

## **Unit and Integration Tests**

This project contains many core features that interact with each other during the game. Each of these needed unit testing to ensure they behave correctly on their own, before being tested together in integration tests. There are approx 85 tests written in total to test our game.

### **1. Door mechanics**

What it is: Doors can open and close when the button is pressed.

Why it needs testing: To ensure that doors start as closed (initial state), behave correctly when toggled and respond to button interactions.

These tests are unit tests written in “DoorTest.java”

### **2. Button mechanics**

What it is: Buttons can be pressed or released, affecting connected doors and lasers.

Why it needs testing: Ensures button state updates correctly and affects other objects correctly.

These tests are unit tests written in “ButtonTest.java”, “DoorButtonTest.java”,  
“LaserButtonTest.java”

### **3. Laser mechanics**

What it is: Lasers are a type of punishment that can be on or off when button is pressed.

Why it needs testing: To ensure that laser behave correctly when toggled and respond to button interactions. They should also respond to player collision (level restarts when collision happens).

These are unit tests written in “LasersTest.java”

### **4. Player**

What it is: Player is implemented as a singleton to ensure only one instance exists. Player can collect power ups and will be affected by punishments.

Why it needs testing: It prevents multiple instances of player that could cause inconsistent game state like multiple players and would cause the game to crash. It is a design decision that must behave correctly, hence it is tested. Additionally, the effects of powerups and punishments need to be tested too.

These are unit tests written in “PlayerTest.java”

## 5. Crystal collection

What it is: Crystals are collectible items (rewards) and are required to open teleporters.

Why it needs testing: Crystals should be collected by player and the collected state should update correctly.

These are unit tests written in “CrystalTest.java”

## 6. Teleporter mechanics

What it is: Teleporter is the final destination of the player for each game and opens when all crystals are collected and time remains.

Why it needs testing: To ensure that teleporter only opens if conditions are met.

These are unit tests written in “TeleporterTest.java”

## 7. Turret mechanics and vision

What it is: Turrets are stationary enemies that have a range of vision and can also rotate. When they rotate, their vision rotates as well. If player comes in this range of vision, they are caught and dead (level restarts).

Why it needs testing: To ensure turrets have vision attached, properly detect the player in the vision range and vision updates correctly when rotated.

These are unit tests written in “TurretTest.java” and “AttacherTest.java”

#### 8. Alien mechanics and vision

What it is: Aliens are moving enemies that follow a certain path. The player dies if they collide with the aliens.

Why it needs testing: To ensure that aliens are following the path assigned, detect the player correctly and collisions behave as expected.

These are unit tests written in “AlienTest.java” and “UtilsTest.java”

#### 9. Radiation, Slime and Lasers

What they are: These are punishments that affect the player in negative ways.

Why they need testing: To ensure they apply on the player correctly and trigger changes in GameLogic.

These are unit tests written in “RadiationTest.java”, “SlimeTest.java”, “LasersTest.java” and “PunishmentTest.java”

#### 10. Timestop, Jetpack, Alien charm, invisibility

What they are: These are powerups that affect the player in positive ways.

Why they need testing: To ensure they apply on the player correctly and trigger changes in GameLogic.

These are unit tests written in “PowerupTest.java”, “JetpackTest.java”, “TimestopTest.java”, “AlienCharmTest.java”, “InvisibilityTest.java”

## 11. Collision

Player collision with crystal, powerups, punishment, button and teleporter is also tested.

These are unit tests written in “GameLogicTest.java”

## 12. Interactions tested – door and button, laser and button

What they are: Door and laser work along with button.

Why it needs testing: To ensure the connection works and disables collision

These are integration tests written in “DoorButtonIntegrationTest.java”

## 13. Crystal collection interaction

What it is: Collecting crystals updates the remaining crystal count in the game logic

Why it needs testing: collecting crystals should remove them from the environment, reduce the remainingCrystals counter and affects condition of opening teleporter.

This is an integration test written in “CrystalIntegrationTest.java”

## 14. Power ups and Aliens

What it is: Some power ups also affect aliens, along with player.

Why it needs testing: To make sure the powerups apply on all relevant game entities as expected, such as player, enemies and timer.

These are integration tests written in  
“AlienCharmIntegrationTest.java”,  
“InvisibilityIntegrationTest.java”,

“TimestampIntegrationTest.java”

## 15. Game states

What it is: Game states such as winning, pausing, game over, restart need to be tested.

These are integration tests written in  
“GameLogicIntegrationTest.java”

## **Excluded tests**

- Turret rotation: This was not tested because the rotation logic was dependant on real time behaviour which is complex and unreliable for testing. The rotation uses “Game.loop().getDeltaTime()” and updates depending on the frame rate. These values keep changing. As a result, the output angle for each update would also change between test runs, making it unreliable for testing. Additionally, the turret rotation is synced with turret vision and triggers line of sight ray recalculation. To test all these things, we would have to create helper functions that take random discrete values which wouldn’t be an accurate test, since our game doesn’t work that way.
- The “screens” package (includes main menu, HUD, in-game screen, etc) was excluded from testing and coverage analysis because it contains UI rendering and graphics and depends on Litiengine that cannot be reliably tested in a headless JUnit environment. Our testing focuses on core logic that is coded by us.
- Things that were dependent on LITIengine such as spawning. Spawning is handled by LITIengine’s map loader and is not explicitly coded by us.
- Vision animation of Turret and Alien are also dependent on Litiengine. We would have to mock the engine for this to work smoothly.

## **Findings**

Writing and running the unit and integration tests helped us understand the game's systems much better. We discovered some minor inconsistencies, logic errors and missing checks in the code that were overlooked before. For example, after reaching the teleporter, the level wasn't incrementing and moving to the next level, some power-ups did not always reset properly after use, and certain collisions were not handled as expected in edge cases. Writing the tests allowed us to see and fix these issues, which improved the stability and reliability of the game.

We didn't change the logic or any main code for our classes. We only added helper functions (getters and setters) to make testing easier.

### Test Coverage

- Overall
  - Line Coverage: 75%
  - Branch Coverage: 55%
- Model directory (all our classes)
  - Line coverage: 82%
  - Branch coverage: 54%
- App directory (GameLogic and main class)
  - Line coverage: 76%
  - Branch Coverage: 65%