CS246E—Assignment 5, Project (Fall 2023)

B. Lushman

Due Dates: See project guidelines

DO NOT EVER SUBMIT TO MARMOSET WITHOUT COMPILING AND TESTING FIRST. If your final submission doesn't compile, or otherwise doesn't work, you will have nothing to show during your demo. Resist the temptation to make last-minute changes. They probably aren't worth it.

This project is intended to be doable in the alloted time. Because the breadth of students' ablilities in this course is quite wide, exactly what constitutes the amount of work that a student can do is difficult to nail down. Some will finish quickly; others won't finish at all. We will attempt to grade this assignment in a way that addresses both ends of the spectrum. You should be able to pass the assignment with only a modest portion of your program working. Then, if you want a higher mark, it will take more than a proportionally higher effort to achieve, the higher you go. A perfect score will require a complete implementation. If you finish the entire program early, you can add extra features for a few extra marks.

Above all, MAKE SURE YOUR SUBMITTED PROGRAM RUNS. The markers do not have time to examine source code and give partial correctness marks for non-working programs. So, no matter what, even if your program doesn't work properly, make sure it at least does something.

Note: This assignment requires you to manipulate the terminal screen in a non-sequential way. Before you start programming, you will need to learn to use the Ncurses library (or something with equivalent functionality). You may also want to spend some time thinking about how you might make this library (which was written for C) more C++-friendly (e.g., build some abstractions around the core functionality).

The AGE engine

In this project, you will produce a game engine to support the creation of ASCII art video games. In addition, you will produce some simple games to demonstrate the utility of your engine.

An AGE game will consist of a game screen, 80 columns by 25 rows. Three rows will be reserved at the bottom for status info, and the game screen will be surrounded by a one-character frame, meaning the playable area will be 78 columns by 20 rows. An example follows:

```
+------------------------------------------------------------------------------+
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
```

Within the playing area, you may place ASCII elements of various shapes, and with various movement and interaction properties. These elements, although displayed in a 2-dimensional plane, will have a "height" attribute indicating how high above the plane of the terminal screen they conceptually lie. This allows for some items in the game to pass by each other without interacting, and others to collide.

The architecture of the game will be clocked. A clock will "tick" each 0.05 seconds, and during each tick, each element will take one step of its programmed actions, interacting with other objects as appropriate.

The border of the play area can be one of two types:

- solid: if an object comes into contact with the border, it counts as a collision
- view: if an object comes into contact with the border, it continues on past it, according to the object's movement specification

An entity that has gone off screen through the view border may despawn if it has been completely invisible for longer then ten ticks (you can make this configurable on a per-object basis if you have a need for an object to persist, but in general there must be a policy under which objects not on the screen are cleaned up without having been destroyed by game play). A player-controlled entity cannot leave the screen.

The objects in the game can be of several shapes:

- a single character (assumed to be printable ASCII)
- a rectangle, made up of a single character, repeated
- a bitmap, specified by a vector of (x, y, char) triples

The movement characteristics of objects in the game can be any of several:

- moving in a straight-line path

- stationary, cycling through a periodic sequence of k forms (this could be used to implement blinking, rotating, growing, shrinking, morphing, etc.)
- gravitating towards one of the borders of the screen (basically, in addition to whatever other motion the object is performing, it will also, if not blocked, take one step towards the edge that it's gravitating to, for each tick)
- player-controlled (during each tick, the game will respond to one keystroke from the user)
- combinations of the above may be possible

Additionally, any of the above may have the ability to spawn other objects under circumstances the game programmer specifies (this could model shooting projectiles, or entities splitting, etc.)

When objects collide, any of a variety of things might happen: If the objects are of different heights, they pass through each other, with the higher object showing. This would be one way to implement "background art" or "fog of war". For objects at the same height, a variety of interactions are possible:

- They may pass through each other.
- They may collide and bounce off one another.
- They may collide and stop right there (e.g., landing on the ground).
- One item (or both items) may be damaged or destroyed.
- It may trigger a win (or a loss).

A game programmer will need to specify some class to encapsulate the game state. The actions of the entities in your game must have the ability to update the game state under the correct conditions, but to the extent possible, must be decoupled from the actual game state itself, since the entities are provided by the engine, and the game logic is provided by the client.

Your task is to design a set of classes that provide these objects and interactions (you may add more if you want), and then implement two games in your engine. Your games should each consist of a client class that places objects of various kinds on the board and then calls a `go` method on your game board that sets the simulation in motion.

Architecture

Your implementation of this project should employ the MVC design architecture (loosely defined). Information on MVC will be given in a tutorial. As part of this requirement, you must separate the classes that manipulate the core editor data structures from those that communicate with the user.

The basic structure of this engine should be fairly simple; the challenge will come from answering all of these little design question, and to come up with an abstraction that is truly general, such that a client could actually use it to program a variety of games.

For this reason, we allow and encourage the class as a whole to participate in discussions, on Piazza, about how you might design around some of these ideas. Course staff might participate as well. You may discuss design tradeoffs in this public forum to great detail, but you must not show any of your own code (only your partner may see your code). There are many ways to imagine a project like this coming together, and this will be a great chance for you to debate something

open-ended and real. (This is an experiment, and future classes may or may not get the same chance, depending on how it goes.)

Evaluation

The correctness and completeness portion of your project will be weighted 60% for the engine, and 20% for each of the two sample games. A very small portion of the grade for the games will reward creativity, but this is largely to ensure your games aren't boring, rather than something to worry about. The majority of the creativity score will go to ensuring that two games are actually different from each other. We will publish a list of game suggestions, any two of which will be considered sufficiently different for the purpose of the project. But you are also welcome to execute your own ideas (just be careful you don't take on more than you can handle).

The two games should exercise as many of the required elements of the engine as you can manage. You are not necessarily required to execute all of them within your two games, but make a decent attempt. You are also permitted to have a third "junk" game, whose purpose is just to show those elements that you were not able to incorporate into the other two.

Grading

Your project will be graded as follows:

Correctness and Completeness	60%	Does it work? Does it implement all of the requirements?
Documentation	20%	Plan of attack; final design document.
Design	20%	UML; good use of separate compilation, good object-oriented practice; is it well-stuctured, or is it one giant function?

Even if your program doesn't work at all, you can still earn a lot of marks through good documentation and design, (in the latter case, there needs to be enough code present to make a reasonable assessment).

If Things Go Wrong

If you run into trouble and find yourself unable to complete the entire assignment, please do your best to submit something that works, even if it doesn't solve the entire assignment. For example:

- can't handle all motion types
- can't handle motion types in combination
- can't handle all interaction types

You will get a higher mark for fully implementing some of the requirements than for a program that attempts all of the requirements, but doesn't run.

A well-documented, well-designed program that has all of the deficiencies listed above, but still runs, can still potentially earn a passing grade.

Plan for the Worst

Even the best programmers have bad days, and the simplest pointer error can take hours to track down. So be sure to have a plan of attack in place that maximizes your chances of always at least having a working program to submit. Prioritize your goals to maximize what you can demonstrate at any given time. We suggest: start with getting your program to display the game board correctly. Then add support for simple, non-moving objects. Then add moving objects. Then continue to add elements, one-by-one. Player-controlled objects are perhaps the trickiest, so may be best left until you have many of your other elements working. At the same time, you should be aware of this and other more complex objects when putting your design together, so as to avoid needing a full rewrite to accommodate them. Take the time to work on a set of tests at the same time as you are writing your project. The main tests of your engine will be the games that you write, but you will need smaller tests throughout your development so that you can verify your work incrementally.

You will be asked to submit a plan, with projected completion dates, as part of your documentation for Due Date 1.

If Things Go Well

If you complete the entire project, you can earn up to 10% extra credit for implementing extra features. These should be outlined in your design document, and markers will judge the value of your extra features.

To earn significant credit, enhancements must be algorithmically difficult, or solve an interesting problem in object-oriented design. Trivial enhancements will not earn a significant fraction of the available marks.

Submission Instructions

See **project_guidelines.pdf** for instructions about what should be included in your plan of attack and final design document.