CS 3110 Problem Set 6: *STEAMWORMS*

Assigned: November 11, 2011

Final submission due: December 1, 2011 **(no extensions)**
Design meetings: November 17 - November 23

1 Updates

Update v1.1

Added new graphics updates in Definitions.ml. Made the corresponding changes to Netgraphics.ml. These graphics updates include adding the worm color to AddWorm and adding the UpdateWorm update to change the health. GUI updated to not crash, and give helpful error messages.

The package comes with a new constants.ml. (Most) Useless constants were removed. These constants represent constants more along the lines that you will see at the tournament. Unless otherwise stated on Piazza, you shouldn't make assumptions on the constants that you will be receiving.

2 Introduction

In this assignment, you will develop a game called *SteamWorms*. There are few constraints on how you must implement this project. This freedom does not mean you can abandon what you have learned about abstraction, style and modularity; rather, this project is an opportunity to demonstrate all three in the creation of elegant code.

You should start by carefully designing your system and presenting it at a *design meeting* where you will meet with a course staff member to discuss your design. Part of your score will be based on the design you present at this meeting.

On TBA, after the problem set is due, there will be a *SteamWorms* tournament. We highly encourage you to submit your bot programs for this tournament. There will be lots of free food and the chance to watch your bot perform live. The winner gets bragging rights and will have their name posted on the 312/3110 Tournament hall of fame. Rumor has it there will also be potentially awesome prizes involved!

2.1 Reading this document

The types referenced in this document that are not inherent to OCaml are defined in `definitions.ml`. Constants follow the following naming convention: They begin with a lowercase `c`, and the rest is a descriptive name of the constant in all caps. For example, `cGRAVITY` specifies the acceleration of gravity. The reason for this mysterious `c` in front of all the constants is that OCaml actually doesn't allow value names to begin an uppercase letter–only type constructors can do that. For all chance based constants, the value given is the percent chance that the relevant event occurs.

Also, note that some constants might not make sense initially. This serves two purposes. It ensures that your implementation of the game does not depend on any specific descriptions of the constants but rather the constant itself. Second, it allows the course staff to modify the constants in order to create a tournament environment that is balanced.

For the purposes of this problem set, assume units are standard metric units for all things related to physics, unless otherwise stated. All objects in this game will be treated as circles, which makes collision detection much simpler.

2.2 Updates to Problem Set

Updates to the problem set will be marked in red.

2.3 Point Breakdown

- Design meeting – 5 pts
- Game – 40 pts
- Bot – 30 pts
- Documentation and design – 5 pts
- Written problem – 20 pts

3 Game Rules

3.1 General Rules

The game begins with each team starting with `cTEAM_SIZE` Basic worms on their team. From here, the player can choose to either upgrade their worm to a different type by training for the appropriate promotion time. Each worm has a specific projectile that they can use, and the goal is to kill all of your opponents worms. The first team to do that wins!

3.2 The Worms

There are 6 different types of worms. Each worm has its own projectile which has a specific role. Utilize all of their abilities to achieve victory! Each of the different `worm_types` have several constants which you will find important:

- `*HEALTH`: the worm type's starting health
- `*SPEED`: the worm's max move speed
- `*PROMOTION_TIME`: the time it takes to promote to that worm type
- `*RADIUS`: the size of each worm
- `*ATTACK_COOLDOWN`: the amount of time a worm has to wait before it can attack again.

Basic

The Basic worm shoots a Bomb as a projectile. This worm has average skills. However, he is the only worm type that can be promote to other worm types. The Bomb projectile has a medium radius and is fired at a medium rate. This worm is best used to transform into the more specialized worms.

Grenader

The Grenader is a specialized siege unit. The grenade can be thrown relatively far compared to other weapons and unlike other weapons, detonates on a timer, rather than on contact with the ground. Use this worm to create havoc from a distance.

MissileBlaster

The MissileBlaster is a powerful tanky unit. He does big damage but has a slower firing rate. He also has a higher hp to make up for his lower mobility.

Miner

The Miner can set up mines across the map that detonate after a certain time. The mine can be used to limit enemy movement so you can hit with your bigger attacks!

PelletShooter

The PelletShooter is quick and mobile and can shoot pellets at a rapid speed. He can dodge projectiles more easily than other worms. However, he does much less damage compared to the other worms.

LazerGunner

The LazerGunner can shoot Lazers. Lazers are the fastest type of attack but have small blast radius. LazerGunners can effectively be used to pick off worms that are dodging particularly well.

3.3 The Weapons

Each projectile has specific constants which are relevant to it.

- `*DRAG`: the drag constant that you will use to determine the projectile's path
- `*SIZE`: the size of the projectile that you will use to detect collisions
- `*EXPLOSION_RADIUS`: the radius for which damage is applied
- `*DAMAGE`: how much damage the projectile does to worms
- `MAX*MAGNITUDE`: the maximum speed that an worm can shoot that projectile

Aside from each worm's respective projectile type, each worm can also use a Bat. The Bat is a close range instant attack that damages all enemies within `cBAT_LENGTH`. Miners are able to plant mines on the ground within `cMINE_DISTANCE`.

3.4 The Map

In order to make things a little more interesting, the land on which your worms move will not be the same every game. The points which define the land are specified by `cBOARD`.

3.5 Obstacles

As if it wasn't hard enough to hit the other teams slippery worms, there are also going to be obstacles in the way. There are two types of obstacles.

- Clouds: Clouds change the path of your projectile. They add `cCLOUD_DRAG` to your projectile's respective drag, causing projectiles to fall out of the sky much faster.
- Satellites: Satellites block the path of your projectile. If a projectile collides with a satellite, the satellite loses one health. When the satellite has no more health, it disappears from the map.

Use obstacles effectively to protect your worms from harm!

4 Game Module

This module is in charge of implementing all of the rules of the game. The game server calls the functions of this module in order to progress the game.

- `initGame`: This initializes the game into a game type. This contains all of the information for a game which has yet to begin.
- `initWorms`: This initializes the worms of your game type. Worms for the red team spawn on the left side of the map, from `x = 0` to `cRED_START` and the blue team starts on the left side of the map from `x = cBLUE_START` to `800`. Select a location at random to place the worms. Each team is initialized with `cTEAM_SIZE` Basic worms.
- `startGame`: This tells the game that all initialization is done and the game is beginning.
- `handleAction`: This is the method by which the AI will send commands to worm. This method is in charge of updating the state of your worms to reflect AI commands.
- `handleTime`: The server will call this function in order to update a game by a time step.

  Note: The game object which is passed to all of these functions is the one that was originally generated by initGame. Therefore, all modifications that you make to the game can not be done functionally. This means you need to be very careful when modifying the game object!

One Game Step

In `handleTime`, you should follow this guide in order to ensure that SteamWorms is played correctly.

1. First, check to see if the game is over. The game is over when the game has gone on for `cGAME_LENGTH` or one of the worm teams is defeated. In the case of a time out, return the team with a higher score. If the game does not end, progress one step.
2. Update all of the worm promotion statuses. Update the timers of all of the worms who are currently training and promote those who have finished their time.
3. Update the position of all of the worms. Move each worm towards the next spot that its team has queued up for it. If the team has not sent a move, keep the worm still.
4. Update the position of all of the projectiles. The projectiles should be moved according to the equations presented later in the writeup.
5. Add projectiles to the game. All of the worms which have had a projectile command sent to them will add projectiles to the map if they are off of their cooldown.
6. Handle all of the explosions in the game. Detonate all of the grenades and mines if their detonation time has expired. Detonate all other projectiles if they have made contact with the ground.
7. Remove all of the dead things from the games. This includes satellites, worms, and detonated projectiles from the game.

4.1 Game Physics

You will need to implement projectile motion with drag. To simplify the process, all masses are 1 kg, and thus all force vectors are also acceleration vectors. A good resource to review how to do projectile motion is Projectile Motion Wikipedia. This is done most simply by finding the trajectory in the x and y planes separately and then combining the results.

5 Communication

5.1 Client-Server Framework

SteamWorms makes use of a client-server framework. Under this framework, the game server is responsible for keeping track of the game state, applying the game rules, and so on. Clients (i.e., players) are run as an entirely separate processes and can keep track of whatever information they want, but they need to send messages to the server to perform game actions or receive information. Players communicate with the game server by sending information over channels. The protocol for messages is defined by the type `action` in `definitions.ml`.

There are seven types of messages defined for the client to communicate to the server:

- `QueueShoot(v)`: The client uses this to tell a worm that it should shoot with vector v after it is done performing its current shots.
- `QueueMove(v)`: The client uses this to tell a worm that its next waypoint should be position v.
- `QueueBat`: This tells a worm that its next available shot should be used to swing its bat. This function requires no arguments.
- `ClearShoot`: This clears a worm's current shot queue.
- `ClearMove`: This clears a worm's waypoint queue.
- `Promote(worm_type)`: This begins a Basic worms promotion to `worm_type`.
- `Talk` - This is used for the worm to talk on the GUI! Please keep all conversation civil.

The server should return `Success` or `Failed` depending on whether the action was legal or not. This is particularly necessary for teams that try to cheat! The client also has the ability to request information from the server about the state in order to update the game. These updates are requested with messages of type status.

- `WormStatus id`: Returns the `worm_data` for the worm labelled `id`
- `ProjectileStatus id`: Returns the `projectile_data` for the projectile labelled `id`
- `TeamStatus color`: Returns all of the information for the team labelled `color`
- `ObstacleStatus`: Returns the information for obstacles on the map
- `GameStatus`: returns all of the information related to the game

6 GUI

6.1 GUI Client

In order to view the game, you will have to set up a GUI client program. The client has been coded for you, and is located in the `client` directory. The client renders the game using Java and Swing. This module will be sufficient for simple rendering of the game, though you can make bonus modifications if you wish. To use the GUI, you must have Java.

The game server is responsible for sending graphical update messages to the GUI, as described below. The game server will listen for clients to send the updates to throughout the game. However, if a GUI connects halfway through the game, it will have missed the message that initialize the board. Therefore, if you do not start the process for the GUI before you start the processes for the teams, you will not be able to see anything from the GUI.

6.2 Sending messages to the GUI

We have provided a simple module called `Netgraphics` with functions to send graphical updates. The functions are specified in `netgraphics.ml`. Note that the `init` function is called by the `Server` module and `sendUpdates` is called regularly by the `Server` module. So in order to send a graphics update to the clients, the game module calls `Netgraphics.addUpdate` with the appropriate `update` type. The one exception to this is that the `InitGraphics(vlst)` update must be sent by itself, so the game module should send this update directly by calling `Netgraphics.sendUpdate`. The argument `vlst` will always be `cBOARD`.

6.3 Graphics Commands

	Arguments	Meaning
`InitGraphics`	vlst	Tell the GUI client to initialize the graphics in preparation for a new game starting
`DisplayString`	`id, string`	Adds the message to the GUI by the worm labelled `id`
`AddWorm`	`id, pos, health`	Adds the worm to the field at position `pos`.
`MorphWorm`	`id, worm_type`	Changes the worm labelled `id` to the specified `worm_type`
`MoveWorm`	`id, position`	Moves the worm labeled `id` to the given position
`RemoveWorm`	`id`	Removes the worm labeled `id` from the the GUI
`AddProjectile`	`id, weapon_type, pos`	Adds a projectile with label `id` to position `pos`
`MoveProjectile`	`id, pos`	Moves the projectile labeled `id` to position `pos`
`RemoveProjectile`	`id`	Removes the projectile labeled `id` from the GUI
`UpdateScore`	`color, score`	Updates the team color's score to `score`
`GameOver`	`game_result`	Updates the GUI with the games result
`AddObstacle`	`obstacle`	Adds the obstacle to its position on the GUI (note that the obstacle type contains a position)
`RemoveObstacle`	`id`	Removes the obstacle labelled `id` from the GUI
`DoBat`	`id`	Has the worm labelled `id` perform a bat hit

Remember that the GUI knows very little about the game, so it is your responsibility to make sure that the GUI is updated properly!

7 Provided source code

Many files are provided for this assignment. Most of them you will not need to edit at all. In fact, you should only edit and/or create new files in the `game` and `bot` directories (plus any edits you need to make to the compilation scripts). Here is a list of all the files included in the release and their contents.

`build*.bat`	Build scripts for the game server, teams, and GUI client (see Section 8)
`game/constants.ml`	Definitions of game constants
`game/definitions.ml`	Definitions of game datatypes
`game/game.mli`	Signature file for handling actions, time, rules, and the game state
`game/game.ml`	Stub file for actions, rules, and time
`shared/util.ml`	General use helper functions
`game/netgraphics.mli`	Signature file for sending updates to the GUI client
`game/netgraphics.ml`	Implementation of sending updates to the GUI
`game/server.ml`	Starts the game server and deals with communication
`shared/connection.mli`	Signature file for connection helper module
`shared/connection.ml`	Implementation of connection helper module
`shared/thread_pool.mli`	Signature file for thread pool helper module
`shared/thread_pool.ml`	Implementation of thread pool helper module
`team/team.ml`	Basic framework for a client to interact with a game server
`team/babybot.ml`	Stub for a team AI

7.1 Code Structure

To understand how to implement the game, you must first understand how the code we have provided you operates. One crucial aspect is the relation between the `Server` module and the `Game` module. The `Server` module deals primarily with receiving connections from the teams and calls the `Game` module for all issues related to the game rules. *You do not need to modify the* `Server` *module, graphics commands, or GUI client.* Your modifications and additions will take place in the `Game` module, `State` module, and any other modules you choose to add.

7.2 Server

At a high level, `server.ml` does the following things:

1. Calls `Netgraphics.init`, which accepts connects from GUI clients
2. Waits until for enough teams to join the game (see Section 8.1 for more details)
3. Repeatedly calls `Game.HandleTime` with the responses of the clients, outputting the state and request to the client
4. Uses the actions of team to call `Game.handleAction`
5. Uses the statuses of the team to call `Game.handleStatus`
6. Repeatedly sends graphics updates to the GUI

You will need to think carefully about how you design and implement the game to meet the `Server` module's expectations.

7.3 Game

All the functions in the `Game` module referred to above are specified in `game.mli`. Your design may require you to add or modify the type declarations or functions we have provided.

7.4 State

We strongly suggest that you have a `State` module in your final design, as the distinction between game rules and game state is significant. In other words, you should *not* put all of your code in the `Game` module.

8 Running the game

You will need four command prompts to run the game.

8.1 Game Server

From `ps6/game`, run `build_game.bat` to build the game server. In this command prompt, run `game.exe`.

8.2 GUI

The GUI is already written for you in Java. It is distributed in a jar file called `gui_client.jar`. The GUI needs the `data` folder for game assets. To run the GUI, just run the jar file (from the command line, `java -jar gui_client.jar`). Enter your Game Server connection information (by default it is localhost at port 10501), and press the `Connect` button.

8.3 Team

From `ps6/team` run `build_team.bat teamname` (or `build_team.sh teamname` for Mac/Linux systems) to build the bot in the file `bot/teamname.ml`. Next, run `teamname.exe localhost 10500`. Run this command in two separate command prompts, and watch the magic happen!

8.4 The Whole Game

9 Your tasks

There are several parts to the implementation of this project. Make sure you spend time thinking about each part before starting. Start on this project *early*. There are many things you will have to take into consideration when designing the code for each section. Remember, **no extensions!**

9.1 Design meeting

Your first task is to create a design for your *SteamWorms* implementation and meet with the course staff to review it. Each group will use CMS to sign up for a meeting, which will take place between November 13 and November 17. If you are unable to sign up for any of the available time slots on CMS, contact the course staff, and we will try to accommodate you.

At the meeting, you will be expected to explain the design of your system, explain what data structures you will use to implement the design, and hand in a printed copy of the signatures for each of the modules in your design. You are also expected to explain your initial thoughts about strategies for your player bot. In designing module interfaces, think about what functionality needs to go into each module, how the interfaces can be made as simple and narrow as possible, and what information needs to be kept track of by each module. Everyone in the group should be prepared to discuss the design and explain why the module signatures are the way they are. We will give you feedback on your design.

9.2 Implementing the game

Your second task is to implement the *SteamWorms* game in the file `game/game.ml` and any files you choose to add. Note that you should add files only to the `game` and `team` directories. You must implement the rules as described in Section 3 and handling of actions. You must also make sure that the actions units take are rendered in the graphic display using the interface detailed in Section 6. You can use the sample team program we provide to test your game, but for full testing coverage you will need to write your own tests.

9.3 Designing a bot

Your third task is to implement a bot to play the game. A very weak bot that you can use as a basis for your bot code is provided. There are many different strategies for building a good bot. This is your chance to be creative and have fun creating a good AI. We will also provide some staff bots to test against, at our discretion. Further information on the server will be provided soon.

In the next couple weeks, we will release stronger bots for you to test your own against. Use this opportunity to evaluate your own bot, so you can create a bot that will do well at the tournament. Also, feel free to test your bots against other students. It can be used as an opportunity to test your game implementation and test your bot!

9.4 Documentation

Your final task is to submit a design overview document for this project. Since this project is both large and quite open-ended regarding the way one may choose to implement it, documentation becomes even more important. Your design overview document should cover *both* your implementation of the game itself and the bot you created. In discussing your bot, you should make note of what strategies you experimented with, and what you found to be most effective.

9.5 Things to keep in mind

Here are some issues to keep in mind when designing and implementing the game:

- **You need to make a good design**. This project is both large and complicated; without spending time on making a design that is both solid and complete, you *will* very quickly get bogged down when you go to implement things. The importance of design cannot be overemphasized. Trying to write code before your have your design is a recipe for disaster on a project of this magnitude.

  Before writing any code, you should have a very clear idea of *all* of the following:

  - What concurrency issues exist and how to deal with them?
  - What information needs to be kept track of to fully represent the game?
  - How that information will be stored and accessed efficiently?
  - What the interface between your modules will be?
  - What invariants will hold between your modules?
  - Which modules will enforce those invariants?

- **Think carefully about how to break up your program into loosely coupled modules.** The program will be complex and difficult to debug unless you can develop modules that encapsulate important aspects of the game. Design the interfaces to these modules carefully so that you can work effectively with your partner and do unit testing of the modules as you implement.

- **Make sure that what is going on in the game matches what is going on in the graphics.** Updating one does not automatically update the other. If you are watching the game and something seems to go wrong, remember that it could just be the code controlling the output to the screen. Moreover, just because the graphics look correct doesn't mean the game is acting properly. It would behoove you to maintain some sort of invariant between the status of the game and the status of the graphics.

- **Problems in the game might actually be problems with the bots.** If you are using your own bots to test the actions and something seems wrong, the bots could just as easily be at fault.

- **Implement and test the actions one at a time.** Don't try to implement all of the actions and test them with one single bot. Start with easier actions and work up to the harder ones.

- **Remember to finish the game in time to write a good bot.** The bot part of the project is worth as much as the game part, so don't put off the bot part until the end. We WILL be grading it based on how well it performs against our test bots.

9.6 Final submission

You will submit:

1. A zip file of all files in your `ps6` directory, including those you did not edit. We should be able to unzip this and run the `build_game.bat` script to compile your game code, and the `build_team.bat` script to compile your bot code (i.e., you should modify the scripts to include all necessary files). This should include:

   - Your game implementation
   - Your bot, named `bot.ml`, in the `team` directory along with any files it needs to build and run

   It is very important that you organize your files in this manner, as it greatly simplifies grading.

2. Your documentation file, in `.pdf` format.

Although you will submit the entire `ps6` directory, you should only add new files to the `game` and `team` folders; the other folders should remain unchanged. If you add new `.ml` or `.mli` files, you should add them to the compilation scripts. Note again that we expect to be able to unzip your submission and run the `build_game.bat` script in the newly created directory to compile your code without errors or warnings. **Submissions that do not meet this criterion will lose points.**

10 Written Problem

After watching a confusing film about time travel during movie night, you go back to your office to figure out what actually happened. All of a sudden, someone who looks suspiciously like you bursts through the door, warning you about a disastrous future. The mysterious individual (who chooses to go by the pseudonym Raz) tells tales of a land where all CS majors work for the same Biz, operated by the all-powerful "Hair Man." Raz says you must both travel back to the future to stop the Hair Man together, but you find that the time machine used to travel back to 2011 has broken down. Now, you must help Raz repair the time machine by solving recurrences. Raz warns against relying too heavily on the Master Method, since it may not work in every case, and the Prince of the Rebels frowns upon rote memorization of different cases.

TASK: For each of the following recurrences, provide the tightest possible upper and lower bounds you can, using the asymptotic notation learned in class and section. Justify each of your answers mathematically. Assume that $R(n)$ becomes constant when we have sufficiently small n. Note that the Master Method only works on recurrences of a very particular form, which most of the recurrences below do not immediately follow.

1. $R(n) = R(n/2) + R(n/4) + R(n/8) + n/16$
2. $R(n) = R(n^{\frac{1}{3}}) + \lg \lg n$
3. $R(n) = R(n-4) + 4 \lg n$
4. $R(n) = R(n/2 + \sqrt{n}) + \sqrt{42}$
5. $R(n) = 4R(n/7) + n \lg n$
6. $R(n) = R(n-1) + 1/n$
7. $R(n) = R(n/2) + 2^n$
8. $R(n) = 13R(n/2) + 123n^{4.3}$
9. $R(n) = \sqrt{n}R(\sqrt{n}) + \sqrt{2}n$
10. $R(n) = 5R(n/5) + n/\lg n$