Smart Home

Christos Christou

Department of Computer Science

May 2019

Department of Computer Science

Smart Home

Christos Christou

Thesis advisor: Mr. Andreas Pitsillides

This thesis was submitted towards partial completion of the requirements for a Bachelor degree of Computer Science, from the department of Computer Science of the University of Cyprus.

May 2019

Acknowledgements

Special thanks to my father, for his time and the engineering knowledge he provided.

This project would not be possible without the constant support from my closest people.

Abstract

This thesis describes the procedure for creating a complete smart home system. It goes through the implementation of the core system, mobile and smartwatch apps, and a real-world example with a smart home. Every process is explained in detail, explaining all of the tradeoffs, the decisions, and the research behind each step.

Table of Contents

Chapter 1 - Introduction

1.1 Background 1

1.2 Goals 3

Chapter 2 - Definitions

2.1 Software 6

2.2 Hardware 8

2.3 Definitions 10

Chapter 3 - Main System

3.1 First steps 15

3.2 Electron core app 18

3.3 Speakers and microphones 20

3.4 Microphones and custom wiring 21

3.5 Google Assistant API 24

3.6 Front-End 27

3.7 Back-End 34

3.8 Keyword detection engine 35

3.9 Custom commands parser 39

3.10 Recap 41

Chapter 4 - Expansions and real-world applications

4.1 Introduction 42

4.2 Smart display housing 43

4.3 House model 45

4.4 Components 48

4.5 Mobile apps 50

4.6 Smartwatch app 53

4.7 Communications 57

Chapter 5 - Conclusions

5.1 Evaluation 58

5.2 Future 60

Chapter 1 - Introduction

1.1	Background	1
1.2	Goals	3

1.1 Background

We live in an era of technology. The free flow of knowledge and opinions, the communication, the comfort and the accessibility to everyday conveniences and goods, are closely related to the continuous and stable rising of technology. Computer scientist and researchers, have the capability, and as a result, the responsibility of shaping the future and improving everyone's lives.

Among the hottest topics in the technological world for the last few years have been the Internet of Things, Artificial Intelligence and Human-Computer Interaction. Arguably the most profound application of the combination of all of these is the Smart Home. However, what is the Smart Home? It's more of an ideology, rather than an application. It's a complete system, integrated into your home, making your everyday life more comfortable, providing a helping hand whenever you need it. It doesn't depend on appliances or gadgets, you never notice it, unless you need it. It's a natural extension of your everyday life and routine.

Every day coming by, we see breakthroughs coming to light, innovations being born. Artificial Intelligence has come a long way, allowing for the appearance of smart assistants. Advancements in speech recognition and speech synthesis brought a natural way to interact with computers. Embedded systems and micro-computers opened a new world of possibilities in internet-connected devices, both in quality, as in quantity.

As a result of all of the above, the first Smart Home systems started to emerge. Smart speakers, such as the Google Home and the Amazon Echo, made their first appearance in 2017. Following the great success of these products, developers started to research new ways to extend the interfaces and interactions of those systems. This resulted in the very first prototypes of smart displays making their appearance in 2018 and the first commercial products coming to market in 2019.

Being inspired from all the research on smart, interactive systems, and understanding the impact that such technology could have in people's lives, I decided to take a new approach on Smart Home and develop a new, fully customizable system, open for everyone, that combines all the best current technology, while it also provides a layer of personalization and control.

This thesis describes the full development, from start to finish, of a complete Smart Home system, including an AI assistant, a smart display, connected devices, voice interactions and feedback, mobile and smartwatch companion apps and more.

1.2 Goals

Despite the great challenge emerging from the rapid development of Smart Home systems, my goal was to implement a complete system, robust and expandable, making it competent in every way, to every other existing solution.

To achieve that, and to accomplish a user-first experience, I constructed a set of principles, to act as guidelines throughout the design and the implementation processes.

The principles are described below:

- **Keep the interruptions minimal**

  This goal describes the purpose to keep the system as distraction-free as possible. Ideally, the system should be available whenever you need it and become “invisible” when you don’t. This behavior provides a natural character to the system, making it available and helpful, but not distractive or annoying.

  This goal is being accomplished, by leveraging several mechanisms to prevent the system from being distractive. For example, when the system needs to ask you about something, remind you, or warn you, I set a time limit, to make sure that it won’t ask you the same question twice in a set period of time.

- **Provide a multitude of interactions**

  A system can never be designed for everyone because every person has different needs, capabilities, and tastes. My goal is to make this system more accessible and bring it closer to the users. I implemented a multitude of ways to interact, provide input and receive feedback from the

system, including voice input and feedback, touch controls, graphical output, mobile, and smartwatch apps.

Expanding the Human-Computer Interactions of the system makes all the interactions more natural and fluid for everyone, but the importance of this goal extends way beyond that. It also opens the doors to all people with disabilities. For example, a blind person can fully interact with the system using his / her voice and enjoy some music, get information about the weather, check out the latest news, or get notified when it starts to rain, to allow the system to close the windows automatically.

- **Completeness and robustness of the system**

  Sometimes it's better to have a complete system with fewer features, rather than an incomplete one with many features. A system, to be complete, needs to implement a coherent set of features, that satisfy a specific purpose. Ultimately, a complete system doesn't need any external systems to complete its specified tasks. Robustness, on the other hand, describes the ability of a system to maintain a fully functional state, under any circumstances.

  Although achieving completeness is not an easy task, I managed to do that with the implementation of a broad variety of tools, interactions, and procedures, such as mobile apps, notifications, and interfaces. To maintain the robustness of the system, I build the system from the ground up, with significant control over the errors, agile handling of edge cases and a self-untangling communications system.

- **Control and customizability**

  As I mentioned before, a system cannot be designed for everyone. For that reason, I decided to include a complete set of features while allowing developers to build on top of them and expand the functionality in various ways. I also ensured the customizability of the system, primarily at the user level, with different voices, names, and other preferences, but also at

the developer level, with custom commands, User Interface options and more.

To ensure expandability and customizability, I built the system with total control over all of the components, procedures, and sub-systems. This is due to the elimination of “black boxes” and the implementation of programmatic and user interfaces.

- **Serious attention on privacy**

  As technology progresses and gets deeply integrated into our lives, and with the market continuously growing, more and more concerns are coming to light every day. One of the greatest fears of the latest technological advancements is the loss of privacy. Unfortunately, leaks and scandals have proven most of these concerns legitimate.

  As a computer scientist, trying to create tomorrow’s technology and shape the principles at its foundations, I feel responsible for providing solutions without sacrificing a fundamental human right such as privacy. Achieving this, can be tremendously more complicated than it sounds. Firstly, this means that I cannot rely on closed systems on which I don’t have the full control, and secondly, I have to make my own systems as transparent as possible, in order to be able to guarantee for their privacy respect.

  For this Smart Home system, I only used open source software, and implemented my system along with it, maintaining absolute control of the data. On top of that, I decided not to rely on cloud speech-to-text services and provide the real-time keyword recognition pipeline offline, utilizing the capabilities of the local device.

Chapter 2 - Definitions

2.1	Software	6
2.2	Hardware	8
2.3	Definitions	10

2.1 Software

InVision Studio

InVision Studio is an advanced prototyping and screen design platform. It allows for the design of the UI / UX, interactions, and animations of an application.

Keyshape

Keyshape is a tool for creating complex animations. It uses Scalable Vector Graphics and provides a versatility of export options for the web.

Draw.io

Draw.io is a web-application that allows for the creation of complex two-dimensional diagrams. It provides a variety of shapes and tools for creation and customization.

Adobe Photoshop CC

Adobe Photoshop is the world-leading software for raster graphics creation and editing. It is the industry standard for photo editing, digital art, and more.

Blender

Blender is a free, open-source 3D graphics suite. It provides a great variety of tools for 3D modeling, texturing, material creation, rendering, and more.

Through the many years of its lifespan, it proved to be one of the best tools for 3D content creation, and it is used in a multitude of applications, such as 3D Artwork, Game Development, and CGI.

2.2 Hardware

Raspberry Pi 3B

The Raspberry Pi is a series of small single-board computers developed by the Raspberry Pi Foundation. They are the most popular SBC on the market and they are used in education, home automation and industrial automation. [3, 5]

Sunfounder 10.1” Touchscreen Display

A 10.1” capacitive touchscreen display, designed for the Raspberry Pi. Contains a display driver board, along with a mount for the Raspberry Pi.

ReSpeaker 4-Mic Array

A quad channel microphone, designed for smart home systems.

DRV8825

One of the most powerful stepper motor driver boards. It regulates power and provides an electronic interface for interacting with the motor.[2]

NEMA17 Stepper Motor

A precise, robust motor with relatively small footprint. It is usually used in 3D Printers.

Stereo Speakers

A set of generic 5V stereo speakers, with a standard 3.5mm input.

DHT22

A small temperature and humidity sensor, with high accuracy and a digital interface.[1]

APDS 9301 Ambient Light Sensor

An I2C-compatible luminosity sensor, that measures lux in the environment. It combines an RGB sensor with an Infrared sensor, to provide accurate results. [4]

SMD 5050 LED Strip

A bright, RGB LED Strip that uses a 12V power, and three channels for color data.

IRL N-Channel MOSFET

A transistor with variable conductivity, that allows for controlling the current that passes through a connection.

2.3 Definitions

Porcupine

Porcupine is an open source keyword-detection engine, created by Picovoice. It analyzes audio input in real-time and tries to detect a specified keyword.

SBC - Single Board Computer

A single-board computer (SBC) is a complete computer built on a single circuit board, with one or more microprocessors, memory, input/output (I/O) and other features required of a functional computer.[5]

Node.js

Node.js is an open-source, cross-platform JavaScript run-time environment that executes JavaScript code outside of a browser. It's used in server-side implementations and desktop applications.[5]

Electron

Electron is an open-source framework developed and maintained by GitHub. It allows developers to build cross-platform graphical desktop applications using web technologies. It uses Node.js for the backend and an embedded Chromium build for the frontend.[5]

Google Assistant

Google Assistant is an artificial intelligence system developed by Google in 2016. It is primarily available on mobile and smart home devices.[5]

Google Assistant SDK

The Google Assistant SDK is an open source software development kit, created by Google, to let developers embed the Google Assistant in their projects.[5]

Cross-Platform

In computing, cross-platform software (also multi-platform software or platform-independent software) is computer software that is implemented to run on multiple computing platforms. Cross-platform software may be divided into two types; one that requires individual building or compilation for each platform that it supports, and another that can be directly run on any platform without any special preparation.[5]

Software Development Kit

A software development kit (SDK or devkit) is typically a set of software development tools that allows the creation of applications for a certain software package, software framework, hardware platform, computer system, video-game console, operating system, or similar development platform. To enrich applications with advanced functionalities, advertisements, push notifications, and more, app developers implement specific software development kits to help and enrich development.[5]

Operating System

An operating system (OS) is system software that manages computer hardware and software resources and provides common services for computer programs.[5]

Node Modules

A node module is a package, in the form of an archive containing computer programs and additional metadata needed by package managers. While the archive file format itself may be unchanged, package formats bear additional metadata, such as a manifest file or certain directory layouts. Node modules may contain either source code or executable files.[5]

Package Manager

A package manager or package-management system is a collection of software tools that automates the process of installing, upgrading,

configuring, and removing computer programs for a computer's operating system in a consistent manner.[5]

NPM

npm (originally short for Node Package Manager) is a package manager for the JavaScript programming language. It is the default package manager for the JavaScript runtime environment Node.js. It consists of a command line client, also called npm, and an online database of public and paid-for private packages, called the npm registry. The registry is accessed via the client, and the available packages can be browsed and searched via the npm website. The package manager and the registry are managed by npm, Inc.[5]

SSH

Secure Shell (SSH) is a cryptographic network protocol for operating network services securely over an unsecured network. Typical applications include remote command-line login and remote command execution, but any network service can be secured with SSH. SSH provides a secure channel in a client–server architecture, connecting an SSH client application with an SSH server.[5]

I²C

I²C (Inter-Integrated Circuit), pronounced I-squared-C, is a synchronous, multi-master, multi-slave, packet switched, single-ended, serial computer bus invented in 1982 by Philips Semiconductor (now NXP Semiconductors). It is widely used for attaching lower-speed peripheral ICs to processors and micro-controllers in short-distance, intra-board communication.[5]

GPIO

A general-purpose input/output (GPIO) is an uncommitted digital signal pin on an integrated circuit or electronic circuit board whose behavior—including whether it acts as input or output—is controllable by the user at run time. GPIOs have no predefined purpose and are unused by default. If used, the

purpose and behavior of a GPIO is defined and implemented by the designer of higher assembly-level circuitry: the circuit board designer in the case of integrated circuit GPIOs, or system integrator in the case of board-level GPIOs.[5]

HAT - Hardware Attached on Top

A HAT, or Hardware Attached on Top, is an add-on board, usually for Arduino or Raspberry Pi systems, that conforms to a specific set of rules which make the set-up process easier. A significant feature of HATs is the inclusion of a system that allows the Arduino or Raspberry Pi to identify the connected HAT and automatically configure the GPIOs and drivers for the board.[3, 5]

SVG

Scalable Vector Graphics (SVG) is an XML-based vector image format for two-dimensional graphics with support for interactivity and animation. The SVG specification is an open standard developed by the World Wide Web Consortium (W3C) since 1999.[5]

UI / UX

A user interface (UI) is a point of interaction between a computer and humans; it includes any number of modalities of interaction (such as graphics, sound, position, movement, etc.) where data is transferred between the user and the computer system.[5]

User experience (UX) refers to a person's emotions and attitudes about using a particular product, system or service. It includes the practical, experiential, affective, meaningful, and valuable aspects of human–computer interaction and product ownership. Additionally, it includes a person’s perceptions of system aspects such as utility, ease of use, and efficiency.[5]

ANSI C

ANSI C, ISO C and Standard C refer to the successive standards for the C programming language published by the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO). Historically, the names referred specifically to the original and best-supported version of the standard (known as C89 or C90).[5]

Binding

In computing, a binding from a programming language to a library or operating system service is an application programming interface (API) providing glue code to use that library or service in a given programming language. Binding generally refers to a mapping of one thing to another. In the context of software libraries, bindings are wrapper libraries that bridge two programming languages, so that a library written for one language can be used in another language.[5]

ES6

ECMA-Script (or ES) is a scripting-language specification standardized by Ecma International in ECMA-262 and ISO/IEC 16262. It was created to standardize JavaScript, so as to foster multiple independent implementations. JavaScript has remained the best-known implementation of ECMAScript since the standard was first published.[5]

Chapter 3 - Main System

3.1	First steps	15
3.2	Electron core app	18
3.3	Speakers and microphones	20
3.4	Microphones and custom wiring	21
3.5	Google Assistant API	24
3.6	Front-End	27
3.7	Back-End	34
3.8	Keyword detection engine	35
3.9	Custom commands parser	39
3.10	Recap	41

3.1 First steps

The first steps of the implementation process, include the basic set-up of the hardware, the installation of the operating system and the set-up of the basic configurations.

I started by connecting the 10.1” display to the Raspberry Pi.

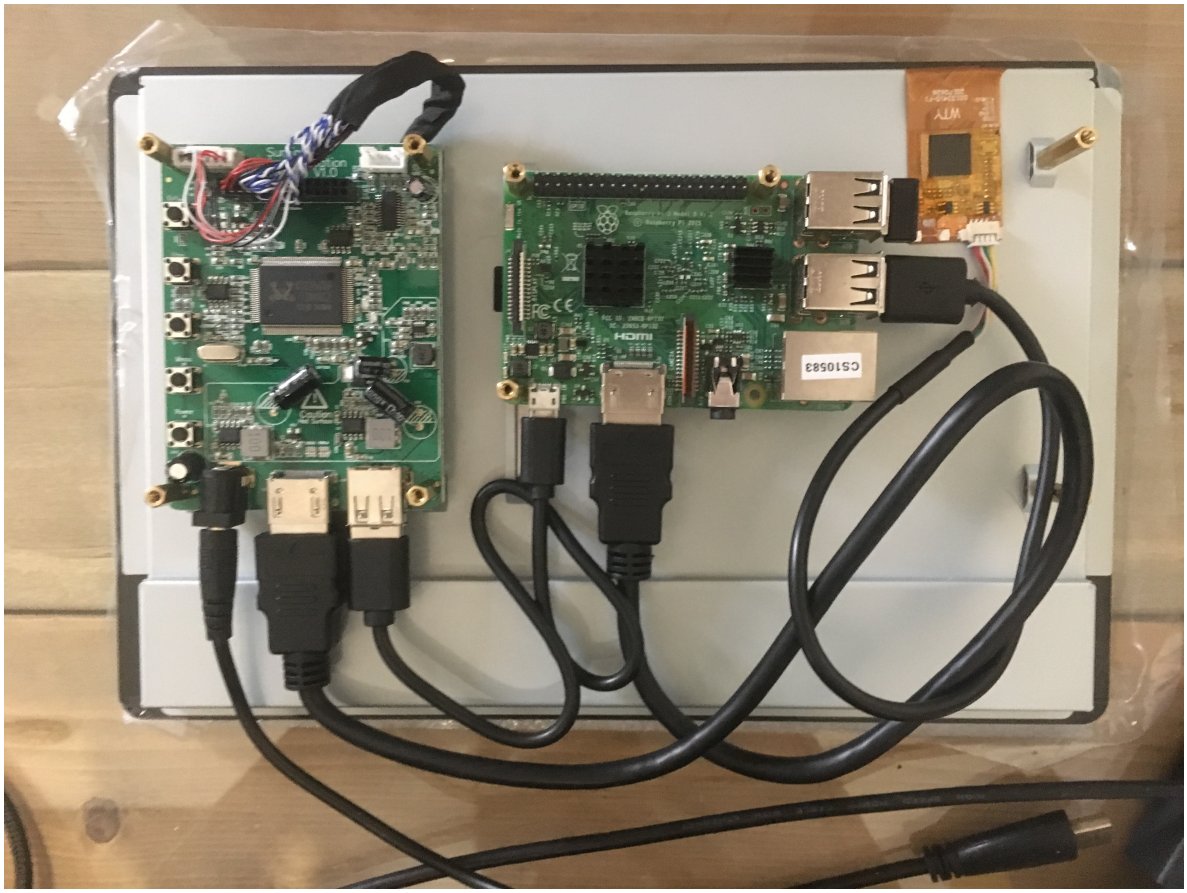

Figure 3.1: Connection of display and Raspberry Pi

Following that, I downloaded a copy of the latest version of Raspbian, an operating system, based on Debian, that is tailored to the Raspberry Pi. I burned the image of Raspbian on a micro SD card, and installed it on the system. The display didn't need any drivers or configuration, as it was pre-programmed to work with a Raspberry Pi. I hooked up a mini wireless keyboard-trackpad combo that I had from previous projects, in order to be able to control the device, and apply some basic configurations.

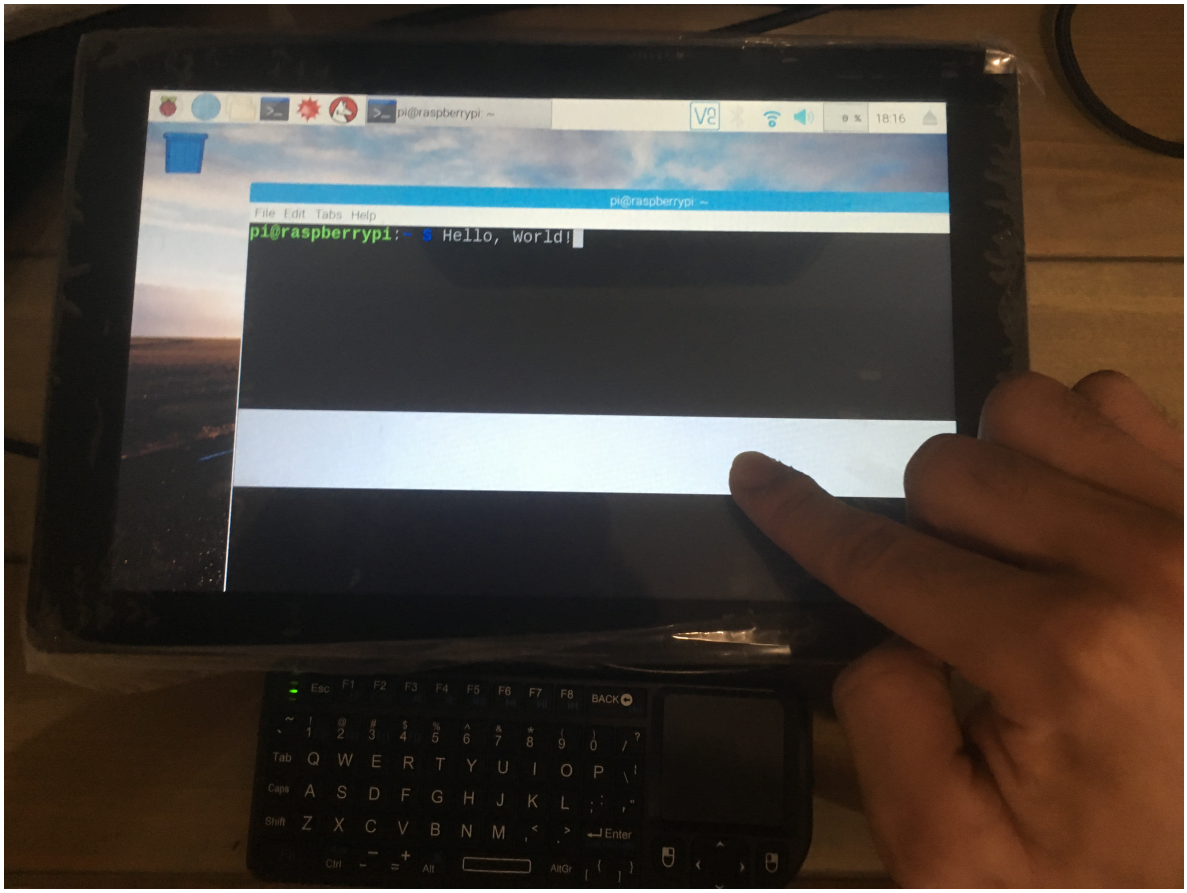

Figure 3.2: First boot

Among the first tasks, I enabled the SSH service on the device, to be able to connect to it remotely and work on it, in a more convenient and productive environment.

3.2 Electron core app

My initial plan for the implementation of the system was to build it in C++ or python. Due to the need of the system to interact with various components at a low level, I naturally leaned towards the native direction of software development.

When I got my hands on the Raspberry Pi and began the implementation, I realized that the native approach was not the best to say at least, for multiple reasons. The most profound problem with this approach was the platform and OS dependency of the system. This dependency not only limits the system's extent to the architecture and operating system of my current set-up but also forces me to rely on the Raspberry Pi for testing and other aspects of the implementation process.

The most obvious solution when it comes to building a cross-platform system is the utilization of web technologies. Having all the experience from several web applications that I built in the past, I decided to use Node.js, a javascript environment for backend applications, through Electron, a framework developed by Github that serves a Node.js application on the desktop.

This new approach gave me significant benefits over the previous one. First and foremost, my system now became platform-agnostic and hardware-agnostic, meaning that I could write the same code and test it across any operating system and any hardware configuration. This feature allows me to distribute my system freely to the users, without worrying about any platform-specific implementations or modifications. The users can then use the system under Windows, Linux or macOS, with their specific displays, speakers and microphones. Moreover, the Electron development stack gives me the ability to implement and test my system in any environment (i.e., my personal computer) and then to be able to move it over to the Raspberry Pi without any modifications. This boosts the implementation and testing phase

tremendously, due to the convenience of working on my personal computer, rather than on the Raspberry Pi.

Apart from the cross-platform related benefits, implementing the core of the system in javascript and Node.js, allows me to utilize the comprehensive toolset of javascript, along with its versatility and error-tolerance, and most importantly, it allows for the use of node modules, through a package manager.

The package manager I used for the management of the project is NPM, since it is one of the most reliable options available, and I have been using it for several years. I created two scripts inside package.json, the configuration file of a Node.js project. The scripts are described below:

- **start:** Starts the execution of the system using the electron command.
- **build:** Builds all the node modules of the project with the help of the electron-rebuild module.

The “build” script was necessary, because NPM is configured to be used only within Node.js projects by default. Electron projects, on the other hand, need some modifications in the building phase. Github provides the electron-rebuild module that rebuilds the entire project with the appropriate configuration for the Electron target.

3.3 Speakers and microphones

After the implementation of the core Electron app, I installed the official node module of the Google Assistant SDK from Google. Before being able to implement an assistant process with voice input and audio feedback, I needed to set-up the microphone and the speakers. For the speakers, I used a cheap stereo speaker set, with a 5V USB power input and a 3.5mm jack audio input. To drive and control the speakers through my system, I used the node-speaker module. For the microphones, I used the ReSpeaker 4-mic array, a quad-channel microphone system designed for the Raspberry Pi, targeted towards home assistant systems with voice input. The four microphones are necessary, to get a clear voice signal from all directions, even at longer distances. In order to properly set-up the microphone array with the Raspberry Pi, I needed to download and install the custom drivers from the ReSpeaker's Github repository. After the driver installation, I installed the node-record-lpcm16 module on my project, to be able to interface and read the input from the microphones from within my system.

3.4 Microphones and custom wiring

Everything went smooth so far, but there was still an issue with this build; the microphone array was designed to be a HAT. The Raspberry Pi features 40 connector pins, for power, interfaces, and GPIO. Everything I am planning to connect to the Raspberry Pi must go through these pins. HAT, or Hardware Attached on Top devices, are designed to provide an easy, push-pull interface with the Raspberry Pi. The tradeoff with this is the fact that they use all 40 pins that the Raspberry Pi provides, for connections and IO. Since I was planning to connect several other devices on my system (sensors, LEDs, motors, and others), I couldn't afford to have a HAT device attached. Fortunately, the ReSpeaker 4-mic array that I am using is an open hardware project, meaning that all of its schematics and diagrams are available online. I looked through the schematics, in order to reverse-engineer the connections of each pin and eliminate any pins that are not necessary for the operation of the microphone array, making them available for other devices. The schematics I used are shown below:

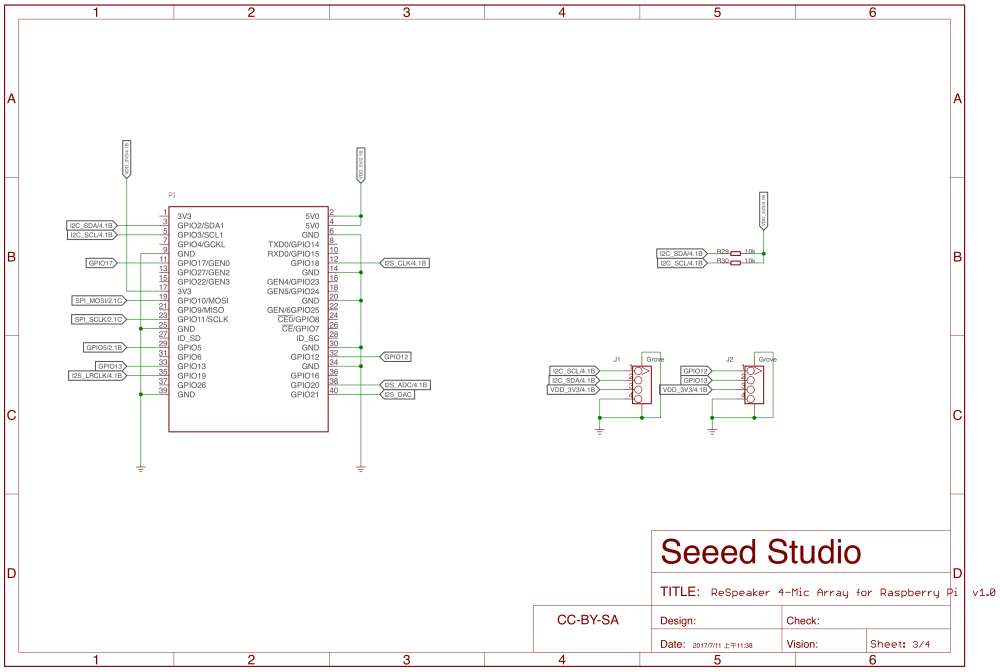

Figure 3.3: Microphone array schematic

After studying the schematic, I eliminated all the unused pins, the duplicate ground and power connections, and the GPIO pins that were used to power the LEDs on top of the microphones. I created a new schematic that describes the simplified connection between the microphone array and the Raspberry Pi. The new schematic is shown below:

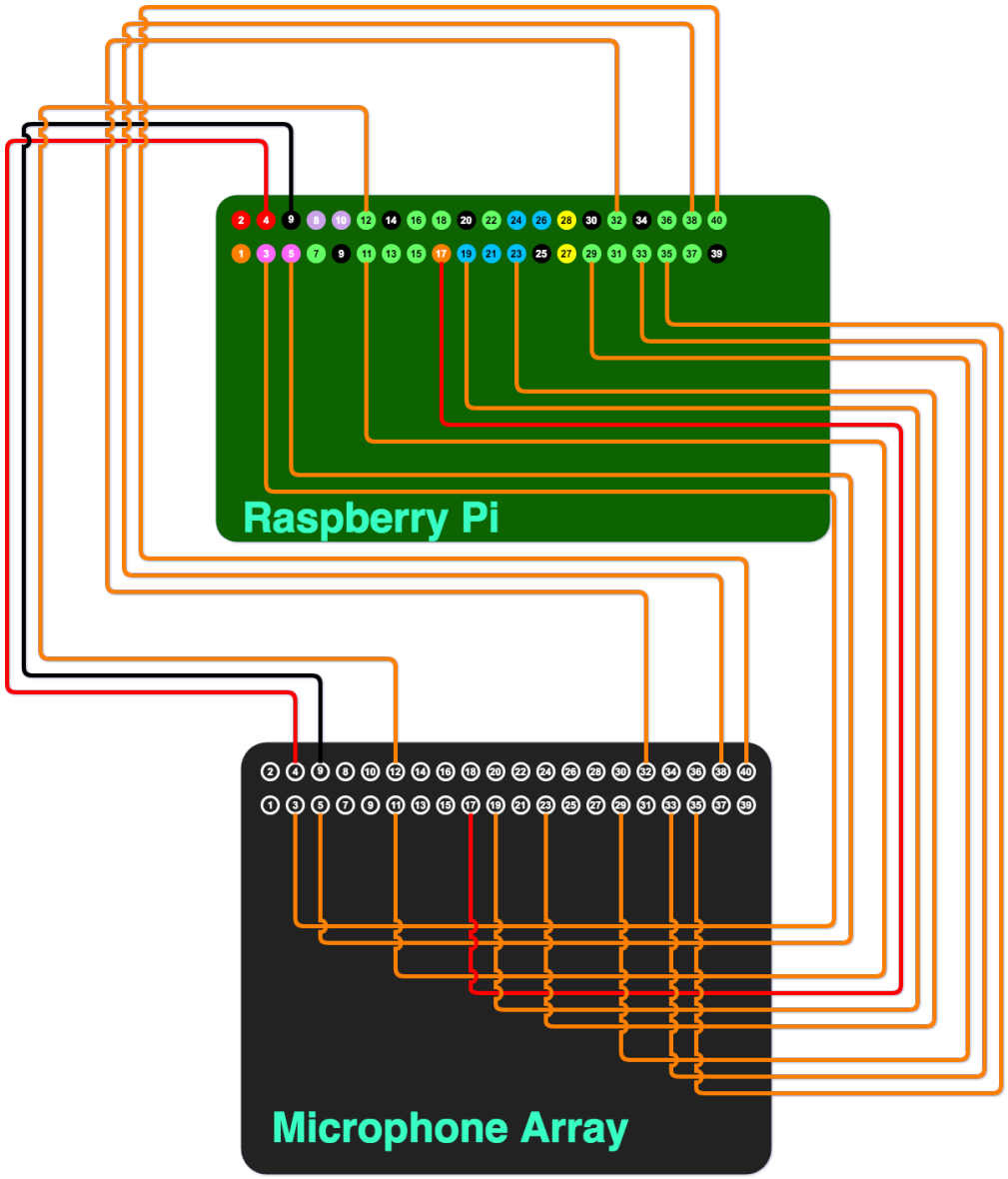

Figure 3.4: New schematic for the microphone array

As we can see from the new schematic, I managed to release 25 pins, out of which 14 can be used for General Purpose Input / Output. Following the

schematic, I applied the connections between the microphones and the Raspberry Pi using jumper wires. After the new wiring was done, I tested the functionality of the microphone using Audacity, and everything worked exactly like previously.

3.5 Google Assistant API

Now that the speakers and the microphones were adequately connected, I was able to continue with the development of the Google Assistant API. I used the google-assistant module, which provides a convenient programmatic interface to the Google Assistant SDK. I constructed a configuration in javascript, containing all the options and parameters that I pass to the Google Assistant instance, such as sample-rate, language, and device id. I followed a procedure provided by Google, to register a new assistant device and generate the credentials and keys for it. Then, I implemented all the event listeners for the Google Assistant API. The Google Assistant events are described below:

- **audio-data:** When the Google Assistant responds with audio data, such as voice feedback.
- **end-of-utterance:** When the user finished a voice command.
- **transcription:** Live transcription of the user's voice command to text.
- **screen-data:** When the Google Assistant responds with basic visual data, in HtML format.
- **response:** The response of the Google Assistant in text format.
- **volume-percent:** When the Google Assistant detects a command that contains percentage data.
- **device-action:** Event used for device actions, a feature of Google Assistant to add and interact with different devices.
- **ended:** When a command-response cycle finishes.
- **error:** In case the Google Assistant detects an error during a conversation.

The implementation of the event handlers for all the events described above acted as a dissection of the Google Assistant pipeline. This structure allowed me to take full control over Google Assistant's behavior, altering the functionality, embedding new parts, or even remove unnecessary procedures for the custom needs of my system.

Below you can see a diagram that I made, showing the normal Google Assistant pipeline:

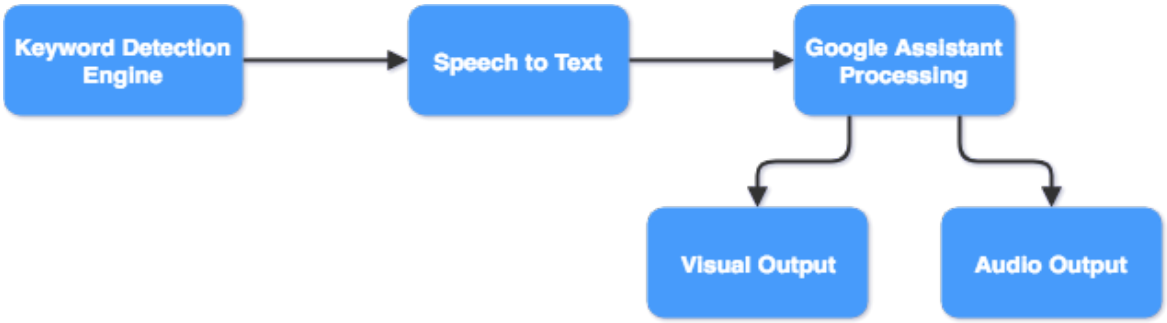

Figure 3.5: Google Assistant pipeline

- **Keyword detection engine:** The keyword detection engine continuously records the user (even when the user is not aware of it) and sends the data to Google's cloud backend, to check if the user said a predefined keyword (Ok, Google).
- **Speech to Text:** If the keyword is detected during the previous step, a conversation starts, and the user can say a command. The command is translated to text in real-time, using Google's speech-to-text AI procedures.
- **Google Assistant Processing:** After the command is translated to text, it is sent over to the Google Assistant backend, in order to be processed by pre-trained AI agents, to find the user's intent and to provide an answer.
- **Output (visual and audio):** After an answer is constructed for the command, a speech synthesis model produces the audio feedback. A basic visual output is also constructed into an HTML file and sent back to the user.

Although my system uses some of the Google Assistant's well-polished procedures, I structured my system in a way to be able to maintain full control over the pipeline. This means that, while I can use Google's sophisticated systems, such as speech-to-text and speech synthesis systems, I don't have to rely on them, making them interchangeable at any

time. This was a target from the very beginning, to combine all the greatest technology available, not having to reinvent the wheel, while keeping the full control over the functionality. So, as a result, I designed a custom pipeline that reflects better my goals for this project.

The diagram below shows the custom pipeline I designed, that uses some of the Google Assistant's procedures (blue nodes), while also including my custom procedures and additions to the system (red nodes).

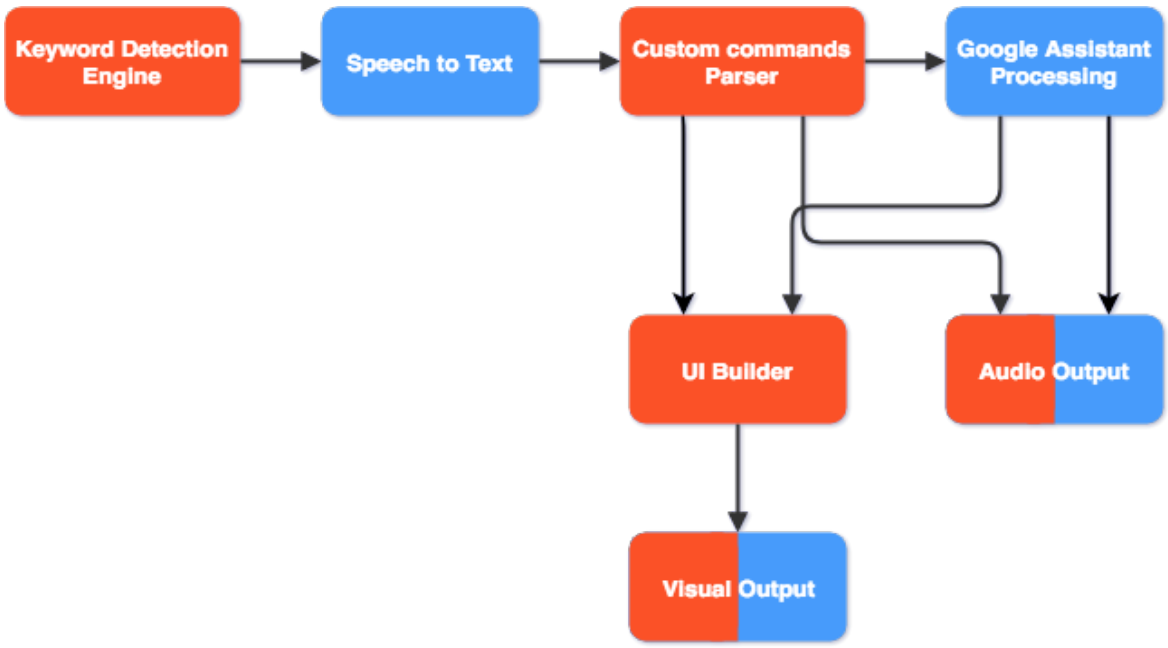

Figure 3.6: System pipeline

All of the custom parts and procedures of the system will be explained in detail in the next chapters.

3.6 Front-End

Analyzing the Google Assistant's functionality, through its extensive documentation, and also with some trial and error, I discovered that all of the responses follow a specific format. I categorized all of the responses into three types, which I list below with a simple explanation:

- **Simple response**: A simple response consists of a title and a small paragraph containing the answer to the question asked.
- **Image response**: The image response contains a title and a paragraph, just like the simple response, but it also includes an image, related to the question's topic.
- **Complex response**: Any response that doesn't follow the layout of the previous categories falls into this category. Usually features a complex layout, containing text, images, icons, and more.

In order to provide a seamless and coherent visual style for my system, I created several prototypes and designed the User Interface and the User Experience of my system for every scenario. The program I used to design the UI / UX of my system is inVision Studio. Below you can see the prototypes I created for every response type:

Figure 3.7: Simple response design

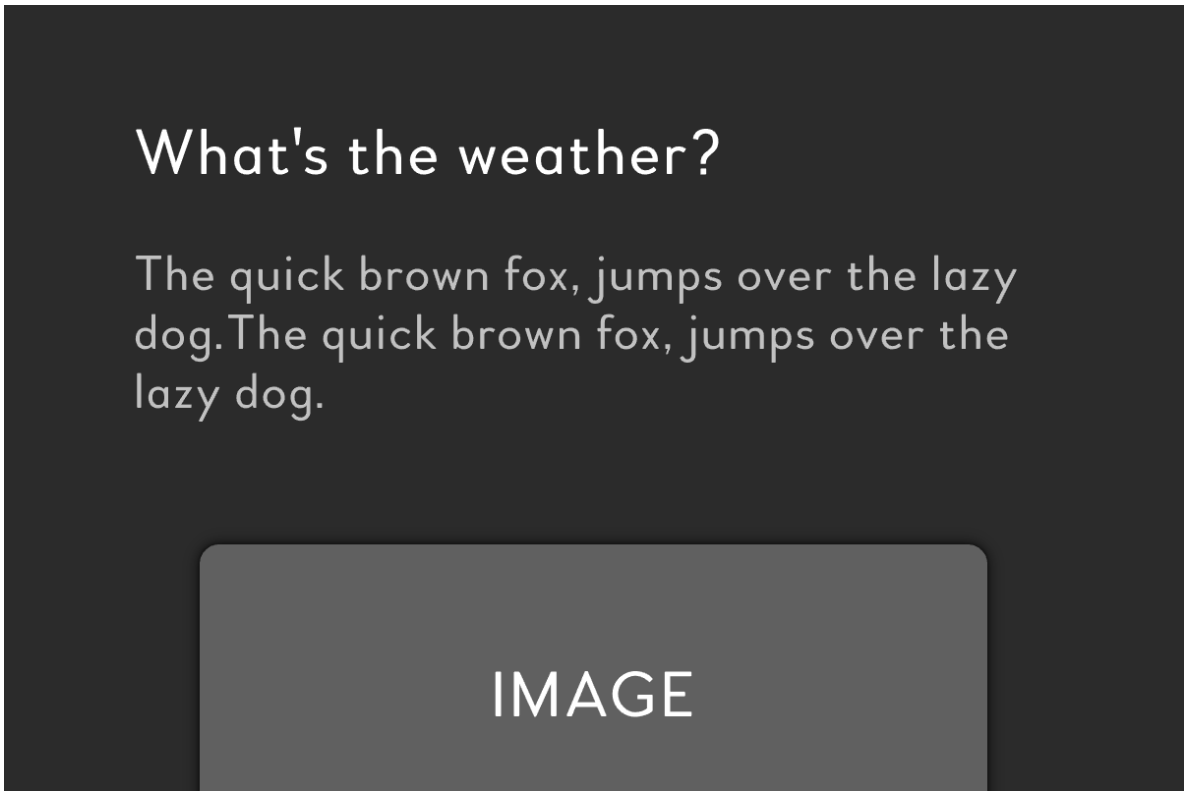

Figure 3.8: Image response design

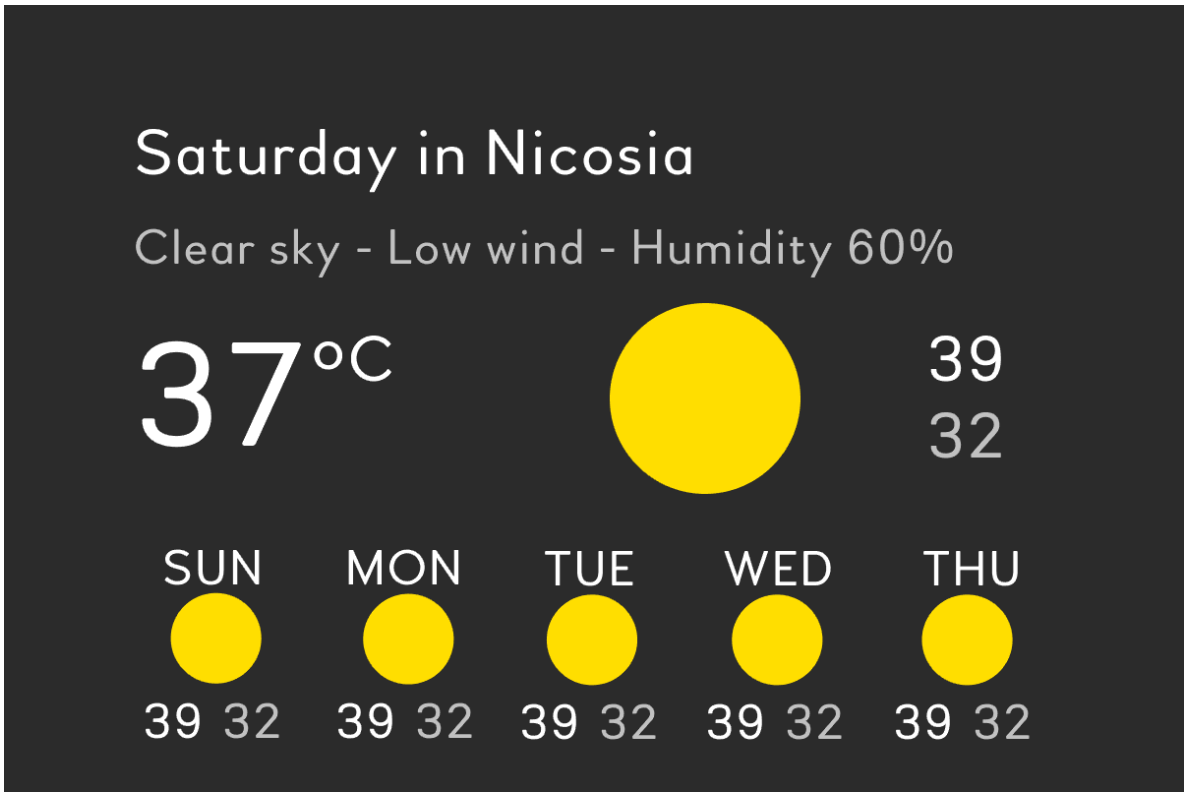

Figure 3.9: Complex response design

Apart from the responses, the system consists of several other screens and interactions. Firstly, I wanted to create a beautiful home-screen. The home-screen is going to be presented to the user by default, including a wallpaper and some useful information. Although this system can run on many platforms and hardware configurations, I designed the home-screen with a wall-mounted solution in mind. Having a smart display mounted on a wall requires a meticulous design of the home screen, which is arguably the most important and the most viewed screen of the system. For this reason, I chose some beautiful, high-resolution wallpapers to act as a painting on the wall. Also, while I wanted to provide some basic information on the home screen, I decided to keep it minimal and clean, including only the time and the date. Adjusting the layout, the fonts, and the colors inside inVision Studio, I constructed the final design. You can see the design for the home-screen below:

Figure 3.10: Home screen design 1

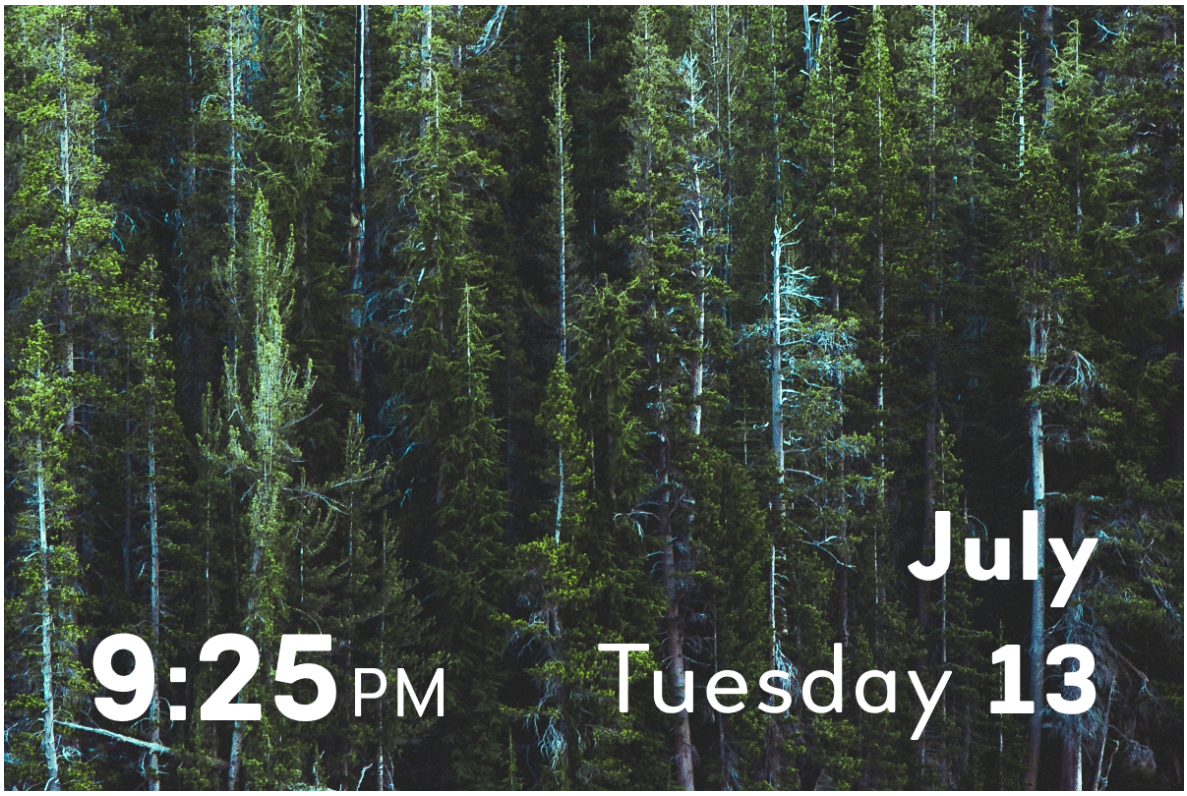

Figure 3.11: Home screen design 2

Figure 3.12: Home screen design 3

Finally, the design process of the main system concluded with the creation of the “talking” indicator. This indicator shows up when the user triggers the system to ask a question. I decided to make the indicator clearly visible, in order to give a proper feedback to the user but to also keep it out of the way, so it doesn’t block the home-screen or the previous response. The design of the indicator is shown below:

Figure 3.13: Indicator design on response

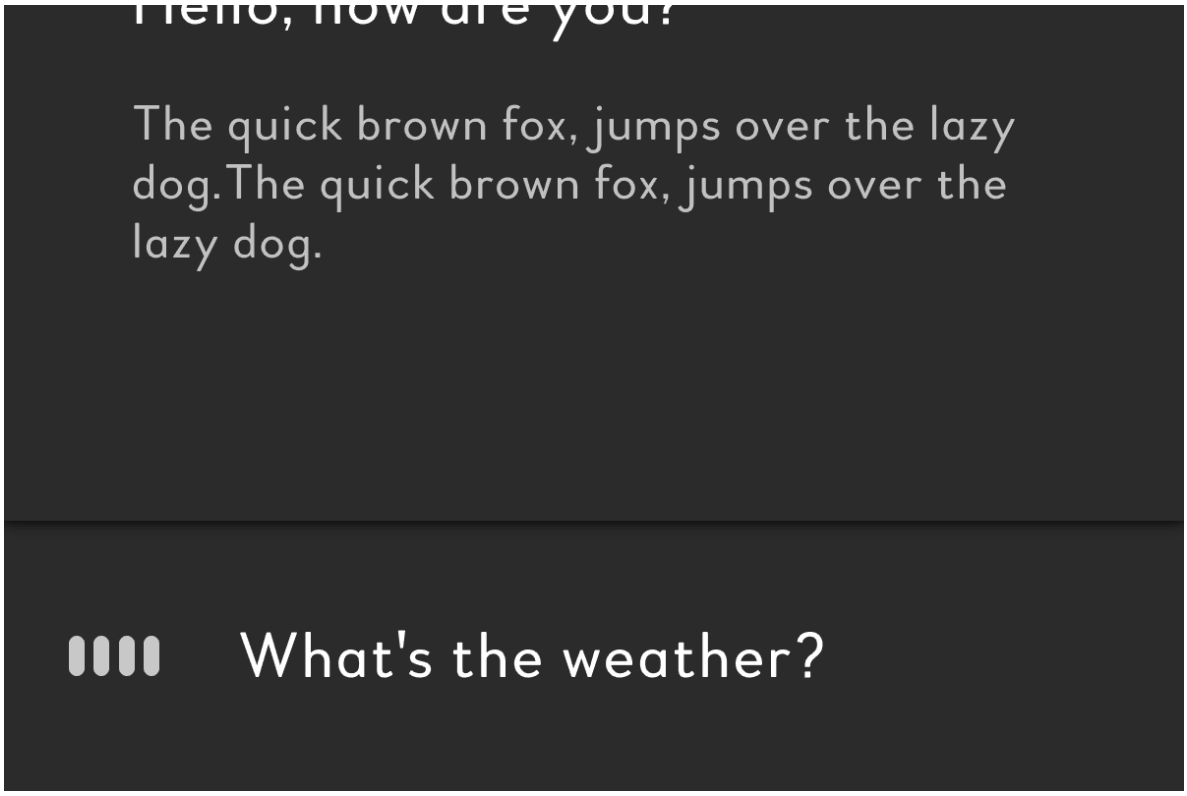

Figure 3.14: Indicator design on home screen

After the design process was done, my next goal was to program the User Interface of my system, according to my prototypes. First of all, I created the indicator animation. The small lines on the left side of the indicator were intended to be animated to simulate a waveform, to give a voice input feedback to the user. I wanted to create the animated component using the SVG format, to be scalable without losing quality, and small in file size. I discovered the Keyshape program, an advanced tool for creating SVG animations, with powerful procedures and versatile export options. After I studied the tool, I managed to create the animated SVG and export it with embedded CSS, containing all of the animations.

Moving back to the Electron app, I developed a front-end structure using HTML, CSS, and Javascript. I moved all of my assets in the project and created the home-screen and the indicator components. Finally, I implemented the date and time functionality in Javascript.

For the multiple types of responses, I developed an output parser, that takes as input the basic HTML response of the Google Assistant, deconstructs it and recreates it with my specified design. The deconstruction of the HTML file is done using HTML Templates. After that, I remove unnecessary content, such as scripts, styles, and unused elements, and I generate the new style dynamically, according to the type of the response. I implemented a set of pre-made styles in CSS, which are then referenced and attached to the response objects accordingly.

3.7 Back-End

After finishing the front-end of my system, I moved back to the back-end of my Electron app, to set-up all of the necessary communications between the two sides. Using Electron's provided communication system, I managed to create the necessary connections between the renderer procedure (front-end) and the main procedure (back-end). This system acts like a typical web-socket implementation, so I developed an API to handle all of the communications. The description of the API is shown below:

- **hotword**: This signal is sent from the main process to the renderer process, to indicate the beginning of a command. The renderer process displays the "talking" indicator, after receiving this signal.
- **end-of-utterance**: The end-of-utterance signal indicates the completion of a command. It is sent to the renderer process, which, upon receiving it, closes the "talking" indicator.
- **transcript**: The transcript message passes the user's spoken words in real-time to the front-end of the application. The renderer process then takes the message and presents it inside the "talking" indicator.
- **response**: This message is sent from the main process to the renderer, containing the basic HTML response from the Google Assistant. This output is then handled from the output parser, inside the renderer process, and after the reconstruction is presented to the screen.

From the API above, the only information that is missing is the audio response. The reason for this is my decision to handle all of the audio interfacings on the back-end. As with any client-server implementation, only the back-end of the system can "talk" natively with the operating system and interface hardware devices. As a result, the audio data of my system must be handled in the main process, which interfaces and passes the data to the speakers.

3.8 Keyword detection engine

In order to be able to interface a system completely using voice input, the “hotword”, or “trigger word” mechanism was invented. To avoid confusion and false positives, regarding the voice input, the hotword mechanism acts as a layer of separation between normal talking and voice commands to the system. Examples of this mechanism can be observed in all major voice-first systems, such as the Google Assistant, Amazon Echo, and Siri, with the hotwords of “Ok, Google”, “Alexa”, and “Hey Siri” respectively.

As you can see, the hotword directly relates to the character and personality of the system and even acts as the name of the virtual assistant. Customizability, was one of my primary goals from the beginning, for this system, so that the system adapts to the user’s needs and not the other way around. Following this goal, it’s clear that the user should be able to customize such an essential aspect of the system, such as the hotword. Unfortunately, big companies don’t offer this layer of control. This is happening mainly for two reasons; the decisions regarding allowances to the user, and the technical limitations, due to the utilization of pre-trained neural networks for the keyword detection process.

Another, even more critical aspect regarding keyword detection engines is privacy. A keyword detection engine is a procedure that given a voice input, tries to detect the appearance of a specific keyword or phrase. In other words, it is the algorithm that detects when the user says the hotword. Users often overlook the direct relation of a keyword detection procedure to privacy. The engine needs to scan all of the vocal data continuously, in order to be able to detect and trigger the system for a command. As a result, smart assistants continuously record everything, even when the user is not aware of it, and send the data to the company’s servers for keyword analysis. Of course, besides the keyword analysis, the same data are exploited for other purposes, such as advertisement and statistics. In other words, a typical virtual assistant always listens, records, and analyzes all of your

conversations collecting personal data. Now, you can see the enormous impact of keyword detection procedures in the user's privacy.

For all the reasons mentioned above, I decided to implement a custom keyword detection solution for my system. I conducted research and narrowed down all the possible routes I could follow for this task. I compiled below a list with all of the viable options:

- **Cloud based speech-to-text**: This option implies the utilization of a cloud-based speech-to-text API. There is a variety of such services provided by Google, Microsoft, Amazon, and more. This is the easiest solution to implement.
- **PocketSphinx**: PocketSphinx is a lightweight, Python-based, offline speech recognition engine.
- **Snowboy**: Snowboy is an offline hotword detection engine, running on multiple platforms. This seems to be the most popular option for custom keyword detection engines.
- **Porcupine**: Porcupine is a small, lightweight, offline keyword detection engine made by Picovoice. This engine takes a different approach for keyword detection, showing impressive results in terms of speed and accuracy.

The cloud-based solution is the easiest one to implement, and with the rapid development of speech-to-text services, provides a simple and very accurate answer to the problem. Despite that, I decided against it mainly because it doesn't resolve the privacy problem, as this method still requires the transportation of the voice data to foreign hands. The second reason that made me discard this option was the price. Although these services are relatively inexpensive, they tie the user to a monthly payment, which is always not welcome.

Removing the cloud option from the list meant that I had to review and evaluate all of the offline solutions. An offline keyword detection engine, runs directly on the device, utilizing the local processing power, without sending any data over the internet, which is precisely what I wanted. Resolving the privacy issue, I generated another one though: performance. Since I decided to run the keyword detection procedure locally on the device, the performance of the whole process is limited to the capabilities and the processing power of the device. A keyword detection engine is usually very power demanding, as it needs to process audio data in real-time continuously. Combining this with the fact that I wanted the system to be able to run on embedded systems, such as the Raspberry Pi, introduced a strong metric for comparison between all of the viable options.

I initiated the selection process by setting up all three engines. The PocketSphinx was the most difficult one to get to work, as it required a lot of configuration, and it relied on many dependencies. After performing the first tests on the three systems, getting similar results, I searched for customization options and libraries that would help me to integrate them with my system. Snowboy was the most convenient out of the three to implement, as there are libraries to use it through a Node.js environment. The problem that I found with this particular option though, is the lack of multiple keyword options and the very expensive plans offered by the company. This led me to eliminate this option and focus on the other two. PocketSphinx although seemed promising at first, because of accuracy issues, but most importantly, because of the inconvenience to install and set-up correctly. Not wanting to pass the hassle of configuration and installation to any users of my system, I was left with the last option: Porcupine. Porcupine, not only was the smallest in size, but it also performed really well. On top of that, it provided the most accurate results and required almost zero configuration and set-up. The project is open source on GitHub, and the base of the system is implemented in ANSI C. Below, you can see a graph, comparing Porcupine's accuracy with PocketSphinx and Snowboy.

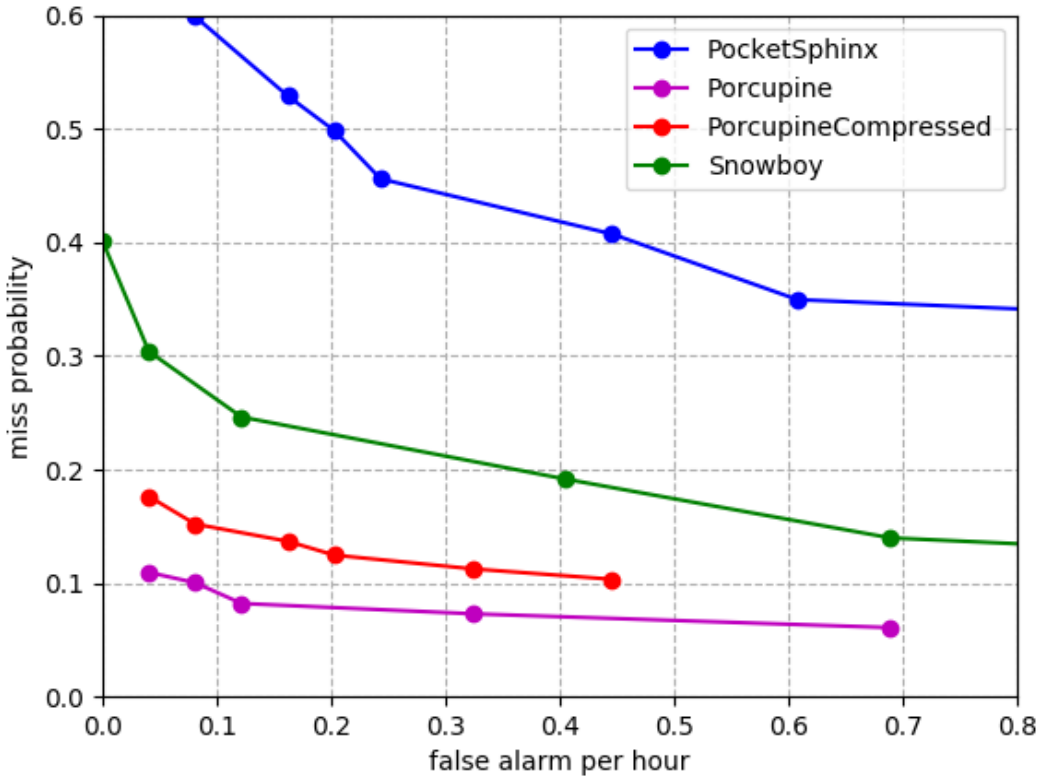

Figure 3.15: Accuracy comparison of keyword detection engines

Although Porcupine provides bindings for a lot of languages and platforms, unfortunately, it doesn't provide a Node.js interface. As a result, I needed to implement the necessary Node.js bindings to be able to call, run, pass data, and get results through Porcupine's C procedures. Starting off, I analyzed the structure and functionality of the engine. After some research on creating Node bindings to run native code through Node.js, I managed to create a class containing all of the necessary functions to interface the Porcupine library. Then, I built the library into a Node binary and wrapped the project into a Node module. After this, I was able to include the module into my project and create a Porcupine class with the proper parameters and finally get the engine to work. I attached the engine to the pipeline of my system, which now became a full virtual assistant with the latest addition.

3.9 Custom commands parser

The last functionality that was left to be implemented on the system was custom commands. The general logic of custom commands is that any question that falls into the custom command category will be handled separately by the system and not by the Google Assistant. If the command doesn't fit into any of the templates for a custom command, is then passed into the Google Assistant's general AI for further processing.

For this mechanism to work, I needed a way to "pause" the functionality of the Google Assistant, pass the command through a command parser and finally accept or decline Google Assistant's response, in favor of the custom command handler. The entire pipeline of the system is implemented using asynchronous programming. Every procedure of the pipeline runs independently and notifies the next procedure to begin, using Javascript events. This is a good practice, as it adds speed and abstraction to the entire pipeline. This structure introduces a problem for the custom command parser though, regarding the pausing mechanism, as injecting a procedure into asynchronous code, is not as easy as calling a function. To manage to "synchronize" my asynchronous pipeline, I made use of Javascript Promises. A promise is an object that signifies the successful or failed completion of an asynchronous procedure in an uncertain time. With the utilization of promises, I managed to block Google Assistant's responses from being propagated, until I check the command through the parser. If the parser finds a match, Google Assistant's responses get rejected, and I proceed with the custom handling of the command. Otherwise, the promises get resolved, and the operation flows normally, accepting Google Assistant's responses.

Apart from the custom procedures that I attached to each custom command, I also wanted an audio and visual response from the system. To keep consistency across the responses, I had to get the Google Assistant to somehow respond to my custom commands, with a controlled response. To achieve that, I leveraged a mechanism, using Google Assistant's "Repeat

after me” command. This command, as the name implies, repeats everything that follows after the call. To inject this mechanism, I further tweaked the flow of the pipeline, to allow me to spawn assistant instances with a set text as the command. After everything was set-up correctly, the custom responses worked, but as a minor issue, Google Assistant attaches the phrase “You said:” before both the visual and the voice feedback. For the visual feedback, I removed the unnecessary part, using the output parser on the front-end. Fixing the audio feedback was more complicated, as it involved a dynamic “cutting” of the audio response, to a specific frame.

3.10 Recap

The core system was completed at this point. In this section, I will summarize the entire functionality of the system, as explained in the previous chapter and describe the pipeline and the procedures involved.

The system is build using the Electron framework. The front-end of the system is responsible for everything related to the User Interface, as well as for all the interactions with the user, such as touch input and visual output. On the front-end also resides the output parser; a procedure responsible for the structure of the visual responses. On the back-end of the system, all the interactions with the hardware take place. On top of that, the assistant procedures, the keyword detection engine, and the custom commands parser also live there.

When the user calls the predefined keyword, the keyword detection engine triggers a google assistant procedure to begin. The user's command is captured by the procedure and sent over to the custom commands parser. After it is parsed, it gets handled either by the parser or by the Google Assistant. A voice response is generated and played, along with a basic visual response, that is sent over to the front-end of the system, to be handled by the output parser and to be presented on the screen.

Chapter 4 - Expansions and real-world applications

4.1	Introduction	42
4.2	Smart display housing	43
4.3	House model	45
4.4	Components	48
4.5	Mobile apps	50
4.6	Smartwatch app	53
4.7	Communications	57

4.1 Introduction

After the completion of the core system, my next goal was to build a real-world solution, demonstrating the various aspects of my system. Specifically, I wanted to demonstrate how a custom solution could be built upon, expand, and work along with the system, in a home environment.

In order to adequately demonstrate this, I designed a set of scenarios in the form of real-world use cases, which take place in a small house model. Moreover, I designed and built a housing structure for the main system, resembling a wall-mounted solution. Finally, the system was expanded, becoming more complete and accessible, with the addition of convenient mobile apps for iOS and Android, along with smartwatch apps.

4.2 Smart display housing

Finishing off the main system, I decided to build a structure to hold it properly, with all of its components. I wanted this structure to resemble a wall-mounted solution of the system, as it would be implemented in a house.

I constructed a 3D design of the housing, which you can see below:

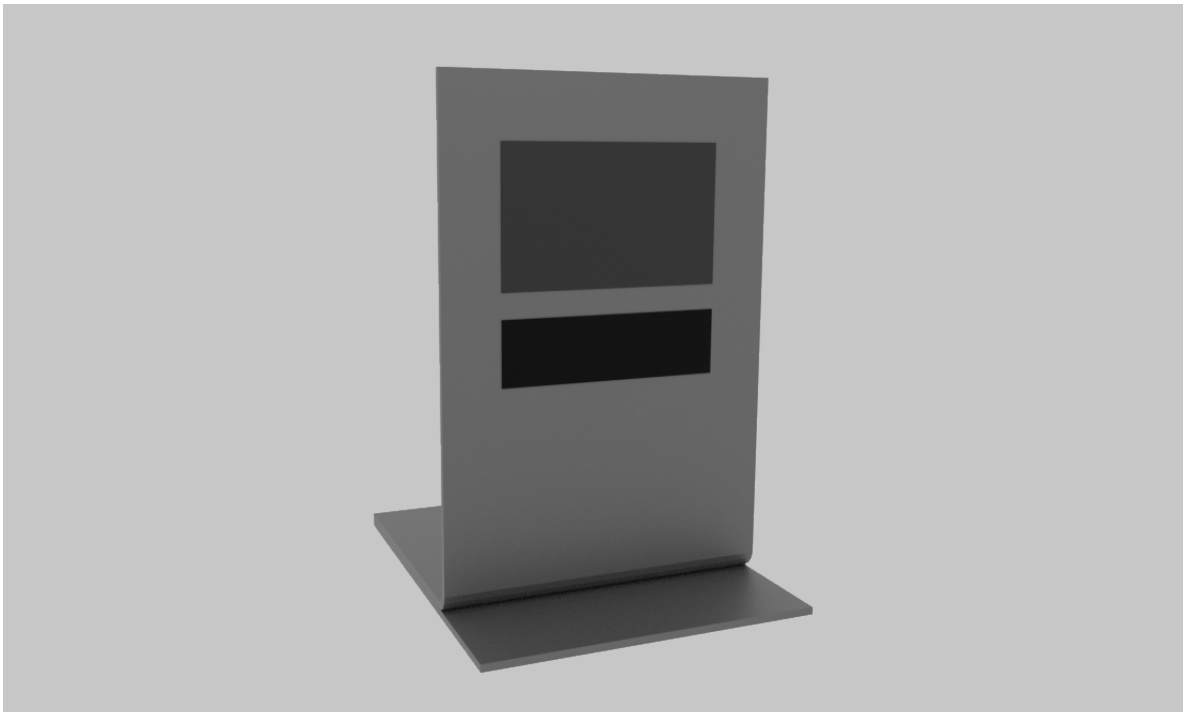

Figure 4.1: 3D design of the smart display housing - front

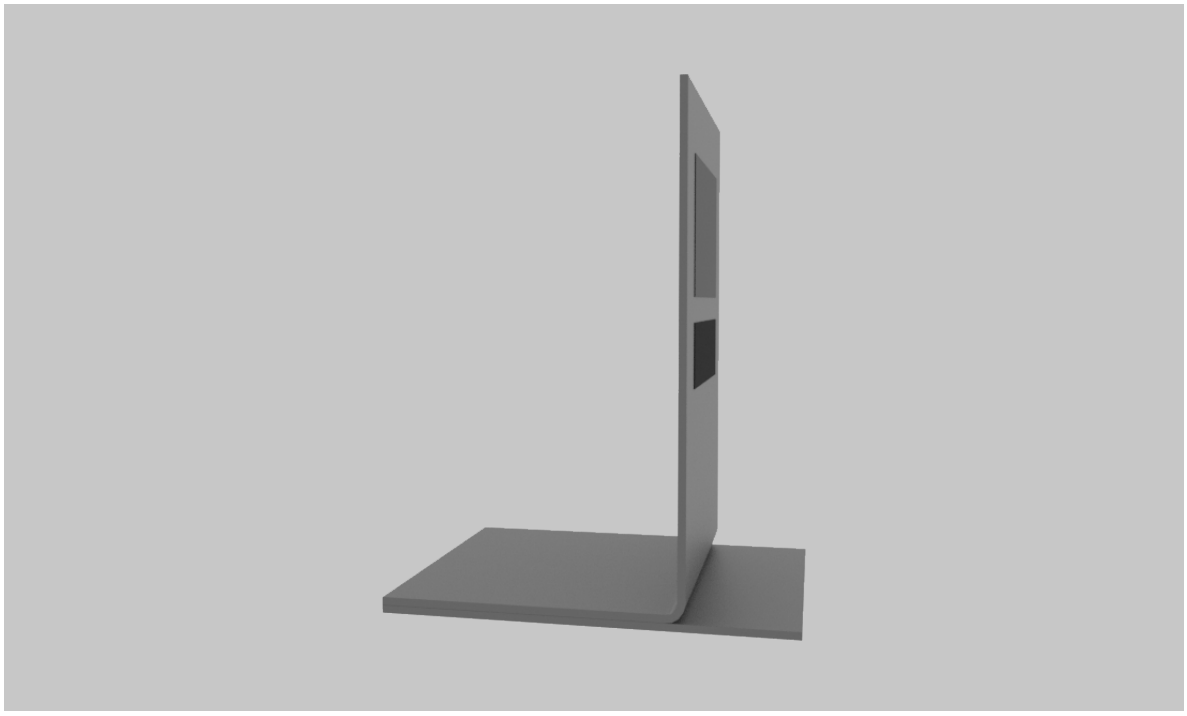

Figure 4.2: 3D design of the smart display housing - side

The design above was then implemented using aluminum and polyamide. The materials were chosen carefully, based on their accessibility, workability, and strength over weight.

The upper black rectangle is the 10.1” display of the system, while the lower horizontal stripe consists of a fine mesh, beneath of which, the stereo speakers are mounted, along with the quad-channel microphone. On the back side of the display, resides the display driver board, the Raspberry Pi, and all the interfaces and connections with the smart home. As a final small detail, four rubber feet were placed at the bottom of the structure, to support its weight and prevent it from scratching the surface beneath.

4.3 House model

As I was planning the real-world scenarios for my system, I realized the necessity for an environment to facilitate them. Having this in mind, I decided to build a small house model to hold and demonstrate all of the use cases of the system in a real home, as well as some new ideas, regarding comfort, user experience, and energy saving.

Starting off, I designed a 3D model for the house, using Blender. I wanted to achieve a balance between small and comfortable. On top of that, I wanted to create an architecturally pleasing model, that resembles a real house. My idea was to build the ground floor, of a house, consisting of a living room, a kitchen, and a bathroom. The placement of the rooms, as well as the dimensions, were chosen in a mixed procedure of visual design and mathematical coherence. Below, you can see the final 3D design of the house model:

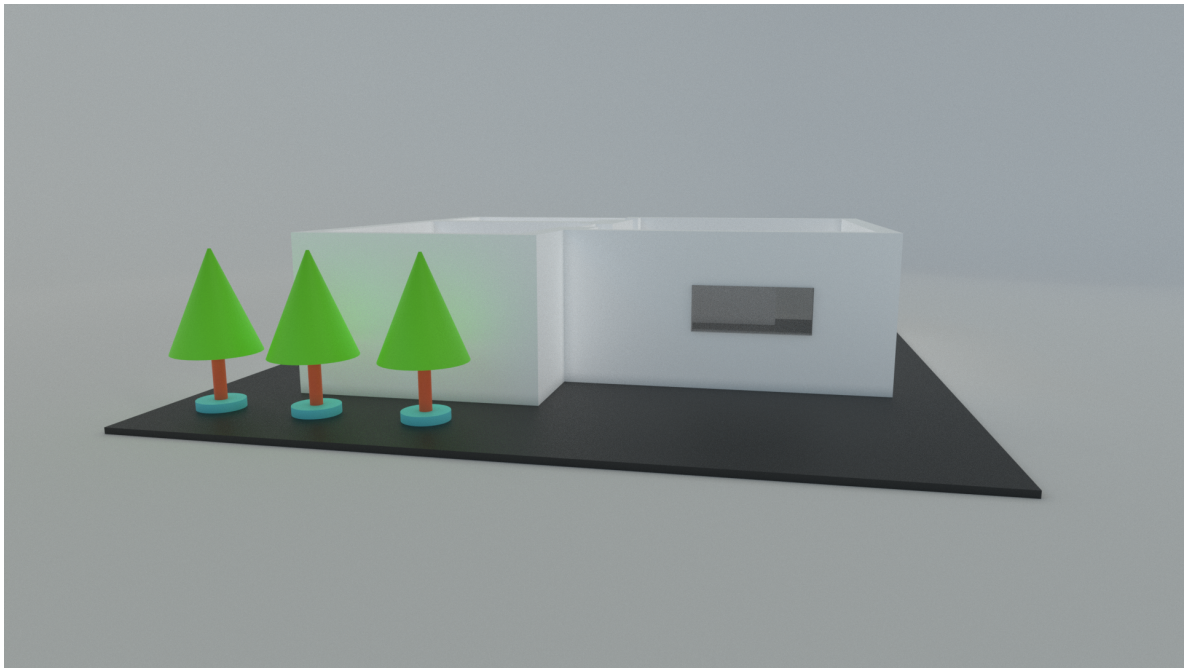

Figure 4.3: 3D design of the smart home model 1

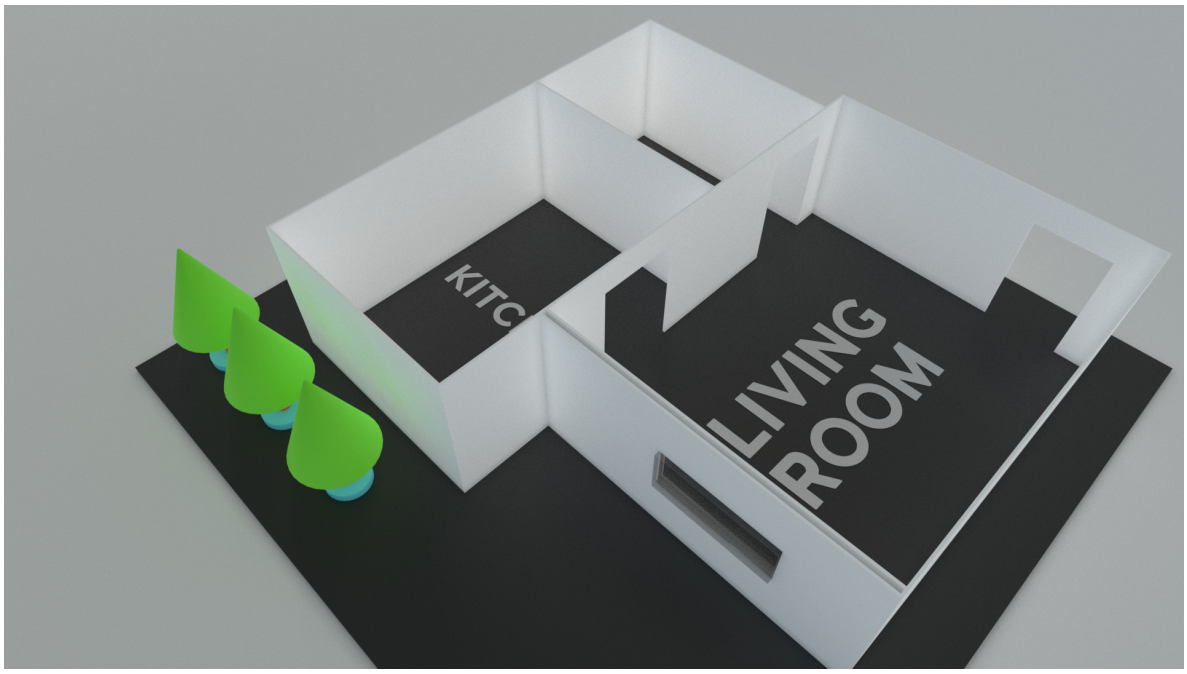

Figure 4.4: 3D design of the smart home model 2

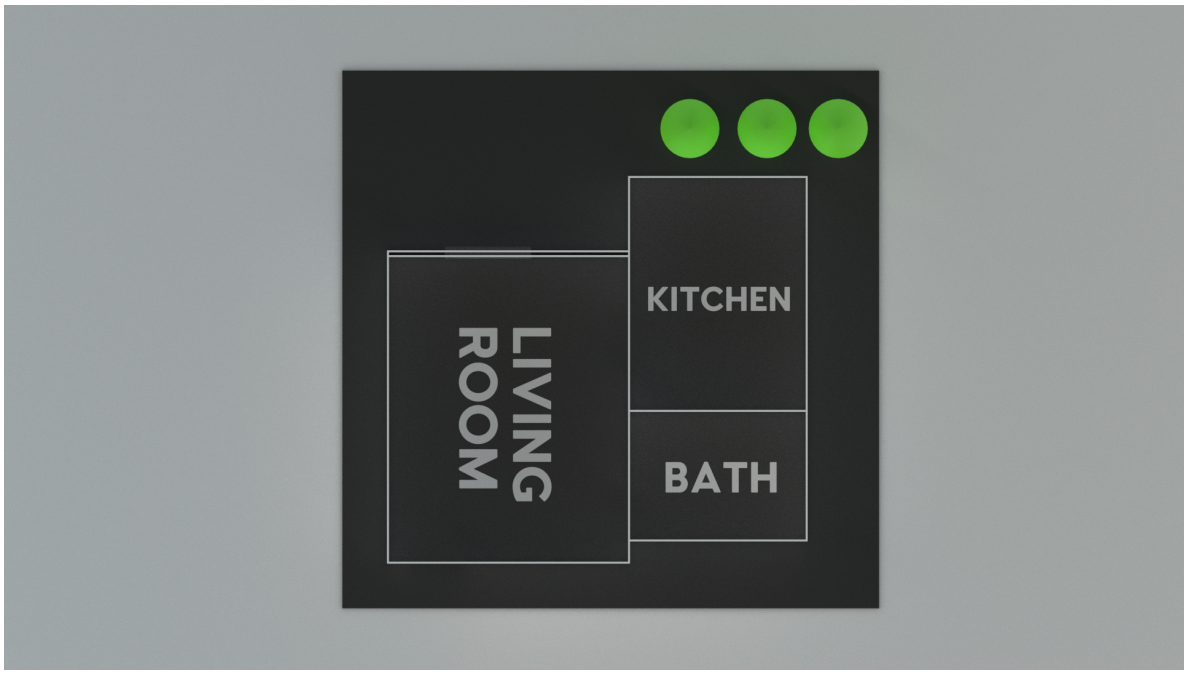

Figure 4.5: 3D design of the smart home model 3

The real model was created using strong marine plywood and painted with a semi-glossy paint.

After the model was built, I designed a set of interactive features for the house. Typically in a smart home, we need to consider both inputs and outputs. For the first, I chose a temperature-humidity sensor, and for the interactive components I decided to use RGB LED lights, a ceiling fan, and a smart window, that can be opened and closed electronically.

4.4 Components

For the LED lights, I used an SMD 5050 RGB LED strip, which I placed around each room individually. The strip was placed around the upper part of the walls of each room. In order to preserve the aesthetic of the house, as well as prevent direct light emission, I used an LED diffuser. The diffuser consisted of an aluminum track for the LEDs, as well as a diffuser cap. I cut the aluminum strip and placed it along the walls of each room, then I placed the LEDs inside and finally closed it with the diffuser cap.

For the ceiling fan, I used a small phone-powered fan, which I disassembled and re-wired to fit my purpose. The fan was then attached to an L-shaped aluminum strip, in order to be brought to the middle of the living room, without blocking the top-view of the model.

Finally, for the window, I chose to further expand my initially planned functionality by using an LCD light valve. This allows me to electronically control the transparency level of the window, switching between black opaque, and transparent. On top of that, I designed a mechanism for controlling the window position, allowing me to open and close the window electronically. For the opening and closing mechanism, I used a stepper motor for lifting the window. Below is the diagram with the design of the mechanism:

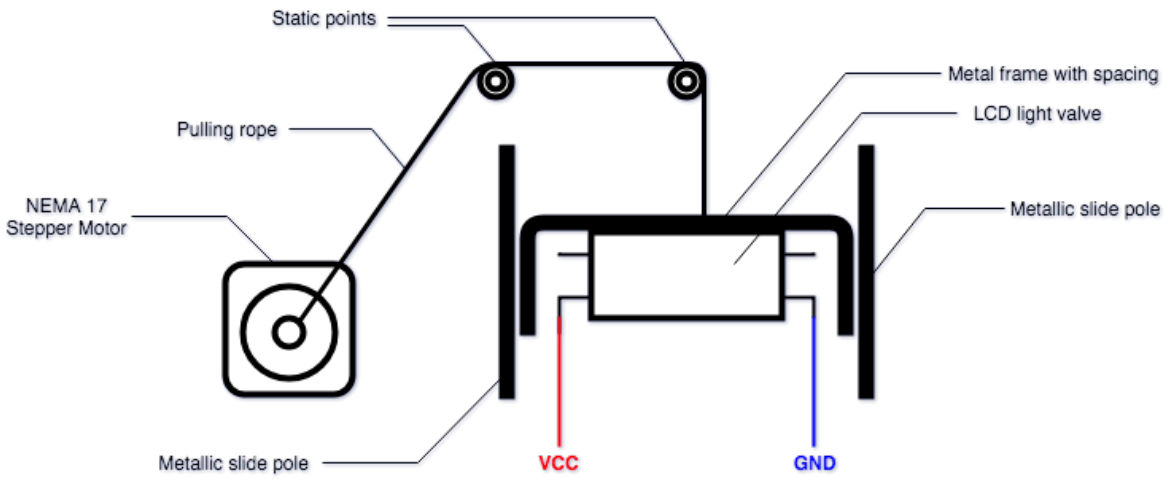

Figure 4.6: Window mechanism design diagram

The window mechanism design process was harder and more laborious than I expected, but the final results were great.

Since the smart components needed a 12V power supply, I couldn't power them directly from the Raspberry Pi. To be able to control the various components through the main system, I used an external 12V PSU, which's flow to each component is controlled by using Mosfets. Each Mosfet was then connected to a GPIO pin of the Raspberry Pi, and after the implementation of custom drivers, I managed to control the fan, the LEDs state, and color in each of the three rooms, and the window opening and closing, along with the dimming mechanisms.

4.5 Mobile apps

The final step for providing a complete set of interactions for the system was the development of companion apps. Since I wanted to implement a mobile app, for both iOS and Android, I decided to use the benefits of modern web technologies and build a Progressive Web App (PWA). A PWA is a web app, along with a manifest and device-specific configurations, which help to achieve a native-like experience. The major benefit from following this route is the cross-platform capability, which makes both the development and the maintenance of the app easier.

For the mobile app, I designed the User Interface using inVision Studio. As with the main system, I tried to keep a clean style while providing all of the necessary features for controlling a smart home.

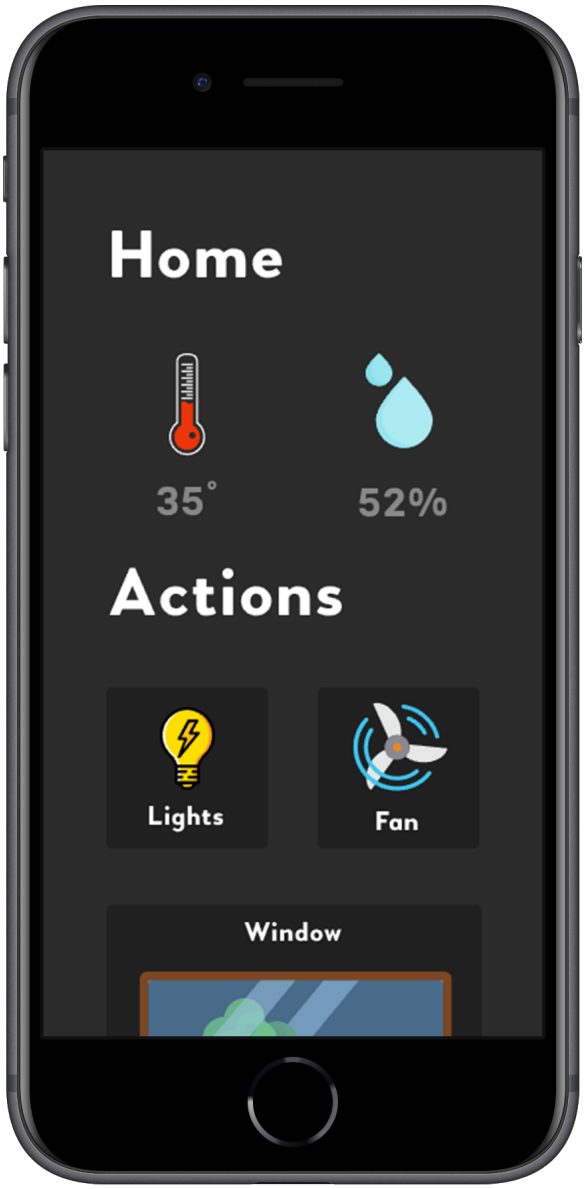

Figure 4.7: Mobile app design 1

Figure 4.8: Mobile app design 2

4.6 Smartwatch app

Although mobile apps are convenient and will cover the needs of most users, having my Samsung Gear S3 smartwatch on my wrist gave me the idea to take the interactions to the next level. Samsung smartwatches run on the Tizen operating system. To develop a smartwatch companion app, I installed and configured Tizen Studio, along with emulators and SDK's that help with the development.

To achieve the functionality I wanted for the companion app, I managed to dig through and modify the core libraries for Tizen development, called Tau. After days of iterative design and programming, I finally completed the app. Through the implementation process, I was testing the app using the provided emulators. When it was the time to install it on a real device, things were not that easy. After a long process of developer account creations and key management, I managed to install the app on my smartwatch, with everything working as expected.

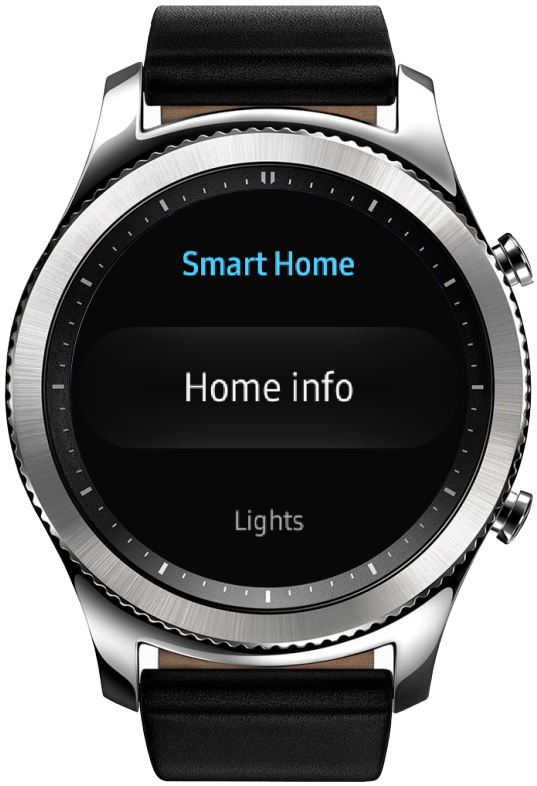

Figure 4.10: Smartwatch app screen 1

Figure 4.11: Smartwatch app screen 2

Figure 4.12: Smartwatch app screen 3

Figure 4.13: Smartwatch app screen 4

Figure 4.14: Smartwatch app screen 5

Figure 4.15: Smartwatch app screen 6

4.7 Communications

To manage all of the communications between the mobile apps, the smartwatch apps, and the main system, my two options were Bluetooth and HTTP. I quickly discarded the first option as it has a limited range and most importantly, a limited number of parallel connections. Since I wanted to allow for as many mobiles and smartwatches to be connected to the system, my only viable option was HTTP.

For the HTTP communications, I could either use a local server, or a remote server, with each option having its advantages and disadvantages. For the local option, the system can only connect with devices that share the same wifi network. The benefit, though, for this option, is a high-speed and reliable connection. For the remote server route, devices can communicate from all around the world, but the speed of the connection may vary.

The solution I chose was to use a remote server and build all of the communications around WebSockets instead of HTTP. WebSockets is a full-duplex TCP communication protocol, with minimal overhead, which results in ultra-fast real-time communication. This allows me to combine remote accessibility with fast connections, which ends up being the best solution overall.

For the handling of the communications, I set-up an AWS EC2 server, to handle the web socket messages from all the devices and pipe the requests to the main system. This server acts as a “middle-man” between the devices and the system, with each device only communicating with a single end-point. The “middle-man” architecture, results in a self-untangling communications system, where all the devices are constantly synced together, as there is only one source of truth.

Chapter 5 - Conclusions

5.1 Evaluation
5.2 Future

5.1 Evaluation

Building the house model allowed me to test the system against real-world situations. The system was tested in both the regular command-response functionality, as well as in the home control and the automation scenarios.

For the first part, the system performed very well, as it could handle a pleasant conversation and provide some great responses. Of course, the intelligence behind the system is the Google Assistant, so in this process, I tested mostly the integration aspect. The system was functioning as expected with both the front-end and the back-end working flawlessly. The speakers and the microphones were functioning correctly, the keyword detection engine was responsive, and the User Interface was seamless, with the home screen design and the structured visual response, as processed through the output parser. Many types of question were tested, from interactive games, weather reports, definitions, and everyday casual questions.

The second phase of testing was about home control and the integration with various types of smart components. The testing involved the communication between the system, the mobile apps, and the smartwatch app, and the handling of control commands, regarding smart home components.

As expected, the WebSockets communication was perfect, with almost zero lag. Both device-system and device-device communications were performed in real-time, with all of the devices staying synced at all times.

For the custom commands parser, I tested many variations of control commands, such as “Turn on the kitchen lights”, “Make the lights blue”, “Open the window”, and more. Everything worked as expected, and the system provided a visual and audio response while executing the appropriate action.

Although the system worked flawlessly, there is still one aspect that could be improved. It seems that the bottleneck of the system is the Raspberry Pi. This project really put Raspberry Pi to its limits, in power, memory, and processing capacity. In a real-world application, I would prefer a more robust system, that is expandable and more stable in terms of power.

5.2 Future

Testing the system in real-life scenarios, with the creation of the house model, led me to realize some important aspects regarding the future of this system.

As for the day-to-day use of the system, as an all-in-one personal assistant solution, the project can be launched as a cross-platform desktop application for everyone to download and try. There are two additions that can be added to the system, regarding this use-case. The first one is the addition of other assistant back-ends, such as the Amazon Alexa, and Microsoft Cortana, with the user being able to choose from them. The second addition, is the integration with other desktop applications, such as web browsers, email clients, and music players.

For the integration of the system with smart components, there are two viable options:

The first option is to ship the system as a complete package, along with all of the necessary hardware, including an embedded computer, speakers, microphones, and a display. This option is along the lines of the smart-display devices that start to appear on the market nowadays.

The second option is to ship the system, in a DIY package, where some of the hardware components are provided, and the rest can be bought separately, and configured by the customer. This option is the easiest and the most cost-effective to follow, and can be implemented iteratively.

Apart from the two options above, in order for the system to have a seamless integration with off-the-shelf smart components, an additional layer of communication must be built, to support the available formats of the market today. Of course, there are a lot of services providing this layer as of today,

so the most viable solution would be to integrate one of these into the system.

References

1. https://www.adafruit.com/
2. https://www.pololu.com
3. https://www.raspberrypi.org/blog/
4. https://www.sparkfun.com/
5. https://www.wikipedia.org/

Appendix A - Photos of the completed system

Figure A.1: Smart display housing - front

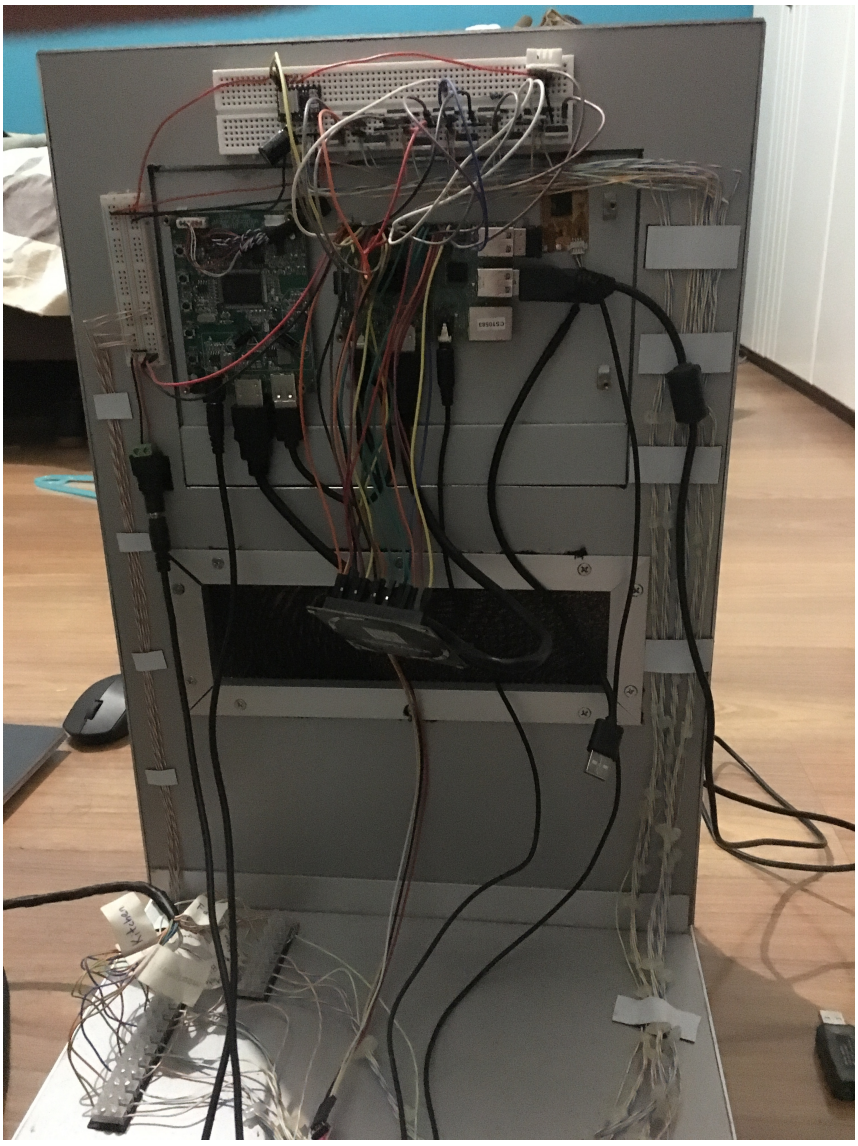

Figure A.2: Smart display housing - back

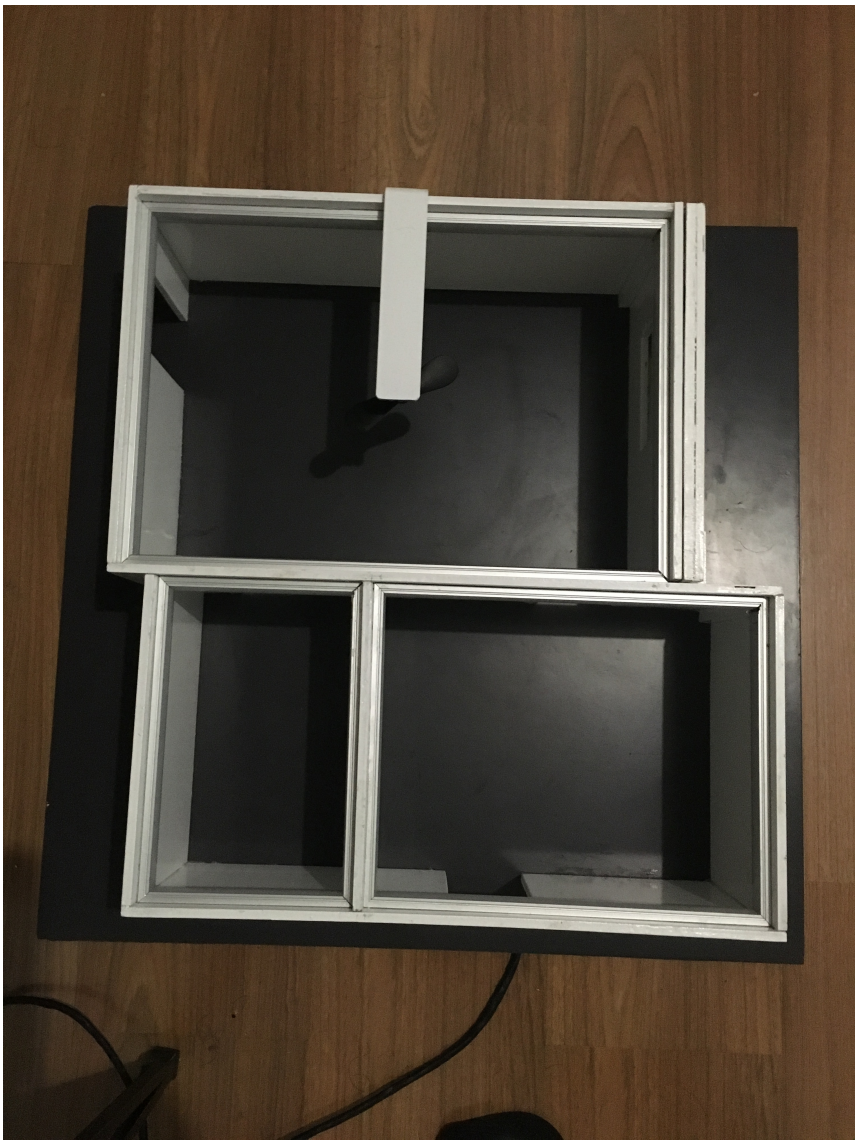

Figure A.3: Smart home model - top

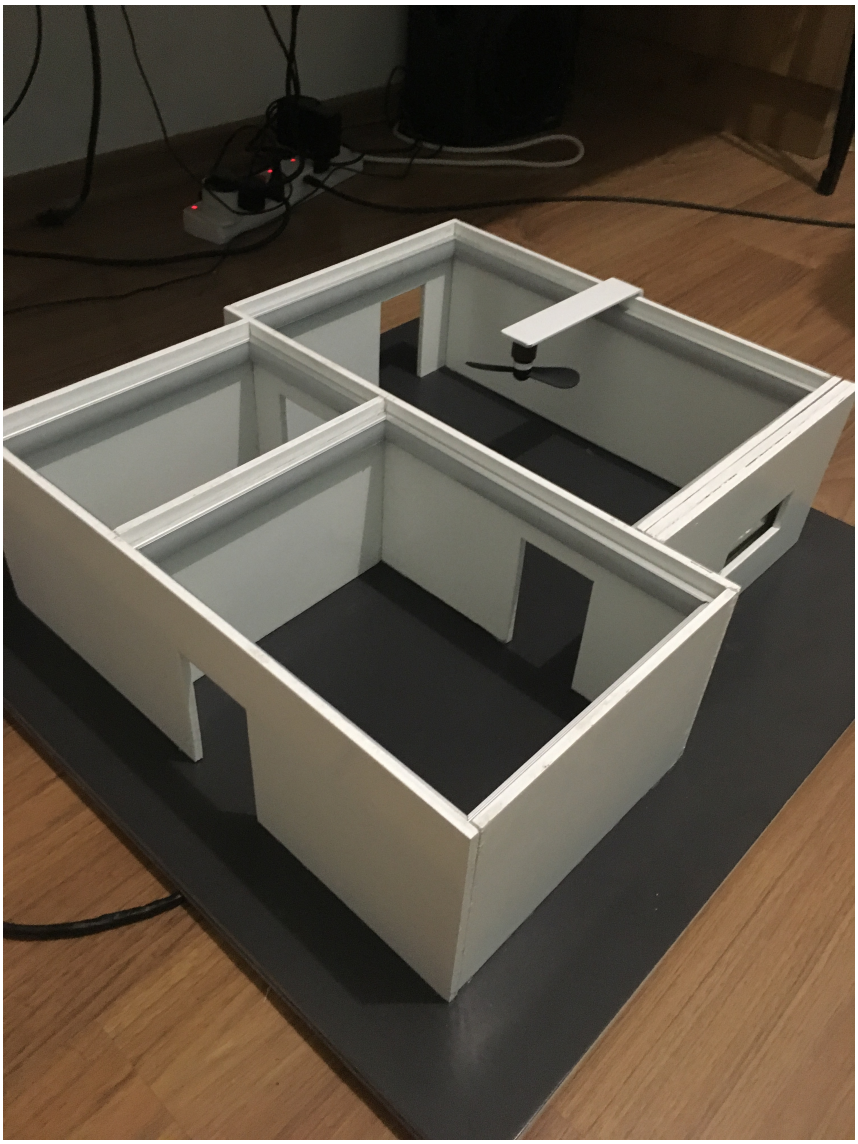

Figure A.4: Smart home model - perspective