Final Project: Splendor

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

System Description

Core Vision:
We built 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
- Multi-player "pass-and-play" functionality, with varying numbers of players
- An AI can take the place of one or more player
- A simple GUI that tracks bought cards, gems taken, and points

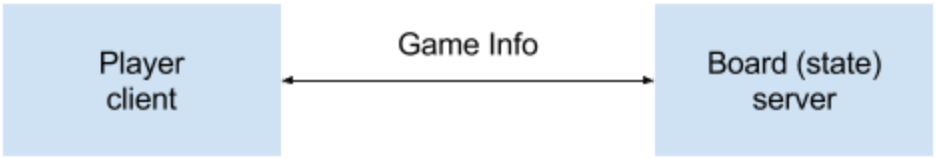
Narrative description:

We created an implementation of *Splendor* similar to the existing app for the game. The game is playable by 2-4 players, as the board game itself is, and any number of players can be replaced by an AI. The AI has a strategy that it employs, which involves changing its goals depending on the state of the board (in particular, taking into account the dominant colors on the board and the cost/point value of cards) and whether or not it is early or late in the game. The game tracks necessary gameplay elements (gems, cards bought, and points).

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

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, information about the players, information about the cards in play, list of nobles, and gem information. All that combined allows the player to make a play. So during each player's turn, the player requests game info from the state server.

The player client is represented as having a list of cards, points, discounts, reserved cards, gems (detailed in the data section), and a few other attributes. The server/board has all 3 tiers of cards, all players and their information, information on the nobles, and information about gems available. The communication takes 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 processes the request of the client and modifies and updates the data received from the client, and returns a new accurate data representation taking into account the move. 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.

Play:

Play handles all of the data for the game -- it initializes all fields of the game and updates the state as necessary after any player makes a move.

Main:

Main is the module that will be used to run the actual game.

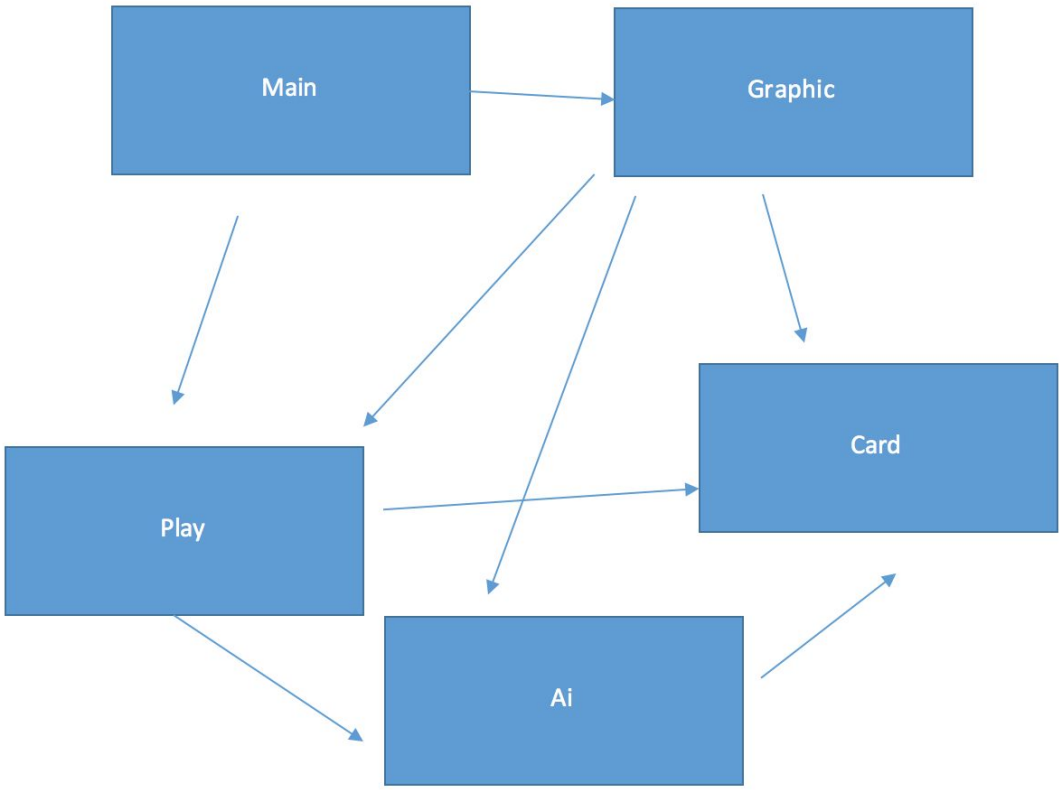
Ai:

Ai will handle the AI decision making process.

Graphic:

Graphic 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 or typing keys. It also handles the REPL loop and events such as the game ending.

MDD:



Data

- State
  - Player list (each player contains the following data):
    - Name
    - 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
        - Player type (human or AI)
        - Gold (number of gold gems held)
    - The tier 1 deck (a list of cards)
    - The tier 2 deck (a list of cards)
    - The tier 3 deck (a list of cards)
    - The tier 1 cards available
    - The tier 2 cards available
    - The tier 3 cards available
    - 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
    - The number of turns taken
    - The number of gold gems available
- 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 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 updates the state record after a player takes a turn (and the order of the player list rotates to indicate that the previous player's turn has ended). The game ends 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.

The graphics module was heavily utilized as our only external dependency since we needed to use drawing of the main colors available to draw the cards on the table, the gems available on the table, and the hands for each of the players (showing how many gems, the discounts and reserved cards of players hands).

Testing Plan

Actual Testing Model:

- Our team stuck with the piece-wise incremental testing fashion and made sure that each independent component did work with each other however, very early on the process of developing a test-harness, we realized how ineffective and inefficient building this harness in the beginning was. Since we needed to constantly update the mli files and types of our state, and that our state was not mutable, we had to redo the state by hand, we decided that utop and manual testing was definitely the best practice. Although we have a trade off for no automated tests, the number of edge cases are so minimum in each possible situation that our piece wise testing fit in perfectly with manual testing in utop. We had each components (AI, Graphics, Play) all test the functions in their files on their own to ensure they worked. Then as we began to integrate graphics and play, solving the bugs and issues that arose from that combination and likewise when we integrated the AI.

Old Testing Model:

- 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.

Division of Labor

Nathaniel Kaplan worked primarily on the Ai--he completed the entire Ai module on his own. He also completed the Makefile functionality and helped with overall implementation and design when working out the last pieces of the game--in particular, the logic connecting Play, Graphic, and Ai, as well as discarding gems and determining the player that won. He also set up main and did extensive final bug testing for the last version (5 hours +) to catch major final issues, as well as cleaning up stray code, fixing mli files, and following all other necessary steps to prepare the project for final submission. He worked for about 45 hours total.

Lev Akabas worked primarily on Play, which controlled the major aspects of gameplay. He worked on all of the functions that change the state for each type of move, functions that handle players earning nobles, and the overarching play function, which calls the appropriate move functions and passes on error messages to display in the GUI. He helped debug and test aspects of Graphic and the interaction between Play and Graphic. He worked for about 35 hours total.

Sean Viswanathan worked on some of the helper functions for play, i.e the buy_card and handling of whether players are updated properly when they take gems, updating the state in that scenario and then logic for calculating how many golds need to be used, when and why they will be used. Additionally he

worked on a large part of the graphics including the interactions of processing where users click, buying, reserving, cancelling and updating the UI to reflect user actions. He handled the ending of the game as well as displaying the winners of the game and other required rules. He also integrated graphics with play so that the UI and game logic smoothly worked together. He worked for about 35 hours total.

Jad Rahbany worked on graphics to set up the interface and process the state such that gems, cards, player information is all shown nicely on the table. He also worked on mapping where users click on a screen and if that is a relevant move and how that is processed. His part focused on presenting the board and making sure that it was robust enough to update every time the state updated with a move, and was set up so that it would be easy to move forward with game play. He worked for about 35 hours total.