Replaying Turn-Based Games

Bradley Rosenfeld

Introduction

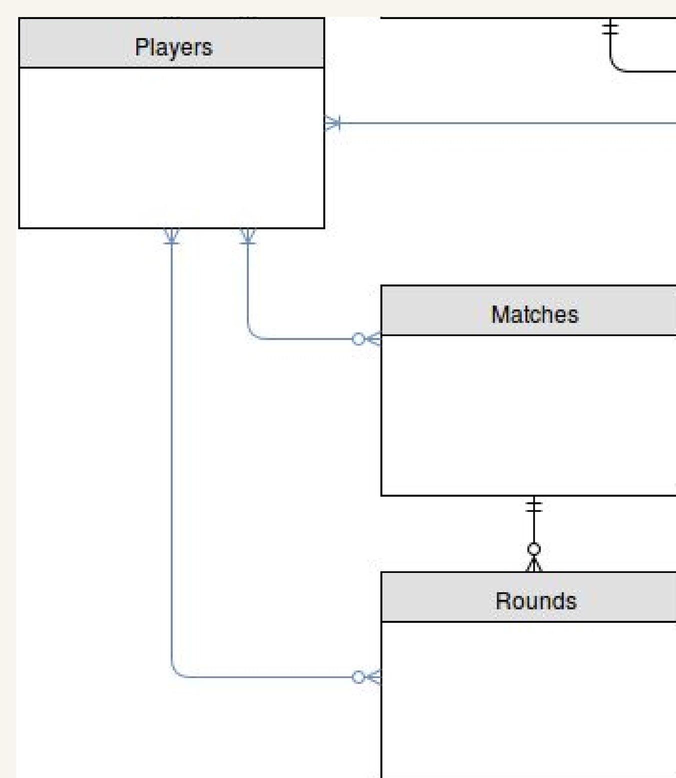
The Game Contest Server is an effective way for professors to manage game contests between student developed players. But it is an inherently closed system. Programs are run without human intervention and there isn't a way for students to see how their programs run as they execute. Visualizing gameplay during a round helps students learn how their players are behaving.

Replaying rounds is an essential part of the Game Contest Server. Over the course of the project, I collaborated with our team to design and implement this feature.

Normalizing Rounds

Initially, the system was designed with matches in mind. Professors create contests which can have tournaments. Each tournament has matches between the assigned players. Matches should have 1 or more repeated rounds between the same players. Referees might or might not support rounds. Checkers does not support rounds by default, while Battleship does. To get around this problem, the system was designed to "emulate" rounds by creating many, duplicate matches; or just one match if the referee supported rounds internally.

This design was flawed because we needed to be able to access individual rounds in order to show the replay of that round. Without a concept of a round in the database, the system had no way of figuring out which "matches" were actually related. Furthermore we couldn't effectively determine the results for an individual match. To solve this problem, we added a new rounds entity to the database design.



Manager TCP Protocol

Referees are executable files uploaded by professors that manage gameplay during competition matches. Previously, players and referees would often be run in the same executable which made cheating possible. The Game Contest Server intentionally spawns players and referees as separate processes for isolation. TCP sockets are used to handle communication between processes.

I designed a simple TCP protocol so that referees can report the status of the game as well as moves during the course of the round. The protocol is defined with *command:value* pairs that the game manager parses. Pipes are used as delimiters to separate fields in the value.

Example

```
port:2222
match:start
round:start|{}
move:description|movedata
gamestate:{}
round:end
roundresult:playername|result|score
roundresult:playername|result|score
match:end
matchresult:playername|result|roundswon
matchresult:playername|result|roundswon
```

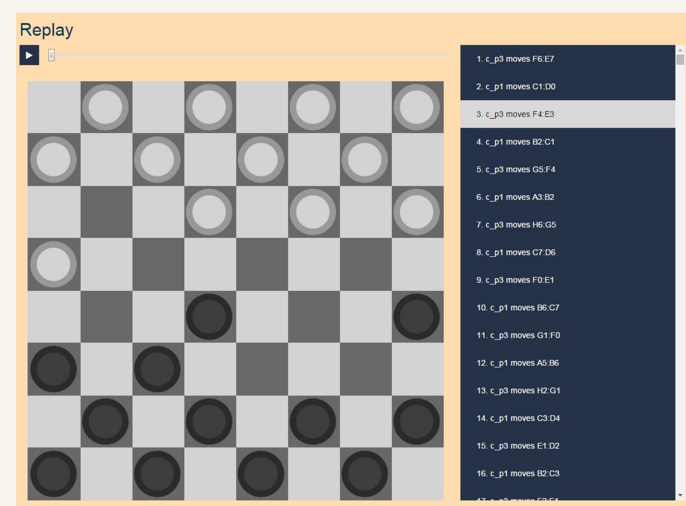
Logging

Saving the results of a round goes beyond wins or losses. We also need to have a record of moves. Referees do not have knowledge of our file system, so I needed to implement a universal way to handle logging of moves made during each round of a match. On successful completion of a match, the Ruby round runner exports the parsed game log as a JSON file. This game log is generated from messages sent over the TCP socket. A very simple API *match-logs/{matchID}/{roundID}.json* serves the data to our in-browser replay viewer via an AJAX request.

Replay API

The Game Contest Server supports an infinite number of game types. Because the referee executable actually defines the game logic, the system has no real sense of what a "game" is. Designing a replay viewer for the browser was a unique challenge as it needed to be flexible enough to display any game.

Games can have thousands of moves during each round. Saving a gamestate for every move is very inefficient, but attempting to calculate the gamestate from individual moves is performance intensive in the web browser. We use a hybrid model where the referee sends a gamestate every *N* moves. In the browser we calculate the game states between each delta move. This reduces the amount of storage required and makes calculating a gamestate trivial.



I made the choice to write the replay viewer in vanilla JavaScript. Using a framework like Angular or a library like jQuery would have sped up the initial development process, but any new developers would have needed to learn the framework. One of the advantages of using JavaScript's object prototype is the high level of extensibility for plugins.

Instructors upload a Replay Plugin along with with their referees. This plugin can modify the Replay object prototype in order to implement game specific parsing of the log file. We decided to use PIXI.js for a cross-browser 2D rendering engine.

Future Work

To encourage adoption of the system, Replay Plugins should be written for all the games used by the CSE Department (Risk, Battleship, Settlers of Catan, etc...)

Conclusion

I learned a lot about writing high performance JavaScript during this project. The unique requirements of the project constrained me to writing code that could be run thousands of times on many devices. Collaborating with my team to implement features forced me to learn to rely on other programmers for key features.

Department of Computer Science & Engineering
Taylor University, Upland IN 46989

January 2016