EECS 211 Lab 5

Type Racer

Winter 2019

Today we will be looking at another a C++ program using the GE211 game engine in a slightly more advanced example game. The concept of the game is simple enough: Words appear on screen as letters in circles, and you to type the letters you see on the keyboard in order.

This game uses the model–view–controller pattern not-yet-described in class, which allows defining the look, user interaction, and "business logic" of an interactive program as separate components. Provided are the `Model` class which defines the internal game state, the `View` class for rendering game to the screen, and the `Controller` class for reacting to user input and tieing it all together.

Getting the starter code

For this lab, the starter code is provided as a ZIP file here: http://users.eecs.northwestern.edu/~jesse/course/eecs211/lab/eecs211-lab05.zip. Extract the archive file into a directory in the location of your choosing. Once you have your new directory containing the starter files, you can open it in CLion.

Be careful, as CLion will only work correctly if you open the *main project directory* with the CMakeLists.txt in it. If you open any other directory, CLion may create a CMakeLists.txt for you, but it won't work properly.

General idea

You have been given a fully functioning Type Racer that loads a dictionary file (which can be found at Resources/dictionary.dat), and then displays each word from the dictionary—in order—as letters inside colored circles. The player's goal is to type the word, and the controller takes keyboard input to update the player's progress through the word. The circles start out yellow, and as the player progresses through the word, each circle changes to green for a correctly typed letter, or red for a mistyped or timed-out letter. Upon finishing a word the game loads the next word, until all words in the dictionary have been exhausted. Try to identify these components and trace their logic in the source code provided before continuing.

Randomize the dictionary

In controller.cpp, one of the constructors for the `Controller` class calls a helper function `load_dictionary()` for loading the dictionary file into a `std::vector<std::string>`. Since the dictionary file is alphabetized and the model goes through the word vector in order, this means that you see the same words, starting with “a,” every time.

Your job is to modify the code of the `Model` and `Controller` classes to randomize the order of the words after the words are read in. You will do this with a Fisher-Yates shuffle, which is a simple and efficient algorithm for randomly permuting the order of a vector. The algorithm is:

In other words, if the vector has length n, first you choose a random element from index 0 to $n - 1$ to put in position 0. Then choose a random element from index 1 to $n - 1$ to put in position 1, and so on.

```
procedure SHUFFLE($v$: vector):
    for $i$ in 0 to len($v$) $- 2$:
        $r \leftarrow$ a random integer from between $i$ and len($v$) $- 1$;
        swap($v[i], v[r]$)
```

We don’t want the model to randomize the words unconditionally, because that would make testing too difficult. So instead, the shuffling itself should happen in a new member function of the model at the controller’s request. Here’s one way you can do it:

1. Add a public member function to the `Model` class whose purpose is to shuffle the `dictionary_` vector. For a source of randomness, this function should take a reference to a `ge211::Random`. If `rng` is a `ge211::Random&` then you can generate a random integer between `a` and `b` (inclusive) with the call `rng.between(a, b)`.
2. Add a call to your shuffling function to the body of the `Controller::-Controller(std::string const&)` constructor. That way, when the game reads words from a file their order is randomized, but you can also avoid the shuffling by providing the vector directly.

Keep score

Another thing that would be nice for this game is to add score keeping of some kind. You could give 2 points for every correct letter typed, -5 for every incorrect letter typed, -1 points for every letter timed out, and 10 points for every word completed without errors. When the game ends, have it stop and display the score instead of looping on the word “gameover.”

Here’s a plan:

1. Add a private member variable to the model to hold the score, and a public member functiom to allow the view to access it.
2. Figure out how to detect the scoring events in the model code, and update the score for each.
3. Add private `ge211::Font` and `ge211::Text_sprite` member variables to the view class. For creating the font, note that `"sans.ttf"`

Don’t write a “setter,” because no one needs to set the score from outside the model.

Define constants for the event values. No magic numbers!

is included with GE211. For the text sprite, the initial text should be the number 0.

4. Modify `View::draw` to:

   (a) reconfigure the text sprite to contain the current score, and

   (b) add the score sprite to the sprite set.

   In order to reconfigure the text sprite, you will need to create a `ge211::Text_sprite::Builder` and then add text to it. It looks something like this:

```
my_text_sprite_.reconfigure(
    ge211::Text_sprite::Builder(my_font_) << my_value)
```

You might also modify the view to keep track of the last score that it saw, so that it only needs to reconfigure the text sprite when the score changes.

Testing

It's hard testing whether your shuffle is producing permutations uniformly. You could do repeated trials and check that you get a reasonable distribution, but that's fairly tricky as soon as $n > 2$. Easier, however, is to use `std::is_permutation` to check that the result of your shuffle is a permutation of the original. `std::is_permutation` can take two begin–end pairs of *iterators* to compare, and `std::vector` has public member functions `begin()` and `end()` for getting such iterator pairs. Thus, you can check whether two vectors `v` and `w` are permutations of each other with:

```
    CHECK( is_permutation(v.begin(), v.end(),
                          w.begin(), w.end()) );
```

You should also test that your model detects scoring events and keeps score properly.

Other ideas

What if the game displayed a count-down timer for each letter's timeout? (How can you format seconds with one decimal digit using C++'s iotreams?) Or instead of the numeric time, what if it showed a bar whose length shrunk as the time ran out? (Does that require creating a new `Rectangle_sprite` to change its size, or can you scale it by passing a ge211::Transform to the four-argument form of `add_sprite`?)

Can you make the game time the entire word for `2 * current_word_.size()` seconds instead of 2 seconds per letter as it does now? (Then you'd really want some kind of display of the time remaining.)