Project documentation

Tile-matching

Project Manager and Front-end developer: Ella Eilola
System architect: Jaakko Koskela
Back-end developer: Leo Kivikunnas
Software Engineer: Eero Hiltunen

Project overview

Our project was to build a tile-matching game in style of Candy Crush and Bejeweled. Our goal was to have a game mode, start menu, pause menu, high-score listing, map editor and special game modes.

For the game itself we implemented basic back-end functionality and game user interface with sounds and graphics. Game back-end contains functions e.g. for checking validity of moves, filling map, clearing matches and keeping score. Game UI checks user inputs and draws updated maps to window. Maps are read from local text files.

The program has two menus, a start menu and a map menu for choosing maps, as well as a scoreboard that lists top 10 local scores. The initial plan was to implement a pause menu as well but we didn’t see it as a necessity in the end. Also AI and multiplayer-functionality were seen irrelevant in this kind of game.

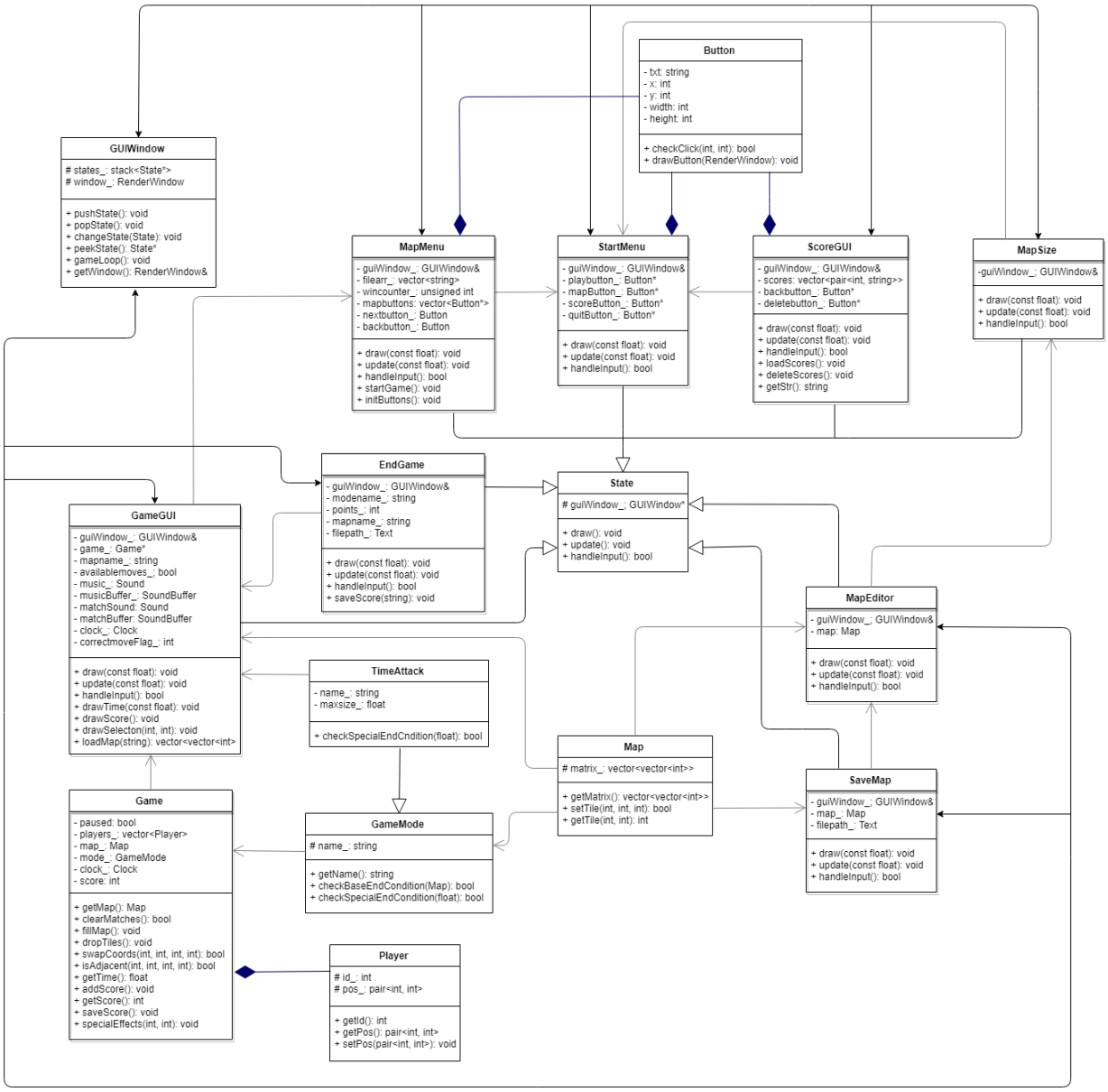
Software structure

The program has multiple classes, most of which are different states, e.g. for menus, scoreboard and the game itself. Each class is defined and implemented in its own .hpp- and .cpp-file. State classes inherit from a pure virtual parent class State. In addition to this, the program has a state manager class GUIWindow, Game for game back-end and some other small classes.

When the program is launched, an instance of GUIWindow is created. GUIWindow has a stack of State-objects and a game loop -function. At the beginning of the program a StartMenu-obejct is created and pushed to the stack, window is created and game loop -function is called. The game loop calls user input, draw and update methods of the state on top of the stack. In each state class there’s a state specific event loop which checks user inputs like button presses.

Once the map is set in MapMenu, a GameGUI-object is created with a map parameter and pushed to the stack and the game starts. First the map is loaded from file and filled with random tiles and a Game-object is created. The GameGUIs event loop will loop for user input until a valid move is made or user closes the window. GameGUI calls Games methods for filling the map, dropping tiles, clearing matches and checking validity of moves. The GameGUI-class will handle background sounds, drawing graphics and user inputs related to running the game and and calls Game for methods handling the game logic. This offers clear division between game back-end and user interface.

The user interface is made using SFML -multimedia library. SFML was used for creating window, drawing graphics, checking button inputs and background sounds.



Class diagram

Introduction for building and using the software

The program uses the SFML -graphics library. To install SFML on Ubuntu, follow the official SFML guide (https://www.sfml-dev.org/tutorials/2.5/start-linux.php). SFML 2.3 should be installed by default on Aalto Linux machines. Because the Aalto machines have version 2.3, we decided to use version 2.3 aswell. If you are running a newer version of SFML, the compiler might produce some warnings, but the program should still compile.

To compile and run the program, use the provided makefile. To use the makefile, open a terminal and navigate to tile-matching-2/src, and type “make run main”. To clear output files, you can type “make clean”.

The software has a click button based GUI, and thus the usage is very intuitive. The menus are navigated by clicking said buttons. In-game, the swaps are performed by clicking two tiles.

Testing

The program was tested mainly by using it. When the GUI was not yet implemented the testing was made by giving the program some inputs and printing outputs to console. When the game was almost ready we ran Valgrind tests for memory debugging. Valgrind tests produce no errors.

Future improvements

Currently, new tiles appear into the map from “nowhere”. To improve visuals, they could be dropped using the already existing drop algorithm. This could be achieved by building another 8x8 grid in the backend above the map shown in the gui. Then the tiles could be spawned into the grid above the shown map and then dropped into it.

Work log

The distribution of roles in our group was based on group members skills and personal preferences. Therefore, they were the main factors when considering the assignment of suitable roles for each member. Ella took care of the project management tasks, which included distribution of tasks, problem solving and scheduling. She was also responsible for coordinating the documentation of the project. Jaakko was responsible the back-end architecture and algorithm design. Leo was the main back-end developer of the project and he did most of the work in the beginning of the project before the GUI was implemented. Eero made sure that the user requirements were met in the final product and that agreed development method was followed accordingly. He also worked with the GUI extensively.

The following roles were assigned this accurately to distribute the burden of software development equally to all members of the group. It also needs to be noted that every member of the group were involved in the basic programming work that included the back-end and GUI.

Description of what was done each week:

Week 1 (Ella 5 h, Leo 5 h, 2 commits):
Leo did some initial coding for the back-end, and Makefile for compiling was built.

Week 2 (Eero 7 h, Ella 3 h, Jaakko 10 h, Leo 8 h, 16 commits):
We wrote the project plan and designed the software. Significant time was also spent to create a UML diagram of the software structure. Some of the back-end classes and algorithms were implemented.

Week 3 (0 commits):
Rest week

Week 4 (Eero 3 h, Ella 2 h, Jaakko 2 h, Leo 10 h, 15 commits):
Final work for the back-end was done and the decision to use SFML for the project was done. We also got familiar with SFML.

Week 5 (Eero 15 h, Ella 20 h, Jaakko 6 h, Leo 10 h, 29 commits):
We had to rehaul our software structure a bit to make it work properly with SFML. A lot of work was done to get the basic GUI to function.

Week 6 (Eero 3 h, Ella 5 h, Jaakko 15 h, Leo 1 h, 29 commits):
Initial bugs of the new software structure were fixed and new features added for the gui, such as time and highlighting of tiles. In addition, reading maps from files was made possible.

Week 7 (Eero 35 h, Ella 20 h, Jaakko 32 h, Leo 36 h, 83 commits):
A lot of work for the software was done this week. Each day we delivered new valuable features and fixed bugs. We added features, such as high score, map editor, game modes and special blocks. There’s dozens of commits so it’s difficult to describe the things done in details but a lot was done to say the least. We also wrote the project documentation.

Self-assessment of own project

1. **Overall design and functionality (6/6p)**
    - 1.1: We produced a tile-matching game as requested and implemented all required basic features. We also added extra features, such as the map editor, high score listing, sounds, special game modes, and animations. Implementing these features kept the four members of our team busy as all of the work was done only in 6 weeks.
    - 1.2: The software structure fits the project and makes adding extra features and screens simple. The structure is documented and shown in a UML diagram.
    - 1.3: Use of an external library, SFML, was documented and well justified.
2. **Working practices (5/6p)**
    - 2.1: Git was used actively throughout the entire project and descriptions of commits are understandable and logical
    - 2.2: Every member of the group worked with the project in regular basis and had a relevant role that was stated in the project documentation. Roles were assigned based on the individual group members skills and preferences.
    - 2.3: Quality assurance was done by testing the software by running it extensively and these methods were documented.
3. **Implementation aspects (6/8p)**
    - 3.1: Building the software is easy with the provided ‘Makefile’ and the use of it is instructed in README.
    - 3.2: No memory leaks or segmentation faults were detected in the testing process that included valgrind testing. RO3/5 was followed diligently.
    - 3.3: We utilized C++ standard library, for example, in storing multiple objects in containers instead of own solutions.
    - 3.4: Exception handling is mainly done when loading files and the implementation works robustly in situations where user does something unexpected.

4. **Project extensiveness (9/10p)**
    - Project contains features beyond the minimal requirements: We added extra features, such as the map editor, high score listing, sounds, special game modes, and animations. All of the additional features that were listed in the project instructions were not implemented as they were not reasonable for our project. However, our course assistant encouraged us to add animations and we did just that.
5. **Project plan (3/3p)**
    - Our project plan described all necessary aspects and was as specific as possible.