Continuous Integration Report

Group 8 Members

Zac Challis

Viktor Atta-Darkua

Dandi Harmanto

Harsh Mohan

Danny Burrows

Continuous Integration Methods and Approaches

Understanding the importance of the early adoption of continuous integration is key to producing well tested software in a timely manner. Our team was acutely aware of this and as such we made a point to implement continuous integration as soon as possible.

Our first action post-acquisition of Team 3's codebase was to implement some basic testing; allowing all of our team to gain familiarity with the code by writing simple tests and also set ourselves up for success in terms of full code coverage down the line. In order to fully capitalise on this our very next action was to set up continuous integration; meaning every subsequent alteration to the codebase henceforth would be backed by our ever growing testing suite.

We chose to integrate a full run of our testing suite with our development process using Continuous Integration. Our belief was that pushes to the dev branch should never break, as such we chose to do full runs of all tests every time a pull request was opened on the dev branch. This would ensure integrity of the code. The choice for the dev branch to be the target was unanimously agreed, the core of the decision being that release candidates would be picked from dev and successful picks would be merged with main. Therefore a guarantee that a release candidate is passing all tests is paramount.

Our CI Approach

We chose to use a customised Github Actions template originally designed for generic Gradle projects. This would provide the base of dependencies required for testing the project.

Using Github Actions we configured CI using the standard method, a yml file in the base of the repository. This file outlines all major aspects of the CI pipeline and serves as a good basis to outline our approach:

```
on:
  workflow_dispatch:
        inputs:
        git-ref:
        description: Git Ref (Optional)
        required: false
  push:
        branches: [dev]
  pull_request:
        branches: [dev]
```

First we specify the events on which we wish to run our pipeline, as outlined above, for our group this was push and pull requests on the dev branch.

Next the jobs themselves and the parameters and constraints are outlined.

```
runs-on: ubuntu-latest
```

We chose to run our test suite on ubuntu as this was recommended for headless LibGDX projects and it provided the most familiarity to our team for debugging.

The pipeline itself consists of five tasks:

- Setup JDK and libGDX headless environment.
- Build the project using the Gradle build system.
- Run the full test suite.
- Archive any previous testing reports.
- Deploy the report of the test's outcome for later analysis.

Building the project makes use of the **gradle-build-action** provided by github. Archiving and Deployment of testing reports make use of *JamesIves/github-pages-deploy-action@v4.2.5* to help link outputting the report with commit and pushing the report to the branch running the tests. With these extra actions combined into one we kept our CI implementation well organised, the combination of these elements allowed for a modular approach when using separate actions for storing testing results in a specific branch.

After a full run of the pipeline is complete the results are deployed to repo in the form of a human readable report for analysis by the team and the summary of the outcome is displayed as the github action’s success or failure badge. Upon failure of any tests in the suite, the team is automatically notified via an email so that speedy resolution of bugs can be achieved.