CS 3110 Problem Set 6: *Steam Fortress*

Assigned: November 16, 2010 Final submission due: December 2, 2010, 11:59 PM **(no extensions)**

1 Introduction

In this assignment, you will develop a game called *Steam Fortress*, which involves two teams fighting for control of an area using units with a variety of weapons and ability. Each team will be run as a separate process, communicating through channels with a game server that executes actions and enforces the rules.

You will implement the mechanics for this game in OCaml, as well as the code for a team to play the game. We have provided you with some graphical support that you can use to display the game. Source code for getting started on this project is available in CMS.

There are few constraints on how you implement this project. This does not mean you can abandon what you have learned about abstraction, style and modularity; rather, this is an opportunity to demonstrate all three in the creation of elegant code.

On TBA, after the problem set is due, there will be a *Steam Fortress* 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, it's name is mentioned in parantheses afterwards. For example, you may see a constant like 51 (`cBOARD_LENGTH`). This means that the name of the constant as defined in `constants.ml` is `cBOARD_LENGTH`, and its value is 51. You should write OCaml code using the symbolic names (e.g., `cBOARD_LENGTH`) 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 Updates to Problem Set

Any updates other than minor fixes will be recorded here.

1.3 Point Breakdown

- Game – 50 pts
- Team – 20 pts
- Documentation and design – 10 pts
- Written Problem – 20 pts

2 Game Rules

Steam Fortress is a two-player game in which each player controls a team of units. The units fight for a sequence of control points on a rectangular board, with the goal of seizing control of the entire board. The two teams will be referred to as the red team and the blue team.

The details of how units communicate with the game server are contained in Section 3. You may wish to review this section again after having read and absorbed the information on communication.

Information on the structure of the code we have provided as a framework for implementing the rules described in this section can be found in Section 6.

Steam Fortress is played on a rectangular board of length 51 (`cBOARD_LENGTH`) squares and width 25 (`cBOARD_HEIGHT`) squares. The board is centered at $(0, 0)$. As such, a coordinate (x, y) is on the board if $|x| \leq 25$ and $|y| \leq 12$. Only integer-valued coordinates are valid. Moving to the east increases the x-coordinate, and moving to the north increases the y-coordinate.

Each square on the board is designated as high ground, low ground, or a ramp. This is referred to as the elevation of the square. The elevation of a square has an effect on both movement and targeting of abilities; for more details, see Sections 2.5 and 2.10.

2.1 Control Points

Five squares on the board are marked as control points, and each is assigned a number from 0 to 4. Each control point has a state indicating whether it is controlled by the red team, controlled by the blue team, or neutral.

The red team begins the game controlling points 0 and 1, while the blue team begins controlling points 3 and 4. Point 2 begins neutral. Game maps will be structured so that control points 0 and 1 have negative x coordinates, control point 2 has x-coordinate of 0, and control points 3 and 4 have positive x coordinates, thus creating a "red side" and a "blue side" for the board. The "most advanced control point" controlled by a team refers to the highest-numbered control point red controls or the lowest-numbered control point blue controls.

Control points can be "captured" by keeping units nearby for long enough. A point changes control by one step the red direction (i.e., from blue-controlled to neutral or neutral to red-controlled) under the following conditions:

- Point n is currently neutral or controlled by blue
- All points numbered less than n are controlled by red
- The red team controls a majority of the units within 1 (`cPOINT_CAPTURE_RADIUS`) squares of point n (i.e., in the nine squares surrounding the control point) for 5000 (`cPOINT_CAPTURE_TIME`) milliseconds

Similarily, a point changes control by one step in the blue direction (i.e., from red-controlled to neutral, or from neutral to blue-controlled) under the equivalent conditions:

- Point n is currently neutral or controlled by red
- All points numbered greater than n are controlled by blue
- The blue team controls a majority of the units within 1 (`cPOINT_CAPTURE_RADIUS`) squares of point n (i.e., in the nine squares surrounding the control point) for 5000 (`cPOINT_CAPTURE_TIME`) milliseconds

Destroyed units will respawn at the highest-numbered controlled point for red, and the lowest-numbered controlled point for blue (see Section 2.7).

2.2 Scoring and Winning

A team wins the game immediately when it controls all five control points.

If no team has won after the time limit of 300000(`cTIME_LIMIT`) milliseconds has been reached, the team controlling more control points wins. If both teams control an equal number of control points, the game is a draw.

2.3 Units

Each team controls up to 5 (`cUNITS_PER_TEAM`) units, each of which is one of three classes, and directs them to perform actions by sending appropriate messages to the server (see Section 3 for more details). Each unit on a team has an integer id, different from the id of any other unit in the game.

A unit may be either living or destroyed. Living units have a value for their health, a position on the board, and a direction they are facing.

A unit is destroyed when its health drops below zero (see Section 2.6). A unit that is destroyed is removed from the board and cannot perform most actions (see Section 3.4). A unit may be recreated (possibly changing its class) by respawning (see Section 2.7).

2.4 Initial Positioning

The initial units for a team are placed at and adjacent to control point 1 (for the red team) or control point 3 (for the blue team). The first unit created is placed on the same square as the control point; the remaining four units are placed aribtrarily on the four adjacent squares.

2.5 Movement and Turning

As a result of commands issued by the team that controls it, a unit can attempt to move.

When a unit attempts to move, the attempt is either to move forward, in the direction the unit is facing, or to "strafe" left or right. In either case, whether the move suceeds depends on the contents on the new square, and the elevation of both the old and new square.

A unit that tries to move from high ground to low ground or vice versa fails and remains where they are, as does a unit that tries to move into a square occupied by another unit. Movement that would go off the edge of the board also fails.

In other cases, the movement is successful, and the unit is immediately removed from its current square and placed in the new square. So a move is successful if both squares share the same elvation, or if either square is a ramp. Note that this means that you can effectively "go up" ramps from any direction (think of them as steam jets, if that helps).

The unit is then unable to move or turn again for an amount of time equal to 20 (`cMOVE_DELAY_SCALE`) times the unit's speed. The speeds for the the classes of units are 19 (`cSPEED_MEDIC`) for medics, 17 (`cSPEED_PYRO`) for pyros, and 20 (`cSPEED_SOLDIER`) for soldiers. (A low speed is better than a high speed).

Similarly, after turning, a unit is unable to move or turn again for an amount of time equal to 5 (`cTURN_DELAY_SCALE`) times the unit's speed.

2.6 Health

All units have a health value, which represents how much damage they can take before they die. Units start with the maximum health equal to 1000 (`cHEALTH_MEDIC`) for medics, 1000 (`cHEALTH_PYRO`) for pyros, and 1000 (`cHEALTH_SOLDIER`) for soldiers. Various abilities increase or decrease the health of affected units. A unit's health cannot go above the initial health for its class; if an ability would cause it to do so, the unit's health is simply set to the proper initial health value.

If a unit's health becomes zero or below, the unit is destroyed and must respawn. A destroyed unit is removed from the board, and all actions it attempts fail.

2.7 Respawning

A team may recreate destroyed units by sending a respawn message. After such a message has been received and 10000 (`cRESPAWN_TIME`) milliseconds have passed since the unit was destroyed, a new unit of the appropriate class is created at the closest free square to the most advanced control point controlled by that unit's team (see Section 2.1). The recreated unit keeps the same id as the destroyed unit.

If the team controls no control points, the unit is created at the closest free square to $(-(\texttt{cBOARD\_LENGTH}-1)/2, 0)$ for red, or $((\texttt{cBOARD\_LENGTH}-1)/2, 0)$ for blue (i.e., the closest free square to the middle of the appropriate edge of the board).

Respawned red units should face east. Respawned blue units should face west.

2.8 Ability Overview

As a result of commands issued by the team that controls it, a unit can use various special abilities, in addition to performing a variety of other actions (see Section 3.4 for details on the other actions).

Each unit has multiple abilities that it can use, which either assist its team or damage the opposing teams. What abilities are available to a unit is determined by whether it is a Medic, Pyro, or Soldier. After using an ability, a unit is unable to use other abilities for an amount of time (the "delay") dependent on what ability was used. Abilities also have a "cooldown", which is a length of time during which other abilities may be used normally, but the ability on cooldown may not be used.

All three classes have two abilities, one labeled "primary" and the other labeled "secondary", to enable us to simply refer to the primary or secondary ability of the unit.

2.8.1 Medic Abilities

- Medigun (primary) – The medigun restores the health of a friendly target at short range.
- Steam Pistol (secondary) – The steam pistol deals a small amount of damage to a target at medium range by hitting them with a dense packet of steam.

2.8.2 Pyro Abilities

- Steamthrower (primary) – The steamthrower deals a large amount of damage to all enemy units within a medium range cone by blasting them with scalding steam.
- Steam Axe (secondary) – The steam axe deals a great deal of damage to enemies in front of the Pyro by hitting them with an axe made of steam.

2.8.3 Soldier Abilities

- Steam Rifle (primary) – The rifle deals moderate damage to an enemy at medium to long range by hitting them with a precision burst of steam.
- Steam Rocket (secondary) – The steam rocket deals good amount of damage to all units within a circular burst at long range by shooting exploding steam.

2.9 Ability Details

The following table presents the details of the abilities in *Steam Fortress*. Delay is the amount of time (in milliseconds) afterwards in which no abilities may be used. Cooldown is the amount of time (in milliseconds) afterwards in which that particular ability cannot be used. The values for ability delay, cooldown, damage, and range are defined in `constants.ml` as `cMEDIGUN_DELAY`, `cMEDIGUN_COOLDOWN`, etc.

Ability	Delay	Cooldown	Damage	Targeting	Range	
Medigun	400	0	-125	Square	3	Only affects allies, heals
Steam Pistol	400	0	50	Square	5	
Steamthrower	400	0	125	Cone	3	
Steam Axe	400	0	250	Melee	-	
Steam Rifle	400	0	100	Square	7	
Steam Rocket	400	10000	200	Square	8	Deals 200 extra damage to the nine squares within one square of the point of impact (including the square hit by the normal damage)

2.10 Ability Targeting and Range

2.10.1 Square Targeting

Most abilities target a square. Abilities that target a square follow a path from the unit using the ability to the targetted square, and may hit other units or obstacles in that path before reaching the targetted square. See the ability details above for which abilities target a square.

Shots are modeled as projectiles taking a straight-line path, starting from the center of the unit's square and ending at the center of the targeted square. The squares that the shot goes through on its path is determined by Bresenham's Line Algorithm, a description of which can be found at Wikipedia.

All abilities that target a square have a limited range. If an ability is given a target square farther from the unit's square (in Euclidean distance) than the range of the ability, the ability goes a distance up to its range in the direction of the target square, then stops. Note that the range of an ability may be affected by the elevation of the unit using the ability and the targetting square, as outlined below. When the projectile reaches the target square, it stops and affects any unit at that position.

Other units or changes in elevation along the path a projectile takes may cause the projectile to hit an intervening object and stop, depending on the elevation of the target square and the elevation the shot was fired from. For all purposes related to targeting by square and projectile pathing, ramps are treated as high ground. Given that, there are four cases for what happens to objects along the path of the projectile:

- Unit low, target low: A unit on low ground on the path to the target is hit by the projectile. A square of high ground on the path to the target causes the projectile to stop on the square before.
- Unit low, target high: A unit on high ground on the path to the target is hit by the projectile. The range of the projectile is modified by -1 (`cLOW_RANGE_BONUS`).
- Unit high, target low: A unit on high ground on the path to the target is hit by the projectile. The range of the projectile is modified by 1 (`cHIGH_RANGE_BONUS`).
- Unit high, target high: A unit on high ground on the path to the target is hit by the projectile.

Note that the Soldier's rocket uses square targeting, but also deals damage to units in squares adjacent to the square it stops at. It does not matter what elevation those squares are at.

2.10.2 Self Targeting

Units with abilities that use square targeting cannot target their own square. A unit that fires a rocket can still be affected by the extra damage the rocket deals to units adjacent to the rocket's point of impact.

2.10.3 Cone Targeting

The Pyro's steamthrower hits everything in a cone in front of the Pyro. The cone affects all units within 3 (`cTHROWER_RANGE`) squares of the Pyro (using Euclidean distance) in a 90 degree arc in the direction the Pyro is facing, emanating from the Pyro. If the Pyro is on a ramp, units at all elevations are affected. Otherwise, only units at the same elevation as the Pyro and units on ramps are affected. The steamthrower does not affect the Pyro's own square.

2.10.4 Melee Targeting

The Pyro's steam axe hits only the squares immediately to the front, front left, and front right of the Pyro (hitting three squares total). If the Pyro is on a ramp, units at all elevations are affected. Otherwise, only units at the same elevation as the Pyro and units on ramps are affected.

3 Communication

3.1 Client-Server Framework

Steam Fortress 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., teams) are run as an entirely separate process and can keep track of whatever information they want, but need to send messages to the server to perform game actions or receive information.

Teams communicate with the game server by sending information over channels. The protocol for messages is defined by the type command in `definitions.ml`.

There are three types of messages defined in the protocol:

- control messages, which deal with starting and ending the game
- action messages, which always come with an id and cause units to perform various actions
- result messages, which always come with an id and return information requested by an action

See the sections below for details on the three types of messages.

3.2 Communication as a client

When your team first starts, it should open a connection to the server using the included Connection module, and send a GameRequest message. The server will then respond with a list of unit ids. Each Action message sent by a unit must include this id to uniquely identify the given unit. Once your main team thread has received this list, for all but one of these ids, a new thread should be spawned using Thread.create, each of which will be responsible for controlling a different unit on your team. Then, your main team thread should take the unassigned id, and begin controlling that unit. Each thread must be responsible for controlling a single unit, however units may communicate using shared data structures.

Before the game can start, each unit must open their own connection to the server using the Connection module, and send a Respawn message, providing the class that units wishes to spawn as. Once the server has received such a request from each unit, then it will send a GameStarted message to every unit that is still connected (however, it isn't guaranteed that each unit will always receive this message, as the connection could close unexpectedly). After the game has started, each unit interacts with the game world through its connection to the server.

3.3 Control Messages

Control messages are exchanged between the team client and the server to manage the beginning and end of the game.

3.3.1 Control Quick Reference

`GameRequest`	Request to start a game.
`Units`	Response to GameRequest, provides list of unit ids to use.
`GameStart`	Informs the client that the game has started.
`GameEnd`	Informs the client that the game has ended and whether the client's team won.

3.4 Action Messages

Action messages are sent by the team client to the server, and tell the server to have a particular unit perform the given action. The possible actions are defined in `definitions.ml` as the `action` type. All action messages come with an id indicating which unit is attempting the action.

For some actions, the unit performing the action is unable to perform certain other actions until a certain amount of time has past. In particular, after moving or turning a unit cannot move or turn again for some time; after using any ability a unit cannot use any ability for some time; and after using some specific abilities, a unit cannot use that ability again for some time.

Note that the effects and delay for certain actions are dependent on the type of the unit performing the action.

3.4.1 Action Quick Reference

`Move`	Move one square forward, left, or right.
`Turn`	Turns to the left or right.
`PrimaryAbility`	Uses the units primary ability.
`SecondaryAbility`	Uses the units secondary ability.
`Scan`	Returns a list of all units within a certain range.
`Inspect`	Returns the elevation, unit data, and control point data of a square.
`MyStatus`	Returns the position, health, and facing direction of the unit.
`GameStatus`	Returns the remaining time and a list of control points.
`Respawn`	Has a destroyed unit respawn.
`Talk`	Outputs a string to the chat area.

3.4.2 Action Specification

The following table describes the effects of the possible actions. There are some general cases in which things work differently, as noted here:

- If a unit is destroyed, the move, turn, and ability actions automatically return Failed. Additionally, `MyStatus` returns a health of zero (and arbitrary values for the other data).
- If a unit is not destroyed, the `Respawn` action automatically returns Failed.
- If invalid arguments are provided for any action (e.g., off-board positions for the square-targeting abilities or inspect), it should return Failed. A unit targeting its own square with a square-targeting ability counts as an invalid argument.
- If the action being performed is too soon after an action that prevents it from being used for a time, it should return Failed.
- If an action, move, or turn returns Failed, no delay should be applied.

Command	Args	Delay	Returns	
`Move`	hand option ho	Speed * 20	Success or Failed, as appropriate	If ho is None, the unit moves one square straight ahead. If ho is Some(h), the unit moves sideways one square according to h. Returns Failed if for any reason the unit remains in its original square at the end of handling the action
`Turn`	hand h	Speed * 5	Success or Failed, as appropriate	Turns a unit 90 degrees in the direction h indicates. Returns failed if for any reason the unit remains facing the original direction at the end of handling the action
`PrimaryAbility`	position p	Depends	Success or Failed, as appropriate	Performs the unit's primary ability; see Section 2.8 for details. Returns Failed only in the case of invalid arguments or the unit still being delayed
`SecondaryAbility`	position p	Depends	Success or Failed, as appropriate	Performs the unit's secondary ability; see Section 2.8 for details. Returns Failed only in the case of invalid arguments or the unit still being delayed
`Scan`	int n	None	ScanData(l)	where l is a list with one `class * health * position * bool` pair for each unit within n squares (using Euclidean distance), where the boolean value is true for allies and false for enemies
`Inspect`	position p	None	InspectData(e, uo, no, bo)	where e is a the elevation at p, uo is an option that contains the class of the unit at p, its current health, and a `bool` that is true if it is an ally, no is an int option that contains the number of the control point at p, and bo is a bool option that is None if there is no control point or the control point is neutral, true if the unit's team controls it, and false otherwise
`MyStatus`	None	None	MyData(p, hp, d)	where p, d, hp are the position, current health, and facing direction of the unit, respectively
`GameStatus`	None	None	GameData(t, l)	Where t is the remaining time in milliseconds and l is a list with one `position * int * bool option` tuple for each control point, where the bool option is None if the control point is neutral, true if the unit's team controls it, and false otherwise
`Respawn`	unitClass c	None	Success if the unit was recreated as class c, Failed otherwise	See Section 2.7 for details on respawning, and Section 3.1 for its use at the start of the game
`Talk`	string s	None	Success	Outputs the string s to the chat area

3.5 Result Messages

Result messages are sent by the server to the team client in response to action messages. The possible results are defined in `definitions.ml` as the `result` type. All result messages come with an id indicating which unit is getting the action.

Every action message is responded to with some result message. Most actions simply get back a result of either Success or Failed, but actions dealing with information receive the requested data as the result. Further details are contained in the action specification above, including when to return the various result messages and what data to include.

3.5.1 Result Quick Reference

`Success`	The action was successful.
`Failed`	The action failed.
`ScanData`	Result of a Scan action.
`InspectData`	Result of an Inspect action.
`MyData`	Result of a MyStatus action.
`GameData`	Result of a GameStatus action.

4 Maps

Games of *Steam Fortress* may be played on a variety of maps. Maps are stored in text files, and the server loads a map file at the start to use for the current game by first using the function Game.loadBoard to parse the map file into a 2D character array, and then passing that to Game.initGame.

In a map file, each square is represented by a character, and the map as a whole represented as 25 (`cBOARD_HEIGHT`) lines of 51 (`cBOARD_LENGTH`) characters each. A map file is invalid if there are duplicate control points of the same number, less than five control points, not enough lines or characters, or unrecognized characters.

The following characters have a recognized meaning in map files:

Character	**Meaning**
l	Low ground
r	A ramp
h	High ground
0-4	Control point of the given number on low ground
5-9	Control point of the given number minus five on high ground

Maps are expected to satisfy the following constraints:

- The left and right halves of the map are symmetrical
- Lower-numbered control points have lesser x coordinate than higher-numbered control points
- Control points 0 and 1 have negative x coordinate
- Control point 2 has x-coordinate 0
- Control points 3 and 4 have positive x coordinate
- Any control point is reachable by a unit starting at any other control point
- A unit standing on the control point can successfully move in any direction (assuming no other unit is in the way)

These properties give maps a consistent baseline structure while still allowing for variety.

5 GUI

5.1 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 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.

5.2 Building the GUI Client

To build the GUI client, run either `buildClientWin.bat` or `buildClientUnix.sh` depending on your operating system. The GUI client uses SDL, a cross-platform two-dimensional graphics library, in order to present the graphics of the game. In order to run the game, you will have to have the SDL dlls installed on your machine. We have provided the development libraries and precompiled the c stub files for you, so as long as the SDL runtime library is installed on your machine, you will be able to view the graphics. More information about SDL, as well as a download link, can be found at http:///www.libsdl.org. Once you have built the GUI client, you can run the program `client.exe port`, where `port` is the port on which the GUI should listen for graphical updates.

5.3 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 reguarly by the Server module.

5.4 Graphics Commands

	Arguments	Meaning
`InitGraphics`	-	Tell the GUI client to initialize the graphics
`DisplayString`	`color * string`	
Display a string to the GUI as said by a team		
`SetElevation`	`position * elevation`	Set the elevation of a position in the GUI
`SpawnUnit`	`id * color * position * direction * health * unitClass`	Display a new unit at a position with the given properties in the GUI
`UpdateUnit`	`id * position * direction * health`	Update the propeties of a unit in the GUI
`RemoveUnit`	`id`	Remove a unit in the GUI
`AddControlPoint`	`int * position * color option`	Display a new control point in the GUI
`UpdateControlPoint`	`int * color option`	Update who controls a control point in the GUI
`GameOver`	`color option`	Tell the GUI the game is over and the given team won

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 split between two zips on CMS: `ps6.zip`, which contains the code for the game server and team, and `gui.zip`, which contains the code for the GUI client. Here is a list of all the files included in the release and their contents.

`build*.bat`	Build files for the game server, teams, and GUI client (see Section 7)
`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
`game/state.mli`	Partial signature for the game state
`game/state.ml`	Stub file for the game state
`game/util.mli`	Signature file for utility functions
`game/util.ml`	Implementation of utility 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
`team/connection.mli`	Signature file for connection helper module
`team/connection.ml`	Implementation of connection helper module
`team/team.ml`	Sample team that demonstrates how to start a game and send actions
`client/*`	Files for the GUI client (see Section 5)
`client/graphics.ml`	Implementation of graphics module, based on SDL library
`client/client.mli`	Signature file for GUI client module
`client/client.ml`	Implementation of GUI client module
`boards/*`	Various sample boards, from the simplistic to the complex (see Section 4)
`data/*`	Graphics files for the GUI

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.

6.2 Server

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

1. Calls Netgraphics.init
2. Calls Game.loadBoard
3. Calls Game.initGame with the result of Game.loadBoard
4. Waits until two teams have requested a game and all units have submitted a desired class
5. Calls Game.initTeam once for each team
6. Calls Game.startGame and sends GameStart messages to all units
7. Spawns a thread that regularly calls Game.handleTime with the current time
8. Enters a loop that catches messages from the teams, spawning a thread that calls handleAction for each action message it receives

You will need to think carefully about how to structure your design of the game and your implementation of these functions to meet the Server module's expectations. Note in particular that the way Server spawns a different thread for handling each action creates concurrency issues for the Game module (and that the code we have provided you does nothing to account for concurrency). For more on what needs to go into your design, see Section 8.4.

6.3 Game

All the functions in the Game module referred to above are specified in `game.mli`. The function loadBoard is implemented for you, and most of the other functions are left unimplemented. However, to give you more of an idea of how to start, we have provided an initial definition of the type game, and partial implementations of initGame, handleAction, and two subfunctions of handleAction (inspect and move).

Depending on your design, you may need to add to or modify the type declaration and these functions, but what we have given you is a start down one viable path.

6.4 State

We have also provided a partially specified but completely unimplemented State module, with just enough specified in `util.mli` for the partial implementations of the functions in Game to be valid. The functions currently specified do basic lookups by either unit ID or board position, both of which should be efficient. We have not defined the associated state type or implemented any of the functions.

We strongly suggest that you keep something like the State module in your final design, as the distinction between game rules and game state is significant.

6.5 Util

We have provided a utility module with a number of minor functions you may find useful. This module also includes an implementation of Bresenham's line algorithm. The functions are specified in `util.mli`.

6.6 Netgraphics

We have provided a simple module that provides functions to send graphical updates. Initialization and connection is done from the Server module. The functions are specified in `netgraphics.ml`.

7 Running the game

To run the game, you will need to download the release files from CMS. Once you have that, to run a game on the local machine, you must:

1. Run `buildGameServer.bat` to create `gameServer.exe`
2. Run `buildTeam.bat` to create `team.exe`
3. Build and run the GUI client as described in Section 5 (and leave it running)
4. Run `gameServer.exe`, providing a port number to listen on, a map file to use, the hostname to connect to for the GUI, and the port to connect to for the GUI (and leave it running)
5. Run two instances of `team.exe`, providing the address of the game server to connect to (and leave them running)

The two teams will now play out a game. Note that with the initial code you are given, the server will immediately shut down with a Failure exception due to unimplemented code.

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 Implementing the game

Your second task is to implement the *Steam Fortress* 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 2 and handling of actions as described in Section 3.4. You must also make sure that the actions units take are rendered in the graphic display using the interface detailed in Section 5. 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.

8.2 Designing a team

Your third task is to implement a team to play the game. Your team is required to have a thread for each unit that is responsible for sending action messages and receiving result messages for that unit. A very weak team that you can use as a basis for your team code is provided.

There are many different strategies for building a good team. Consider, for instance, that your units can communicate and share memory that the opposing team cannot access. Use it to your advantage to coordinate your maneuvers.

Your team will be graded on multiple criteria, focusing on its performance against a variety of test teams, strategic sophistication, and general effectiveness.

We will provide a server running our implementation of the game that you will be able to connect to, allowing you to see the game in action, test that your team works correctly, and try your team against other people's teams. We may also provide some staff teams to test against, at our discretion.

Further information on the server will be provided soon.

8.3 Documentation

Your final task is to submit a design overview document for this project, just like the ones for the previous assignments. 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 should probably be as long or longer than your design overviews for the previous assignments.

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 you dealt with concurrency, how you dealt with communication between units, what strategies you experimented with, and what you found to be most effective.

8.4 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 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. For example, start with getting the already-written `MyStatus` to work, then try another simple action like `GameStatus`.

8.5 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 `buildGameServer.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 *Steam Fortress* 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

Recall the binary search algorithm. The time required for finding an element is $O(\log n)$, but to add a new element, we require $O(n)$ time in the worst case.

Über-hacker Zoe Marti has come up with a new data structure: the Zardoz array. This data structure supports two operations, SEARCH and INSERT. Define $k = \lceil \log_2(n+1) \rceil$. We denote the binary representation of n as $n_{k-1}n_{k-2}...n_0$. A Zardoz array has k sorted arrays, $A_0, A_1, ..., A_{k-1}$. Each A_i has space for 2^i elements. If $n_i = 0$, then A_i contains 0 elements, and if $n_i = 1$, then A_i contains 2^i elements. Note that an array can't be partially filled. We also note that the total number of elements in a Zardoz array is still n, since $\sum_{i=0}^{k-1} n_i 2^i = n$. Finally, while each A_i is sorted, we do not enforce any relationship between the elements of different arrays.

1. Describe how to perform SEARCH on a Zardoz array. Analyze its worst-case running time.
2. Describe how to perform INSERT on a Zardoz array. Analyze its worst-case running time, and, using the potential method, analyze its amortized running time.