Super Stolee Bros

This was originally assigned as a project for COMS 229 in Spring 2014.

1 Project Summary

Super Stolee Bros is a completely original video game[1]. The goal is for the heroe(s) to eat all of the "eatable" items in the map, but to not be eaten by any of the enemies! Right now, the game is built for the user to control the heroe(s) by the keyboard, but your job is to create AI[2] to replace the heroes, enemies, and even the powerups!

Each instance of the game takes place on a 2D map that is placed on a grid. Certain locations are marked as walls, and you cannot walk through walls. Otherwise, you can typically walk left, right, up, down, or even stay put, but in certain places there are "teleporters" that allow a jump to another place in the map. There are also "treadmills" that force movement in one direction. The map is stored in a data structure called a `GraphMap` that stores lists of the possible moves from each location. Each map is loaded to the game from a text file.

Here are some basic rules:

- When a *hero* overlaps an *eatable* item, that eatable item is removed.
- When a *hero* overlaps with an *enemy* (that is not *eatable*), then the hero is removed.
- The *heroes* win if all eatable items are removed.
- The *enemies* win if all heroes are removed, or if time runs out.

Your goal is to create actors that perform certain actions:

- `simplehero` : This hero will eat all eatables, given sufficient time, no enemies, and no "fast" powerups. (This is the only strategy required to be done by Part A.)
- `smarthero` : This hero will eat all eatables, given sufficient time, "slow" enemies, and no "fast" powerups.
- `smartenemy` : This enemy will have a coordinated strategy that will guarantee the enemies win (by eating the hero, or by making time run out).
- `smartpowerup` : This powerup will try to make the enemies win by avoiding the hero and "hiding" near the enemies.

[1] It should really be called PAC++MAN, but I don't want to get sued.
[2] Not really "Artificial Intelligence," but Automated Interaction!

Note: This project involves working with compiled binaries created by the instructor. You will be given C++ header files and compiled object files and this will allow you to interact with these objects without modifying the code yourself. As this code may contain bugs that you will discover during your project, the binaries and header files may be subject to modification *at any time*. Typically, this will be a small bug fix, but it may also be due to adding features, changing an interface, and/or changing the rules (slightly).

Whenever an update is made to the binaries, we will send out an email warning you of this fact. If there is a change to the interfaces, then the sample `main` method will be updated to account for this. This is unfortunately something that is unavoidable as we make concrete plans for how to grade and test your AIs.

2 Project Goals

There are several learning objectives for this project. We want you to learn how to solve problems by breaking them down into a sequence of small decisions in order to create a full algorithm. We want you to predict possible failures in your algorithms. You should flex some creative muscles in order to come up with solutions. You will learn to deal with third-party libraries and APIs by looking through documentation and troubleshooting. In addition, of course, we want to make sure you learn to use C++.

In addition, you will demonstrate proficiency in the following C++ programming language features: classes, inheritance, polymorphism, pass-by-reference, source code organization, compilation, makefiles.

You may also find the following concepts helpful: the Standard Template Library, templates, debuggers (such as `gdb`), memory error detectors (such as `valgrind`),

3 Detailed Project Specification

Generally, there are three main classes to this game.

- `GameManager` — Controls all action during the game, and reports if there are any special events in the game.
- `GraphMap` — Stores the current state of the game, including all positions of the actors.
- `Actor` — Extensions of `Actor` contain the strategies of the different players of the game, including heroes, enemies, powerups, and eatables.

Your main task is to extend the `Actor` class to create the four strategies `simplehero`, `smarthero`, `smartenemy`, and `smartpowerup`. For **Part A**, only `simplehero` must be implemented, but all four must be implemented in the version submitted for **Part B**. These should all be implemented as distinct extensions of the `Actor` class and then passed to the `GameManager`. Command-line arguments will specify which actors are to be used in each run of the game.

- Implement the `getNetId` method, which will return a string containing your NetID.
- Implement the `getActorID` method, which will return a string containing the id of the actor (one of `simplehero`, `smarthero`, `smartenemy`, or `smartpowerup`).
- Implement the `duplicate` method, which will create a new instance of your class. (This will also allow you to connect your instances in order to collectively form a strategy.)
- Implement the `selectNeighbor` method, which is how the actor selects the next move, given the current state of the game map.

These methods are all used for different reasons. You will pass all of your actors to the `GameManager` in an array, and it will use `duplicate` to create extra instances whenever the map says to place an actor of that type. The command-line arguments will be used to select which actor ID is tied to each actor type. (The `GameManager` takes the arguments as part of its constructor, so you do not need to parse the arguments.) During the method `GameManager::play()`, it will manage the map and ask each actor to select a move using `selectNeighbor`.

Important: The map is stored in a data structure called `GraphMap`. You can interact with this object during `selectNeighbor`. Each (x, y) position is given a *vertex id*, which is a number from 0 to `numVertices()`-1. You will discover which vertices have edges to other vertices by using pass-by-reference parameters with the (x, y) coordinates. The *structure* of the map will not change (the possible moves at each position) but the positions of the actors may change. See `GraphMap.hpp` for more on this interface.

The `GraphMap` will also store a list of the actors in the game, but you will only discover their type and position. Use that information to pursue eatables and avoid enemies (or pursue heroes or enemies, depending on the strategy!)

3.1 Command-Line Arguments

The executable should always be called `ssbros`. The program is run as

```
./ssbros [map] [arguments]
```

Thus, the first command is to a text file containing the map for the level. There are several examples in the supplied tarball in the "maps" directory. Other arguments are as follows:

- `--moves #` – How many turns to allow before quitting.
- `--hero [id]` – Use the actor with the given id for the heroes.
- `--enemy [id]` – Use the actor with the given id for the enemies.
- `--eatable [id]` – Use the actor with the given id for the eatables.
- `--powerup [id]` – Use the actor with the given id for the powerups.

- `--delay-[type] #` – Let actors of the given type only make moves every # turns.
- `--delay #` – Make every move last # miliseconds.
- `--render-off` – Do not render the map, or make timed delays. Useful if you want to use `printf` for debugging.

3.2 Hints on how to get started

On Monday March 24th, we will have a lecture about graphs, directed graphs, and path-finding algorithms. DON'T MISS IT! In this lecture, we will learn an algorithm, called *breadth-first search* which can be used to find paths in a directed graph. Using this, you should create a method that will determine the distance between any two vertices, and use that as a subroutine for your more advanced algorithm. There are some excellent Wikipedia articles at the end of this document, but also you can find many many resources about these algorithms!

3.3 README

As part of your source-code deliverable, you will write a text file called `README` that contains a description of your project design. The `README` file should contain the following elements:

1. Your Name and NetID
2. A high-level description of your strategy for each actor.
3. A description of any extra details assumed due to ambiguous requirements in the project specification.
4. A file listing of the source code files, specifically mentioning the purpose of each code file.

3.4 Compilation and Makefile Requirements

All software must compile and run on `pyrite.cs.iastate.edu` without error. Simply typing `make` should build all of your executables. Also, `make clean` should remove the executables and any object files. You may include other targets for your convenience as you choose (e.g., "`make tarball`")

Your source code should be delivered as a gzipped tarball with the file name `netid.tar.gzip` (where `netid` is replaced by your NetID). Inside the tarball should be *source code only*, no compiled binaries. The grader will type the following commands to compile the software:

```
tar xvf netid.tar.gzip
make clean
make
cat README
```

Thus, your tarball should contain the `Makefile` in the root directory, and compile all binaries to that same directory. In addition, your makefile should use the `-Wall` and `-g` flags when compiling via `g++`. We will use `valgrind` to make sure that no memory errors occur, and that all allocated memory is deallocated by the termination of the program. (If there is a memory leak or bug in the compiled binaries provided, then let us know and we will fix them!)

If all of your source code is given in `.cpp` and `.hpp` files in one directory, you can compress your tarball using the command

```
tar czf netid.tar.gz *.cpp *.hpp Makefile README
```

SPECIAL FOR PROJECT 2: You are *not* to change the header files given to you in the project specification. In fact, we will *overwrite* these files and hence you may actually not compile if you change your header files.

3.5 Submitting your work

You should turn in a gzipped tarball containing your source code, makefile, and a README file that documents your work. The tarball should be uploaded in Blackboard.

Your executables will be tested on `pyrite.cs.iastate.edu`. You should test early, and test often on pyrite. We will use some scripts to check your code; this means you should not change the name of the executables or the order of the command-line arguments.

Since all input in this project is given in the form of command-line arguments and by file input/output, your binaries should not output anything to the shell unless there is an error condition. Likewise, they should not take any input from standard in.

4 More Resources

http://en.wikipedia.org/wiki/Directed_graph

http://en.wikipedia.org/wiki/Breadth-first_search

http://en.wikipedia.org/wiki/Strongly_connected_component

http://en.wikipedia.org/wiki/Actor_model

http://en.wikipedia.org/wiki/Automated_planning_and_scheduling