Virtual Reality Meditation Visualisation

Test Plan Report

Simon Zaragoza-Dorwald, Rosie Bartlett, Lavish Bhojani, Dravin Gupta, Joseph Dunne, Callum Gray

January 31, 2023

Table of Contents

1 Introduction

1.1 Product Overview

For our software engineering project, we were tasked with designing a virtual reality meditation application. The system would provide a meditative environment in virtual reality (VR) and adapt based upon sensor data from the user. Our goal was for our system to be able to collect physiological data, from these sensors, for further analysis and study. This would then facilitate research for our client, Alexandra Cristea, into the effectiveness of using virtual reality and biofeedback for meditation.

1.2 Testing Overview

When testing, we wanted to ensure that the tests we produced were comprehensive but also thorough. We began by relating our testing back to the context of our project and through this, identified the three main parts to test: the sensors, the environment and the server. The nature of how we tested these three parts would be distinct.

The sensors, in the context of our project, was a polar H10 heart rate monitor, and a MyndPlay EEG headset. When thinking about these, we wanted to test that they could reliably transfer data via Bluetooth to the VR headset. Initially we had planned to perform black box testing by using the actual sensors, however they arrived a lot later than anticipated. Instead, when waiting for them, we started writing white-box tests, ensuring that for some mock data they would have the expected behaviour.

The environment is built using Unity Game Engine out of GameObjects, which contain a list of components or classes, relating to their functionality. Due to this, we used the black box approach of contract testing, where our oracles were based on the contracts that were given by these components. More specifically, these contracts related to functionality to correctly immerse the user in the environment and for certain objects to orbit them.

For our server, testing would be functional with its oracle being equal to a successful POST request to the server. Further down, we tested these POST requests and successfully obtained an output matching this oracle.

1.3 Testing Coverage

1.3.1 Unit Testing

Due to the variance in how different parts of our system were written, it seemed appropriate to define a ‘unit’ specifically for each.

For the sensors our initial plan was to program code to interface with the Bluetooth API. Through these tests, we would be validating that the received data was accurate and at a frequent rate. As such our units, in this case, were equal to the correct functioning of these sensors. However, as the sensors arrived later than anticipated, we first wrote branch testing to ensure the correct branch would be reached for a specific input.

When conducting the unit tests, we passed in mock values to try and imitate the sensors, testing whether they were coming in at an acceptable rate and whether the values were in an expected range. Both these tests were crucial to validation, as for our data to be usable in research by our client, we needed it to be both accurate, and quick enough for the environment to adapt in real time.

As the environment was programmed in Unity, we believed that specific classes would be appropriate units. Due to the limit on the number of unit tests, we had to carefully consider what were the most important. The first test we decided upon was the camera's functionality by verifying movement in the real world correctly corresponded to movement in VR. We chose this as a unit, because it was an essential part of our system. This is shown through it directly corresponding to our functional requirement FRE-2, which has a MUST HAVE priority.

Secondly, we wanted to test that the objects moved correctly. Our environment had been designed to be minimalistic and for it to react to sensor data, we first needed to test that it could change at all. Similar to before, this unit links to another functional requirement, FRE4, for the environment with a priority of MUST HAVE.

For the server, we programmed it in Rust. The units, therefore, could not be equal to classes, as Rust does not have classes. Instead, we focussed on the functionality. For the server we wanted it to be capable of two things: creating a new user and storing session data. We chose the following tests because they would verify that the server was performing in the expected manner. Moreover, such functions were in isolation, fast to test and easy to automate and repeat - which would assist with regression testing we would later carry out.

Test Case ID	Target	Type	Oracle
unit_test-01	`fn new()`	Black-Box	1. The server creates a new user and sends HTTP code 200(OK) and the new user ID 2. The server accepts the login and sends HTTP code 200(OK) and the corresponding user ID

unit_test-02	`fn submit()`	Black-Box	The server responds with a HTTP code 200(OK)
unit_test-03	`OVRCameraRig.cs`	Black-Box	Both the classes and functions correctly update the camera position and cause then rendering to happen close to real time
unit_test-04	`ObjectVibration.cs`	Black-Box	1. The session can call the prefabs from previous scripts and initialise object instances based on these object prefabs, the objects are visible to the user 2. The vibration differences can clearly be seen and the objects behave accordingly
unit_test-05	`check_frequency()`	White-Box	For each time-stamp, it should be detected if there is more than 10 seconds in time between transmissions of data. If there is, this should be logged for debugging.
unit_test-06	`check_value_range()`	White-Box	For each value it should be detected if it is outside the boundary range, which is then logged for debugging

1.3.2 System Testing

Below you can find a summary of the system tests we have produced. Within these tests, we tried to relate them to our requirements, and bridge the gap to the user acceptance tests. Due to the limit of six, we tried to make them cover as many requirements as possible and include both functional and non-functional ones. Through doing this, we hoped to encapsulate testing to ensure functionality and performance.

Test Case ID	Description	Nature of Requirement (Functional/Non-Functional)	Oracle
system_test-01	Data collection and storage to a profile	Both	Sensors should maintain a connection with the device running Rust server, frequently sending the collected user data. The data sent should be recorded to a session specific to both the user and the current date and time of collection.
system_test-02	Collected sensor data updates the environment, UI or audio	Both	3D objects, audio cues and textual prompts should change throughout the experience guiding a user through the meditation.
system_test-03	VR menus are used to log into the user account and access data	Both	The user should be able to login with correct (normal) credentials and then be able to see the progress of their previous meditation sessions. If the user provides incorrect (erroneous) credentials, it should not show any data saved.
system_test-04	Meditation experience can be paused, with relevant options to select from this	Functional	Pausing the experience should temporarily stop all visual and audio aspects of the meditation. All menu options should work including returning back

			to the experience where all stopped visual and audio aspects should continue as before.

1.4 Integration Testing

Building upon our unit tests, we deemed it necessary to include tests to support their functionality when combined.

For our integration strategy, we considered the possibility of using a big bang or a phased integration approach.

We believed our project was too complex to apply a big bang approach and there would be a high risk of different components not integrating properly. Furthermore, we worried we would not be left with enough time to properly carry out our desired tests. As a result, we adopted a phased integration approach. Using this, we would test in stages, before completing the program, allowing us to reduce the risk of multiple major issues occurring simultaneously. It also fitted well with the Extreme Programming (XP) methodology which we had adopted, by continuously allowing us to test throughout producing code.

After deciding upon phased integration, we then discussed our approach to integration testing, coming to a consensus on applying a sandwich approach.

From a bottom-up approach, we liked the idea of being able to work independently on different components. Despite collaborating and discussing at regular intervals, we found programming independently had been working for us and liked the idea of continuing this. On the other hand, our project was particularly reliant on our components integrating well together. We feared that a bottom-up approach might focus too much on the individual units and wanted to emphasise our testing on the integration. In addition, for some parts of the project, such as the transmission of data from the sensors, we would be relying on pre-defined code - which presumably would work fine - and so we didn’t want as big an emphasis on finding bugs for this.

A top-down approach would suit our project by allowing us to focus on the architectural design, which we were prioritising due to the many parts we needed to integrate. Furthermore, testing from the top would allow us to produce a prototype quickly, enabling quick feedback on our project from the client. Despite this, we were concerned about completing all the testing on time. Having to write many stubs to be able to test would make the process of testing much slower.

Ultimately, we believed it was in our best interest to combine both of these approaches with a sandwich approach. We were confident that this would allow us to focus on the architectural

design whilst also enabling us to complete the tests within time. In addition, such an approach might allow us to deliver prototypes of the system to the client, and give us the opportunity to change and adapt our project from their feedback.

1.5 User Acceptance Testing

User acceptance testing (UAT) is testing that satisfies our client's scope and requirements, which can differ from our requirements as developers. For these tests, the customer acts as the test oracle so they can be confident in the product once it is handed over to them no matter our interpretation of their needs. This ensures that the product meets our client's set of acceptance criteria by using "real-world" testing.

The majority of the UAT will be pilot testing which involves everyday use of the meditation visualisation system which is less formal than benchmark or parallel testing. Given that our client has access to other versions of tailor-made virtual reality meditation visualisation systems, they may run parallel tests. This would compare our product with these other products side-by-side addressing functionality differences.

2 Test Cases

2.1 Unit Tests

Test Case ID	unit_test-01
Description of test	Test the server's ability to create a new user and let existing users login
Related requirement spec/design spec details	None
Prerequisites for test	An internet connection and configuration of an API key cookie whose value matches the cookie value in the configuration file

Test procedure	1. To create a new user, send a POST request with JSON data to /api/new :
	`fn new() {`
	`  let client = Client::tracked(launch()).expect("valid rocket instance");`
	`  let resp = client`
	`  .post(uri!("/api/login"))`

	.cookie(Cookie::new("key", API_KEY))
	.body(
	object! {
	"uname": "test",
	"pin": 1234
	}
	.to_string(),
	)
	.dispatch();
	assert_eq!(resp.status(), Status::Ok);
	assert_eq!(resp.into_string(),
	Some("000".to_string()));
	2. To login in, send the same HTTP request with the same JSON data to /api/login

Test material used	Laptop connected to the internet
Expected result (test oracle)	1. The server creates a new user and sends HTTP code 200(OK) and the new user ID 2. The server accepts the login and sends HTTP code 200(OK) and the corresponding user ID
Equivalence Classes	Normal: We create a new user, and then log in with these user details Boundary: We create a new user, but when trying to log in, we add an additional character to the password
Failure Severity	Serious
Comments	The server and tests are written in rust. API key cookies are stored in a config.toml file
Created by	SCZD
Test environment(s)	Windows 10

Test Case ID	unit_test-02
Description of test	Test the server's ability to correctly store session data
Related requirement spec/design spec details	FRD-1, FRD-2, FRD-4

Prerequisites for test	An internet connection and configuration of an API key cookie whose value matches the cookie value in the configuration file

Test procedure	1. Send session data via a POST request in the general format below to /api/submit:
	`fn submit() {`
	`  let client = Client::tracked(launch()).expect("valid rocket instance");`
	`  let resp = client`
	`  .post(uri!("/api/submit"))`
	`  .cookie(Cookie::new("key", API_KEY))`
	`  .body(`
	`  object! {`
	`  "user_id": "000",`
	`  "time_start": 1671080669,`
	`  "hr_data": [`
	`  { "time": 1671080702, "pulse": 40 },`
	`  { "time": 1671080712, "pulse": 35 }`
	`  ],`
	`  "gaze_data": [`
	`  { "time": 1671080702, "yaw": 0, "pitch": -3.9 },`
	`  { "time": 1671080707, "yaw": 12, "pitch": 55 }`
	`  ]`
	`  }`
	`  .to_string(),`
	`  )`
	`  .dispatch();`
	`  assert_eq!(resp.status(), Status::Ok);`
	`}`

Test material used	Laptop connected to the internet
Expected result (test oracle)	The server responds with a HTTP code 200(OK)
Equivalence Classes	Normal : The sensors send complete session data Erroneous: We send a post request without a body
Failure Severity	Intolerable
Comments	The server and tests are written in rust. API key cookies are stored in a config.toml file

Created by	SCZD
Test environment(s)	Windows 10

Test Case ID	unit_test-03
Description of test	Test camera behaviour and reaction to the user
Related requirement spec/design spec details	FRE-2
Prerequisites for test	Functioning VR headset

Test procedure	1. Looking around and update the field of view using
	`void Update() {`
	
	`public float smoothTime = 0.3F;`
	
	`mousePos = new Vector3(Input.mousePosition.x, Input.mousePosition.y, 0); Vector3 targetRotation = mousePos + new Vector3(0, 0, 0); transform.rotation = Quaternion.RotateTowards(transform.rotation, Quaternion.Euler(targetRotation), smoothTime);`
	
	Making sure that the camera position is calculated accurately
	
	2. `Use UnityEngine.VR.InputTracking.Recenter()`
	To recenter the camera instantly, verifying quick rendering and POV adaptation

Test material used	Laptop with Bluetooth, Oculus Meta Quest 2
Expected result (test oracle)	Both the classes and functions correctly update the camera position and cause then rendering to happen close to real time
Equivalence Classes	Normal: The camera is moved at realistic pace, leaving the user plenty of time to explore around

	Boundary: The view direction is changed by a large degree, i.e. from the very right to the very left, leaving calculations for position, shader and the rendering to take longer
Failure Severity	Mild
Comments	Code from https://forum.unity.com/threads/unity-vr-style-camera-movement.748310/; Integrated in our own code We use a universal rendering pipeline in our project
Created by	SCZD
Test environment(s)	Windows 10, Meta system software

Test Case ID	unit_test-04
Description of test	Test the object initialization and movement in the environment
Related requirement spec/design spec details	FRE-4
Prerequisites for test	Functioning VR headset, Laptop with Bluetooth

Test procedure	1. Once the game is initialised, the objects are created by pulling existing prefabs:
	`newObj = GameObject.Find("ExamplePrefab");`
	`myClone = Instantiate(newObj, transform.position, transform.rotation);`
	2. Calling the class `ObjectVibration()` with different speed and intensity parameters and measuring their effect on the `GameObject()` class

Test material used	Laptop connected to the internet, Meta Quest 2

Expected result (test oracle)	1. The session can call the prefabs from previous scripts and initialise object instances based on these object prefabs, the objects are visible to the user 2. The vibration differences can clearly be seen and the objects behave accordingly
Equivalence Classes	Normal: A standard amount of objects initialise and obfuscate and vibrate around Boundary: There is a focus on one type of object, causing an increase in intensity and speed of vibration, possibly interfering with other game objects
Failure Severity	Disturbing
Comments	`ObjectVibration()` is a behavioural script for the game objects.
Created by	SCZD
Test environment(s)	Windows 10, Meta Quest system software

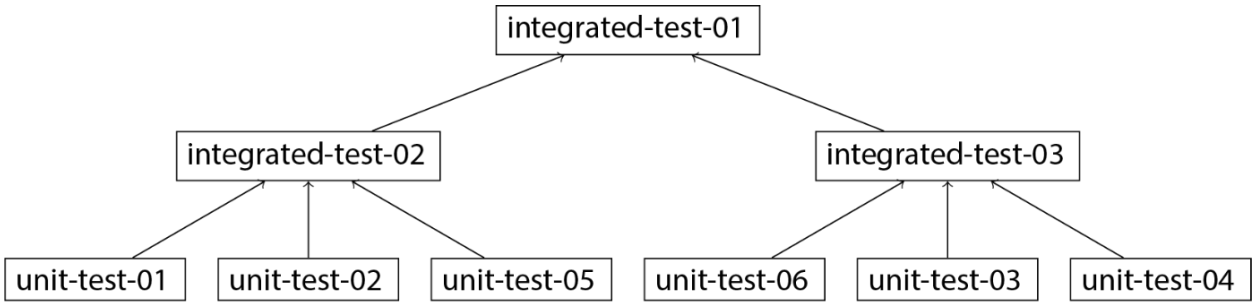
Test Case ID	unit_test-05
Description of test	Test to detect that 'mock' sensor readings are within a certain frequency
Related requirement spec/design spec details	FRS-2, FRS-3
Prerequisites for test	A laptop to process the data
Test procedure	1. Take a list of mock interval values for the EEG and Heart Rate Monitor data 2. Pass the time intervals to `check_frequency()`
Test material used	Laptop

Expected result (test oracle)	For each time-stamp, it should be detected if there is more than 10 seconds in time between transmissions of data. If there is, this should be logged for debugging.
Equivalence Classes	Normal : A set of values, passed in with less than 10 seconds between them. Boundary : A set of values, passed in with less than 10 seconds between them, except for one value, that is passed in at 10.01 seconds.
Failure Severity	Intolerable
Comments	Function written in Python
Created by	SCZD
Test environment(s)	Windows 10

Test Case ID	unit_test-06
Description of test	Test to detect that ‘mock’ sensor readings are within a certain range
Related requirement spec/design spec details	None
Prerequisites for test	A laptop to process the data
Test procedure	1. Take a list of mock values for the EEG and Heart Rate Monitor data 2. Pass each list through to `check_value_range()`
Test material used	Laptop
Expected result (test oracle)	For each value it should be detected if it is outside the boundary range for that specific sensor. If it is, then it is logged for debugging
Equivalence Classes	Normal : A set of values in the range 40 to 180 for heart rate and 0 to 100 for EEG

	Boundary : A set of values in the range 40 to 180 for heart rate with one value of 39, and a set of values of 0 to 100 for EEG with one value of 101
Failure Severity	Disturbing
Comments	Function written in Python
Created by	JD
Test environment(s)	Windows 10

2.2 Integration Tests



Further to deciding upon a sandwich approach and using phased integration, we designed how we would integrate the units. The figure above, shows how each unit test builds the foundation for the following integration tests. The three further integration tests are summarised in the tables below.

Test Case ID	integrated_test-01
Description of test	Testing that the environment changes, based on a stream of sensor data, that is then stored on the server
Test procedure	1. Use a stub for obtaining example sensor data 2. Pass these readings into Unity 3. Pass this from Unity into a very basic rust program
Child Tests	integrated_test-02, unit_test-01, unit_test-02, unit_test-05, integrated_test-03, unit_test-06, unit_test-03, unit_test-04
Expected result (test oracle)	By outputting the readings, all parts of the pipeline should produce the same result.

Failure Severity	Intolerable

Test Case ID	integrated_test-02
Description of test	Sensor data passed in can be stored for a given user
Test procedure	If we reach this stage from going top down then we will perform all steps, otherwise we will only need to perform step 3. 1. Make a basic server to store data 2. Make a python program that has fake sensor data 3. Pass data from the sensors, in python to the server
Child Tests	unit_test-01, unit_test-02, unit_test-05
Expected result (test oracle)	The sensor data in python, should be equivalent to the sensor data stored within the server
Failure Severity	Intolerable

Test Case ID	integrated_test-03
Description of test	The environment changes, for sensor data, and adapts to a users physiological state
Test procedure	If we reach this stage from going top down then we will perform all steps, otherwise we will only need to perform step 3. 1. Make a basic unity environment with a GameObject that alters in size and colour depending on sensor values passed in. 2. Make a python program that contains fake sensor data. 3. Pass data from the sensors, in python to Unity
Child Tests	unit_test-06, unit_test-03, unit_test-04
Expected result (test oracle)	The sensor data in python, should change the environment according to what is expected
Failure Severity	Intolerable

2.3 System Tests

Test Case ID	system_test-01
Description of test	Sensor data collected (from HR monitor, EEG headset and eye tracking) and saved to user's profile
Related requirement spec/design spec details	FRD-1, NFR-2
Prerequisites for test	Device running Rust server and connected to the internet. Working heart rate monitor and EEG headset connected wirelessly to the device running the Rust server.
Test procedure	1. User to correctly put on heart rate sensor, EEG headset and virtual reality headset 2. Connect the sensors to the device running the Rust server (either directly or through another Bluetooth enabled device) 3. Vary the data collected by the sensors (by performing a short meditation) and check that the data is saved to a specified user
Test material used	Laptop, Polar H10 heart rate monitor, and Myndband EEG headset.
Expected result (test oracle)	Sensors should maintain a connection with the device running Rust server, frequently sending the collected user data. The data sent should be recorded to a session specific to both the user and the current date and time of collection.
Failure Severity	Serious
Comments	Test procedure steps 1 and 2 must be done in that order as the sensors can only connect once they have been put on.
Created by	CG
Test environment(s)	Bluetooth, and Windows 10.

Test Case ID	system_test-02
Description of test	Sensor data collected (same as before) and update the 3D environment, any UI prompts and audio backing where necessary (personalisation)
Related requirement spec/design spec details	FRE-3, FRE-4, NFR-3
Prerequisites for test	Working heart rate monitor and EEG headset connected wirelessly to the virtual reality headset.
Test procedure	1. User to correctly put on heart rate sensor, EEG headset and virtual reality headset 2. Connect the sensors wirelessly to the virtual reality headset (either directly or through another middle device) 3. Take part in a meditation experience giving a range of readings from both sensors
Test material used	Laptop, Meta Quest 2, Polar H10 heart rate monitor, and Myndband EEG headset.
Expected result (test oracle)	3D objects, audio cues and textual prompts should change throughout the experience guiding a user through the meditation.
Failure Severity	Serious
Comments	Test procedure steps 1 and 2 must be done in that order as the sensors can only connect once they have been put on.
Created by	CG
Test environment(s)	Bluetooth, Quest system software, Unity, and Windows 10.

Test Case ID	system_test-03
Description of test	Use VR menus to log in to user account and access previous data

Related requirement spec/design spec details	FRD-4, FRE-1, NFR-4
Prerequisites for test	Data stored for at least one user that can be accessed, and a working virtual reality menu unit.
Test procedure	1. Log in using the menu on the virtual reality headset 2. Access their previous meditation data stored to their profile using the menu
Test material used	Meta Quest 2, and laptop.
Expected result (test oracle)	Normal : The user should be able to login with correct credentials and then be able to see the progress of their previous meditation sessions. Erroneous : If the user provides incorrect credentials, it should not show any data saved.
Failure Severity	Disturbing
Comments	None
Created by	CG
Test environment(s)	Quest system software, Unity, and Windows 10

Test Case ID	system_test-04
Description of test	Allow meditation experience to be paused displaying a pause menu with relevant options.
Related requirement spec/design spec details	FRE-3
Prerequisites for test	Rough working meditation experience, and a working virtual reality menu unit.
Test procedure	1. Begin a guided meditation experience and pause the experience during the meditation 2. Use each aspect of the menu 3. Close the menu, continuing the meditation

Test material used	Meta Quest 2
Expected result (test oracle)	Pausing the experience should temporarily stop all visual and audio aspects of the meditation. All menu options should work including returning back to the experience where all stopped visual and audio aspects should continue as before.
Failure Severity	Disturbing
Comments	None
Created by	CG
Test environment(s)	Quest system software, and Unity

2.3.1 Regression Tests

During the testing process, it is likely that more faults will occur. Therefore, it is vital we perform regression tests to iteratively test and eliminate faults. Code used to test the units and systems is contained within our documentation so that we can reuse it to efficiently find new faults.

For this project we have chosen a partial regression testing strategy. This would allow us to focus on specific sections of the system that have been changed, rather than performing a full regression test on all components. We believed that this approach would be rigorous, yet more efficient. To determine which modules would be worth testing, we would follow the diagram in the integration tests section, by starting at the root node, and testing all the way down to the level of where the code was changed at. This would provide us with a good estimation of what modules would likely have been affected.

2.4 User Acceptance Tests

Test Case ID	user_acceptance-01
Description of test	The VR environment feels natural and ideal for mediation.
Related requirement spec/design spec details	FRE-2, FRE-4, NFR-1
Prerequisites for test	Functioning virtual reality meditation environment.

Test procedure	1. The user to take part in a full guided meditation experience 2. The user should focus on usability and overall experience rather than technical details
Test material used	Meta Quest 2
Expected result (test oracle)	The user should be able to fully immerse themselves in the 3D virtual reality environment and comfortably follow through the process of meditation.
Failure Severity	Mild
Comments	None
Created by	LB
Test environment(s)	Quest system software

Test Case ID	user_acceptance-02
Description of test	Virtual reality experience is responsive, natural and tailored to the user
Related requirement spec/design spec details	NFR-2, NFR-3
Prerequisites for test	Functioning virtual reality meditation environment.
Test procedure	1. The user to take part in a full guided meditation experience 2. The meditation will be adapted to the users proficiency level
Test material used	Meta Quest 2
Expected result (test oracle)	The user should be able to complete the session. They must not feel distracted by elements in the scene. The 3D experience and menus should feel smooth and should not have a negative effect on the user's meditation.

Failure Severity	Disturbing
Comments	These 3D virtual reality objects and elements are there to enhance the user's session rather than feeling distracted or irritated.
Created by	LB
Test environment(s)	Quest system software

Test Case ID	user_acceptance-03
Description of test	The data recorded is presented in a useful and easy to understand form.
Related requirement spec/design spec details	FRD-1, FRD-3, NFR-4
Prerequisites for test	Functioning virtual reality meditation environment and session data store.
Test procedure	1. The user will put on the heart rate monitor and EEG headset 2. The user will participate in a meditation experience following the guidance 3. Upon finishing the experience, the user will view their data
Test material used	Meta Quest 2, Myndband EEG headset, Polar H10, and Laptop
Expected result (test oracle)	The session ends with a presentation of data collected throughout the meditation practice in easy to understand compilation. This will be in a visually pleasing way with graphs.
Failure Severity	Serious
Comments	This gives the user scope for self analysis.
Created by	LB

Test environment(s)	Quest system software, Windows 10, and Bluetooth.

Test Case ID	user_acceptance-04
Description of test	The meditation visualisation experience is personalised to each user using live data from sensors.
Related requirement spec/design spec details	FRD-2, FRD-4
Prerequisites for test	Functioning virtual reality meditation environment and processing of the sensor data.
Test procedure	1. The user should wear all sensors and connect them properly 2. The user should start the meditation and focus on how the environment adapts to their calmness
Test material used	Meta Quest 2
Expected result (test oracle)	The user will see changes in the environment (3D objects, textual inputs and audio cues) to aid and guide their meditation.
Failure Severity	Disturbing
Comments	This was a main, recurring point from our client that they wanted to see in the system.
Created by	LB
Test environment(s)	Quest system software

Test Case ID	user_acceptance-05
Description of test	The raw data stored will be accessible by the client so that it can be further processed after handover for our client's research.

Related requirement spec/design spec details	NFR-5
Prerequisites for test	Meditation data stored to a user's profile.
Test procedure	1. All data shall be time stamped and well documented 2. Query the required data from the database
Test material used	Laptop
Expected result (test oracle)	Data should be accurate with minimal missing values. The data should be in an appropriate format so it can be processed automatically with large amounts of data entries.
Failure Severity	Serious
Comments	As our client is involved in research at Durham University, they stated that they would like to be able to use the data collected for their own research.
Created by	CG
Test environment(s)	Windows 10

2.5 Test Results

Where code was already finished, we completed tests outlining their results and, where applicable, any changes that needed to be made. System, integration and user acceptance tests have been intentionally excluded from this section since some units are not complete meaning so we cannot combine these incomplete units for the system and integration tests.

These tests were completed with code within our documentation so that it can be reused for regression testing.

Test ID	Outcome	Comments
unit_test-01	Test passed	Ran the server, and made a POST request to the API. Returned a code 200 (ok).

		When doing the boundary case, where we submitted a slightly wrong password, it responded correctly with code 401 (unauthorised response)
unit_test-02	Test passed	Ran the server, and made a POST request to the API. Returned a code 200 (ok). When passing erroneous data it responded correctly with code 400 (bad request)
unit_test-03	Test not written	Unfortunately, our team member who had completed the programming for the environment, has gone missing. We have tried to contact them, to get access to the environment which they have shown us during our meetings, but we have been unable to get a response.
unit_test-04	Test not written	Unfortunately, our team member who had completed the programming for the environment, has gone missing. We have tried to contact them, to get access to the environment which they have shown us during our meetings, but we have been unable to get a response.
unit_test-05	Test failed	Successfully ran the code to ensure a stream of mock data was passed for normal data. When passing erroneous data, it would not handle this correctly and treat it as normal data. This was rectified by detecting incorrect data and logging it appropriately.
unit_test-06	Test passed	Successfully ran the code to ensure a stream of mock data was passed and any bugs were correctly logged depending on whether the input was boundary or normal data.

3 Testing Context

3.1 Testing Environments

Our meditation experience will be run on the Meta Quest 2 virtual reality headset both during testing and by our clients after the product handover. The Meta Quest 2 uses a custom made operating system known as Quest system software, which is based upon Android 10. Initial set up the headset also requires the use of a smartphone. We will be using our own personal smartphones to set up the virtual reality headsets which are required to be running at least Apple iOS version 12.4 or Android version 5.0.

The meditation experience will be built in Unity (version 2021.3 LTS). A long term support (LTS) version of Unity was used as these are generally more stable than more recent, non-LTS versions. Development in Unity was specifically requested by our clients so that after the product handover they can edit the source code if needed to further suit their research. During testing, Unity will be connected to the headsets to allow for effective testing as we can easily debug any faults. For the product handover, the experience will be deployed to the headsets so that no connection will be needed.

We will use physical sensors to collect data from the user to personalise and record progress in the meditation experience. One sensor is the Polar H10 heart rate monitor. This sensor goes around the user's chest and connects to another device wirelessly using Bluetooth. The other sensor to be used is the Myndplay Myndband Electroencephalogram (EEG) Headset and also connects wirelessly to another device. These sensors will be used in our testing environment to iteratively measure and improve the virtual reality meditation experience. These sensors require a user to wear them to collect readings.

Our server storing user login and meditation session data is run on the Rust programming language. This requires a computer to run it which will need to have at least Windows 10 or macOS 11.

We also require a device to connect wirelessly to our two data collection sensors. This is likely to be the same machine that is used to run the Rust server. Therefore, it has the same requirements of at least Windows 10 or macOS 11 with an additional requirement that the device is Bluetooth enabled.

3.2 Failure Severities

The range of failure severities of our tests are outlined in the table below going from Mild which is the least severe to Intolerable which is the most severe.

Severity Level	Description/Examples	Relevant Tests

Mild	Slight visual errors: misspelt words, UI/UX visual errors, slight visual discrepancies in virtual reality environment.	unit_test-03, user_acceptance-01
Disturbing	Occasional loss of data. Incorrect data shown to the user.	unit_test-04, unit_test-06, system_test-03, system_test-04, user_acceptance-02, user_acceptance-04
Serious	Loses user data. XML database transactions regularly fail, visual failures in the virtual reality environment.	unit_test-01, system_test-01, system_test-02, user_acceptance-03, user_acceptance-05
Intolerable	Database corruption/loss, failure in virtual reality environment causing discomfort to the user.	unit_test–02, unit_test-05, integrated_test-01, integrated_test-02, integrated_test-03
Infectious	Causes the system, and other connected systems, to shutdown. Necessary to fix as it prevents any functionality.	None

Mild failures should be addressed before the product handover however they negligibly impact the functionality of our program. These failures can cause slight misunderstandings for the users but not to the extent that it hinders progress. Therefore, fixing these issues is not vital for the operation and should be prioritised last during development.

Failures of **disturbing** severity can break some functionality of our system but will not affect the user's overall progress in the experience. These failures might lessen the user's immersion and enjoyment of the meditation experience. These failures will have low to medium priority as could have small negative impacts on the program.

A **serious** failure causes significant functional requirements to break. This can have a more severe effect on the user's experience, making the user less relaxed, possibly giving misleading or incorrect results for the meditation data. These failures have a medium priority as they can affect the data produced from each meditation session and can significantly affect the user's experience.

Intolerable failures can cause major loss of functionality across several areas. This can cause the majority of the functional and non-functional requirements not being met. These failures have high priority as failing to rectify these issues will make the meditation experience unplayable.

An **infectious** failure causes significant damage to both the system it runs on and other interconnected systems. This means it could damage the user's hardware, software or other third party services. These failures have the highest priority to mitigate damage done to other systems.