SJLR FNAF Test Plan Document

Our Five Nights at Freddy's project is primarily made up of two major components within the code: the several monsters that roam around the map, and the user's interactions with the environment by the use of cameras, doors, and lights. As such, we partitioned the majority of our tests into two suites to focus on these major categories separately. As long as both of these modules operate as intended, we know that our game as a whole functions properly as well.

In order to test our monsters, we first created helper functions to initialize the four animatronic characters in a default environment in their usual starting location in the map. Having these helper functions made the process of simulating a level (or one night, in game terms) much easier for testing purposes. In addition to having a base constructor for the monsters, we also added a way to create a randomized monster such that its name, location, and movement cooldown were all set to random values (within acceptable bounds). As FNAF is a very fast-paced game, we thought it was essential to make sure that our monsters' code would continue to function under any circumstances.

With this setup, we started our monster test suite by testing the many initial variables that each monster keeps track of, such as its name and starting location. We did so by initializing a monster with set values and ensuring that our constructors appropriately attributed the correct initial values to each variable. We dedicated the rest of our monster testing to their movement, which was the most complex component. The monster's rate of movement is a function of many other elements in the game state, such as whether the generator is running or if the difficulty level is set high. In order to make sure our code ran smoothly under any combination of these conditions, we first isolated each of these variables and tested their individual effect on the monsters. We then ran full-night tests to ensure the monsters' behavior was consistent over longer time intervals.

While we as testers were aware of the game mechanics that dictated the monsters' movement, the tests we wrote in this section fall under black box testing as we were more concerned about the external behavior and the transitions that the monster made rather than its internal calculations. We also used QCheck to check that certain invariants of the monsters were never broken over any interval of time, like having a positive location, not entering the office if the door is closed, and moving faster if the generator is on under certain conditions. Because we based these tests off of general specifications and not the implementation of the functions, these tests also fall under the black-box testing category.

In testing our game state and user interactions, we first had to ensure that our system for accepting user commands was reliable. We evaluated these mechanics by trying to overload the game with too many commands or inserting invalid combinations of inputs, intentionally testing our system for processing and validating user input from a glass box testing approach. We then concluded our game state testing with the generator, which is controlled by the user. We simply made sure that it could be toggled on and off correctly and that it had the intended effect on the user’s battery when used, since we had already tested its effect on the monsters.

Most of the game tests were glass-box tests. A notable example is the tests for toggle_generator. Given the numerous if statements, we made sure to include many battery levels (below 90, at 90, above 90 and below 100, and at 100) and the generator status to test every possible combination. We used QCheck for one game test: namely, apply_random_power_up. This function has a fifty-fifty chance of adding or subtracting ten from the battery, so we made sure that the battery level would never exceed 100 or drop below 0 as a result of this power up. In addition, because we wrote the test to target the specific implementation of this function, this is also a glass-box test.

Some functions, like random_events, game_loop, choose_difficulty, and start_or_tutorial, were not tested in OUnit, either due to the usage of print or user input. Unfortunately, testing user input and print outputs wasn’t feasible using OUnit, so we opted to play test. This approach allows us to observe the system's behavior in real-time and also gives us a better sense of what our users would experience as they play the game. This allows us to tweak certain game mechanics (especially pace) to adjust the game difficulty.

By partitioning our tests into two major suites focusing on monster behavior and game state/user interactions, we thoroughly validated each component of our game. The use of black-box and glass-box testing methods allowed us to verify both the external behaviors and internal logic of our functions. Property-based testing with QCheck helped ensure that invariants were maintained under various conditions. Play testing further complemented our automated tests by providing real-time feedback and ensuring a seamless user experience. Together, these methods provide validation of our game's reliability.