CS 3110 Problem Set 6: *Steam Kart*

Assigned: November 13, 2009

Final submission due: December 4th **(no extensions)**
Design meetings: November 17th - November 20th

1 Introduction

In this assignment, you will develop a game called *Steam Kart*, a variation on the game Mario Kart. Each car 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 car 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.

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

On December 13, after the problem set is due, there will be a *Steam Kart* tournament which you are encouraged to submit your bot programs to. There will be lots of free food, and the chance to watch your bot perform live. The winner gets bragging rights and will have their name posted on the 312/3110 Tournament hall of fame.

1.1 Reading this document

The types referenced in this document that are not default in OCaml are defined in `definitions.ml`. Constants follow the following naming convention: they begin with a lowercase `c`, and the rest is a descriptive name of the constant in all caps. For example, `cTRACK_WIDTH` specifies the width of the track. The reason for this mysterious `c` in front of all the constants is that OCaml actually doesn't allow value names to begin an uppercase letter - only type constructors can do that.

1.2 Updates to Problem Set

- 11/15: Added note that the `InitGraphics` graphics update must be sent by itself, and revised what the server module does.
- 11/15: Added `Countdown` to the update type.
- 11/16: Added various clarifications based on questions in the PS6 overview sections:
  - Armors protect you from tesla coils
  - Only pass activated hazard items to `Movement.update`
- 11/19: Added clarification about `ScanData` and lap number.
- 11/24: Fixed typo in the specification of `ScanData`. All positions should be in (t, w) coordinates, not (x, y) coordinates.
- 11/26: Switched default port numbers (for KOTH server).

1.3 Point Breakdown

- Design meeting – 5 pts
- Game – 35 pts
- Car – 35 pts
- Documentation and design – 10 pts
- Written problem – 15 pts

2 Game Rules

Steam Kart is a multiplayer game in which each player controls a car. The cars race around a track, attempting to complete `cLAPS` laps first. They can pick up items which give them special abilities they can use.

The details of how the cars communicate with the game server are contained in Section 3. You may wish to reread 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.

2.1 Board

We have provided a type `board` to represent the board, as well as some useful functions for them in `definitions.ml`. You will not need to modify the Board module (however, you may if you'd like). Boards will be loaded from text files as described in Section 4.

2.1.1 Track

The track is represented as a closed piecewise smooth parametric function. If you are unfamiliar with these terms, the Wikipedia's articles on parametric functions and piecewise functions give nice descriptions for those terms. Smooth means that the derivatives are continuous, and closed means that the beginning and the end of the track are joined. These functions represent x and y in terms of t, and represent the center of the track. The tracks have constant width (`cTRACK_WIDTH`). A position on the track is represented in (t, w) coordinates. Intuitively, t represents how far along the track the position represents, and w represents the position across the track. w will be represented as a float in $[-1/2, 1/2]$, representing the distance from the center as a fraction of `cTRACK_WIDTH`. To represent these functions, we will use the expressions code from PS2. We have provided our PS2 solution so you don't have to debug yours if it doesn't work.

We will refer to a "segment" as a part of the track corresponding to one of the parametric functions in the piecewise curve.

The `(expr * expr) array` in the `board` type specifies the parametric functions for each segment, in the form $(x(t), y(t))$. The first `float array` contains the domains for these functions. To explain what that means, let $[f_0(t), \ldots, f_{n-1}(t)]$ be the array of functions, with $f_i(t) = (x_i(t), y_i(t))$, and $[t_0, \ldots, t_n]$ be the domain array. Then the domain of f_i is $[t_i, t_{i+1}]$. In other words, if f represents the entire piecewise function, then f is defined on the interval $[t_0, t_n]$, and:

$$f(t) = \begin{cases} f_0(t) & t \in [t_0, t_1] \\ f_1(t) & t \in [t_1, t_2] \\ \vdots & \\ f_{n-1}(t) & t \in [t_{n-1}, t_n] \end{cases}$$

Note that the length of the domain array is 1 bigger than the length of the function array. The requirement that the board is closed and piecewise smooth means that the following equations hold for all $i \in [0, n-1]$ (where $f_i'(t) = (x_i'(t), y_i'(t))$):

$$f_i(t_{i+1}) = f_{i+1}(t_{i+1})$$
$$f_i'(t_{i+1}) = f_{i+1}'(t_{i+1})$$

where just to make the notation easier, $f_n = f_0$. In other words, f_i ends where f_{i+1} begins, and the derivative of f_i at the end of its domain is the same as the derivative of f_{i+1} at the beginning of its domain (ie, there are no corners).

2.1.2 Road Types

There are four road types: `Speedup, Normal, Slowdown, Water`, in order of fastest to slowest. They will each have an associated friction used in the physics module to calculate the new velocity in terms of the currect velocity and acceleration.

Each segment is split (parallel to the direction of the track, like lanes of a road) into 4 pieces. Each piece of each segment has its own road type. The outer two pieces will have width 1/8 of the width of the track (`cOUTER_LANES_WIDTH`), and the inner two pieces will have width 3/8 the width of the track (`cCENTER_LANES_WIDTH`). The `board` type contains a `(roadtype * roadtype * roadtype * roadtype) array` to store the road types for each segment.

2.1.3 Item Box Rows

Item boxes are boxes that cars can hit to receive an "item", which gives them some ability, as described in Section 2.5. Just like in Mario Kart, item boxes appear in rows across the track at various points. How they will be placed in a row will be discussed in Section 2.5.4. But the locations of these rows are stored in the `board` type, in the second `float array`. The locations of the rows are specified by their t coordinate.

2.1.4 Time Limit

The `board` type also stores a float that corresponds to the time limit for a race on the board (in seconds). The reason this is stored in the board type and not just a constant is that some boards will be longer than others. So a reasonable time limit for one board may be way too short for another.

2.2 Winning

A car completes the race when it finishes `cLAPS` laps. When that happens, the car is instantly removed from the board, and no longer has any effect on the game. The game continues until either all cars have completed the race, or until the time limit for the board has passed, whichever comes first. If the game ends because time runs out, then the cars a placed according to how close they were to finishing the race. In other words, if car A and car B both did not complete the race, but car A completed more laps than car B, then car A places ahead of car B. If they both completed the same number of laps, then the car with the higher t value wins. If they have exactly the same t value (which probably won't ever happen), then just arbitrarily pick who wins.

2.3 Cars

Each car is controlled by its own process. The car is controlled by sending messages to the game server, as discussed in Section 3. Each car has a unique id, which is determined by the game server and sent to the AI when the car connects to the game server.

In addition to its unique id, each car has the following attributes:

1. position on the board, in (t, w) coordinates
2. velocity
3. acceleration
4. items that are buffered, held or currently being used by the car
5. lap number
6. character

The first three attributes are all `float * float` tuples, the fourth is discussed in Section 2.5, the fifth is the number of laps completed so far (it starts at 0, and the car is finished when its lap equals `cLAPS`), and the last is discussed in Section 5.3. Note that the position of the car is the position of the center of the car.

Cars control their movement by specifying their acceleration as a tuple (a_x, a_y). How their position is updated is discussed in Section 2.6.1. Cars can also use items and receive information about the area around them.

2.4 Initial Positioning

In *Steam Kart*, there is a maximum of four cars in a race. The initial positioning is in a row along the track at $t = 0$, with the first car to join the game at $w = 1/8$, the second at $w = -1/8$, the third at $w = 3/8$, and the fourth at $w = -3/8$.

2.5 Items

2.5.1 Items Overview

Just like in Mario Kart, item boxes appear throughout the map. A car can acquire an item by driving into an item box. The items give the cars special abilities. The items are as follows: Oil Slicks, Pressure Mines, Steam Shells, Turbo Encabulators, Tesla Coils, and Armors. For those of you familiar with Mario Kart, here is the conversion from *Steam Kart* items to Mario Kart items:

***Steam Kart*:**	**Mario Kart:**
Oil Slick	Banana
Pressure Mine	Fake Box
Steam Shell	Red Shell
Turbo Encabulator	Mushroom
Tesla Coil	Lightning Bolt
Armor	Star

Cars can have up to two items at one time: a buffered item and a held item. When a car hits an item box, the item is placed in their "buffer". The items that represent tangible objects, as opposed to abilities, are "holdable". These items are the steam shells, oil slicks and pressure mines. The best way to think about this is that they pick up the box and put it in their backpack. However, if they already have a buffered item, then their backpack is full, so they can't keep the item: they just pick it up and throw it away. If the item in their buffer is a holdable item, they can take it out of their backpack and hold it in their hand. This means that if a car hits an item box when they are just holding an item, the new item is added to their buffer.

A held item also protects the car from other tangible items. Continuing the above analogy, if a car is holding an item, it uses it to block other tanglible items from affecting it. However, when that happens, the held item breaks, so it loses its held item.

When a car uses an item, it is instantly removed from its buffer/hand (depending on which item the car used).

2.5.2 Item Details

Below is a detailed description of what each item does:

- `OilSlick`: A bottle of oil that the car can drop on the track, trying to cause other cars to slip. When used, an oil slick is placed at the car's current position. It has an activation time of `cOIL_SLICK_ACTIVATION_TIME` seconds, meaning it takes `cOIL_SLICK_ACTIVATION_TIME` seconds until cars can slip on it. The activation time is to prevent the car from instantly being hit by the item after it drops it. When a car hits an oil slick, the car crashes, which will be defined in Section 2.6.2, and the oil slick disappears from the track.

- `PressureMine`: This is very similar to an oil slick, except that once used, it looks like an item box. As described later, when other cars scan the board around them, pressure mines appear in their list of item box positions.

- `SteamShell`: When a car shoots a shell, the shell chases the car immediately ahead of the car with speed `cSTEAM_SHELL_VELOCITY`. When the shell hits a car (any car, not necessarily the one it is chasing), it causes that car to crash. If the shell hits a wall, the shell disappears. If the shell hits another item on the board (including another shell), both the item and the shell disappear. Shells also have an activiation time of `cSHELL_ACTIVATION_TIME`. Note, assigning the shell's velocity is done in code we've given you; however, you need to calculate which car the shell should chase after. If there is only one car in the race, the shell should do nothing.

- `TurboEncabulator`: Gives the car an immediate velocity boost. Specifically, when a car uses a turbo encabulator, its velocity is immediately multiplied by `cTURBO_ENCABULATOR_VEL_MOD`. As discussed in Section 2.6.1, the car's velocity will be slowly lowered back to normal by the max acceleration and friction, which implies that unlike the mushroom in Mario Kart, the turbo encabulators do not have a timing aspect.

- `TeslaCoil`: The maximum speed of all other cars is decreased temporarily. More precisely, for `cTESLA_COIL_TIME` seconds, `cCOIL_FRICTION_MOD` is added to the friction for all other cars. This will be discussed in more detail in Section 2.6.1.

- `Armor`: The maximum speed of this car is increased and the car is immune to other items for `cARMOR_TIME` seconds. How the max speed change is implemented is similar to the coil: `cARMOR_FRICTION_MOD` is added to the car's friction. The car being immune to other items means that if it hits an item or gets hit by a shell, the item/shell disappears and has no effect on the car. Similar for tesla coils: if car A is using an armor when car B uses a tesla coil, car A is unaffected by the tesla coil.

2.5.3 Item Terminology

Here are a few terms used to describe items:

Holdable: *Holdable* items are items that represent tangible objects, rather than an ability. Steam shells, oil slicks and pressure mines are *holdable*, the rest aren't.

Hazard Item: A *hazard item* is a used item on the track that has not been hit yet. So when a car uses an oil slick or a pressure mine, it becomes a hazard item. Once a hazard item is hit by a car or a shell, it is deleted from the board, so is no longer a hazard item. Shells are not considered hazard items because they are very different since they move.

Active Item: At any point in time, each car has some set of items acting upon it. For this definition, we are only concerned with items that have a timing aspect (so everything except turbo encabulators), so along with each item is the amount of time remaining that that item affects the car. So for example, if a car uses an armor, then an armor is added to its active items for `cARMOR_TIME` seconds.

2.5.4 Item Boxes

Item boxes appear in rows across the track at the points specified in the `board` type. In each row, there are four item boxes, at w-coordinates $-3/8,\ -1/8,\ 1/8$ and $3/8$. When a car hits an item box, the item box disappears for `cITEM_BOX_RESPAWN_TIME` seconds. As discussed earlier, if the car's buffer is empty, they receive a random item. Note that the item box disappears regardless of whether the car's buffer was full or not.

2.6 Movement

2.6.1 Basic Physics

Cars direct their movement by setting their acceleration in (x, y) coordinates. Hint for the AI: The direction of the track at point t_0 is the vector $(x'(t_0), y'(t_0))$.

`cMAX_ACCEL` is the maximum acceleration allowed (in terms of the norm of (a_x, a_y)). So if a car tries to set its acceleration to (a_x, a_y) where $|(a_x, a_y)| \geq$ `cMAX_ACCEL`, truncate the vector so that it has length `cMAX_ACCEL`. Specifically, set the acceleration to be:

$$\frac{\texttt{cMAX\_ACCEL}}{|(a_x, a_y)|}(a_x, a_y)$$

Note that the max acceleration along with the friction on the track induces a maximum velocity for the car. You don't need to understand the physics behind that... One thing to note if you know something about physics is that "friction" in our game is backwards: in our game, a higher friction means the car can go faster.

Each road type has its own friction value (`cSPEEDUP_FRICTION`, `cNORMAL_FRICTION`, `cSLOWDOWN_FRICTION` and `cWATER_FRICTION`). The items that affect a car's friction (`TeslaCoil` and `Armor`) add their corresponding friction modifications onto the friction from the road type. So if an armored car is on a slowdown road type, then its friction is `cSLOWDOWN_FRICTION+cARMOR_FRICTION_MOD`.

2.6.2 Crashes

A car crashes when it collides with a hazard item or a shell. When this happens, its velocity and acceleration are immediately set to 0, and it is unable to move for `cHIT_TIME_DELAY` seconds. Also, during that time, it cannot be hit by anyone or anything else.

2.6.3 Code Given

We have provided the code to update the positions and velocities of the cars, items and shells. The modules containing this code are the `Physics` module and the `Movement` module. The only function you should need to call directly in the `Physics` module is `trackToCartesian`, which converts a point in (t, w) coordinates to (x, y) coordinates.

The `Movement` module exposes the following types. This list is meant to supplement `movement.mli`, so you should look at that in addition to this list.

Type	Description
`in_car_state`	Contains the car's id, position, velocity, acceleration, active items, and a boolean indicating whether the car is currently holding an item
`in_hz_state`	Contains the item and position of the hazard item
`in_shell_state`	Contains shell's id, position, velocity, activation time left, and the position of the car it is chasing
`in_item_box`	Contains the item box's position
`out_car_state`	Contains the car's id, new position, new velocity, a list of hazard items/shells it hit, a list of items it acquired from item boxes, and a boolean indicating whether the car is still holding an item
`out_hz_state`	Contains a boolean indicating whether the hazard item should be deleted, with `true` meaning that it should be deleted
`out_shell_state`	Contains the shell's id, new position and new velocity
`out_item_box`	Contains a boolean indicating whether the item box was hit

The `Movement` module also exposes the function `update`. This function takes in the board, the time that has passed since the last update, and the current state of the cars, hazard items, shells and item boxes, and outputs the new state for them. You will use this function in `Game.handleTime` to update the data for these objects. Here is how to use it:

1. Convert your state representation to lists of types `in_*`.
2. Call `Movement.update` on those lists along with the board and the amount of time that has passed since the last update.
3. Update your state representation according to the output of `Movement.update`. The lists `update` return will be in the same order as the input lists. In other words, if `boxes` is your list of `in_item_box`'s, and `boxes'` is the list of `out_item_box`'s that update returns, the i^{th} element of `boxes'` indicates whether the i^{th} box of `boxes` was hit during the time step.

Note that `update` treats all hazard items as activated, so you should only pass the list of activated items to `update`. The same is NOT true for shells however (ie, you should send the list of all shells to the `update` function). This is because an unactivated shell moves, so its data still needs to be updated.

3 Communication

3.1 Client-Server Framework

Steam Kart 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., cars) 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.

Cars 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 the id of the car and cause the car to perform various actions
- result messages, which always come with the id of the car 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 car 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 an Id message containing an id for the car. Each action message sent by a unit must include this id to uniquely identify the given unit. The car then waits to receive a `GameStart` message from the server. When the car receives that message, the game has begun and it may start racing.

3.3 Control Messages

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

3.3.1 Control Quick Reference

`GameRequest`	Request to start a game, with the desired character as an argument.
`Id`	Response to `GameRequest`, provides a unique id for that car.
`GameStart`	Informs the car that the game has started.
`GameEnd`	Informs the car that the game has ended and/or that the car has finished the race. Contains an `int` indicating what place the car was in, and a `float` indicating how long it took for the car to finish the race.

3.4 Action Messages

Action messages are sent by the client to the server, and tell the server to 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 car is attempting the action.

3.4.1 Action Quick Reference

`GetBoard`	Returns the board.
`SetAccel`	Sets the car's acceleration.
`Brake`	Stops the car completely.
`UseItem`	Uses an item, if available.
`HoldItem`	Holds an item, if available.
`Scan`	Returns a list of cars, hazard items and item boxes in range.
`MyStatus`	Returns the car's position, velocity, acceleration, held and buffered items, current place, and lap number.
`Talk`	Outputs a string to the chat area.

3.4.2 Action Specification

The following table describes the effects of the possible actions.

Command	Args	Returns	
`GetBoard`	none	Board(b)	where b is the board.
`SetAccel`	(ax, ay)	Success or Failed	Sets the car's acceleration to be the float tuple (ax, ay). Fails if either ax or ay are infinity, negative infinity or "not-a-number".
`Brake`	none	Success	Set's the car's velocity and acceleration to 0.
`UseItem`	none	Success or Failed	If the car is currently holding an item, use that item. Otherwise, use its buffered item. If it does not have a held item or a buffered item, return `Failed`.
`HoldItem`	none	Success or Failed	If the car is currently not holding an item, and it's buffered item is a holdable item, hold that item. Otherwise, return `Failed`.
`Scan`	none	ScanData(cars, boxes, items)	cars is a `(position * held_item * place * lap) list` containing information for all other cars that have not finished the race (so the car that called Scan is NOT in this list). If the car is out of range (further than `cSCAN_HELD_ITEM_RADIUS` away from the calling car in terms of (x, y) distance), then the `held_item` should be `OutOfRange`. Otherwise, it should be `WithinRange` of the item that car is holding. boxes is a `position list` containing the positions of item boxes and pressure mines within range of the car (within `cSCAN_ITEM_BOX_RADIUS` of the car). items is a `(item * float * float * float) list` containing information for all the oil slicks and shells within range (`cSCAN_HAZARD_ITEM_RADIUS`), where the first two floats are (t, w) and the last one is the amount of time until the item is activated.
`MyStatus`	none	MyData(pos, vel, acc, held, buffered, active_items, place, lap)	active_items is a `(item * float) list` containing all items that are currently affecting the car, with the amount of time left for that item. The rest of the arguments should be self-explanatory.
`Talk`	s	Success	Outputs the string s to the chat area

Note that after a car has finished the race and/or the race is over, the response to all actions from that car should be `GameEnd` with the car's place and end time as arguments.

3.5 Result Messages

Result messages are sent by the server to the car 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 response.

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.
`MyData`	Result of a `MyStatus` action.
`Board`	Result of a `GetBoard` action.

4 Boards

Games of *Steam kart* may be played on a variety of tracks. Boards are stored in text files, and the server loads a board file at the start to use for the current game by first using the function `Game.loadBoard` to parse the board file into a board, and then passing that to `Game.initGame`.

How a board file is specified is shown in board.mli. It is not important that you understand this unless you want to make your own boards.

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 client renders the game using OpenGL. This module will 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 GUI, 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`.

The game server is responsible for sending graphical update messages to the GUI, as described below. The game server will listen for clients to send the updates to throughout the game. However, if a GUI connects halfway through the game, it will have missed the message that initialize the board. So be sure the GUI connects before the game starts if you want to be able to see everything. In other words, start the process for the GUI before you start the processes for the cars.

5.2 Sending messages to the GUI

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

5.3 Characters

When a car requests a game, it specifies which character it wants to be. We have included all of the characters from Mario Kart 64 (Mario, Luigi, Bowser, Yoshi, Donkey Kong, Peach, Toad and Wario), plus a Custom character. The custom character allows a bot to pick its own images to display. To do so, the bot sends `Custom(s1,s2)` as its character, where `s1` is the filename of the image to display normally and `s2` is the image to display when the car has crashed. Here are the requirements for the images:

1. Must be bitmaps (`.bmp`).
2. Must be 64×64 pixels in size.
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 $0, 0, 1$). To accomplish this in photoshop, put the image you'd like to use on one layer, then go to Image, Adjustments, Replace Color. Then set the selection color to $0, 0, 0$ and set the replacement color to $0, 0, 1$. Then set the background layer to be black $(0, 0, 0)$. I'm sure other programs have similar options...
4. The images should face to the right, meaning the unrotated image should be what you want to be shown when the car is moving to the right.

The custom images should be placed in the `bmp` directory. Note that the bot only specifies the filenames, not the whole image, so if the files aren't present on the computer the GUI is running on, the GUI will not be able to find the images. If the GUI can't find the images, it will display Mario instead.

5.4 Graphics Commands

	Arguments	Meaning
`InitGraphics`	`board`	Tell the GUI client to initialize the graphics
`Countdown`	`int`	Tells the GUI how many seconds to the game starts
`DisplayString`	`car_id * string`	Display a string to the GUI
`AddCar`	`car_id * position * velocity * character`	Tells the GUI to add the car
`UpdateCar`	`car_id * position * velocity * bool * item * item`	Updates the data for the car with the given id number. The first `item` is the held item, and the second `item` is the buffered item. The `bool` indicates whether the car is currently crashed.
`RemoveCar`	`car_id`	Remove the car with the given id from the board
`AddItemBox`	`position`	Adds an item box at the given position in (t, w) coordinates
`UpdateItemBox`	`position`	Adds an item box in the given position if one doesn't already exist there, otherwise, does nothing.
`RemoveItemBox`	`position`	Removes the item box at the given position in (t, w) coordinates
`AddHazardItem`	`position * item`	Adds a hazard item at the given position
`RemoveHazardItem`	`position`	Removes the hazard item from the given position
`AddShell`	`int * position * velocity`	Adds a shell. The int is an id for the shell. At any point in time, there must be only one shell with a particular id. However, after a shell is removed, you may reuse its id.
`UpdateShell`	`int * position * velocity`	Updates the shell with the given id
`RemoveShell`	`int`	Removes the shell with the given id
`GameOver`	`(car_id * float) list`	Notifies the GUI that the game is over. The `(car_id*float) list` is the list of cars and the time they took to finish the race, in order of 1st place, 2nd place, ...

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 `bot` directories (plus any edits you need to make to the compilation scripts). Here is a list of all the files included in the release and their contents.

`build*.bat`	Build scripts for the game server, teams, and GUI client (see Section 7)
`game/constants.ml`	Definitions of game constants
`game/definitions.ml`	Definitions of game datatypes
`game/expression.mli`	Signature file for expression helper module
`game/expression.ml`	Implementation of expression helper module
`game/board.mli`	Signature file for the board module
`game/board.ml`	Implementation of the board module
`game/physics.mli`	Signature file for the physics module
`game/physics.ml`	Implementation of the physics module
`game/movement.mli`	Signature file for the movement module
`game/movement.ml`	Implementation of the movement module
`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.ml`	General use helper functions
`game/netgraphics.mli`	Signature file for sending updates to the GUI client
`game/netgraphics.ml`	Implementation of sending updates to the GUI
`game/server.ml`	Starts the game server and deals with communication
`shared/connection.mli`	Signature file for connection helper module
`shared/connection.ml`	Implementation of connection helper module

`shared/thread_pool.mli`	Signature file for thread pool helper module
`shared/thread_pool.ml`	Implementation of thread pool helper module
`bot/bot.ml`	Sample bot that demonstrates how to start a game and send actions
`client/graph.ml`	Implementation of graphics module
`boards/*`	Various sample boards (see Section 4)
`bmp/*`	Graphics files for the GUI and code needed 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`, which accepts connects from GUI clients
2. Waits until for enough cars to join the game (see Section 7.2 for more details)
3. Calls `Game.loadBoard`
4. Calls `Game.initGame` with the result of `Game.loadBoard`
5. Calls `Game.initCar` once for each car, and sends `Id` message to each car
6. Starts the countdown, and after 3 seconds, calls `Game.startGame` and sends `GameStart` messages to each car
7. Spawns a thread that regularly calls `Game.handleTime` with the current time
8. Spawns a thread that regularly calls `NetGraphics.sendUpdates`
9. Spawns a thread for each car that listens for action messages
10. Enters loop that accepts new connections from cars (if their old connection breaks)

You will need to think carefully about how you design and implement the game to meet the Server module's expectations. Note in particular that the way `Server` spawns a different thread for handling each car creates concurrency issues for the `Game` module (and that the code we have provided you does nothing to account for concurrency).

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. To help you get started, we have provided an initial definition of the type `game`, and partial implementations of `initGame`, `handleAction`, and a subfunction of `handleAction` to write a string to the GUI.

Your design may require you to add or modify the type declarations or functions we have provided.

6.4 State

We have also provided a partially specified but completely unimplemented `State` module. We have not specified the `state` type. That is one of the major things you need to do. You do not need to keep any of the functions we specify; they are just suggestions.

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

7 Running the game

7.1 Note for Mac/Linux Users

We have provided batch files to compile the game, GUI and cars. To make them work on a non-Windows system, you just need to remove the ".exe" in the scripts. And to run them, run "`sh build_script.bat`" instead of just "`build_script`".

7.2 Game Server

Run `build_game` to build the game server. The game server has five command line arguments:

1. The file name of the board file to use
2. The minimum number of cars desired for the race (1)
3. The maximum number of cars desired for the race (2)
4. The port number to listen on for the cars (10503)
5. The port number to listen on for the GUIs (10501)

The first argument is required, the other four are optional. The default values for the optional ones are in parentheses above. You should not change the port numbers unless you have another program on your computer already using that port (which you shouldn't).

The server waits until the minimum number of cars joins the game. Then if the maximum number of cars is strictly greater than the minimum number of cars, the server waits for up to `cMAX_WAIT_CONNECTION_TIME` seconds for the maximum number of cars to join, then it starts the game.

7.3 GUI

The first time you run the GUI, you will have to run `build_bmp.bat`. This compiles all the files in the `bmp` directory. You will only have to do that once. To build the GUI client, run `build_gui.bat`. The GUI will have to be rebuild if the constants or type definitions change (most likely, we will have to change the constants to balance the game). The GUI has the following command line arguments:

1. The IP address of the game server
2. The port number that the game server is listening on for GUI connections (10501)

The second argument is optional, with 10501 as the default value.

7.4 Car

Run `build_bot` to build the bot in the file `bot/bot.ml`. The provided bot has the following command line arguments:

1. The IP address of the game server
2. The port number that the game server is listening on for car connections (10503)
3. A boolean indicating whether the bot should move back towards the center of the track when it gets to far from the center (true)

The second argument is optional, with 10503 as the default value. The third argument changes the functionality of the bot. When you write your bots, you may consider doing this kind of thing to test if new strategies are better or not.

7.5 The Whole Game

In order to run the whole game on a local machine, follow the instructions in the above three sections, with 127.0.0.1 as the ip address, running multiple cars if desired. 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 Design meeting

Your first task is to create a design for your *Steam Kart* implementation and meet with the course staff to review it. Each group will use CMS to sign up for a meeting, which will take place between November 17th and November 20th. If you are unable to sign up for any of the available time slots on CMS, contact the course staff, and we will try to accommodate you.

At the meeting, you will be expected to explain the design of your system, explain what data structures you will use to implement the design, and hand in a printed copy of the signatures for each of the modules in your design. You are also expected to explain your initial thoughts about strategies for your car. 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 *Steam Kart* 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 `bot` 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 car program we provide to test your game, but for full testing coverage you will need to write your own tests.

8.3 Designing a bot

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

There are many different strategies for building a good bot. This is your chance to be creative and have fun creating a good AI.

We will provide an online 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 bot works correctly, and try your bot against other people's bots. We will also provide some staff bots to test against, at our discretion. Further information on the server will be provided soon.

In the next couple weeks, we will release a guide to what your bot will need to be able to do, in the form "you need to be able to complete this track in this amount of time, and be able to consistently beat these staff bots on the KOTH server". Stay posted here and on the newsgroup for more details as the due date approaches.

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.

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

8.5 Things to keep in mind

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

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

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

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

- **Think carefully about how to break up your program into loosely coupled modules.** The program will be complex and difficult to debug unless you can develop modules that encapsulate important aspects of the game. Design the interfaces to these modules carefully so that you can work effectively with your partner and 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 bots.** If you are using your own bots to test the actions and something seems wrong, the bots could just as easily be at fault.

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

- **Remember to finish the game in time to write a good bot.** The bot part of the project is worth as much as the game part, so don't put off the bot part until the end. And unlike in past semesters, we WILL be grading it based on how well it performs. It will be VERY hard to get 35/35 on this part.

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 `build_game.bat` script to compile your game code, and the `build_bot.bat` script to compile your bot code (i.e., you should modify the scripts to include all necessary files). This should include:

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

9 Tournament

On December 13, after the problem set is due, there will be a *Steam Kart* tournament which you are encouraged to submit your bot programs to. There will be lots of free food, and the chance to watch your bot perform live. The winner gets bragging rights and will have their name posted on the 312/3110 Tournament hall of fame.

10 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 `.pdf` 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 $\Delta\Phi$.)