Cards with Friends: A Generic Card Game Engine

David Ding[*]
Dept. of Physics,
Mathematics, and Astronomy
ding@caltech.edu

Theresa Lee
Dept. of Computing and
Mathematical Sciences
theresa@caltech.edu

Mike Qian
Dept. of Computing and
Mathematical Sciences
mqian@caltech.edu

Benjamin Razon
Dept. of Computing and
Mathematical Sciences
brazon@caltech.edu

Alexander Wein
Dept. of Physics,
Mathematics, and Astronomy
awein@caltech.edu

ABSTRACT

In recent years, networked and online gaming have become ubiquitous; in particular, card games like poker and hearts have flourished and found dedicated communities online. However, any specific card game will have countless variations that are popular within certain circles or regions, and if such a variation that someone wants to play does not exist online, there is currently no fast and easy way for someone to create it. We aim to solve this problem by creating the application Cards with Friends, a generic card game engine. In the application, we provide users an API that they can use to write arbitrary card games, with functions they can use that are necessary for online play (like passing cards, bidding, and playing and receiving cards). This way, users would be able to implement a non-existing card game from scratch and play it online, or take the implementation of an existing game and make a minor tweak to it. In this paper, we discuss existing previous efforts for generic card game engines that did not succeed or persist over time. Then, we outline the flow of the creation of the application, as well as the tools we used in our implementation. Finally, we go into extensions of the application that we may create in the future.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software—*reusable libraries*; K.8.0 [**Personal Computing**]: General—*games*; K.8.1 [**Personal Computing**]: Application Packages—*freeware/shareware*

Keywords

Generic game playing, card games, AI, Flask, Socket.IO, Gevent, Heroku

[*]All group members attend the California Institute of Technology and are pursuing a Bachelor of Science in Computer Science.

1. INTRODUCTION

The concept of networked games has been around for a very long time. Back in 1975, people were already playing multiplayer videogames over a network such as STAR, OCEAN, and CAVE. Card games were also popularized on the computer and were easily extended to a multiplayer network format. These games can be played both from standalone applications (e.g. Internet Spades, which is bundled with the Windows operating system) and within the browser (e.g. card games available on Yahoo's website). Most of the popular card games can be played online in a variety of websites or standalone applications. However, there does not appear to be a single leading site in online card games, leading to both fragmentation in user base and a need for multiple systems to access all the games the users want. This fragmentation is evident in the dozens of websites providing multiplayer card games, not to mention the numerous standalone applications for specific games.

The difficulty in having a single platform for all card games largely stems from the vast number of card games. There are hundreds of different card games, each with many variations. Therefore, as a developer it is practically impossible to manually support all the potential games and variations on your own. However, as crowdsourcing has become more popular, it may be possible to crowdsource the rules implementation to players interested in specific games. This system is very attractive for supporting small variations on popular games since making small changes to existing rules should be quite simple and not time-consuming. Supporting completely new games may be difficult as programmatically entering all the rules to a card game might be too complicated for many users. It might be possible to automate this process by using natural language processing on human readable rules of the game.

Still, it is unclear what the best option for describing card games would be. As discussed earlier, having a simple language for defining rules would reduce the barrier to creating a new game. However, using formalized language for describing games, such as the GDL (Game Description Language) would allow the engine to work with the General Game Playing (GGP) project from Stanford and thus provide AI automatically for all supported games. Unfortunately, most of this work seems to have been done for board games rather than card games. Card games were not describable until the recent extension of GDL to GDL-II to account for unseen and random events. It is therefore unclear if a sufficiently powerful general playing program yet exists that supports card games.

Card games consist of a much smaller set of categories than board games. Examples of popular categories include trick-taking games (like bridge and hearts), comparison games (like poker and blackjack), shedding games (like Uno and rummy), and accumulation games (like Go Fish and war). By separating these categories it should be much easier for users to implement new games within a category. Ideally, the engine will be generic enough to support any game, but by creating individual category templates, there will be significantly more code reusability since every new game can inherit most of its basic structure from its parent category. Furthermore, defining a category is much more abstract and complex, and thus is better left to experienced programmers than to end users. This system would have the significant benefit of allowing a developer to focus on one category (e.g. trick-taking games) and only expand given enough time.

Therefore, the primary focus of our project was to implement a game engine that supports networked play of trick-taking card games within a web browser. Ideally, this was to consist of three general components:

- A frontend web application that displays game state and facilitates user interaction. Different interfaces provide the ability to begin or join games and to create or modify games and rules.
- A server that handles all gameplay by receiving information from users of the web application and making decisions about legal moves, state transitions, and scoring, which it relays to the frontend. The server contains all the games and rules supported by the engine. The server also stores user session information, historical gameplay data, and statistics for use by the AI (computer players) to improve heuristics and algorithms.
- An AI component with one or more difficulty settings capable of playing games against human and other computer-based players.

The first card game we attempted to support through this system was Spades, a relatively simple trick-taking game where the objective is to score points by winning enough tricks as determined by bidding before each hand. In this case, bidding is one example of a distinguishing game characteristic that must be supported by our engine, as not all trick-taking games involve bidding. Through the term of the project, we created a good abstract framework such that any card game can be implemented and played on the network.

2. LITERATURE REVIEW

The first aspect of a literature review was searching for previous projects with similar goals and scope. Volity [3] was described as a website which allowed users to play generic card games, but it was shelved in 2011 and none of the codebase is publicly available. Furthermore, it is unclear exactly what functionality this project managed to achieve as the site is now down and most mentions of it are relatively vague. Another project which attempts to allow users to play generic card games is gcge [4], but it appears that the lack of inheritable categories makes it difficult to reuse code. For instance, the project does not attempt to define broad game categories and instead gives users freedom to programatically create additional rules. It currently has only War and Fluxx implemented. We believe that the complexity involved with implementing new games combined with a lack of browser based user interface severely crippled the project. The developers also chose to omit an AI system from their scope. Most other programs were designed for generic collectible card games, which allow the cards themselves to be replaced with anything and do not enforce any rules. These may be useful to cherry-pick the presentation layer.

We then considered specific attempts to formalize game rules, and specifically card game rules. One natural option is to use Stanford's GDL which is designed to describe the state of a game and the possible transitions. Because of its highly formalized nature, it would likely be too cumbersome for end-users, but we could feasibly translate simpler syntax to this formalism. Because GDL is used for general game playing competitions, this would potentially allow us to use existing AI for any user created game with no development. The original GDL does not support the limited information and potentially random nature of card games, however GDL-II extends the machinery to handle these games. We have found entries of GDL-II players on the Dresden GGP server, but we have been unsuccessful in finding one that we could incorporate into our project.

An online search for generic card game languages was thoroughly unsuccessful. Perhaps unsurprisingly, no generic card game artificial intelligence seems to exist (probably since it would be difficult to construct without a generic card game language). Most existing card game AI systems seem to be tailored to specific card games and programmatically follow their creators' strategies. This approach clearly will not work for our general system.

Since we planned to first implement Spades and then continue refactoring and abstracting the code as additional features were needed, we considered using a similar strategy for implementing a potential AI. Although we did not end up implementing such an AI, we did a fair amount of research into the possibility. To gain insight into the feasibility of this approach, we reviewed an existing open source Spades AI program called SARC. The project provides a non-networked version of Spades for one person to play Spades and includes a UI. However, the code is very specifically tailored to Spades and is not properly broken down into modular components. The AI seems to be hard-coded with many simple heuristics. One thing that the AI relies on is understanding which cards can still be played. While we did not find this option in any literature or example problems, it seemed like a simple option would be relaxing the constraint that the computer plays with limited information. As long as the game experience is enjoyable and the AI is still beatable, it is possible to provide the computer with complete information, thus significantly simplifying the AI. This would allow us to use standard approaches such as alpha-beta pruning with minimax.

Some broad suggestions for the AI found online include using machine learning on previous games to learn strategy. Another option is to use Monte Carlo Tree Search to try to model the game space. Anecdotal evidence seems to indicate that this strategy would likely need some type of heuristics to prune particularly poor branches . It might also be possible to use more general automated planning and scheduling techniques such as Markov decision processes.

3. TIMELINE

3.1 Initial Timeline

Below is the initial rough timeline of the phases in the project and the estimated time of completion for each phase. The goal was to complete the project within 9 weeks, with the output described in Section 4 (End Product). Note that we planned for some of the phases to be done in parallel. Additionally, we realized that some phases may or may not have required more than the estimated time depending on the time spent in debugging.

0. *Version-Control System (1 day)*
   A version-control system such as git will need to be set up in order to allow multiple people to work on the same code.

1. *Designing Game Structure and Abstraction (1 week)*
   During this first phase, we need to create all the necessary framework for a generic game. This involves creating the abstract base classes in which specific games will extend from, including a CardGame class to represent an general card game, a Deck to represent the deck of cards (as well as a Card class), a Player, which can be split into a HumanPlayer and AIPlayer, and Phases, which describe the phases of the game. The exact details of the implementation as well as any extra specifications of a game must also be determined and constructed.

2. *Game Implementation with Spades (1-2 weeks)*
   This phase will be done in parallel with the server backend. After the initial phase with the abstract game classes, this phase will involve actual implementation of those abstract classes. In particular, we will implement the Spades card game extended from the CardGame class, and define the rules and phases of the game. This will be tested out on the server, once that is completed. Restructuring of the abstract classes may also be required if any issues occur during this implementation.

3. *Server Backend Part I - Game Playing (2-3 weeks)*
   This phase involves the implementation of the server backend that will be hosting the game. We need to design the process by which the server takes a game (an extension of the CardGame class) and plays it. An implementation of the rule system of the game must also be well-defined, so the server can understand valid moves, keep score, and knows when a game ends and who the winner is. This phase will involve a lot of testing with the specific game in the second phase (Game Implementation with Spades). This will primarily be tested via a command-line interface.

4. *Server Backend Part II - Game Creation (2-3 weeks)*
   Prior to this phase, some further research may be required. This phase requires heavy planning and the creation of a GDL to describe a game, its phases, its players, and its rules. Afterwards, a way to create a new game using the GDL, which can then be translated into a class, needs to be implemented.

5. *Middleware - Message Passing and Communication (1-2 weeks)*
   A well-defined method of communication between the the server backend and the web frontend must be designed. The structure of the message must be designed such that it contains the necessary details so the web interface will understand what to display to the user, and the server will understand what the user is doing on the web interface. The messages will be constructed and optimized in a way to reduce server load, although this may be done later if we are running out of time.

6. *Frontend Part I - Playing the Game (1-2 weeks)*
   This phase requires the creation of a simple web interface that an average user can interact with to play game. This will involve some designing on paper in order to figure out where to put the components of the game (such as the cards, other players, scores, etc.). The message system must also be fully refined in order for smooth message-passing between the server and the web interface. The logistics of the game must also be implemented, such as the flow of the game (how the user joins a game, starts a game, and exits a game), user feedback (valid and invalid moves), and other bonus features such as playback of games may also be established. If time becomes an issue here, we may instead borrow a web interface from a pre-established online card game, although that is probably not necessary.

7. *Frontend Part II - Creating a Game (1 week)*
   The web interface that allows the user to create new games with new rules as well as variations of standard games must be implemented. Also in this phase or in an earlier phase, the server and frontend needs to be transferred from a local server to a web server in order to test and simulate real conditions.

8. *AI (2-3 weeks)*
   The AI will be created to allow users to play against a computer if no one else is avaiable to play against. Details on how the AI will understand good game-playing will need to be defined (either via machine learning or manual training).

3.2 Actual Timeline

Below is the acutal timeline of our progress based on weekly scrums that we held throughout the course of the term.

1. *Week 1*
   We discussed what frameworks to use for web development. After considering some alternatives, we decided

to use Flask [5], which would integrate well with our backend (implemented in Python). A lot of our discussion formalised what 'trick-taking' games encompassed as well as how users would input their 'rules' into the application. David wrote an initial version of Hearts. We all looked into Flask to discuss what functionalities it provided as well as how it would work with our backend.

2. *Week 2*
   We created the Deck and Card classes as well as the abstract classes Game and TrickTakingGame. In addition, we discussed the preliminary design of card game web UI. Furthermore, we discussed the level of abstraction needed in games, and wrote functions accordingly. We also wrote a Hand class representing a player's active hand to support the TrickTakingGame class. We also familiarized ourselves with Flask.

3. *Week 3*
   We continued discussions on how best to implement server-side communication between the backend and frontend components of the software. We also finished most of the abstract framework for the backend and began looking into extensions of this code for public use.

   We also researched HTML5 WebSockets and decided that gevent-socketio (a Python implementation of WebSockets) should be easy to use. We got some chat server code working using this package and we decided that we would be able to get all the necessary server push notifications for the card game working with gevent-socketio. In addition, we edited and added more functions into the Game and TrickTakingGame classes, recalling and continuing discussions about the level of abstraction needed in various functions.

   We also created a Player class to keep track of a person's name, score, hand, and other attributes, and wrote functions accordingly to make accessing these attributes easy. Furthermore, we continued changing our implementation of Hearts to reflect new developments (like the creation of the Player class), which we thought would be pretty close to what the custom game code would look like in the end. Additionally, we edited the Card and Deck classes to be more consistent with the other classes we had written.

4. *Week 4*
   We built a prototype webpage for the game-playing interface using Flask and Socket.IO:

   - Two players can connect to a server.
   - They are each dealt a hard-coded hand of cards.
   - They then take turns playing cards until their hands are empty.
   - Cards can be displayed, clicked on, and removed from the screen.

   We also added more code development structure, including internal object representations for debugging purposes. We finished rewriting the Hearts code. Furthermore, we tried exercising some of the code manually. We also introduced the notion of a thread pool or "event manager" for scheduling tasks. Some game actions needed to be run simultaneously for different players (e.g., card passing), and obviously the game engine needs to support running multiple games concurrently at some point. We implemented card passing e.g., for use with the Hearts "trading" phase.

   We added basic bidding functionality to the Player class, which was the last major component of a trick taking game that we had not yet covered. We also created an implementation of Spades, a basic trick taking game with bidding, in order to get another sample game created as well as think about how much abstraction is needed with bidding. Furthermore, we wrote a rough draft of the API, which will be essential for other people to create their own games.

   We also made a "highest card" dummy trick-taking game in order to test interaction with the frontend in the coming week.

5. *Week 5*
   We implemented the GetPlay function in the backend to begin the process of connecting the backend and the frontend. While doing this we realized that the backend doesn't have a system to notify the frontend, so we also decided on a subscription model to handle this. We also made some pages on the website to explain the project and have a login system that persists sockets across pages.

   In addition, we worked on establishing communication between the frontend and backend so that a game can be played using the graphical interface.

   We also implemented a basic server app using JavaScript and Node.js to test how different users would log into the same "table" for card games. We also implemented a rudimentary representation of cards played onto the table (not images, only text); and when players play their cards the card images disappear from their hand. Furthermore, we worked on sequencing the order of the players so that the user will be able to see the correct order of players; for this week it was simply implemented as a numerical order.

   We added support for 3 and 5 player games to the Hearts module (simple variants). We also created functions for finding a card by properties (suit, number, etc.) within a deck or hand. At this point in time, we still needed to make sure that there's a unique identifier we can stick to when referring to objects. We also planned for a robust subscription model to handle message-passing.

   We created a command-line interface on which to test and play games we have written, by editing the Player class. The interface currently supported playing cards and bidding, but it did not support passing, since passing is often a simultaneous action between all players, and it was unclear to us how to do this in a terminal. We also used the command-line interface to debug the demo game we were testing the frontend with, as well as Spades and Hearts (with no passing). This way, once the frontend and backend interaction had solidified, we could test gameplay on the frontend without worrying about correctness of the backend code.

6. *Week 6*
   We refactored the demo of the basic card game into flask and expanded the demo of the basic card game to see how rearrangement of the placement of the user worked in relation to other players. We also added some elements to the frontend of the table for games (i.e. message boxes, player attributes such as tricks taken, money, and current score).

   We also implemented a message-passing paradigm so that the backend can request cards/bids/etc. from the frontend application as well as notify it of state changes (e.g., cards were added to the player's hand). We also fixed trick scoring in Hearts and Spades, which hadn't taken into account ace high. Additionally, we began work on a main module that will import all games and maintain the mappings of all the objects.

   We implemented frontend functions that the game engine backend calls through message-passing. We also changed bidding functions to make sure they took arbitrary strings instead of integers. In addition, we started work on Bridge, which seems to be far more complicated than any game we had so far. We also attempted to get a game working with the frontend graphical interface, but failed.

   Furthermore, we successfully implemented the entire game-joining interface. Players could now login, host games, delete games and join games. Additionally, no two players are permitted to use the same username. When a game is full, the host has the option to start the game (although the "start game" button did not yet work).

7. *Week 7*
   This week we were able to get a demo working.

   We worked on how to handle redirecting users to the game table once enough players have joined and the host chooses to start the game. We also discussed some frontend UI design choices as we started to flush out the remaining missing components in our frontend/backend integration.

   We finished implementing all middleware stubs except for getting bids. We also added support for displaying card images in the trick taking area. We also created a system for initializing the game_table with initial values for all the player information.

   We also started learning about Heroku, a cloud platform supporting Python on which we can host our app.

8. *Week 8*
   We implemented getting bids from players and started implementing getting multi-card plays from players. While doing this, we ran into the issue of how to validate the combination of cards. Ultimately, we decided that the backend should do validation. In addition, we let people view the last trick by adding a delay before clearing the trick.

   We also solved a circular import issue that we were experiencing while importing server components. We also began work on a GameManager to keep track of existing games and score history. Some of this code would need to be merged with the existing frontend/server code.

   In terms of deployment, we also tried to get our application working with Heroku. The application managed to get past the login screen, but got stuck when trying to create a new room. The issue seemed to be related to how gunicorn interacts with the other packages we were using.

9. *Week 9*
   We finished implementing multi-card plays, passing cards, a combo box for bids, and error messages shown on invalid card clicks. We also fixed some remaining bugs in the game list. Additionally, we changed card dealing so that cards are added to player's hands in sorted order, in order to facilitate more natural display of cards to users in the frontend.

   We also figured out how to use Doxygen to automatically generate documentation in a nice format based on the documentation in our backend code and briefly looked into other documentation options such as Epydoc and Sphinx.

   We also implemented drop-down lists on the room list page that allows the host to choose a game (e.g. Hearts, Spades, Bridge) and a number of players. Each game supports a different set of values for the number of players.

10. *Week 10*
    We allowed users to upload their own code files, finished support for selecting different games at start time, implemented end game dialog, and extensively tested everything.

    Additionally, we got the code uploaded to Heroku so that Cards with Friends can be hosted remotely.

    Finally , we filled in any existing holes in the documentation and finalized a format for the documentation.

4. END PRODUCT

4.1 Initial Goal

The final product, ideally, was to be a website or web application, where a person would be able to use the following features:

- Search for an existing card game using its common name or nicknames, or using a subset of the rules of the game.
- Add a new card game if it does not yet exist on the website, via a web interface where the user would be able to specify the rules. For example, one would need to specify such aspects of a game as the cards to use, the number of players, the criteria for ending a round, points received per card, and other guidelines. Alternatively, if someone wants to create a minor variant of a game (add "house rules"), then they would be able to make a copy of the rules of the existing game, and then simply edit a few parameters. The newly created game would be made available to everyone.
- Start a game with a private group of friends, or in a public room. In the case of a private group of friends, one person would create a unique ID or URL (similar to the current implementation of Google+ Hangouts),

which others would then be able to join given the ID. If a user wants to play with anybody, they would specify the game they want to play, which would then be visible to everyone else in the room, who could then join. The game would start once there are enough players.

- Play through a specified card game with other people, where scoring and legal moves are automatically determined by the rules (gameplay will be similar to that of Microsoft Hearts).
- Play through a game with one or more computer players, which will have multiple difficulty levels.

In addition to these main features, we realized it was possible to expand in many directions if time permitted (which we realized was admittedly a bit unlikely, even at the very beginning). For example, if we made our implementation of a card game sufficiently abstract, we might have also tried to implement other general categories of the games such as shedding or accumulation games. With extra time, we might also have tried to make our AI increasingly sophisticated. At the most basic level, the AI would simply play random cards. More advanced AIs would include some aspect of machine learning, like looking at previous games played and analyzing which moves of a given game resulted in the most points for a round. To do this, we would also need some standardized method of storing previous games. Another potential direction we considered would be to improve the look of the user interface. At the very least, a user's cards would be displayed in text form, but we could also add actual pictures to represent cards, or animate cards as they go from a user's hand to the center (again, similar to Microsoft Hearts).

In the worst case, we planned to have the general abstract structure to play a card game, along with perhaps one implementation of a card game like Spades (hard-coded in, as opposed to the desired method of inputting the rules). Additionally, we wanted to have the web interface that would display cards and games, but non-functional interaction with the server, so that it would not really be possible to play an actual game online or create one.

4.2 Actual Product

After working on this project for a term we were able to create a website, hosted on Heroku, where multiple players can connect and play card games together. Details of the frontend and backend follow.

4.2.1 Frontend

1. *Login Screen*
   Each player starts at the login screen. Here they must enter a nickname that will uniquely identify them during this particular session. No two players are allowed to have the same nickname. In the future we could actually give each player an account with a password that they log in to every time they come back to play. For now, however, nicknames only last for a single session.

2. *Room List*
   After choosing a nickname, players are redirected to the room game list. Here, players can create and join "rooms". A room eventually becomes a game when enough players join. On the room list page, players can choose to create a new room. In doing so they select the game type and number of players from a dropdown list. We currently have support for Hearts with 3-5 players and Spades with 4 players. We of course hope to expand this as users submit rules for their own games. Once enough players have joined a room, the host has the option to start the game.

3. *Game Table*
   When the host starts a game, all players in the room are redirected to the game table page. Here, the card game begins. Figure 1 depicts the GUI that players use on the game table page. This GUI supports features such as passing and bidding that may or may not be used by any given game. A log window displays messages to players instructing them what they need to do next in the game. Players can play cards simply by clicking on them. The cards played in the current trick are displayed in the center of the screen. Statistics such as number of tricks taken and total score are displayed for all players in the game.

4. *Game Submission*
   When a user decides to implement their own version of a game, they write a Python file desribing their game. In order for this file to be integrated into the server it must be uploaded. Currently a rudimentary system is in place to upload the file from the users computer and generate a submission file with important information such as the author's name and email. After the file is submitted, it must be manually inspected and tested. If there are any problems the author will be emailed, otherwise it will be moved into the games folder and added to the game creation list. In the future, we hope to automate some or all of this process. This will necessitate the creation of a sandbox environment for security purposes but is clearly worthwhile as users will receive instant gratification when creating games while also providing a reasonable testing suite for development. Automated uploading will also accelerate the development cycle of these users.

4.2.2 Backend

The first components of the backend were the Card and Deck classes, which provide information and images for a standard deck of cards, as well as methods for interacting with the cards. We also have a Player class, in which we store a player's name, as well as their hand, their score, and the cards they have taken. These classes are used in the Game class, which has general methods to start and reset a game. This class has the subclass TrickTakingGame, which has methods more specific to trick-taking games. When users create games, they inherit functions from the TrickTakingGame class. Specific details about classes and functions can be found in the documentation of the application on the application website.

4.3 Tools Used

We used a couple of tools to make Cards with Friends. Figure 2 illustrates the various components of the application and how they interact.

1. *Flask*
   Flask is a microframework for Python based on Werkzeug and Jinja2. This allows Flask to be a very lightweight and extensible web framework. Flask is able to handle routing and serving webpages while using a sophisticated templating system to create customizable pages based on a Python backend. This was an ideal system to integrate the backend game logic, which was also written in Python, with the frontend which is displayed within the browser.

2. *Heroku*
   Heroku is a cloud application platform which we used to deploy our web application. We used Heroku to host our web application on a website rather than hosting it locally on a personal machine. Heroku is not ideal for hosting our application because it causes it to run very slowly. One reason for this is that it does not fully support WebSocket and falls back to XHR long-polling. Ideally we will eventually host the project elsewhere but Heroku was the easiest option for now.

3. *Socket.IO*
   Socket.IO is a JavaScript library for real-time web applications. It includes a client-side library and a server-side library for Node.js. Socket.IO connects the browser to the server, which was written in Python.

4. *gevent*
   gevent is a concurrency library for Python based on greenlet and libevent. gevent fundamentally modifies Python's threading to create a cooperative threading environment. This means that all context switches must be performed cooperatively instead of forced by some external thread manager. This is very efficient for applications with network and user latency, such as our WebSockets. Unfortunately, the learning curve was non-trivial and led to some tricky race conditions.

5. FUTURE WORK

5.1 Automated Submission Process

Currently, to submit a game, users upload their file, which is moved to a staging area, and reviewed by a developer before it is made available to everyone. An automated process to put users' games online would greatly streamline the game creation process and make it more convenient for all parties involved. Relatedly, we would also want to implement a sandbox in which users would be able to test their games before submission. Clearly, for all of this, we would also have an automated code review process so that users would not be able to execute arbitrary code.

5.2 Social Integration

To make the gameplay experience more human, we could add in social components to our application. Natural paths we could take include adding video chat to ongoing games, so that players can see each other; however, this would require a significant upgrade to the web hosting service we use. We could also add forums to the website, so that users can discuss such topics as variations on games and implementations of said variations.

5.3 Improved Interface

The current interface is authentically spartan. The initial landing page greets you with unformatted text and jarring hyperlinks. The game playing interface also trades aesthetics for functionality. A push towards beautiful design would definitely help the product feel much more elegant and polished. Functionally, the user interface is missing some small features such as simultaneous plays, chat, and clearing unneeded clutter such as money in non-monetary games and stale error messages.

5.4 AI Player

Something else that could be added to Cards with Friends would be the implementation of a rudimentary AI or computer player that could play against humans or other AI players. An advantage of having an AI player would be that there would not be a required number of human players to play card games that require a specific number of players.

6. CONCLUSIONS

In conclusion, card games encompass a varied landscape of different games. The wide range of possibilities makes card games appeal to many different people. Unfortunately, this diversity also makes providing support for the entire range of possibilities a daunting task. By providing users with a public API, supporting new games can be effectively crowd sourced. We explored this possibility by creating a minimal viable product to better understand the feasibility and desirability of this approach. To do so, we limited the scope of the project to only support trick-taking games because the several broad categories of card games can easily be developed separately and then integrated. Furthermore, developing in an overly generic setting can be stifling, so starting with trick-taking games helped motivate our decisions. The product we made was surprisingly compelling. Supporting new games, while not trivial, was also not overly cumbersome. Clearly there are still mountains of low-hanging fruit which we detailed in the future work section. However, the initial version shows enough promise that further exploration seems worthwhile. Based on the research conducted on prior work, it appears that this approach may actually be novel and capable of unifying all card games in a single website.

7. ACKNOWLEDGMENTS

We would like to thank our faculty advisor, Professor Steven Low, for guiding us throughout the course of the project and for providing us with helpful feedback on our progress. We would also like to thank Professor Mani Chandy and Professor Adam Wierman for co-hosting the class.

8. REFERENCES

[1] Browne, Cameron, Powle, Edward, et al. *A Survey of Monte Carlo Tree Search Methods.* IEEE Transactions on Computational Intelligence and AI in Games, vol. 4, no. 1, Mar. 2012. <http://www.doc.ic.ac.uk/~sgc/papers/browne_ieee12.pdf>.

[2] *Card Game.* Wikipedia. 12 Mar. 2013. Web. 15 Mar. 2013. <http://en.wikipedia.org/wiki/Card_game>.

[3] *Volity.* <www.volity.net>.

[4] *General Card Game Engine.* http://code.google.com/p/gcge/.

[5] *Flask (A Python Microframework).* <http://flask.pocoo.org>.

[6] *Game Description Language.* Wikipedia. 11 Jan. 2012. Web. 15 Mar. 2013. <http://en.wikipedia.org/wiki/Game_Description_Language>.

[7] *General Game Playing.* Wikipedia. 4 Dec. 2012. Web. 13 Mar. 2013. <http://en.wikipedia.org/wiki/General_game_playing>.

[8] Love, Nathaniel, Hinrichs, Timothy, et al. *General Game Playing: Game Description Language Specification.* Stanford University: Stanford Logic Group, 2008. <http://games.stanford.edu/language/spec/gdl_spec_2008_03.pdf>. Print.

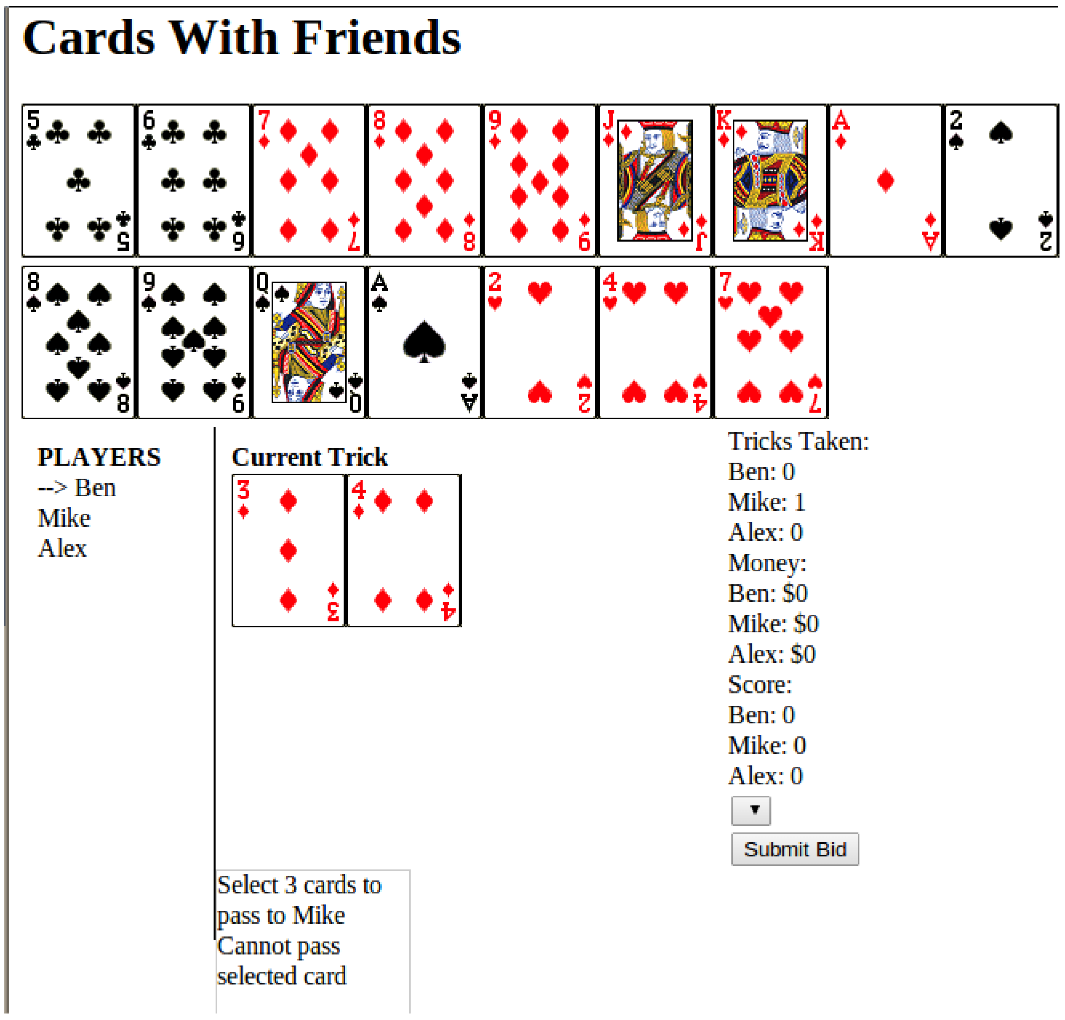
[9] *Markov Decision Process.* Wikipedia. 15 Mar. 2013. Web. 15 Mar. 2013. <http://en.wikipedia.org/wiki/Markov_decision_process>.

[10] McLeod, John. *Card Games and Tile Games From Around the World.* Card Game Rules. Pagat.com. 1 Mar. 2013. Web. 4 Mar. 2013. <http://www.pagat.com/>.

[11] *Multiplayer Video Game.* Wikipedia. 14 Mar. 2013. Web. 15 Mar. 2013. <http://en.wikipedia.org/wiki/Multiplayer_video_game>.

[12] *Spades.* Wikipedia. 8 Mar. 2013. Web. 15 Mar. 2013. <http://en.wikipedia.org/wiki/Spades>.

[13] Tielscher, Michael. *A General Game Description Language for Incomplete Information Games.* University of South Wales: School of Computer Science and Engineering, 2010. <http://www.cse.unsw.edu.au/~mit/Papers/AAAI10a.pdf>. Print.


Cards With Friends

PLAYERS
--> Ben
Mike
Alex

Current Trick

Tricks Taken:
Ben: 0
Mike: 1
Alex: 0
Money:
Ben: $0
Mike: $0
Alex: $0
Score:
Ben: 0
Mike: 0
Alex: 0

Select 3 cards to pass to Mike
Cannot pass selected card

Figure 1: GUI for playing generic card games. Players can click on cards to play them. Features such as passing and bidding are supported. A textbox displays relevant instructions, warnings, or other messages to the user.

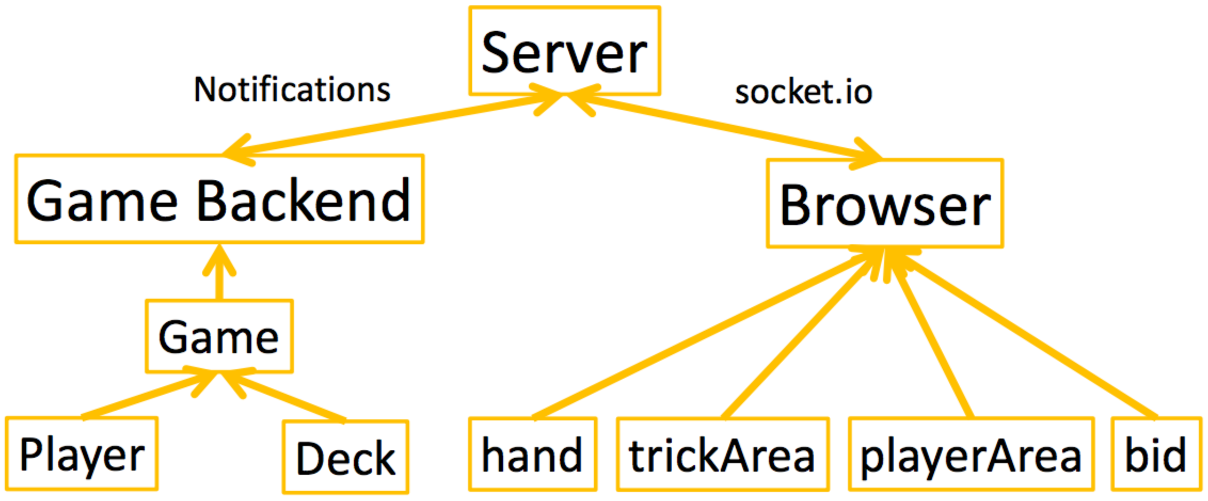


Figure 2: Schematic diagram of how the application works.