Modern C++ Project: Triviador

System Design Document

Balogh Szilárd, Opra-Bódi Botond, Trofin George Ionuț, Vitályos Norbert

Universitatea Transilvania din Brașov, 2022-2023

This document aims to collect as many design decisions and implementation details as possible during the development of this project. This helps in (eventually) setting ideas in stone, making sure everyone is on the same page concerning the architecture of the system and the features that will and will not be implemented, and how exactly they will be implemented.Everything here is a work-in-progress, sections will probably need to branch off or be reorganized completely during the document's evolution. Remove this comment once the project's done; at that point this will serve as a very detailed documentation. Use LyX notes as comments.

Part I.
Overview

Triviador is a desktop trivia game with a graphical user interface, which can be played by 2-4 humans and/or computer agents over a local area network or the internet.

1. Inspiration

The inspiration for the game comes from a specific mobile game called Triviador. The mobile game has many more capabilities and game modes than what's required of us to implement. We are going to implement only a small (and slightly modified) subset of the game and a single game mode. We are going to call our game Triviador as well in this document, please don't sue.

2. Goals

- Implement all the base requirements outlaid in [1]
  - The abstract game model described in Part II
  - Client–server model with the possibility to create at least 2 client instances and 1 server instance which can communicate over the network
  - User management system and a login/register page. Only registered and logged in users can play. Usernames must be unique
  - User profile, which displays the user's game history

  - The size and shape of the map and the number of rounds should be modifiable
- Use a DBMS for storage of user data, questions, statistics
- GUI for the client to display the login/register page and a visual presentation of the game

3. Non-goals

- Secure network communication and secure user data storage
- Flashy and polished graphical interface
- Other game modes

4. Technologies used

- Microsoft Visual Studio
- Qt
- SQLite
- SQLite ORM [2]

Part II.
Abstract game model

This part contains an expanded and more detailed version of the game description given in the "Triviador" section of [1]. It contains the description of a game of Triviador on an abstract level, modeling the different entities and their interactions that compose the game. This section therefore does not contain implementation details.

5. Short overview

The game is played on a map which is divided into discrete territories. The goal of the game is for each player to amass as many points as possible (more than the other players) by fighting over territories. Players capture territories by dueling with each other in a certain order. A duel consists of one or more questions asked in a series that have to be answered by both dueling players. The player giving the correct answer, the answer closest to the correct answer, or giving the answer fastest, wins the the duel, and captures a contested territory. The exact details will be described in the following paragraphs.

There are two types of question: single-choice and numeric. Both types are generally open-ended. In case of a a *single-choice question*, the game presents 4 possible answers, with only one being correct. A *numeric question* has a non-negative integer as an answer, typically below 100 000, and the player can input an arbitrary number as their guess.

6. Definitions

map A connected undirected planar graph , in which the vertices are *territories* and the edges signify whether two territories are adjacent to each other on the map.

territory A vertex in the *map*. A territory can be either unoccupied, or occupied by a *player*, called the territory's *owner*. A territory has a score, which is initially 100 for an *ordinary territory* and 300 for a *base*. Territories can be *captured* by *players* other than the *owner* according to , in which case the territory becomes occupied by that player and its score is updated according to .

base Each *player* has exactly one *territory* named base throughout the *game*. The base has a score of 300 initially. If a *player*'s base is *captured*, that *player loses* the game, their *score* becomes 0, and all the losing player's territories are transferred over to the capturing player. The detailed rules of territory capturing are described in .

player An agent who takes part in the decision making during a *game*. A player has a *score*, which will determine their *ranking* at the end of the game according to . A player can make the following decisions during a *game*: answer *single-choice questions* and *numeric questions*, activate *amplifiers*, choose *territories* during the *expansion stage* , choose *territories* to *attack* during the *dueling stage*.

player score A non-negative integer assigned to a *player*, which gets updated according to and determines the *player*'s *ranking* at the *determining the winners stage* according to .

territory score A non-negative integer assigned to a *territory*, which gets updated according to and takes part in the calculation of the *player score*.

single-choice question A question for which there are 4 possible answers, of which one is labeled correct. Single-choice questions are only present during the *dueling stage*, where they are the first question asked during a *duel*. Both *dueling players* are asked the same single-choice question, and they can select 0 or 1 of the possible answers. For the full rules concerning single-choice questions, see .

numeric question A question for which there is one, non-negative integer answer. The *player* can give an arbitrary non-negative integer as an their answer. These are the only type of question during the *territory expansion stage*, and the type of the second question in a *dueling question substage*. For the full rules concerning numeric questions, see .

answer correctness order The *players* answering a *numeric question* are ranked in an answer correctness order based on how close their answers are to the correct answer and how quickly they answered. Technically, the players are first sorted in ascending order by $|a_c - a_p|$, where a_c is the correct answer and a_p is the *player*'s answer; in case of equal such values, the affected *player*s are further sorted by the order in which the *players*' answer submissions were registered. The *players* are given an index from 0 to $P-1$ after the ordering, with 0 marking the *player* with the "most correct" answer.

attacker Denotes a *player* whose turn it is to choose one *territory* of an opponent during the *dueling stage*, trying to occupy it. The *owner* of the chosen *territory* is the *defender*. The attacker and the *defender* are entered into a *duel*, whose winner determines the fate of the chosen *territory*.

defender Denotes a *player* whose *territory* was chosen by an opponent during the *dueling stage*. This opponent is called the *attacker*. The attacker and the *defender* are entered into a *duel*, whose winner determines the fate of the chosen *territory*.

duel The duel is the second part of a *round* during the *dueling stage*. A duel is played between an *attacker* and a *defender*. First, both players are asked the same *single-choice question*. I

7. Stages

The game consists of 4 stages: base selection, expansion, dueling, and ranking. They play out one after another in this order. Notations used in this section: P – number of players, T – number of territories, $r(p)$ – ranking of player p, $s(p)$ – score of player p, $\mathcal{P}$ – set of all players.

7.1. Base selection

Every player starts out with a score of 0 and no territories. All players are asked a numeric question. The answer correctness order defines the order in which players will select a base for themselves. Each player is asked in this order to select a territory as their base, with the only restriction that a player can't choose a territory that is directly adjacent to the previous players' choice of base. A base's score is initially 300 and the same amount is added to every player's score.

7.2. Expansion

All players are asked a sequence of numeric questions, and after every question the players choose some territories to occupy. The order in which players are asked to choose territories is in increasing answer correctness order. The nth player in the answer correctness order chooses $P - n - 1$ unoccupied territories to occupy; if the number of remaining unoccupied territories is less than $P - n - 1$, only this number of territories will be chosen by the player affected. This means that the maximum number of territories chosen after a question is $\frac{1}{2}P(P-1)$, and therefore the number of questions asked in total is $\left\lceil \frac{2(T-P)}{P(P-1)} \right\rceil$. A player has to choose all their territories before the next player can choose. A particular territory can be chosen if exactly one of the following conditions is satisfied: 1. it's unoccupied and adjacent to a territory occupied by the currently choosing player, or 2. there are no unoccupied territories adjacent to any territory occupied by the current player. The territories are chosen one by one, therefore these conditions are reevaluated after every choice. Every territory occupied in this stage is assigned 100 points, and this amount is added to a player's score after every territory occupied.

7.3. Dueling

This stage consists of a variable number of rounds, $2P$ by default. Every round, a player (*attacker*) chooses a territory occupied by another player (*defender*) to try and take it for themselves. The order of attackers is determined according to. This *chosen territory* has to be adjacent to one of the attacking player's occupied territories, with some exceptions. The attacker and defender are entered into a duel. If the winner of the duel is the attacker and the score of the chosen territory is 100, the chosen territory gets occupied by the attacker. If its score is more than 100, the chosen territory stays in the possession of the defender and the territory's score is decremented by 100. If the winner was the defender, the territory stays in the defender's possession and its score is incremented by 100.

7.4. Ranking

The players are ranked in the descending order of their score. All players p get a numeric rank according to the following formula: $r(p) = |\{p' \in \mathcal{P} \mid s(p') \geq s(p)\}|$. This means that tied players get an equal rank, but lowered rank. The player(s) with the maximum score is declared the winner. This stage marks the end of the game.

Part III.
System

This part is concerned with concrete implementation details.

8. Overview

The project is structured as a Microsoft Visual Studio solution with 5 projects: *client*, *server*, *model*, *net_common*, *common*. The SQLite database engine is used on the server-side for the data storage needs.

9. Solution structure

The solution is divided into 5 projects, each with a well-defined responsibility and minimal cross-coupling to ease development and deployment.

9.1. Project: *model*

This project contains the implementation of the Abstract game model. It is not responsible for spinning up game instances for players to play, that is the responsibility of the *server* project. It is also not responsible for presenting the interactive environment through which the players can make their decisions, that is done by the *client* project (as a GUI implementation).

The game model is implemented using a state machine and the game rules are encoded in its transitions.

9.2. Project: *server*

This project implements a server capable of hosting multiple game instances, to which *client*s can connect and play games over the network. Every server instance manages its own database of users. Clients need to register with the server and log in before they can create and join games.

9.2.1. Database - sqlite ORM

The base component of the whole database management is the ***DatabaseManager***. This is essentially an interface (*virtual class*), that serves the purpose of defining the basic structure of every other specific database manager class. Essentially it encapsulates the name of the database as the private field ***m_databasePath***, the only common feature of all the database manager classes.

This interface is then implemented by all the database manager classes. A concrete example is the ***DatabaseQuestionManager***. Though it has a long name, it is suggestive enough that anybody reading through the code can understand its purpose. In addition to the properties of ***DatabaseManager*** it contains a variable of type ***Storage*** which is actually responsible for dealing with the reading from and possibly writing to the database. With this field at our disposal we can abstract any unnecessary complexity the *sqlite ORM* library might contain (more in the following paragraph).

The ***Storage*** field is a key building block of the database. It is what actually communicates with the database and does the dirty work. To make it possible to have a Storage field, I needed its type. This, however, turned out to be a complicated issue, mostly because this type depends on other template parameters we can't deduce manually. That's why I used **decltype**, which essentially does the same thing, except correctly and seamlessly. We have a functor ***QuestionStorageInitializer*** that returns a storage created as ***auto***. This is then interpreted by decltype and we have our Storage type.

The ***QuestionManager*** is another layer of abstraction on the database manager classes. It is what communicates with the other parts of the program, or with the server. The hard work is mostly done, this makes it easier for anyone outside the database manager classes to get questions with a simple ***getQuestion<QType>()*** method. This method is templatized, so it is really easy to get the specific question needed.

9.3. Project: *client*

This project implements a GUI application through which the users can log onto a server, create games hosted on the server, connect to games, and play them.

9.4. Project: *net_common*

This project contains network communications code common to both the *server* and *client* projects.

9.5. Project: *common*

This project contains common functionality used by the other projects which doesn't conceptually/strictly belong to any of them.

Part IV.
Recycle bin

PUT SENTENCES, PARAGRAPHS, IDEAS HERE IF THEY HAVE NO PLACE YET, SO YOU DON'T FORGET TO ADD THEM TO THEIR PROPER PLACE LATER. YOU CAN USE IT AS A SANDBOX AND WRITE ANYTHING HERE IF IT'S NOT COMPLETELY FLESHED OUT YET OR YOU DON'T KNOW WHERE TO PUT IT. JUST KEEP WRITING, DON'T GET BLOCKED FOR THESE REASONS. YOU CAN ADD SUBSECTIONS FOR SEPARATION.

The player is allowed to choose a single option, without the possibility of changing their decision once it's made.

A player's score is calculated as being the sum of the scores of the territories occupied by the player.

Question Handling System: *Questions* represent a crucial component of **Triviador**. In our version of Triviador, there are ***two types*** of Questions:

- Single-choice Question
- Numerical Question

Questions are part of the **Game** class, they are stored in and read into memory from a **Database** (see []). The underlying data type of the questions is a *vector*. All types of Questions contain a *statement*, which is stored in *string*. While every Question has a *correct answer*, depending on the type of the Question, this can be a *simple numerical value*, an unsigned integer (Numerical Question), or a *string value*, an index to the correct answer in the possible choices (Single-choice Question).

The only problem is, that we need unique questions. It can be done in multiple ways, the easiest being to just have an unordered set of questionID's, and a random number generator from 0 to the number of questions there are, number being the ID, representing the question. If the ID was already read, we will generate another number until we have a unique question. The issue with this is that this can, in theory, take inifite amount of time (let's say we generate already chosen ID's for a long time), and we don't want this to happen while the game is running, and question is solicited. Obviously, this isn't the best solution, but we will use it for testing purposes Another solution would be to store as many questions as we need before a game starts (it will be part of the loading process) in some kind of container (we need fast insertion and deletion, so list is a good choice) with an adaptor. The issue with this second approach is that our QuestionManager class has to know about the number of players, the size of the map to know how many questions to load into the specific container. We really don't want that, because the sole purpose of the QuestionManager (or any other class mentioned above) is to make the connection between the database and the game and to make it easy to solicit questions. So we have chosen the first solution. First problem: We have two types of questions, that are stored in two different tables, so there is no way to differentiate between the question with ID 1 in the first table from the one with the same ID in the second table. We can use two unordered sets, but that seems too inefficient. So we settled at a template function with a simple while loop, that runs until a unique ID is found. I'm yet to implement the unordered set for the numerical questions, and the issue mentioned above is, of course, there, though this will be minimized with the addition of more questions to the database.

References

[1] *Official project guidelines.* https://docs.google.com/document/d/1qI5q5SosuVvnK4QGh8-X_CfR0o-boQCTMbijoQqY72k

[2] *SQLite ORM.* https://github.com/fnc12/sqlite_orm