows Forms, which is a part of the .NET libraries. Figure 7 shows the main form of the application. A second form can be displayed via clicking on the “Settings/Controls”-button in the

upper left corner. The rendering of the video stream predominates the centre of the screen. In the bottom left corner is the feedback area, which shows all directional commands currently sent to the drone. Next to the feedback area, are a label with numerical values of the directional commands and a custom control to display the charge of the drone's battery. Lastly, the bottom right corner provides additional space for debug information.

The video rendering is realized by using a picture box control. Every 30 ms (~33 Hz) MyoPilot will check whether a picture of the drone arrived in the meantime. If this is the case, it will set the image of the picture box to the most recent bitmap received from the drone. This frequency is ideal, since the drone streaming codec only supports a maximum frame rate of 30 FPS[17].

Similarly, the debug label gets updated twice a second. In this case, it is not essential to have such a high frequency within the update cycle, for the reason that the information does not change often. A half second interval is enough to keep the data on the screen up to date.

In contrast, the feedback section, displaying movement commands issued to the drone, updates as soon as directions are generated. In order to get notified, the GUI registers itself to the InputManager, which will publish all commands. It extracts movement information out of the commands and changes the colour of the arrows to red, if the drone should fly in that direction.

The straight arrows display movement in the horizontal plane, whereas the curved arrows are lit when the drone turns. Changes in altitude are shown by the double arrows.

Rendering the arrows in different colours could be done with static pictures. However, this would require two images with altered colours for each arrow. The

[17] Parrot, 2012, p. 84

idea to use an iconic font was adapted from the predecessor project, LeapPilot[18]. This approach has the advantage that the design of the icons is already finished, and colours can be changed easily by changing the font colour. Like LeapPilot, MyoPilot uses the iconic font “Font Awesome” by Dave Gandy[19]. Of course the program needs to load the font before using it to render text. Tree different ways of doing so are discussed in the following paragraphs.

The first and standard way is to use one of the installed system fonts. In code, this is merely a call to the `Font` constructor. However, this requires the font to be installed first, either by the user or an installer. An installer is currently not planned, thus users need to perform an additional step before they can build and utilize the application.

Another method loads the font from a file to a `PrivateFontCollection`. This class allows using a font without installing it[20]. For this approach, it is necessary to know the path of the font file. Visual Studio supports this task, as it can mark a file in the project as “content”. It then copies all contents to the output folder during a build. This ensures that the file is copied to the same folder as the executable and allows MyoPilot to access the file by using relative file paths.

In the third technique, the font file is loaded as a resource. This makes Visual Studio copy the file directly into the executable. When the program starts, Windows loads code and resources into memory. From there, the `PrivateFontCollection` can add them as fonts[21]. This method does not rely on external files, since everything is included in the executable. On the other hand, there are also shortcomings. The `AddMemoryFont()` method, used to import the

---

[18] Wolff & Pece, 2014, p. 32

[19] http://fontawesome.io, SIL OFL license

[20] Microsoft, 2015a

[21] Microsoft, 2015a

font to the `PrivateFontCollection`, takes a pointer of the resource's memory location as an argument. Thus, memory needs to be static. Since the C# garbage collector can relocate variables, the pointer needs to get "pinned", which is only temporary[22]. Unfortunately, it is not specified in the documentation how long pinning should occur in order to safely import the font. This may cause problems with memory management. An alternative could be to allocate unmanaged memory and duplicate the font there, but then it needs twice the space. Another downside is, that using `PrivateFontCollection.AddMemoryFont()` requires a special compatibility mode for rendering[23] which is inferior to the default rendering mode[24].

Because the third option has more disadvantages than advantages, it is not feasible to implement it. Instead, a combination of the first two was chosen: The application first checks the installed system fonts for Font Awesome. If it does not find it there, it tries to load it from the application directory. Using both methods, the application has extended robustness without much effort in implementing it.

## 4.2 Settings window

The settings window mentioned earlier will allow the user to configure the drone, as well as tweak parameters of MyoPilot. In addition, it provides information about how to fly the drone using the different inputs.

In the code, the class uses settings. This is a feature of .NET which allows to persist values between application execution sessions. Settings are basically a set of key-value pairs which are shared in the entire application. Therefore, it is straightforward to adjust a property in the settings dialog and modify behaviour of

[22] Microsoft, 2011

[23] Microsoft, 2015b

[24] Microsoft, 2015c

another class elsewhere in the program. Everything related to serialization and persistence of the values is handled automatically[25].

Like the settings, the configuration of the drone consists of key-value pairs. However, synchronization with the drone is not as smoothly. The drone occasionally refuses to accept settings pushed over the wireless link. The reason for this is unclear.

Another problem is caused when the resolution of the video stream changes. In order to save computing time, the video pipeline reuses the bitmap to store the decoded picture. If the number of pixels increases, more memory is needed to store the picture. Due to the fact that this memory is reused and not repeatedly allocated, the program will crash with a memory access violation exception. This is because it tries to write to memory it is not allowed to access. Similarly, if the resolution decreases, the picture gets distorted. To mitigate these effects, the video pipeline is stopped and reset, before the changed settings are pushed to the drone. After this, the video pipeline can be re-initialized with a different resolution. But in uncommon cases the drone does not apply the settings in time. Hence, the resolution in the stream and the resolution in the video pipeline are different, and the problems mentioned earlier still occur because of this race condition.

The settings dialog utilizes a technique called data binding. This term refers to automatic synchronization between controls of a form and an underlying data storage. This is usually a database although it can also be an object with properties[26]. In MyoPilot, this is either an instance of settings for the program or a configuration object for the drone. Per default, the data source is only updated

---

[25] Microsoft, 2006

[26] Qwertie, 2008

when the form is validated. Because the dialog has multiple controls bound to the same property, for example a slider and a number box, the binding needs to be configured to update every time the property changes. On occasion, automatic binding is not available because of data in different formats, which require conversion. For example, the drone uses radians for Euler angels, but for humans it is easier to understand angles measured in degrees. For these properties, databinding needs to be implemented manually.

## 4.3 Input handling

Input handling uses the event feature of the .NET framework. Events are messages sent by an object to a previously unknown receiver, similar to a publish-subscribe pattern. For this technique, a subscriber registers an event handler, which is basically a function, to the event. Every time the event is raised, it invokes all registered event handlers. It is possible to include data in the event message[27]. Using events simplifies programming of the classes related to input, because these classes do not need to be aware of who will consume the events.

All event handlers registered for an event need to have the same, predefined signature. Currently, the GUI and the drone client subscribe to input events. For the reason that the drone client is implemented by an external library, it is hard to modify the signature of the methods used as event handlers. Therefore, all input events have the same signature as these methods.

[27] Microsoft, 2015d

All classes in the input namespace have a shared superclass. Consequently, all classes provide the same events for theses commands:

- Emergency: Stop all rotors immediately
- Land: Automatically land the drone
- Takeoff: Take-off the drone
- Hover: Tell the drone to hold position
- Progress: Move the drone in the specified direction(s)

Considering that a subscriber would have to register event handlers for each event in every possible input module, the input manager is introduced. Its task is to hold references to available input modules and aggregate their events. Modules are added to the manager, therefore subscribers only have to register to the events of the input manager.

One problem is sending the hover command. Whenever the user does not issue a command, the drone should hold its position in the air. However, each input module is capsuled and can only notice if the user does not currently operate the associated input device. Yet it misses other modules sending commands to the drone. Obviously, sending out multiple different commands causes interference between each other. To solve this problem, the input modules do not send hover commands at all. Coupled with the assumption that the user can only use one input module at a time, this ensures that only one command is valid at a given timespan. Since the input manager already aggregates all events, it can keep track of any commands sent by the modules in the interval. If nothing was sent in a cycle, it issues the hover command itself, filling the gap in the command stream.

Of course, all input modules need computation time to read the state of the input device and generate commands. According to the drone SDK, this should happen

approximately 30 times per second[28]. Therefore, a timer calls the `processInput()` method of the input manager every 30 ms. The input manager in turn calls this method on every input module. Afterwards it checks whether any instruction was issued to the drone, otherwise it will generate the hover command. How the input modules process their input device, is described in the following chapters.

## 4.4 Keyboard Input

By default, a windows forms application uses events to get notified when the user presses or releases a key on the keyboard. However, this makes it difficult to generate commands periodically as described in chapter 4.3. It would require the module to keep track of all events, save them to a keyboard state, and generate commands from the state when the `processInput()` method is called.

Another option is to employ a library, which can be directly queried about the state of the keyboard. One class that allows accessing the keyboard state, is the `Keyboard`[29] class. It provides an `IsKeyDown()` method, which takes one value from the `Key`[30] enumeration as an argument and returns true if the user presses the button and false otherwise[31]. During the update cycle, the roll, pitch, yaw or gaz angle of the progress command are set to either +1 (maximum value) or -1 (minimum value) according to the key mapping. If no key is pressed – which means all four angles are set to zero – then no command is generated. Likewise, commands for actions like take-off are triggered when the corresponding key is in the down state.

---

[28] Parrot, 2012, p. 11

[29] `System.Windows.Input.Keyboard`

[30] `System.Windows.Input.Key`

[31] Microsoft, 2015e

| Command | Key | Command | Key | Command | Key |
|---|---|---|---|---|---|
| Forward | W | Up | Up | Takeoff | Space |
| Backward | S | Down | Down | Land | Space |
| Left | A | Turn Left | Left | Emergency | Return |
| Right | D | Turn Right | Right | | |

**Table 2: Default keymapping**

In the default keyboard configuration, the drone's horizontal movement is controlled with the W, A, S, and D keys. The up and down arrow keys make the drone vary its altitude, and pressing the left or right arrow key will tell the drone to turn. Due to the fact that it is only possible to take-off when the drone is on the ground and land when it is in the air, take-off and land can be mapped to the same key. Pressing enter will cause the drone to switch into emergency mode, which means it will stop all rotors immediately. There is no key to change the drone back to normal mode, however a button will appear on the GUI which allows to reset the emergency.

There is an option to alter the key mapping of the commands mentioned above in the settings dialog. As described in chapter 4.2, the settings-feature of .NET is used to persist the values. Unfortunately, it is not possible to save values of the type `Key`, which is needed to query the keyboard state, because it is an enumeration. As it seems, settings do not support any enumerations as type of the persisted values. However, enumerations in C# have an underlying data type, which is an integer per default[32]. For the `Key` enumeration, this can be verified

[32] Microsoft, 2015f

with the code snipped listed in the Appendix. Therefore, the value can be casted to an integer for the persistence layer, and casted back to a `Key` in the program.

Another problem is, that there is no UI control which allows the user to enter a key. For that reason, a custom control was derived from a button. Whenever the user clicks the button, the control awaits a key press and saves it. Pressing escape cancels setting the key.

The `Keyboard` class processes keyboard input independently from the keyboard handling features of Windows Forms. This is the reason keystrokes frequently manipulate the UI and fly the drone at the same time, which is undesirable behaviour. To prevent this, the method `ProcessCmdKey` of the form is overwritten to always indicate that the pressed key has been handled as a command key[33]. As a result, further processing of the key down event is disabled.

Nevertheless, the [Alt] key bypasses this interception. It could be disabled by overwriting `ProcessDialogKey` similarly to `ProcessCmdKey`, though this would interfere with significant shortcuts, for example [Alt] + [F4], which is used to terminate the program in Windows. Instead of implementing a way to allow all these shortcuts, choosing [Alt] to manoeuvre the drone was restricted.

Another detail with keyboard handling is, that it continues to work even if MyoPilot is not focused. This could lead to unexpected behavior when using another application while MyoPilot is running in the background. Therefore, a listener monitors the `Activated` and `Deactivated` events of the window and disables keyboard input when the application is out of focus.

[33] Microsoft, 2015g

## 4.5 Xbox Input

The ability to control the drone with an Xbox controller is an additional way to interact with MyoPilot. For this, an Xbox 360 controller for windows was used. MyoPilot requires the XInput API to access the controller. XInput is pre-installed on all Windows versions since Vista. For simplicity, XInputDotNet by Remi Gillig[34] was utilized. This library wraps the XInput API and enables its use through C#. It provides the state of the controller, which can be processed to commands.

Figure 8 shows the layout of the controller used for steering the drone. The left thumb stick controls horizontal movement of the drone. With the right thumb stick, the drone changes altitude or turns around the vertical axis. Other commands are issued through pressing one of the buttons.

Because of the way the API is structured, the controls are not easily customizable. Therefore, the controls are hard coded and cannot be changed in the settings menu like the keyboard.

[34] https://github.com/speps/XInputDotNet, MIT licence


Figure 8: Steering with the Xbox controller

The thumb sticks always report movement, even when the user doesn't touch the stick and they are therefore centred. For this reason, the application needs to filter out these values. This is done with so called "dead zones". Only if the coordinates returned from the thumb stick are far enough away from the centre, it is assumed that the measures are valid[35]. These dead zones can be calculated independently for each axis. Another option is to have a circular dead zone.

[35] Microsoft, 2015h

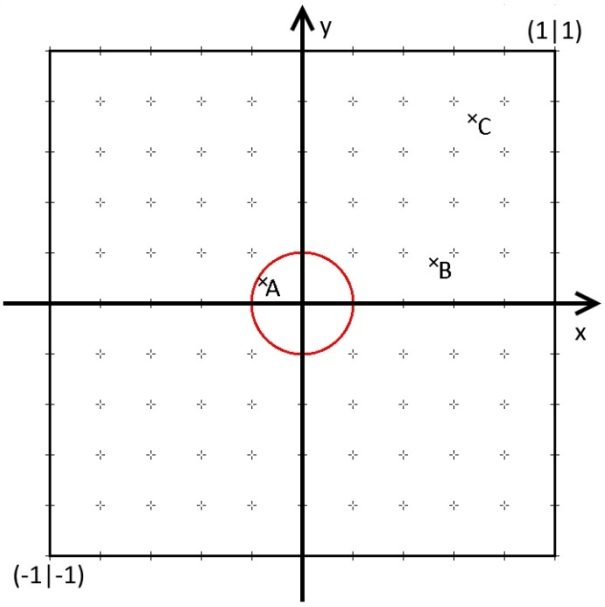


**Figure 9: Circular dead zone**

Figure 9 is a visual representation for the circular dead zone. Every point measured by the controller is transformed to a vector. After that, the length of the vector is calculated and checked against the circle diameter. If the point is inside the circle and therefore inside the dead zone, its x and y component are adjusted to zero. Only if the point is outside the circle, the values are valid. In this case, the point needs normalization so its components are in the range from 0.0 to 1.0[36].

[36] Microsoft, 2015h

The example shows three measurements. Point A is inside the dead zone. For this reason, it is considered invalid and zeroed. B and C are outside the dead zone and are valid. Their values need to get normalized.

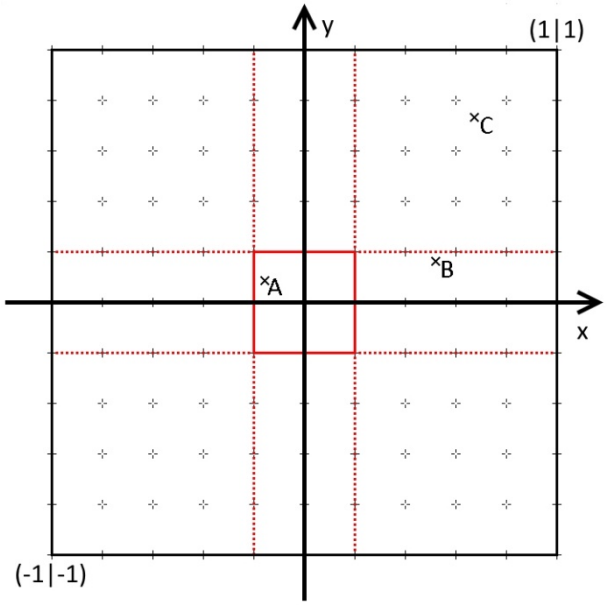


Figure 10: Dead zone with independent axis

As already mentioned, another approach looks separately at the x and y coordinate (Figure 10). This is visualized by the red lines. If one coordinate is in the area near the axis, it is zeroed. Otherwise, it needs normalization to cover the range from 0.0 to 1.0.

The graphic shows three exemplary points. Point A is both near the x axis and near the y axis, hence it is an invalid point. In contrast, point B has only an invalid y coordinate because it is close to 0.0. Its x coordinate however is valid and is transformed to the correct range. Unlike the others, point C has two usable coordinates, which only need transformation.

The application calculates the dead zones for each axis individually. This approach allows better distinguishing between the different commands. For example, when the user pushes the right thumb stick to the right in order to rotate the drone (e.g. point B in the examples), there is a little tolerance before

interferences on the y-axis influence the altitude. With the circular dead zones, the user would change the altitude simultaneously, although the changes are rather small.

Microsoft suggests to apply a non-linear transformation to the coordinates in order to improve handling in the program or game. This is caused by behaviour of the user, which either gently pushes the thumb sticks in a direction for careful movements, or shifts them all the way to move as fast as possible[37]. With this in mind, the coordinates from the thumb sticks are cubed. Hence, the input is more sensible in the lower parts.

## 4.6 Myo Input

The primary use case of MyoPilot is to use the Myo to control the drone. This is implemented in the Myo Input module. To communicate with the Myo armband, it uses the Myo.Net wrapper by Reegan Layzell[38]. In order to work properly, this project requires Myo Connect for Windows to be running on the PC.

Since the Myo armband currently supports only 5 gestures, a direct mapping to the 10 commands (8 directions + take-off + land) is neither possible nor ergonomically reasonable. Instead, the gestures are combined with the data from the gyroscope, which allows to add the orientation of the arm to the gestures.

Ideally, the gestures to control the drone start with the forearm in a horizontal position. Moving the drone in the horizontal plane is done by making a fist and then change the orientation of the forearm. Lowering the hand while making a fist will instructs the drone to move forward, raising the hand moves the drone backwards. Panning the fist to the left manoeuvres the drone to the left and vice

[37] Microsoft, 2015h

[38] https://github.com/rtlayzell/Myo.Net, MIT license

versa. To turn the drone around the vertical axis, the users rotate or twist their arm like using a key in a lock. Changing the altitude is done similar to moving forward and backward: This time, the users spread their fingers on their hand and raises it to gain altitude or lowers it to lose height. Similar to the other input modules, lift-off and landing are triggered by the same gesture, which is in this case the double tap gesture. When none of the above gestures are detected, MyoPilot instructs the drone to hover in the air.

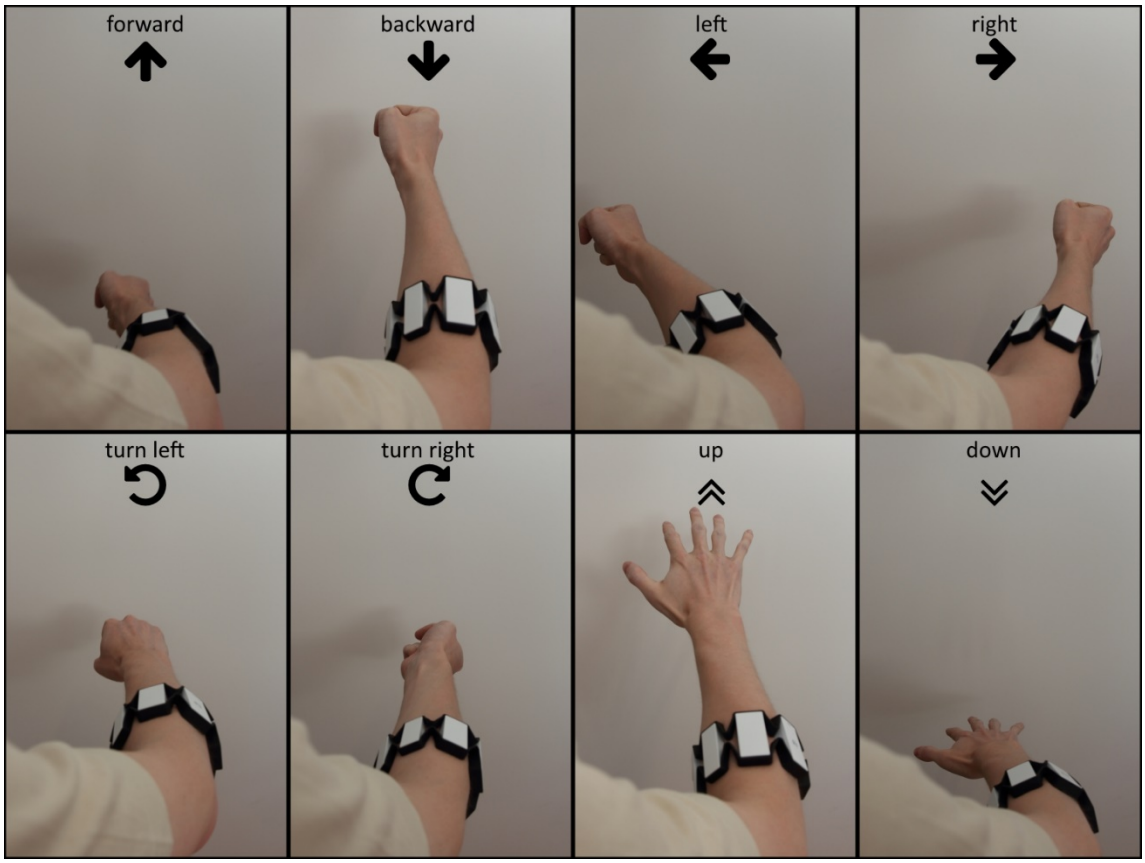


**Figure 11: Hand gestures contolling the AR.Drone**

Figure 11 visualizes the hand gestures implemented to fly the drone. In this image, the gesture for forward and backward are inspired by the movement of the drone, which lowers its nose to move forth. Whereas the arrows represent the icons found on the GUI (Chapter 4.1).

Usually, the double tap gesture is used to unlock the Myo for a short while and allow processing of further gestures. Although MyoPilot does not use this locking policy, the effects are similar, for the reason that the only command the grounded drone will accept is the take-off command. However, since normal applications require a sequence of two gestures (timed unlock + gesture) and MyoPilot does not, it is more likely to accidentally instruct the drone to start. The user should be aware of this situation. On the other hand, requiring the user to unlock the Myo for every gesture would have a negative impact on ergonomics.

There are some possible alternatives for implementing the turn left and right gestures. For example, it would be possible to use the wave left and wave right gesture for this. Another alternative would be to pan the arm left or right in combination with the spread fingers gesture. However, the way it is implemented allows to combine all movements in the horizontal plane. For example, it is possible to fly forward and turn left at the same time. As a result, the drone is more agile. Considering that there can be only one active hand pose at a time, this would not be possible with the other options.

Status information from the Myo armband is received through events. Bearing in mind that events are not directly compatible with the periodic input processing described in chapter 4.3, the Myo module has to keep track of the state manually. For this purpose, the current pose and the current orientation represent the state of the Myo. The saved state can then be processed to commands. This works well for the fist and spread fingers gestures, which are evaluated frequently to progressive commands. However, this is not possible with the double tap gesture, which is used to take-off or land, because the drone should modify its state only once. The gesture could change before the input cycle realizes it, or double tap is recognized during multiple cycles. For this reason, the event handler sets a flag to indicate that there was a double tap gesture. This happens when the Myo detects a transition from one gesture into another, thus the event is only raised

once. Inside the input cycle, this flag can be checked. The flag is reset once a take-off or land command was issued.

Although Myo Connect displays the currently detected gesture on the screen, the Myo module provides additional feedback. Considering that the user habitually observes the drone instead of the screen, the Myo module uses the vibration ability of the Myo armband. When it detects that the user makes a relevant gesture or switches from a relevant gesture to any other gesture, it instructs the armband to vibrate. If the gesture is the double tap pose used for take-off and landing, the vibration is of medium length. Otherwise, the gesture is either a fist or the spread fingers pose, which are used for progressive movement, and the Myo should vibrate for a short time span.

The values from the gyroscope are measured in radians and need to be transformed to a number in the range from -1.0 to 1.0 for the drone. Since the position of the Myo on the arm of the user can vary between different users and sessions, a reference orientation is captured every time the user switches to the fist or spread fingers gesture.

Once the orientation changes, the module calculates the difference between the two orientations for each roll, pitch, and yaw angle of the Myo (refer to Figure 5; orientation is independent for Myo and drone). Because the gyroscope returns values in a circular domain, a simple subtraction of both values will not necessarily yield the correct delta angle. The shortest way from one point to the other might go around the circle, through the $-\pi \triangleq +\pi$ point. In order to detect this, one of the points is duplicated twice. In this case, the reference measure $R_0$is duplicated to $R_1$ by subtracting $2\pi$, and $R_2$ results from adding $2\pi$ to $R_0$. Afterwards, the differences between $R_1, R_2, R_3$ and the current measure $C$ are calculated. The smallest difference corresponds to the correct delta angle. Figure 12 visualizes this approach.

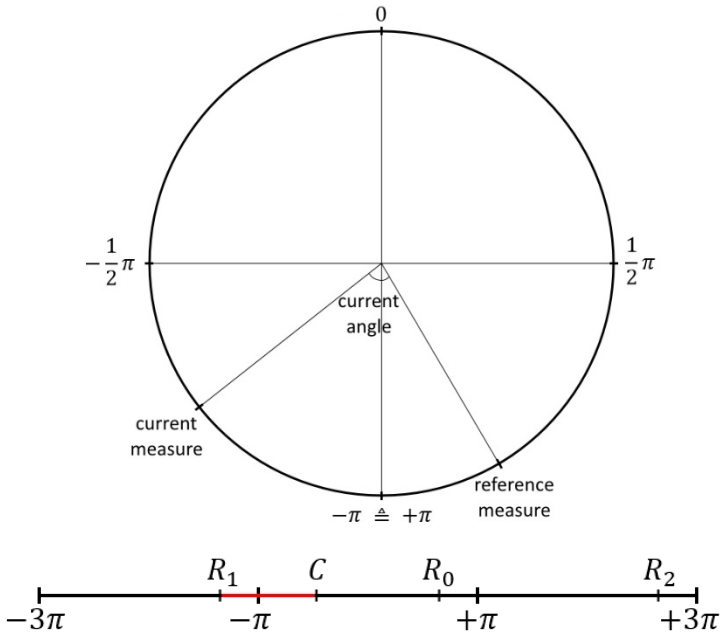


**Figure 12: Transforming values from gyroscope to relative angles**

After calculating the delta angle, the application has an analogue input. Like for the Xbox controller, a dead zone is applied to filter out the noise. Again, dead zones are applied individually for each angle, similar to the axis independent approach described in chapter 4.5. The size of each dead zone can be configured in the settings dialog.

The next step is to transform the value to a range between -1.0 and 1.0. For this, the application needs to know the angle which equals to 1.0. This angle is the maximum (max), which also can be configured within the settings. If a value is bigger than max, it is capped. Currently, there are no values smaller than the dead zone, because they are adjusted to zero in the previous step. Therefore, transformation needs to shift values in order to form a continuous range. These rules can be calculated by the following transformation function.

$$T(x) = sign(x) \cdot \frac{\min(|x|, max) - deadzone}{max - deadzone}$$

$$sign(x) = \begin{cases} -1, & x < 0 \\ 1, & x \geq 0 \end{cases}$$

$$\min(x, y) = \begin{cases} x, & x \leq y \\ y, & x > y \end{cases}$$

Finally, the results are cubed equally to the values from the Xbox controller. This should give the user more control over fine grained movements.

# 5 Comparing MyoPilot to the AutoFlight Connector

A few weeks after MyoPilot was able to pair the Myo armband with the quadcopter, Paul Bernhardt from Thalmic Labs released a connector for the same task[39]. Connectors are pieces of MyoScript, which is a dialect of LUA. They are used to extend the ability of Myo Connect to interact with applications[40]. In this case, the application to interact with, is AutoFlight[41]. It is a program which allows to fly the AR.Drone with either a keyboard or a game controller.

The gestures used to fly the drone are similar to the ones used with MyoPilot (described in chapter 4.6). The AutoFlight connector uses double tap to take-off and land the quadcopter. Waving left makes the drone descend, wave right ascends respectively. When the user makes a fist and rotates the arm, the drone will spin. In order to move the drone on the horizontal plane, the user first makes a fist briefly. This captures the current orientation and uses it as a reference. Now, panning the arm results in a movement relative to the reference orientation. Moving the hand down makes the drone move forward, raising the hand yields

[39] http://developerblog.myo.com/myocraft-fly-your-parrot-ardrone-2-0-with-autoflight/

[40] Thalmic Labs, 2014b

[41] http://lbpclabs.com/autoflight/

backward motion and panning left or right will result in a manoeuvre in that direction. Once the arm should no longer control the drone, the spread fingers gesture makes it hover at the current position[42].

The way the connector handles the reference orientation, the user needs to explicitly make the spread fingers gesture in order to decouple the Myo from sending further commands to the drone. Whereas in MyoPilot, it is sufficient to let up the fist gesture. On the other hand, it is less exhausting to navigate with the hand in the rest pose.

A look in the code of the connector reveals, that it also implements dead zones to filter out measures with irrelevant peak. Like MyoPilot, it calculates the dead zone separately for pitch, roll, and yaw.

The connector emulates keyboard input in AutoFlight. Therefore, the drone either goes full speed or does not move in a particular direction. Analogue values comparable to MyoPilot's approach are not possible, due to the fact that keys on a keyboard can only have two states. Hence, the strength of a gesture cannot be transferred via the keyboard. Connectors could use the position of the mouse to communicate this kind of data, but AutoFlight does not support this. In other words, it is impossible for the connector to leverage analogue values to determine a drone speed and sent it to AutoFlight.

In short, the connector uses a less fatiguing gesture mapping, but cannot vary the speed of the drone.

[42] Thalmic Labs, 2015d

# 6 Conclusion

In conclusion, a gesture model for controlling the AR.Drone with the Myo armband was designed. The program MyoPilot implements these gestures. It allows users to fly the drone with the Myo armband, although there is no official application yet.

Before the actual controlling code was written, the infrastructure had to be prepared. This paper describes the architecture of MyoPilot with the different components, both infrastructure and input related. It also discusses how the different parts work together, and what ideas influence their implementation. Correspondingly, it explains the input models for the different devices supported by MyoPilot. Hence, it serves as a documentation of the existing program. It therefore enables further development on MyoPilot.

Apart from controlling the AR.Drone with the Myo armband, MyoPilot supports the use of a keyboard and game controller as alternative input devices. The input mechanics are designed to allow extension with new input modules. New modules just need to be added to the input manager during initialization of the program. Obviously, replacing or modifying an existing input module is also possible.

Through this way, further projects could add a solution to pilot the drone with two Myos. This could allow the user to move the drone horizontally with one hand and vertically with the other, at the same time. Likewise, the added gesture capabilities can be used to enable controlling more features of the drone, for example trigger the “flip” flight animation or start and stop video recording. With LeapPilot in mind, it would also be possible to try a different gesture model using Leap motion. Of course, any other input method can be added, too. These additions could be done in the scope of future student projects.

# 7 Appendix – Determining underlying datatype of the Keys enumeration

The underlying datatype of an enumeration can easily be determined with this code snippet.

```
using System;

class Program
{
    static void Main(string[] args)
    {
        Type underlyingType = Enum.GetUnderlyingType(typeof(System.Windows.Input.Key));
        Console.WriteLine(underlyingType.ToString());
    }
}

/* Output:
System.Int32
*/
```

# 8 References


Lai, R. (2014, January 10). *When Parrot AR.Drone meets Myo armband, magic ensues* . Retrieved May 19, 2015, from engadget: http://www.engadget.com/2014/01/10/parrot-ar-drone-thalmic-labs-myo/

Microsoft. (2006, August). *Using Settings in C#*. Retrieved May 12, 2015, from Microsoft Developer Network: https://msdn.microsoft.com/en-us/library/aa730869(v=vs.80).aspx

Microsoft. (2011, April). *fixed Statement (C# Reference)*. Retrieved Mai 2015, 2015, from Microsoft Developer Network: https://msdn.microsoft.com/en-us/library/vstudio/f58wzh21(v=vs.100).aspx

Microsoft. (2015a). *PrivateFontCollection Class*. Retrieved Mai 12, 2015, from Microsoft Developer Network: https://msdn.microsoft.com/en-us/library/system.drawing.text.privatefontcollection(v=vs.110).aspx

Microsoft. (2015b). *PrivateFontCollection.AddMemoryFont Method*. Retrieved May 12, 2015, from Microsoft Developer Network: https://msdn.microsoft.com/en-us/library/system.drawing.text.privatefontcollection.addmemoryfont(v=vs.110).aspx

Microsoft. (2015c). *Using Fonts and Text*. Retrieved May 12, 2015, from Microsoft Developer Network: https://msdn.microsoft.com/en-us/library/a3a2bads(v=vs.110).aspx

Microsoft. (2015d). *Events and Delegates*. Retrieved May 13, 2015, from Microsoft Developer Network: https://msdn.microsoft.com/en-us/library/17sde2xt(v=VS.100).aspx

Microsoft. (2015e). *Keyboard.IsKeyDown Method*. Retrieved May 13, 2015, from Microsoft Developer Network: https://msdn.microsoft.com/en-us/library/system.windows.input.keyboard.iskeydown(v=vs.110).aspx

Microsoft. (2015f). *Enum Class*. Retrieved May 15, 2015, from Microsoft Developer Network: https://msdn.microsoft.com/en-us/library/system.enum.aspx

Microsoft. (2015g). *How Keyboard Input Works*. Retrieved May 15, 2015, from Microsoft Developer Network: https://msdn.microsoft.com/en-us/library/ms171535(v=vs.110).aspx

Microsoft. (2015h). *Getting Started With XInput*. Retrieved May 15, 2015, from Microsoft Developer Network: https://msdn.microsoft.com/en-us/library/windows/desktop/ee417001(v=vs.85).aspx#dead_zone

Parrot. (2012). *AR.Drone Developer Guide* (SDK 2.0 ed.). Retrieved November 25, 2014, from https://projects.ardrone.org/attachments/download/514/ARDrone_SDK_2_0_1.tar.gz (Registration required)

Qwertie. (2008, March 25). *A detailed Data Binding Tutorial*. Retrieved May 12, 2015, from Codeproject: http://www.codeproject.com/Articles/24656/A-Detailed-Data-Binding-Tutorial

Thalmic Labs. (2014a). *Myo SDK 0.8.1: Getting Started*. Retrieved Mai 11, 2015, from Thalmic Labs: https://developer.thalmic.com/docs/api_reference/platform/getting-started.html#myo-connect-guide

Thalmic Labs. (2014b). *Myo SDK 0.8.1: Writing Myo Scripts*. Retrieved May 20, 2015, from Thalmic Labs: https://developer.thalmic.com/docs/api_reference/platform/script-tutorial.html

Thalmic Labs. (2015a). *How do you wear the Myo armband?* Retrieved April 19, 2015, from Thalmic Labs: https://support.getmyo.com/hc/en-us/articles/201169525-How-do-you-wear-the-Myo-armband-

Thalmic Labs. (2015b, January 22). *How to perform the sync gesture*. Retrieved April 19, 2015, from Thalmic Labs: https://support.getmyo.com/hc/en-us/articles/200755509-How-to-perform-the-sync-gesture

Thalmic Labs. (2015c). *Myo - Tech Specs*. Retrieved April 19, 2015, from Thalmic Labs: https://www.thalmic.com/en/myo/techspecs

Thalmic Labs. (2015d, April). *AutoFlight - Myo Market*. Retrieved May 20, 2015, from Myo Market: https://market.myo.com/app/553d45fde4b0f33421ce1fdf/autoflight

Wolff, W., & Pece, T. (2014, June 9). *Next Generation of User Interface Design - Speech & Gesture recognition.* Retrieved Mai 12, 2015, from GitHub: https://github.com/LeapPilot/NUI/blob/master/Documentation.pdf

# 9 Glossary

| | |
|---|---|
| .NET | Software framework |
| AR.Drone 2.0 | Quadcopter |
| C# | Programming language |
| Git | Distributed revision control system |
| Github | Web-based Git repository hosting service, often used for open-source software projects |
| LUA | Scripting language |
| Myo | Armband for detecting hand gestures |
| Quaternion | Mathematical number system used in 3D calculations |
| Vector | Geometric object consisting of direction and length |
| Visual Studio | Integrated Development Environment for C# |

## 10 List of Figures

# 11 List of third-party resources utilized in MyoPilot

| Author | Description | Project | URL |
|---|---|---|---|
| Ruslan Balanukhin | AR.Drone API | AR.Drone | https://github.com/Ruslan-B/AR.Drone |
| Remi Gillig | Xbox360 controller API | XInputDotNet | https://github.com/speps/XInputDotNet |
| Reegan Layzell | Myo language bindings | Myo.Net | https://github.com/rtlayzell/Myo.Net |
| Dave Gandy | Iconic font | Font Awesome | https://fortawesome.github.io/Font-Awesome/ |
| Shannon Moore | Xbox Controller clip art | clker.com | http://www.clker.com/clipart-285099.html |
| Willi Wolff & Tunahan Pece | Drone icon | LeapPilot | https://github.com/LeapPilot/NUI |