Continuous Integration

Team 5:
Kevin, Zuhur, Erin, Jack, Aidan, Ryan, Sam

Continuous Integration (CI) Method and Approaches

For our CI method and approaches we focused on keeping our integration efficient and reliable. To keep it reliable we reduced the likelihood of major bugs by integrating often. We made sure new features were integrated on new branches of our repository and then merged these into the main branch once they were complete. This stopped us from having commits that broke the main build. We tried to keep new features small and break large tasks into multiple steps. This was so a new branch wouldn't be active for more than 3 days. We then made sure we were pushing our local repositories to Github at least once every 24 hours. These two approaches were crucial as it made sure each developer was up to date and reduced the likelihood of conflicts. We also focused on keeping our commits minimal so if there were any issues they would be easier to debug.
When it came to collaboration we tried to keep developers working on separate features that wouldn't crossover. This made it easier to collaborate remotely as it reduced the chance of one person's code breaking someone else's. Then in our weekly meetings we would bring these features together so tests could be run and if there were issues it would be easy to collaborate to work on a fix.

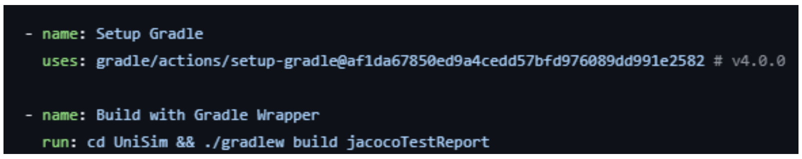
After we had discussed the practices our team would follow we looked into how we could improve efficiency using tools that were available to us. Using Github actions we set up automated building and testing. We made sure this was setup early on in the project as it would make implementation easier. Automated testing gave us many benefits, such as speeding up our debugging process, reducing the load on our local machines and improving test creation. Through test reports we were able to improve our test coverage and understand what tests we were missing as well as easily find where bugs in our code were. Automated builds allowed for members who didn't work on implementation to test the game without having to set up the coding environment. This allowed us to test the game's playability and made sure it aligned with our requirements.

These methods and approaches were appropriate for our project as we had a short deadline. We wanted to have our implementation completed by early December as members were going to struggle for time going into Christmas and we wanted to reduce remote working. Setting up CI early on allowed us to work efficiently with each other and kept bugs to a minimum. Our practices made sure that our code remained reliable as small feature changes reduced the risks of bugs. It kept us efficient as frequent integration reduced stagnation and regular integration allowed the project to progress steadily without developers falling behind. Our automated tests allowed us to stay confident in our code and reduced the time spent on manual testing. Finally it made it easy to be adaptable, if we had needed to work remotely these methods and practices wouldn't need to change in order for implementation to proceed.

Continuous Integration Infrastructure

For our CI infrastructure we used Github actions, Gradle and Jacoco. Github actions made it easy to automatically run the infrastructure we set up. This was because we could make it run after anything was pushed to the main branch. We could then easily access build and test reports from Github and these would also be shared with all team members. This allowed for us to have a single-source for our code and reduced confusion. We created a new workflow within Github actions and wrote what steps it should take using YAML.

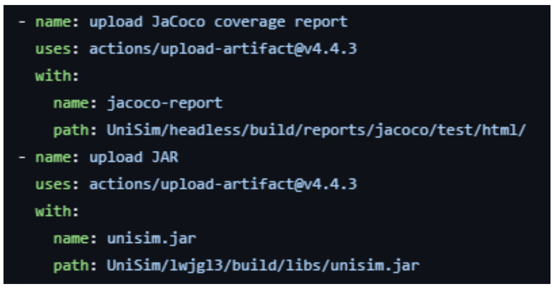
We set up automated builds using Gradle. Using the Gradle action we could use their wrapper to build the project using the "build.gradle" files in our project. This made it easy to add new dependencies without having to update our CI. The code would be built using a remote ubuntu machine so we didn't have to use our own resources. The code below shows how we set up Gradle actions and then run the gradlew build command using our main build.gradle in our top-level directory.
We also automated testing using Gradle and created reports using Jacoco. When the project is built our tests are run at the same time. After tests are run the Jacoco library creates a test report. This is also seen in the command below.



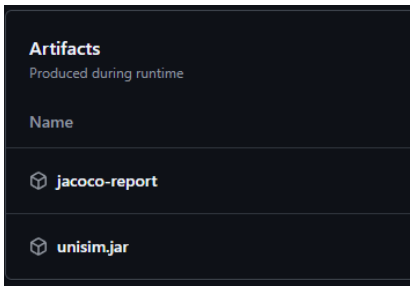
```
- name: Setup Gradle
  uses: gradle/actions/setup-gradle@af1da67850ed9a4cedd57bfd976089dd991e2582 # v4.0.0

- name: Build with Gradle Wrapper
  run: cd UniSim && ./gradlew build jacocoTestReport
```

After the build and test report have finished running, the files created are automatically uploaded as artifacts to that workflow instance. This is again using Github actions and by giving it the file path of the report and JAR file they are uploaded as shown below.



```
- name: upload JaCoco coverage report
  uses: actions/upload-artifact@v4.4.3
  with:
    name: jacoco-report
    path: UniSim/headless/build/reports/jacoco/test/html/
- name: upload JAR
  uses: actions/upload-artifact@v4.4.3
  with:
    name: unisim.jar
    path: UniSim/lwjgl3/build/libs/unisim.jar
```



Artifacts
Produced during runtime

Name
jacoco-report
unisim.jar

Finally we set up automatic releases. We did this with Github actions and made it so that if a developer made and pushed a new tag to the repository with the format vX.X.X where X is a number 0-9 then it would run a second part of the workflow to create a release. This would automatically upload the JAR file if the code builds successfully and create a release linked with the tag.

The entire workflow code can be found here.