SteamMaku Design Document

Luke Benning (lab292) & Anton Gilgur (ag766)

December 6, 2013

1 Summary

In the following sections, we will describe our implementation of SteamMaku. We chose to implement the entire game, including the powerups and UFOs. 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. For the next steps, you will need to have four terminals open. Change the directories of one of the terminals to the "game" directory, then run game.exe with "./game.exe". In another terminal, change directories to "gui" directory, then run the GUI with "java -jar gui client.jar", and then press "Connect" on the display that comes up. In the third and fourth terminals, change directories to the "team" directory. In each one run the bot with "./botname.exe localhost 10500" where botname is the name of your executable bot file.

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, moving, and deleting bullets as well as moving and deleting powerups.
- UFOMechanics - Provide functions for creating, moving, and deleting UFOs as well as handling any damage to the UFOs and creating scattered powerups when a UFO is destroyed.
- CollisionMechanics - Functions for handling collisions between players, bullets, powerups, and UFOs. Also handles any deletions as a result of collisions.

These modules are continually called in game.ml when 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 a few invariants: they are given valid game objects as arguments, such as bullets, powers, teams, players etc. with valid properties such as position, velocity, radius, ID, color, charge, lives etc., where valid is as specified in the writeup, and that given a valid game object, the value returned is similarly valid under the rules of the game as stated in the writeup. One example would be that negative lives, negative score, or other such invalid values are assumed to not be passed to the modules as per the invariants. As another example, functions in team that edit team data return teams that are correct according to game specifications.

3.2 Architecture

Most of the 5 modules do not depend on each other so that one can be changed and another is not affected.

- PlayerMechanics does not depend on any other modules as it only edits players, which require no extraneous functions from other modules. TeamMechanics then uses the PlayerMechanics module to edit the team player when necessary. TeamMechanics is the only dependent module, and as PlayerMechanics is not used elsewhere, they are both within the same file with PlayerMechanics first and then TeamMechanics. Having them both in one file is a representation of their relationship.
- The UFOMechanics module also has no dependencies, as for UFOs it simply handles UFO movement and damage until it is destroyed, and then it creates powerups with velocities toward the players after a UFO

is destroyed. Although it needs to have players from the teams passed in, this is handled in game.ml in order to keep this independent of any other module.

- The WeaponMechanics module does not rely on any of the other modules as it merely creates bullets and moves and removes bullets and powerups that are passed in from game.ml.

- CollisionMechanics similarly does not require any other module as valid game arguments are passed in from game.ml, which makes all the calls to other modules prior to sending in arguments to CollisionMechanics. CollisionMechanics checks for collisions and processes any outcomes of collisions (such as removing bullets that hit UFOs), and so only requires that valid game objects, such as bullet lists, power lists, UFOs, etc are passed in.

- Game.ml uses all of the modules except for PlayerMechanics in order to process and retrieve all changes to the game state.

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 and if the time limit has been reached.

- 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 long each player is invincible after getting hit by a bullet.

- Red and blue bomb invincibilities - Track how long each player is invincible after using a bomb. The two invincibilities are specifically tracked separately as they have different effects. One example is removing bullets upon graze, which is for bomb invincibility only.

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 as specified in the writeup. This made our code slightly complex and not as elegant as we would have liked in game.ml, since our handle time function essentially became many edits in a row to the game record, and sometimes several edits to one record. Although it was not as elegant, all changes were successfully made in the correct order and did not cause any bugs in anything further down in the order. At the same time, having everything sequential made debugging easier as we could test parts of our code until we found where the bug would occur. 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 mainly 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, have been easier to write, and would need to be edited far less for readability, reusability, and style if we had built the modules first. Luke Benning wrote most of the initial module and game code. Anton Gilgur made significant additions to the game and module code, wrote the graphics code and bot, edited for style, reusability, and readability, fixed errors in previous code, and tested and debugged any and all problems.

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. After a new feature, such as a new module, new edit to the record, or new game object, was added, we would extensively test the feature to make sure it produced what we expected. One of the tests we would employ was to put new code into the toplevel, give valid arguments to the new functions, and see if the output matched our expected output. Another test was to run the babybot with most of the code commented out except for the one that would test our new feature (such as only having it shoot a spread bullet, only having it move, etc).

We also brainstormed together many scenarios that could occur, no matter how unlikely (such as multiple collisions), wrote code to handle these cases, and specifically tested to make sure our code handled these cases robustly. Piazza was a very helpful resource for understanding game functionality in many of these cases (such as a bullet hitting a UFO and player in the same time step).

This approach allowed us to test all new features independently of one another, which allowed for faster and more accurate debugging (as there was less code that was tested at the same time). This also meant that after testing each feature separately, we knew that it worked correctly as our modules are nearly all indepedent of each other and we maintained the invariants after the modules were all placed together. As expected, our final tests on the entire codebase as a whole revealed only a few bugs that were due to the invariants not being held up (as we incorrectly passed invalid input). We are very confident that our testing gives good enough coverage of our codebase because of our incremental test driven approach.

5 Extensibility

Our modular design allows for easy extensibility, which was an important objective in designing our program.

The following assume that definitions.ml, constants.ml, and all match statements have been changed already to accomodate anything totally new (as they are not just new "objects," they cannot simply "extend" others, and therefore new cases need to be written out).

- New bullet types - We would simply add new helper functions to WeaponMechanics to create the new bullet types after being passed into the "deploy" function. Depending on if the bullet does anything special upon impact, it could require extra functionality being written in CollisionMechanics.
- New collectibles - If the collectible was a weapon or powerup (had the same type as a bullet), we would add functions for it inside WeaponMechanics, which already handles powerups and bullets equally as they are the same type, otherwise we could create a new, independent module for various pick up items that would have new types associated with them. In either case, the collision functionality would then be handled in CollisionMechanics.
- Bomb effects - We would code these effects into WeaponMechanics, with any collision details handled in CollisionMechanics. If the bomb did something unique to teams or player characters, then we might implement that functionality in TeamMechanics or PlayerMechanics.
- 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. It also might be valid, if it had the same type as a UFO, for its movements and creation to be handled in UFOMechanics as UFOs are already neutral entities. However, it might be easier to just create a new, independent enemies module to handle these new enemies if they are significantly different from anything already created.

6 Known Problems

None known. All game functionality, including extra problems such as the ufos and powerups, works correctly.

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, especially for testing, debugging, and designing as it was very large.

The game had many rules, and a lot of edge cases arose that weren't answered in the writeup. We'd recommend being a bit more clearer on the collision specs., as they weren't comprehensive. Testing was difficult as we had nothing to compare our game with and we did not really know how it was supposed to actually function or even look. We would recommend that at least an example video be given to compare functionality. A deterministic, and not entirely random, bot would also be helpful as what we would see on the video would be something we could replicate. 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, we 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. Similarly, creating our own design exposed to us to a variety of problems and concepts that we had to handle and reason about that we had never got the chance to do in any previous problem set. And lastly, applause to Chris Yu for spending a lot of time making this, it must have been a monumental effort! (P.S. We really appreciated the attention to detail with the graphics and sound, the hilarious prologue, all of the coding/functional jokes, and the secret passwrod!)