FACULTY OF SCIENCE PALACKÝ UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

SOFTWARE PROJECT

kMancala

May 2011

Dalibor Hořínek
Applied Computer Science, 3^{rd} grade

Abstract

This project is about a realization of a board game Kalah, also known as Mancala. It's a count-and-capture like game for two players. The goal of this project is to correctly and fully implement game rules, imitation of real board of the game, possibility of combination human and computer players, logging every move user/computer makes and restore board into any position in the log. Also project tries to follow rules and standards of KDE Games family where kMancala may belong in future.

Contents

1. Introduction

KMancala was created as a school software project at Palacky University, Department of Computer of Science. There was a specification of requirements the application has to follow, such as history of moves, changing of players in any point of game, save and load a game. Game runs on KDE and primarily runs on GNU/Linux operating system, also other platforms where KDE can be run are supported, but not tested. This manual is a brief reference of mayor parts used in this application. It does not describe all methods, because only the important ones are metioned. For full documentation see the source code. Application is under GPLv2 licence.
Project homepage: `http://www.horinek.net/projects/kmancala.html`

2. Game architecture

The game consists of two logical parts. First part is an internal part which provides general classes and methods to handle general rules, management of a game and players, implementation of a board. Second part is a GUI part which provides graphical representations of current board state, player settings, also takes care about interaction with user.

2.1. Internals

This part consists of four classes which are **ai**, **board**, **controller** and **player**.

2.1.1. ai

Artificial Intelligence (ai) provides interface to generate legitime moves of computer player, also plays a role in suggestion of best move. At this time AI is handled by minimax[1] algorithm which is a recursive algorithm for choosing the next move at this two-player game. It goes trought a game-state tree and evaluates leafs of this tree. Leafs inicates some states of the game state. A value is computed to indicate how good it would be for a player to make that move.

int ai::getMove(int player_id, board *p_board)
Returns next move for player *player_id* at board given in *p_board*.

int ai::_minimax(int player_id, board *copy, int depth)
Implementation of minimax algorithm. Returns evaluated value.

int ai::_evalPosition(int player_id, board *copy)
Returns evaluation of board state **copy* for player *player_id*.

2.1.2. board

Implementation of game board. There is an arrau of 14 integers for purpose of storing number of stones in each pit. Valid move is defined as an integer in inerval ¡0,5¿. Also it defines some basic macros described below.

```
#define TREASURY_PLAYER_1 6
#define TREASURY_PLAYER_2 13
#define PLAYER_1_FIRST_GAP 0
#define PLAYER_2_FIRST_GAP 7
```

These macros defines which pits belong to what player and player's main pits (treasuries).

```
#define STATUS_EXTRA_MOVE 1
#define STATUS_CAPTURE -1
#define STATUS_OK 0
```

Macros which starts by **STATUS** word describes possible states which can happen after a move is done.

STATUS_EXTRA_MOVE Indicates that player on turn has an extra turn. Also the *_extraMove* variable is set to ***true***.

STATUS_CAPTURE Indicates capture of opponent's stones.

STATUS_OK Indicates standard move was done successfuly.

```
#define FIELDS 14
#define BEANS 4
```

Macro **BEANS** defines number of stones used in new game in each pit excluding main pits (these are set to 0), **FIELDS** defines number of pits.

int board::getFieldBeans(int i)
Returns number of stones in a pit identified by *i*.

int board::nextField(int field)
Returns next pit in a direction of a game play.

int board::doMove(int player_id, int move)
Modificates the board by doing the move *move* for player *player_id* and returns one of possible states defined by **STATUS_*** macros.

void board::moveToTreasuries(void)
Redistributes unused stones into main pits when game is over.

2.1.3. controller

This 3rd part of the application handles management of players, board and game play. Manages player on turn, provides methods for checking game state, validation of the move accorging game rules, identify winner when game is over.

bool controller::checkMove(int player_id, board *b, int move)
Returns ***true*** when player *player_id* can do the move *move* on board *b*, otherwise returns ***false***.

bool controller::gameOn(board *b)
Returns ***true*** when game is not over according state of board *b*, otherwise ***false***.

void controller::setCurrentPlayer(int i)
Sets player on turn to player *i*.

void controller::switchPlayer(void)
Switches players on turn.

int controller::winner(board *b)
Returns a player who wins according board state *b*.

2.1.4. player

Last internal part handles player settings, such as name, type of player (human/computer) and in case of computer player also AI level (or difficulty level). Basically this class simply stores information into it's slots and provides methods for setting and getting these values.

2.2. Graphical User Interface

GUI is created using Qt Framework and its extension by KDE. Main part of the application is Qt Graphics View Framework[2] which provides item-based approach to model-view programming. Two major parts of Graphics View Framework are Scene and View, the Scene is an interface for managing items, propagating items and so on. The View visualizes the content of the Scene. Main window of the application is created using KXmlGuiWindow[3]. It provides a mechanism for managing a layout of toolbar and menu using an XML file (appui.rc file).

2.2.1. kmancala

Main class of the application, inherits from KxmlGuiWindow and provides interaction of toolbars and menu for user.

void kMancala::newGame()
Pops up a player settings dialog and sets up a new game.

void kMancala::openGame(void)
Pops up a open game dialog, parses selected file and loads a game into saved state.

void kMancala::saveGame(void)
Pops up a save game dialog, creates an XML file and stores whole history of game.

void kMancala::changePlayers(void)
Pops up a change players dialog which is the same as in case of new game and changes current player settings.

void kMancala::playPause(void)
Pauses the game. If game is already paused, then it unpauses the game.

2.2.2. kmancalamain

This class is main/central widget of the game, this is the part under the toolbar. It contains the Graphics View Widget and History Widget.

void kMancalaMain::_init(player **players)
Initializes the game and sets players.

void kMancalaMain::_startGame(board *b, int playerOnMove)
Initialize the scene, game settings and starts the game.

void kMancalaMain::startGame(player **players)
Starts new game using _init and _startGame methods, but also initializes history nad board.

void kMancalaMain::loadGame(QList¡kMancalaHistoryItem*¿ history)
Initialize history using history QList which is an list of moves (usually loaded from a file).

void kMancalaMain::updatePlayers(player *p[2)]
Updates players if they were changed, emits signal to scene to update players.

void kMancalaMain::_playerMove(int currentPlayer, int move)
Handles a move *move* of the player *currentPlayer*. Actually do the move and checks if the game is not over, switches players and emits signals in case of extra move and capture.

void kMancalaMain::_manageToolbarItems()
Handles Undo and Redo actions, disables/enables when it's appropriate.

void kMancalaMain::playerHumanMove(int i)
Handles move by user, checks if human user doesn't play AI's move and if the move is valid.

void kMancalaMain::playerAIMove()
Waits 800ms and calls startPlayerAIMove() method for handling AI move.

void kMancalaMain::startPlayerAIMove()
Handles AI move using **ai** class in internal part to gain best move for the user. Sets the level of AI before gaining the move.

void kMancalaMain::historyShow(int currentRow)
Restores a board to a state defined in history on row *currentRow*, also resets players configuration into state as they were in the history and checks if there was a game over in the history.

void kMancalaMain::historyUndo(void)
Undos a game into one step back if possible accorging the history.

void kMancalaMain::historyRedo(void)
Redos a game into one step forward if possible according the history.

void kMancalaMain::pauseGame(void)
Pauses a game.

void kMancalaMain::unPauseGame(void)
Unpauses the game. If there was a history item selected, it continues from that state and removes all following history items.

void kMancalaMain::hint(void)
Provides a hint for human player. Uses **ai** class as **startPlayerAIMove()** method.

2.2.3. kmancalaboard

This is the View Widget, it inherits from QGraphicsView and creates the scene and populates it by graphical items, transforms mouse click events to actions. It uses **kmancalarenderer** class for rendering pixmaps from a SVG file, which is something as a theme file. More information is below in **kmancalarenderer** class description (Page 8, section 2.2.4.).

void kMancalaBoard::init(board *b, player **p, int currentPlayer)
Initializes the game using board *b*, players *p* and sets player on move to *currentPlayer*.

void kMancalaBoard::drawBoard()
Draws pits and main pits, score boxes.

QPointF _countPosition(QPointF p, double angle)
Counts new position of point *p* moved by *angle* in degrees.

void kMancalaBoard::drawBeans(board* b)
Draws current state of each pit (gap). It means draws the stones (beans) in each pit according state of the board as in *b*.

void kMancalaBoard::drawPlayers(player **p)
Draws the player names and player type icons according player settings in *p*.

void kMancalaBoard::drawHilightPlayer(int player, QColor color)
Hilights the player on turn.

void kMancalaBoard::drawHilightGap(int i, QColor color)
Hilights suggested pit (gap).

void kMancalaBoard::drawBackground(QPainter *painter, const QRectF &rect)
Draws a background of the scene.

void kMancalaBoard::updateBoard(board* b)
Slot used for updating board.

void kMancalaBoard::updatePlayers(player **p, int currentPlayer)
Slot used for updating players.

void kMancalaBoard::playerSwitched(player *p)
Slot used when players were switched.

void kMancalaBoard::gameOver(player *p)
Slot used when game is over. It draws a game over text and the winner, which is passed in *p* argument.

void kMancalaBoard::pause()
Slot used when game is paused. Draws pause text.

void kMancalaBoard::play()
Slot used when game is unpaused. Removes the pause item.

void kMancalaBoard::hint(int id, int currentPlayer)
Slot used when hint is suggested.

2.2.4. kmancalarenderer

This class is used for rendering pixmaps from a SVGZ File (GZip Compressed SVG File) located at ***appdata*/themes/default.svgz**. Each object in the file has it's id and this is used by kmancalarenderer for selecting the object. It inherits from QSvgRenderer class.

QPixmap kMancalaRenderer::renderPixmap(QString name, const QSize size)
Renders pixmap from object identified by name. Resulting size will be *size.*

2.2.5. kmancalahistory

This class stores and generally handles history of moves. It inherits from QListWidget.

void kMancalaHistory::init(player **p, board *b)
Initialize history on beggining of the game. Inserts "start game" history item.

void kMancalaHistory::init(QList¡kMancalaHistoryItem*¿ history)
Initialize history according history of moves in QList *history.*

void kMancalaHistory::addState(player **p, board *b, int playerLastTurn, int playerNextTurn, int mo

Adds new history item into the list.

void kMancalaHistory::removeState(int id)
Removes state on row *id.*

2.3. Compilation

To compile this application you need Qt Framework[4] and also KDE[5] Library, also you will need KDE Games Library.

2.3.1. Debian/Ubuntu

On debian you have to install packages cmake, kdelibs5, kdelibs5-dev, libqt4-dev, libkdegames, libkdegames-dev, libqt4-svg, libqt4-xml. After you have installed these packges, to compile the application follow these steps.

```
tar xjfv kMancala-version.tar.bz2
cd kMancla-*
mkdir build
cd build
cmake ..
make
make install
```

Note: *version* replace by current version of the application.
There are also available debian and ubuntu packages at the homepage of project.

References

[1] Strategies and Tactics for Intelligent Search [online]. Algorithms.
Available at: `http://www.stanford.edu/~msirota/soco/minimax.html`.

[2] Qt Reference Documentation [online]. Graphics View Framework.
Available at: `http://doc.qt.nokia.com/latest/graphicsview.html`.

[3] KDE 4.0 API Reference [online]. KXmlGuiWindow Class Reference.
Available at: `http://api.kde.org/4.0-api/kdelibs-apidocs/kdeui/html/classkxmlguiwindow.html`

[4] Qt - Cross-platform application and UI framework. [online] Qt - A cross-platform application and UI framework.
Available at: `http://qt.nokia.com`

[5] KDE. [online] KDE.
Available at: `http://www.kde.org`