EECS314 Project Report: Checkers

Matt McKee (mwm67), Patrick Landis (pal25)

05/02/2013

Project Description

This project involves building a checkers game in MIPS. Specifically this project will use SPIM for the backend and Python to generate a GUI and input to SPIM. This project will also attempt to implement two versions of the game – a two player version and a one player version against an AI.

Major Challenges

Most of the problems were with the MIPS code. The biggest problem is that debugging in SPIM is a time consuming process whereas the Python part of the project had a rather fast turnaround time. Thus, while the first version of the MIPS code was finished before the python code, functionality testing was time consuming, especially for more complicated scenarios, such as when pieces jump each other.

Another major challenge was in implementing the checker movements themselves. Each move had two main stages: validating the move as a legal move, and updating the board data to reflect the move that had been made. Validation proved to be a more complicated issue than expected: each move had to be checked to insure that the piece was player controlled, that the destination was vacant, and that the destination was adjacent to the current position. With our board positions numbered straight though, it took some mathematical pattern manipulation to insure that players couldn't move off of the left side of the board and "wrap around" to the right.

Furthermore, validating a jump required that the destination space be two squares away in a certain direction, checking the middle square to make sure that it was both occupied and that the occupying piece was opponent controlled, and that a jump, like a move, didn't wrap around the sides or top or bottom of the board.

Key Components

1. State Machine: this state machine will be the main method used to "run" a game. It will contain states to cover necessary game portions, such as "initialization", "player input", "player move" and "AI turn", and will handle information retrieved from the "board" data structure (explained below) about when to "conclude" a game (such as loss, forfeit, or program termination). A player's turn will be broken down into several states, to handle any input received from the player. The AI turn state will be much simpler, as it will simply need to communicate with the AI. A "check" state will be used to watch for stalemates.

2. Data Structures: Data for the same is contained in three structures: a "haspiece" structure, a "color" structure, and a "rank" structure. Early on, the design decision was made that, since there are 32 valid squares of movement in checkers, and MIPS supports mainly 32 bit registers, the three structures could simply be 32 bit bit-arrays operating in parallel. Both the board and these bit arrays were 0-indexed, so it was easy to link the "zeroth" space on the board to the "zeroth", or least significant bit, position of each array. This meant that accessing the data for a specific space on the board was as simple as left shift each of the three registers by the "value" of the board position. This access was easy, and creating masks to alter specific squares on the board likewise became almost trivial.

3. Checkers AI: The checkers AI will implement an Alpha-Beta search algorithm which will evaluate moves based on metrics that include taking a piece, protecting pieces, if it can be moved onto one of the sides of the board, etc...

4. Python Frontend: The Python frontend will receive data by piping stdin/stdout to/from the Python. The chess program will be able to send and receive messages to Python which will take actions based upon those messages to update the game board and send key presses to the MIPS backend.

Component Integation

Frontend

The Python frontend was initially built separate from the MIPS backend. In addition it was developed in stages to facilitate easy debugging. Initially the screen built because it would facilitate testing the rest of the frontend. Once the screen was built a working prototype of the game was setup with checker pieces that were able to move but player controlled. Once a working version was complete, work was started on the message passing system between Python and MIPS. Python has to be able to parse messages and then update the state and game board. Finally work begain on integrating Python and MIPS together.

Backend

The MIPS backend was designed around taking simple message from the Phython front end, interpreting those messages either as moves or as program commands, and then reporting a message string back to Python. The input that MIPS expects is simply an integer: either a space that is a part of a movement command from a player, or commands coded to mean that a player is ending his turn or restarting the game. Thus, integration with Python could be kept mostly minimal: Python would make sure that the move involved only the 32 legal squares of the board, and that the piece was player controlled. From there, MIPS handles all validation of movement and updating of the gameboard data, which it passes back to Python as one long message printout. Python then parses the message, and obtains either an error code, or a validation code followed by the data needed to redraw the board.

Specific Contributions

Matt's Contributions

1. MIPS memory location assignment to handle storage of critical elements.

2. MIPS code to parse an input message, and react properly to commands to either change game state or make a move.

3. MIPS code to organize the program flow during a turn, alternating between input, validation, and output steps, and responding to illegal input.

4. MIPS code to organize the program flow between player turns, and to signal the AI to play for player two in a one player game, and to set and reset the board data structures at the start of a new game.

5. MIPS validation of movement commands, insuring each move was properly checked for origin, destination, player control.

6. Further MIPS validation of jump moves, insuring that each jump contained a square with an opponent controlled piece between the origin and destination.

7. Assisted with writing and debugging code to update the data structures containing the information on the board state, specifically for capturing and removing jumped pieces.

Patrick's Contributions

1. Python message parser which is responsible for reading messages from MIPS and determining the actions to take based on those messages.
2. Implemented the Python message communication to MIPS which involved taking data and packing that dta into a message and then piping that message to MIPS.
3. Python game screen responsible for all the graphical functionality of the game.
4. Python game board and sprites which are responsible for populating the screen with objects that users can view.
5. Python game state which is maintained in sync with from the MIPS game state.
6. MIPS draw board functionality responsible for reading the data structures for MIPS's representation of the game.
7. MIPS board message communication responsible for sending the data from the MIPS draw board to Python.
8. MIPS update board functionality which is responsible for updating the data structures that store the game information after a move is made.