Phizi

Testing Document

Members:

Batya Bialik, Itay Mor, Aviehu Yudilevitch, Hila Klein

Contents

Chapter 1

Testing functional Requirements

	Requirement description	tests
1	Sampling the client's webcam	User test
2	Extract skeleton (x, y, z for 33 landmarks) points from the sampled video	User test
3	Draw a 2D skeleton on a user's video input in real time	User test
4	Calculate angles for specific joints in 3D space	Unit test
5	Calculate angles for specific joints in 3D space	Unit test
6	Calculate speeds for joints in 3D space	Unit test
7	Send each frame to the server as a JSON	User test
8	Display 3D Skeleton in real time and according to the user movements for testing purposes	User test
9	Store all the user's information (last recordings, login info, treatment progress…)	User test
10	Provide Client with all the exercises needed for a specific user using existing data and data received by the user	User test
11	Registration to the system	Acceptance Tests, System Tests
12	Login into the system as admin\user	Acceptance Tests, System Tests
13	Calibration of user data in first use	Acceptance Tests, System Tests
14	Logout from the system	Acceptance Tests, System Tests
15	Adding poses as admin	Acceptance Tests, System Tests
16	Admin can view details on the clients	Acceptance Tests, System Tests
17	Admin can assign a goal to clients	Acceptance Tests, System Tests
18	Admin can add a new session	Acceptance Tests, System Tests

Chapter 2

Testing non-functional Requirements

Performance -

We measured the frame rate of the system and ensured that it meets the minimum requirement of 30FPS. This was done by running the system under various conditions and measuring the frame rate.

Reliability -

To test the reliability of the pose-matching feature, we will conduct tests with users of different physical attributes to ensure that the system can accurately match poses regardless of the user's physical attributes.

Usability -

To test the usability of the system, we conductd user testing with children to ensure that the system is easy to use and understandable for them.

Format -

To test the format of the points output, we verified that the output file is in JSON format and that it contains the expected data.

Robustness -

In addition, we have tests that check the robustness of our Server with some load tests that run for 10 seconds and send 10 requests per seconds so we can tell if our system can handle a lot of traffic.

Chapter 3
Test-Driven Development

We employed the TDD strategy, specifically focusing on creating unit tests. This approach proved to be highly beneficial in identifying and resolving bugs within the program. By writing tests before writing the actual code, we were able to establish clear expectations for the behavior of each component.
As we implemented the code, running the unit tests helped us catch various types of errors, such as type errors, boundary cases, and logical inconsistencies. The tests acted as a safety net, highlighting any deviations from the expected outcomes. This allowed us to quickly pinpoint and rectify the issues, ensuring the program's correctness and reliability. Moreover, the unit tests provided a valuable means of regression testing, preventing regressions in previously working code during later stages of development or refactoring. Overall, the adoption of TDD with unit testing greatly enhanced the quality of our program by detecting and resolving bugs early in the development cycle.

Chapter 4
Testing the user interface

We tested our application’s UI manually, we interacted with the application as an end user would. We navigated through the different screens and menus, clicked on buttons and links, entered data into forms, and performed other actions to ensure that the UI behaved as expected. We checked that the layout and design of the UI were consistent and visually appealing, and that all elements were properly aligned and positioned. We also verified that the UI responded correctly to different screen sizes and device orientations. Throughout the testing process, we took note of any issues or inconsistencies that we encountered and tried to fix them as good as possible.

Chapter 5
Testing build, integration & deployment

Each time changes are pushed to GitHub, we are running all the tests written in the codebase to ensure a clean build, error-free functionality, and seamless integration.
GitHub allows for the execution of unit tests within the CI/CD pipeline. These tests validate the behavior and correctness of individual components, ensure they function correctly in isolation, and can be particularly useful in identifying integration issues at a granular level.

Our application is hosted on a server and not installed locally, so the focus is on ensuring that the server environment meets all the requirements.
To verify that we perform several checks:

1. Resource Availability: We validate that the server has adequate resources such as CPU, memory, and disk space to accommodate the application's requirements.
2. Dependency Checks: We verify that all the necessary dependencies and libraries required by the application are present in the server environment.
3. Compatibility Testing: We conduct compatibility testing to ensure that the server's operating system and any other required software components are compatible with the application. This helps prevent conflicts or compatibility issues that could affect the functionality of the app.
4. Error Handling: We implement robust error handling mechanisms to capture and handle any issues that may arise during the process. This includes logging relevant error messages, providing informative error notifications, and performing rollback actions if necessary.