CONTINUOUS INTEGRATION REPORT

Team 20: Tequila Mockingbird

Exam Numbers: Y3872854, Y3877651, Y3872779, Y3873031, Y3869788

Methods and Approach

First of all it is important to define what continuous integration is. Continuous integration is an agile method based coding philosophy that allows developers to meet requirements and code quality. The basis of continuous integration is to only make small changes to the code base and to make sure that the code being worked on is both the most up to date code and also code that passes all checks to make sure that the code will not break. Some pieces of software can aid this by automating these tests before the code is pulled, an example of this kind of software would be CircleCI. However in theory these tests don't have to be done automatically and can be performed manually.

Due to errors with Junit working with libgdx, automated testing proved difficult and therefore using services like CircleCI had to be delayed, due to delays in writing tests. Therefore when approaching the development of the we had to use a form of manual continuous integration by means of notifying the other developers in the team whenever a push was made to the Github Repository. The team pushed to Github every time a goal was achieved not trying to do multiple goals at once, to keep the differences in code as small as possible allowing for easier adaptation of the code if another person is working on the code at the same time.

Due to the problems experienced with libgdx only a few automated unit tests could be achieved and we added these to CircleCI so that any future updates to the code will have to pass these test in order to push to Github stopping the breaking of the code whenever a push to the repository is made

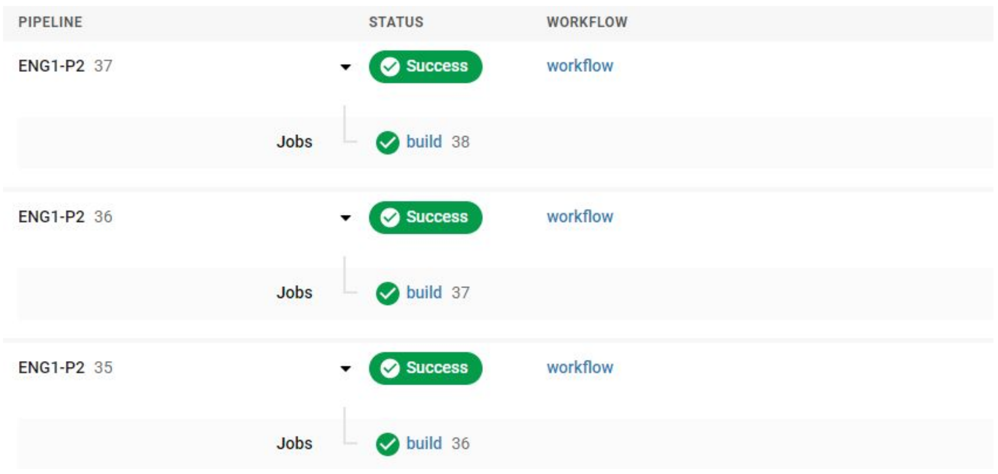
Typically games do not often implement unit tests as the games are typically a visual thing and issues are typically seen when playing the game as some things just can't be tested with a unit test. Due to this fact whenever the code was pulled from the Github repository we would run the code to check if there were any errors that you could see. If there were any errors the first priority is to go through the related code and re-run the game to see if the error was fixed.

There is however a caveat to these tests, as the actual algorithms being tested are based in classes that use libgdx the algorithms have been copied over with hard coded values into the test classes. As a result any change to the algorithms being tested with junit must be copied across to the test class that represents that algorithm and make the appropriate changes so that the tests are still valid and hold value.

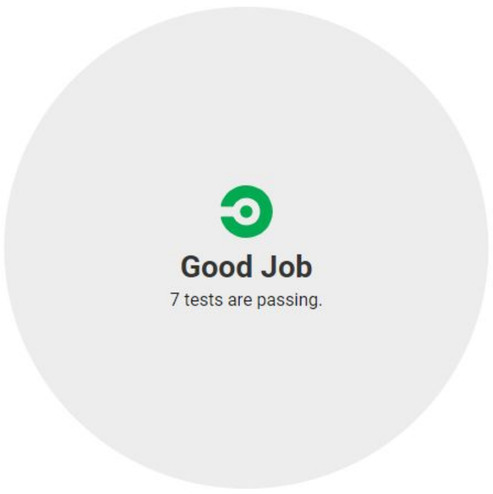
See below the infrastructure we put in place with the CircleCI config used for the project going forward and also a screenshot of the pipeline page showing the successful build and past test that automates tests to make sure the code doesn't break when new changes are made to the code.

```yaml
version: 2 # use CircleCI 2.0
jobs: # a collection of steps
  build:

    environment:
      # Configure the JVM and Gradle to avoid OOM errors
      _JAVA_OPTIONS: "-Xmx3g"
      GRADLE_OPTS: "-Dorg.gradle.daemon=false -Dorg.gradle.workers.max=2"
    docker: # run the steps with Docker
      - image: circleci/openjdk:11.0.3-jdk-stretch # ...with this image as the primary container; this is where all `steps` will run
        auth:
          username: mydockerhub-user
          password: $DOCKERHUB_PASSWORD  # context / project UI env-var reference
      - image: circleci/postgres:12-alpine
        auth:
          username: mydockerhub-user
          password: $DOCKERHUB_PASSWORD  # context / project UI env-var reference
        environment:
          POSTGRES_USER: postgres
          POSTGRES_DB: circle_test
    steps:
      - checkout # check out source code to working directory
      - restore_cache:
          key: v1-gradle-wrapper-{{ checksum "gradle/wrapper/gradle-wrapper.properties" }}
      - restore_cache:
          key: v1-gradle-cache-{{ checksum "build.gradle" }}
      - run:
          name: Run tests
          command: |
            ./gradlew test
      - save_cache:
          paths:
            - ~/.gradle/wrapper
          key: v1-gradle-wrapper-{{ checksum "gradle/wrapper/gradle-wrapper.properties" }}
      - save_cache:
          paths:
            - ~/.gradle/caches
          key: v1-gradle-cache-{{ checksum "build.gradle" }}
      - store_test_results:
      # Upload test results for display in Test Summary: https://circleci.com/docs/2.0/collect-test-data/
          path: core/build/test-results/test
      - store_artifacts:
          path: core/build/test-results/test
      - run:
          name: Assemble JAR
          command: |
            # Skip this for other nodes
            if [ "$CIRCLE_NODE_INDEX" == 0 ]; then
              ./gradlew assemble
            fi
      # As the JAR was only assembled in the first build container, build/libs will be empty in all the other build containers.
      - store_artifacts:
          path: build/libs
      # See https://circleci.com/docs/2.0/deployment-integrations/ for deploy examples
workflows:
  version: 2
  workflow:
    jobs:
    - build
```



PIPELINE		STATUS	WORKFLOW
ENG1-P2 37		✓ Success	workflow
	Jobs	✓ build 38	
ENG1-P2 36		✓ Success	workflow
	Jobs	✓ build 37	
ENG1-P2 35		✓ Success	workflow
	Jobs	✓ build 36	



Good Job
7 tests are passing.