Continuous Integration

Methods and Approaches (and why they are appropriate)

Our chosen method for implementing our Continuous Integration was using Github Actions. We chose this method for many reasons, chief among them being how easily it integrates with our use of Github in the rest of the project, in addition to the large quantity of documentation available for the method. This saved us time organising our continuous integration between the development team, and used a platform that everyone was familiar with, reducing the likelihood of misuse or misunderstanding the continuous integration platform. Overall, using Github actions was the best option for our development team to use, due to their familiarity with the existing platform, and how our team was already working.

Github Actions allows us to easily and automatically test our codebase every time someone pushes an update to the Github repository, as well as giving us valuable feedback on how many tests have passed or failed, in addition to information on how much of our codebase is covered by our tests. This meant that the code being written was less likely to cause problems when merged, saving time in the development of the game, and making it less likely that time had to be taken fixing parts of the codebase that broke when merged. These features allowed us to implement our Continuous Integration efficiently and effectively, and minimised risks to slowing development time over the course of our project.

To ensure that there were minimal conflicts through the use of CI, we ensured that we only used 1 repository for our project as opposed to having multiple and merging them later down the line. This allowed for us to make slight changes over time and identify issues with the build as they cropped up. If multiple repositories were used, this would run the risk of the project being too difficult to salvage as a result of catastrophic build failure.

When CI picked up an issue, all production would be halted until the issue could be resolved. This way we didn't run the risk of having more to fix later down the line due to further issues stemming from the original. Furthermore, all changes, no matter how small, had to be committed to github for CI testing. Our reasoning for this was that different team members were running the project from different OS'; namely Windows and MacOS. This prevented the potential issue of the project compiling on some machines but not others.

During our research into CI we found an approach known as the plan-do-check-act (PDCA) cycle. This cycle saw us identifying a potential change to be made, implementing the change on a small scale, using the data provided by the implementation to determine whether it made any difference and finally implementing the change, assuming it was successful. If it was unsuccessful, the cycle restarted. This approach worked perfectly with our project, as changes to be made could be added slowly, with issues immediately being recognised and handled.

Our actual integration structure

Our continuous integration is built using Github Actions. It is triggered every time a merge is pushed to the main branch, and builds the whole project to check for any compilation errors. It also runs the j-unit tests, and if any fail, fails the build and sends a notification email to the group so that everyone knows to check the project for any issues and to not make any further commits until the issue is resolved. If the build is successful no email is sent, as this could become a hassle with the team getting constant emails with multiple changes being made a day.

When a build is run whenever a change is made, an artifact is produced: testsummary.md, a markdown file which specifies each test and whether they passed or not. This file was used by us whenever a build failed to identify exactly why the build did fail, at which point we were then able to make changes until the build was successful.

Our integration allows us to review test coverage, which gives us information on code coverage with our tests. This was largely beneficial, as not only did it help us to ensure that all parts of the project were tested to a reasonable degree, but also that specific parts of the project that were likely to run into issues had explicit coverage. Furthermore, we ensured that all of our requirements were individually tested

All of our tests were stored together in a file (fittingly stored under a directory named “test”), with these being tested through Junit. Information on the success/failure of each test was then accessible to us, which helped us identify problems at the earliest convenience.