SteamMaku Design Document

Luke Benning (lab292) & Anton Gilgur (ag766)

December 2, 2013

1 Summary

In the following sections, we will describe our implementation of SteamMaku. We will discuss our choice of modules and what functionality they are responsible for, as well as describing how our code works at a high level. Our design choices will be discussed, along with the reasons for making such choices. We will describe our testing process and how it contributed to a stable implementation of the game. Finally, we will discuss the feasibility of ways that the game can be extended, such as by adding more enemies or different bullet types.

2 Instructions

To run our game, run the Make file in the top folder to compile the game. Then run game.exe, the gui via the command java -jar gui client.jar and then connect to the server thouh default addresses, and finally run the bots with the server and port as arguments. The server can just be localhost and the ports can be 10500 (or similar).

3 Design & Documentation

We will now describe the parts of our SteamMaku implementation in the following sections.

3.1 Modules

We designed our program to be modular in the sense that many functions for handling game functionality were placed into relevant modules. Our game has 5 modules: TeamMechanics, PlayerMechanics, WeaponMechanics, UFOMechanics and CollisionMechanics.

• TeamMechanics - Provides functions for accessing and editing team data. Examples - retrieve character, edit score, power, charge etc.
• PlayerMechanics - Provides functions for creating the initial player, moving the player and changing focus.

• WeaponMechanics - Implements functions for creating and moving bullets and powerups.
• UFOMechanics - Provide functions for creating and moving ufos as well as handling damage.
• CollisionMechanics - Functions for handling collisions between players, bullets, powerups and ufos. Specifically handles destroyed ufos and collected powerups.

These modules are continually called in Game.ml as the game state is updated during a timestep or when handling AI commands. The interfaces for each module allow access to functions that are relevant in Game.ml. Any internal functions needed only in the module are hidden. The modules maintain the invariants that whenever they take some game object as an argument (such as bullets, powers, teams, players etc.), the object returned is valid under the rules of the game. For example, functions in team that edit team data return teams that are correct according to game specifications.

3.2 Architecture

The 5 modules depend on each other in a specific ordering. PlayerMechanics is the module that does not depend on any others (only edits player - which requires no extraneous functions). TeamMechanics then uses the PlayerMechanics module to edit the team player when necessary. The WeaponMechanics module relies on these two previously mentioned modules, since the bullets and powerups need access to team data for firing/collecting, as well as the player locations for movement purposes. The UFOMechanics module also has no dependencies, as it simply handles ufo movement and damage until iti s destroyed. All of these modules are then required for CollisionMechanics, which needs the game objects in order to check for collisions and the outcomes associated with destroyed objects.

3.3 Code Design

We chose to accurately represent the game state by storing relevant data inside a mutable record. This record contains the following data:
• Red and blue teams - For keeping track of team data.
• Ufo * int list - Keep track of all ufos along with the time they were created.
• Bullet and powerup lists - For tracking bullets and powerups currently in play.
• Time elapsed - For tracking when to spawn a new ufo.
• Red and blue move lists - To move the red and blue characters by one move each time step.
• Red and blue mercy invincibilities - Track how much long each player invincible.
• Red and blue bomb invincibilities - Track how much long each player invincible (Not same as mercy - different effects).

We decided to continually update the game record sequentially during a time step since it was convenient for doing updates in the correct chronological order. This made our code slightly complex and not as elegant as we would have liked, since our handle time function essentially became many edits in a row to the game record. We did not specifically employ any notable algorithms, however we wrote our program in a modular and fairly intuitive way to allow for easier debugging and future extensibility.

3.4 Implementation

We employed a mianly top down approach, starting with the game.ml implementation and introducing modules where appropiate. This seemed like a good plan in foresight since we could write the modules as we went along and not worry about thinking hard at the beginning about what modules to use. In hindsight however, our code might have looked a bit neater and have been easier to write if we had the modules built first. Luke Benning wrote most of the module and game code. Anton Gilgur made several additions to the game and module code, as well as writing the graphics code and bot.

4 Testing

Our test plan involved adding part by part to our game code, while at each step testing for correct functionality. We first added players, then bullets, bombs, ufos and powerups. We brainstormed together many scenarios that could occur, no matter how unlikely, and wrote code to handle these cases (such as multiple collisions). Piazza was a very helpful resource for understanding game functionality in many of these cases (such as bullet hitting ufo and player in same time step). We are very confident that our testing gives good enough coverage of the scenarios because of our incremental test driven approach.

5 Extensibility

Our modular design allows for easy extensibility.

• New bullet types - We would simply add new functionality to WeaponMechanics to handle more bullet types.
• New collectibles - If the collectible was a weapon, we would add functions for it inside WeaponMechanics, otherwise we could create a module for various pick up items. In either case, the collision functionality would then be in CollisionMechanics.
• Bomb effects - We would code these effects into WeaponMechanics, with any collision details handled in CollisionMechanics.
• Neutral Enemies - We could create another neutral player in PlayerMechanics and assign it to a team in TeamMechanics, then handle its movements, weapons and collisions in the existing modules. However, it might be easier to just create an enemies module to handle these new enemies.

6 Known Problems

None known.

7 Comments

The assignment turned out to be pretty fun, as implementing a game is more interesting than building a scheme interpreter or infinite-precision reals (though those problem sets were still cool!). The assignment turned out to be conceptually easier than the previous problem sets, though it still took a long time to do. The game had many rules, and a lot of edge cases arose that weren't answered in the writeup. I'd recommend being a bit more clearer on the collision specs., as they weren't comprehensive. Also, the problem set should state at the outset that it won't work on Windows, rather than 5 days before it is due. This would be much more helpful, as it would prevent windows users from spending valuable time trying to get the game to work. On the bright side, I liked how we had a lot of freedom in this problem set. Given free reign to make our own design was great, as it felt more natural rather than being forced to use someone elses strict set of guidelines. And lastly, applause to Chris Yu for spending a lot of time making this, it must have been a monumental effort!