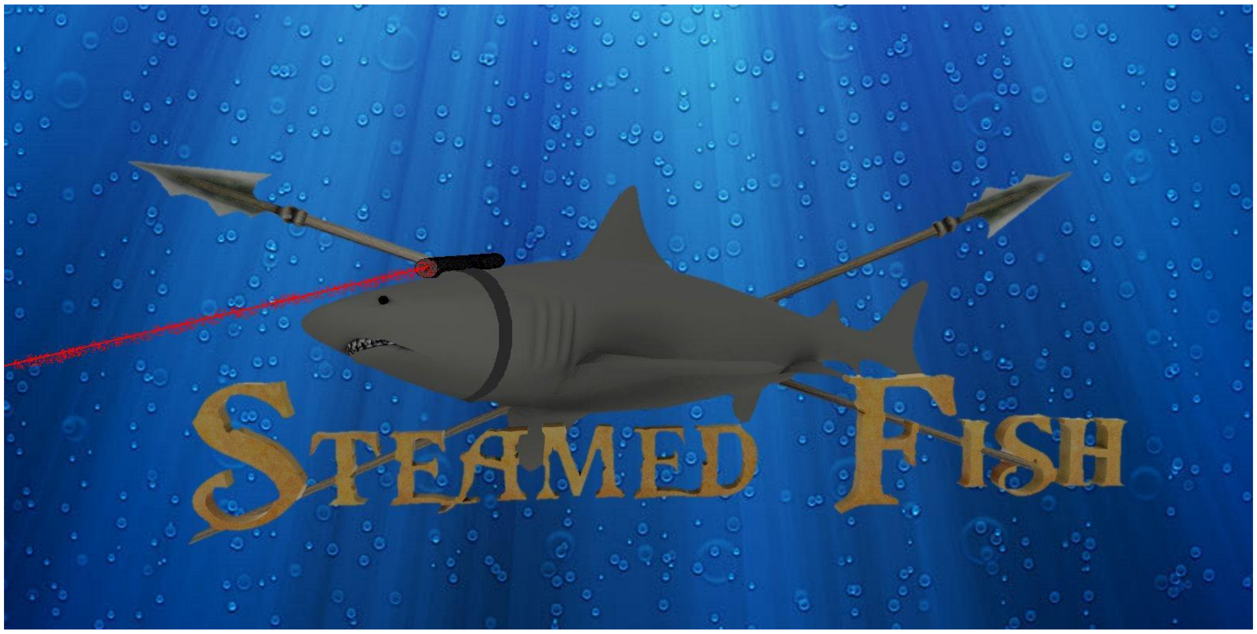
CS 3110 Problem Set 6: *Steamed Fish*

Assigned: April 15, 2010

Final submission due: May 6, 2010, 7 PM **(no extensions)**
Design meetings: TBA



0.1 Updates to Problem Set

Updates to the problem set writeup will appear in red as needed.

Section 2.4.1 - Petard explosions do not cancel each other out. If an enemy petard is activated and hit by a blast, it gets affected normally and is deactivated.

Section 2.7 - New section on donating energy.

Section 3 - Clarified game tick event order and added note on graphics updates.

Section 3.2 - Changed out of bounds rule.

Section 4.1 - GameStart now specifies how to initialize a team's spawn fish.

Section 4.2 - Vacuum beam affects all fish, not just enemies. It can be used as a strategic maneuver to pull fish out of trouble.

Section 7 - New instructions for running the game.

0.2 Supported Platforms

The game has only been tested on Ocaml 3.11, so if you are running an older version, you will likely need to upgrade your version of Ocaml. To run the gui, you will need at least JRE 1.6.

1 Introduction

In the game of **Steamed Fish**, you control a team of fish and your goal is to destroy the opposing team's fish. There are five different types of fish, each with their own abilities. They can attack other fish by colliding with them. One team wins the game when it has destroyed all of the other team's fish, or if it is ahead in score when the game time runs out. In this problem set, you will implement the game framework for the Steamed Fish game and an artificial intelligence bot that plays the game.

You start by carefully designing your system, and presenting this design at a design meeting partway through the assignment where you will meet with a course staff member to discuss your design. You are required to submit a printed copy of the signatures for each of the modules included in your design at the design meeting. Part of your score will be based on the design you present at this meeting.

After the problem set is due, there will be a Steamed Fish tournament which you are encouraged to submit your team programs to. There will be lots of free food, and the chance to watch your team perform live. The winner gets bragging rights and has their name posted on the 312/3110 Tournament hall of fame.

1.1 Reading this document

This writeup refers to a variety of constants, which are all defined in the file constants.ml. Whenever a constant is mentioned, its name is mentioned in parantheses afterwards. For example, you may see a constant like cBASE_RADIUS = 1. This means that the name of the constant as defined in constants.ml is cBASE_RADIUS, and its value is 1. You should write OCaml code using the symbolic names instead of just the current value, because we may tweak the values of the constants to improve gameplay. The types referenced in this document that are not default in OCaml are defined in definitions.ml.

1.2 Point Breakdown

Game - 50 points
AI Bot - 30 points
Written Problem - 10 points
Overview Document - 5 points
Design Reviews - 5 points

2 Game Overview

The following section describes in detail all of the features of the game which you will be implementing.

2.1 Basic Game Implementation Structure

Your game will run within a client server framework; the server is the implementation of the game itself, and the clients are the teams that connect to the server to play the game. In `game.ml`, the core module of the game server, you will be responsible for implementing all the rules of the game. You will also be responsible for keeping track of the game state. `sever.ml` will have a few concurrently running threads that will call functions in `game.ml`. When the server receives a client connection, it will start a connection thread to handle communications with that client. The clients will then use these threads to send commands to the server, either actions or status commands. `Game.handleAction` and `Game.handleStatus` will be called by `server.ml` to deal with these commands. When there are two teams connected, it will initialize the game by calling `Game.initGame()`. It will also start the threads to update the game (through `Game.handleTime`) and update the gui (through `Netgraphics.send_updates`). Creating a framework that allows you to successfully implement these three functions in `game.ml` (`handleStatus`, `handleAction`, and `handleTime`) will be your main task. Because the game is multithreaded in nature, you will need to be careful about the way your implementation deals with concurrency.

2.2 Collisions/Attacks

A fish attacks another fish by colliding with it. The amount of damage done by a certain type of fish is defined in `constants.ml`. Whenever a fish collides with a fish of another team, all fish involved in the collision lose an amount of energy equal to the attack damage done by the enemy fish. To avoid attacks occurring too quickly after another, an attack is accompanied by a delay, also defined in `constants.ml`. A delay is the time during which a fish deals no damage, even if it collides with an enemy fish. If one fish from the red team collides with many fish from the blue team, for example, the red fish's attack damage will be deducted from each of the blue fish. Damage will be deducted from the red fish for each of the blue fish involved in the collision. When a fish's energy is reduced to zero, it dies and is removed from the game. When this happens, the other team's score increases depending on the type of fish that was killed (defined in `constants.ml`).

2.3 Fish Attributes

There are a number of attributes that each fish possesses. A general description of each of is provided below:

1. **Radius** - Each fish has a particular radius (in pixels).
2. **Energy** - Energy is required to live. A fish gains energy at a fixed rate when it isn't accelerating. The rate at which a fish gains energy is type dependent and specified in `constants.ml`. For example, a petard regains energy at `cPETARD_REFRESH_RATE`. An attack (collision with an enemy fish) causes a fish to lose energy. When energy is less than or equal to 0, the fish dies.
3. **Speed** - Each fish moves at a fixed velocity based on the type of fish.
4. **Attack** - Attack is the amount of damage a fish does per attack. This depends on the fish type.
5. **Attack Delay** - Attack delay (in seconds) is the amount of time between attacks. This depends on the fish type. This means that when a fish collides with another fish while it is delayed, it will deal no damage, but it can still be damaged.
6. **Level** - This is the level of the fish. Every fish except for carnivores stays at level 0 throughout the game. Carnivores level up by killing other fish, which increases the size and energy of the fish as defined below. The max level a carnivore can achieve is `cMAX_LEVEL`.

2.4 Fish Types

There are five different types of fish. Below is a description of each type of fish, along with the default attribute values for that fish.

2.4.1 Petard

This fish is small and fast. It is weak but it can turn into an explosive device. When a petard is activated, it becomes invincible and after a certain delay causes an explosion. During this activated time, it cannot move, and does not deal any damage, so the swim queue should be cleared and any swim actions that get sent should fail. A petard can be deactivated by the controlling team, or by the vacuum or petard explosion of an opposing fish. The explosion affects the energy (lowers it) and acceleration (directs affected fish away from the explosion) of all enemy fishes in the area.

radius	cPETARD_RADIUS
energy	cPETARD_BASE_ENERGY
speed	cPETARD_SWIM_VELOCITY
attack	cPETARD_ATTACK
attack delay	cPETARD_ATTACK_DELAY

2.4.2 Carnivore

This fish is the second smallest when starting out. However, when its attack causes a fish to die, it increases in radius and energy as specified below. When a fish levels up, the energy is refilled completely. There is no other effect to the leveling of the fish.

radius	(level * cCARNIVORE_RADIUS_BONUS) + cCARNIVORE_RADIUS
energy	(level * cCARNIVORE_ENERGY_BONUS) + cCARNIVORE_BASE_ENERGY
speed	cCARNIVORE_SWIM_VELOCITY
attack	cCARNIVORE_ATTACK
attack delay	cCARNIVORE_ATTACK_DELAY

2.4.3 Vacuum

This fish can use energy to shoot a vacuum beam, pulling other fish towards it. A vacuum beam has a length and angle, radiating out in an arc. Let v be the vector specified for the vacuum, and let p be the vector from a fish that may be getting vacuumed to the vacuum fish. If the angle between the vectors is less than the vacuum angle (`cVACUUM_ANGLE`) and the distance is less than the vacuum length (`cVACUUM_LENGTH`), then the fish is affected by the vacuum beam. Note that the angle is given in radians. To compute the angle between p and v, normalize the two vectors and take their dot product, then use ocaml's built in `acos` function on that value. Affected fish get an acceleration vector pointed towards the vacuum fish with length `cVACUUM_ENERGY_EFFECT`, regardless of the distance. Like an attack delay, there is also a vacuum delay (`cVACUUM_DELAY`); this is the time after using the vacuum

beam during which the vacuum fish cannot use it. The vacuum comes at a cost (cVACUUM_ENERGY_COST), and if the vacuum fish doesn't have enough energy the action fails.

radius	cVACUUM_RADIUS
energy	cVACUUM_BASE_ENERGY
speed	cVACUUM_SWIM_VELOCITY
attack	cVACUUM_ATTACK
attack delay	cVACUUM_ATTACK_DELAY

2.4.4 Heavy

This fish is large and slow, but has a very fast attack.

radius	cHEAVY_RADIUS
energy	cHEAVY_BASE_ENERGY
speed	cHEAVY_SWIM_VELOCITY
attack	cHEAVY_ATTACK
attack delay	cHEAVY_ATTACK_DELAY

2.4.5 Spawn

This fish is large and slow. It cannot attack (when it collides with a fish it deals no damage), but it can spawn new fish. When it spawns a fish, it loses the energy of the spawned fish * cSPAWN_ENERGY_FACTOR. After spawning a fish, it activates a spawn delay, during which time another fish cannot be spawned. Note that the energy factor allows the spawn fish to spawn other spawn fish. If this happens, the new spawn fish (as well as the old one) should have its spawn delay triggered so as to prevent mass rapid infinite spawning.

radius	cSPAWN_RADIUS
energy	cSPAWN_BASE_ENERGY
speed	cSPAWN_SWIM_VELOCITY

2.5 Plankton

Plankton are stationary points on the map with a fixed radius (cPLANKTON_RADIUS), and fixed locations. cPLANKTON_SPAWN is an initial list of the positions of all plankton in the game. Each of these plankton starts out with a certain amount of energy (cPLANKTON_ENERGY_CAPACITY). Fish regain energy from plankton at a particular rate (cPLANKTON_ENERGY_RATE). Plankton enegry is depleted by the same amount of energy gobbled up by the fish. When a plankton's energy is exhausted it is removed from the game. Controlling plankton will likely be critical for a good AI.

2.6 Energy Donation

Energy can be donated from one fish to another during the game. To donate energy, a fish must send a donate energy request to the server with a recipient id and an amount. A fish can donate up to (but not including) its current energy. If the energy would cause the recipient fish to exceed its max energy, then only enough energy to fill the max energy is donated. If a fish doesn't have as much energy than is specified in a donation, no donation should occur.

3 Game Tick Event Order

The `handleTime` function in game processes a tick of the game. It should update the state of the game in the following order.

1. Update the Game time.
If the game is not over, then:
2. Handle detonations
3. Update the velocity and position for all fish
4. Handle Attacks
5. Handle Plankton Collisions
6. Remove dead fish
7. Apply level up bonuses

Otherwise, end the game.

Note You should call `Netgraphics.send_update` once, during `initGame` (this was provided). You never have to call `Netgraphics.send_updates()`, as that function gets called repeatedly on a separate thread by the server once the game begins. In terms of GUI updates, your responsibility is adding all the appropriate calls to `Netgraphics.add_update` whenever a GUI-related state change occurs.

3.1 Handling Detonations

When handling detonations, the game should go through each petard fish, check if it has been activated and if the activation time is over. If so, the petard should detonate. If the petard detonation hits an enemy fish (the distance between the fish is smaller than `cPETARD_BLAST_RADIUS` and v is the vector from the petard to the enemy, the enemy fish loses energy equal to (`cPETARD_ENERGY_EFFECT` divided by the magnitude of v). The enemy should have the following velocity vector added to its current velocity: (`cPETARD_VELOCITY_EFFECT` divided by the magnitude of v) * v. Dead fish that result from the detonation should be removed. Note that this will cause enemy petards that are activated in the blast radius to be deactivated.

3.2 Updating velocity and position for each fish

Each fish has a queue of waypoints they are trying to swim towards (one at a time, in order). Fish move at a fixed velocity, based on their type (see `constants.ml`). When there is a swim waypoint in the queue, you compute the vector from the current position towards that waypoint. If the fish would swim past a waypoint on a given movement (the swim velocity is larger than the direction vector), its movement should be adjusted to just swim to the waypoint (set the velocity vector to be equal to the direction vector). If the magnitude of the direction vector is larger than the swim velocity, you should normalize the direction vector and multiply by the swim velocity. In addition to the commanded movement, fish can be accelerated by the detonations of petards or the beams of vacuum fish. Velocity and position are updated in the following way:

$v' = v + at$
$p' = p + v't + \frac{1}{2}at^2$
$if(a > 0) : a' = ||a|| - (cACCEL\_LOSS\_RATE * t)$
$else : a' = 0$
Note: Set the direction of a' to the direction of a.

If the new position for a fish is out of bounds, you should set the position as the nearest valid position on the board. The velocity is not affected. (They are NOT stopped as suggested before).

3.3 Handling Attacks (Fish Collisions)

In handling collisions, the game should go through all the fish and for every attack (collision where the opposing fish is allowed to attack) and reduce the energy accordingly. Collisions should be processed this way until all fish have been traversed. There is no collision blocking. Collisions occur if the distance between two fish is less than the sum of their radii. However, movement is not impeded (two fish can swim `through` each other).

3.4 Scoring

If an attack kills an enemy fish, the team's score increases by `c(`$fish\_type$`)_KILL_BONUS`. For example, if the red team kills a petard from the blue team, the red team's score increases by `cPETARD_KILL_BONUS`.

3.5 Handling Plankton Collisions

When a fish collides with a plankton they regain energy from plankton at a particular rate (`cPLANKTON_ENERGY_RATE`). When the plankton's energy has been exhausted, it should be removed from the game.

3.6 Leveling up

When a carnivore makes a kill as a result of an attack during a tick, they should level up. This increases both their radius and energy as defined above. When a carnivore levels up, their health is automatically reset to full. Leveling up occurs after the removal of dead fish, so if a fish has died on a turn, it will die even if it has leveled up on that same turn.

3.7 Ending the Game

The game can end in two ways:

1. One team has no remaining fish, in which case the other team wins.
2. The game time runs out, in which case the team with the higher score wins.

4 Game Commands

Commands are the way the client interacts with the server during the course of the game. There are three basic types of commands: controls, actions, and results. The possible commands are defined in `definitions.ml`.

4.1 Controls

Controls are passed between the client and the server to set up and then end the game. As specified in `definitions.ml` a `control` is performed by sending a `command` of the form `Control of control`.

- `GameStart`
  `GameStart` signals the beginning of the game. At this point the game spawns the intial armies, giving each team `cTEAM_SIZE` spawn fish. If the team is red, each initial spawn fish is spawned at an x coordinate of 10 and a random y coordinate. If the team is blue, each initial spawn fish is spawned at an x coordinate of `cBOARD_WIDTH` - 10 and a random y coordinate.

- `GameRequest`
  A `GameRequest` is issued by a team to enter the game.

- `Team of color`
  The server responds to the team with `Team(Blue)` or `Team(Red)` so that the client knows what team they have.

- `GameEnd`
  `GameEnd` signals the end of the game.

4.2 Actions

Actions are sent from the team client to the server to execute a certain action. As specified in `definitions.ml` an action is performed by sending a `command` of the form `Action of fish_id * action`. For an action to succeed the fish_id given must be able to perform the type of action. The different actions are specified in detail below. You will need to implement all of the pieces necessary for the `handleAction` function in `game.ml`.

Command	Args	Returns	
`Swim`	(x, y)	Success or Failed	Commands a fish to swim to location (x, y). Erases the other existing locations in the swim queue.
`QueueSwim`	(x, y)	Success or Failed	Adds the location (x, y) to the queue of waypoints.
`DonateEnergy`	(fish_id $receiver\_id$, energy e)	Success or Failed	Donates e units of energy to fish $receiver\_id$.
`DoSpawn`	(fish_type $type$)	Success or Failed	Creates a new fish of type $type$ with default attributes. Reduces energy of spawn fish by (default energy for new fish * cSPAWN_ENERGY_FACTOR). Activates a spawn timer during which no other fish can be spawned
`DoVacuum`	(vector v)	Success or Failed	Shoots a vacuum bean in a direction v. Beam should affect all fish (enemies and allies)proportional to the distance along the axis of the beam and across it.
`Talk`	string s	Success	Outputs the string s to the chat area

4.3 Results

Result messages are sent by the server to the team client in response to action messages. The server will respond with a command of the form: `Result of fish_id * result`. The id passed back is the fish the result refers to and the result will be one of the options specified below (and in `definitions.ml` as the `result` type).

- `Success` - The action was successful.
- `Failed` - The action failed.
- `SpawnResult of fish_id` - The spawn was successful and the id is the id of the new fish that was spawned.

4.4 Status

Status messages are sent by the client to the server to determine the status of some aspect of the game. These commands are of the form `Status of status`. The types of status request are specified below (and in `definitions.ml` as the `status` type). You will need to implement all of the necessary functionality for `handleStatus` in `game.ml`.

- `FishStatus of fish_id` - requests the information about a specific fish
- `TeamStatus of color` - requests the information about a specific team
- `PlanktonStatus` - requests the information about plankton
- `GameStatus` - requests information about the game as a whole

4.5 Data

Data messages are sent by the server to the client in response to a status message. These commands are of the form `Data of data`. The types of data returned are specified below (and in `definitions.ml` as the `data` type).

- `FishData`(id, p, v, e, l, st) - Result of a FishStatus request. id is the fish id. p is the current position. v is the current velocity. e is the energy. l is the level of the fish. s is a status variable, either `Normal` or `Activated of timer`, where timer is the time left until petard detonation. This is to inform the AI of petard activations. See `definitions.ml` for the status type.

- `TeamData(`$sc, (id, p, v, e)$ `list)` - Result of a TeamStatus request. sc is the score. The values in the list correspond the the same values as in FishData for each fish on the team.
- `PlanktonData((`$id, p, e)$ `list)` - The PlanktonData returns a list of all the plankton, each described by an id id, a position p, and an energy e.
- `GameData(`$t1, t2, p, t$`)` - Result of a GameStatus action. $t1$ is the TeamData for the red team. $t2$ is the TeamData for the blue team. p is the plankton data as specified just above. t is the time remaining in seconds.

5 Communication

5.1 Server, UI, and Client Interaction

The entire game architecture is split into the server (or game server), client, and GUI client. The game server is in charge of managing the game state. It updates the game state during every tick, processing actions from clients and sending graphical updates to the GUI client. The client (AI) represents an entire team. Clients are responsible for playing the game, by sending game actions to the game server. It should give orders for all the fish. It can send one or more commands to the game at a time. Every command sends a response, some with information and some with just an acknowledgement. The server sends graphical updates to the GUI client. More specifically, it informs the GUI client of game updates that require changes in the front-end. The GUI client maintains a game state and updates it based on graphics commands sent from the server. It updates the display based on changes to the game state.

5.2 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 `gui` directory. The game server is responsible for sending graphical update messages to the GUI, as described below. Importantly, the game server will try and connect to the GUI and exit if no client is found after a certain period of time, so be sure to run a client when you run your server.

5.3 Building the GUI Client

The GUI will be provided to you as a Java jar file. You can use `gui_client.bat` in the `gui` directory to run the GUI.

5.4 Sending messages to the GUI

We have provided a simple module called Netgraphics with functions to send graphical updates. The graphics updates specified below can be sent to the gui using the `Netgraphics` module. In general, you will will call the updates in the following way: `Netgraphics.add_update(update)`. The Server will then send the updates as a batch. The only exception to this is `InitGraphics`, for which the proper calling is specified below.

- `InitGraphics`: Initializes the GUI. Should be called once using `Netgraphics.send_update` within the call to `initGame`. This code is provided.
- `DisplayString(`id, s`)`: Display a string s from fish id in the chat window.
- `AddPlankton(`id, pos, en`)`: Report a new plankton to the gui with id id, position pos and energy en.
- `UpdatePlankton(`id, en`)`: Reports the update of energy en for plankton id.
- `RemovePlankton(`id`)`: Remove plankton id from the gui.
- `AddFish(`id, c, ft, pos, vel, en`)`: Report a new fish to the gui. c is the color, ft is the type.
- `UpdateFish(`id, pos, vel, en`)`: Report updated fish information to the gui.
- `UpdateLevel(`id, inc`)`: Adds inc to the current level of fish id

- `RemoveFish(`id`)`: Remove fish id from the gui.
- `UpdateScore(`c, sc`)`: Update the score for the team with color c. sc is the amount to add to the existing score.
- `GameOver(`c`)`: Report a game over in favor of a team. c should be a color option, None to indicate a tie game.
- `ReportTimeLeft(`t`)`: Report how much time is left to the game, in seconds.
- `ActivatingUpdate(`id, tl`)`: Report or update activating petards by updating their time left tl until detonation. This is used for graphical effects.
- `DeactivateUpdate(`id`)`: Report the deactivation of petard id.
- `DetonateUpdate(`id`)`: Report the detonation of petard id.
- `VacuumUpdate(`id, p`)`: Report a vacuum action. p should be the absolute position of the target of the vacuum, such that the vacuum travels along the vector from the vacuum fish to p. The magnitude doesn't matter since the vacuum magnitude is fixed.

6 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 `team` directories (plus any edits you need to make to the compilation scripts). The files are in `ps6.zip` which can be obtained from CMS.

6.1 Code Structure

To understand how to implement the game, you must first understand how the code we have provided you operates. The crucial aspect to understand 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 Running the game

There are four commands you need to do (in separate terminals/tabs/cmds) to run the game:

1. In the first terminal window, in the game directory, run: `build_game.bat` and `./game.exe`
2. In the second terminal window, run: `gui/gui_client.bat` -¿ connect
3. To build the AI bot, run: `build_team.bat/sh <team-name>`
4. To run the AI bot, in the third and fourth terminal windows, run: `<team-name.exe> localhost 10500` (this will start running two instances of the team)

The teams will then play against each other.

8 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.

8.1 Design meeting

Your first task is to create a design for your *Steamed Fish* 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 April 19 and April 22. 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.

By the time the meeting happens you should be very familiar with this entire writeup. You should be able to talk fluidly about what the game implementation requires and how you are going to do it. You are 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 should also be able to explain your initial thoughts about strategies for your team. 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.

8.2 Implementing the game

Your second task is to implement the *Steamed Fish* 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.

8.3 Designing a team

Your third task is to implement a team to play the game. A very weak team that you can use as a basis for your team code is provided.

8.4 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. The above link describes the components a design document should contain. A reasonable length for a design document for a project of this size would be about 4-5 pages.

Your design overview document should cover *both* your implementation of the game itself and the team you created. In discussing your team, you should make note of how what strategies you experimented with, and what you found to be most effective.

8.5 Things to keep in mind

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

- **Think carefully about your design before beginning to code**. 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 can 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, 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 teams.** If you are using your own teams to test the actions and something seems wrong, the teams 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 team. Start with easier actions and work up to the harder ones.

8.6 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 `buildGame.bat` script to compile your game code, and the `buildTeam.bat` script to compile your team code (i.e., you should modify the scripts to include all necessary files). This should include:
   - your game implementation
   - your team, named `team.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 `buildToplevel.bat` script in the newly created directory to compile your code without errors or warnings. **Submissions that do not meet this criterion will be docked points.**

9 Tournament

On TBA after the problem set is due, there will be a *Steamed Fish* tournament which you are encouraged to submit your team programs to. There will be lots of free food, and the chance to watch your team perform live. The winner gets bragging rights and has their name posted on the 312/3110 Tournament hall of fame.

10 Written Problem

Please submit your solution on CMS as `written.pdf`.

A sorted array (or vector) is an appealing data structure for storing ordered data, because it offers the same $O(\lg n)$ lookup time as a balanced binary tree but has a compact representation and a good asymptotic constant factor. Unfortunately it doesn't support fast insertion.

Mr. Clawfinger Erymanthian Krakatoa has an idea for a mutable ordered set abstraction that will be fast for small n. Instead of storing all the elements in the sorted array, Howland will maintain a separate short linked list of up to $f(n)$ elements, where $f(n)$ is some function yet to be determined.

```
type set = {sorted: element array ref, recent: element list ref}
```

When the data structure is searched, both the list `recent` and the array `sorted` (of length n) are traversed. When an element is added to the data structure, it is appended to the list in constant time. If the `recent` list becomes longer than or equal to $f(n)$ elements, the $f(n)$ elements are sorted using mergesort and then merged in linear time with the n elements, which are already in order.

a. What is the complexity of a single lookup on this data structure, expressed as a function of $f(n)$ and n? To achieve complexity $O(\lg n)$, as with a balanced binary tree, what should Howland set $f(n)$ to?

b. The goal with this structure was to make inserts cheaper. As a function of $f(n)$ and n, what is the complexity of $f(n)$ inserts into this structure, starting from an empty `recent` list? (This should cause exactly one sort and merge).

c. We can reduce both insert and lookup to an amortized complexity of $O(\sqrt{n})$. Your goal is to prove this bound using potential functions. Recall that the amortized complexity T_A of an operation changing structure s to s' is defined as the actual cost of operation T plus $\Delta\Phi = \Phi(s') - \Phi(s)$.

   Provide a Φ and a definition of $f(n)$ such that the complexity of one lookup is $O(\sqrt{n})$ and the amortized complexity of one insert is $O(\sqrt{n})$, and prove both of those facts. (Hint: Choose Φ carefully, keeping in mind that amortized complexity is closely related to $\Delta\Phi$.)

11 Karma

11.1 Game Optimizations

The simplest approach to collision detection handling would involve comparing every two entities and checking if they collide. However, this doesn't scale well when you increase the number of fish and plankton in the game. You can improve performance by creating a grid-based data structure with entries representing spatial coordinates. To check for collisions in a data structure like this, you would only need to check the nearby positions, reducing the number of comparisons required. However, this data structure is still fundamentally tied to the board size, and scales quadratically as the size increases. Ideally, we'd like a data structure that scales well with both the number of entities in the game and the size of the game board. Karma will be awarded if you can implement a data structure that meets this goal. We will stress test your game on a huge board with a large number of fish to prove it!

11.2 AI

11.2.1 King of the Hill Server

Roughly a week before the problem set is due, we will set up a King of the Hill Server that allows students bots to compete against each other (and against staff bots). Karma will be awarded for good performance on the King of the Hill Server. More details will be provided about the server when it has been set up.

11.2.2 Quality AI

Karma will be awarded for exceptional AI. We will provide a set of bots of various difficulty. To get full credit, you have to beat the simple bot. For karma, beat the hard bot. Karma will automatically be awarded to the top 3 finishers in the tournament.