Our implementation approach

Initially, we met a few times to discuss the logistics, the library to use, etc. For the first few days, everyone was getting comfortable with the library and then we divided the logic for the classes. We first decided to work on the visuals and animations after the entire logic was completed, but we started doing everything together, rather than separating them. We decided to make three levels, with increasing difficulty to include and showcase everything in our game. Level 1 would be simpler and level 2 and 3 would have more features, increasing the difficulty, such as getting a key to open the teleporter.

External libraries used

- Litiengine
- Reason - We chose to use LITIENGINE because it was easier to set up and integrate with Maven. Once we became familiar with the latest APIs, it made development much simpler and faster. LITIENGINE provides many built-in features such as map handling, sprite animations, entity management, and collision detection, which helped us focus more on gameplay design rather than low-level engine setup.

  Compared to a framework like LibGDX, LITIENGINE is more beginner-friendly. It has a clear structure for game objects, scenes, and resources, which made it easier for our team to organize the project. The engine also offers documentation and tools like the LITIENGINE utiliti for editing maps and resources, which saved us time during development. Overall, it provided everything we needed to build our 2D game efficiently without having to write extensive engine-level code.

Design changes made from Phase 1

- **UML (class) changes** - We did not make many major changes because our original design worked well. One change we made was removing the wall class that door and teleporter were supposed to extend. Instead, we created separate door and teleporter classes to make animations and interactions in LITIENGINE work more smoothly. Another change was not creating an enemy parent class with children for enemies and turrets. We used the built-in creature class from LITIENGINE for both the player and the enemies, which made the implementation simpler.

- **Use cases changes** - No changes to Use cases.

Management and Division of Work

Zihan - Punishments

Jugraj - HUD, Main menu, Settings, Audio

Palash - Effects, pickups and collision logic for Power-ups

Praneet - Enemies (alien and turret), Door, Button, Rewards (crystals), Teleporter, Animations, Game logic, HUD, Main menu

Michael - Visual assets, Maps and base for the level, Player (astronaut), Game Logic, Animations, Vision, Alien pathfinding

Process:

- **Initial Division:** Roles were established based on core UML classes (Enemies, Powerups, Punishments).
- **Integrated Development:** Due to the complexities of LITIengine, team members often collaborated across logical boundaries.
- **Check-ins and Review:** Regular communication and code reviews were maintained.
- **Version control:** Everyone worked on their separate branches to ensure parallel development without conflicts.

Challenges faced

- When deciding which library to use, we were initially unsure between LITIENGINE and LibGDX. LibGDX is a more popular and widely used framework, but we chose LITIENGINE because it integrates more easily with Maven. LibGDX is mainly designed for Gradle, so using it with Maven would have been more difficult and could have caused build or compatibility issues later on. Although LITIENGINE suited our setup better, we did face some challenges while using it. The official website is not fully updated, and several methods and APIs have changed over time. Many tutorials online also use older versions of the engine, so we had to spend time learning and finding the updated APIs to implement in our code.

Individual challenges

- Jugraj

One of the biggest challenges I faced was working with the LITIengine itself. Since it was my first time using this game framework, I had to spend a lot of time figuring out how its systems actually interacted, especially the screen management. The documentation was often unclear or incomplete, so I relied heavily on experimenting and debugging to understand how to make things work. Once I got comfortable, I worked on building the main menu, settings screen, and HUD all of which required me to connect the LITIengine screen system with our custom game logic and audio manager. Another challenge was managing branches in Git. There was an instance when my merges caused conflicts or broke the build, so I learned how to fix those issues and properly test before pushing. Through this process, I became much more confident with both Git and Java, and I learned how to structure code in a clean, modular way that other teammates could easily build on.

- Palash

The initial learning curve was really steep because I was introduced to three big things all at once: Git, Java, and the LITIengine library. I had to start learning everything from nothing. The most stressful part was learning Git for version control. I was always scared of messing up or accidentally deleting the main project code. That fear is completely gone now. I spent time mastering the basics, like committing and merging, and I feel much more comfortable with Git. It has changed from a scary thing into a reliable tool that helps manage and track the team's progress. This has been a major skill gain for me.

- Zihan

My responsibility was implementing the Punishment system, which includes the SilmeZone, RadiationZone, and the LaserBeam. Initially, I designed the Punishment as a single class, but it seems like there are too many differences between each hazard zone. It is too different to be handled with one implementation. As a result, to make the system more modular and easier to extend, I refactored it using an abstract factory pattern, which allows each punishment type to inherit and extend from a comm base class. In my opinion, one of the biggest challenges was that LITIengine's collision detection and update system behaved differently from what I thought it would be.
The engine doesn't automatically update every entity unless it implements IUpdateable. Another challenge was implementing the Laser, because our design included two types of lasers: intermittent switching and those that could be turned off by a button. Implementing the connection with the button left me feeling quite a headache.

Fortunately, I solved it using the button::isPressed method from the Button class provided by my teammate.

- Michael

A challenge that I faced was using utiLITI, a program the LITIengine developers created to help with designing levels. When it works, it makes implementing everything very easy. You drag and drop your maps and sprites in, and even create collision boxes for walls. But when it doesn't work, it's incredibly buggy and I had to delete and restart the tutorial map 3 separate times because once something is imported, deleting it could corrupt the entire litidata file. The collision mechanics are another challenge. I was under the assumption that collisions could trigger an event, but the documentation made it difficult to understand and implement those triggers. At the moment, I'm using collision box to bounding box overlap instead of collision box to collision box, which is not ideal nor standard in the industry. I also helped with animations since I created the sprites, but again, the convoluted documentation made the process for that much longer than it needed to be.

- Praneet

I struggled with figuring out animations for the door opening when the button is pressed. I was trying to achieve this by editing litidata. First, I added animations for the door and button within the class, and that didn't work. I also tried to add in the litidata and call it in the classes, tried changing the sprite type (like "prop", "button", "door"), combining frames in one PNG and referring to each frame as an index, etc, but none of those methods worked.