Development Testing

Challenges

- The cooperative nature of our systems make it very hard to test individual systems and their methods. Systems like our **Front-End UI** (including front-end sub-systems like the **Level Handler**, **Customizations**, etc) and **Graphics Engine** make it very difficult to use unit testing frameworks (i.e. JUnit Testing).
- Continuous Integration was only partly achieveable as we were not able to test the deployability of the whole system (via the Google Playstore).
    - The application had not been in a state of publishing onto the Playstore until the very end of our project cycle.
    - This rendered a full continuous integration unhelpful in testing the system (at least for the most part).

Testing Strategies

For each system in the high-level architecture diagram (Architecture - **Figure 1**), we decided to test their robustness through these testing strategies:

- (1) Front-End UI - **APK Build Testing**
    - (1.1) Level Handler - **APK Build Testing**
    - (1.2) Customization - **APK Build Testing**
- (2) 3D Graphics Rendering - **End-To-End Testing**
- (3) Physics Engine - **JUnit Testing**

APK Build Testing

Android Studio provide a feature, allowing a applicaiton state to be integrated and built, resulting in an APK which can be distributed and ran to test integrated systems. This was extremely useful for the systems which were either intrinsically intertwined with several others in the application (1.1, 1.2) or systems with no concrete figures to actually test (interaction heavy systems likea UI (1)).

End-To-End Testing

This involves the use of APK Build Testing (defined above) but including system calls which help represent the state of the back-end system and its front-end visualisation. This was the only feasible way of testing the output of the graphics engine (2) as it is dependent on the state of the physics engine (3). This allowed us to check the consistency of the values between these 2 interacting systems.

JUnit Testing

JUnit Testing allows independent testing on the correctness of a system and its functions. Due to the independent nature and concrete values that is ouputted by the physics engine, it became the only system in the application that would benefit from unit testing. This allowed us to test the correctness of the methods calculating the physics interactions in normal and edge cases to prove the robustness of the back-end system.

Testing Table Sample:

Test	Testing Condition	Pass/Fail
@Test forceAppliedTest	- Given a set inputForce, currentAcc and a manually calculated expectedAcc - Use ForceApplied() method to calculate a resultantAcc - Assert the resultantAcc is the same as the expectedAcc	**Pass**
@Test CalcVelTest	- Given an inputAcc, currentVel and a manually calculated expectedVel - Use CalcVel() method to calculate a resultantVel - Assert that the resultantVel is the same as the expectedVel	**Pass**
@Test CollisionTrueTest	- Set the position, size and orientation of 2 objects in the scene and where they collide - Assert that the methods for collision detection set the collide value to true	**Pass**
@Test UpdateMovement Test	- Given an inputForce, initial acc, vel and pos - Cycle through the methods that dictate 1 frame's movement - Assert the resultant acc, vel and pos are the same as their expected values	**Pass**