# CS3110 Hnefatafl Design Doc

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

November, 12 2015

## 1 Structure of the Implementation

The Hnefatafl game that we created is a historic Viking game with several different play modes. The board game was created by the Vikings and is meant to simulate the defense and attack of two uneven sides. The defenders goal is to protect their king and move the king to specific locations on the board. The attacks want to capture the king. Because this is a historic game, there are several different versions (we call them modes). The modes differ in their rule sets and initial configurations. They are each described in detail below. To increase the complexity of the game, we implemented TWO/THREE different game modes. We allow the player to decide, at runtime, what version on the game they would like to play. The rules and intial setup of the game are then loaded into the main program and the player can begin.

## 2 Hnefatafl Game Mode Descriptions

### 2.1 CS3110 Hnefatafl

We created this version of the game after our inital study of the rules. After further study we realized that this game mode is not as accurate historically as Fetlar. For this reason, we are leaving this mode of the game as the default (because it is the easiest to explain and understand) but naming it CS3110 Hnefatafl (because it does not perfectly match any true rule set).
The game is played on an $9 \times 9$ checkered board with two uneven sides. 8 white pieces are positioned around a white "king" at the center of the board creating a cross, with 16 black pieces arranged on the edges surrounding them.

The black team moves first.

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."

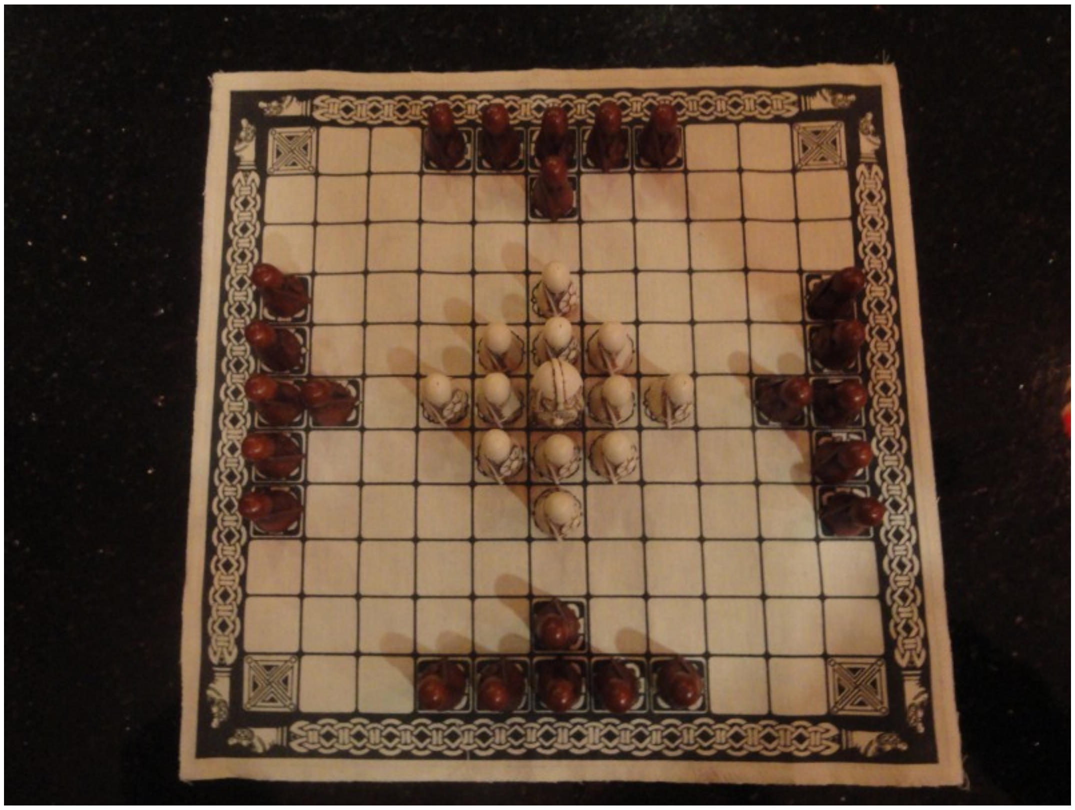

Figure 1: Hnefatafl starting board position. Source: `https://medium.com/war-is-boring/you-have-to-play-this-1-600-year-old-viking-war-game-cef088ae4e2d#.10kof827g`

3. **Winning**
   Black wins when the king is captured.
   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`

## 2.2 Fetlar Hnefatafl

This is a true version of the game. The setup is an 11 × 11 checkered board with 12 white pieces positioned around the white "king" at the center and 24 black pieces around the edges of the board. The black team moves first.

See 2.2.

Main mechanics of Fetlar:

1. **Restricted Squares**
   In this version of the game, there are restricted squares. These are the throne (center square of the board) and the four corner squares. These squares limit the movement of pawns and can be involved in captures.

2. **Movement**
   Pieces move like rooks in chess (horizontally or vertically). There is no limit on the number of

pieces that the king can move. Pawns cannot land on a restricted square. They can, however, pass through the throne if it is unoccupied by the white king.

3. **Capturing**
   Capturing in Fetlar is very similar to capturing in our default game. Capturing of pawns is done by actively flanking them (sandwiching the captured piece between two opponents). Capturing of the king is done by actively flanking him by surrounding him on all four sides. In this version, however, the king can participate in captures. There are also the restricted squares. This means that the 5 restricted squares can replace one of the attacking pieces in a flanking move. Even with the restricted squares, black cannot capture the king when he is against a wall. The throne square will always be hostile to black pieces and it will be hostile to white pieces when it is empty. This means that the king can be captured if he sits immediately next to the throne and is surrounded on the other 3 sides by black pieces.

4. **Winning**
   The black team wins by capturing the king. The white team wins by having the king escape. He does this by landing on a corner square.

### 2.3 Copenhagan Hnefatafl

This is a harder form of Hnefatafl, but it has similar features to Fetlar. The thing that makes this game more difficult is additional ways that pieces can be captured.

Main mechanics of Copenhagan:

1. **Restricted Squares**
   This is the same idea as Fetlar.

2. **Movement**
   The same rules apply as in Fetlar.

3. **Capturing** The generics of capturing are the same as Fetlar. There is an additional way of capturing called a shieldwall. This is accomplished by bracketing the opposing team against a wall. For this move to be complete, the offensive team must be placed one in front of each defensive player, backing them into a wall. The line of defensive pieces must also be pegged in on the bookends with an offensive piece. A corner square can also act as a bookend piece. You can capture as many pieces as you want by doing this move. This includes capture of the king.

4. **Winning** Once again, the king must move to the corner squares to escape. White wins when this occurs. The black team wins when the king is captured.

## 3 Interface:

There are four possible "GUI"s (actually just interfaces, but we named everything incorrectly. Ah well) that the user can decide between at runtime: A command-line interface, a text-based interface, a 2D interface, and a 3D interface. described below. Every GUI includes the command that q and esc are quit. This means quitting up to a higher level menu. If you quit a game, there is no way to re-enter it. It is important to use the q and esc keys when quitting a GUI because they are pre-programmed to exit gracefully. Users that do not use these keys (e.g. by closing the window) will see errors in their terminal. It turns out, windowing libraries don't have an easy way of handling windows closing.

### 3.1 Command-line

We call this GUI "CLI". It is in the file named "CLI.ml". I am a firm believer that every application should have a command line interface so that it is easier to interact with the application programmatically.

### 3.2 Text-Based

We call this GUI "ascii". It is in the file named "ascii.ml". The controls for this GUI are the arrow buttons (or hjkl) to move. The space bar selects a piece. The location of the cursor is shown with a blue, highlighted square. This GUI is implemented using Termbox which takes over the terminal. If your terminal is screwed up after an error in the program, this is why. You'll have to restart the terminal emulator.

### 3.3 2-Dimensional

We call this GUI "2D". It is in the file named "graphical.ml". The controls for this GUI are clicking and dragging with the mouse. A selected piece moves with the mouse. This GUI is implemented using the built-in OCaml graphics library and opens a new window.

### 3.4 3-Dimensional

We call this GUI "3D". It is in the file named "threeD.ml". The controls for this GUI are arrow keys (or hjkl) to move the cursor. The cursor is shown with a lit tile. Selection of a piece is done with the space bar, and a ghost version of the piece you selected will follow the cursor. The ghost-piece will be tinged green if the move you are making is valid, and red otherwise. (unfortunately, the way we implemented the code means that you are able to select a piece when it is not that piece's turn, and the "valid move indicator" will show that the move is valid. You can also select blank squares. The reason for this decision was that was didn't want the GUI thinking too much about what was going on in the game, since it is just a GUI. We also wanted to limit communication between GUI and program logic, so the GUI doesn't communicate anything when a piece is **selected**. If I were redesigning the code, I would relax one of these constraints.) This GUI is implemented using OpenGL and it opens a new window.

## 4 Artificial Intelligence

The AI computes, with no data structure, the optimal move at each turn. The optimal move is computed using a utility function. The utility function is dependent on the number of captured pieces and the distance of the white king to the win condition. White and black actually use the same utility function. This helps the black team not only capture pieces, but oppose the movement of the white king. Relating the number of pieces captured to the movement of the king was one design consideration. If you value the kings movement too highly, the white team AI will rush to the win condition with little consideration for the welfare of its pieces. If you value the movement too little, the white AI turtles and only defends that king without actively trying to win. We determined the valuation ratio between a lost piece and movement of the king by having AI with randomly generated constants play against each other and take the best value. This is an informal version of a neural-network.

## 5 Architecture

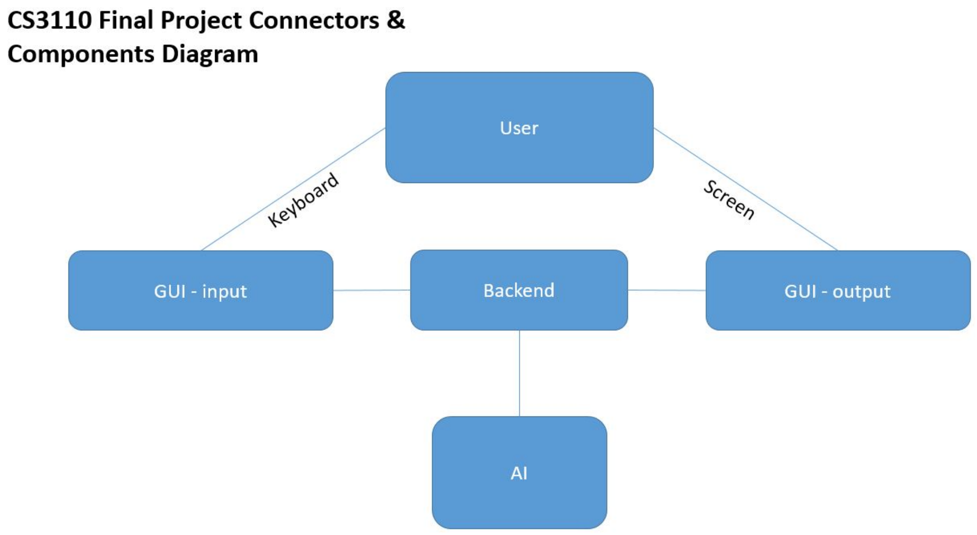

Figure 2: Connector Component Diagram of the pipe and filter architecture for our project. Unmarked lines represent connections via function calls (APIs).

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 5 for the connector and component diagram.

## 6 System Design

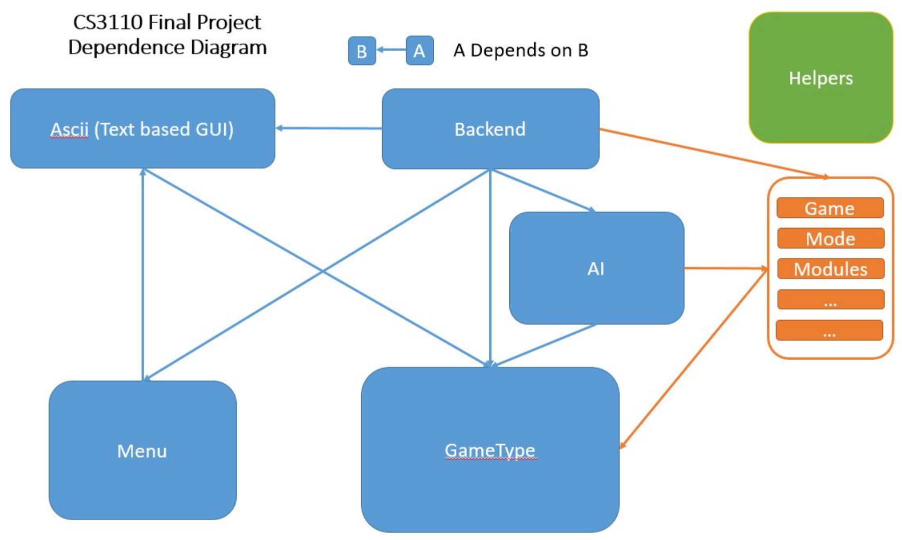

Figure 3: Diagram of dependencies between modules in our project. The helpers module (green) is depended on by everything. The game mode modules (orange) may or may not end up in the final version

### 6.1 Main

Main 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

### 6.2 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.

### 6.3 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.

### 6.4 Helpers

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

### 6.5 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)

## 7 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.

## 8 External Libraries

- Termbox for creating an ascii board
- lablgl for creating a more graphical gui
- OcamlSdl for windowing

## 9 Testing Plan

- Watch the AI play against itself
- Get play testers

## 10 Division of Labor

- Joseph
  - Artificial Intelligence
    - AI optimization
    - Neural networks
    - Heuristic function
    - Testing
  - Blender piece design
- Taeer
  - Fetlar.ml
    - Initial board setup
  - Complete work on the four GUIs
  - Integration of Menus
  - Loading of game modes, AI, and GUIs from menu selections
  - 3D models and export script from blender
  - Testing
- Elizabeth
  - Testing
  - Prepared final Design Document
  - Game Modes
    - Winning conditions
    - Capture specifics
    - Valid move specifics
    - Board initializations