**Music-Synced Smart Lighting System**

A dissertation submitted in

partial fulfilment of  
The requirement for the degree of  
**MASTER OF SCIENCE**

in

**Software Development**

in  
**The Queen’s University of Belfast**  
By  
**Ryan Urquhart**

**40099112  
05/05/2025**

**SCHOOL OF ELECTRONICS, ELECTRICAL ENGINEERING and COMPUTER SCIENCE**

**CSC7058 – INDIVIDUAL SOFTWARE DEVELOPMENT PROJECT**

A signed and completed cover sheet must accompany the submission of the Individual Software Development dissertation submitted for assessment.

Work submitted without a cover sheet will **NOT** be marked.

**Declaration of Academic Integrity**

Before signing the declaration below please check that the submission:

1. Has a full bibliography attached laid out according to the guidelines specified in the Student Project Handbook
2. Contains full acknowledgement of all secondary sources used (paper-based and electronic)
3. Does not exceed the specified page limit
4. Is clearly presented and proof-read
5. Is submitted on, or before, the specified or agreed due date. Late submissions will only be accepted in exceptional circumstances or where a deferment has been granted in advance.
6. Software and files are submitted via Canvas.

**I certify that that the submission is my own work, all sources are correctly attributed, and the contribution of any AI technologies is fully acknowledged.**

**I declare that I have read both the University and the School of Electronics, Electrical Engineering and Computer Science guidelines on plagiarism - http://www.qub.ac.uk/schools/eeecs/Education/StudentStudyInformation/Plagiarism/ - and that the attached submission is my own original work. No part of it has been submitted for any other assignment and I have acknowledged in my notes and bibliography all written and electronic sources used.**

**I am aware of the disciplinary consequences of failing to abide and follow the School and Queen’s University Regulations on Plagiarism.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name: (BLOCK CAPITALS)** | **RYAN URQUHART** | | |
| **Student Number:** | **40099112** | | |
|  |  |  |  |
| *Student’s signature* |  | *Date of submission* | **05/05/2025** |

**Acknowledgements:**

* Bee-Yen Toh – My supervisor, who was with me every step of the way
* Cherise McClean – My wife, a constant (and welcome) distraction
* SoDes – On GitHub, created a package that made this whole project possible on Windows (PyAudioWPatch)
* aiXander – On GitHub, created a repository that was adapted, providing fast Fourier transformations

The project discussed in this report is hosted on a repository at:

https://github.com/RyanMcClean/wizlight-music-sync.

If you have trouble accessing the repository, please contact rurquhart01@qub.ac.uk.

Abstract:

Bulb Bop is a Music-Synced Smart Light System. Bulb Bop takes advantage of audio data from PyAudio to allow detection of the music beat and change bulb brightness in time with music. Bulb Bop will work with any number of bulbs, provided they are Wi-Fi enabled and of the Wiz brand. It provides the user with a basic user interface. This will aid in pairing bulbs with the system and enable the user to start the audio synchronisation with audio, either from the local ambient noise through a microphone, or from the audio going to the speakers of the host machine.

Table of Contents

[Introduction: 7](#_Toc197356867)

[1. Understanding the Problem 8](#_Toc197356868)

[2 User Interface Design 10](#_Toc197356869)

[3 Architecture Design and Algorithm Explanation 16](#_Toc197356870)

[**3.1** **Installation Process** 16](#_Toc197356871)

[**3.2** **API Endpoints** 17](#_Toc197356872)

[**3.3** **Bulb Database Model** 19](#_Toc197356873)

[**3.4** **Bulb communication** 19](#_Toc197356874)

[**3.5** **Host Audio Parsing** 21](#_Toc197356875)

[**3.6** **Beat Detection Algorithm** 22](#_Toc197356876)

[**3.7** **Security Considerations** 25](#_Toc197356877)

[**3.8** **Future Development Mindset** 26](#_Toc197356878)

[**4** **Testing** 31](#_Toc197356879)

[**4.1** **Automated Testing** 31](#_Toc197356880)

[**4.2** **Manual Testing** 33](#_Toc197356881)

[**5** **Evaluation and Conclusion** 36](#_Toc197356882)

[**5.1** **Evaluation of Project Success** 36](#_Toc197356883)

[5.1.1 “The system should connect to and control a Wi-Fi-enabled smart light bulb” 36](#_Toc197356884)

[5.1.2 “The system should detect music rhythm and temp from an audio source and synchronise the light settings accordingly” 36](#_Toc197356885)

[5.1.3 “The system should provide a primary control interface” 36](#_Toc197356886)

[5.1.4 “The system should offer multiple lighting modes.” 37](#_Toc197356887)

[**5.2** **Reflection of Process** 37](#_Toc197356888)

[**5.3** **Evaluation of Software** 38](#_Toc197356889)

[**5.4** **Suggestions for Further Work** 41](#_Toc197356890)

[**5.5** **Conclusion** 42](#_Toc197356891)

## Introduction:

This report focuses on the development of a Music-Synchronised Light Control System (hereafter referred to as Bulb Bop). The report will consist of five chapters: Understanding the Problem; User interface design; Architecture design and algorithm explanation; Testing; Evaluation and Conclusion. The content of these chapters will be explained in the rest of the introduction.

Chapter One will explore the problem and the basis on which Bulb Bop was developed. This will include my initial understanding of the project based on the title and the preliminary meetings with my supervisor. It will then progress into my initial investigations about potential solutions to the problem at hand. Finally, this chapter will end with the decided-upon plan of action for development.

Chapter Two will contain the plan and development of the user interface. This will include the early iterations of the user interface and how these early iterations influenced the final design. It should be noted, however, that the bulk of the work in this project was the beat detection algorithm, and therefore, due to the complex nature of the algorithm, this was deprioritised in preference for the algorithm.

Chapter Three will contain the plan and development of the architecture and algorithms design. This section will explain the architecture upon which Bulb Bop can be run and the algorithms that allow it to run. This is the chapter in which the bulk of this project's work will be explained.

Chapter Four will contain the testing carried out on the project during its development and before the submission of the project. This will include risk assessments of the project, and a discussion of the rationale of the tests that were designed, as well as an examination of the areas of the project that were not thoroughly tested and an explanation as to why.

Chapter Five will be the final evaluation and conclusion of the project. Discussing the success or failure of the development of Bulb Bop. It will evaluate different technologies utilised and how they have impacted the final system shape. There will be a reflection on the development of Bulb Bop, highlighting areas that were handled well and areas where there could have been further development.

# Understanding the Problem

The initial description of the project is as follows,

“Develop a system that integrates music with dynamic lighting displays to create an 'instant smart disco light' function or a 'smart lighting solution for improved well-being.’ The system should control a Wi-Fi-enabled smart light bulb, adjusting its brightness and colour in sync with music rhythms, enhancing the atmosphere in any setting.”

In addition, there were four main functional criteria for developing Bulb Bop.

1. The system should connect to and control a Wi-Fi-enabled smart light bulb
2. The system should detect music rhythm and tempo from an audio source and synchronise the light settings accordingly
3. The system should provide a primary control interface
4. The system should offer multiple lighting modes

No restriction was placed on the technologies to be utilised in this project, such as specific bulbs, programming languages, or web frameworks. However, due to having to use third-party hardware (the smart bulbs), a particular API dictated by the manufacturer would have to be used.

The initial prompt does not define the user for whom the application is supposed to be designed. However, through discussions, the intended user was defined broadly, from families to DJs to people hosting a party. The decided goal was narrowed down to someone hosting a party in their house. This user would either use a microphone attached to the computer running Bulb Bop or play the music from its speakers.

Research was carried out from the initial description of the project, which covered two main areas. The first was the bulb, whose brand would be appropriate for the project. The second was the programming language; this programming language had to fulfil two criteria: it needed access to the audio from the operating system (Windows and Linux) and be able to host a web server, which would provide the user interface.

The research into the bulbs brought up many different options for use, Wiz, TP-Link Kasa, TP-Link Tapo, YeeLight, Shelly, Philips Hue, Athom tech, and Switchbot. From all these bulbs, it was decided to use Wiz bulbs. The reasons for this was the cost, the open API (bulbs like TP-Link Tapo have a closed API so users must use their app), the app does not require a hub to operate (Philips Hue requires a hub), and from reading various forums*,* Wiz was highly recommended as a bulb brand. Based on this research, A WizBulb A60.E27 bulb (A colour-changing bulb) was purchased, and later a non-colour-changing bulb was procured.

The research into the programming language to use initially started with Java, the reason for this simply being that much of the CSC7063 Computer Programming course has been taught in Java. Two libraries looked at were TarsosDSP(1) and Jipes by Tagtraum(2), both of which were unable to gain access to audio data from the host machine. Jipes was unsuited as it was not made for real-time processing of audio information. With regards to TarsosDSP, it used Gradle to build it, which was something that, given the time needed to get the project started, was too complicated for the initial developer to implement.

As such, another programming language that would allow access to this host audio data was researched. For this, Python was used; there is a Python package named PyAudio, and it reliably and easily allows developers access to the host machine's audio data. The issue, however, is that PyAudio does not allow access to the loopback API present on Windows machines (3), which allows the developer access to real-time speaker audio data. This was solved when a Windows-specific patch of the PyAudio library was found, called PyAudioWPatch. This allows PyAudio the ability to access the loopback API, with the Django library that can serve web apps with a great deal of flexibility, making Python the perfect choice.

A significant consideration to take into account was the security of Bulb Bop. In the case of this project, there is little to no system security. While it was considered to add security, it was decided against for the following reasons: There are no personal details saved in Bulb Bop, nor would the addition of security, like a login page or OAuth authentication, protect the bulbs from being manipulated by someone who knows how to do so (as the bulbs themselves have no security on the local network).

# User Interface Design

A screenshot of a computer

AI-generated content may be incorrect.

Figure 1 - Index Page (PC View)

The user interface for this system is quite simple. The user can access four different views, each of which is dynamic to varying degrees. Django gives the option to develop HTML that will dynamically change based on a context provided by the developer. For instance, the bulbs on the index page render depending on whether they are present in the database or not.

The first aspect of the user interface we will discuss are the shared parts of all views. The central part of this is the navbar, which contains the title, links to other views, buttons to activate functions of Bulb Bop, and a light/dark mode toggle button. There is also a footer on the page, a line that contains the text, ‘2023-25: Bulb Bop: Ryan Urquhart: 40099112’. Part of these aspects are dynamic; the audio sync option, for instance, only shows when bulbs are recorded in the database.

The second view we will look at is the home page, the index page (Figure 1 and Figure 2), which is the main page that the user will be engaging with. This page displays icons for the bulbs that are recorded in the database, their names, as well as graphically displaying their state. Some tooltips appear when the user hovers their mouse over the bulbs, which inform the user that, should they click on it, it will toggle the state of the bulb. Should the user not yet have connected Bulb Bop with any bulbs, then the index page will prompt the user to use the discover function of Bulb Bop (Figure 3).

A black and white image of a person standing in front of a white sky

AI-generated content may be incorrect.

Figure 2 - Index View (Mobile View)

A screenshot of a computer

AI-generated content may be incorrect.

Figure 3 - User prompts to add bulbs

A screenshot of a computer

AI-generated content may be incorrect.

Figure 4 - FAQ's Page (PC View)

The third aspect of the user interface we will discuss is the ‘FAQ’ (frequently asked questions) page (Figure 4). This page covers some basic setup questions users may have, from connecting lights to troubleshooting, and a contact email for support (a placeholder email). This page is referenced in the tooltip to help new users get started with the application. It also links to the WizBulb app, to show users how to connect their bulbs to the local network (unfortunately, using Bulb Bop to connect the bulbs to the local network was impossible as it requires Bluetooth and is undocumented).

A screenshot of a computer

AI-generated content may be incorrect.

Figure 5 - About View (PC View)

The fourth page of Bulb Bop to discuss is the ‘About’ page. This page simply displays information about the project, its goals, and some information about the project author. Like every other page, this page has the navigation bar, which allows the user, should they have bulbs saved in the database, to activate the audio sync.

The final page to discuss in this section is the ‘Edit’ page for the bulbs. This page allows the user to edit or delete the bulbs that they have saved in the database. The user can edit the bulb's name and IP address. When Bulb Bop has a bulb saved with a name and an IP address, then it will query the bulb and update its status in the database itself. When the user clicks on edit, it will show a modal that the user can interact with to edit the bulb. When the user clicks save, edit, or delete, the page will display a success or error message in the bottom right corner.



Figure 6 - Edit View with two bulbs

A red sign with white text

AI-generated content may be incorrect.

Figure 7 - Example Error Message

A green screen with white text

AI-generated content may be incorrect.

Figure 8 - Example Success Message

To finish, the dynamic feature of Django HTML templates has allowed for the addition of logic into the rendering of the HTML files for the user. For instance, the error toasts won’t be rendered to the page unless the ‘context’ passed into Django indicates that an error is present. This dynamically rendered HTML allows for incredibly flexible views while lessening the amount of code needed to render them. One example of this, is the HTML shared by all views is created by a master template, which all other views build on top of (Figure 10).

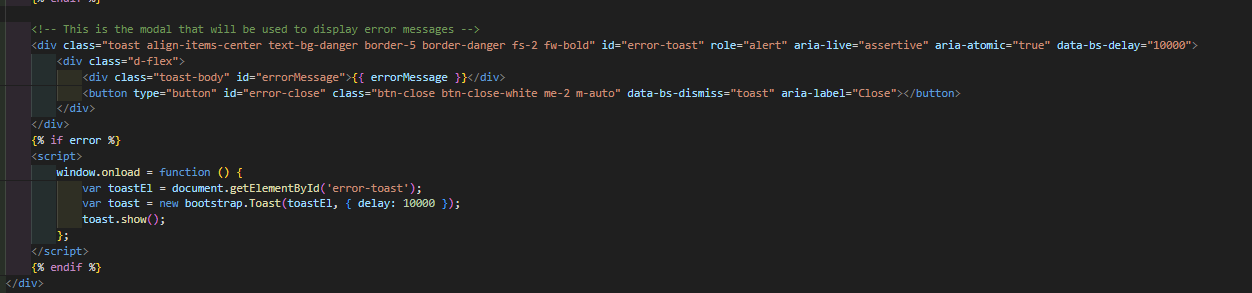


Figure 9 - HTML snippet with Django logic

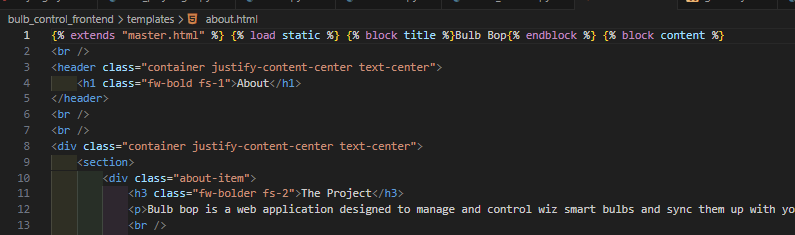


Figure 10 -' about.html' extending 'master.html'

# Architecture Design and Algorithm Explanation

In this section of the report, the discussion will centre on the ‘backend’ of Bulb Bop. This will include a description of the API endpoints, the bulb database model, how Bulb Bop was configured to interface with bulbs on the network, a description of how access was gained to the audio data of the host machine, an explanation of the algorithm to detect beats in music, and finally a section on how Bulb Bop was produced with future development in mind. This section will encompass most of the work done on this project and, therefore, will be the longest section in this report.

## **Installation Process**

The installation process for Bulb Bop is a rather involved process, it is well documented for new users in the ‘README.md’ file in the repository (Figure 11). It requires the user to be somewhat familiar with the command line interface, and the installation of the requirements for Bulb Bop can take some time. During development, PyInstaller was the ideal candidate for creating a single executable file that could be used to run Bulb Bop. While using PyInstaller did create a single executable file, it failed to follow all of Django’s dependencies and would not run. The installation process has been marked as a candidate for future work.



Figure 11 - Installation Process Documented in README

## **API Endpoints**

This project has 14 API endpoints, not including requests that get images and JavaScript code. Four of these endpoints are for different views or pages, of the user interface, one is reused for bulb form submission, therefore, these will not be discussed in this section, as the views were mentioned in chapter 2. There are three endpoints for bulb control, four (including the reused view endpoint) for database operations and, two for error and success handling.



Figure 12 - List of API endpoints

The endpoints for bulb control are as follows: “/discover/”, “/toggleBulb/”, “/queryBulb/”, and “/colorBulb/”. Of these four, only three are currently utilised in Bulb Bop, “/colorBulb/” was developed for use, but when testing with a colour changing bulb, and a non-colour changing bulb this endpoint caused issues, so was left for future development, which ultimately ended up being scoped out of the project.

“/discover/” is used to discover new bulbs on the network. It sends a broadcast UDP packet, to which bulbs connected to the network respond with their current state. This response contains the IP address of the bulb, as well as its current state, this allows it to be saved in the database for future communication. This response is shown to the user, who can give the bulb a name, which is then sent to another endpoint, which will be mentioned later.

“/queryBulb/’ is similar to “/discover/”, except rather than sending a broadcast packet to all IP addresses, it sends packets to specific addresses to query the state of particular bulbs, either bulbs that are saved in the database, or have recently been discovered. This state is then used to update the icon on the user interface, coloured orange if the bulb is turned on, or grey if the bulb is off.

“/toggleBulb/” is the primary endpoint the user will trigger from the user interface, when they click on the icon of a bulb. This action will send a UDP packet to the specific bulb and instruct it to turn off or on, depending on the state of the bulb recorded in Bulb Bop.

The endpoints that control database operations are “/”, “/delete/<str:ip>/”, and “/edit/<str:ip>”. The “/” endpoint is the index endpoint. When this endpoint is triggered with a GET request, it renders the home page; when it’s triggered with a POST request, it submits a Django model form.

This form is how Bulb Bop saves new bulbs in the database. When the form is submitted, it is checked to ensure that it contains no errors. The criteria for this check is that the IP address is valid, and that the name of the bulb is not null. The other information for the bulb is filled in from the return from the broadcast packet that was sent during discovery.

The next two endpoints are accessible to the user through the edit bulbs page, “/delete/<str:ip>/” and “/edit/<str:ip>/” both of these endpoints require the IP address of the bulb in which they are editing, as a unique identifier. “/delete/<str:ip>/” will delete the bulb with the specified IP address, and “/edit/<str:ip>/” will submit a form, that is verified, of a new IP address or name for the bulb, this then updates the model in the database for that bulb.

Two endpoints, “/clearSuccess/” and “/clearError/”, are implemented to manage the handling of error and success messages, preventing redundant or repeated displays. These endpoints are never interacted with by the user manually, they are triggered through JavaScript that runs when every page loads. While this does not interfere with a message that is being displayed in the currently loaded page, it will prevent a message from being displayed in another page that the user navigates to.

There are two endpoints for controlling the audio synchronisation, “/activateSync/”, and “/stopsync/”. These are self-explanatory, “/activateSync/”, this endpoint accepts a POST request that contains the index number of the audio device selected by the user. This then activates the audio sync in Bulb Bop for that audio device, Bulb Bop waits thirty seconds to ensure that the sync program has activated, then it sends a status message to the user interface.

This concludes the API endpoints activated by the user or by the JavaScript that the pages contain. The next step of this chapter is to discuss how Bulb Bop achieved communication with the bulbs, the documentation that was followed, and the technologies involved in its process.

## **Bulb Database Model**

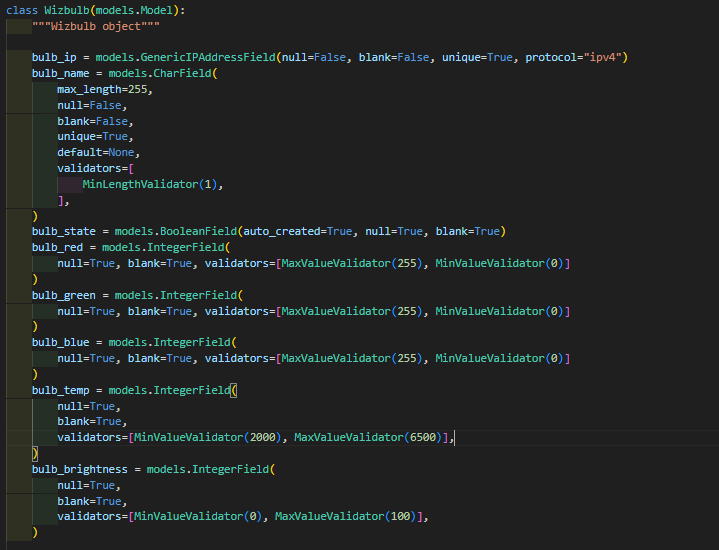


Figure 13 - Bulb Database Django Model

The database design in this project is simple, it only contains one model, the bulb. This model consists of seven fields: the IP address, name, state, red, green, blue, temperature, and brightness. While all of these fields are updated when a bulb is queried, only the IP address, the name, and the state are used in Bulb Bop. For each field, there are several Django validators applied, which prevent bulbs from being created unless the criteria are met. For example, the IP address must be unique, it cannot be null or blank, and it must be a valid IPV4 address.

The simple nature of this database means that it is easily maintainable and should Bulb Bop be expanded to handle more brands of bulbs in future, it would be little work to expand it to encompass those bulbs.

## **Bulb communication**

One of the reasons that the Wiz brand of bulbs was chosen for this project was the level of documentation about the bulb (4). To augment this, there is also a Python package called, ‘pywizlight’ which utilises this documentation to provide communication to wiz devices (5)I did not use this package and primarily relied on the Wiz documentation to enable communication with the bulbs.

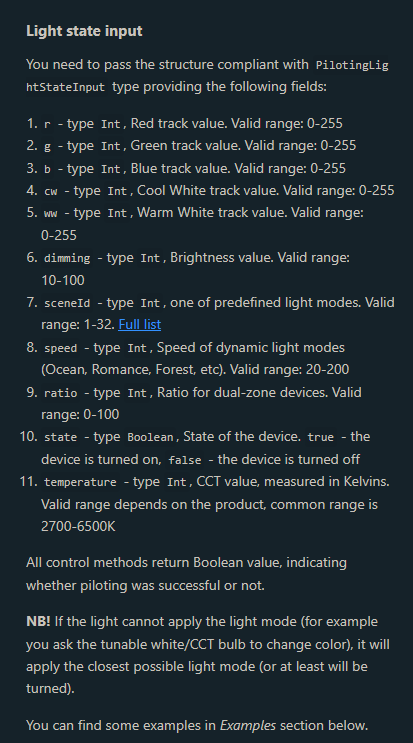


Figure 14 - Excerpt from API Reference Site

Utilising the API reference, five UDP packets have been implemented for Bulb Bop to use. Two other UDP packets are also present, which are not currently used, but can be implemented in further development. The seven total packets have the following functions: bulb discovery, turn on bulb, turn off bulb, turn to half brightness, turn to full brightness, bulb registration, and turn bulb to colour. Bulb registration and turning the bulb to a colour are unused by Bulb Bop, while both were used in the development of Bulb Bop, so it was unfeasible to implement them properly in the time given for the project.

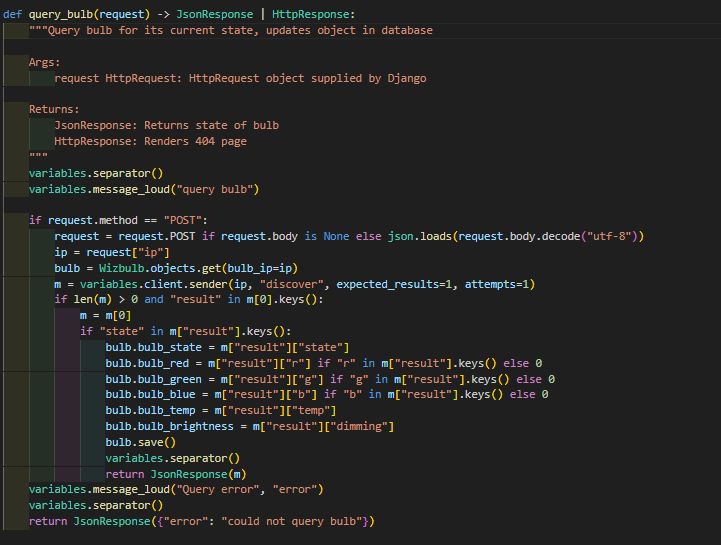


Figure 15 - Query Bulb Function that parses the bulb packet response

Functions that can read and parse the returned UDP packets from the bulbs have also been implemented. These packets contain information that allows Bulb Bop to update the bulb object and database model with their current state. This allows the index view, for instance, to change the colour of the bulb icons depending on the current state of the bulbs.

One of the unused functions to turn a bulb to a specific colour, was implemented in the initial designs of the project. However, upon expanding the project to work with multiple types of bulbs (colour changing, or non-colour changing) this caused an issue with bulbs that do not change colour. It is difficult and currently beyond the capabilities of Bulb Bop to distinguish between types of bulbs. This difficulty in communication is why this function is currently unused. This has also been marked for future development.

## **Host Audio Parsing**

The reason that Python was used over Java was that Python has easy access to the audio of the host machine. Initially, only PyAudio was used, and Bulb Bop was able to access microphones attached to the host machine; however, then PyAudioWPatch was discovered.

PyAudioWPatch takes advantage of Windows Audio Session API (WASAPI) (3), which allows the audio of output devices to be intercepted and parsed by the library. This allows Bulb Bop to be able to parse music played from local machine and control the lights accordingly, this is a considerable improvement over having to use Bulb Bop with a microphone which would have picked up all stray noises, making the beat detection a much harder task (using a microphone is still as possible as using a speaker is). On Linux PyAudioWPatch does not work, as WASAPI is a Windows only API, however, this can be overcome by using PulseAudio on a Linux machine, it is easy to create a loopback of a speaker so that it registers as an input device.

So, due to this Bulb Bop had access to the raw audio output of a selected microphone or speaker of the host machine. This, however, was unhelpful when it comes to detecting beats, as PyAudio (or PyAudioWPatch) return a stream of un-processed data. This un-processed data needed to be sorted into frequencies and amplitudes, necessitating the use of Fast Fourier Transformations. This, however, is a complicated process. A repository was found GitHub that would handle this calculation, Realtime\_PyAudio\_FFT (6), which was added as a submodule to Bulb Bop. The repository itself was helpful; however, it was not designed for the needs of Bulb Bop, thus it was modified to suit, changes can be seen on its repository (7). This meant making it work with PyAudioWPatch, but it had to still work with PyAudio (to allow it to work on Windows and on Linux).

A screenshot of a computer

AI-generated content may be incorrect.

Figure 16 - List of commits to Realtime\_PyAudio\_FFT submodule

Realtime\_PyAudio\_FFT was changed so that rather than visualising the audio output the repository would instead just return the stream of frequencies and amplitudes that are being produced by the audio device. To make the software more robust, the repository was also changed so that it would use the ‘PyAudioWPatch’ library, if possible, but would fall back to ‘PyAudio’ if ‘PyAudioWPatch’ is not available (i.e. on Linux rather than Windows). Another change that made, was the editing of channels from one to two, and audio rate, this was necessary to allow the use of speakers with Bulb Bop, as speakers often use two channels, for left and right, they also tend to use a higher rate for the audio than microphones do.

With the libraries and changes made above, once the user selects an audio device and clicks to start the synchronisation of music, Bulb Bop will start a stream of amplitudes and frequencies. This stream is then parsed by Bulb Bop and using the beat detection algorithm will send UDP packets to the bulb(s) to turn them up, or down, in brightness in time with each other and the music. This beat detection algorithm is what will be discussed in the final section in this chapter.

## **Beat Detection Algorithm**

Now Bulb Bop is getting a stream of computed frequencies and amplitudes from the audio device, as the final goal of the project is to have the beat of the music be what influences the control of the bulbs, the audio data will need to be manipulated to detect the beat. From my research this is an extremely difficult task, a lot of beat detection does not happen in real-time, rather it observes an audio file in its entirety and detects the beat that way (8). Inspiration was taken from an example of beat detection that is used in real-time (9), this example comes with formulas that were adapted or created to serve the purposes of Bulb Bop.

When it came to parsing audio for Bulb Bop, the initial task was to figure out the frequencies that needed to be considered, and which ones could be ignored. Ultimately it was decided that all frequencies over one hundred and fifty hertz were to be ignored. This is because the higher frequencies contain vocals and other instruments that are not indicative of a beat in the music. The lower frequencies, however, are where most of the drums are contained, this is where the beat will be most easily detectible, and therefore these are the frequencies that Bulb Bop will be using in the beat detection. The rest of this section will cover the input of frequencies and amplitudes and how it is manipulated to define a threshold for beat detection.

The above equation is how Bulb Bop gets the sum of the amplitudes (E) for a particular set of frequencies (i to k+n). This is achieved by calculating the sum of the Fast Fourier Transformation (FFT) amplitudes (FFT[i]) where ‘i’ is the frequency. This is done for the range i=k, i.e. the starting frequency of the sum, and k+n, where k+n is the max frequency of the sum. In Bulb Bop this is done for three ranges of frequencies, where n always equals 50, but i is equal to, 0, 50, and 100. This leaves Bulb Bop with a sum of amplitudes, from 0-49hz, 50-99hz, 100-149hz. Every frequency above 150hz is ignored.

This sum of amplitudes is then added to an array, one amplitude for each unit time. The unit of time in this case is dependent on how fast the host machine can compute the amplitudes of the audio data which it is manipulating. This means that an average can be calculated for each array of amplitudes, this is done as below.

This average can then be used to determine the fraction of the average compared to the sum of the amplitudes for that frequency range, this fraction can be considered the variance. The sum of these variances for the length of ‘buffersize’ can then be considered to be ‘F’.

This sum can then be used to calculate the average variance (V), however, to ensure that the average variance is greater than the one (this is needed so that the beat limit is greater than the average frequency range amplitude), when calculating the average variance, if the variance is less than one, then the inverse of the variance is taken.

Now that the average variance (V) has been calculated, this can be used to calculate a threshold value, this is the threshold which if the sum of amplitudes for a frequency is greater than, then it will be considered a beat. The equation to calculate the threshold is as follows.

Upon the detection of a beat by the explained algorithm above, Bulb Bop will then send a UDP packet to all bulbs connected to Bulb Bop and it will change the brightness of the bulbs appropriately. Figure 17 shows how the algorithm is expressed in code.

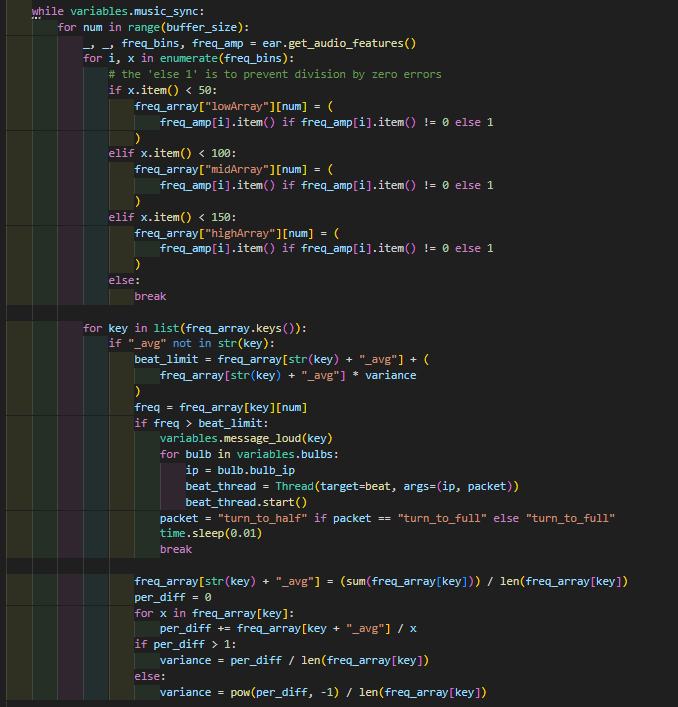


Figure 17 - Beat Detection Algorithm in Python code

## **Security Considerations**

As with all applications, Bulb Bop needs to take security into account during its development. To do so, risk assessments are needed to be able to implement context-aware security (10). These risk assessments cover, what points of Bulb Bop could be vulnerable, where potential ‘attacks’ could come from, and the damage from those potential attacks.

Vulnerable parts of Bulb Bop include any parts that could have access to a users personal data, or devices. Bulb Bop does not handle any users personal data, but does handle the users personal devices (the bulbs). The risk to the users devices has to be assessed in context with the current implemented security on those devices already. With regards to Wiz bulbs, there is no local security or authorisation on the bulbs, this means that the security on these bulbs is only the security already present on the network (Firewalls on routers, closed ports etc.). As such additional security added to Bulb Bop to prevent ‘bad actors’ from manipulating the bulbs would be unnecessary, as if they have gained access to the local network then the bulbs are already vulnerable to their control. As such, the decision was made not to add security to Bulb Bop, should in future the API on Wiz bulbs change, or should other bulbs that require authentication be added to Bulb Bops capabilities, then the questions around security should be re-visited.

## **Future Development Mindset**

Part of the requirements for this project was to ensure that the project was left in a position that would be conducive to future development. This means that the code is clean, organised, and written in such a way that should another developer with no experience of Bulb Bop (but experience in the language and framework), pick up this project where it has been left off, they would have no difficulty understanding the development choices and code. To go beyond, this project also includes organisation of the GitHub repository that it is hosted on (11).

With regards to clean code, this means that Bulb Bop is organised into directories, each of which is named to label the contents clearly. For example, the templates folder holds all the Django template files that are used to render the webpages.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 18 - Display of Directory structure

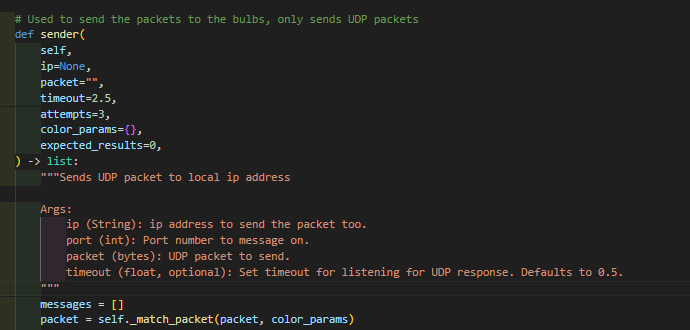


Figure 19 - Example of docstring’s and comments

While this is helpful, it is not enough if the content of these files is hard to read or understand. To combat this deficiency, two approaches have been taken. One, the code base is automatically formatted by one of two formatters. For python files, the code is formatted by the “Black” formatter (12), if the file is an ‘HTML’ file, or a ‘JavaScript’ file, then it will be formatted by the ‘Prettier’ formatter (13). The second approach is the manual addition of comments, docstrings, and helper files.

The addition of formatters into the project means, that whoever is adding code into the project, the style will be consistent making it easier to understand. ‘Prettier’ formatter, is manually configured with settings that will maintain consistent ‘HTML’ and ‘JavaScript’ formatting. The ‘Black’ formatter maintains python files in a format that is consistent with Pythons PEP 8 guidelines (14).

The project is also linted using, Pylint(15). This linting provides a score out of ten for how closely the code follows the PEP8 standards. This score is displayed on the README on the GitHub page.

The focus of the PEP8 style guide alongside comments, docstrings, and helper files is readability. Each module, function, class, and method have docstrings present on them, which explain the intended purpose of the following code making it easy for a future developer to quickly understand each function.

There are specific parts of Bulb Bop that are written but not in use. One example of this is a function that produces a packet that can turn a WizBulb to a specific colour. This is something that was intended to be included in the final project, however given time constraints this was not implemented. These sections of code were left in place to allow for their use without further research in the future. These sections are clearly labelled as not in use to ensure that they are not misunderstood.

The GitHub repository for this project contains extensive organisational rules, guidelines, and continuous integration (CI) to ensure it maintains a consistent order for future development. Some of these rules have had to be ignored for the initial development of Bulb Bop. Others, like the CI for instance were extremely helpful in catching bugs and maintaining code style.

Branch protection rules, have been implemented to prevent direct commits to the main branch without prior review via a pull request. For future and larger development teams this is important to reduce the likelihood of faulty or suboptimal code bring pushed to critical branches. A template has been implemented to keep pull requests to the same standardised workflow, as well as the addition of various labels to help organise pull requests and issues. This makes it simpler for a future developer to understand the version history of the project.

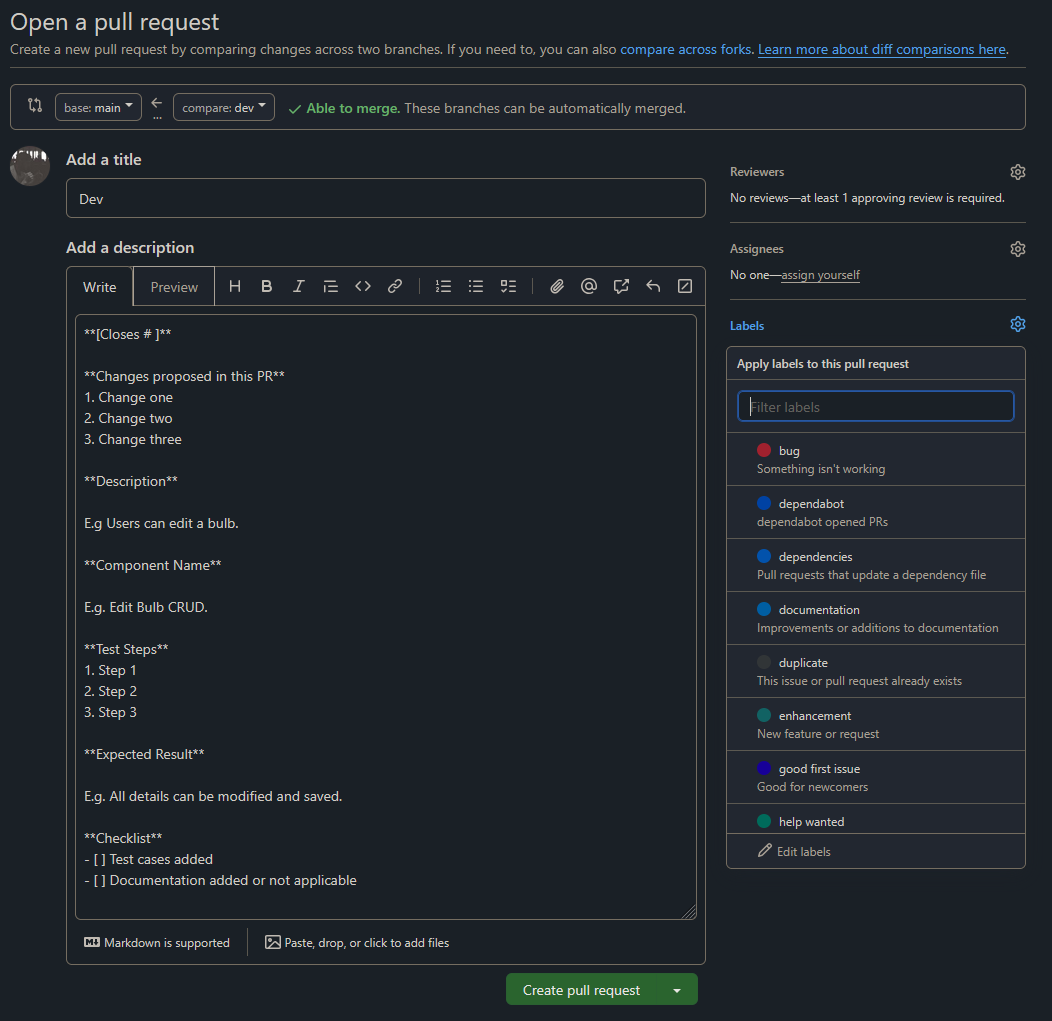


Figure 20 - Example of Pull Request Template

The most helpful feature of GitHub was the CI, which allowed the testing of commits and pull requests made to the GitHub repository automatically. Using CI the repository was able to run the entire testing suite on each pull request, and scheduled each night at midnight, this then sends email notifications should a CI run fail. On top of that, the CI is able to handle other automations that are helpful to maintaining an awareness of the state of the code in the repository, such as the Pylint badge, the coverage badge, and a badge that displays the status of the last CI run.

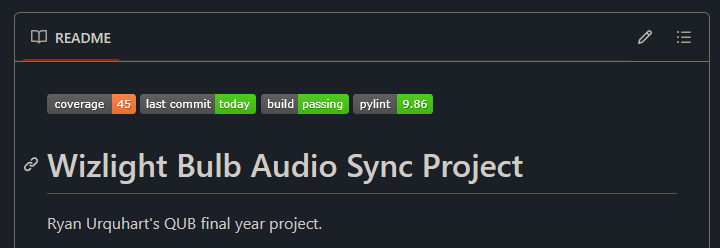


Figure 21 - Screenshot of GitHub buttons

# **Testing**

In this chapter the testing that has been carried out on Bulb Bop will be discussed. This will cover two main sections, automated testing, and manual testing. Test cases for both will be discussed, their results, future testing that could take place, and weaknesses of the testing procedure will also be discussed.

## **Automated Testing**

Most of the testing that has been carried out on Bulb Bop was automated, this is due to the CI on GitHub that allowed the automated tests to be carried out again and again. The automated testing was extremely helpful as it allowed for the ability to check on the status of code changes before committing to the repository. The development of the tests focused on the ability of the user to start the audio synchronisation, as this was the main objective of the project. This was also the main objective of the manual testing on the project, which will be discussed in the next section.

The automated testing in this project takes on three shapes, firstly unit tests, its function, and the database; secondly running a mock live server and testing that the UI functions as expected; and finally testing the API endpoints to ensure that the code functions as expected. With these three approaches, it is possible to cover most of the users experience with Bulb Bop, as well as many of the functions that the user does not interact directly with. There are, however, some functionality that is not tested in this approach, this can be due to several reasons. Code that is too complex, code that can only be ran on a live and active (has bulbs connected and live) system, or simulating the inputs is too complex, for example, simulating the input of an audio device into Bulb Bop is too complex for the scope of this project.

To discuss the testing the format will follow the user journey to starting the audio synchronisation. The steps are as follows:

1. Install Bulb Bop
2. Load the index view
3. Discover bulbs
4. Save bulbs
5. Select an audio device
6. Start the audio synchronisation

There are other actions that the user can take. These are also tested, such as loading the FAQ view, or editing a bulb, these are also tested for, and the documentation for these can be found in the attached appendix (Appendix 1).

There are two ways that Bulb Bop can be installed from scratch, one is manual and the other is automated. The automated way is via GitHub actions. The actions are configured to pull the GitHub repository and then to build and run tests against it. Each time the GitHub actions run Bulb Bop is installed on two separate windows virtual hosts, and two separate ubuntu virtual hosts, one of each of these sets is ran with Python 3.11 and the other with Python 3.12. This ensures that Bulb Bop remains installable, but also that it works with Python 3.11, Python 3.12, Windows, and Ubuntu.

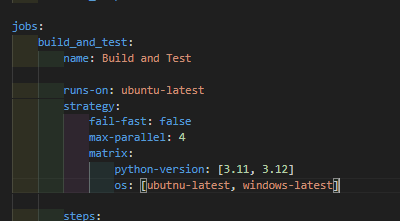


Figure 22 - Configuration of systems to run GitHub actions on

This test has been incredibly helpful in discovering bugs in Bulb Bop, for instance, there was a bug that when Bulb Bop was installed it would try and access the Wizbulb database model before the table was created. This caused an error that would prevent installation of Bulb Bop, this bug was discovered by a run of the GitHub actions that failed. Bulb Bop on the development host machine missed this bug as the database was already created on the machine.

For tests that involve loading and ensuring that the page has been loaded correctly Playwright, a browser automation testing framework, was used to run these tests. Django has the capability to load a live server with an empty database, allowing Playwright to load the views of Bulb Bop and ensure that elements are loaded and in the appropriate state, within the DOM (i.e. visible, attaches, etc). This is more valuable than just simply checking that the correct template was rendered, as it more accurately checks the users' experience of the user interface.

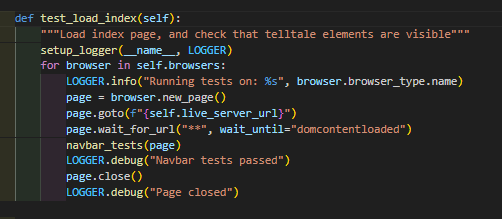


Figure 23 - Playwright load index test

The Playwright testing allows for some small compatibility testing as well. The tests can be run on multiple kinds of browsers, they are configured to run on Firefox, and Chrome. Due to this testing, it is proven that Bulb Bop is not only cross OS compatible, but it is cross-browser compatible as well. There are a series of these tests that run, testing each view in turn, as well as navigation between views. Those not discussed here are documented in the appendix (Appendix 1).

For the user to discover bulbs they must click the ‘Find Bulbs’ button on any of the views on Bulb Bop. This navigates the user to the ‘/discover/’ view. This page is almost identical to the index view; however, Django has templates that allow the HTML files to change dynamically based on the context. Navigating to the discover page changes this context and now the page will display un-saved bulbs connected to the local network on the index view.

There is a difficulty in automated testing of the bulb discovery function of Bulb Bop. When running these tests on GitHub runners, which are virtual machines with virtual networks, there are no bulbs that can be discovered. This means that the discovery function of Bulb Bop is unable to be fully tested by automated testing as it would fail when the bulbs were not discovered. This is something that must be handled via manual testing, however, successful navigation to the discover page can be tested as below (Figure 24).

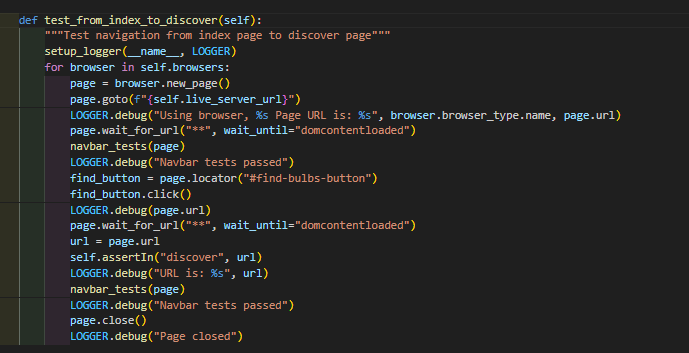


Figure 24 - Test Navigation from Index to Discover Views

The final three sections of the user journey to starting the audio synchronisation need manual testing to fully test. There is some small amount of testing that can be carried out via automated testing. Such as ensuring forms submitted properly or improperly are accepted or rejected as appropriate or testing the API endpoint to start the audio synchronisation. These are helpful but not complete until the manual testing is finished.

## **Manual Testing**

To be able to completely test the user journey of Bulb Bop manual testing is needed. These cover testing the ability to discover bulbs, to toggle bulbs, to start the audio synchronisation, and to stop the audio synchronisation. There are many functions of Bulb Bop that are covered by the manual tests, the ones discussed here will cover the user journey for the main functionality, other tests not discussed here will be available in the appendix (Appendix 1).



Figure 25 - Discover Bulbs Manual Test

The above test covers the first step in the user journey to activating the audio sync, discovering the bulbs, this can only be done manually because the automated tests don’t always run on a network that has bulbs connected, and therefore, the tests cannot be configured to expect them to be present. Documentation for the tests will all follow the format of Figure 25 and will be present in the appendix (Appendix 1).

Figure 26 - Steps for saving a bulb to Bulb Bop

The next step of the user story is the ability to save bulbs, this has been tested successfully multiple times. The next step is then to test the ability to select an audio device, this step had to be done manually because when using GitHub runners, they are virtual machines which have no access to an audio device, therefore the dropdown menu in Bulb Bop has nothing to be populated with. Even when manually running this test the dropdown menu can be confusing, as the names for the audio devices are pulled directly from the host machine. These names can be confusing unless the user has some awareness of the audio devices attached to their host machine.

Figure 27 - Test Steps for Selecting an Audio Device

With the ability to discover bulbs, save bulbs, and then select an audio device, the final step that needs tested in the main user journey is the ability to start the audio synchronisation. This again is a step that needs manual testing as the tester needs to watch the bulbs to make sure they change in time with the music. This final step is also the step that was used to test the beat detection algorithm, this was rather difficult to test as beats in music can be quite fast, or hard to pick up.



Figure 28 - Test steps for Starting the Audio Sync

Once these manual tests were run in accordance with the automated testing the main requirements of Bulb Bop were tested and proven functional.

# **Evaluation and Conclusion**

## **Evaluation of Project Success**

At the start of this section, the initial requirements of Bulb Bop will be reiterated:

1. The system should connect to and control a Wi-Fi-enabled smart light bulb
2. The system should detect music rhythm and tempo from an audio source and synchronise the light settings accordingly
3. The system should provide a primary control interface
4. The system should offer multiple lighting modes

This section shall take each of the four requirements in turn and discuss their completion or lack thereof.

### “The system should connect to and control a Wi-Fi-enabled smart light bulb”

As seen through the manual testing (Figure 25, Figure 26 and Figure 29), Bulb Bop is able to discover bulbs connected to the network and save them in the database for future control. The only weakness of Bulb Bop here is the inability of Bulb Bop to pair bulbs to the network in the first place. This weakness cannot be solved, WizBulb does not document the pairing process of the bulbs to the network and must be done via their proprietary application.

### “The system should detect music rhythm and temp from an audio source and synchronise the light settings accordingly”

As seen in Figure 27 and Figure 28, Bulb Bop is able to detect audio devices on the host machine, parse their audio input/output, and use that to change the brightness setting of bulbs saved to Bulb Bop. The algorithm that is used was documented in Chapter 3, section 5, this algorithm, is not perfect, through manual testing it can been seen to react to the beat, but will occasionally miss a beat, or flicker the bulbs when there is no beat.

### “The system should provide a primary control interface”

The index view of Bulb Bop handles this. When bulbs are saved in the database and present on the network, the bulbs can be toggled (turned on and off) by clicking on the appropriate icon (Figure 29). The user is also able to edit the name of the bulbs, that are saved in the database (as well as the IP address). The user can also start or stop the audio synchronisation from any page on Bulb Bop. All of these make up the primary control interface.



Figure 29 - Bulb Icon with User prompt to toggle bulb

### “The system should offer multiple lighting modes.”

Initial work was done to allow changing coloured bulbs, and this is still present in the code repository, but it is not currently implemented. This requirement should be a priority for any further iterations of Bulb Bop.

Bulb Bop has met three-quarters of the initial system requirements; however, substantial strides have been made towards the final requirement. Examples of this are API endpoints that can change the colour of capable bulbs and registration packets that are still present in the code repository. Not only this, but the code repository has been created in such a way as to make future development as easy as possible.

As such, the final evaluation of this project's success is that it was indeed a success, with the caveat that future development is needed before it will fulfil the full user requirements, but for the limited time given for the project, it has been developed as much as was possible.

## **Reflection of Process**

The software development process of Bulb Bop was well planned out, initial research and software development proceeded well. As the project progressed, the development became less structured, it still progressed at a healthy pace despite this. As mentioned previously, there was a significant amount of enforced structure through GitHub. This saved a significant amount of work in development, from allowing the maintenance of a working git branch while working on a development branch, to nightly tests that showed issues extremely quickly.

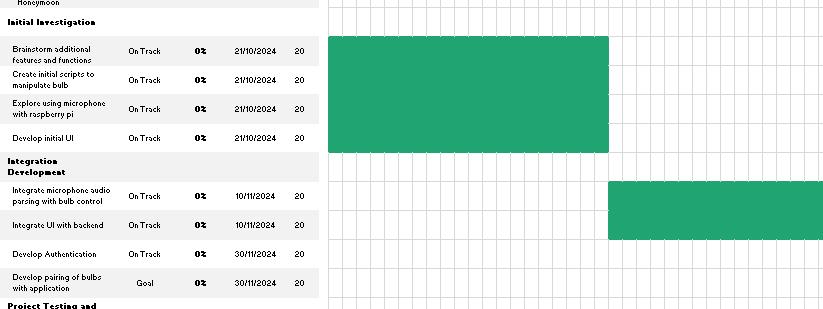


Figure 30 - Start of Gantt Chart

One part of the process that the development of Bulb Bop significantly benefited from was the early addition of testing. While it was also an example of the development of Bulb Bop becoming less structured, as testing was planned to be implemented in the final few weeks of the development timeline, it also massively benefited Bulb Bop by ensuring consistently working code and highlighting issues when they appeared.

The lack of structure that came to the development in its later stages was unhelpful in a few ways. Occasionally there would be some issues in the software that would go untouched for quite a while before it was handled. Often the main issue that needed sorted would be ignored while other less important issues were handled. Such as time being spent finding bulb icons, rather than working on the beat detection algorithm.

The development of this software would have benefited from a mid-way review of the process. This was not carried out, but would have allowed for a review of the user requirements and the current state of Bulb Bop. Thus, the final half of the development could have been planned out, and it would have prevented the loss of structure.

## **Evaluation of Software**

In this section, a fair evaluation of Bulb Bop will be carried out; this will cover an evaluation of the user interface, the backend of Bulb Bop, and the overall functionality of Bulb Bop. The final conclusion of this section is that while Bulb Bop is functional, and usable, it still requires work. Some work is required to expand the functionality, and to ensure a more usable environment for the end-user.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 31 - Screenshot of Index Page in light mode

As can be seen in the images of the user interface in this report (Figure 1, Figure 2, Figure 3, Figure 4, Figure 5, Figure 6, Figure 7, Figure 8, Figure 29 and Figure 31) the interface is very basic. There are a lot of dynamic icons, the ability to toggle dark and light mode, and multiple different views. Bootstrap 5 is used to manage the CSSs styling and the various elements of the HTML, there is minimal custom CSS which means that the user interface is very stable and cross compatible with many devices. Despite these the user interface is bland and at times unintuitive. This would require a developer with more experience in user interface design to remedy. However, as the main function of the user interface is to allow the user to activate the audio/light synchronisation the user interface still serves its function, just doing so un-aesthetically.

A screen shot of a computer program

AI-generated content may be incorrect.

Figure 32 - Index Endpoint Backend Code Screenshot

With regards to the backend of Bulb Bop, the code quality, functionality, and testability will be commented on here. In general, the code quality of Bulb Bop is excellent; it follows the PEP8 style guide and has enough comments to enhance the understandability of the function of the code without being excessive which would only add confusion. The code is functional, there is little delay in the code, with threading being used as often as possible to speed up response time. If there was an issue in the code, it would be the complexity of the beat detection algorithm, which makes it extremely difficult to test. It is impractical to simulate the input of an audio device into the algorithm due to this, as such the detection algorithm has had to be manually tested to ensure that it works as expected. Not only this, but the detection algorithm is the only part of the code base that suffers from lower quality code. For instance it has more nested for loops than is ideal.

A screen shot of a computer program

AI-generated content may be incorrect.

Figure 33 - Beat Detection Algorithm in code

Finally, regarding the overall functionality of Bulb Bop, as previously mentioned, Bulb Bop meets three of the four initial user requirements. The missed requirement was missed due to several factors, the biggest of which was the time needed to figure out how to manage multiple Wi-Fi bulbs of different capabilities. Despite this Bulb Bop does provide a good base for future development on this requirement.

## **Suggestions for Further Work**

This report is going to make five suggestions for future development, which will be listed in order of priority, from highest priority to lowest priority.

1. Complete work on the fourth user requirement.

This work should be prioritised above all others so that Bulb Bop can meet all of the user requirements. To achieve this requirement, work will need to be carried out to programmatically discover the type of bulb that is connected, to allow for packets to change depending on the bulb that is connected to ensure that colour change packets aren’t sent to a bulb without that capability, and for the user interface to be modified to allow the user to control this extra functionality.  
To reiterate, some of the needed work for this development is already present in the repository; there is a function that creates a packet to change the colour of a bulb.

1. Easy installation

The current state of Bulb Bop means that the user needs to have an understanding of how to run a Python script so that they can run the web application. This could be simplified to a single executable file. A suggestion for this development would be to look into the use of PyInstaller (16).

1. Redesign User Interface

With the addition of the final user requirement, Bulb Bop will have increased functionality. This should be reflected in the user interface when that is developed; however, as mentioned previously, the user interface thus far is rather basic. The user experience is also similar. Both should be updated to make Bulb Bop much more user-friendly, and something that people with little developer experience would have no difficulty using or understanding.

1. Expand usable bulb brands

Bulb Bop only supports Wi-Fi-enabled bulbs from the Wiz brand. This could be expanded to support many more brands of bulbs, which would allow Bulb Bop to be used by many more people without the need for them to purchase Wiz brand bulbs. This work would rely on the bulbs having an open API that can be used locally (cloud-based Wi-Fi bulbs often have increased latency).

1. Develop beat detection algorithm

The beat detection algorithm developed for Bulb Bop works, but sometimes it can miss a beat, and when starting or stopping a song, it can take a few seconds for the algorithm to catch up or change tempo. This could be improved upon by tweaking the algorithm or changing to a different programming language that better supports multi-threading and thus can run Bulb Bop faster.

## **Conclusion**

To conclude this report, Bulb Bop is a limited success; it met three of the four main user requirements, only losing out on the fourth due to complexity and time constraints. Bulb Bop is robustly tested and well organised, aiding any future development that occurs. It meets PEP8 standards, has well-organised code, and the beat detection algorithm, upon which Bulb Bop was built, works. Not only this, but the GitHub repository in which the code is stored is robust, has continuous integration checks, and rules that will allow others to help in future development without fear of damaging the working system.

Bibliography:

1. Six J. JorenSix/TarsosDSP [Internet]. 2025 [cited 2025 Apr 16]. Available from: https://github.com/JorenSix/TarsosDSP

2. Schreiber H. hendriks73/jipes [Internet]. 2024 [cited 2025 Apr 16]. Available from: https://github.com/hendriks73/jipes

3. drewbatgit. About WASAPI - Win32 apps [Internet]. 2021 [cited 2025 Apr 19]. Available from: https://learn.microsoft.com/en-us/windows/win32/coreaudio/wasapi

4. WiZ Pro API Reference [Internet]. [cited 2025 Apr 19]. Available from: https://docs.pro.wizconnected.com/#introduction

5. Traub S. sbidy/pywizlight [Internet]. 2025 [cited 2025 Apr 19]. Available from: https://github.com/sbidy/pywizlight

6. aiXander/Realtime\_PyAudio\_FFT: Realtime audio analysis in Python, using PyAudio and Numpy to extract and visualize FFT features from streaming audio. [Internet]. [cited 2025 Apr 19]. Available from: https://github.com/aiXander/Realtime\_PyAudio\_FFT

7. McClean R. RyanMcClean/Realtime\_PyAudio\_FFT [Internet]. 2025 [cited 2025 May 3]. Available from: https://github.com/RyanMcClean/Realtime\_PyAudio\_FFT

8. Beat detection. In: Wikipedia [Internet]. 2021 [cited 2025 Apr 20]. Available from: https://en.wikipedia.org/w/index.php?title=Beat\_detection&oldid=1019885396

9. Beat detection algorithm – Parallelcube [Internet]. 2018 [cited 2025 Apr 20]. Available from: https://www.parallelcube.com/2018/03/30/beat-detection-algorithm/

10. labs R. What is Context-aware security? Adapting Security to Digital Era Challenges [Internet]. 2023 [cited 2025 May 4]. Available from: https://cyberpedia.reasonlabs.com/EN/context-aware%20security.html

11. Urquhart R. RyanMcClean/wizlight-music-sync [Internet]. 2025 [cited 2025 Apr 24]. Available from: https://github.com/RyanMcClean/wizlight-music-sync

12. black: The uncompromising code formatter. 2025.

13. Prettier · Opinionated Code Formatter · Prettier [Internet]. 2025 [cited 2025 Apr 24]. Available from: https://prettier.io/

14. Python Enhancement Proposals (PEPs) [Internet]. 2025 [cited 2025 Apr 24]. PEP 8 – Style Guide for Python Code | peps.python.org. Available from: https://peps.python.org/pep-0008/

15. Pylint 3.3.6 documentation [Internet]. 2025 [cited 2025 Apr 25]. Available from: https://pylint.readthedocs.io/en/stable/

16. pyinstaller: PyInstaller bundles a Python application and all its dependencies into a single package. [Internet]. 2025 [cited 2025 May 2]. Available from: https://www.pyinstaller.org/

Appendix:

* + - 1. Testing

(Tables are also provided in the excel file in the source code)

Test cases:

 Automated:























.

































Manual Testing:









* + - 1. External Code:

Realtime\_PyAudio\_FFT:

Original found at:

https://github.com/aiXander/Realtime\_PyAudio\_FFT

Modifications specific to this project can be found at:

<https://github.com/RyanMcClean/Realtime_PyAudio_FFT>

Beyond this all other code is either from libraries used (like Django), or self-generated, with sources for ideas referenced in this report.