FOOP 2021 - Final

§ Design Report

Team Members

- b08902068 黃政穎
- b08902069 郭承諺
- b08902070 戴培凱

Responsibility

- b08902068 黃政穎
    - 所有Manager classes，trail，dart / bloon / monkey的程式 + 圖片
    - Class diagram
- b08902069 郭承諺
    - design report, bloons, monkeys
- b08902070 戴培凱
    - StartScreen, MenuScreen, UI + 相關圖片

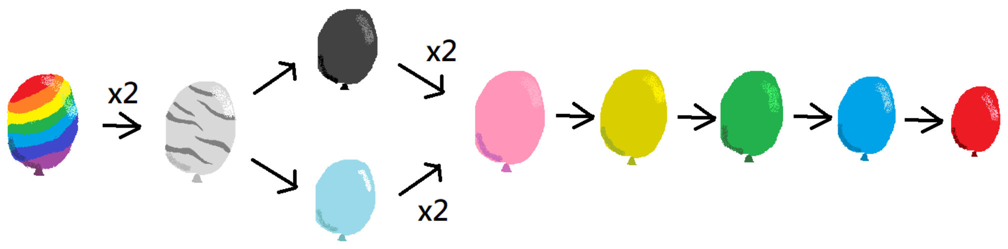
Inspiration

- Inspiration comes from classical game : **BloonTD**

- Basically all of the character, mechanism and textures (e.g. monkeys, ballons, weapons) are based on this game, with textures and map redrew by Windows Paint. Our goal is to try to remake it’s first version (BTD1), with some monkey variant from later versions to create more behavioral veriance. Not all mechanism of BTD1 is made (e.g. ability to freeze bloons), as we adopt making more behavior variance on darts and monkeys instead of bloons themselves.

Classes

- Hierarchy
  - LibGDX class: class `Game` → class `Screen` → class `Stage` → class `Actor`
- Game
  - class `BalloonTD`
    - entry of the game itself, used by other platform-dependent main class. (e.g. desktop, html, android)
    - including `StartScreen` , `MenuScreen` and `GameScreen` .
    - manages and shows desired `Screen` .
- Screen
  - including multiple stages
  - construct buttons for player to press
  - Subclass including:
    - class `StartScreen`
      - performing at the beginning of the game
    - class `MenuScreen`
      - handle menu part on the screen
      - construct pages
    - class `GameScreen`
      - responsible for most part of the game
      - have Trail, RoundManager, BloonManager, DartManager, MonkeyManager, Player, Map
- Stage
  - Composition of actors. Can delegate `act(float delta)` and `draw()` to actors and also pass some input events to them.
- Actor
  - abstract class `Bloon`
    - Supports abstract methods like `pop(Dart dart)` to spawn lower-level bloons
    - Subclasses including `RedBloon` , `BlueBloon` , `GreenBloon` , `YellowBloon` , `PinkBloon` , `BlackBloon` , `WiteBloon` , `ZebraBloon` , `RainbowBloon` .



  - abstract class `Monkey` :
    - `act(float delta)`

        - use class `BloonManager` to get the bloons in shooting range, and calls abstract `shoot()` to perform different shooting behavior for individual monkey.
      - `shoot(List<Bloon> bloons)`
        - Abstract method to determine how a monkey shoot darts given a set of bloons in sight.
      - Subclasses including: `DartShooter`, `TackShooter`, `BombShooter`, `SuperMonkey`, `BoomerangShooter`, `DartlingGunner`

  - abstract class Dart

      - has `hit()` that determines the behavior when hitting a bloon. Most common action is to just call `bloon.pop(this)`.
      - `move(float delta)` delegates the moving mechanisms to subclass, making special darts possible (e.g. boomerang that curls back to shooter).

  - class BloonManager, MonkeyManager, DartManager

      - a composition of all bloons / monkeys / darts. Maintains a list of those kinds of things on screen. Can delegate `act()` and `draw(...)` to individual instances.
      - Used in order to avoid changing raw `List<bloons / monkeys / darts>` directly, which may cause potential undefined behavior of “changing list while iterating“.
        - Adding element: use a temporary buffer to add things in, and dump buffer into actual list only at the end of `Screen::render()`.
        - Deleting element: make bloons / monkeys / darts have an alive state, and remove any dead items at the end of `render()`.
      - Thus the list everyone gets when inside `render()` would not change in the span of `render()`, except at the end of `render()` when managers clean up things.

  - class UserInterface

      - resposible for interaction with user(player).
      - show `Player` and `Monkey` info on the right side of the screen.
      - buy, sell, level up `monkey`
      - show ranges of each monkey to user.

- Other

  - class `Player`
    - as a manager of HP and money.
  - class `Trail`
    - defines a trail for bloons to walk on. Translates “how much distance walked from start” to “the actual coordinate on map”.
    - Subclass including `StraightTrail` and `CompositeTrail`
  - class `RoundManager`
    - Controls any round-related tasks, e.g. start round, spawn bloons, give rewards, etc.
    - has `IBloonSpawner` and `RoundParser`.
  - interface `IBloonSpawner`, class `BloonSpawnerClass`
    - Used by `RoundManager` to spawn bloons. `Bloon spawn(String name)` can spawn bloon given its name. Dependent on file format of game round information.
  - class `RoundParser`

    - Parse game round files, which records what bloons to spawn in each round. Located in `RoundManager.java` as non-public class.

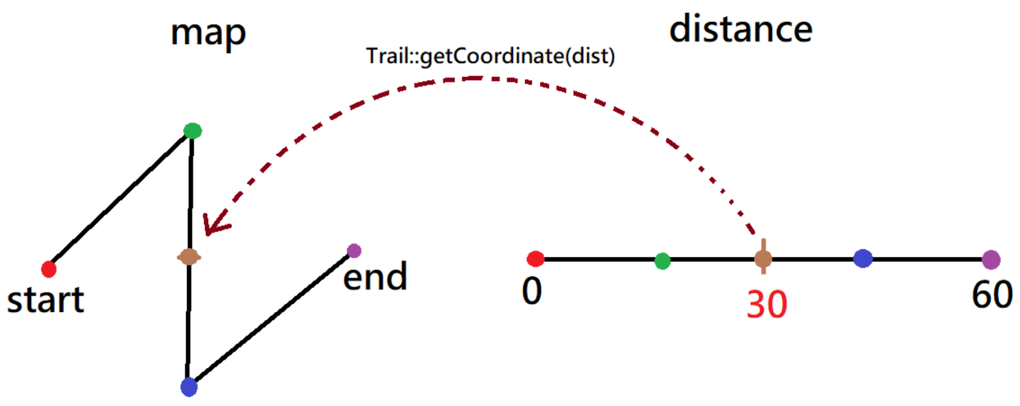
Class Diagram

- Drawn with draw.io
- PNG file: https://drive.google.com/file/d/1QJC7Cj_-uLSAjW5h2gyDqH9ETK6HjCDx/view?usp=sharing

Advantage

Trail extensibility & constant speed

By defining a coordinate translation between “how much distance you walked” and “the actual coordinate”, bloons can walk by same speed by adding $v \cdot dt$ to its own variable `dist` recording distance walked, and let Trail do the translation.



Our map is a collection of straight lines. The coordinate translation of a straight line is very easy (i.e. `actual_coord = start_pos + (dist / total_length) * (end_pos - start_pos)`), and the “collection of lines” part is handled by CompositeTrails, which can connect multiple trails and delegate the coordinate translation to them.
With definition of coordinate translation and CompositeTrail, there will be no difficulty to add new trail shapes and connect them together. In theory, we can even just use a large amount of StraightTrail to achieve any trail shape.
This can also tackle the bloon spawning problem: when popping a rainbow bloon, 2 zebra bloons will spawn. They need to spawn at roughly the same position as the popped rainbow bloon, but also space a little bit out. Without this translation, it will be a hard task to implement those.

`Bloon::pop` , `Dart::hit` and `Dart::move` to support special darts and bloon levels (or low level bloon spawning).

There are many behavioral variance for darts. We decided that giving darts an interface `Dart::hit(bloon)` to let the darts themselves decide what they can do next. For example, there are 2 variant in this game:

1. Normal dart: Only calls `bloon.pop()` , nothing special.
2. Bombs: For all bloons close enough to the bloon being hit, also `hit()` those bloons as well

If there are more behavioral variant (e.g. shoots special darts to inflicit special effects to bloon), it can also be achieved through `Dart::hit(bloon)` .

`Dart::move` is another behavioral variance supported by darts. The darts can decide how they move. For example, we can make seeking darts by searching the nearest bloon and walk towards that direction. The implemented behavioral variance has 2 kind:

- Normal dart: simply moves forward.
- Boomerang: curls back to shooter.

Managers to avoid undefined behavior

Managers did things below to ease burdens of developer:

1. Act as a list of things, supporting adding and deleting easily (by a defined procedure).
2. Avoids "changing list when iterating" problem, or modification in iteration.
3. Act as a group of actors. Those actor's `act(float delta)` and `draw()` is called by manager's delegation of the same method, which is called by Stage.

(well, actually LibGDX has a thing called `SnapshotArray`, which tackles modification in iteration well. We discover this the day before deadline. At least that means our concerns are a real problem and needs to be tackled with.)

Disadvantage

Some shapes and positions are fixed

- UI

The pictures' size are fixed, and the size and position of UserInterface are also fixed, so screen size may not be flexible. But this problem can be easily fixed, as long as we change all the value into some scale of screen size.

- Bloons

We implemented `Bloon::getTouchRadius`, which basically defines a circle cented at the center of bloon, and dart can then use this circle to determine whether the dart has hit the bloon (by `Dart::touched(bloon)`).

This lacks extensibility since that means bloons cannot have different shapes other than circle (e.g. a blimp has a somewhat rectangular or elliptical shape, which can't be modeled by a circle.) A way to improve is to make a somewhat universal `Shape` object, supporting intersection between shapes and other methods.

Repetitive code...?

If you dive into the source code of monkey subclasses, what we'll find is a similar structure among all of them. They all have a private static float array for levelup costs, the same private static texture for every instance of the class, and a lot of them needs a `ShootBehavior` class, which utilize strategy pattern to deal with behavioral variance of upgrading monkeys.

To our opinion, that stems from the fact that abstract class `Monkey` is not concrete enough. Maybe if we can make a concrete enough class that implements what a normal monkey kind needs, and let most of the monkey subclass inherit from it. This way, special monkeys can still inherit directly from `Monkey`, but if requirements meet, there is a class that handles most common things for normal monkeys.

Packages

- libGDX: `com.badlogic.gdx.*`

How to play

- Game flow
    - start of game
    - bloons start comming through the trail one by one in every round
        - note that the bloons that reach the end point will cause the loss players hp
    - monkeys on the map try to destroy the bloons automatically
    - if player's hp drops to 0
        - game over
    - if player's hp > 0
        - enter next round
        - if pass all the challenge, end of game
- monkey
    - usage
        - destroy the passing bloons
    - how to use
        - use mouse to click on the monkey to get information(level, sell upgrade)
        - buy or upgrade with money
        - use mouse to pull the monkey to specific space
- money
    - how to gain
        - initially assigned
        - destroy bloons
        - finish a round
        - sell monkeys : get a proportion of money spent on it back
    - how to use
        - buy new monkeys & upgrade monkeys
            - both can be perform during the round
- bloon
    - automatically generated by program using a file recording all spawning information.
- User Interface
    - Current level, HP, monkey is showed on the right top
    - click "start round" to start a new round, click "exit" to return to menu
    - buy monkey mode(preset mode)
        - click the "buy monkey" button to enter this mode(If currently in monkey info mode)
        - click "pre" or "next" to switch between monkey
        - click "buy" to buy monkey, then drag the monkey to map
        - If clicking "buy" button, click "cancel" to cancel buying

- monkey info mode
    - click on the monkeys in the map to enter this mode
    - it will show the name, level, sell price and upgrade info
    - If put the mouse on the monkey, it will show the shooting range of the monkey
    - click “sell” to sell the monkey
    - click “Lv.up” to upgrade the monkey