CS2134 Data Structures and Algorithms

Extra Credit Assignment B: Scott's Games

Due 11:00 PM November 26 2016

Core Instructions

This assignment is extra credit, and therefore completely optional.

This assignment is worth up to 100 extra credit points, to be added back to one homework where you lost the most points on[1] . If you decide to submit this EC assignment, you cannot submit the Maze EC assignment. (That is, you can only select one of the two for extra credit.)

In order to receive credit for this assignment, you must code up a working solution and explain your code to one of the TAs. You will first need to submit this assignment to NYU Classes. You will be able to receive partial extra credit for this assignment, based on how much you have completed. This assignment is purposely slightly open-ended so that you have the freedom to approach it in a manner that is interesting or educationally valuable to you.

This assignment is broken up into three parts. The first part is worth 50 points, the second part is worth 30 points, and the third part is worth 20 points. You will be able to get credit even if you only finish one of the three parts. Below, the instructions for each part are described (courtesy of Scott Lee). If you have any questions on this assignment, please post on Piazza.

1 It's Like Procrastination, But Classy (Queues)

This part of the assignment is worth 50 points.

The Event Queue and You

For those of you who play a lot of fast paced games, you might be familiar with the term "frames per second" (fps). This is a general term used to refer to the number of times the game state updates in a single second, or to put it another way, the number of times the game loop runs in a second.

During each run of this game loop, the computer is busy processing all the things that happened in the last few milliseconds. But what if your player decides to click on something or hit a bunch of keys during this processing step? The game shouldn't drop everything it's doing to accommodate the inputs. Instead, it could add these events to a queue, and process them in all in the next update cycle. It's almost as if your game gets a bunch of requests, pushes it off onto some to-do list, and when it feels like it, it'll finally get around to doing that thing you've been nagging it to do.

This way, we can process all of the events that happened during our cycle in the order they happened, without potentially breaking something because of a poorly timed interrupt. We as players don't really notice this behavior because games tend to run at a pretty high fps, but if your game suddenly gets really slow, and you click a bunch of things, you might see the game try to process all of your clicks in the same frame. This method of deferred execution is also pretty commonly used in systems that have to process a lot of requests at the same time.

[1] That is, these extra credit points do not add directly on top of all your homework scores together. Rather, the extra credit will only count for **points back on the one homework assignment** where you lost the most points.

Exercise

For this exercise, we're going to be creating a twist on the old-fashioned RPG-style battle system. Our battle system is going to follow these basic rules:

- The game starts with two teams of 3 Fighters each (a player team and an enemy team), which have 5 health each.
- A Fighter is capable of attacking, countering, and resting.
  - An attack hits a designated target for 1 health.
  - A counter deals damage if and only if the Fighter was attacked earlier in that turn. It deals 2 damage to all Fighters who attacked the user earlier that turn.
  - Resting recovers 1 health. A Fighter's health cannot be raised above 5.
- When a Fighter's health reaches 0, they die, and are removed from the fight.
- The battle ends when all of the fighters on one team are dead.

These are the basic rules of our game. If you are feeling creative, these rules are pretty flexible, so feel free to have some fun with it. However, you need to implement how turns work in exactly this way:

1. Randomly select a Fighter (either player or enemy), and call its getChoice() method (which you will implement) that returns some representation of a choice (between attacking, countering, and resting). Keep in mind that attacking requires a target, and so your representation should be able to handle that.
   - For an enemy Fighter, you can select the move randomly.
   - For a player Fighter, you should take input from a player (by cin or otherwise). You should give the player information about the game state when this happens to help them make their decision, but make sure not to reveal previous decisions! We want to keep that a surprise.
2. Perform step 1 a total of 4 times, ensuring that no Fighter is chosen twice. Essentially, 4 of the 6 fighters in the game will get to move on a given turn.
3. Process the moves in the order they were gathered.
   - Keep in mind that if a Fighter is supposed to perform an action this turn, but died before they could do so, then their action is cancelled.
4. Print out the events that transpired this turn in the order they transpired.
5. Repeat steps 1, 2, 3, and 4 until the game is over.

Figure 1: Everything is better when you make it about pancakes.

2 Towers of Hanoi (Stacks)

This is the second part of the assignment, and is worth 30 points.

2.1 How to include Figures

Introduction

The Towers of Hanoi is a classic puzzle involving moving disc shaped objects of varying sizes from one tower to another. Why? I can hardly imagine, but hey, it's an interesting puzzle.

The rules are as follows: There are three towers. One tower already has a full stack of disks on it. To win, one must move the entire stack from the far left tower to the far right tower. Players are only allowed to move one disk at a time, and only off the top of one of the three towers. This disk can be placed on the top of either of the other towers. Additionally, larger disks cannot be placed on smaller disks.

Exercise (Part 1)

Recreate the towers of Hanoi puzzle using stacks to represent towers. To make a move, the user will input two integers, both between 1 and 3. A disk will be moved from the top of the tower corresponding to the first integer, and moved to the top of the tower corresponding to the second integer. If the move is improper (placing a larger disk on top of a smaller disk), we will disallow the move and notify the user. The game ends when all of the disks are on tower 3.

Exercise (Part 2)

There are few hotkeys I use more than Ctrl+Z. I don't know what I would do without undo. For this exercise, we'll be implementing undo and redo for the Towers of Hanoi program made earlier. In addition to the integer inputs, we will allow users to input "undo" and "redo". When undo is used, it will reverse the most previous action. It can be used repeatedly to reverse multiple actions, eventually going back to the original state. Redo should only be used immediately after an undo, and it reapplies the action reversed by undo. If an action wasn't undone, redo shouldn't do anything. Redo can be used as many times in succession as undo was used, and should redo actions in the reverse order they were undone.

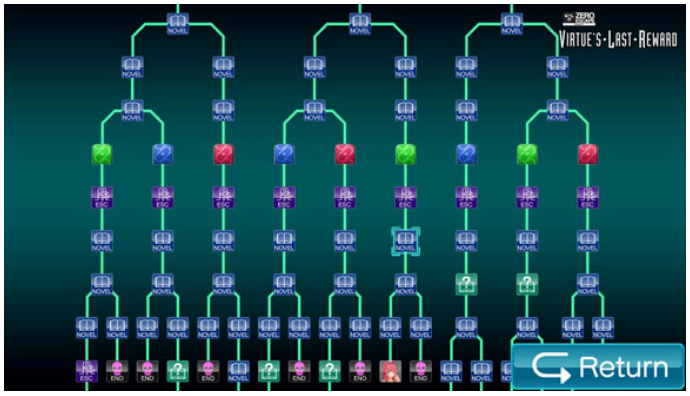

Figure 2: Because linear stories are so 20th century.

A Storytelling Tree

This is the third part of the assignment, and is worth 20 points.

Introduction

When I really think about it, probably my personal favorite game on the 3DS is Zero Escape: Virtue's Last Reward. For those of you who are unfamiliar, VLR is a visual novel, a genre of games which revolves around the story being told, and most of the gameplay revolves around making choices to affect the narrative. Most commonly, players will be presented with choices at key points in the game, and the story diverges based on the choice they made. In many ways, it's kind of like a digital choose-your-own-adventure novel. One of the things that sets VLR apart from other visual novels is the Flow system, which presents the narrative as a tree, and allows players to traverse it, and see where they are in the timeline. If they're unhappy with how a decision ended up, they're free to go back to that decision point, or any other decision point for that matter, and go down a different path.

Exercise

For this assignment, you will get to tell your own story.

- You will load your story and parse it into a tree. This time, you get to decide how your text file is formatted. You have total control over the structure by which you save and load your narrative. You will be asked to explain your choice, and walk through how the tree is constructed. Note: YOU ARE NOT ALLOWED TO USE ANY PRE-BUILT SERIALIZATION LIBRARIES. You have to write your own parsing code
- You will write code that allows us to explore your narrative. Print the text, take an input, and print the appropriate response. Just using cin and cout to do this is perfectly acceptable.
- Optional (No extra credit for this, just an exercise for those who want the practice): allow players to undo their most recent choice, and bring them back to the decision point. Undoing repeatedly should eventually bring players back to the beginning.

Requirements: Your tree should have a height of at least 3 (at least 2 interactions to reach an ending).
Note: You are not expected to write the next great American novel. You will not be graded on the quality of your story, only on the code you use to present it. If you want to have a simple series of yes or no questions, that's your prerogative. If you want to have fun with it, please do so. I encourage it.