Implementation

Team 13

Darcy Adams

Davids Kacs

Jakub Grzmil

Nam Duong

Samin Alborzi Movahhed

Will Dunlop

Architecture

A few words on how our abstract architecture influenced the final implementation. All of these changes are discussed in more detail further in the document.

`Obstacle` class has changed in the abstract architecture into a `Collidable` class. This was directly reflected in the implementation. However, the abstract architecture suggests that this would become the base class from which both `Obstacle` and `Powerup` inherit. Whilst this was originally the plan, both classes ended up being combined into one `Collidable` class. This was because we discovered that it would be too costly to implement this way due to how obstacle spawning was implemented in the original project. This satisfies UR_POWERUPS and FR_POWERUP_RATE, whilst still performing all the functions it needs to satisfy all of the requirements `Obstacles` was responsible for.

We knew that we would need to keep track of the game difficulty (for requirements FR_DIFFICULTY_SELECTION and UR_DIFFICULTY_BEFORE_GAME). This is why the `Difficulty` enum was created. This translated directly into code, though this enum ended up doing more then we originally anticipated, and was used to actually switch the difficulty, not just to keep track of which one was selected.

A `PauseScreen` was also introduced. This was a very simple addition, hence was not marked on the diagram, but it was crucial in implementing saving (FR_SAVE_GAME, FR_PAUSE_SCREEN). This was just as simple as anticipated.

Changes to the previous project

Changing Obstacle to Collidable

This change was motivated by requirement UR_POWERUPS. We discovered that the easiest way to implement this (and by extension the one requiring fewest changes to the original implementation) would be to spawn powerups with the same mechanism as obstacles. The class was renamed to `Collidable`, and the enum became `CollidableType`. The name change was to minimize confusion as powerups are not obstacles.

Previously the way boats took damage from obstacles was by subtracting the value of the `damage` field of the obstacle they collided with. This system was unsuitable for powerups, so instead each `ObstacleType` now has a lambda associated with it (stored as a functional interface `CollidableEffect`). Each lambda takes a `Boat` object and modifies it. For the obstacles this is as simple as calling the `addHealth` method of the `Boat` class with a negative value. Powerups call other methods such as `addSpeed()` or `setBuff()` (which changes the multiplier for the amount of damage boats take). See Figure 1 for an example.

Changing the way screens create, store and use their resources

In the older version of the project, each screen would store each texture and button in their own variables. In order to increase readability and make the intended function of the code more apparent, textures and the buttons are now stored in arrays. A third array was made to store prefixes of

Figure 1: Example of an obstacle and a powerup with their respective lambdas.

```
ROCK("rock.png", 50, boat -> {
    boat.addHealth(-ROCK_DAMAGE * boat.getBuff() * Settings.OBSTACLE_DAMAGE_MULTIPLIER);
}),

HEAL("heal.png", 0, boat -> {
    boat.addHealth(boat.getBoatType().getHealth() * HEAL_BY);
});
```

texture names, this allowed us to store texture names for icons and buttons in a single array (the reason for this will become apparent in the next section).

In some cases where the action taken by each button was very different, a fourth array was created to store the actions of each button, this is similar to how different actions of collidables are implemented. See Figure 2 for an example.

Figure 2: Example of different actions invoked by buttons stored in an array

```
buttonActions = new Runnable[]{
        () -> Gdx.app.exit(),
        () -> game.setScreen(previousScreen),
        this::save
};
```

Factories

Classes `ButtonFactory` and `VectorFactory` were made to manage the creation of `Button`s and (some) `Vector2`s. This was done for the same reason as the previous change. This was especially important in screens. The `Button` constructor takes a position vector and 2 textures. The textures have names in the form texturename_button_active.png and texturename_button_inactive.png. This is now abstracted behind a collection of static methods in the form `ButtonFactory.nameOfScreen("texturename")`. Allowing us to simply pass a single prefix of a texture name and have the `ButtonFactory` methods append _button_(in)active.png to the end of that name. These methods also calculate the position of the buttons on the screen. The position is automatically adjusted each time the method is called (tracked with a static variable). See Figure 3 for an example.

New features

CollidableTimer

This is a class, which creates a timer. This timer runs asynchronously and invokes the lambda passed to the constructor of the class when the time runs out. This is used for powerups which need to run out (like invulnerability or speed up). E.g. when the timer runs out the speed is brought back to its normal state.

Figure 3: Examples of instantiating buttons in the previous and current versions

```
// Original
this.exitButton = new Button(new Vector2((Gdx.graphics.getWidth() - EntityType.BUTTON.
    getWidth()) / 2.0f, 100f / Settings.SCALAR), "exit_button_active.png", "
    exit_button_inactive.png");

// new
for (int i = 0; i < buttons.length; i++) {
    buttons[i] = ButtonFactory.mainMenu(textureNames[i]);
}
```

Difficulty selection

These features were added to complete UR_DIFFICULTY_BEFORE_GAME and FR_DIFFICULTY_SELECTION. The `Difficulty` enum was added. This enum contains the 4 difficulties (`EASY`, `MEDIUM`, `HARD`, `VERYHARD`) and a method which sets all of the appropriate settings for each respective difficulty. Each difficulty takes a `Runnable` as a parameter. When invoked, the `Runnable` calls the `setVars` method with appropriate values for each setting.

Selecting a difficulty invokes the `Runnable`. The selection happens when the user reaches a newly added `DifficultySelectScreen`. This screen is very similar to the `BoatSelectScreen`. It is created by the `BoatSelectScreen` and takes its place in creating a `MainGameScreen`.

Pause screen

Added to complete FR_PAUSE_SCREEN, this is the screen the user sees when they press Escape during game play. The main purpose of this screen is to allow the user to save at any point (as specified in UR_SAVE_FEATURE), however it also has the buttons to resume the game as well as to exit. This functionality is stored in an array of `Runnable`, as can be seen in Figure 2.

Saving and restoring

Functionality required by UR_SAVE_FEATURE, FR_SAVE_GAME and FR_LOAD_GAME.

The class `Prefs` was added to control the saving and restoring. This class has 2 static sub-classes, `Save` and `Restore`. Both of these classes have methods `open()` and `close()` to be called before and after each save and restore operation. These methods ensure that after data has been written to the save file it is marked as valid, and before it was restored, the validity of the file is checked. This is done by writing a special key 'save_exists' to the save file. If this key does not exist when the save is being restored, a `SaveDoesNotExist` exception is thrown. See the next section for how this exception is handled.

The saving process starts when the user presses the 'save' button on the pause screen. This does several things. The newly added `save()` methods are called for all objects which are not trivial to save, i.e. not primitive datatypes or objects which can be easily transformed into primitive datatypes. `DragonBoatRace`, `MainGameScreen`, `Settings` and `CollidableStats` are examples off such datatypes that need to be saved on the `PauseScreen`. Each `save()` repeats the process, until there are no non-primitive types to save.

Primitive types are saved using `Prefs.Save.putType()` where 'Type' is the type to be saved. In case of float and int, these are as simple as being wrappers for LibGDX `Preferences.putFloat()` and `Preferences.putInteger()`. Others like `Prefs.Save.putVector2()` have to save their components individually.

In most cases it is possible to use the variable name as the key when storing that variable. This becomes an issue when storing arrays of objects. For `Float[]` arrays, a method `Prefs.Save.putArray()` exists, which serializes the array to JSON and stores it as a string. This, however, was impossible to do with `Boat`. Boat has its own `save()` method. All boats have a `name` field, which is used to 'salt' the key to avoid writing data to the same key multiple times.

Reloading is very similar, but it starts in `MainMenuScreen` where the user presses a button to 'resume' their game. For each `Prefs.Save.putType()` there exits a `Prefs.Restore.getType()` which returns that type. If `Prefs.Restore.open()` Throws an exception, this exception is caught and a popup screen is presented to the user stating 'Could not find a saved game', if the user then chooses 'play' the game begins normally.

Popup screen

Popup screen is a screen used to convey some information to the user. It can be constructed with a `message` which will be displayed. The words 'Press Space to continue' are also displayed. Upon pressing the space bar the user will be returned to the screen which spawned the popup.

This is used to notify the user that their game was saved, when they press the 'save' button, as well as to tell them if no valid save file exists when they press 'resume'.