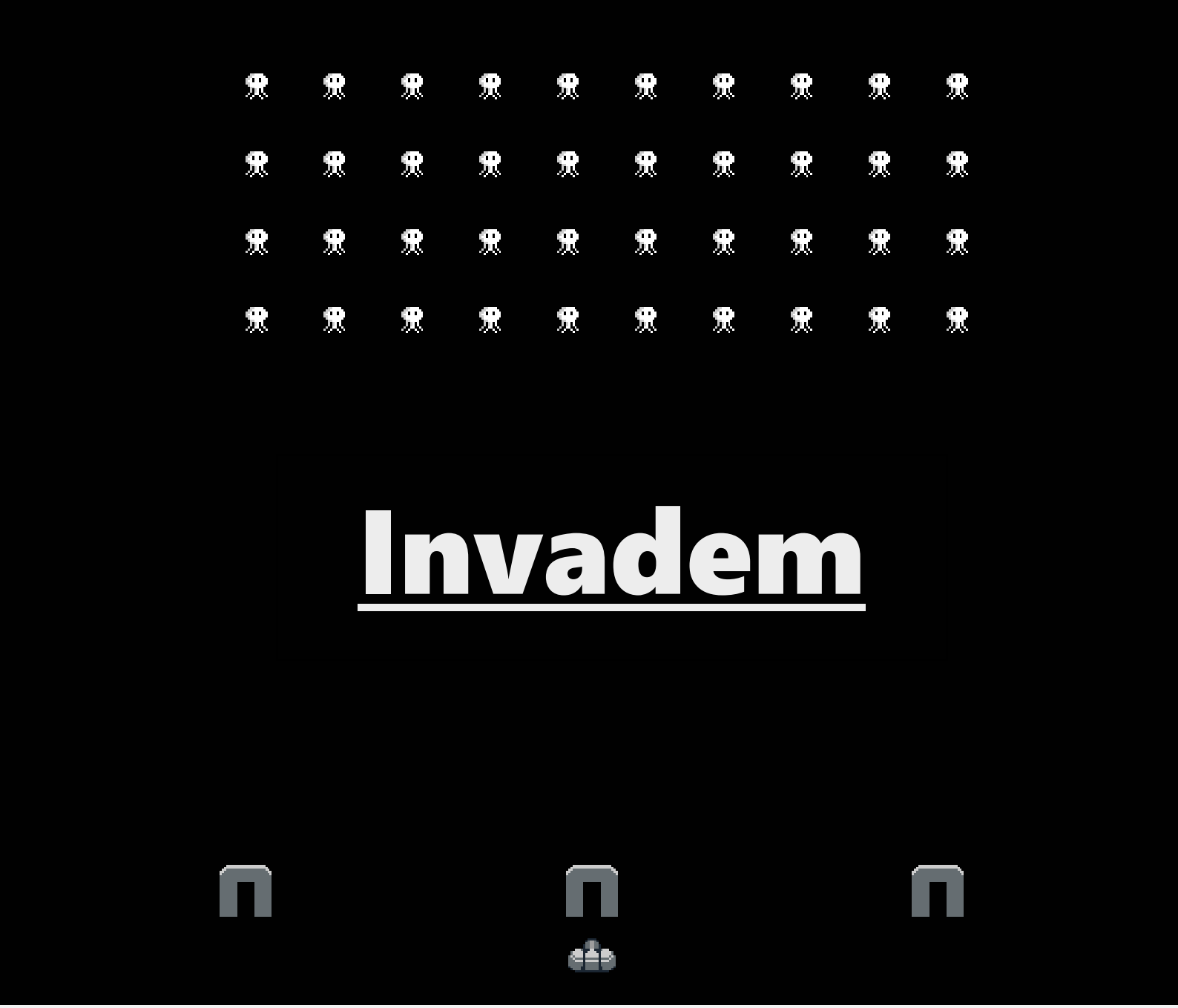

Invadem

Design/Implementation Process

This report outlines my process of designing and implementing the code for Invadem, a retro 2D shoot'em up.

Raunak Sitoula | rsit3870

Milestone Submission

Setting Up

The initial setup with Gradle was a little confusing, but once I finished setting it up and managed to successfully build it without errors, the milestone submission was a fairly straightforward process. I started off creating two classes:

- *Tank* – this class contains all (or almost all) data and logic related to the tank that players control in the game.
- *Invader* – this class is for the enemies the tank will be fighting off.

Next, I knew I would later be using some methods (such as collision detection on multiple classes; Tank, Invader, as well as the barriers and projectiles), so I created some interfaces for these:

- *IMoveable* – I created this interface with the intent of enforcing the same methods and properties on all objects in the game that move or can be moved. Within this, I also defined an enum, *Direction*, as I assumed I would need a more readable way of defining movement directions at some places in my code.
- *IHittable* – This interface houses a property for getting the health points of the object, and a method to "hit" and make the object receive damage.
- *IEntity* – This interface extends *IHittable* with a number of methods that are more specific to interactable entities in the game. It didn't make sense, for example, for an interface called *IHittable* to have methods and properties related to things like its size, or whether or not it is "dead". I decided to extend it into a single interface *IEntity* as I would still be using all of these things within the same context, and having these split up across multiple interfaces would be cumbersome and inefficient.

I also considered abstract/base classes, but decided against it as I had some entities who did not have the exact same requirements (Barriers and Tanks can both be hit, but Barriers can't move), and inheriting from multiple classes is not possible. Therefore, interfaces were the obvious choice for me.

Processing

The next step was to get something visible up on the screen; I spent a bit of time figuring out processing and found it quite similar to the p5js framework for JavaScript which I had worked with before, so it didn't take me long to get the hang of things and have the tank rendered up on the screen.

Input

Input was slightly confusing as I wasn't sure if I should use the input event system, the built-in override methods that get called, or checking for pressed keys in *Update()*. After a bit of experimentation and looking through the docs, I found that having my own boolean key press values updated through checking the input in *Update()* gave me the best results.

Submission

After that, all that was left to do was:

- Create the projectile, *Bullet*, which used an *IEntity* field to determine who shot it so that it can deal damage to the appropriate targets using *IEntity.GetEntityType()*.
- Spawn the invaders at the start of a level, which I implemented using 2 for loops, one nested inside the other; one for each row, and one for each item along the row to form columns.

Final Submission

Setting Up Test Cases

Test cases in this project seemed quite confusing at first, but I took a look at the example test cases, and it made sense just looking at it immediately. To be sure my guess was right, I put in an always invalid result in one of the test cases, and built the project – I was right, the test cases were run automatically during build, and it even showed the results in a nice .html file!

I first modified the default test cases to work properly with my code structure, since it was a bit different to what the example test cases assumed my method names and code structure to be like. After that, I wrote a few test cases to test some basic, as well as more niche functionality of the program.

Finishing Up Core Functionality

Finally, I created a static class for collision checking – or rather, a final class, since I found that Java only allows nested classes to be static for some reason, and the only decent workaround appeared to be to make it a final class after looking around on StackExchange for answers. It contains a static method for comparing two *IEntity* classes, and returns a Boolean value for whether or not they collide.

Using this class, I implemented collision checking between bullets and other entities (excluding other bullets), and called the *Hit()* method upon collision as all *IEntity* classes included the *IHittable* interface to cause damage to them.

Last but not least, I put in the Game Over and Next Level screens; the brief said I needed these screens to be up for 2 seconds each, so assuming the game ran at a fixed framerate of 60 per second, I simply left these screens up for 120 frames each when needed using a counter in the update loop.

Extension Task

The extension task was fairly easy; I already had my code setup to scale up to something like this, so it was just a matter of adding an enum to identify different kinds of invaders and bullets, and slightly modify the constructors for the invaders and the projectiles. The only thing I needed to look up, really, was the processing docs for displaying text, which didn't take too long!

Concluding Notes

This was a pretty fun assessment, and something that's useable in a portfolio, so thank you to the staff that were involved in putting this together! It was also a nice experience learning to use Gradle and IntelliJ, and made the programming experience in Java much more enjoyable!