CS3110 Hnefatafl Design Doc

Elizabeth VanDenburgh (eav38) Joseph Dwyer (jmd456) Taeer Bar-Yam (tb442)

November, 12 2015

1 Hnefatafl Game Dynamics

The game is played on an 11×11 checkered board with two uneven sides. 12 white pieces are positioned around a white "king" at the center of the board, with 24 black pieces arranged on the edges surrounding them. See 1. There are three main mechanics involved in the game:

1. **Movement**
   Pieces move like rooks in chess (horizontally or vertically). Only the king has a limited motion of three squares at a time.

2. **Capturing**
   When a piece is flanked by two pieces of the opposing color, that piece is "captured" and is removed from the board. This flanking must be done actively by the offense (i.e. the center piece can move between the others without being captured).
   The king cannot participate in flanking an opponent. In the case of the king, it must be flanked on all sides to be considered "captured."

3. **Winning**
   Black wins when the king is captured, or the king's only escape is to move back to the center square of the board.
   White wins when the king is moved to one of the side squares of the board.

See: `https://en.wikipedia.org/wiki/Tafl_games#Hnefatafl`

2 Interface:

Ultimately we would like to implement the interface as a GUI using openGL, and allow click-and-drag motion commands. Initially, we will implement a text-based interface that uses chess-like notation for movement and output a ASCII grid to show positions of pieces. This interface will likely remain available as an option
This will be playable against another human player, or against an AI.

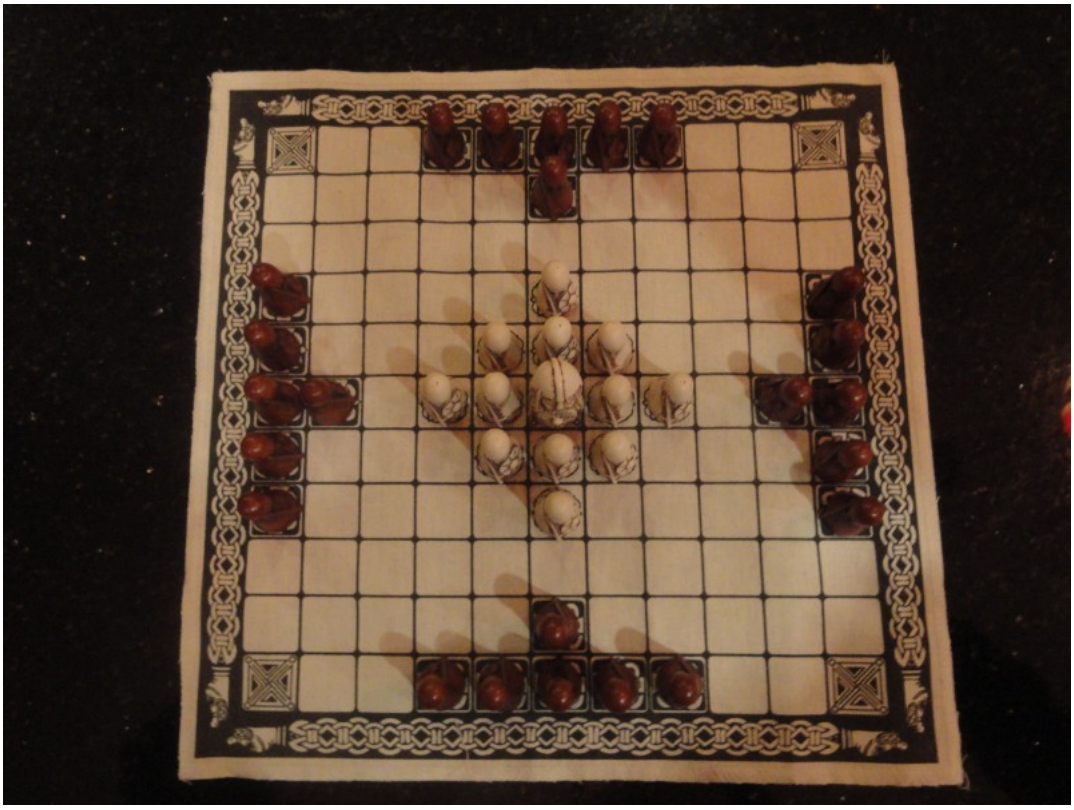

Figure 1: Hnefatafl starting board position. Source: https://medium.com/war-is-boring/you-have-to-play-this-1-600-year-old-viking-war-game-cef088ae4e2d#.10kof827g

3 Architecture

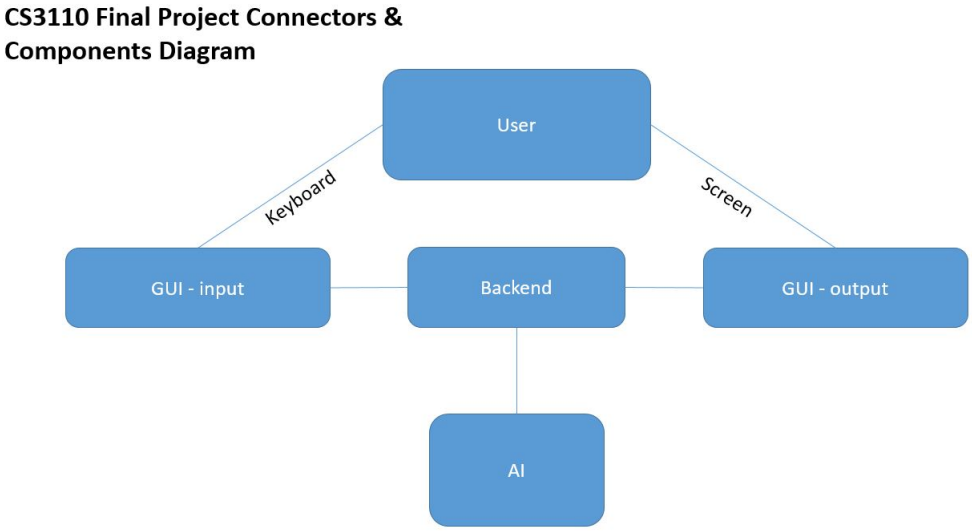

Figure 2: Connector Component Diagram of the pipe and filter architecture for our project. Unmarked lines represent connections via function calls (APIs).

The backend forms a pipe-and-filter architecture that applies an attempted move to the board state given the rules of the game. The backend and AI components form a server-client like connection where the backend communicates with the AI code to determine what the best move is. In this case the AI is the server and the backend is the client. See 3 for the connector and component diagram.

4 System Design

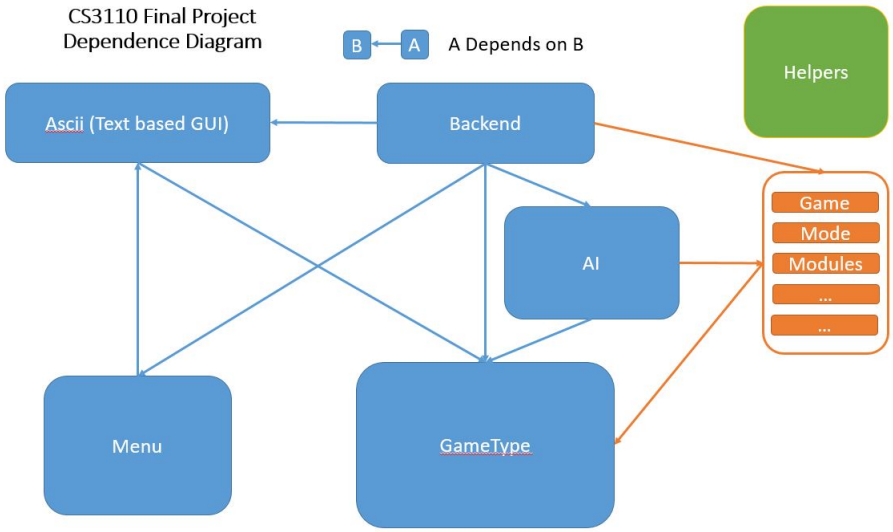

Figure 3: Diagram of dependencies between modules in our project. The helpers module (green) is depended on by everything. The game mode modules (orange) may or may not end up in the final version

4.1 GUI

Two options of GUI modules: Text-based and graphical.
The purpose of this module is to enable interaction between the user and the backend. The GUI does none of the game or board computations. It only has knowledge of the screen state, and what the backend tells it to print. It **is** in charge of interpreting piece movement by the user. That is, it does not report every keystroke, mouseclick, etc. the user makes, only what move the user would like to make.

4.2 Backend

Backend is in charge of the following tasks:

- Initialization of menu and storage of configuration options
- Keeps track of the baord and player state
- Validates moves from the GUI
- Calculates if a move resulted in a piece's capture
- Calculates if the game is over and who won
- Calculates if the game is unwinable (hopefully)
- Prompts the GUI for the user's next move
- Prompts the AI for it's next move

4.3 Menu

In the Menu module is what text to display in the start of game menus. It tells the GUI which menus to display and the GUI responds with which selection the user made. The menu composes a game configuration based on the choices and passes that back to the Backend.
Menu will also present a "Do you want to quit?" type message on quit.

4.4 AI

If playing as one person, the AI module will be passed the current board state by the Backend. Using this, it will create a decision tree and decide on an "intelligent" move to execute.
We can put the AI calls in a separate thread to allow more computation for a smarter AI.

Ideally we would like to create either a general AI for all game modes or a stronger AI for each individual game mode. Based on the amount of time that we have, and the complexity of building such AI, we will decide which AI implementation to use.

4.5 Helpers

Basic functions that either we feel should have been included in OCaml or most modules will need are included in Helpers.

4.6 GameType

GameType includes basic types for the board, pieces on the board, and other game-state related objects. It also includes small helper functions that relate to those types. (e.g. piece_at for the piece at a location on the board)

4.7 GameModes?

We have two ideas for how to implement variants on the game mode (e.g. differences in win conditions, valid movements, etc.). One involves dynamically loading one of a set of module that will contain functions for these aspects of the game.
The alternative would be to load preferences out of a plain text file.

Pros/Cons:

\+ Doesn't require us to write text parsing code

\+ More flexible (can run arbitrary code)

\- Less secure (can run arbitrary code)

\- Dynamically loading modules is awkward

5 Module Design

See attached .mli files.

6 Data

Structures include:

- Piece types (variant)
- Coordinates (type synonym for int * int)
- Action variant (what actions the user can perform)
- Game configurations
- Board with dimensions and list of piece locations

State data:

- Which player's turn it is
- Current board state
- Which game configuration was selected

The GUI also stores the current selected piece (the piece the user would like to move) until the user selects a location to move it to.

Everything is passed through function calls. There is no imperative or global data.

7 External Libraries

- Termbox for creating an ascii board
- OpenGL for creating a more graphical gui
- Potentially Dynlink for dynamically linking game modes

8 Testing Plan

- Write unit tests for state of the board after certain moves
  - Valid moves
  - Invalid moves
  - Win conditions
  - Piece removal conditions
  - All of these, in various game modes
- Watch the AI play against itself
- Get play testers

9 Stretch Goals:

- Hnefatafl is an old game, and there are many versions of it that exist. Time permitting, we would like to implement many of these variants, working off of a general game core.
- Detect if the game has become unwinnable.
- Collecting statistics on the game (who won, how many turns it took, etc)
- Storing the state of the game at every turn, and enabling turn by turn playback