Creating a game with a GUI - DEV4

Nathan Furnal (55803)

August 15, 2022

Contents

1 *Introduction*

This paper will cover the different parts of the project and the implementation choices that were made. In short, the project follows elements of both the MVC (Model-View-Controller) pattern and the Observer pattern.

Then, the model is served as a static library, which is used in the TUI, GUI and tests.

While the TUI firmly relies on the MVC pattern and only uses the Observer pattern to update the display; the GUI sheds the Controller to rely strictly on Qt's Model-View architecture where the View behaves as a Controller as well. Qt's own signals and slots were used to deal with the Model-View interaction.

2 *Modeling*

To get a feel of the rules of the game and a high level overview of the code, please read the `Doc.md` and `README.md` files respectively. Here under, I detail both the C++ specific features that I used as well as the algorithm managing players' moves.

The three classes you will find in the model are: `Button`, an enumeration of possible values for the button, `ButtonStack`, which represents one stack of buttons and `Game`, the overall game, containing 9 stacks of buttons, per the game's rules.

There are also the `Observer` and `Observable` abstract classes for the pattern implementation as well as a `util.h` header, useful for the TUI (for parsing mainly).

2.1 Technical details

I tried to maximize the use of `constexpr`[o] To that end, `ButtonStack` is a header-only class that uses `constexpr` exclusively.

[o] Which lets us use constant expressions, for evaluation at compile time. Note that it isn't a guarantee, functions merely become eligible for compile-time evaluation.

I tried to apply that logic to the game class as well, which wasn't always possible because I used vectors, which are not `constexpr` friendly as of C++17.

The implementation was kept simple: the `ButtonStack` is a hybrid of a list with constant time ($O(1)$) insertion and an array with constant time random access. This is possible because the size of the array is known beforehand and the game's mechanics are simple. This class also has helper functions, for example to check if the stack is empty, if it has a white button or if the two buttons at its top are the same color.

The `Game` class introduces state management to know which player's turn it is at any given time. It also keeps track of the relevant round and game values to display the score. Also, some of the methods have a `bool testMode` variable added, which is never used in the game itself but was introduced to ease testing.

2.2 Algorithms & Optimizations

I added a couple optimizations when possible. For example, the rule book remarks that the game always ends in 8 turns. Which means that no specific analysis of the game is required and only counting turns lets us know when the game ends.

Another one, was to find the non empty stacks in the order they will be used in, rather than simply iterating over of the stacks from the beginning every time. When a move is played in a column, the iteration is started from that column, not sorting or re-arranging of items is necessary.

```
std::vector<int> Game::nonEmptyStacks(int startIdx) {
  std::vector<int> nonEmptyIndices;
  nonEmptyIndices.reserve(N_STACKS); // Avoid resizing because we know the max size beforehand
  for (size_t i = 0; i < N_STACKS; i++) {
    int idx = (startIdx + i) % N_STACKS;
    if (!m_stacks[idx].isEmpty()) {
      nonEmptyIndices.push_back(idx);
    }
  }
  return nonEmptyIndices;
}
```

There is not much in terms of algorithms but for the buttons movements during the game. The algorithm is displayed here under and I'll explain the different steps.

The move is entered only when the stack in question contains a white button. Then, the method keeps track of the possibility of replay, the indices of non empty stacks as well as the number of buttons inf the current stacks and the number of stacks available.

If the number of stacks available is superior to the numbers of buttons to distribute, it's simply a question of pushing each button on top of those stacks and if the last stack has the two same buttons on top, the current player can play again.

In the case of having more buttons to distribute than valid stacks, it is a matter of repeating the above process until there is only one stack left and then dump all the remaining buttons on that last stack.

Finally, empty the current stack because it will always be empty after the move, increase the turn counter and update the state accordingly.

```
void Game::moveStack(int srcPos, bool testMode) {
    if(!m_stacks[srcPos].hasWhite() && !testMode){
        throw std::invalid_argument("Button stack does not contain a white button.");
    }
  bool replay = false;
  auto nonEmptyIndices = nonEmptyStacks(srcPos);
  int nButtons = m_stacks[srcPos].nButtons();
  int nStacks = (int)nonEmptyIndices.size() - 1; // because current stack shouldn't be counted
```

```
  if (nButtons <= nStacks) {
    for (int i = 0; i < nButtons; i++) {
      // avoid pushing to self, hence +1
      m_stacks[nonEmptyIndices[i + 1]].push(m_stacks[srcPos][i]);
    }
    if (m_stacks[nonEmptyIndices[nButtons]].has2SameButtons()) {
      replay = true;
    }
  } else { // number of buttons is strictly superior to available stacks
    int i = 0;
    while (i < nStacks) { // do the same as above until there's one stack left
      // avoid pushing to self, hence +1
      m_stacks[nonEmptyIndices[i + 1]].push(m_stacks[srcPos][i]);
      i++;
    }
    // dump what's left on the buttons in the last usable stack
    int lastIdx = i;
    while (i < nButtons) {
      m_stacks[nonEmptyIndices[lastIdx]].push(m_stacks[srcPos][i]);
      i++;
    }
  }
  // non-stack operations were used and the current stack was emptied, reset is needed
  m_stacks[srcPos].reset();
  m_turns++; // increase turn counter after a successful move
  // depends on the state
  if (!replay) {
    m_state = (m_state == GameState::RED) ? GameState::BLACK : GameState::RED;
  }
  if (isGameOver()) {
    m_state = GameState::GAME_OVER;
  }
  notifyAll();
}
```

3 Graphical interface

The graphical interface makes use of two very useful Qt classes : `QTableView` and `QAbstractTableModel`. In short, the abstract model already provides useful signals to update the view and the view can easily interact with the model by signaling any click it receives. This design is commonly used for spreadsheet like behavior[1] but I used it to represent a 2D view of the game instead. This strategy completely bypasses the need to manage buttons independently since they are a part of the `QTableView`.

Once this is done, it is just a matter of implementing the signals and slots necessary to update the view after each move and keep the model up to speed with which buttons were pressed.

[1] In fact, the Model-View tutorial demonstrates a simple interactive spreadsheet example.

4 Memory analysis

Once the game was functional, I ran `memcheck`[2] on several occasions to find any possible unfree'd memory or memory leaks. Because of Qt's liberal use of pointers, most of the errors had to be corrected there, mostly because memory initialized in the constructors wasn't freed by the end of the program.

This was fixed in later iterations and now all memory is freed when the program ends.

[2] Which is included in the Valgrind suite of tools.

5 Conclusion

The game was optimized using modern C++ patterns and represented using native Qt functionalities (signals & slots). Memory analysis was performed as well to fix any potential leaks or unfree'd memory. More information about the methods can be found in the documentation generated with `Doxygen`.