CS 3110 Problem Set 6: Steamcraft

Assigned: November 16, 2008

Final submission due: December 6, 2008, noon
Design meetings: November 18–25, 2008

1 Introduction

In the previous assignment (Problem Set 5), you developed an interpreter for a concurrent programming language. This part will allow you to put that language to good use, by developing a game called Steamcraft. In this game, two players each attempt to use solar-powered robots (bots) to capture the opponent's flag. In your version of the game, each bot will be controlled by a separate CL thread.

You will implement the mechanics for this game in OCaml, as well as the code for a game player in CL. Your evaluator from Problem Set 5 will be used to run the programs controlling the two teams. You should be able to reuse your Problem Set 5 code with only minor modification and bug fixes. As in Problem Set 4, we also have provided some graphical support that you can use to display the game. Source code for getting started on this project is available in CMS. You should keep the same partner you had for the previous assignment; consult Professor Myers if this is a problem.

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.

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.

On December 9 during study week there will be a Steamcraft tournament which you are encouraged to submit your robot team programs to. (We won't need the rest of your system.) The winner gets bragging rights and has their name posted on the 312/3110 Tournament hall of fame.

1.1 About game constants

This writeup refers to constants written in code font. For example, you may see a constant like 5 (`cWINNING_SCORE`). This means that the name of the constant as defined in `constants.ml` is `cWINNING_SCORE`, and its value is 5. The

same constants are available to CL programs in the file `constants.cl`. You should write OCaml and CL code using the symbolic names (e.g., `cWINNING_SCORE` in OCaml and `WINNING_SCORE` in CL), because we may later tweak the values of the constants to improve gameplay.

1.2 Updates to Problem Set

Any updates other than minor fixes will be recorded here.

- 11/19: Clarified how energy field is measured
- 11/19: Added details of additional features to graphics and image loading, see Section 5
- 11/21: Extended due date by 12 hours. The late penalty on this assignment will be 6% per day. It may be turned in up to 3 days late.
- 11/21,22: Changed the formulas for energy use due to acceleration and braking, see Section 2.7
- 11/23: Added documentation for TimeLeft GUI command, see Section 5
- 12/1: Corrected the arguments to `UpdatePlayer` and fixed a typo in the flag position spec.
- 12/5: Increased energy cost of being caught in an explosion. Decreased cost of making a mine.

1.3 Point Breakdown

- Design meeting – 5 pts
- World – 40 pts
- CL team (AI) – 20 pts
- Documentation and design – 10 pts
- Barrier Abstraction – 10 pts
- Complexity – 15 pts

2 Game Rules

Steamcraft is a two-player capture-the-flag game in which each player controls a team of steam-powered bots. The bots use mirrors to focus solar energy, heating a boiler that then builds up steam pressure. This steam energy allows bots to apply acceleration (by jetting out steam) and to lay high-pressure proximity mines that damage the bots of the other team. Points are scored by having a bot pick up the opponent's flag and bring it back to your own flag area.

2.1 Scoring and Winning

There are two teams, each of which controls a number of bots and has a team flag. Teams score one point for each time they capture the flag. The game ends once a team gets 5 (`cWINNING_SCORE`) points or after 180000 (`cMAX_TIME`) milliseconds have passed. If the game ends because time runs out, the team with more points is the winner. If the two teams have equal points when time runs out, the game is a draw.

2.2 Board

Steamcraft is played on a rectangular board of length 1500 (`cBOARD_LENGTH`) feet and width 1000 (`cBOARD_HEIGHT`) feet. The board is centered at $(0.0, 0.0)$, as shown in Figure 1. Given a coordinate (x, y), x is the horizontal offset from the center, and y is the vertical offset. Therefore, a coordinate is on the board if $|x| \leq$`cBOARD_LENGTH`$/2$ and $|y| \leq$`cBOARD_HEIGHT`$/2$. Note that coordinates are *floating-point*, so all of the following are valid coordinates: $(1.0, 2.0)$, $(3.0, -2.5)$, $(3.14159, 2.71828)$, $(750.0, 500.0)$. When floating-point values are reported to bots, they are rounded to the nearest integer.

2.3 Robots

In this game, each bot is controlled by its own CL thread, and each CL thread represents a bot, in a one-to-one correspondence. The CL threads control bots and interact with the game using the CL `do` expression. By performing the appropriate action with `do`, bots can set their acceleration, lay a mine, determine the position of other bots, and get a variety of other information about the state of the game.

Each bot has the following attributes at any given time: steam energy level, position on the board, velocity, and desired acceleration. Energy is a floating-point value and can go negative; position, velocity, and acceleration are pairs (x, y) where x and y are floating-point values.

Robots control their movement by setting a desired acceleration, which controls the direction and amount of steam they are jetting. This steam changes the trajectory of the bot over time. As discussed in Section 2.7, actual acceleration achieved may be less than desired acceleration.

Robots can communicate with each other using message passing in CL. Of course, this requires creating mailboxes before bots are spawned, and that are in scope in the bots' code.

If a thread terminates, the bot it represents self-destructs and is removed from the game. A good bot generally has a single infinite loop inside it somewhere, implemented as a tail-recursive function.

2.4 Teams and initial positioning

Each team begins with a single bot (at position (-600, -400), or $(-\texttt{cINIT\_LOC\_X}, -\texttt{cINIT\_LOC\_Y})$ for team 1, and position $(600, 400)$, or $(\texttt{cINIT\_LOC\_X}, \texttt{cINIT\_LOC\_Y})$ for team 2). The first team's flag rests at (-700, 0), or $(-\texttt{cBOARD\_LENGTH}/2 + \texttt{cFLAG\_POSITION\_OFFSET}, 0.0)$; the second team's flag rests at (700, 0), or $(\texttt{cBOARD\_LENGTH}/2 - \texttt{cFLAG\_POSITION\_OFFSET}, 0)$.

2.5 Spawning New Robots

A new bot is created whenever a new thread is successfully spawned. All newly created bots are placed at same location as the initial bot was at the start of the game, and begin with 5000 (`cINITIAL_ENERGY`) steam energy. The new thread created by spawning controls the new bot.

New threads may be spawned by a team *only* if the total number of threads spawned by that team so far—including the initial thread—is *less* than 5 (`cBOTS_PER_TEAM`). If the team has already spawned 5 threads, the spawn fails. At all times, each team will have less than or equal to 5 bots. Further, a bot that terminates is gone forever, and the team will have to make do with fewer total bots from then on.

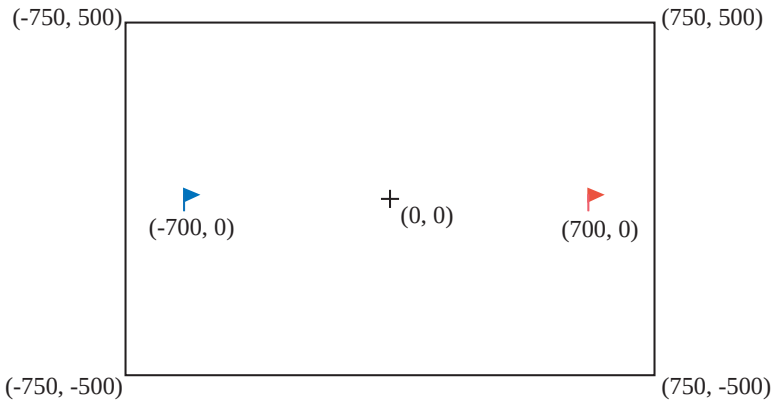

Figure 1: The board and coordinate system

2.6 Scheduling

How much each bot is allowed to evaluate is important to the fairness of the game. You are going to use the evaluator from PS5 to evaluate the CL code for the bots. In each evaluation cycle, each bot of each team is stepped exactly once.

However, this is a real-time game. The `Loop` module (`loop.ml`) (which you do *not* need to modify) will notify the `World` module via `World.reportStep` whenever some time has passed (according to `Sys.time()`). The argument to `reportStep` indicates the amount of time t (in seconds) that has passed since the last call to `World.reportStep`. Note that `reportStep` will only be called when time has actually passed, so $t > 0$.

2.7 Movement

At the end of every tick, all bots move based on their current velocity $\vec{v} = (v_x, v_y)$, desired acceleration $\vec{a}_d = (d_x, d_y)$, and current energy level E. How much the bots move also depends on t, the length of the tick.

The actual acceleration $\vec{a} = (a_x, a_y)$ applied to the bot is calculated as follows. If the magnitude of the desired acceleration $\left(|a_d| = \sqrt{d_x^2 + d_y^2}\right)$ is greater than 150 (`cMAX_ACCEL`) ft/sec^2, both components of the bot's desired acceleration are scaled down so that the magnitude of the desired acceleration is equal to `cMAX_ACCEL`.

Given the bot's current position (x, y), the bot's new position (x', y') is calculated as follows: $x' = x + v_x t + \frac{1}{2} a_x t^2$, $y' = y + v_y t + \frac{1}{2} a_y t^2$.

Moving for time t takes energy $0.005|\vec{a} \cdot \vec{v}|t$ (note `cENERGY_COST_PER_ACCEL` = 0.005). (Recall that $\vec{a} \cdot \vec{v} = a_x v_x + a_y v_y$.) The bot's velocity is also updated, based on the acceleration: $\vec{v}' = \texttt{cFRICTION}^t \vec{v} + \vec{a}t$. Note that before the acceleration is applied, the bot's velocity is multiplied by the friction constant raised to the power t. This corresponds to a frictional force proportional to velocity.

It is possible that the bot does not have enough energy to reach its desired acceleration. If the amount of energy that acceleration would use up is greater than E, both components of the actual acceleration $\vec{a}$ are scaled down so that the amount of energy the acceleration uses up is equal to E.

If a bot's motion would cause the bot to go off the board, the emergency anchor is automatically fired instead. The emergency anchor sets the bot's velocity and acceleration instantly to zero, but uses up all the energy that would be required to stop the normal way: $0.005\ v^2$. This can cause the bot's energy to go negative. The anchor is immediately retracted after deployment.

When a bot ends a tick within 50 feet (`cMINE_RADIUS`) of an active mine, that mine immediately explodes, knocking the bot over. This stops the bot and jars its valves open, leaking steam (see Section 2.9).

When a bot ends a tick within 75 feet (`cFLAG_PICKUP_RADIUS`) of the enemy flag (while the enemy flag is not being carried by any bot) and has positive energy, that bot automatically picks up the enemy flag. Since this check happens after mine explosions, a bot could enter both the flag radius and a mine radius in the same tick, go below zero energy, and not be allowed to pick up the flag.

When a bot carrying the enemy flag comes within 75 (`cFLAG_PICKUP_RADIUS`) of its own flag, while its own flag is not being carried by any bot, that bot's team gains one point. The enemy flag is then removed from the bot and returned to its initial position.

2.8 Energy

At the end of each tick, after movement and all other updates are resolved, each bot gains energy from the sun. The amount of energy gained is based on the distance from the sun, which is located above the board at $(0.0, 0.0)$ (the center of the board). A bot at distance d from the center of the board gains $P_0 t e^{-d^2/s}$ energy at the end of a tick of length t, where $P_0 = 750$ is the `cPOWER_AT_CENTER` constant, and where $s = 1000000$ is the constant `cCENTER_ENERGY_SCALING`. This is a Gaussian distribution, resulting in a bell-curve shape for the rate of energy gain as a function of the distance from the center of the board.

Robots also lose energy based on their distance from the initial position of their flag. A bot at distance d from the initial position of its flag loses $P_1 e^{-d^2/s} t$ energy at the end of a tick of length t, where $P_1 = 300$ is the constant `cPOWER_PENALTY_AT_FLAG`, and $s = 10000$ is the constant `cFLAG_ENERGY_SCALING`.

This means that to calculate the total energy change for a bot, first the gain from the energy source is added, then the loss based on distance from the flag is subtracted.

If a bot is carrying the enemy flag, it loses an additional $P_2\,t$ energy, where $P_2 = 100$ (`cFLAG_ENERGY_PENALTY`).

Robot cannot go over 5000 (`cMAX_ENERGY`) energy. Their valves start leaking at that pressure.

Robots at negative energy at the beginning of a tick are in an automatic stationary repair mode in which they absorb solar radiation and conduct repairs. They are not affected by mine explosions in that state.

2.9 Mines

Robots have the ability to lay mines. Mines are triggered by proximity sensors, so anyone coming near them, even from the same team, will cause the mine to explode, knocking over nearby bots. Fortunately, bots also have the ability to scan for nearby mines, and perhaps avoid them.

Mines cost a lot of energy to lay because the explosive power of the mine comes from the bot's own pressure. A bot is able to successfully lay a mine only if it has more than 3000 energy (`cENERGY_COST_PER_MINE`). When a bot successfully lays a mine, it loses `cENERGY_COST_PER_MINE` energy.

When laid, an inactive mine is created at the bot's current position. Each bot may only have one mine on the board at a time. If a bot successfully lays a mine while it already has another mine on the board, the older mine is removed from the board. A newly placed mine activates after 1000 milliseconds (`cMINE_ACTIVATION_TIME`).

When a bot ends a tick within a 50 feet (`cMINE_RADIUS`) of an active mine, that mine explodes, emitting a blast wave. The mine is removed from the board.

All bots with nonnegative energy within a radius of `cMINE_RADIUS` are affected by a mine explosion. They are knocked over temporarily, setting their velocity to zero. They also lose 5000 energy (`cENERGY_LOSS_EXPLOSION`) because of valve leakage. A bot can be hit by multiple mine explosions and will be affected by all of them; however, bots that were at negative energy at the beginning of the tick do not lose energy from any mine explosions.

If a bot carrying a flag is knocked over by an explosion, the flag is removed from the bot and returned to its initial position.

2.10 Flags

Points are scored by capturing the opposing team's flag. Robots can pick up the enemy flag by coming near the position of the enemy flag, and can score by bringing it close to their own flag.

When a bot comes within 75 feet (`cFLAG_PICKUP_RADIUS`) of the enemy flag, while the enemy flag is not being carried by any bot, and it has energy greater than zero, that bot automatically picks up the enemy flag.

When a bot carrying the enemy flag comes within `cFLAG_PICKUP_RADIUS` of its own flag, while its own flag is not being carried by any bot, that bot's team gains one point. The enemy flag is then removed from the bot and returned to its initial position.

When a bot carrying the enemy flag drops to zero or less energy, the enemy flag is removed from the bot and returned to its initial position.

If a bot has been carrying the enemy flag for more than 30000 milliseconds (`cFLAG_HOLD_LIMIT`), the enemy flag is removed from the bot and returned to its initial position.

3 Actions

Several available actions can be invoked using `do`. Each action returns a value. Some actions have effects on the game world.

3.1 Action Quick Reference

`A_BRAKE`	Fires the anchor, setting velocity to zero but using energy (possibly going to negative energy).
`A_SET_ACCEL`	Sets desired acceleration.
`A_LAY_MINE`	Lays a mine at the current position, using up energy.
`A_GET_POS`	Returns the bot's current position.
`A_GET_VEL`	Returns the bot's current velocity.
`A_GET_ENERGY`	Returns the bot's current energy.
`A_GET_IS_CARRYING_FLAG`	Returns whether the bot is carrying the enemy flag.
`A_GET_ENERGY_FIELD`	Returns the current power at a point.
`A_GET_FLAG_POS`	Returns the position of your own flag or the enemy flag.
`A_GET_BASE_COOR`	Returns the initial position of your team's flag.
`A_SCAN_MINES`	Returns the positions of all mines within a radius of the bot.
`A_SCAN_UNITS`	Returns the positions of all units on the board.
`A_GAME_STATUS`	Returns the current score and other data about the game.
`A_TALK`	Outputs a string to the chat area.

3.2 Action Specification

The following pages describe the actions in greater detail. Recall from PS 5 that $(v_0, ..., v_{n-1})$ represents an array literal, which evaluate to an array with the value at index ith index (for $i = 0, ..., n-1$) equal to v_i, and all other array components equal to zero.

All numerical values returned by actions are returned as ***integers***, even though some things (such as position, velocity, and energy level) are actually real-valued. The `int_of_float` function should be used to do conversion between game values and values to return to CL. The loss of precision is minor in most cases.

Some of the actions return lists to CL. Lists are represented in CL as arrays, with the length n of the list at index -1 of the array, and the elements of the list at indices $0, ..., n-1$. The value at all other indices should be 0.

Command	Args	Effects	Returns	
A_BRAKE	None	The bot fires its emergency anchor on the next tick, as described in Section 2.7. This always succeeds, but may result in negative bot energy	0	
A_SET_ACCEL	(d_x, d_y)	Sets bot's desired acceleration to (d_x, d_y)	0	
A_LAY_MINE	None	Attempts to place a mine at the bot's current location. Energy is used if mine placement succeeds.	(b, x, y)	where b is 1 if placing the mine was a success and 0 otherwise, and x and y are the coordinates the mine was placed at.
A_GET_POS	None		(x, y)	
A_GET_VEL	None		(vx, vy)	
A_GET_IS_CARRYING_FLAG	None		n	where n is 1 if the bot is carrying the enemy flag, and 0 otherwise
A_GET_ENERGY	None		E	where E is the bot's current energy level
A_GET_ENERGY_FIELD	(x,y)		r	where r is the rate of energy gain (per second) at (x, y). This includes both the solar power and the energy drain from the flag.
A_GET_FLAG_POS	b		(x, y, f)	where x and y are the coordinates of the bot's team's flag if b is 0, and the coordinates of the enemy flag otherwise; f is 1 if the flag in question is currently being carried, and 0 otherwise
A_GET_BASE_COOR	None		(x, y)	where x and y are the coordinates of the initial position of the team's flag
A_SCAN_MINES	None		l	where l is a list of the positions of all mines within 250 feet (cSCAN_RADIUS) of the bot.
A_SCAN_UNITS	None		(l_1, l_2)	where l_1 and l_2 are lists of the positions of all bots on your team and the enemy's team, respectively
A_GAME_STATUS	None		(s, s_o, t)	where s is the bot's team's score, s_o is the enemy's score, and t is the amount of time left before the end of the game (in milliseconds)
A_TALK	s	Outputs the string s to the chat area	0	This action has been partially implemented for you for debugging purposes

4 CL Features and Interpreter Updates

Some CL features might not have used much in PS 5, but will be very useful for testing the game and implementing your bot. Further, to aid in your design we have added a few useful features to CL, none of which should require any modifications to your current interpreter. We have also made some slight modifications to the structure of the interpreter, to aid in implementing some aspects of the game.

4.1 Recursive Functions

The support for recursive function already present in CL will be extremely helpful for this assignment. To review, you can define recursive functions by using the keyword `rec`, as in the example below:

```
let fact =
  rec f in
    fun n -> if n=0 then 1 else n * f(n-1)
in fact 3
```

4.2 Includes

You may wish to write code that multiple CL programs can use. You can do this using the `#include` command. The argument is the name of the file to include. Keep in mind that the path for the file should be relative to the directory from which you run SML, not the directory in which the CL file is located. If you execute SML from the project directory and want to include in a unit the constants.ch file that we have written, which is in the cl directory, you would use:

```
#include "cl/constants.cl"
```

When this line is read, the file `cl/constants.cl` is automatically loaded and its contents replace the `#include` line. You will most likely use `#include` to declare a bunch of commonly used functions. For instance, you might include a file called `functions.ch` with `#include "functions.ch"`, containing:

```
let trydo = fun x -> if x = 0 then (do x)
                     else trymove (x - 1)
in let calcpos = fun x -> fun y -> fun z -> ...
in
```

The file being included should end with `in` so that any code following the `#include` declaration is treated as the body of the `let`.

4.3 Other Libraries

An abstraction for representing and manipulating booleans is provided in `cl/bool.cl`:

`true`	Boolean constant true
`false`	Boolean constant false
`and`	A curried function that performs the logical conjunction of its arguments
`or`	A curried function that performs the logical disjunction of its arguments
`xor`	A curried function that performs the logical exclusive disjunction of its arguments

We have also provided some array manipulation functions you may find useful in `cl/arrays.cl`, and some functions for robots in `cl/botFun.cl`.

4.4 Interpreter Updates

We have updated the PS 5 interpreter by removing the old `world.mli` and `world.ml`. Further, `debug.ml` and `runcl.ml` are no longer used.

5 GUI

We have provided a module called `Graph` that handles the graphical aspects of the game by rendering using OpenGL. This module should be sufficient for simple rendering of the game, though you are welcome enhance it for karma if you wish (with the full power of OpenGL available to you, a much fancier interface is possible). To use the `Graph` module, you must install a library called lablGL, which provides OpenGL bindings for OCaml. Installation of this library is fairly simple, and we have provided instructions in `dependencies.zip`.

To initialize the GUI, you must call `Graph.init_graphics`. This opens the GUI and begins the main graphics event loop on a new thread. To communicate with this thread, we have provided two methods: `queue_event`, and `send_events`.

The GUI must be updated when the game state changes, which includes changes in score, bot movement, flag captures, etc. To update the GUI, you must queue draw events by calling `Graph.queue_event`. At the end of each tick, you must call `Graph.send_events` to send the list of queued draw events to the gui. The list of possible draw commands is included below. For all the commands, the team field is a boolean: true stands for the first team, and false for the second team. The GUI commands are as follows:

Name	Args	Effect
`DrawPlayer`	`int * float * (float * float) * (float * float) * bool * bool`	Draws a player to the screen. This should be called when spawning new processes. The arguments are as follows: bot id, bot energy, (x and y coordinates), (x and y velocities), team, holding flag. The id must be a unique identifier for the bot.
`DrawMine`	`int * (float * float) * int`	Draws a mine to the screen. The arguments are as follows: mine id, (x and y coordinates), milliseconds to activation. The id must be a unique identifier for the mine.
`RemovePlayer`	`int`	Removes the player with the given id from the display.
`RemoveMine`	`int`	Removes the mine with the given id from the display.
`UpdatePlayer`	`int * float * (float * float) * (float * float) * bool * bool`	Updates an existing player. The arguments are the same as DrawPlayer: bot id, bot energy, (x and y coordinates), (x and y velocities), team, holding flag. The player should have already been initialized with `DrawPlayer`. If only one field is being updated, you may use the more specific commands listed below.
`UpdatePlayerPos`	`int * (float * float)`	Updates an existing player's position, given a bot id.
`UpdatePlayerEnergy`	`int * float`	Updates an existing bot's energy level, given a bot id.
`UpdatePlayerFlag`	`int * bool`	Updates an existing bot's flag field, given a bot id. True means the player has the flag, false means the player does not.
`UpdatePlayerVel`	`int * (float * float)`	Updates an existing bot's velocity field, given a bot id.
`UpdateMineTime`	`int * int`	Updates the time until activation for a mine, given a mine id. The arguments are: mine id, milliseconds to activation.
`DrawFlag`	`(float * float) * bool`	Draws a flag to the screen. The arguments are (x and y coordinates) and team.
`RemoveFlag`	`bool`	Removes a flag from the screen. This should be called when a bot picks up a flag.
`SetScore`	`int * bool`	Updates the score of the game. The score begins at (0,0), so this should be called when bots score. The arguments are: score and team.
`TimeLeft`	`string`	Updates the amount of time left to play. The argument is the string to display for how much time is left.
`GameOver`	`bool option`	Has the GUI display the game end picture. If the argument is None the game is considered a draw. Otherwise, true indicates a win for the first team and false indicates a win for the second team.
`Talk`	`string * bool`	Sends talk commands to the GUI. The string represents the sentence that the bot utters, and the boolean is the team of the bot.

5.1 Robot Images

We have also provided support for displaying images for bots. The graphics module looks for files `bmp/team1*.bmp` and `bmp/team2*.bmp` and draws those images to represent bots of the first and second team, respectively. If there are multiple images with file name starting with `team1` or `team2` in the `bmp` directory, the graphics will use different images for different bots. We have provided default images in the `bmp` directory with the proper names.

If a bot has no energy, the graphics module will look for a file with the same name but with `_tired` before the `.bmp`, and display that instead. (So if you had one image named `team1a.bmp`, and also had `team1a_tired.bmp`, when the bots using the `team1a.bmp` image went to negative energy, the graphics would display `team1a_tired.bmp` for those bots). If no `_tired` image is found for a given image, the normal image will be used throughout.

Images are rotated based on the velocity of the bot. For the rotation to accurately reflect the direction of the robot, images should be "facing" to the right.

Images for use by the game:

1. Must be bitmaps (`.bmp`). Several programs can save in this format, including MS Paint.
2. Must be 2^k by 2^k pixels in size, for some k. It is probably best to keep things below 256×256.
3. Treat black (i.e., RGB value $0, 0, 0$) as transparent. If you want to make a bot image that uses black, instead use something very close to black (like RGB value $0, 0, 1$).

If you want to switch images, just move the old images elsewhere and put the new images in their place.

6 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.

6.1 CL interpreter

For the game to work, the CL interpreter must be correct. For the game to work well, the CL interpreter must be reasonably efficient. We are not asking you to do any new implementation work on the CL interpreter, but you are expected to fix any bugs in the interpreter that you submitted for Problem Set 5, and make it run at a reasonable speed.

We have added new functions to some of the interpreter files to ease implementation of certain aspects of the game. For the updated interpreter files that are included in the PS6 download, you should merge any changes you made into the files. For the other interpreter files, you should simply copy over your files from PS5.

6.2 Designing the world

Your first task is to create a design for your *Steamcraft* implementation and meet with the course staff to review it. Your second task is to implement the *Steamcraft* world in the files `world/world.ml` and `world/game.ml`, and any files you choose to add. Note that you should add files only to the `world` and `cl` directories. You must implement the actions listed in Section 3. You must also make sure that the actions bots take are rendered in the graphic display using the interface detailed in Section 5. You can use the sample bot program we provide to test your world, but for full testing coverage you will need to write your own tests.

6.3 Designing a team

Your third task is to design a CL team in a file `cl/team.cl`. To receive any credit, your team must be able to consistently beat the team provided by the course staff, which is a very weak team. You will be graded based on how well your team performs and how effective its strategy is against a number of test teams, not just the one team provided.

Your bot should be named `team.cl` and should be in the `cl` directory of the project zip file you submit (see Section 6.7).

There are advantages to coordination between your bots. Think about how to use the message-passing features of CL to make your bots more effective.

6.4 Documentation

You should 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.

6.5 Things to keep in mind

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

- **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 world 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 world is acting properly. It would behoove you to maintain some sort of invariant between the status of the world and the status of the graphics.
- **Problems in the world 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 a simple action like `A_GET_ENERGY`.

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

6.6 Design meeting

For this assignment, there will be a *design meeting* partway through the assignment. Each group will use CMS to sign up for a meeting, which will take place between November 18 and November 25. 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 are expected to explain the design of your system, and hand in a printed copy of the signatures for each of the modules in your design, in addition to giving a brief description of the design of your CL bots. 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.

We strongly encourage that you come discuss your design with the course staff during consulting and office hours, both before and after the meetings.

6.7 Final submission

You will submit:

1. A zip file of all files in your `project` directory, including those you did not edit. We should be able to unzip this and run the `buildToplevel.bat` script to compile your code (i.e., you should modify it to include all the files necessary file). This should include:
    - your world implementation

- your bot, with the bot named `team.cl` in the `cl` directory along with all the custom libraries it uses (if any)

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

2. Your documentation file, in `.txt`, `.pdf`, or `.doc` format.

Although you will submit the entire `project` directory, you should only add new files to the `world` and `cl` 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.**

7 Tournament

Sometime during finals period, most likely on Tuesday, December 9, we will hold a competition for students who wish to compete. Each group may submit a CL team that will play against other students' teams. Details on the tournament time, location, and the submission procedure will be available later.

8 Provided source code

Many files are provided for this assignment. Most of them you will not need to edit at all. In fact, other than merging your evaluator from PS5 into the `eval` directory, you should only edit and/or create new files in the `world` and `cl` 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.

`bmp/*`	Support for bot images
`bmp/team1.bmp`	Image loaded for the first team
`bmp/team2.bmp`	Image loaded for the second team
`parser/*`	Updated versions of the lexer and parser for CL
`eval/*`	Where you should put your evaluator from PS5
`world/constants.ml`	Definitions of game constants
`world/world.mli`	Signature file for handling actions and world state
`world/world.ml`	Functions for handling actions and world state
`world/game.mli`	Signature file for the functions contained in the game
`world/game.ml`	Handles the game state
`world/graph.mli`	Signature file for the graphics functions
`world/graph.ml`	Handles graphics
`world/loop.ml`	Starts and continues the main game loop
`world/*`	Other utility files for the world
`cl/simple_bot.cl`	Sample team program that you need to beat
`cl/*.cl`	CL libraries and other files

9 Running the game

To run the game, you will need to download the release files from CMS and copy your evaluator from PS5 into the `eval` directory. You will also need to install the GUI dependencies from `dependencies.zip` in CMS.
Once you have that, these are the steps you take to run a game on the local machine.

1. *Compile the game*. We have provided a `buildTopLevel.bat` script (which can be run on both Windows and Linux) that creates a toplevel environment that includes the game code, which you can then run using the script `runTopLevel.bat`. If you create new files that your implementation uses, you will need to add them to these scripts. If you added files to the PS5 evaluator, you will need to add them to these scripts.

2. *Start the game.* After you have built and started the toplevel, you can start a game between two teams by running `Loop.start "team1.cl" "team2.cl"`, where `team1.cl` and `team2.cl` are files containing the CL code for the two teams you want to play. This will initialize the GUI and immediately run the game.

10 Implementing a barrier abstraction in CL

Your third task is to use CL to implement a standard synchronization primitive called a *barrier*, corresponding to this OCaml specification:

```
(* A barrier is a synchronization primitive for a group of n threads.
 * Any thread from the group that reaches the barrier must block
 * until all n threads have reached the barrier. Then all threads in the group may proceed. *)
type barrier
(* makeBarrier(n) creates a barrier for n threads. *)
val makeBarrier: int -> barrier
(* waitB(b) causes the current thread to block
 * until the required number of threads have all called waitB(b). *)
val waitB : barrier -> unit
```

Your barrier implementation should be submitted to CMS as `barrier.cl`. Your file must be a CL header file that works with `#include`; if you put `#include barrier.ch` at the top of another CL file and run it, it should work and have access to all the barrier functions.

11 Written Problem

In addition to the game and bot implementation tasks described above, this project also includes a written problem on amortized complexity. This written question should be submitted to CMS, in `.txt`, `.pdf`, or `.doc` format.

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.

Überhacker Zoe Marti 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, Zoe 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.

1. 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 Zoe set $f(n)$ to?

2. 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)

3. 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, and use them to show that the complexity of one lookup is $O(\sqrt{n})$ and the amortized complexity of one insert is $O(\sqrt{n})$. (Hint: Choose Φ carefully, keeping in mind that amortized complexity is closely related to $\Delta\Phi$.)