CI2

Cohort 1, group 9:
Chris Sewell, Fedor Kurochkin, Matt Durham, Max Peterson, Oladapo Olaniran, Wojtek Tomaszewski, Yuqi Fu.

Summary of approaches

We have adopted Continuous Integration (CI) principles into our project, automating numerous tasks to improve our efficiency as well as the quality of our product. This approach utilizes GitHub Actions and is triggered on every push and pull request to our repository. GitHub Actions were an appropriate choice for our project as it integrates natively with our version control system (GitHub). In addition, GitHub Actions allows us to use standardised Ubuntu runners that reset on each run, ensuring the Continuous Integration process remains stable.

We have implemented an automated build pipeline with dependency caching to ensure our build environment is fast, while remaining consistent and reproducible (rather than only working for some developers, or breaking irrevocably). We use the specialized "gradle/actions/setup-gradle" action to aggressively cache our dependencies and improve the build times for our pipeline, giving us faster feedback when problems occur.

Our GitHub Action workflows automatically run a series of tools on our codebase to ensure any problems a developer may have introduced are immediately caught. These include automated unit testing, static analysis, and code style analysis. We also analyse the testing coverage to ensure nothing slips past our tests.

Our CI solution is encapsulated within a single YAML configuration file that defines all the jobs and their steps to run. Our pipeline consists of two jobs which run in parallel, following a pull request or push to the GitHub code repository:

1. Job 1: Build & Analysis.
    - Input: The source code from our GitHub repository
    - Output: An executable .jar file of our game, and a JaCoCo HTML coverage report outlining the completeness of our unit testing
    - Process: In addition to compiling our game and uploading it, this job runs Spotless, PMD, and JUnit tests. Spotless and PMD are used to analyse code quality, and will cause the job to fail if they are not perfect. The number of violations can be seen in the log file of the GitHub action.
2. Job 2: Dependency Submission
    - Input: Our project's Gradle configuration files
    - Output: A dependency graph that is automatically submitted to GitHub's security API
    - Process: This job analyses the libraries our game uses and checks for security vulnerabilities. These can be seen in the "Security" tab on our GitHub repository.

Infrastructure Report

Our Continuous Integration workflows use GitHub actions and triggers based on pull requests to the repository. They utilise a disposable Ubuntu-based VM running on GitHub's servers to ensure a reliable build process. We use the Temurin distribution of JDK 17. JDK 17 is chosen as it is the same version as our project is configured to use. We use two jobs, which are both defined inside the .github/workflows/gradle.yml file in our Git source code repository.

We utilise the "gradle/actions/setup-gradle" step to set up the build environment as fast as possible, ensuring dependencies are cached so the running environment does not have to redownload them every time our action is executed.

Our pipeline runs two jobs in parallel for efficiency.

1. Job 1: Build & Analysis:
    a. The job executes "./gradlew build -x pmdMain" to build the program. We exclude pmd to ensure the build does not fail even if there are pmd violations as our development team have decided to ignore some PMD violations.
    b. Next, we use "actions/upload-artifact@v4" to upload the game in "lwjgl3/build/libs/*.jar" and the JaCoCo coverage report in "core/build/reports/jacoco/test/html/".
    c. Finally, we run "./gradlew check", to execute the PMD, Spotless, and JUnit analyses and tests.
2. Job 2: Dependency Submission
    a. We use the "gradle/actions/dependency-submission" action to generate and submit a dependency graph update to the GitHub security API.

For our testing system we use JUnit, which tests many methods in our code to ensure no bug is introduced.

PMD is used for static analysis, helping us to identify dead code and catch bugs early. Spotless is used for analysing our code style, which encourages us to keep our code clean and maintainable.

We have also enabled the GitHub dependency graph and have set up Dependabot alerts to automatically notify us when a security issue is found with a dependency our project relies upon.

Our CI pipeline publishes the jar file of the game as an artifact on GitHub after a successful build, making it available for download from our project repository. We have made the deliberate decision to avoid running the code analysis until after the build is completed and uploaded, as we often need to download the latest version of our game so that our testers can use it.

We have added JaCoCo to our CI pipeline, which is an automated code coverage library. This analyses our code coverage on every commit, and generates a HTML report which we have included on our website.