# Project Code

## Enum.cs

namespace WPDraughts

{

public enum PlayerColours { White, Black };

public enum MoveDirections { TopLeft, TopRight, BottomLeft, BottomRight };

public enum Difficulty { Hard, Normal, Easy };

public enum GameStates { TitleScreen, SelectDifficultyScreen, Playing, PieceSelected, NextJump, GameOver};

public enum PlayerTurn { WhiteTurn, BlackTurn };

public enum PlayerTypes { Human, Computer };

public enum GameType { TwoPlayer, OnePlayer };

}

## Player.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace WPDraughts

{

public class Player

{

private PlayerColours playerColour;

private PlayerTypes playerType;

public Player(PlayerTypes playerType, PlayerColours colour)

{

PlayerColour = colour;

this.PlayerType = playerType;

}

public PlayerTypes PlayerType

{

get { return playerType; }

set { playerType = value; }

}

public PlayerColours PlayerColour

{

get { return playerColour; }

set { playerColour = value; }

}

}

}

## DraughtPiece.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace WPDraughts

{

public class DraughtPiece

{

PlayerColours pieceColour;

int horizontalPostion, verticalPostion;

Boolean isKing;

public int HorizontalPostion

{

get { return horizontalPostion; }

set { horizontalPostion = value; }

}

public int VerticalPostion

{

get { return verticalPostion; }

set { verticalPostion = value; }

}

public Boolean IsKing

{

get { return isKing; }

set { isKing = value; }

}

public PlayerColours PieceColour

{

get { return pieceColour; }

set { pieceColour = value; }

}

public DraughtPiece(PlayerColours pieceColour, int horizontalPostion, int verticalPostion)

{

isKing = false;

PieceColour = pieceColour;

HorizontalPostion = horizontalPostion;

VerticalPostion = verticalPostion;

}

public DraughtPiece ClonePiece()

{

return (DraughtPiece)this.MemberwiseClone();

}

}

}

## GamePieceMove.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace WPDraughts

{

public class GamePieceMove : IEquatable<GamePieceMove>

{

private int newHorizontalPostion, newVerticalPostion, originalHorizontalPostion, originalVerticalPostion;

private MoveDirections moveDirection;

private bool isJump = false;

private bool isKingBeforeMove, isKingAfterMove;

private DraughtPiece movingPiece, jumpedPiece;

public GamePieceMove(DraughtPiece piece, int newHorizontalPostion, int newVerticalPostion)

{

NewHorizontalPostion = newHorizontalPostion;

NewVerticalPostion = newVerticalPostion;

MovingPiece = piece;

OriginalHorizontalPostion = piece.HorizontalPostion;

OriginalVerticalPostion = piece.VerticalPostion;

DetermineMoveDirection();

}

public bool IsKingAfterMove

{

get { return isKingAfterMove; }

set { isKingAfterMove = value; }

}

public bool IsKingBeforeMove

{

get { return isKingBeforeMove; }

set { isKingBeforeMove = value; }

}

public int OriginalVerticalPostion

{

get { return originalVerticalPostion; }

set { originalVerticalPostion = value; }

}

public int OriginalHorizontalPostion

{

get { return originalHorizontalPostion; }

set { originalHorizontalPostion = value; }

}

public DraughtPiece MovingPiece

{

get { return movingPiece; }

set { movingPiece = value; }

}

public DraughtPiece JumpedPiece

{

get { return jumpedPiece; }

set { jumpedPiece = value; }

}

public MoveDirections MoveDirection

{

get { return moveDirection; }

set { moveDirection = value; }

}

public bool IsJump

{

get { return isJump; }

set { isJump = value; }

}

public int NewVerticalPostion

{

get { return newVerticalPostion; }

set { newVerticalPostion = value; }

}

public int NewHorizontalPostion

{

get { return newHorizontalPostion; }

set { newHorizontalPostion = value; }

}

public bool IsNewLocationLegal(DraughtPiece piece)

{

if (piece.IsKing)

return IsNewLocationLegalForKingPiece();

else

return IsNewLocationLegalForNormalPiece(piece);

}

public bool IsNewLocationLegalForKingPiece()

{

if (IsMoveToTopLeft())

return true;

else if (IsMoveToBottomLeft())

return true;

else if (IsMoveToTopRight())

return true;

else if (IsMoveToBottomRight())

return true;

else if (IsJumpToTopLeft())

return true;

else if (IsJumpToBottomLeft())

return true;

else if (IsJumpToTopRight())

return true;

else if (IsJumpToBottomRight())

return true;

else

return false;

}

public bool IsNewLocationLegalForNormalPiece(DraughtPiece piece)

{

switch (piece.PieceColour)

{

case PlayerColours.White:

if (IsMoveToTopRight())

return true;

else if (IsMoveToBottomRight())

return true;

else if (IsJumpToTopRight())

return true;

else if (IsJumpToBottomRight())

return true;

break;

case PlayerColours.Black:

if (IsMoveToTopLeft())

return true;

else if (IsMoveToBottomLeft())

return true;

else if (IsJumpToTopLeft())

return true;

else if (IsJumpToBottomLeft())

return true;

break;

default:

return false;

}

return false;

}

private bool IsMoveToTopLeft()

{

if (NewHorizontalPostion == MovingPiece.HorizontalPostion - 1 &&

NewVerticalPostion == MovingPiece.VerticalPostion - 1)

return true;

else

return false;

}

private bool IsMoveToBottomLeft()

{

if (NewHorizontalPostion == MovingPiece.HorizontalPostion - 1 &&

NewVerticalPostion == MovingPiece.VerticalPostion + 1)

return true;

else

return false;

}

private bool IsMoveToTopRight()

{

if (NewHorizontalPostion == MovingPiece.HorizontalPostion + 1 &&

NewVerticalPostion == MovingPiece.VerticalPostion - 1)

return true;

else

return false;

}

private bool IsMoveToBottomRight()

{

if (NewHorizontalPostion == MovingPiece.HorizontalPostion + 1 &&

NewVerticalPostion == MovingPiece.VerticalPostion + 1)

return true;

else

return false;

}

private bool IsJumpToTopLeft()

{

if (NewHorizontalPostion == MovingPiece.HorizontalPostion - 2 &&

NewVerticalPostion == MovingPiece.VerticalPostion - 2)

{

IsJump = true;

return true;

}

else

return false;

}

private bool IsJumpToBottomLeft()

{

if (NewHorizontalPostion == MovingPiece.HorizontalPostion - 2 &&

NewVerticalPostion == MovingPiece.VerticalPostion + 2)

{

IsJump = true;

return true;

}

else

return false;

}

private bool IsJumpToTopRight()

{

if (NewHorizontalPostion == MovingPiece.HorizontalPostion + 2 &&

NewVerticalPostion == MovingPiece.VerticalPostion - 2)

{

IsJump = true;

return true;

}

else

return false;

}

private bool IsJumpToBottomRight()

{

if (NewHorizontalPostion == MovingPiece.HorizontalPostion + 2 &&

NewVerticalPostion == MovingPiece.VerticalPostion + 2)

{

IsJump = true;

return true;

}

else

return false;

}

private void DetermineMoveDirection()

{

if (IsJumpToTopLeft() || IsMoveToTopLeft())

MoveDirection = MoveDirections.TopLeft;

else if (IsMoveToBottomLeft() || IsJumpToBottomLeft())

MoveDirection = MoveDirections.BottomLeft;

else if (IsJumpToTopRight() || IsMoveToTopRight())

MoveDirection = MoveDirections.TopRight;

else

MoveDirection = MoveDirections.BottomRight;

}

public bool Equals(GamePieceMove other)

{

if (this.OriginalHorizontalPostion == other.OriginalHorizontalPostion && this.OriginalVerticalPostion == other.OriginalVerticalPostion)

{

if (this.NewHorizontalPostion == other.NewHorizontalPostion && this.NewVerticalPostion == other.NewVerticalPostion)

return true;

else

return false;

}

else

return false;

}

}

}

## GameBoard.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Collections;

namespace WPDraughts

{

public class GameBoard

{

private const int GAMEBOARDWIDTH = 8;

private const int GAMEBOARDHEIGHT = 8;

private DraughtPiece[,] gameBoardSpaces;

private bool forcedJumpsOn = true;

public bool ForcedJumpsOn

{

get { return forcedJumpsOn; }

set { forcedJumpsOn = value; }

}

public DraughtPiece[,] GameBoardSpaces

{

get { return gameBoardSpaces; }

set { gameBoardSpaces = value;}

}

public GameBoard()

{

SetTheGameBoard();

}

public int EvaluateBoard(PlayerColours playerColour)

{

const int MAXIMUMNUMBEROFOPPONENTPIECES = 12;

int normalPieces = 0, kingPieces = 0, opponentPieces = 0;

for (int i = 0; i <= GAMEBOARDWIDTH - 1; i++)

for (int j = 0; j <= GAMEBOARDHEIGHT - 1; j++)

{

if (gameBoardSpaces[i, j] != null)

if (gameBoardSpaces[i, j].PieceColour == playerColour)

{

if (gameBoardSpaces[i, j].IsKing == true)

kingPieces++;

else

normalPieces++;

}

else

opponentPieces++;

}

return normalPieces + (kingPieces \* 3) + (MAXIMUMNUMBEROFOPPONENTPIECES - opponentPieces);

}

public void ApplyMove(GamePieceMove move)

{

if(IsMoveLegal(move))

{

if (move.IsJump)

RemoveJumpedPiece(move);

gameBoardSpaces[move.NewHorizontalPostion, move.NewVerticalPostion] = move.MovingPiece;

gameBoardSpaces[move.MovingPiece.HorizontalPostion, move.MovingPiece.VerticalPostion] = null;

gameBoardSpaces[move.NewHorizontalPostion, move.NewVerticalPostion].HorizontalPostion = move.NewHorizontalPostion;

gameBoardSpaces[move.NewHorizontalPostion, move.NewVerticalPostion].VerticalPostion = move.NewVerticalPostion;

if (IsPieceToBeCrowned(move.MovingPiece))

move.MovingPiece.IsKing = true;

}

}

public void UnDoMove(GamePieceMove move)

{

GameBoardSpaces[move.NewHorizontalPostion, move.NewVerticalPostion] = null;

GameBoardSpaces[move.OriginalHorizontalPostion, move.OriginalVerticalPostion] = move.MovingPiece;

move.MovingPiece.HorizontalPostion = move.OriginalHorizontalPostion;

move.MovingPiece.VerticalPostion = move.OriginalVerticalPostion;

if (move.IsJump)

ReplaceJumpedPiece(move);

if (IsPieceToBeUnCrowned(move))

move.MovingPiece.IsKing = false;

}

public Boolean IsThereASecondJump(GamePieceMove move)

{

if (GetAllPossibleJumpsForThisPiece(move.MovingPiece).Count > 0)

return true;

else

return false;

}

public GameBoard CloneBoard()

{

GameBoard newBoard = (GameBoard)this.MemberwiseClone();

newBoard.GameBoardSpaces = (DraughtPiece[,])this.GameBoardSpaces.Clone();

for (int i = 0; i <= GAMEBOARDWIDTH - 1; i++)

for (int j = 0; j <= GAMEBOARDHEIGHT - 1; j++)

{

if(GameBoardSpaces[i,j] != null)

newBoard.GameBoardSpaces[i,j] = newBoard.GameBoardSpaces[i,j].ClonePiece();

}

return newBoard;

}

public Boolean IsPieceInThisSquare(int col, int row)

{

const int OUTSIDEOFBOARD = 8;

if (col == OUTSIDEOFBOARD || row == OUTSIDEOFBOARD)

return false;

else if (GameBoardSpaces[col, row] == null)

return false;

else

return true;

}

public List<GameBoard> GetPossibleBoards(PlayerColours colour)

{

List<GameBoard> allPossibleBoards = new List<GameBoard>();

foreach (GamePieceMove move in GetAllPossibleMoves(colour))

{

GameBoard newBoard = this.CloneBoard();

newBoard.ApplyMove(move);

allPossibleBoards.Add(newBoard);

}

return allPossibleBoards;

}

public List<GamePieceMove> GetAllPossibleMoves(PlayerColours colour)

{

List<GamePieceMove> allPossibleMoves = new List<GamePieceMove>();

allPossibleMoves.AddRange(GetAllPossibleJumps(colour));

if(forcedJumpsOn && allPossibleMoves.Count == 0)

allPossibleMoves.AddRange(GetAllPossibleNormalMoves(colour));

else if(!forcedJumpsOn)

allPossibleMoves.AddRange(GetAllPossibleNormalMoves(colour));

return allPossibleMoves;

}

public List<GamePieceMove> GetAllPossibleMovesForThisPiece(DraughtPiece piece)

{

List<GamePieceMove> allPossibleMovesForThisPiece = new List<GamePieceMove>();

allPossibleMovesForThisPiece.AddRange(GetAllPossibleJumpsForThisPiece(piece));

if (forcedJumpsOn && allPossibleMovesForThisPiece.Count == 0)

allPossibleMovesForThisPiece.AddRange(GetAllPossibleNormalMovesForThisPiece(piece));

else if (!forcedJumpsOn)

allPossibleMovesForThisPiece.AddRange(GetAllPossibleNormalMovesForThisPiece(piece));

return allPossibleMovesForThisPiece;

}

public List<GamePieceMove> GetAllPossibleNormalMovesForThisPiece(DraughtPiece piece)

{

List<GamePieceMove> moves = new List<GamePieceMove>();

GamePieceMove move;

move = new GamePieceMove(piece, piece.HorizontalPostion - 1, piece.VerticalPostion - 1);

if (IsMoveLegal(move))

moves.Add(move);

move = new GamePieceMove(piece, piece.HorizontalPostion - 1, piece.VerticalPostion + 1);

if (IsMoveLegal(move))

moves.Add(move);

move = new GamePieceMove(piece, piece.HorizontalPostion + 1, piece.VerticalPostion - 1);

if (IsMoveLegal(move))

moves.Add(move);

move = new GamePieceMove(piece, piece.HorizontalPostion + 1, piece.VerticalPostion + 1);

if (IsMoveLegal(move))

moves.Add(move);

return moves;

}

public List<GamePieceMove> GetAllPossibleJumpsForThisPiece(DraughtPiece piece)

{

List<GamePieceMove> jumps = new List<GamePieceMove>();

GamePieceMove jump;

jump = new GamePieceMove(piece, piece.HorizontalPostion - 2, piece.VerticalPostion - 2);

if (IsMoveLegal(jump))

jumps.Add(jump);

jump = new GamePieceMove(piece, piece.HorizontalPostion - 2, piece.VerticalPostion + 2);

if (IsMoveLegal(jump))

jumps.Add(jump);

jump = new GamePieceMove(piece, piece.HorizontalPostion + 2, piece.VerticalPostion - 2);

if (IsMoveLegal(jump))

jumps.Add(jump);

jump = new GamePieceMove(piece, piece.HorizontalPostion + 2, piece.VerticalPostion + 2);

if (IsMoveLegal(jump))

jumps.Add(jump);

return jumps;

}

public Boolean IsMovablePieceSelected(DraughtPiece selectedPiece, PlayerColours colour)

{

bool allMovesFound = true;

List<GamePieceMove> allPossibleMove = GetAllPossibleMoves(colour);

List<GamePieceMove> allPossibleMoveForThisPiece = GetAllPossibleMovesForThisPiece(selectedPiece);

foreach (GamePieceMove move in allPossibleMoveForThisPiece)

{

if (!allPossibleMove.Contains(move))

allMovesFound = false;

}

return allMovesFound;

}

public Boolean IsMoveLegal(GamePieceMove move)

{

if (IsNewLocationAnEmptySquare(move))

{

if (move.IsNewLocationLegal(move.MovingPiece))

{

if (move.IsJump && IsOpponentPieceInNextSquare(move))

return true;

else if (!move.IsJump)

return true;

else

return false;

}

else

return false;

}

else

return false;

}

public Boolean IsGameOver()

{

int blackCount = 0, whiteCount = 0;

blackCount = GetAllPossibleMoves(PlayerColours.Black).Count;

whiteCount = GetAllPossibleMoves(PlayerColours.White).Count;

if (blackCount == 0 || whiteCount == 0)

return true;

else

return false;

}

private List<GamePieceMove> GetAllPossibleJumps(PlayerColours colour)

{

List<GamePieceMove> allPossibleJumps = new List<GamePieceMove>();

for (int i = 0; i <= GAMEBOARDWIDTH - 1; i++)

for (int j = 0; j <= GAMEBOARDHEIGHT - 1; j++)

{

if (gameBoardSpaces[i, j] != null)

if (gameBoardSpaces[i, j].PieceColour == colour)

allPossibleJumps.AddRange(GetAllPossibleJumpsForThisPiece(gameBoardSpaces[i, j]));

}

return allPossibleJumps;

}

private List<GamePieceMove> GetAllPossibleNormalMoves(PlayerColours colour)

{

List<GamePieceMove> allPossibleNormalMoves = new List<GamePieceMove>();

for (int i = 0; i <= GAMEBOARDWIDTH - 1; i++)

for (int j = 0; j <= GAMEBOARDHEIGHT - 1; j++)

{

if (gameBoardSpaces[i, j] != null)

if (gameBoardSpaces[i, j].PieceColour == colour)

allPossibleNormalMoves.AddRange(GetAllPossibleNormalMovesForThisPiece(gameBoardSpaces[i, j]));

}

return allPossibleNormalMoves;

}

private void ReplaceJumpedPiece(GamePieceMove move)

{

gameBoardSpaces[move.JumpedPiece.HorizontalPostion, move.JumpedPiece.VerticalPostion] = move.JumpedPiece;

}

private void RemoveJumpedPiece(GamePieceMove move)

{

switch (move.MoveDirection)

{

case MoveDirections.TopLeft:

move.JumpedPiece = gameBoardSpaces[move.MovingPiece.HorizontalPostion - 1, move.MovingPiece.VerticalPostion - 1];

gameBoardSpaces[move.MovingPiece.HorizontalPostion - 1, move.MovingPiece.VerticalPostion - 1] = null;

break;

case MoveDirections.BottomLeft:

move.JumpedPiece = gameBoardSpaces[move.MovingPiece.HorizontalPostion - 1, move.MovingPiece.VerticalPostion + 1];

gameBoardSpaces[move.MovingPiece.HorizontalPostion - 1, move.MovingPiece.VerticalPostion + 1] = null;

break;

case MoveDirections.TopRight:

move.JumpedPiece = gameBoardSpaces[move.MovingPiece.HorizontalPostion + 1, move.MovingPiece.VerticalPostion - 1];

gameBoardSpaces[move.MovingPiece.HorizontalPostion + 1, move.MovingPiece.VerticalPostion - 1] = null;

break;

case MoveDirections.BottomRight:

move.JumpedPiece = gameBoardSpaces[move.MovingPiece.HorizontalPostion + 1, move.MovingPiece.VerticalPostion + 1];

gameBoardSpaces[move.MovingPiece.HorizontalPostion + 1, move.MovingPiece.VerticalPostion + 1] = null;

break;

}

}

private bool IsOpponentPieceInNextSquare(GamePieceMove move)

{

switch(move.MoveDirection)

{

case MoveDirections.TopLeft:

if (IsOpponentToTheTopLeft(move.MovingPiece))

return true;

break;

case MoveDirections.BottomLeft:

if (IsOpponentToTheBottomLeft(move.MovingPiece))

return true;

break;

case MoveDirections.TopRight:

if (IsOpponentToTheTopRight(move.MovingPiece))

return true;

break;

case MoveDirections.BottomRight:

if (IsOpponentToTheBottomRight(move.MovingPiece))

return true;

break;

default:

return false;

}

return false;

}

private Boolean IsOpponentToTheTopLeft(DraughtPiece piece)

{

if (gameBoardSpaces[piece.HorizontalPostion - 1, piece.VerticalPostion - 1] != null)

{

if (gameBoardSpaces[piece.HorizontalPostion - 1, piece.VerticalPostion - 1].PieceColour

== GetOpponentColour(piece.PieceColour))

return true;

else

return false;

}

else

return false;

}

private Boolean IsOpponentToTheBottomLeft(DraughtPiece piece)

{

if (gameBoardSpaces[piece.HorizontalPostion - 1, piece.VerticalPostion + 1] != null)

{

if (gameBoardSpaces[piece.HorizontalPostion - 1, piece.VerticalPostion + 1].PieceColour

== GetOpponentColour(piece.PieceColour))

return true;

else

return false;

}

else

return false;

}

private Boolean IsOpponentToTheTopRight(DraughtPiece piece)

{

if (gameBoardSpaces[piece.HorizontalPostion + 1, piece.VerticalPostion - 1] != null)

{

if (gameBoardSpaces[piece.HorizontalPostion + 1, piece.VerticalPostion - 1].PieceColour

== GetOpponentColour(piece.PieceColour))

return true;

else

return false;

}

else

return false;

}

private Boolean IsOpponentToTheBottomRight(DraughtPiece piece)

{

if (gameBoardSpaces[piece.HorizontalPostion + 1, piece.VerticalPostion + 1] != null)

{

if (gameBoardSpaces[piece.HorizontalPostion + 1, piece.VerticalPostion + 1].PieceColour

== GetOpponentColour(piece.PieceColour))

return true;

else

return false;

}

else

return false;

}

private PlayerColours GetOpponentColour(PlayerColours playerColour)

{

if (playerColour == PlayerColours.White)

return PlayerColours.Black;

else

return PlayerColours.White;

}

private Boolean IsPieceToBeCrowned(DraughtPiece piece)

{

const int BLACKSIDEOFBOARD = 7;

const int WHITESIDEOFBOARD = 0;

if (piece.PieceColour == PlayerColours.White)

if (piece.HorizontalPostion == BLACKSIDEOFBOARD)

return true;

else

return false;

else

if (piece.HorizontalPostion == WHITESIDEOFBOARD)

return true;

else

return false;

}

private Boolean IsPieceToBeUnCrowned(GamePieceMove move)

{

if (!move.IsKingBeforeMove && move.IsKingAfterMove)

return true;

else

return false;

}

private bool IsNewLocationAnEmptySquare(GamePieceMove move)

{

if (IsLocationInsideOfBoard(move))

{

if (gameBoardSpaces[move.NewHorizontalPostion, move.NewVerticalPostion] == null)

return true;

else

return false;

}

else

return false;

}

private bool IsLocationInsideOfBoard(GamePieceMove move)

{

if (move.NewHorizontalPostion >= 0 && move.NewHorizontalPostion <= 7 && move.NewVerticalPostion >= 0 && move.NewVerticalPostion <= 7)

return true;

else

return false;

}

private void SetTheGameBoard()

{

gameBoardSpaces = new DraughtPiece[GAMEBOARDWIDTH, GAMEBOARDHEIGHT];

for (int i = 0; i <= GAMEBOARDWIDTH - 1; i++)

for (int j = 0; j <= GAMEBOARDHEIGHT - 1; j++)

{

if ((i == 0 || i == 2) && j % 2 == 0)

gameBoardSpaces[i, j] = new DraughtPiece(PlayerColours.White,i,j);

else if (i == 1 && j % 2 != 0)

gameBoardSpaces[i, j] = new DraughtPiece(PlayerColours.White, i, j);

else if ((i == 5 || i == 7) && j % 2 != 0)

gameBoardSpaces[i, j] = new DraughtPiece(PlayerColours.Black, i, j);

else if (i == 6 && j % 2 == 0)

gameBoardSpaces[i, j] = new DraughtPiece(PlayerColours.Black, i, j);

else

gameBoardSpaces[i, j] = null;

}

}

}

}

## AI.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace WPDraughts

{

public class AI

{

private GamePieceMove bestMove;

private const int EASY = 2, NORMAL = 4, HARD = 8, FIRSTSTEP = 1;

private int numberOfMovesAhead;

private int alpha = 0, beta = Int32.MaxValue;

public void SetGameDifficulty(Difficulty difficulty)

{

if (difficulty == Difficulty.Easy)

numberOfMovesAhead = EASY;

else if (difficulty == Difficulty.Normal)

numberOfMovesAhead = NORMAL;

else

numberOfMovesAhead = HARD;

}

public List<GamePieceMove> GetAIMoves(GameBoard board)

{

List<GamePieceMove> moves = new List<GamePieceMove>();

MiniMax(board);

moves.Add(bestMove);

moves.AddRange(GetAddtionalMoves(board, bestMove));

return moves;

}

private void MiniMax(GameBoard board)

{

int bestValue = 0;

List<GamePieceMove> PossibleMoves = board.GetAllPossibleMoves(PlayerColours.Black);

foreach (GamePieceMove move in PossibleMoves)

{

GameBoard currentBoard = board.CloneBoard();

currentBoard.ApplyMove(move);

currentBoard = MakeAddtionalMoves(currentBoard, move).CloneBoard();

int currentValue = Min(currentBoard, FIRSTSTEP,ref alpha,ref beta);

board.UnDoMove(move);

if (currentValue > bestValue)

{

bestMove = move;

bestValue = currentValue;

}

}

}

private int Max(GameBoard board, int step, ref int alpha, ref int beta)

{

if (board.IsGameOver() || IsDepthReached(step))

return board.EvaluateBoard(PlayerColours.Black);

else

{

int bestValue = 0;

List<GamePieceMove> PossibleMoves = board.GetAllPossibleMoves(PlayerColours.Black);

foreach (GamePieceMove move in PossibleMoves)

{

GameBoard currentBoard = new GameBoard();

currentBoard = board.CloneBoard();

currentBoard.ApplyMove(move);

currentBoard = MakeAddtionalMoves(currentBoard, move).CloneBoard();

int currentValue = Min(currentBoard, step + 1, ref alpha, ref beta);

board.UnDoMove(move);

if (currentValue > bestValue)

{

bestValue = currentValue;

alpha = bestValue;

}

if (beta >= alpha)

return bestValue;

}

return bestValue;

}

}

private int Min(GameBoard board, int step,ref int alpha,ref int beta)

{

if (board.IsGameOver() || IsDepthReached(step))

return board.EvaluateBoard(PlayerColours.Black);

else

{

int bestValue = int.MaxValue;

List<GamePieceMove> PossibleMoves = board.GetAllPossibleMoves(PlayerColours.White);

foreach (GamePieceMove move in PossibleMoves)

{

GameBoard currentBoard = new GameBoard();

currentBoard = board.CloneBoard();

currentBoard.ApplyMove(move);

currentBoard = MakeAddtionalMoves(currentBoard, move).CloneBoard();

int currentValue = Max(currentBoard, step + 1, ref alpha, ref beta);

board.UnDoMove(move);

if (currentValue < bestValue)

{

bestValue = currentValue;

beta = currentValue;

}

if (beta <= alpha)

return bestValue;

}

return bestValue;

}

}

private GameBoard MakeAddtionalMoves(GameBoard board, GamePieceMove move)

{

List<GamePieceMove> addtionalMoves = new List<GamePieceMove>();

GameBoard bestBoard = board.CloneBoard();

int bestBoardValue = bestBoard.EvaluateBoard(PlayerColours.Black);

if (move.IsJump && board.IsThereASecondJump(move))

addtionalMoves.AddRange(board.GetAllPossibleJumpsForThisPiece(move.MovingPiece));

foreach (GamePieceMove m in addtionalMoves)

{

GameBoard currentBoard = new GameBoard();

currentBoard = board.CloneBoard();

currentBoard.ApplyMove(m);

currentBoard = MakeAddtionalMoves(currentBoard, m);

if (currentBoard.EvaluateBoard(PlayerColours.Black) > bestBoardValue)

bestBoard = currentBoard;

board.UnDoMove(m);

}

return bestBoard;

}

private List<GamePieceMove> GetAddtionalMoves(GameBoard board, GamePieceMove bestMove)

{

List<GamePieceMove> moves = new List<GamePieceMove>();

List<GamePieceMove> bestMoves = new List<GamePieceMove>();

int bestBoardValue = board.EvaluateBoard(PlayerColours.Black);

GameBoard nextBoard = new GameBoard();

nextBoard = board.CloneBoard();

nextBoard.ApplyMove(bestMove);

if (bestMove.IsJump)

moves.AddRange(board.GetAllPossibleJumpsForThisPiece(bestMove.MovingPiece));

foreach (GamePieceMove m in moves)

{

GameBoard currentBoard = new GameBoard();

currentBoard = nextBoard.CloneBoard();

currentBoard.ApplyMove(m);

if (currentBoard.EvaluateBoard(PlayerColours.Black) > bestBoardValue)

bestMoves.Add(m);

bestMoves.AddRange(GetAddtionalMoves(currentBoard, m));

board.UnDoMove(m);

}

board.UnDoMove(bestMove);

return bestMoves;

}

private Boolean IsDepthReached(int depth)

{

if (depth == numberOfMovesAhead)

return true;

else

return false;

}

}

}

## XNA Classes

## XNAGameBoardPostion.cs

using System;

using System.Collections.Generic;

using System.Linq;

using Microsoft.Xna.Framework;

namespace WPDraughts

{

public class XNAGameBoardPostion

{

Point gameBoardSpace;

Point phoneLocation;

Vector2 screenLocation;

public Point PhoneLocation

{

get { return phoneLocation; }

set { phoneLocation = value; }

}

public Point GameBoardSpace

{

get { return gameBoardSpace; }

set { gameBoardSpace = value; }

}

public Vector2 ScreenLocation

{

get { return screenLocation; }

set { screenLocation = value; }

}

public XNAGameBoardPostion(Point gameboardSpace)

{

GameBoardSpace = gameboardSpace;

}

public XNAGameBoardPostion(Point gameboardSpace, Vector2 screenLocation)

{

GameBoardSpace = gameboardSpace;

ScreenLocation = screenLocation;

}

}

}

## XNAGameBoard.cs

using System;

using System.Collections.Generic;

using System.Linq;

using Microsoft.Xna.Framework;

namespace WPDraughts

{

public class XNAGameBoard

{

Rectangle[,] boardSpaces;

const int GAMEBOARDWIDTH = 8;

const int GAMEBOARDHEIGHT = 8;

public XNAGameBoard()

{

boardSpaces = new Rectangle[GAMEBOARDHEIGHT, GAMEBOARDWIDTH];

}

public Rectangle[,] BoardSpaces

{

get { return boardSpaces; }

set { boardSpaces = value; }

}

}

}

## XNADraughtsGame.cs

using System;

using System.Collections.Generic;

using Microsoft.Xna.Framework;

using Microsoft.Xna.Framework.Graphics;

using Microsoft.Xna.Framework.Input;

using Microsoft.Xna.Framework.Input.Touch;

namespace WPDraughts

{

public class XNADraughtsGame : Microsoft.Xna.Framework.Game

{

private GraphicsDeviceManager graphics;

private SpriteBatch spriteBatch;

private Vector2 touchPostion;

private XNAGameScreenView xnaGameScreenView;

private XNAGameBoard xnaGameBoard;

private GameStates gameState;

private Player player1, player2;

private AI ai;

private PlayerTurn gameTurn;

private PlayerColours winner;

private GameType gameType;

private GameBoard gameBoard;

private List<Rectangle> squaresMovableTo;

private DraughtPiece selectedPiece;

private XNAGameBoardPostion selectedPoint;

public XNADraughtsGame()

{

graphics = new GraphicsDeviceManager(this);

Content.RootDirectory = "Content";

// Frame rate is 30 fps by default for Windows Phone.

TargetElapsedTime = TimeSpan.FromTicks(333333);

// Extend battery life under lock.

InactiveSleepTime = TimeSpan.FromSeconds(1);

graphics.PreferredBackBufferWidth = 480;

graphics.PreferredBackBufferHeight = 800;

}

protected override void Initialize()

{

gameBoard = new GameBoard();

xnaGameBoard = new XNAGameBoard();

xnaGameScreenView = new XNAGameScreenView(this, gameBoard, xnaGameBoard);

//gameTurn = PlayerTurn.WhiteTurn;

gameState = GameStates.TitleScreen;

TouchPanel.EnabledGestures = GestureType.Tap;

xnaGameScreenView.CreateHowManyPlayersButtons();

xnaGameScreenView.CreateCheckBox();

spriteBatch = new SpriteBatch(GraphicsDevice);

base.Initialize();

}

protected override void Update(GameTime gameTime)

{

// Allows the game to exit

if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)

this.Exit();

if (gameType == GameType.OnePlayer && gameTurn == PlayerTurn.BlackTurn)

{

System.Threading.Thread.Sleep(1000);

HandleBlackTurnForAIPlayer();

}

while (TouchPanel.IsGestureAvailable)

{

GestureSample gesture = TouchPanel.ReadGesture();

if (gameState == GameStates.TitleScreen)

HandleScreenTouch(gesture);

else if (gameState == GameStates.SelectDifficultyScreen)

HandleScreenTouch(gesture);

else if (gameState != GameStates.GameOver)

{

selectedPoint = GetTouchLocation(gesture);

if (gameTurn == PlayerTurn.WhiteTurn)

HandleTurnForHumanPlayer(PlayerColours.White, selectedPoint);

else if (gameType == GameType.TwoPlayer && gameTurn == PlayerTurn.BlackTurn)

HandleTurnForHumanPlayer(PlayerColours.Black, selectedPoint);

}

else

ResetGame();

}

base.Update(gameTime);

}

protected override void Draw(GameTime gameTime)

{

GraphicsDevice.Clear(Color.Black);

switch (gameState)

{

case GameStates.TitleScreen:

xnaGameScreenView.DrawBackGround(spriteBatch);

xnaGameScreenView.DrawTitleScreen(spriteBatch);

xnaGameScreenView.DrawForcedJumpOption(spriteBatch);

if (gameBoard.ForcedJumpsOn)

xnaGameScreenView.DrawCheckedBox(spriteBatch);

else

xnaGameScreenView.DrawUnCheckBox(spriteBatch);

break;

case GameStates.SelectDifficultyScreen:

xnaGameScreenView.DrawBackGround(spriteBatch);

xnaGameScreenView.DrawSelectDifficultyScreen(spriteBatch);

break;

case GameStates.Playing:

xnaGameScreenView.DrawCommonGameElements(spriteBatch, gameType, gameTurn);

break;

case GameStates.PieceSelected:

xnaGameScreenView.DrawCommonGameElements(spriteBatch, gameType, gameTurn);

xnaGameScreenView.DrawPossibleMoveSquares(spriteBatch, squaresMovableTo);

break;

case GameStates.NextJump:

xnaGameScreenView.DrawCommonGameElements(spriteBatch, gameType, gameTurn);

xnaGameScreenView.DrawPossibleMoveSquares(spriteBatch, squaresMovableTo);

break;

case GameStates.GameOver:

xnaGameScreenView.DrawWinner(spriteBatch, winner);

break;

}

base.Draw(gameTime);

}

private void HandleBlackTurnForAIPlayer()

{

List<GamePieceMove> aiMoves = ai.GetAIMoves(gameBoard.CloneBoard());

foreach(GamePieceMove move in aiMoves)

gameBoard.ApplyMove(move);

gameTurn = PlayerTurn.WhiteTurn;

if (IsGameOver())

{

gameState = GameStates.GameOver;

winner = PlayerColours.Black;

}

}

private void HandleTurnForHumanPlayer(PlayerColours currentPlayerColour, XNAGameBoardPostion selectedPoint)

{

switch (gameState)

{

case GameStates.Playing:

GetSelectedPiece(selectedPoint);

if(selectedPiece!=null)

if (gameBoard.IsMovablePieceSelected(selectedPiece, currentPlayerColour))

{

HighLightPossibleMoves(selectedPiece, currentPlayerColour);

gameState = GameStates.PieceSelected;

}

break;

case GameStates.PieceSelected:

if (IsNextClickAMove())

{

if (MakeMove(selectedPoint))

CompleteCurrentTurn(currentPlayerColour);

else

gameState = GameStates.NextJump;

}

else

{

UnSelectPiece();

gameState = GameStates.Playing;

}

break;

case GameStates.NextJump:

if (IsNextClickAMove())

{

if (MakeMove(selectedPoint))

CompleteCurrentTurn(currentPlayerColour);

}

break;

}

HandleAdditionalJumps(currentPlayerColour);

}

private void CompleteCurrentTurn(PlayerColours currentPlayerColour)

{

if (IsGameOver())

{

gameState = GameStates.GameOver;

winner = currentPlayerColour;

}

else

{

gameTurn = GetWhosTurnIsIt(gameTurn);

UnSelectPiece();

gameState = GameStates.Playing;

}

}

private void HandleAdditionalJumps(PlayerColours colour)

{

if (gameState == GameStates.NextJump)

{

Point jumpAbleSquare = new Point(selectedPiece.HorizontalPostion, selectedPiece.VerticalPostion);

XNAGameBoardPostion selectedSquare = new XNAGameBoardPostion(jumpAbleSquare);

HighLightPossibleJumps(selectedPiece, colour);

}

}

private void HighLightPossibleJumps(DraughtPiece selectedPiece, PlayerColours colour)

{

List<GamePieceMove> jumps = new List<GamePieceMove>();

squaresMovableTo = new List<Rectangle>();

jumps.AddRange(gameBoard.GetAllPossibleJumpsForThisPiece(selectedPiece));

foreach (GamePieceMove move in jumps)

{

Rectangle rect = xnaGameBoard.BoardSpaces[move.NewHorizontalPostion, move.NewVerticalPostion];

squaresMovableTo.Add(rect);

}

}

private XNAGameBoardPostion GetTouchLocation(GestureSample gesture)

{

const int OUTSIDEOFBOARD = 8;

touchPostion = gesture.Position;

Point touchPoint = new Point((int)touchPostion.X, (int)touchPostion.Y);

for (int i = 0; i < xnaGameBoard.BoardSpaces.GetLength(0); i++)

{

for (int j = 0; j < xnaGameBoard.BoardSpaces.GetLength(1); j++)

if (xnaGameBoard.BoardSpaces[i, j].Contains((int)touchPostion.X, (int)touchPostion.Y))

return new XNAGameBoardPostion(new Point(i, j), touchPostion);

}

return new XNAGameBoardPostion(new Point(OUTSIDEOFBOARD, OUTSIDEOFBOARD), touchPostion);

}

private void HandleScreenTouch(GestureSample gesture)

{

Point touchPoint = new Point((int)gesture.Position.X,(int)gesture.Position.Y);

if (xnaGameScreenView.OnePlayerButton.Contains(touchPoint))

OnePlayerButton\_Tapped();

else if (xnaGameScreenView.TwoPlayerButton.Contains(touchPoint))

TwoPlayerButton\_Tapped();

else if (xnaGameScreenView.CheckBox.Contains(touchPoint))

CheckBox\_Tapped();

else if (xnaGameScreenView.EasyButton.Contains(touchPoint))

EasyButton\_Tapped();

else if (xnaGameScreenView.NormalButton.Contains(touchPoint))

NormalButton\_Tapped();

else if (xnaGameScreenView.HardButton.Contains(touchPoint))

HardButton\_Tapped();

}

private void CheckBox\_Tapped()

{

if (gameBoard.ForcedJumpsOn)

gameBoard.ForcedJumpsOn = false;

else

gameBoard.ForcedJumpsOn = true;

}

private void HardButton\_Tapped()

{

gameState = GameStates.Playing;

ai.SetGameDifficulty(Difficulty.Hard);

xnaGameScreenView.RemoveChooseDifficultyButtons();

}

private void NormalButton\_Tapped()

{

gameState = GameStates.Playing;

ai.SetGameDifficulty(Difficulty.Normal);

xnaGameScreenView.RemoveChooseDifficultyButtons();

}

private void EasyButton\_Tapped()

{

gameState = GameStates.Playing;

ai.SetGameDifficulty(Difficulty.Easy);

xnaGameScreenView.RemoveChooseDifficultyButtons();

}

private void TwoPlayerButton\_Tapped()

{

gameState = GameStates.Playing;

gameType = GameType.TwoPlayer;

SetUpPlayers(gameType);

xnaGameScreenView.RemoveHowManyPlayersButtons();

xnaGameScreenView.RemoveCheckBox();

}

private void OnePlayerButton\_Tapped()

{

gameState = GameStates.SelectDifficultyScreen;

gameType = GameType.OnePlayer;

SetUpPlayers(gameType);

xnaGameScreenView.RemoveHowManyPlayersButtons();

xnaGameScreenView.RemoveCheckBox();

xnaGameScreenView.CreateChooseDifficultyButtons();

}

private Boolean MakeMove(XNAGameBoardPostion selectedSquare)

{

GamePieceMove move = new GamePieceMove(selectedPiece,

selectedSquare.GameBoardSpace.X, selectedSquare.GameBoardSpace.Y);

gameBoard.ApplyMove(move);

if(move.IsJump)

if(gameBoard.IsThereASecondJump(move))

return false;

return true;

}

private Boolean IsNextClickAMove()

{

foreach (Rectangle boardSquare in squaresMovableTo)

{

if (boardSquare.Contains(new Point((int)touchPostion.X, (int)touchPostion.Y)))

return true;

}

return false;

}

private void GetSelectedPiece(XNAGameBoardPostion selectedSquare)

{

if (gameBoard.IsPieceInThisSquare(selectedSquare.GameBoardSpace.X, selectedSquare.GameBoardSpace.Y))

selectedPiece = gameBoard.GameBoardSpaces[selectedSquare.GameBoardSpace.X, selectedSquare.GameBoardSpace.Y];

}

private void HighLightPossibleMoves(DraughtPiece selectedPiece, PlayerColours colour)

{

List<GamePieceMove> moves;

squaresMovableTo = new List<Rectangle>();

moves = gameBoard.GetAllPossibleMovesForThisPiece(selectedPiece);

foreach (GamePieceMove move in moves)

{

Rectangle rect = xnaGameBoard.BoardSpaces[move.NewHorizontalPostion, move.NewVerticalPostion];

squaresMovableTo.Add(rect);

}

}

private void UnSelectPiece()

{

squaresMovableTo = null;

selectedPiece = null;

}

private PlayerTurn GetWhosTurnIsIt(PlayerTurn currentPlayer)

{

if (currentPlayer == PlayerTurn.BlackTurn)

return PlayerTurn.WhiteTurn;

else

return PlayerTurn.BlackTurn;

}

private Boolean IsGameOver()

{

return gameBoard.IsGameOver();

}

private void SetUpPlayers(GameType gameType)

{

if (gameType == GameType.OnePlayer)

{

player2 = new Player(PlayerTypes.Computer, PlayerColours.Black);

ai = new AI();

}

else

player2 = new Player(PlayerTypes.Human, PlayerColours.Black);

player1 = new Player(PlayerTypes.Human, PlayerColours.White);

}

private void ResetGame()

{

ai = null;

Initialize();

}

}

}

## XNAGameScreenView.cs

using System;

using System.Collections.Generic;

using Microsoft.Xna.Framework;

using Microsoft.Xna.Framework.Content;

using Microsoft.Xna.Framework.Graphics;

namespace WPDraughts

{

public class XNAGameScreenView : Microsoft.Xna.Framework.GameComponent

{

private const int GAMEBOARDWIDTH = 8;

private const int GAMEBOARDHEIGHT = 8;

private const int SQUAREWIDTH = 54;

private const int SQUAREHEIGHT = 54;

private const int INITIALSQUAREX = 25;

private const int INITIALSQUAREY = 60;

private Texture2D title, onePlayer, twoPlayer, crown, blackWins, whiteWins, darkSquare, lightSquare, backGround,

yourTurn, phoneTurn, playerOneTurn, playerTwoTurn, selectDifficulty, easy, normal, hard, whitePiece, blackPiece,

blackKingPiece, whiteKingPiece, possibleMove, forceJumps, checkedBox, unCheckedBox, whiteTurn, blackTurn;

private Rectangle onePlayerButton, twoPlayerButton, easyButton, normalButton, hardButton, checkBox;

private XNAGameBoard xnaGameBoard;

private GameBoard gameBoard;

private ContentManager contentManager;

public XNAGameScreenView(Game game, GameBoard gameBoard, XNAGameBoard xnaGameBoard) : base(game)

{

contentManager = game.Content;

this.gameBoard = gameBoard;

darkSquare = contentManager.Load<Texture2D>(@"Textures\darksquare");

lightSquare = contentManager.Load<Texture2D>(@"Textures\lightsquare");

backGround = contentManager.Load<Texture2D>(@"Textures\background");

crown = contentManager.Load<Texture2D>(@"Textures\crown");

blackWins = contentManager.Load<Texture2D>(@"Textures\blackWins");

whiteWins = contentManager.Load<Texture2D>(@"Textures\whiteWins");

title = contentManager.Load<Texture2D>(@"Textures\Title");

onePlayer = contentManager.Load<Texture2D>(@"Textures\1Player");

twoPlayer = contentManager.Load<Texture2D>(@"Textures\2Player");

yourTurn = contentManager.Load<Texture2D>(@"Textures\YourTurn");

phoneTurn = contentManager.Load<Texture2D>(@"Textures\Thinking");

playerOneTurn = contentManager.Load<Texture2D>(@"Textures\Player1Turn");

playerTwoTurn = contentManager.Load<Texture2D>(@"Textures\Player2Turn");

selectDifficulty = contentManager.Load<Texture2D>(@"Textures\SelectDifficulty");

easy = contentManager.Load<Texture2D>(@"Textures\easy");

normal = contentManager.Load<Texture2D>(@"Textures\normal");

hard = contentManager.Load<Texture2D>(@"Textures\hard");

whitePiece = contentManager.Load<Texture2D>(@"Textures\whitepiece");

blackPiece = contentManager.Load<Texture2D>(@"Textures\blackPiece");

whiteKingPiece = contentManager.Load<Texture2D>(@"Textures\whiteking");

blackKingPiece = contentManager.Load<Texture2D>(@"Textures\blackking");

possibleMove = contentManager.Load<Texture2D>(@"Textures\possibleMove");

forceJumps = contentManager.Load<Texture2D>(@"Textures\ForceJumps");

checkedBox = contentManager.Load<Texture2D>(@"Textures\checked");

unCheckedBox = contentManager.Load<Texture2D>(@"Textures\unchecked");

whiteTurn = contentManager.Load<Texture2D>(@"Textures\whiteTurn");

blackTurn = contentManager.Load<Texture2D>(@"Textures\blackTurn");

this.xnaGameBoard = xnaGameBoard;

}

public Rectangle CheckBox

{

get { return checkBox; }

}

public Rectangle TwoPlayerButton

{

get { return twoPlayerButton; }

}

public Rectangle OnePlayerButton

{

get { return onePlayerButton; }

}

public Rectangle HardButton

{

get { return hardButton; }

}

public Rectangle NormalButton

{

get { return normalButton; }

}

public Rectangle EasyButton

{

get { return easyButton; }

}

public void DrawBackGround(SpriteBatch spriteBatch)

{

spriteBatch.Begin();

spriteBatch.Draw(backGround,

new Rectangle(0, 0, 480, 800), Color.White);

spriteBatch.End();

}

public void DrawCommonGameElements(SpriteBatch spriteBatch,GameType gameType, PlayerTurn gameTurn)

{

DrawBackGround(spriteBatch);

DrawBoard(spriteBatch);

DrawDraughtPieces(spriteBatch);

DrawWhosTurnLabel(spriteBatch, gameType, gameTurn);

DrawWhichColoursTurnItIsLabel(spriteBatch, gameTurn);

}

public void DrawBoard(SpriteBatch spriteBatch)

{

int pixelX = INITIALSQUAREX, pixelY = INITIALSQUAREY;

Texture2D currentSquare = darkSquare;

for (int row = 0; row < GAMEBOARDHEIGHT; row++)

{

for (int col = 0; col < GAMEBOARDWIDTH; col++)

{

DrawBoardSquare(spriteBatch, pixelX, pixelY, currentSquare, row, col);

currentSquare = ChangeSquareColour(currentSquare);

pixelY += SQUAREHEIGHT;

}

currentSquare = ChangeSquareColour(currentSquare);

pixelX += SQUAREWIDTH;

pixelY = INITIALSQUAREY;

}

}

public void DrawDraughtPieces(SpriteBatch spriteBatch)

{

for (int row = 0; row < GAMEBOARDHEIGHT; row++)

for (int col = 0; col < GAMEBOARDWIDTH; col++)

{

if (gameBoard.GameBoardSpaces[row, col] != null)

{

if (gameBoard.GameBoardSpaces[row, col].PieceColour == PlayerColours.White)

{

if (gameBoard.GameBoardSpaces[row, col].IsKing)

DrawDraughtsPiece(spriteBatch, whiteKingPiece, row, col);

else

DrawDraughtsPiece(spriteBatch, whitePiece, row, col);

}

else

{

if (gameBoard.GameBoardSpaces[row, col].IsKing)

DrawDraughtsPiece(spriteBatch, blackKingPiece, row, col);

else

DrawDraughtsPiece(spriteBatch, blackPiece, row, col);

}

}

}

}

public void DrawPossibleMoveSquares(SpriteBatch spriteBatch, List<Rectangle> squaresMovableTo)

{

foreach (Rectangle rect in squaresMovableTo)

{

spriteBatch.Begin();

spriteBatch.Draw(possibleMove, rect, Color.White);

spriteBatch.End();

}

}

public void DrawWinner(SpriteBatch spriteBatch, PlayerColours winningColour)

{

DrawWinningCrown(spriteBatch);

DrawWinningLabel(spriteBatch, winningColour);

}

public void DrawTitleScreen(SpriteBatch spriteBatch)

{

spriteBatch.Begin();

spriteBatch.Draw(title, new Rectangle(0, 100, 480, 150), Color.White);

spriteBatch.Draw(onePlayer, onePlayerButton, Color.White);

spriteBatch.Draw(twoPlayer, twoPlayerButton, Color.White);

spriteBatch.End();

}

public void DrawWhosTurnLabel(SpriteBatch spriteBatch, GameType gameType, PlayerTurn playerTurn)

{

if (gameType == GameType.OnePlayer)

DrawWhosTurnLabelForOnePlayerGame(spriteBatch, playerTurn);

else

DrawWhosTurnLabelForTwoPlayerGame(spriteBatch, playerTurn);

}

public void DrawWhichColoursTurnItIsLabel(SpriteBatch spriteBatch, PlayerTurn playerTurn)

{

spriteBatch.Begin();

if(playerTurn == PlayerTurn.WhiteTurn)

spriteBatch.Draw(whiteTurn, new Rectangle(90, 620, 320, 50), Color.White);

else

spriteBatch.Draw(blackTurn, new Rectangle(90, 620, 320, 50), Color.White);

spriteBatch.End();

}

public void DrawSelectDifficultyScreen(SpriteBatch spriteBatch)

{

spriteBatch.Begin();

spriteBatch.Draw(selectDifficulty, new Rectangle(0, 100, 460, 150), Color.White);

spriteBatch.Draw(easy, easyButton, Color.White);

spriteBatch.Draw(normal, normalButton, Color.White);

spriteBatch.Draw(hard, hardButton, Color.White);

spriteBatch.End();

}

public void DrawForcedJumpOption(SpriteBatch spriteBatch)

{

spriteBatch.Begin();

spriteBatch.Draw(forceJumps, new Rectangle(120, 670, 200, 120), Color.White);

spriteBatch.End();

}

public void DrawCheckedBox(SpriteBatch spriteBatch)

{

spriteBatch.Begin();

spriteBatch.Draw(checkedBox, checkBox, Color.White);

spriteBatch.End();

}

public void DrawUnCheckBox(SpriteBatch spriteBatch)

{

spriteBatch.Begin();

spriteBatch.Draw(unCheckedBox, checkBox, Color.White);

spriteBatch.End();

}

public void CreateCheckBox()

{

checkBox = new Rectangle(330,700,50,50);

}

public void RemoveCheckBox()

{

checkBox = new Rectangle(0, 0, 0, 0);

}

public void CreateHowManyPlayersButtons()

{

onePlayerButton = new Rectangle(90, 400, 300, 150);

twoPlayerButton = new Rectangle(90, 550, 300, 150);

}

public void CreateChooseDifficultyButtons()

{

easyButton = new Rectangle(120, 350, 233, 66);

normalButton = new Rectangle(120, 450, 233, 66);

hardButton = new Rectangle(120, 550, 233, 66);

}

public void RemoveHowManyPlayersButtons()

{

onePlayerButton = new Rectangle(0, 0, 0, 0);

twoPlayerButton = new Rectangle(0, 0, 0, 0);

}

public void RemoveChooseDifficultyButtons()

{

easyButton = new Rectangle(0, 0, 0, 0);

normalButton = new Rectangle(0, 0, 0, 0);

hardButton = new Rectangle(0, 0, 0, 0);

}

private void DrawWhosTurnLabelForTwoPlayerGame(SpriteBatch spriteBatch, PlayerTurn playerTurn)

{

spriteBatch.Begin();

if (playerTurn == PlayerTurn.WhiteTurn)

spriteBatch.Draw(playerOneTurn, new Rectangle(90, 580, 320, 50), Color.White);

else

spriteBatch.Draw(playerTwoTurn, new Rectangle(90, 580, 320, 50), Color.White);

spriteBatch.End();

}

private void DrawWhosTurnLabelForOnePlayerGame(SpriteBatch spriteBatch, PlayerTurn playerTurn)

{

spriteBatch.Begin();

if (playerTurn == PlayerTurn.WhiteTurn)

spriteBatch.Draw(yourTurn, new Rectangle(90, 580, 320, 50), Color.White);

else

{

spriteBatch.Draw(phoneTurn, new Rectangle(90, 580, 320, 50), Color.White);

}

spriteBatch.End();

}

private Texture2D ChangeSquareColour(Texture2D currentSquare)

{

if (currentSquare == lightSquare)

currentSquare = darkSquare;

else

currentSquare = lightSquare;

return currentSquare;

}

private void DrawWinningLabel(SpriteBatch spriteBatch, PlayerColours winningColour)

{

spriteBatch.Begin();

if (winningColour == PlayerColours.Black)

spriteBatch.Draw(blackWins, new Rectangle(0, 0, 480, 100), Color.White);

else

spriteBatch.Draw(whiteWins, new Rectangle(0, 0, 480, 100), Color.White);

spriteBatch.End();

}

private void DrawWinningCrown(SpriteBatch spriteBatch)

{

spriteBatch.Begin();

spriteBatch.Draw(crown,

new Rectangle(0, 0, 480, 800), Color.White);

spriteBatch.End();

}

private void DrawBoardSquare(SpriteBatch spriteBatch, int pixelX, int pixelY, Texture2D currentSquare, int row, int col)

{

xnaGameBoard.BoardSpaces[row, col] = new Rectangle(pixelX, pixelY, SQUAREWIDTH, SQUAREHEIGHT);

spriteBatch.Begin();

spriteBatch.Draw(currentSquare, xnaGameBoard.BoardSpaces[row, col], Color.White);

spriteBatch.End();

}

private void DrawDraughtsPiece(SpriteBatch spriteBatch, Texture2D draughtPiece, int row, int col)

{

spriteBatch.Begin();

spriteBatch.Draw(draughtPiece,

new Rectangle(xnaGameBoard.BoardSpaces[row, col].Left,

xnaGameBoard.BoardSpaces[row, col].Top,

SQUAREWIDTH,

SQUAREHEIGHT), Color.White);

spriteBatch.End();

}

}

}