Following the first assessment, one of our priorities for Assessment Two was implementing a form of Continuous Integration. After facing a particularly difficult integration between two critical systems in our first game, we wanted to create an environment which would allow us a more stream-lined process without as many obstacles.

The approach we made to establish a robust method of Continuous Integration began with collating all of the resources we had used and were going to use for the project into a single repository. This was important to our game's CI as it ensured all resources were conveniently accessible for the duration of development. Throughout the assessment, we made a conscious effort to keep the repository up to date as well as our own local branches of the project.

Communication, especially between people working on closely-related modules, was also emphasised whilst we completed the game. In a project like this one, it is always a danger that two independently developed components won't fit together. Keeping Continuous Integration in mind, we managed to avoid most conflicts within the implementation before they happened and it was much easier to mitigate the conflicts that did occur.

In order to keep our code standardised across all of our classes, we employed the use of the Google Java Style along with a system to check if our code fit this style. In order to accomplish this, we implemented an automatic procedure which, upon anyone pushing to the main branch of the repository, would run through the source-code and check to make sure we were using the Google Java Style. Although it seems superficial almost, this proved to be extremely helpful in the long run as reading through any of the classes/methods we did not understand was much easier than if we had not utilised the Google Java Style.

To complete our Continuous Integration, we used GitHub Actions to create several workflows to help automate the process of developing our game. We used GitHub Actions because with our repository being on GitHub already, it seemed the most logical and convenient.

On the directive to improve our practises from Assessment One, the first automatic process we actually implemented was one that would build a JAR file for us whenever we pushed to the main-branch of the repository. At first, this system was fairly rudimentary, with it building just the jar file for Ubuntu systems without performing any tests. However, this quickly changed.

After we managed to get Github Actions to generate a JAR file for us, we added the running of tests to the same workflow using JaCoCo. This was triggered whenever there was a push to the main branch, and allowed us to quickly see if any new builds broke previously established tests and uploaded the report as an artefact of the workflow.

We used Checkstyle to enforce the Google Java Style across our project's source code, and as such added it, as well as JaCoCo to the build.gradle file in our repository. This is again triggered by a push to the main branch, however, after the workflow was overhauled to build the project for Ubuntu, MacOS, and Windows, this changed from a simple push-to-main trigger to only trigger for the Ubuntu operating system. This is because the source code for all three platforms is uniform, so checking the source code all three times across the different platforms would only serve to slow down the build process.

This change to build the project for those three operating systems was done using a matrix in yaml, which allowed us to reuse most of the code already inside the gradle.yml file and adapt it to work for any member of that matrix.

The final component of our Continuous Integration pipeline is the ability to “release” a build. Unlike the others, this is not triggered by a simple push-to-main but instead only runs if a tag matching the “vX.Y.Z” format of a version number is present. When this happens, the “release” workflow will make 3 JAR files corresponding to each of the different supported platforms our game is designed to run on.