Testing Document

Testing for our project can be broken into two major components: interface testing and back-end logic testing.

Interface testing comprised of aggregating possible user decisions when toggling between different states of the game. An example would be going from the title state to the play state, the pause state, the end-game state, and the title state back to the title. Nam, Rishi, and I used a glass-box approach to testing, understanding that key Raylib and Raygui components were implemented and testing those in the demo runs. Specifically, a sample demo run conducted would be as follows: Choose a song from the start menu, ensure that score, combo, and number of times each of the four current keybinds pressed is 0, click notes at varying levels of accuracy keep track of them manually, resetting the combo often, pause the game in the middle, change key bind for all four notes in settings, play again but hit no notes and ensure that beat of the song is in sync with notes falling down. When the game is over, ensure that accuracy approximately matches with manual derivation, go back to the title screen, select a new song, and make sure all metrics and changes in settings have been reset. This was just one playthrough example, and multiple iterations and paths were followed according to our NFA-styled construction of the state machine for this game. We believe this interface testing approach demonstrates the correctness of the front end because by modeling our testing using an automata-style construction, as we had seen in the lecture, we were able to map out possible playthroughs and interactions they would have with our GUI and test them. We also let other members of our team play the game so those unfamiliar with the exact GUI components would have an unbiased playing experience. Though the interface testing was done manually, we still believe it was grounded and formal enough to provide sufficient confidence in its correctness.

Testing the backend was done primarily through line coverage testing after initially having each developer in their respective modules do in-progress, in-line testing. We employed Bisect to cover the implementation of functions in our compilation units. The specific modules tested were Beatmap, Button, Column, Keybind, Note, Sprite, StateMachine, and Utils (Utility functions). The utility functions were the most straightforward and used randomized testing through QCheck since they were mainly in place to assist with other modules' functionality, like

finding the distance between two points helped in knowing when a button was being pressed. The test cases for Beatmap were developed using glass-box testing because we wanted to achieve maximal line coverage in this module as it handled the synchronization of the beats of the song to the notes falling down. Some of the functions were not able to be unit tested since they require real-time features and audio playing, but we were able to robustly test the other functions (beatmap conversion, index incrementation, etc) using a glass-box approach. The Button module was also tested by looking specifically at the code since we felt it was important to ensure our test cases were testing edge cases, such as the overlap detection function. The tests for the Column module were done with a black-box approach because it heavily other compilation units and the main features to be tested in the Column module were its ability to store notes, update said notes' speed and position in the column, and recycle the notes (removing and resetting the list of notes) once said notes hit or passed the keys. Keybind and Note were made using glass box testing because each held variants as part of their representation type and functions associated with Keybind, like getting the key associated with to change or for Note in converting accuracy to a score necessitated full line coverage. Since the Sprite module is static in regards to the fact that it is the front-end display for the state machines like the title page or settings page, we used black box testing to create new sprite sheets and links for testing. Lastly, the state machine testing used black box testing since we wanted to emulate the game logic and create states to test to pass into our test functions. We needed to look into the full-back-end-only implementation of the StateMachine model, and we also manually designed a graph of how the values in states in the machine should look after a sequence of iterations and updates. We believe our back-end testing demonstrates the correctness of our program because not only did we check when playing our game if the logic was as expected, but we also achieved a high line coverage percentage on back-end functions after we did a combination of glass-box, black-box, and QCheck testing.