Lab 4: Space ... The Final Frontier

Congratulations! You are the first ECE244 student to serve onboard the Starship *Galaxy Explorer*. The ship just had a fierce battle with the Romulans near the Neutral Zone. Although the ship emerged victorious, it sustained much damage. The warp engines are offline, the shields are down and all photon torpedoes are depleted. The Captain is taking the ship at impulse speed to the nearest star base for repairs. However, the ship must first navigate through the notorious Huya-538[1] asteroid belt. To avoid the destruction of the ship, phaser banks must be used, while the ship is going through the belt, to destroy the asteroids. As the leading programming expert on the ship, you are called upon to write a C++ program that receives sensor data for the asteroids and directs phaser fire to destroy them before they hit the ship. Your journey begins ... live long and prosper!

1 Objectives

The main objective of the assignment is for you to learn how to build, use and maintain linked lists. You will do so by extending a game server that simulates the asteroids and the actions of the Starship as it navigates through the belt of asteroids.

2 Game Overview

The game consists of two major components: the *game server* (or simply the *server*) and the *player code* (or simply the *player*), as depicted in Figure 1. The server contains the *asteroids observer*, which randomly generates asteroids with different masses, positions and velocities. This observer reports to the player changes in the status of the asteroids by making calls to a set of *asteroids observer functions*. The player maintains a linked list of the asteroids currently active in the game, based on the asteroids observer function calls made. The player implements a set of linked list methods to manipulate this linked list.

The player has a *Game AI* module, which implements the AI of the game: code that examines the list of asteroids, decides which one to fire at and then uses a set of *ship functions* to instruct the *ship controller* module in the server to rotate the ship and/or fire. The game server updates the status of the asteroids based on the firing and calls back the asteroids observer functions, so the player can also update its linked list of asteroids. The *display manager* in the server graphically displays the asteroids, the ship and the firing actions. The game remains active until all the asteroids are either destroyed or they exit the game (a game win), or until an asteroid hits (and thus destroys) the ship (a game loss). A *score* for the game is determined based on the number and mass of asteroids destroyed.

The code for the server is provided to you as a set of object files. It is the code for the player that you must write, including the code that implements the linked list of asteroids, the code for the asteroids observer functions, which calls the linked list functions, and the game AI.

[1] Huya is the name of the rain god of the Wayuu people of Venezuela and Colombia. The belt is so named because navigating through it is like flying through a heavy "rain" storm of asteroids.

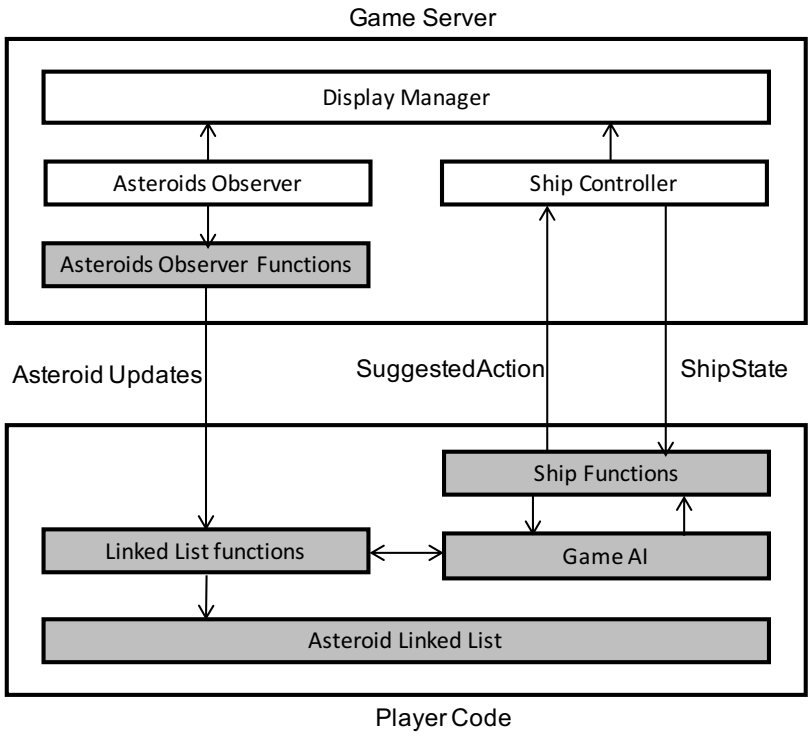

Figure 1: The game server and the player code. The grayed boxes represent code you have to write.

3 Problem Statement

You will write C++ code that: (1) implements methods that create, manipulate and maintain a linked list of asteroids, (2) responds to changes in the status of asteroids (as signaled by the asteroids observer) by making calls to your linked list methods, and (3) implements the game AI. The remainder of this section details the various classes used in the game, pointing out what is implemented for you and what is it that you have to implement.

3.1 The `Asteroid` Class

This class represents `Asteroid` objects and it is defined in the file `Asteroid.hpp`. You should examine the header file to become familiar with the variables used to represent an asteroid and the methods of the class (see Section 4 for the location of this and other class files).

In summary, each asteroid object has an `ID`, a `mass`, a `velocity` and a `health`. The `ID` is an identifier unique for each asteroid. The `mass` is a non-zero integer that reflects the relative mass of the asteroid. The `velocity` is a 2-dimensional vector that has the speed of the asteroid in the `x` and `y` dimensions of the game grid (see Section 3.6 for a description of the game grid). The `health` is an integer that indicates how close the asteroid is to being destroyed. On every phaser hit, `health` is decremented and when `health` becomes `0` the asteroid explodes.

In addition, each asteroid has associated with it a *hit box*. It represents the rectangular area around the asteroid within which a phaser hit is considered a hit of the asteroid. The mass of an asteroid determines how many phaser hits are needed to destroy it. For a small asteroid, a single phaser hit is sufficient to destroy it. For a large asteroid, two hits are needed. A large asteroid hit once may split into multiple smaller ones (but also may not).

The implementation of the `Asteroid` class is within the server code and you need not implement it. Thus, you are **not allowed** to modify the `Asteroid.hpp` file.

3.2 The `AsteroidListItem` Class

This class is used to represent an entry of the linked list of asteroids. The definition **and** implementation of this class appear in the file `AsteroidList.hpp`. An object of this class simply contains a pointer to an `Asteroid` object and a pointer to an `AsteroidListItem` object, the next item on the list. The class has methods to allow you access to its `Asteroid` object and pointer data. You are **not allowed** to modify the contents of the `AsteroidList.hpp` file.

3.3 The `AsteroidList` Class

This class defines a linked list of `AsteroidListItem` objects. Its definition appears in the file `AsteroidList.hpp`. It contains methods to add and delete `AsteroidListItem` objects to and from the list. It also has methods to traverse the list. An example of an `AsteroidList` is shown in Figure 2. The `AsteroidList` class simply contains an `AsteroidListItem` that represents the `head` of the linked list. The `Asteroid` data associated with this head is undefined and should not be used. The `next` pointer points to the first `AsteroidListItem` on the list. You must implement the methods of this class in the file `AsteroidList.cpp`. You are **not allowed** to modify the contents of the `AsteroidList.hpp` file.

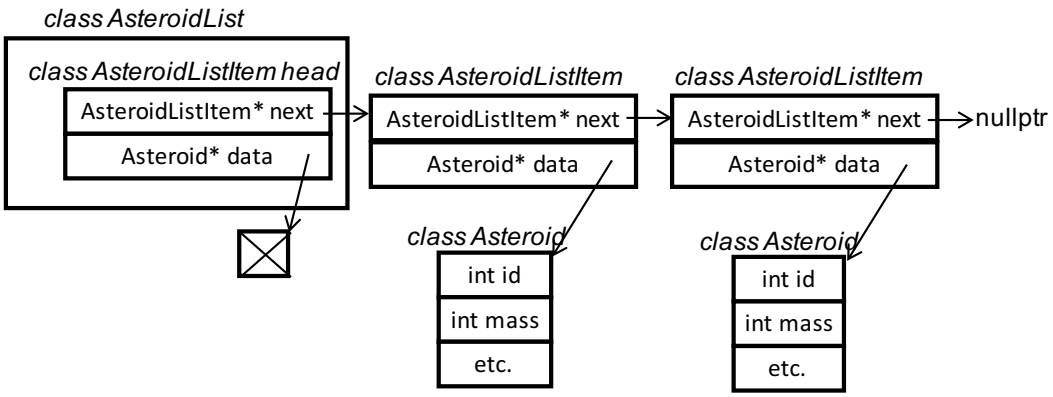

Figure 2: An example of an `AsteroidList` with two items. Note that the first `AsteroidListItem` is the head of the linked list.

A detailed description of the methods of this class appears in the comments of the `AsteroidList.hpp` file. Read these comments to find out what each method should do and implement it correctly.

3.4 The `AsteroidsObserver` Class

This class defines the interface by which the game server provide updates about asteroids to the player. It contains methods that are called by the game server to inform the player when an asteroid appears in the game, has moved, has left the game or has been destroyed. The class definition appears in the file `AsteroidsObserver.hpp` and you must implement its methods in the file `AsteroidsObserver.cpp`. You implementation of these methods must manipulate the linked list of asteroids (pointed to by `asteroid_list` declared in the `AsteroidsObserver` class) to reflect what happened to the asteroids. However, you are **not allowed** to modify the contents of the `AsteroidsObserver.hpp` file.

3.5 The `GameAI` Class

This class defines methods that implement the logic or AI of the game. This class is defined in the file `GameAI.hpp`. The main method in this class is called `suggestAction`. It is where the AI is implemented. This method should examine the list of asteroids and the ship state (contained in a structure called `ShipState`, see below) and then decide what the ship should do. The method informs the game server about the desired action through a structure called `suggestedAction`, see below.

You must implement the `suggestAction` method of the `GameAI` class. You should consider factors such as the proximity of asteroids to the ship, the time until the phaser bank charges, the mass of an asteroid as well as its remaining health. The more and bigger asteroids are destroyed, the higher the score of the game.

3.5.1 The `ShipState` Structure

This structure carries information from the server regarding the state of the ship. This information includes: whether a phaser fire has already been requested, how many more cycles before the phaser banks can fire, the current location and rotation of the ship, and the current game score. The server automatically updates this structure on each cycle (see Section 3.6 for the definition of a cycle). This structure is declared in the file `GameAI.hpp` and you may not modify it.

3.5.2 The `SuggestedAction` Structure

This structure, defined in the file `GameAI.hpp`, simply contains two variables that indicate the yawing of the ship, i.e., its tilt and the firing action. The structure utilizes two `enum`s to list the possible values for each of these variables. It also pre-defines some simple common actions that can be taken (those pre-defined actions are defined outside the structure). You must not modify the definition of this structure.

3.5.3 The `myAIData` Structure

This structure is defined in the file `myAIData.hpp` and it contains sample fields that can be used to store relevant information about your game AI. You will likely need to modify this structure to add more fields to it.

3.5.4 The `debug_rt` Object

The `debug_rt` data member of `GameAI` is a special SFML object that you can draw to, much like you have done in lab assignment 2. It is of tyoe `RenderTarget`, which is similar to `RenderWindow`. Every time the game server updates the screen, the contents of `debug_rt` is drawn over the game. In other words, anything you draw to `debug_rt` will be drawn over the game. Use `debug_rt` to help you debug your AI by visualizing it's state. However, you need not use it if you do not want to.

3.6 Game Operation

The game server operates in *cycles*. In every cycle, the asteroids observer generates new asteroids into the field of the game. It then makes a sequence of calls to the asteroids observer methods to reflect changes to the state of each asteroid (one call per asteroid), both existing and new. The changes in state include: the motion of an existing asteroid, the creation of a new asteroid or the destruction of an asteroid. The server then invokes the `suggestAction` method of the player's

`GameAI` class to get the suggested action of the player in the structure `suggestedAction`. The server updates the game based on the suggested action and updates the screen, thus completing a cycle. It immediately begins the next cycle by repeating the process above. The game server keeps track of your *score* in the game.

Asteroids and the ship are located on a 2-dimensional grid that is used to define the position of an asteroid or the ship. The origin of the grid is the top left corner of the window. The `x` axis is the horizontal, increasing to the right. The `y` axis is the vertical, increasing towards the bottom.

3.7 Sample Game

The executable of a sample game is provided in the release called `galaxy-explorer-ref`) so you can see how the game is played. Run the executable to start the game window. The sample game contains a basic AI module to show you the action of the game.

The asteroids in the game are randomly generated based on random seeds. Thus, every time the game is run, a different set of asteroids appear. While this desirable in a game, it is not very conducive to debugging and to improving your AI. Thus, both the sample game and the game you build can be run with a specific random seed. For example, the sample game can be run with a random seed of `137137411` as follows:

```
galaxy-explorer-ref 137137411
```

You can also play the game without the AI using the keyboard. When the initial game screen starts and before you hit the space key to start, you can press the `a` key. This key toggles the game to use or not use the AI. If the AI is not used you can use the keyboard to control the ship and to fire. Use the left and right arrow keys to rotate and the space bar to fire.

4 Procedure

Make your `ece244` directory (in your home directory) your current working directory (using the `cd` command. Download the `lab4_release.zip` file, un-`zip` it in the `ece244`. This creates a sub-directory called `galaxy-explorer` that contains all the assignment files.

- You will write your code in the following files: `AsteroidList.cpp`, `AsteroidsObserver.cpp`, `GamaAI.cpp` and `MyAIData.hpp`, as described earlier in Section 3. These files are contained in the directory `~/ece244/galaxy-explorer/src/galaxy-explorer`.
- There is a sample game executable in `~/ece244/galaxy-explorer/`. It is called `galaxy-explorer-ref`. You can run this executable to see how the game plays with a simple AI.
- The header (`.hpp`) files described earlier in this document (e.g., `Asteroid.hpp`) are located in the directory `/share/copy/ece244f/lab4/src/galaxy-explorer`. They are only readable to you, thus ensuring that you cannot modify them. In order to view these file, you may do one of two things. At the command prompt, you may use your favorite editor to view the files. For example, you can use `vi` to view `Asteroid.hpp` using the command:
  `vi /share/copy/ece244f/lab4/src/galaxy-explorer/Asteroid.hpp`

  Alternatively, if you are using `NetBeans`, *build* the project and then while holding the CTRL key, click the `#include "Asteroid.hpp"` line.

- The `galaxy-explorer` directory contains a pre-configured project for `NetBeans` as well as a `Makefile` should you want to directly use the command line. To use the supplied `NetBeans` project, start `NetBeans`, open a project through the menus: `File -> Open Project` and select `~/ece244/galaxy-explorer`. You can then compile and run the code in `NetBeans`. You can also locate the include files easily within `NetBeans` (see above).
- To use the command line to compile, make `~/ece244/galaxy-explorer` your current working directory and type `make`. This will utilize the provided `Makefile` and place the executable in `~/ece244/galaxy-explorer/build/EXE/galaxy-explorer`. Thus, you can run this executable by making the directory `~/ece244/galaxy-explorer` your current working directory and typing: `./build/EXE/galaxy-explorer`.
- If you wish to work on the assignment remotely from you home machine, use VNC to connect to `remote.ecf.utoronto.ca`. This is the most convenient way to work remotely, allowing you to use SFML and the provided `Makefile` on ECF while working at home. Refer to the handout titled: "Remote Access to ECF with VNC" for details on how to connect with a VNC client.

When you build your code, either using the command line or using `NetBeans`, a special executable is generated. It is called `test_galaxy-explorer` and it is located in the directory `~/ece244/galaxy-explorer/build/EXE` along with the game executable. You can run this executable by making the directory `~/ece244/galaxy-explorer` your current working directory and typing: `./build/EXE/test_galaxy-explorer -s`. You can also run the test executable from `Netbeans` by changing to the `Debug_Test` configuration.

The `test_galaxy-explorer` executable runs your code against a set of test cases *for the linked list class* (but not for the AI). These tests exercise insertion and deletion into your linked list. They report back what each test does and if it failed. The following are examples of the output for two tests, one that fails and one that passes.

The output of the test that fails is shown below. The test runs the scenario `Baic Insertion`. It is given an empty list and adds two list elements to it using `pushFront()`. Thus, is expects the first element on the list to be the last element inserted. The test also requires the size of the list to be 2. However in this case, the list remains empty and thus the test fails as shown.

```
-------------------------------------------------------------------------------
Scenario: Basic Insertion
     Given: an empty list
      When: pushFront()ing an element
  And when: pushFront()ing another element
      Then: the front element should match the newly pushed-in value
-------------------------------------------------------------------------------
... galaxy-explorer/testing/asteroid_list_test_public.cpp:39
.......................................................................
... galaxy-explorer/testing/asteroid_list_test_public.cpp:40: FAILED:
  REQUIRE( alist.size() == 2 )
with expansion:
  0 == 2
```

The output of the test that passes is shown below. The scenario is `Basic Invariants`. Given an empty list, then the list must be empty. The condition tested for is `alist.begin() == alist.end()`, which is true and thus the test passes.

```
-------------------------------------------------------------------------------
Scenario: Basic Invariants
     Given: a empty list
      Then: the list should be empty
-------------------------------------------------------------------------------
... galaxy-explorer/testing/asteroid_list_test_public.cpp:17
...............................................................................
... galaxy-explorer/testing/asteroid_list_test_public.cpp:18:
PASSED:
  REQUIRE( alist.begin() == alist.end() )
with expansion:
  nullptr == nullptr
... galaxy-explorer/testing/asteroid_list_test_public.cpp:19:
PASSED:
  REQUIRE( alist.size() == 0 )
with expansion:
  0 == 0
```

Since you are able to test your code directly with the `test_galaxy-explorer`, there is no `exercise` for this assignment.

5 Deliverables and Marking

The assignment is marked in **three** parts. The first part auto-tests your code for the correctness of your linked list implementation. This part is worth 60% of the total mark of the assignment. The second part checks for the style (structure, descriptive variable names, useful comments, consistent indentation, use of functions to avoid repeated code and general readability) of your code. This part is worth 20% of the total assignment mark. The last part is based on your score in the game, is worth 20% of the total mark of the assignment, and is assigned on a *competitive* basis.

The scores of all students in class are sorted from highest to lowest. The highest score ($score_{highest}$) receives the full 20%. The lowest score ($score_{lowest}$) receives 0%. A score in between ($score$) receives a $mark$ based on the following formula:

$$mark = 20\% \times ((score - score_{lowest})/(score_{highest} - score_{lowest}))$$

Your score is displayed on the game window. For the purpose of calculating your mark for the 20% component, your score will be capped n two ways. First, the maximum time allowed to play the game is 10 minutes. Second, your score is capped at 50,000.

In order to find out how your score (limited by the 10 minutes of play but un-capped by the 50,000) stands in relation to the rest of the class, you must first submit your assignment (remember that you can submit your assignment multiple times and that every new submission overwrites the previous one). Every night, all submitted assignments will be collected, compiled and executed. The resulting scores will be displayed on the following web page:

http://www.ecf.utoronto.ca/~ece244i/lab4rankings.html.

You can find your score by searching for your student number. Please note that the scores rankings **will not be available on the web site until the second week** of the assignment.

Submit the files in your `~/ece244/galaxy-explorer/src/galaxy-explorer` directory. First, change your working directory to that directory. Second, execute the `submit` command) as lab 4 using the typing:

`~ece244i/public/submit 4`.