Development Testing

Consistent with Test Driven Development, we decided on testing strategies for the various systems we had in our architecture diagram. These testing frameworks were decided on:

- **Front End UI** - APK Build test
- **Graphics Rendering** - Visual Testing
- **.CSV Level Handler** - APK Build Testing
- **Physics Engine** - JUnit Testing

APK Build Testing

Android studio provided an **APK Build** feature which allowed us to test things like functionality of certain *non-numeric* features, as well as robustness of integrated systems. This would compile the project state and export it as a testable APK which installed onto a connected mobile device (a terminal would display system messages describing the events occuring and processes running during application usage). This made more sense for complex integrated systems like the **.CSV Level Handler** and **Front End UI**. A full **APK Build** was the only feasible way of testing and monitoring these complex interactions via insightful system messages. As well as testing the functionality of the system's interactions, it also allowed us to test the useability factor of the application. This is equally as important to test when creating a UI and is only really feasible through active use of a testing build.

Visual Testing

Things like graphic output is difficult to emperically test and requires assets (such as 3D models) to be at the ready in order to test. Earlier on in development, where the 3D assets were not ready, we used visual placeholders in order to ensure the correctness of the renderer during the **APK Build Tests**. Through this, we were able to test that objects created in out graphical engine would be rendered in the correct position, orientation and size.

JUnit Testing

Using the JUnit test framework, method specific tests can created in a separate testing environment. Using a plethora of assertions, testing the correctness of methods in both normal and edge cases prove the robustness of a system and its methods. In back-end systems like our **Physics Engine**, which was implemented with methods that worked completely independent of external systems in the application, creating tests was easy and effective as inputs and expected outcomes could be calculated and could easily be instantiated in the testing framework.

Below is an example of some **JUnit** test cases used to test the **Physics Engine**:

Testing Table Sample:

Test	Testing Condition	Pass/Fail

Test	Testing Condition	Pass/Fail

@Test forceAppliedTest	`//Set the comparison parameters`
	`expectedAcc = (240f, 150f, 0f);`
	`//Set context`
	`result = forceApplied(initAcc, inputForce,`
	`movement.getMass(), movement.frametime);`
	`assertEquals(expectedAcc, result, delta);`
		Pass

@Test CalcVelTest	`//Set Comparison Parameters`
	`inputAcc = (240f, 150f,0f)`
	`expectedVel = (8f, 5f, -6f);`
	`//Set Context`
	`result = movement.calcVel(movement.getVel(), inputAcc,`
	`movement.frametime);`
	`assertEquals(expectedVel, result, delta);`
		Pass

@Test CollisionTrueTest	`//Set context`
	`InitPos = (2543f, 3500f,500f );`
	`movement.setPos(initPos);`
	`movement.setMovementSize(droneObject);`
	`obj = ApartmentsObject(4000f, 660f, 4000f, 1, 1, 1);`
	`movement.isCollision(movement, obj);`
	`assertTrue(movement.collided);`
		Pass

@Test UpdateMovement Test	`//Comparison parameters`
	`expectedAcc = (240f, 150f, 0f);`
	`expectedVel = (6f, 3f, -4f);`
	`expectedPos = (0.27f, 0.17f, 0f);`
	`//Set context`
	`movement.updateMover(expectedAcc, expectedVel,`
	`expectedPos, movement);`
	`assertEquals(expectedAcc, (movement.getAcc()), delta);`
	`assertEquals(expectedVel, (movement.getVel()), delta);`
	`assertEquals(expectedPos, (movement.getPos()), delta);`
		Pass