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

Summary

To simplify the creation of steamMaku, we separated the game into many individual modules. This allowed us to use a more "object oriented" approach by having the modules act like objects. We were told to not use OCaml classes for this assignments, so we had to make due with modules instead. We used mutable records in the modules to handle their individual states and we had getters to allow us to easily retreive the internal data. Additionally, we wrote a simplified entity-component system to handle the different types of bullets and possible new npcs. More details are in the following sections.

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 localhost 10500
7. Open a terminal and run a bot, which will be the blue bot: ./team/babybot.exe localhost 10500

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
  - Abstraction of commonly used functionalitites: gameutil.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

  In order to have these modules act as objects, we created an abstract Object module signature in gameutil.ml and had all the other modules include that one. Thus, all of our modules have a constructor and an update command. To simplify how the game is handled, we had the game just call all of the update commands in it's handle_time function. This allows us to pass all of the work to the modules and just having game.ml unifying the different modules together. Because both bullets and powerups have very similar functionalities, we created a projectile functor to handle all the movement and collisions. All of the modules store their own data using records, but we have getter for the data that allow us to get the correct format for the data needed.

- **Architecture**:
  We employed a simplified version of an entity-component system. Our bullets, powerups and npc were all written as larger entities that can handle many different kinds of components. This allows us to easily extend the game by just adding a new match statement to the module. Additionally, it makes it so our modules dont have to be very tightly coupled, and we can easily change functionality of one without affecting another.
  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**:
  We chose to use mutable records in order to store our own internal state. By doing so, we can very easily modify certain parts of the record at any point. At the same time, we can easily get data from the records as well.
  Our other 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. We do not need to be able to search through the list at a later time to find a bullet. 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. Additionally, we also did incremental testing by changing parts of the bots so that they would do certain actions. By doing this, we could test to make sure we could handle different corner cases.

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:
  For more interesting bomb effects, we could add a match statement to match on the type of bomb. This just allows us to specify the specific behavior we want for that type of bombs. For example, we can just specify

that if we receive a normal bomb command, we will just use a bomb and clear the screen of bullets. If we receive a different bomb command, then we can call different functions in our modules to create a different functioning bomb.

- 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
There does not seem to be any known problems at this time.

Comments

- We spent around 10 hours on this assignment. We spent about 25% of our time on design, 15% on testing and the rest on coding the game. We didn't spend too long on the assignment because after we worked out the interfaces, we managed to have our code decoupled enough so that we could both work on different parts of the code, as described previously. By doing so, we could just work on each individual module and have them satisfy the interfaces that we wanted.
- You definitely should have told us that the assignment would not work on Windows OS. If we were told so from the beginning, we would not have spent a while trying to get it to run on Windows.
- The assignment was much easier than expected. After looking at previous year's final problem sets, we saw that this problem set had much less work than that of what was expected in the past.
- Nothing was really hard about the assignment. The only issues we really encountered was the fact that the assignment didn't work on any windows OS.
- We liked the fact that we got to work on creating a game. However, we would've enjoyed it more if the game was playable between both players and bots: eg. player vs player, player vs bot and bot vs bot. Additionally,

we didn't really like how a huge chunk of the code was already given to us. We felt like we didn't really do much for this assignment.

- I would make the assignment much more open ended and allow the students to be able to submit more creative games. You could do this by letting the students design a lot more of the backend architecture themselves (not give as much code). Additionally, the writeup was extrememely guided and gave little room for our own design decisions. We were also unable to add any additional extensions to the game, such as new npcs and new bullets because we were not really allowed to modify the architecture we were already given.