Continuous Integration

Part A:

In the development of the project, there were three main targets to fulfill under the bracket of Continuous Integration that were not otherwise done automatically by GitHub, those being automating the build, making the build self testing and making it easy to access the latest executable. Continuing from that, the build testing was broken down into two further segments: self testing of the code's functionality and its adherence to the google style document for java. To fulfil these requirements, four interlinked pipelines were implemented for the four different obstacles.

The first to be implemented would be the build pipeline, a pipeline to be called whenever code was pushed from local repositories to the central git repository, taking the input code and generating suitable builds for the latest versions that Github was aware of for Linux, specifically the latest version of Ubuntu.

Before release, the build workflow would then be updated to implement appropriate builds for the latest versions of Windows and MacOS respectively, allowing the program to be more accessible across multiple platforms. Additionally, the pipeline would then be appended to generate a JAR artifact, to create an easily downloadable executable for playtesting, customer feedback or general deployment.

The second implemented pipeline would be the test pipeline, intended to provide a quick and easy to access test of the created build without necessitating a separate trigger. The pipeline is to be triggered as part of the build process, taking the code repository as input and outputting a JaCoCo report to the user, an automatically generated report that quickly summarises which pieces of code have been used and where any errors were generated.

The third pipeline to be implemented is the code style check pipeline, likewise triggered by a within the build process. It follows a parallel process to the test pipeline, taking the same input at the same moment, but outputs a Checkstyle report rather than a JaCoCo report. Rather than checking for errors in the functionality of the code, the Checkstyle report will report on any deviations from the google style documentation, ensuring that the code created and accepted is easily accessible by other programmers.

The fourth and final pipeline is the release pipeline, which is triggered whenever a version tag is pushed to main. As this is still a push to the main git repository, the pipeline waits for confirmation of a successful build, before delivering a full release, containing an executable java executable and version file of the appropriate version tag.

Part B:

Since the project was developed using Github, it was decided that the method used to implement the Continuous Integration pipelines would be the Gradle system, hosting .yml scripts within the project hosted on Github, as well as the JaCoCo and Checkstyle systems for testing, with additional java scripts to support the process.

The build workflow was established within the default gradle.yml program, called each time a new push was made to the central git repository, running the build through the gradle-wrapper.jar file to create builds for the three operating systems allocated within a matrix, those being the latest versions of Ubuntu, Windows and MacOS that GitHub had available to it. This implementation utilises an updated version of the standard GitHub actions gradle.yml page, to correctly build from the correct version of Java and for the correct versions of the three operating systems.

The build pipeline was then be appended to generate a JAR artifact, to create an easily downloadable executable for playtesting, customer feedback or general deployment, which, while not focussing on ensuring that it was clear to be the latest executable, provided a functioning temporary way of accessing and running the latest build. This appendage was made using the Upload Build Artifact action provided by GitHub actions.

The testing pipeline was created as a subtask of the build job of the gradle.yml task, similarly using the Upload Build Artifact action, instead uploading a JaCoCo report. When called by as part of the build workflow, the pipeline accesses the JaCoCo plugin added to the repository using the build.gradle.yml file, delivering an automatically generated JaCoCo report to the user at the end of the workflow, providing a quick and accessible overview of the functionality of the code.

The checkstyle pipeline is implemented in much the same way, initialised within the build job. It utilises the Upload Build Artifact action to upload a Checkstyle report attached as an artifact attached to the build report, accompanying the JaCoCo and JAR artifacts created during the build action.

The release pipeline is created as a separate job within the gradle.yml file, and is triggered by a successful build, and cancelled during an unsuccessful build. In order to ensure a release is only made when intended, as opposed to regular builds, the gradle.yml document specifies that the job is only run when the object pushed is a tag in the format of a version number, with other pushes running the standard build pipeline alone. In order to create a successful release of the build, the job is given permission to write within the contents of the project, allowing it to create and update the executable held within the git repository. To implement the JAR artifact from the build pipeline into a complete executable, the Download Build Artifact action provided by GitHub actions was utilised, allowing the GitHub Release Action to utilise within the release accompanying the executable. The release is done threefold, once for each operating system in the matrix, with the version tag that triggered the pipeline attached to each release, clearly labelling the releases and making the latest executable easily accessible.