Final Project: Splendor

Lev Akabas ***la286*** Nathaniel Kaplan ***nak74***
Jad Rahbany ***jr978*** Sean Viswanathan ***sv287***

System Description

Core Vision:
We are going to build an implementation of the board game *Splendor* by the Space Cowboys.

Key Features:

- A playable version of *Splendor* that implements all of the rules of the existing game, and some "house rules" options
- Multi-player "pass-and-play" functionality, with varying numbers of players
- An AI can take the place of one or more players
- A way to track what cards a player can currently buy
- A simple GUI that tracks bought cards, gems taken, and points

Narrative description:

We are going to be creating an implementation of *Splendor* similar to the existing app for the game. The game will be playable by 2-4 players, as the board game itself is, and any number of players can be replaced by an AI (there must be at least one human player, however). The AI will have a variety of different strategies that it can employ and will change that strategy if necessary over the course of a game. The game will track necessary gameplay elements (gems, cards bought, and points) and will also highlight in some way what cards a player can buy with the gems that player currently has.

Link to *Splendor* rules:
http://www.spacecowboys.fr/img/games/splendor/details/rules/Rules_Splendor_US.pdf

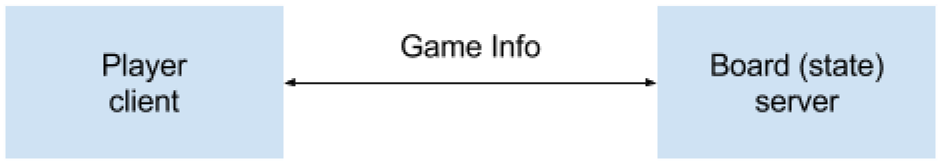
Brief overview of potential AI strategies:

- Standard Engine/Nobles strategy
    - Buy lots of small cards, accelerate into buying larger cards, focus on getting 1-2 nobles over the course of the game
- High-tier strategy
    - Focus on getting a few high-point cards (tier 3), particularly the 4-5 point cards that cost 7-10 gems respectively, by collecting a lot of gold through reserving
- Mid-tier strategy
    - Buy several small cards and then buy a lot of tier 2 cards, sometimes get a noble
- Blocking strategy

- ○ Block what your opponent(s) are trying to do as much as possible over the course of the game, forcing them to reserve most cards they want

Architecture

This game follows a Client-Server architecture where the server is the board which contains the state of the game, and the client is the player which would request information from the board.



Some of the main components of the state server are the number of players, current player, information about the cards in play, list of nobles, and gem information. All that combined would allow the player to make a play. So during each player's turn, the player would request game info from the state server.

The player client is going to be represented as having a list of cards, points, discounts, reserved cards and gems (detailed in the data section). The server/board will have all 3 tiers of cards, all players and their information, data on the nobles and information on all cards, and gems. The communication will take place by the client (player) wanting to make a request to the server (board) with some move (game info/ data to be processed). Then, the server will process the request on the client and modify and update the data received from the client, and returned a new accurate data representation of the client with the move they had. This process repeats for every player until it has determined a winner.

System Design

Modules:

- Play
- Ai
- Main
- Card
- Graphic

Card:

Card contains all of the types necessary for Splendor. It does not have to be implemented further.

Play:

Play handles all of the data for the game--it updates the state as necessary after the player makes a move. It also handles events such as the game ending. The functions in the mli file must be implemented.

Main:

Main is the module that will be used to run the actual game. The function in the mli file must be implemented.

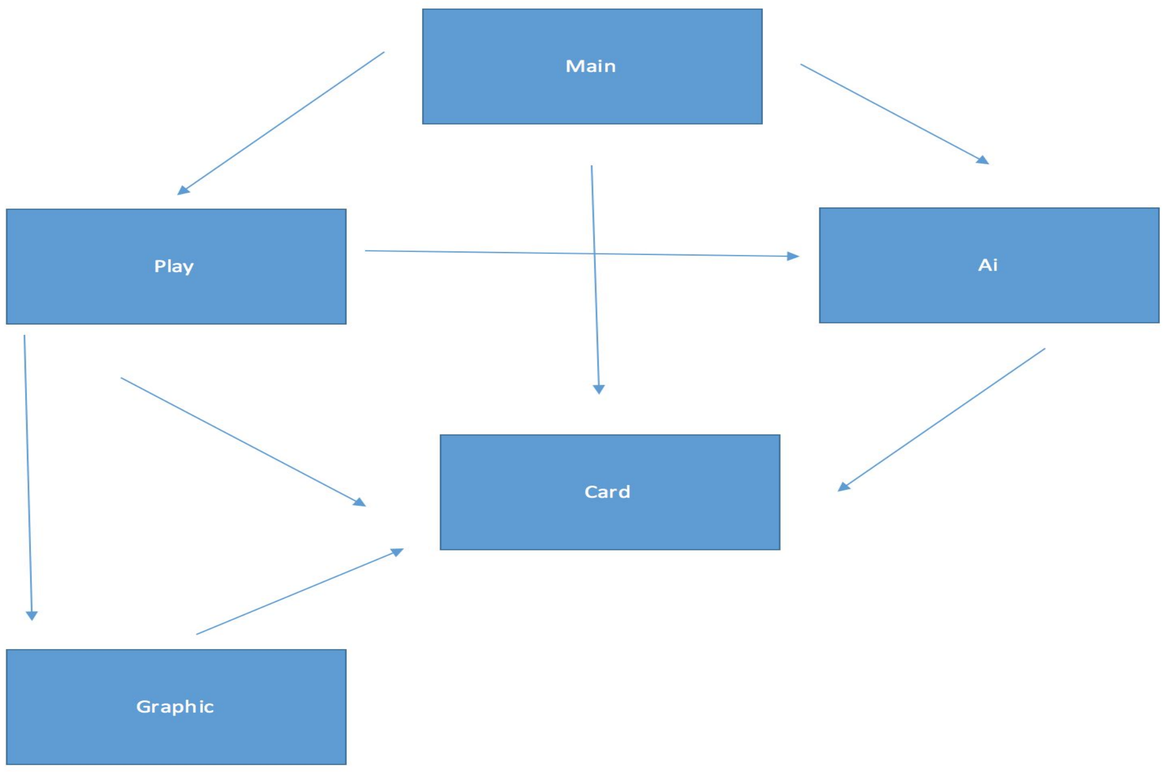
Ai:

Ai will handle the AI decision making process. The function in the mli file must be implemented.

Graphic:

Graphic will handles everything involving the graphical interface, including displaying visually all the information contained in the game's current state and functions allowing the user to play the game by clicking the mouse and typing keys. The functions in the mli file must be implemented.

MDD:



Data

- State
    - Player list (each player contains the following data):
        - Gems held (record of colors)
        - Discounts obtained (essentially the same thing as cards bought, record of color)
        - Reserved cards (list of cards)
        - The number of cards a player has bought (used for tiebreaker)
        - Points
    - The current player
    - The tier 1 deck (a list of cards)
    - The tier 2 deck (a list of cards)
    - The tier 3 deck (a list of cards)
    - A list of the available nobles
    - The gems available for taking, and their quantities
    - The number of gem piles remaining, used for checking illegal moves

- Cards
    - Discount provided (color variant)
    - Cost (record of colors)
    - Points

This is all of the data that the game needs to maintain to run properly. Every type of data--state, cards, nobles, and gems held/available, and card costs, *are all records* due to the quantity of data each must hold. All other data is going to be handled when a player takes their turn and does not need to be stored. A player move is a variant, the colors of gems they want to take is a variant, and AI actions are determined by a function, determine_move, which is entirely based on the current state.

The system will update the state record after a player takes a turn (and current_player will shift to the next player to indicate that the previous player's turn has ended). The game will end when a player's point total meets or exceeds 15.

External Dependencies

We will be using the OCaml Graphics module to implement our graphical interface. There are many useful features in this module that we will use in our program. The five major colors in *Splendor* are all predefined in the Graphics module and we will use those to display gems and cards, by drawing circles and rectangles, respectively, of those colors. The module also includes functions for text drawing, which we will certainly use to display instructions to the user, but we may end up using for other aspects of the graphical interface. For example, a player's hand can be displayed through images, or simply through five numbers, one of each color, indicating how many cards of each color they have. The event variant in the module will be used to identify mouse clicks, after which we will use the status record to determine the location of the click, and key presses, after which we will use the status record to determine which key was pressed.

Testing Plan

- Our team will tackle testing in a piece-wise incremental fashion, so that each component and module will work properly on it's own before being integrated into the system. These Individual components are as follows :
    a. A player's hand which, from the data representation described above, will need to be properly tested so that we meet modifying the state of the hand and properly update records keeping track of a player's hand. Then we will want to test edge cases in which a player tries to have more than ten gems, enforcing that a player cannot have more than the maximum number of cards available to take from, when we imitate a player taking a card we properly update the state of the player so that it reflects their interaction with a card (i.e buying the card from spending gems, and the number of gems and the discounts they own are properly updated).
    b. The next thing we would want to test in isolation is the state of the board that we represent, so here we want to keep track of the gems available to take from (and selecting on the gems such that people can't take gems when there are 0 left) and the maximum number of gems is not exceeded for each color. Additionally, we want to test the number of cards that are displayed and always have four cards of each tier on the board, unless a deck is empty and testing that there are less than four cards available after the deck is empty and cards are taken. Making sure nobles are made available at certain points, when "mock players" want to draw them and they are permanently removed from the board.
    c. Both of these high level modules will need to be thoroughly tested with OUnit test harness that will challenge the scalability, correctness and implementations of the modules. By using OUnit we can easily map out basic to complex to edge cases that we want to test. The testing harness' will allow us to single out where we may be having an issue in their individual implementations and behaviors.
    d. Next we want to focus on testing the interactions between this client-server architecture, and making sure that the player can have proper interactions with the deck, and that both the players state and game state are updated properly and do not surpass the limitations (or edge cases) of the game rules. Since we have understood that each module works in their singularity and isolation we can ensure applications of interactions are the problem if we run into any, during this phase.
    e. A final testing harness will allow us to combine the interactions of the two modules and ensuring that players cannot break the rules and the game state is not broken.

f. We can all hold each other accountable by writing OUnit tests for the functions and helper functions that we write in each module so that when the module itself is complete, it will have tests written and the module in isolation can work. Moving up from this when the modules interact, we can have a better idea of how to debug.