Continuous Integration

Group 27
BlackCatStudios
Jack Vickers, Hubert Solecki, Jack Hinton, Sam Toner, Felix Seanor, Azzam Amirul Bahri

Continuous Integration (CI) Methods and Approaches

The project makes use of one CI workflow file. This gets triggered when someone pushes files to the main branch of the github repository or when they make a pull request from the main branch. Other branches don't trigger the CI workflow as they are only used temporarily for members of the team to develop new features. When these features are finished, code is merged into the main branch. This triggers the CI workflow and a report is generated. This report, in the case of failures, shows where failures occur so code can be fixed before another push is made. We believe that our selection of methods for CI can easily track our version updates. For the case where the code does not work, we can backtrack by rebooting previous versions of the code and work with that. It would also be easier to debug given that we have the flexibility to revisit versions to achieve completeness and correctness of the code by meeting the requirements of the system and showing it in the tests for the project making it appropriate for our project.

Inputs and Outputs:

- **IntelliJ test coverage report** - The workflow file gets the pre-generated test coverage report from the “htmlReport” folder of the directory and uploads this when the pipeline is triggered. This is uploaded so that we can see how much code has been covered by the automated unit tests
- **Executable JAR of the game** - The workflow generates the executable JAR and uploads it when the pipeline is triggered
- **Checkstyle report** - The workflow also generates and uploads a checkstyle report when the pipeline is triggered. This is uploaded so that the quality of the team’s code can be checked. We have attempted to stick to the google java style guide however our code doesn’t strictly follow the style guide. E.g. imports. We tried our best to follow it for functions, classes and variables.
- **“Game” folder of the repository** - the workflow triggers a gradle build command with this as the root directory. This is done for different operating systems so that we know that the game would build correctly on each one. The gradle build command triggers all of the unit tests so we can check the workflow to see if any of the tests have failed.

CI Infrastructure

Our CI pipeline is implemented using GitHub Actions. In the main branch of the github repository, there is a folder named .github/workflows which contains a file named gradle.yml.

- This yml file contains instructions for the CI pipeline.

The start of the file defines when to run the workflow, which as mentioned previously is when there is a push to, or pull from, the main branch of the repository. It has been set up to allow version tags to be assigned to a push so we can keep track of different releases throughout the project. The workflow has two jobs: build and release which are discussed below.

Build Job:

- The build job is set up to run for windows, macos, and ubuntu. This ensures that the Gradle project can be built on these different operating systems, so development is not limited to only windows computers.
- Initially, the build job sets up the java JDK 11 environment, which is the environment which we have used to develop the game. It does this using the setup-java action.
- Then, it runs the chmod command to make the gradle command executable as this command is later used to create the executable JAR.
- After this, it builds the game from the “game” directory using the gradle-build-action. This automatically runs all the unit tests because of how the gradle project has been set up, so as previously mentioned if a test fails, this step will fail and we can see which test failed by checking the build trace.
- Next, the job finds and uploads the pre-generated IntelliJ test coverage report. This is only done for one OS run through of the job since the report will be the same for each one. We originally intended to have the CI workflow generate and upload a Jacoco test coverage report, however we experienced many issues with this and could not integrate it into our gradle project. Hence, we decided to use the pre-generated IntelliJ test coverage report as a workaround. One downside to this is that we have to update the test coverage report before a push for it to be up-to-date.
- After this, the executable JAR gets generated using the gradle “gradlew desktop:dist” command. This also only gets run for one OS run through of the job since libGDX projects are multiplatform so we can use one JAR file on windows, MacOS, and UbuntU.
- Next, the executable JAR gets uploaded so that it is included in the workflow information after the workflow is complete. This allows us to easily download the JAR and check that it works as intended. This step uses the upload-artefact action.
- Finally, a code checkstyle report is generated and uploaded, as mentioned previously. This also is only done for one OS run through of the job since the report would be the same for each OS.

Release Job:

- The release job only runs once per push/ pull. It runs after the build job if code is pushed with a version tag (in the form ‘v[0-9].[0-9].[0-9]’).
- Its first step downloads the executable JAR generated during the build job so that it can be included in the release. This uses the github download-artifact action.
- Its second job uses the github release action to upload the executable JAR file.