Report on Editor for Sound and Actuators for robots

Chiara Bianchimani, Andrea Franchini

2020

Abstract

This report describes our results in creating a graphical software that allows users to generate code that can be used by a robot to perform synchronized, scripted actions to express emotions through non-verbal communication. The software has been implemented with modern web technologies as a single-page application, without the need of a server beside the one serving the files. Users can utilize a clean, modern interface to generate melodies on a timeline, similarly to a music score, and add actuators such as LEDs or motors to perform actions at a specific instant. As a proof-of-concept, the editor can export a C++ Arduino Sketch that executes the coordinated sequences of sounds and actions.

The editor is easily extensible thanks to its modular structure.

1 Introduction

Since the beginning of our pursuit of creating robots, humans always wanted them to use or at least convey emotions. Movement, sound and color are shared in many cultures and societies as a way of portraying emotions. So what better way for a robot to use a non-spoken human-understandable language to share feelings or at least their semblance.

Let's imagine two scenarios. In the first one we want to have a robot convey that it is displeased, in the other one we want the robot to look like it is happy. In the first case, a robot may signal its state with:

- a red light

- either a low pitched sound (for disappointment) or a high pitch sound (out of fear)
- erratic movement

In the other, opposite case, we may want to have the robot display:

- a green light, or a flash of different colors
- a high pitch sound, or a melody
- a dance

These actions are easily scriptable, but creating them can be quite time consuming. Our work doesn't offer users a set of pre-packaged "emotions" but allows them to have the creative freedom to choose how their robot can express a feeling, moreover having the ability to create a personality for their robot. In this report, we explore the possibility of giving users an easy-to-use software to create actions for robots that can convey emotions, with a focus on synchronizing actions (e.g. lights, motors and sound) by using a timeline, akin to what happens in most of moderns video or music editors.

Surely our software isn't the first one to do the aforementioned, there are notable examples such as the Sony Rolly, that provides an editor to create choreographies on a timeline, but others visual editors for robots may achieve similar results, such as the LEGO Mindstorm one.

Our focus is developing an easily accessible software with a modular architecture that is easily expandable in the hands of an advanced user, in such a way that they are free to easily modify the software and make it better suited for their needs.

Since our goal is to make an accessible software we decided to not target any particular platform, landing on the decision that a web app might be the best solution.

Users can save, load, edit, preview melodies, adjust various parameters (e.g. assigned pins) and export their work in JSON format, so that it can be parsed or utilized on the target platforms.

We also provide a compiler that produces an Arduino Sketch, for the ease of test and as a proof of concept.

The paper is structured as follows: the next section presents the choices in materials and methods used, section III presents the design choices, IV the architecture while V the implementations. For results and discussion see the sections VI and VII. At the end of the paper there are the user manual, documentation and code.

2 Materials and Methods

2.1 Design Choices

Our project is split in two parts: software and hardware, the latter used for testing.

Regarding the software, we chose to create a web app, therefore our choice of programming languages and technologies was pretty limited. We also decided to make a "static" web app, meaning that there's no server running behind it beside the one hosting the code. All logic is handled on the client, in order to nullify possible hosting costs, and guaranteeing a responsive client.

Regarding the programming language, we picked TypeScript[1], an extension of JavaScript designed by Microsoft that enforces explicit types and helps avoiding common pitfalls of JavaScript. The use of explicit types also contributes to a better documentation of the codebase.

Regarding the User Interface, we adopted React[2], an open-source framework by Facebook made for developing reusable components, and it is well supported, adopted and documented. React does not provide any graphical elements, but allows to describe the contents of an element with a markup language called JSX[3], that looks similar to HTML, but supports evaluation of inline code.

In order to have a consistent UI that integrates well with React, we picked Fluent UI[4] by Microsoft, that provides styles and basic interaction for elements such as text fields or buttons.

To handle the back end logic, meaning the state of the app, we used Redux[5], a library to manage the state of the app by explicitly invoking actions when altering it. This goes hand-in-hand with the atomic design of React components, which in turn allows for a relatively easy way to extend the app functionalities, thus allowing us to reach our goal of a modular software.

Since we decided to develop a web app, we needed a package manager to handle the aforementioned libraries, and we chose Yarn[6], because of its speed and ease of use. Yarn also handles the React default toolchain, a collection of tools that compile, lint and serve the web application and eventually optimize the app for deployment.

The code is hosted on GitHub and we set up a Continuous Integration system with Netlify in order to deploy on a website the latest changes on the master branch as soon as possible.

The software output is initially a JSON file, that can be parsed externally by users to create their own implementation of the actions on their target platform. As a proof of concept, we also developed a simple parser that

creates an Arduino Sketch and is directly integrated in the code.

Regarding the hardware, we used Arduino for testing. Because of its simplicity and widespread use it seemed the right candidate to verify the correct behaviour of our code. Other more complex devices should presumably work the same, since what Arduino offers is pretty barebone.

Our physical testbench consisted of a series of monochromatic LED, an RGB LED, a piezoelectric speaker and a servo motor. To test other various configurations, we also used a digital simulation on Thinkercad.com.

2.2 Architecture

The software is a single page web app, composed of many modules, and most of them are atomic, which means that they have no dependency on other modules to perform their function. This way, the risk of a failing module compromising the whole app is much lower. A module should often do one thing and one thing only.

The modules called *components* are reusable elements that most of the time display something in the web page. They include JSX (or in our case, TSX, since we're using TypeScript) code that renders the HTML content, and the logic of that component (e.g. in the case of a button, handling the click on it).

There are other modules that handle other features such as parsing the output, saving and loading files, or playing sounds.

The communication between atomic components is handled by a central authority called *store* (handled by Redux), to which each module can subscribe to, and it is a JavaScript object containing the state of the app. Any *mutation* to the *store* has to be explicitly defined, and previous states are saved, therefore allowing for a future implementation of undo/redo functionalities.

Regarding the structure of the data that represents an exportable action, it consists of an array of sequential items representing notes, an array of different actuators (LEDs, motors), each of them containing various informations such as the assigned pins and their actions, and a collection of various data, such as how many millisecond a frame lasts or the name of the action.

2.3 Implementation

Regarding the implementation, let's start with the target of our project, which is an Arduino UNO. One of the limits of the platform is the storage space on board, which is only 32KB. Therefore, our exported code has to

contain all the necessary information to be able to play sound and perform actions without occupying too much space.

One of the first problems is how to handle time, which can be done in two manners: one is absolute, which consists of making the event happen at the moment t, the other is to discretize time into regular intervals. While the first provides a fine-grained control, it presents issues regarding storage space on the target platform (unsigned long is 4 bytes). The second one is easier to implement, since it's comparable to an array in which the absence of a note is a null value. Also, since it's very probable that a user will want to add a sequence of different notes, the second solution is, space wise, more convenient. In fact, in a melody with no pauses, by using the second method we only use

$$[2B(Frequency) + 1B(Duration) + 1B(Volume) + 1B(Type)] * n$$

while with the first one we use

$$[4B(Start) + 4B(Duration) + 1B(Duration) + 1B(Volume) + 1B(Type)] * n$$

using 6 bytes more per note, where n is the number of notes.

We then structured the code in the app accordingly, by having an array of objects representing notes (*SoundFrames*) that can be also *null* (meaning that no note will be played in that instant). Each note lasts for a minimum of milliseconds, which is the same for every note and it's defined globally. A *SoundFrame* is an object containing frequency (pitch), volume (loudness), the waveform (timbre, that is the difference between the sound of two instruments), and also a text reference to the octave(starting from 0) and the note (A-G).

Moving to the actuators, since the frequency of their updates is likely to be less frequent, we opted to represent the actions of a single actuator as an array of objects, each containing the indexes of the frames in which the respective action starts and ends (e.g. a LED turns on at 2 and turns off at 5), along with the values representing the state of that device (e.g. speed of a motor, the intensity of a LED, the angle of a servo motor...).

All the single actuators belong to an array.

Since different actuators have different needs, we decided to have an easily extensible way to add more of them, in the form of a JSON-style object (contained in *utils/actuatorModels.ts*). Users can add their own actuators with a straight forward syntax, and control their state with variables that can be numbers, booleans or colors.

The remaining data,contained in the state of the app, is: the length of the file (mostly used for rendering, as leftover empty values aren't processed

in the output), the *resolution* (how long each interval lasts), and the size of an interval in pixels (used for rendering the timeline).

The app logic is splitted in many modules, which can be reused by other modules. In particular, all modules used to render the user interface are contained in the folder *components*, which contains subfolders to separate the sound timeline, the actuator timeline, menus, buttons, fields and other UI elements. For example, the sound editor module (*components/sound/SoundEditor*) handles a bit of the logic pertaining the chosen instrument or edit mode (draw/erase), and then includes the SoundTimeline (*components/sound/SoundTimeline*) and VolumeTimeline (*components/sound/VolumeEditor*) modules in order to draw the sound pitch editor and the volume editor.

Likewise, the actuator list module (*components/actuators/ActuatorList*) takes care of rendering, a timeline editor (*components/actuators/Channel*) for each actuator. Each actuator editor then makes use of as many blocks as needed to represent a change in the state of an actuator (e.g. turning ON a LED at a certain moment) by using the Rect module (*components/actuators/Rect*), that renders a rectangle containing details regarding that action.

Modals (*components/modals/*) are toggleable windows that are used to display interactive content to the user, such as tuning certain values of an actuator, or loading a previously saved file.

In the *utils* folders there are mostly modules that are used for specific functions, such as generating notes starting from one declared octave (*utils/SoundGenerator.ts*), play sounds (*utils/Player.ts* and *utils/Player2.ts*, which makes use of an external library), create a simplified JSON output file (*utils/JSONFilteredOutput.ts)* and managing files (*utils/FileManager.ts*). This last one in particular is obtained by using the Web Storage API, which is well supported by all modern browsers, and offers more space and ease of access compared to the use of cookies. Each file is serialized and stored as a string, with an associated identifier randomly generated. All identifiers are stored in a list that is saved with a constant key. When loading a file, the module first retrieves the list of files, and then loads the user selected one. It also automatically loads the last modified file by default.

Another important category of modules is the compilers' one: in our case, we've implemented just one (*compilers/arduino_v1.ts*) as an example to show how it works. To reduce the amount of strings in the code containing the logic of the compiler, each device (speaker, actuators) has its own JSON-style blueprint file under the folder compilers/blueprints. Each device is treated as an object with four properties that contains a snippet of C++ code with special keywords with a $ prefix that must be replaced with an

actual value by the compiler. In this way, code can be easily edited and reused by the compiler without having access to its logic. The compiler then puts together a script replacing keywords and adding code to handle the various actuators. Specific libraries can be included in the blueprint, such as the ones to control servos and stepper motors. To play sounds, we made us of the `tone()` function.

The produced script is an Arduino Sketch that can be copied, pasted into the Arduino IDE and uploaded to the microprocessor. To handle multiple devices, after we set up the values to "reproduce" as compile-time instanced constants, we make use of `millis()` to execute code at regular intervals in the `loop()` function.

Regarding the backend logic of the app, we use Redux, a library that provides a unified, central authority on the state of the app, called store. Redux is a variant of the Flux pattern, which makes use of the Observer pattern to handle and notify of changes to the state. With Redux, we must explicitly declare and code actions that can change the state: an action is an object containing a type (usually a string) and a set of properties that we want to alter in the state. It is good practice to create a function for each action to generate these objects, as it doesn't require the user to know the exact type. We can retrieve all or only certain parts of the state, and we can apply a change to it by *dispatching* an *action*. When an action is dispatched, it gets sorted by a *reducer*, which is a function whose result is fed to the Redux store, and in which we can specify how and where we do want to apply this change; it's usually realized with a switch statement on the action type. There are usually multiple reducers, one for each category of actions (in our case, there's a reducer to handle changes to the sound, one for the actuator and one for the global parameters).

The programming paradigm we adopted for this project was mostly functional. JavaScript, and therefore TypeScript, supports functional programming and especially since one of the latest iterations (ES6), it's much cleaner than it was before. Likewise React implemented new ways for functional programming[7].

We made use mostly of functional components, functions that can return JSX code that will be rendered as HTML. In this way, the result is predictable and easier to test. A functional component returning null won't render anything, but we can also return a piece of code containing another component, and that one too will be evaluated and its result rendered on screen.

Since a functional component can be called multiple times, usually when there's something to update in it or in its parent, we made use of functions, either written by us or provided by React to store values across UI updates,

as there's no need to save many variables that handle mostly cosmetic parts in the app state managed by Redux.

3 Results

Since we picked an Arduino UNO as a target platform, we first tested on a web simulator (thinkercad.com) during earlier phases of development [Figure 1], then moved on to a physical one especially when testing the sound part. Tests involved synchronizing various LEDs, a servo motor and a piezoelectric speaker, and we did so at the AIRLab. [Figure 2] The final tests were overall successful, as sound, lights, and motors did effectively synchronize their movement.

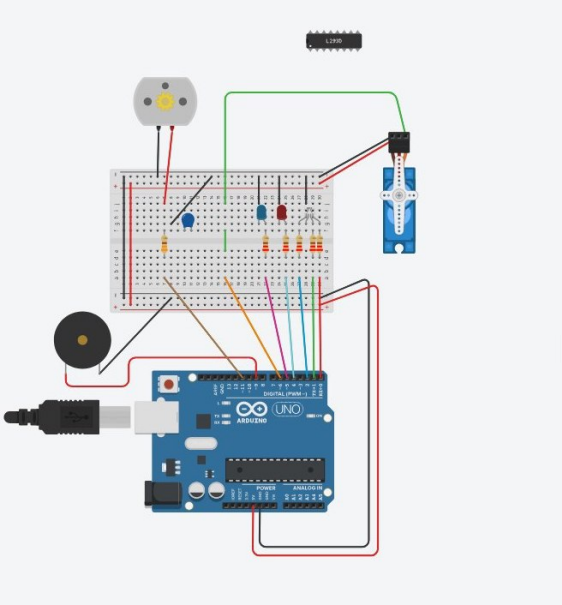

Figure 1: One of the tests made on thinkercad.com

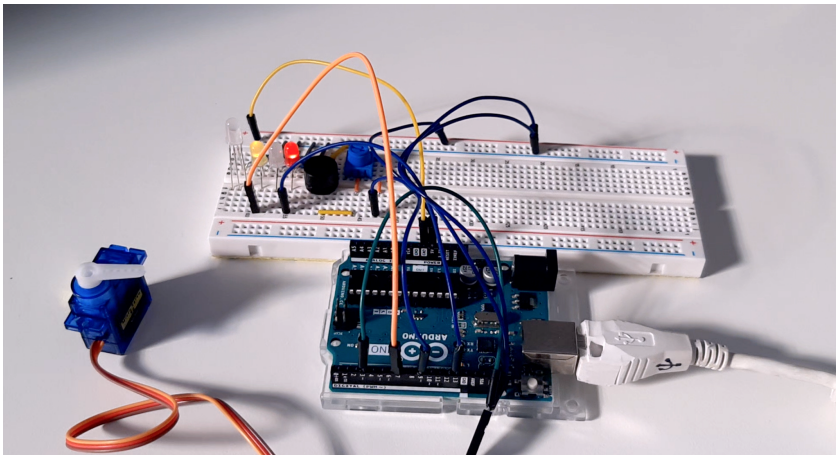

Figure 2: One of the tests made with an Arduino UNO, a yellow led, a red led, a servo motor and a piezoelectric speaker at the AIRlab

We only focused on testing sound with a square wave using the built-in `tone()` function, with no volume regulation. The editor behaves correctly and generates working code in various configurations. The UI is responsive enough and is fairly intuitive. The most critical features are present, such as deleting, renaming, editing components, creating melodies, saving, loading, deleting and exporting files.

We couldn't implement a way to regulate the volume or the waveform type because the `tone()` function doesn't allow to. A possible workaround would presumably consist of handling the sound generation separately in `loop()` by using `analogWrite()`, along with some math manipulation to create a sine wave, for example.

We also didn't manage to accurately test the behaviour of a DC motor in various configurations, such as with an H bridge, as well as with stepper motors, but the compiled code can be generated and in theory should work.

4 Discussion

The results are satisfying enough, as the generated code works without noticeable problems.Since our main goal was to test that automatically generated code can work on Arduino, we didn't focus immediately on making it usable as an external library.

The currently generated code is somewhat easily convertible into an external library by the final user. Once the generated code was working without problems we found it wasn't too hard to convert it to a library, but creat-

ing a compiler that can automatically generate a library requires additional planning

In terms of browser support we focused on Firefox. While we guarantee that this browser works well with the app, other browsers like Chrome and Safari have some issues, caused by how the Web Audio API is implemented on these browsers. In contrast, the new Microsoft Edge, another Webkit-based browser (the same engine Safari is based on) works well with the app.

Playing the composed melody inside the editor is a source of inconsistency across browsers, as generating audio in real time can have some performance issues, especially when it comes to handling it asynchronously.

The editor does its job, but still requires a little more polish, and few features that can improve the user experience will be implemented later.

5 Conclusion

Overall, the major features are present, and the software is a working proof-of-concept.

The feature we implemented are:

- an easy, clean and accessible editor UI with some QoL features, like the movement and extension of Frames;
- users can save, load and download their work as a JSON file and easily switch from one file to another to resume their work;
- supported actuators are: LEDs, DC, Servo and Stepper motors. Users can easily modify the software to add the actuators they need by editing a small number of files, most of them in JSON;
- tools to preview the melody by playing it;

We're working on other features that require additional testing, such as:

- exporting the code as a C++ library for Arduino;
- additional quality-of-life features such as adding frames at a certain moment, or splitting a frame/action;

Additional features that we're looking forward to implement are:

- Add support for different waveforms (sine/triangle/sawtooth) and volume to the exported Arduino Code ;

- In terms of hardware, to support Raspberry Pi, with the possibility of downloading a pre-assembled code or library from the editor in a similar fashion to what currently happens with Arduino;
- Extending support to different major browsers like Chrome and Safari;

Appendices

A1 User Manual

The Web App editor consists of three main sections, the top bar, the sound editor and the actuator editor.

1. *The top bar*

In the top bar [Figure 3] you can find the basics functionalities for creating new files, saving, loading etc. In detail:

- *New*: you can create a new file
- *Save*: you can save your progress and also download it to your computer (as a JSON)
- *Load*: you can load files that you have saved during your session or files that you have previously downloaded to your computer
- *Download*: you can download both the JSON or Arduino Sketch code to your computer.
- *Options*: you can rename the file, give it an author, and edit the Resolution, Length and Frame Size parameters [Figure 4]. The Length is the length in ms of the timeline, the Resolution is the length in ms of a single frame, while the Frame Size is the size in px of a single frame.

For both the Sound and Actuator editors, Frame is the unit of measurement used, and it can be edited in Options. A Frame consists of a number of ms.

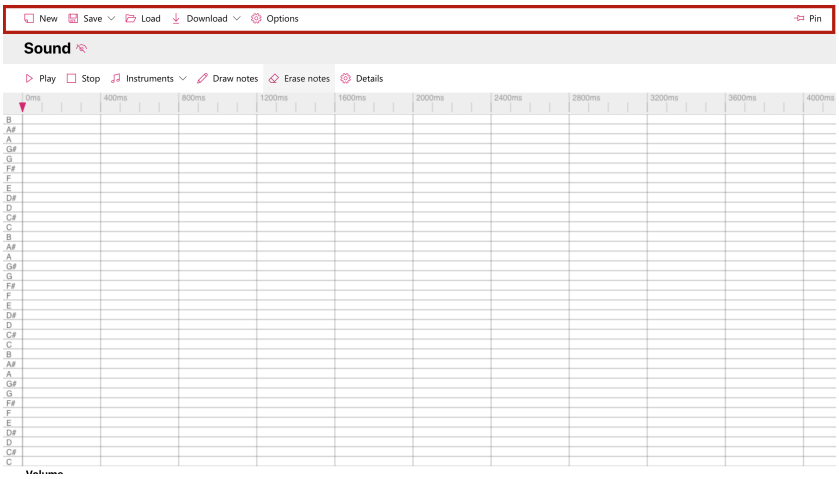

Figure 3: The top bar is higlighted by a red rectangle

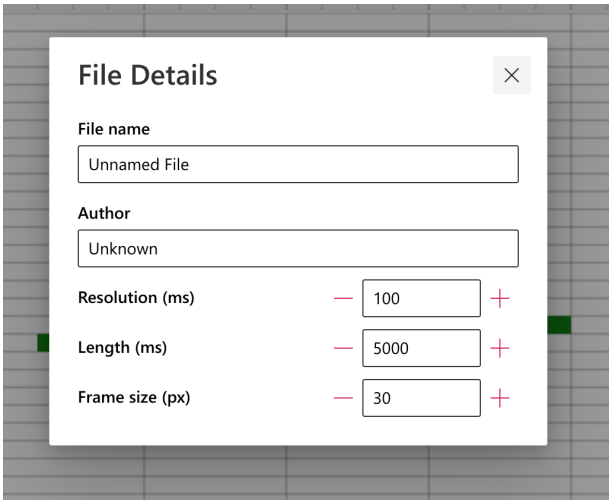

Figure 4: The Options dialog in the top bar

2. *Sound*

To add a note simply click on the "pentagram". The app will reproduce a sample sound of the chosen note. In the sound bar[Figure 5] you can find:

- *Play/Stop*: plays/stops the notes
- *Instruments*: you can choose the type of wave for your notes

- *Erase notes*: to delete a note click on the button and then on the notes you want to delete. When you wish to add a note click on Draw note. You can also delete a note by holding the CTRL key and clicking.

- *Details*: to set the speaker pin assigned on your hardware. The default pin is 9.

Above the pentagram, right under the menu bar, you can find a triangle that show where the melody will start playing. At the bottom of the pentagram you can adjust the volume for each note.

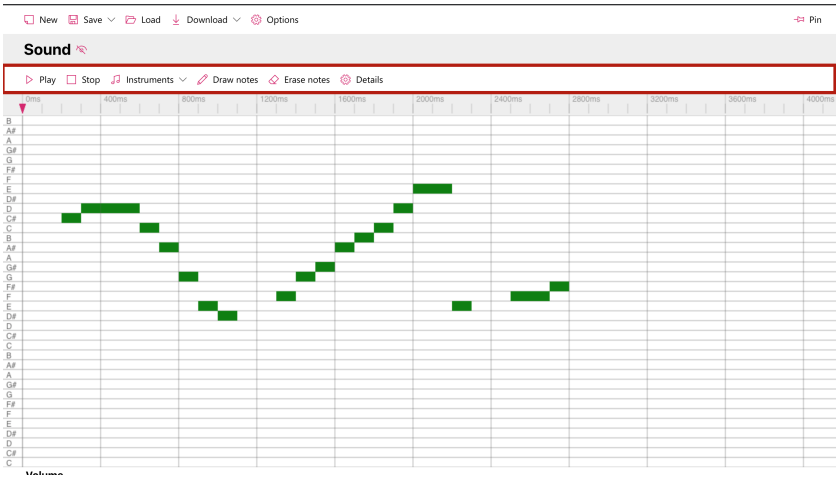

Figure 5: The Sound bar is highlighted by a red rectangle

3. *Actuators*

To add a new Actuator, simply click on New Actuator[Figure 6, 7]. You can now choose the name, the type and pins for power and/or control of your Actuator. When you create a new Actuator a *channel* will appear. In this channel you can perform this actions:

- *New Frame*: add a new frame[Figure 8]. You can choose when to position it on the timeline by choosing its start and end frame.

- *Rename*: rename the current Actuator

- *Delete*: delete the current Actuator

- *Details*: modify the details of the current Actuator

- *Id Information*: at the end of the bar, hovering on the information icon will give you the id of the current Actuator. Each ActuatorChannel has its own id.

The frames in the channel can be moved around by clicking and dragging. Double-click a rectangle/frame to edit its values.

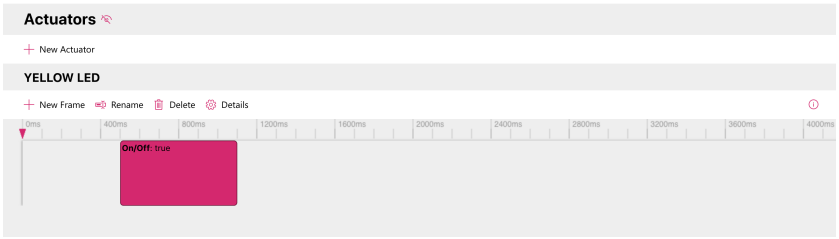

Figure 6: The Actuator editor

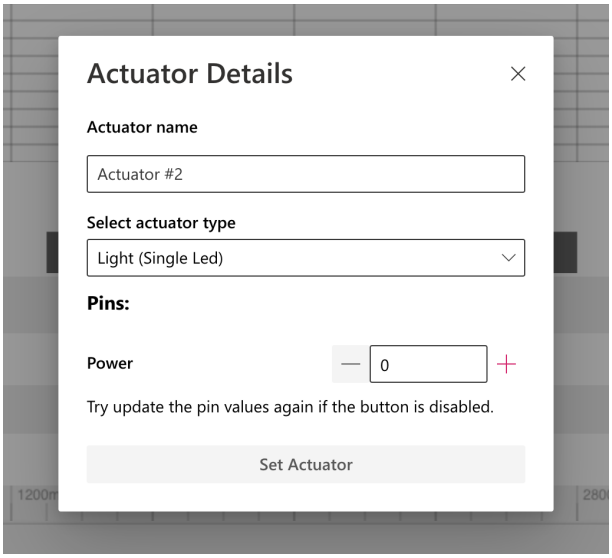

Figure 7: The ActuatorDetails dialog

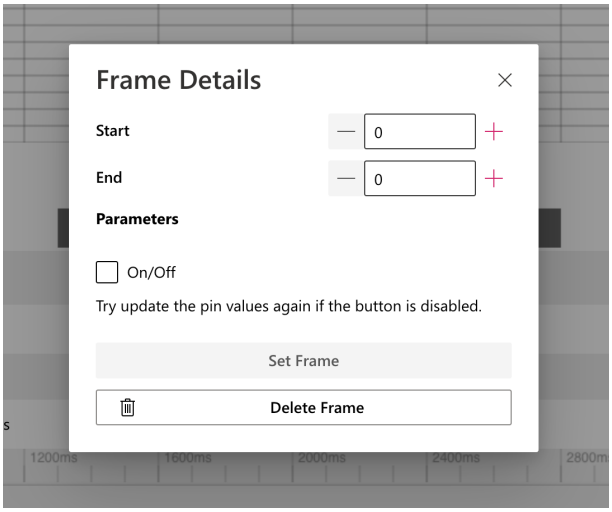

Figure 8: The FrameDetails dialog

A2 SW Documentation and Code

The code and the documentation can be found on GitHub:
Repository: `https://github.com/QUB3X/project-composer`
Documentation: `https://github.andreafranchini.com/project-composer`

References

[1] `https://www.typescriptlang.org`

[2] `https://reactjs.org`

[3] `https://reactjs.org/docs/introducing-jsx.html`

[4] `https://developer.microsoft.com/en-us/fluentui`

[5] `https://redux.js.org`

[6] `https://yarnpkg.com`

[7] `https://reactjs.org/docs/hooks-intro.html`