CS 3110: Functional Programming **Problem Set 6**
Bo Yuan Zhou (bz88), Rene Zhang (rz99)

Summary

Instructions

1. Cd into the src directory
2. Compile the project: Make
3. Run the game server in one terminal: game/game.exe
4. Open the GUI in another terminal: java -jar gui/gui_client.jar
5. Click on the "Connect" button in the GUI.
6. Open a terminal and run a bot, which will be the red bot: team/babybot.exe
7. Open a terminal and run a bot, which will be the blue bot: team/babybot.exe

Design and Implementation

- **Modules**:
  We split up our modules according to what functionalities we needed from our game. Specifically, we laid it out as follows:
  - Keeping track of the game status: game.ml
  - Player-side functionalities (focused/unfocused movement, bombs, lives, charge, score, get hit): player.ml
  - Handling collisions for bullets and powerups: projectile.ml
  - Bullet features (patterns, movement): bullet.ml
  - Powerup features (effects, movement): powerup.ml
  - NPC-side functionalities (movement, get hit, spawn powerups): npc.ml
- **Architecture**:
  Game.ml is our entire overarching structure, so it depends on all of the modules. Meanwhile, player.ml has no such dependencies, since everything we do there does not rely on anything else (it is handled, for the most part, internally). However, it is important in the usage of projectile.ml, since we use player functions there when collisions are detected. Projectile.ml also requires information from npc.ml, since it also handles

collisions for npcs. Bullet.ml and powerup.ml then extend the Collider type in projectile.ml by handling their respective effects and spawnings. Finally, npc.ml relies on projectile.ml, specifically the spawner for powerups, which is called when the npc dies, and player.ml to keep track of who's bullet hit the npc.

- **Code Design**:
  Our important design decisions stemmed mostly from dealing with projectiles and npcs, where we would do most of our computation à la collisions. In both cases, we used lists, but for different reasons. For projectiles, a list is optimal since at every time tick we would have to check collisions on every projectile anyway. For npcs, a more efficient way to deal with collisions on npcs would be to store npcs in a Hashtable rather than a list to make lookup (by ids) easier. However, since we don't deal with large numbers of npcs, we continued with using lists for the sake of simplicity.
- **Implementation**:
  Since we aimed from the beginning to complete the project in its entirety, we had a combination of top-down and bottom-up design where we had one person (Rene) working starting from the general infrastructure (i.e. game.ml, projectile.ml, player.ml) while the other person (Bobby) handled the more specific modules and pieces (i.e. bullet.ml, powerup.ml, npc.ml). After we built down/up accordingly, the two parts were pieced together.

Testing

In order to test our code, we ran the game under different sets of constants depending on what we were testing. For example, if we wanted to test to see if our UFOs were dying as intended, we set cINITIAL_BOMBS to a low number so that bullets wouldn't be removed as often and cINITIAL_LIVES to a high number so we could keep the game running for longer. By doing this, we're confident that we can thoroughly test each aspect of the game as we implement it so that the final version of the game is entirely working.

Extensibility

- New bullet types:
  In our Bullet.spawn we use the defined attributes of a bullet type (speed and radius) to determine its behavior. In order to create new bullet types, we would have to account for the new bullet type's attributes in our definitions and constants and then add new a section in our match statement in Bullet.spawn that creates a bullet with the behavior that we want.
- New types of collectible items:
  We have a function Powerup.spawn that we use to create powerups that

both players can receive and a function Powerup.playerEvent which determines the effect the item has on a player. To create new collectible item types, we would create a match statement like we did for Bullet.spawn that checks the type of the collectible being spawned in Powerup.spawn to determine the behavior and in Powerup.playerEvent to determine the effects.

- More interesting bomb effects:

- Neutral enemies that fire at both players:
  Since we have a function Npc.spawn that creates npcs with a type of behavior (which defines what they do on a time tick), we can modify our Npc.spawn like we did Bullet.spawn so that we can take different Npc types and then create Npcs with different types of behavior using a match statement.

Known Problems

Comments