Connect-Four on Tremola

Tim Goppelsroeder, Rasmus Jensen

July 17th, 2022

1 Preamble

The game of Connect-Four has been enjoyed by lots of kids over multiple generations(including us!). The basic game is quite simple and resembles Tic-Tac-Toe. However we thought additional rules/options might be interesting and make this classic game more exciting.

For those who have never played the classic version of Connect-Four there are six columns and six rows. One can let tiles slide down columns and into place at the bottom. Both players have a separate color. The goal is to connect four tiles in a horizontal vertical or diagonal line.

Our Version includes UFO tiles(i.e. floating tiles) and bomb tiles(i.e. exploding tiles). Floating tiles can be placed at a given height and do not fall down the whole column until they reach the last free space like their normal counterparts from the classic game. Explosive Tiles destroy Tiles to the left ,right and below(theoretically also above but this is practically impossible). Additionally each player can use the normal tiles indefinitely however the two special tiles (UFO and bomb tiles) can only we used once a game by each player respectively.

2 Overview

Front-end:

In Tremola the Frontend refers to tremola_ui.js which contains the game logic and tremola.html which contains the GUI/GUIs for the Tremola application. For our game both of these JS classes needed to be extended to include a GUI for the initialization window, loading window and main window (where the game is actually played). The Game itself is depicted as a table of html buttons where all these buttons are connected to functionality in tremola_ui.js which implements the game logic of a button being pressed. This constitutes letting a tile fall from row to row in a column for a certain amount of time that has elapsed until the tile reaches the last unfilled row in that column or if the whole column is full the move is invalid and the player must repeat his move. The addition of bombs and UFOs was realized by adding buttons that act as switches in the main GUI that allow players to choose between UFO, bomb and normal tiles. When a button is pressed (position in the connect-four game and not one of the aforementioned switches) then the move functionality in tremola_ui.js is also given the type of tile that is indicated by the currently active switch in the game section of tremola.html. UFOs will fall like normal tiles until (but until designated position/row reached and not till bottom/last free space). Bombs fall exactly like normal tiles but once their terminal position is reached they explode and destroy tiles to the left and the right as well as above and below the locus of the explosion.

The file tremola_ui.js additionally contains following functionality:

- to evaluate if a game state is winning (has a player won already). This is done by checking if any entry in the table is part of a solution(could be made more efficient but overhead for a 6 by 6 matrix is constant and not cumbersomely large).
- that checks for a draw which essentially means that the top row is checked and if the top row is full and no one has won the game is a draw(no more space).
- to choose which tile one wishes to use (UFO, bomb or normal).
- to get the list of contacts one could play against and display them as buttons in the GUI.
- to implement waiting on the other player to connect(waiting for game to start).
- to reset the games initial menu state as well as functionality to reset the game state.
- to switch between initial game menu loading window and game window.

- Functionality to start and end games.
- front-end functionality to help determine turn order.
- sleep functionality implemented so we can use delays to make explosions look better/more realistic.

Back-end:

We created a new method/function for in game communication so that the original function for the chat doesn't need to be used (as this would result in game protocol messages being sent via chat as well). To implement this we had to create a new SSB message type for our game specifically.This enabled our game to be able to react only to the messages in the append only logs that were related to the game protocol. Whenever a client receives a new log, this event log passes through the web app interface, specifically through a method called *rx_exent*. Once in there, we check whether the event log is of type game, that is, if the message relates to the game protocol or not. If so, we parse the log and depending on the content, we evoke certain methods in the front end. This way, when connection is established between the two clients, so when the clients share a WiFi connection, the backend can react to these new logs that are being sent and received, and call respective front end methods. The back end offers following functionality:

- In **SSBmsgTypes.kt**: We added a method mkGamePost, which is identical to mkPost, except that the event logs created contain the attribute type game instead of chat. This makes any game related post easy to find.
- In **Game.kt**: A data structure used to store important, game-related settings.
- In **WebAppInterface.kt**: List of received invites *from*, and a list of sent invites, *to*. These structures are used to decide, when a game can be started. When a game starts between two clients, they each remove the peer's address from the *from* and *to* lists.
- In **WebAppInterface.kt**: Some new when-statements have been added to the method *onFrontEndRequest*.
  removeGame - sets current game variable (from Game.kt) to null.
  startGame - puts the client in a waiting screen, and sends a new game post with the invite to the peer. It also adds the peer in the *to* list, which captures all outgoing invites.
  gameProtocol - sends a game protocol related post to the peer. The post contains the method name and method parameters for the peer to evoke in the frontend (game moves mostly).
  onBackPressed - additional code to ensure that when one client presses on back during a game, the game ends for the peer aswell. Again, this is done with a game post.

- In **WebAppInterface.kt**: The method *rx_event* filters the events logs relating to the game and parses these game logs by calling *checkGameEntries*.

- In **WebAppInterface.kt**: To have game logs evoke the correct methods, the method *checkGameEntries* parses the private message in the log and depending on the prefix, certain actions are made.
  /ed25519-start: onBackPressed() - This is the protocol for returning to the main screen of tremola. If this log is received from the peer, it means he has ended the game.
  /ged25519-prot: - This is the main game protocol. It takes the method name and parameters, and calls *eval*, to send the request on to the front end.
  /ed25519-start: - This prefix is used only for receiving game invitations. This also adds the sender to the *from* list, which captures all incoming invites.

- In **WebAppInterface.kt:** The method *checkConnectionOrder* checks if a game is ready to start, and asserts who is playing the first turn. A game is ready to start, once a client is both in the *from* and *to* lists. The turn order is determined by who first sent the invitation. Once a game starts, the client is removed from both lists, to ensure consistency.

communication between Back-end and Front-end:

From front-end to back-end a function called backend() was used to send info to the back-end. From back-end to front-end the function eval() was used to call JavaScript functions in the front-end (with specific arguments even) within the Kotlin code in the back-end. The communication was very confusing at first, but we eventually managed to use the communication channels efficiently.

3 Difficulties

Front-end:

- making the main screen, loading screen and game screen(focus on interoperability)
- implementing falling, exploding and floating tiles (i.e. the core game logic is in pressButton(x,y,is_me))
- learning HTML and how to connect JavaScript functionality to HTML buttons
- Planning out interoperability of game functionality (i.e. integration of individual parts (functions/game-screens etc.) of the game)
- making/planning the layout of the GUI
- make falling tiles and explosions look realistic with delays(i.e. sleep functionality to make explosions look sequential from a temporal standpoint)
- be able to display the contacts one can start a game with as buttons (couldn't have a button list of certain length because contacts has arbitrary size)
- implementing a Min-Max Tree and algorithm which goes through this tree to create a single player AI that can play our game against the user in offline mode (not finished)

Back-end:

- understanding how to interact with the Event logs, how to parse them and manipulate them correctly.
- It was hard finding out how and where to extend the code base. It seemed as though there were many ways of extending, and the trouble was to find where it might be best to do it.
- An issue we had was finding a place to read from the logs, whenever they would be updated. We then needed to find a way to filter out unnecessary logs.
- In several places, we needed to create our own logs. We needed to make sure that the logs we created adhere to the protocol created, and was correctly spotted when filtering for new logs.
- MainActivity.kt is the file, in which the app loading is specified. How to create game states there, and when to initialize it became a problem.

- We noticed that for event log creation, it would be easier if we could change the log type, instead of having to use the same message type as the private chat did.
- The implementation of the first turn boolean on both peers was an initial breakthrough in the backend. The use of *to* and *from* lists made the problem easier.

communication between Back-end and Front-end:

- Using the *eval* function in the back-end required had a slight learning curve, as the commands for nested strings and string formatting was not known before hand.
- adding when statements to the FrontToBackEndRequest() method
- debugging errors when doing back-to-front or front-to-back requests were always pretty hard to catch, as there was no JS debugging in IntelliJ. We could debug in the back-end, though, and look at the last line executed there.
- Quitting the game for a peer when the player decides to quit, this required some slight readjustment of onBackPressed.
- Implementing turns also required some thought of all the edge cases that might exist. We spent alot of time debugging turn errors.
- Adding acknowledge logs for when a game was decided to start

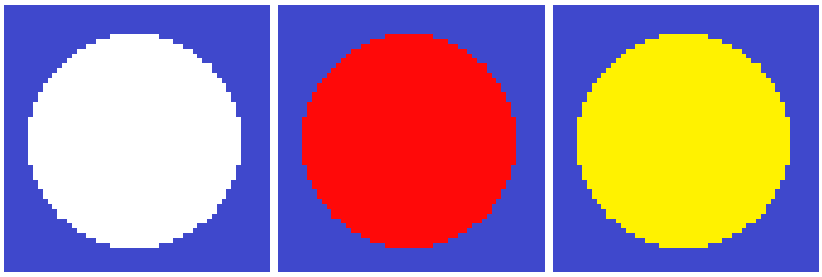
4 Overview of game-play and GUI elements

4.1 Game-play

The game starts with it being the red tile players turn. They can then choose a tile type and position that constitutes their move. There are three different tile types, the default tile that acts like a normal tile and can be used arbitrarily many times, the explosive tile that may be used only once by each individual player and destroys tiles below and to the left and right of the explosion (above technically as well but it is infeasible for a tile to be placed below another tile) and the gravity tile which also can only be used once by each player and allows a player to choose a row ass well as the column number where a tile should go if this field is not filled yet. The winning condition (and draw condition) are however the same as in a classical connect-four game.

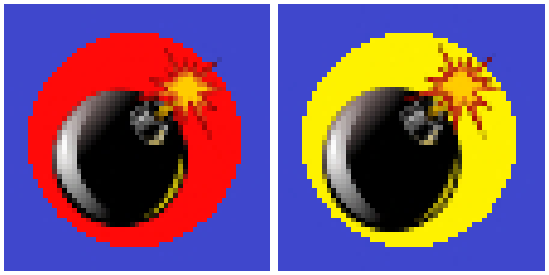
4.2 main tiles

The main tiles used in any connect-four game

- 

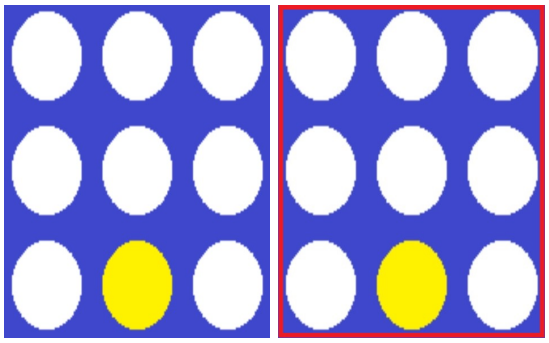
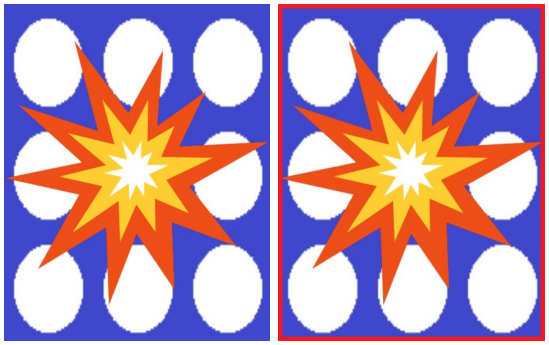
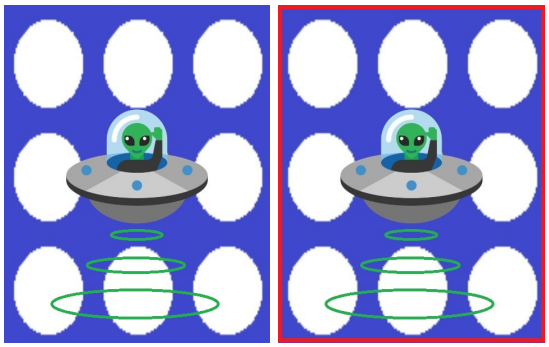
4.3 special tiles

additional tiles that are used in game(bomb tiles)

- 

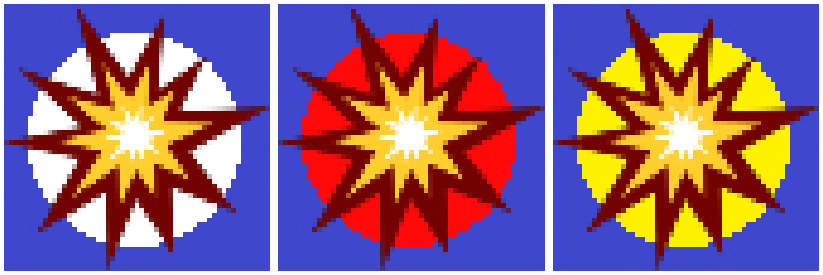
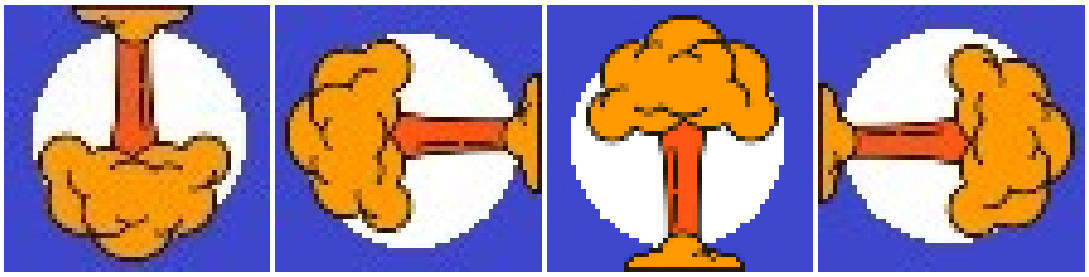
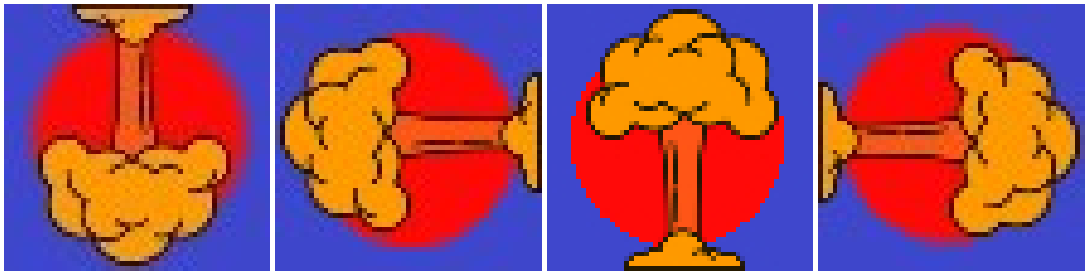
4.4 tile symbols for selection purposes

selected and non-selected version of normal-, bomb- and gravity tile.

- 
- 
- 

4.5 directional/non-directional explosion images

General explosion used for center/locus of explosion. Mushroom cloud image used to show surrounding tiles being destroyed by a bomb(mushroom cloud is a directional explosion).

- 
- 
- 
-