Institiúid Teicneolaíochta Cheatharlach
INSTITUTE of TECHNOLOGY CARLOW
At the Heart of South Leinster

Viking Chess Using MCTS

Technical Manual

Declan Murphy – C00106936
Supervisor: Joseph Kehoe
2016

Contents

1. Introduction

This document details the source code for the Viking Chess game. There are two languages present in the application. The first is C#, this is used for the base model classes and the XAML code-behinds. The second are .xaml files which represent the user-interface.

At the time of writing, the code is available for download from Github at the following address:

https://github.com/MasterDex/VikingChess

2. C# Code

2.1. Namespace: VikingChess.Model

The VikingChess.Model namespace contains all classes necessary to play a game of Viking Chess.

2.1.1. Game.cs

2.1.1.1. Description

Contains the methods necessary to process a Viking Chess game.

2.1.1.2. Code

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Windows.UI.Xaml.Controls;

namespace VikingChess.Model
{
    /// <summary>
    /// The <see cref="Game"/> class contains the methods needed to process moves.
    /// </summary>
    public class Game
    {
        /// <summary>
        /// The Attacking Player
        /// </summary>
        private Player player1;
        /// <summary>
        /// The Defending Player
        /// </summary>
        private Player player2;
        /// <summary>
        /// The current turn count;
        /// </summary>
        private int turnCount = 0;
        /// <summary>
        /// The current player
        /// </summary>
        private Player currentPlayer;
        /// <summary>
        /// A <see cref="String"/> to store the move notation
        /// </summary>
        private string moveNotation = "";
        /// <summary>
        /// A <see cref="bool"/> representing if a piece has been captured or not.
        /// </summary>
        private Boolean pieceCaptured = false;
        /// <summary>
        /// The number of white pieces
        /// </summary>
        private int whitePieces = 0;
        /// <summary>
        /// The number of black pieces
        /// </summary>
        private int blackPieces = 0;
        /// <summary>
        /// The piece (if any) that has been captured.
        /// </summary>
        private Piece capturedPiece;
        /// <summary>
        /// A <see cref="bool"/> representing if the king has escaped or not.
        /// </summary>
        private bool kingEscaped = false;
        /// <summary>
        /// A <see cref="bool"/> representing if the king has been captured or not.
        /// </summary>
        private bool kingCaptured = false;
        /// <summary>
        /// A <see cref="bool"/> representing if the <see cref="Game"/> is over or not.
        /// </summary>
        private bool gameOver = false;
        /// <summary>
        /// The <see cref="Board"/> to use for this <see cref="Game"/>
        /// </summary>
        private Board board;
        /// <summary>
        /// A <see cref="bool"/> representing if the <see cref="Game"/> is being simulated or not.
        /// </summary>
        private bool simulated = false;
        /// <summary>
        /// A <see cref="string"/> defining the type of <see cref="Game"/>.
        /// </summary>
        private string gameType = null;
        /// <summary>
        /// The score for this <see cref="Game"/>
        /// </summary>
        private int score = 0;
        /// <summary>
        /// Contains all previous <see cref="Board"/> objects so that a move can be undone.
        /// </summary>
        private Dictionary<int, Board> stateHistory = new Dictionary<int, Board>();

        /// <summary>
        /// Blank Constructor
        /// </summary>
        public Game()
        {

        }

        /// <summary>
        /// Basic Constructor
        /// </summary>
        /// <param name="board">The <see cref="Board"/> to use for the <see cref="Game"/>.</param>
        public Game(Board board)
        {
            player1 = new Player(Enums.Color.BLACK, Enums.PlayerType.HUMAN);
            player2 = new Player(Enums.Color.WHITE, Enums.PlayerType.HUMAN);
            currentPlayer = player1;
            board.setCurrentPlayer(currentPlayer);
            this.board = board;
            this.blackPieces = board.countPieces(Enums.Color.BLACK);
            this.whitePieces = board.countPieces(Enums.Color.BLACK);
            Board newState = new Board(board);
            stateHistory.Add(turnCount, newState);
        }

        /// <summary>
        /// Constructor for simulated games. This is used only for the <see cref="MCTS"/> algorithm.
        /// </summary>
        /// <param name="board">The <see cref="Board"/> to use for the <see cref="Game"/>.</param>
        /// <param name="simulated"><see cref="simulated"/></param>
        public Game(Board board, bool simulated)
        {
            this.simulated = simulated;
            player1 = new Player(Enums.Color.BLACK, Enums.PlayerType.CPU);
            player2 = new Player(Enums.Color.WHITE, Enums.PlayerType.CPU);
            if(board.getCurrentPlayer().getColor().Equals(player1.getColor()))
            {
                currentPlayer = player1;

            }
            else
            {
                currentPlayer = player2;
            }
            this.board = board;
            blackPieces = board.countPieces(Enums.Color.BLACK);
            whitePieces = board.countPieces(Enums.Color.WHITE);
            Board newState = new Board(board);
            stateHistory.Add(turnCount, newState);
        }

        /// <summary>
        /// The Main Constructor
        /// </summary>
        /// <param name="board">The <see cref="Board"/> to use for the <see cref="Game"/>.</param>
        /// <param name="parameters">A <see cref="Tuple{T1, T2, T3, T4, T5}"/> containing the setup 
parameters for this game</param>
        public Game(Board board, Tuple<string, string, string, string, string> parameters)
        {
            if (parameters.Item2.Equals("ATTACKER"))
            {
                if (parameters.Item1.Equals("HUMAN"))
                {
                    player1 = new Player(Enums.Color.BLACK, Enums.PlayerType.HUMAN);
                    player2 = new Player(Enums.Color.WHITE, Enums.PlayerType.HUMAN);
                }
                else if (parameters.Item1.Equals("CPU"))
                {
                    player1 = new Player(Enums.Color.BLACK, Enums.PlayerType.HUMAN);
                    player2 = new Player(Enums.Color.WHITE, Enums.PlayerType.CPU);
                }
            }
            else if (parameters.Item2.Equals("DEFENDER"))
            {
                if (parameters.Item1.Equals("HUMAN"))
                {
                    player1 = new Player(Enums.Color.BLACK, Enums.PlayerType.HUMAN);
                    player2 = new Player(Enums.Color.WHITE, Enums.PlayerType.HUMAN);
                }
                else if (parameters.Item1.Equals("CPU"))
                {
                    player1 = new Player(Enums.Color.BLACK, Enums.PlayerType.CPU);
                    player2 = new Player(Enums.Color.WHITE, Enums.PlayerType.HUMAN);
                }
            }
            if(parameters.Item3 != "")
            {
                player1.setName(parameters.Item3);
            }
            else
            {
                if (player1.getType().Equals(Enums.PlayerType.CPU))
                {
                    player1.setName("CPU");
                }
                else
                {
                    player1.setName("Player 1");
                }

            }
            if(parameters.Item4 != "")
            {
                player2.setName(parameters.Item4);
            }
            else
            {
                if (player2.getType().Equals(Enums.PlayerType.CPU))
                {
                    player2.setName("CPU");
                }
                else
                {
                    player2.setName("Player 2");
                }
            }
            gameType = parameters.Item5;
            currentPlayer = player1;
            board.setCurrentPlayer(currentPlayer);
            this.board = board;
            blackPieces = board.countPieces(Enums.Color.BLACK);
            whitePieces = board.countPieces(Enums.Color.WHITE);
            Board newState = new Board(board);
            stateHistory.Add(turnCount, newState);
        }

        /// <summary>
        /// Sets the <see cref="currentPlayer"/> for this <see cref="Game"/>
        /// </summary>
        /// <param name="player">The <see cref="Player"/> to set as the <see
cref="currentPlayer"/>.</param>
        public void setCurrentPlayer(Player player)
        {
            currentPlayer = player;
        }

        /// <summary>
        /// Sets the <see cref="score"/> for this <see cref="Game"/>
        /// </summary>
        /// <param name="score"></param>
        public void setScore(int score)
        {
            this.score = score;
        }

        /// <summary>
        /// Returns the <see cref="Board"/> of this game.
        /// </summary>
        /// <returns><see cref="board"/></returns>
        public Board getBoard()
        {
            return board;
        }

        /// <summary>
        /// Updates the <see cref="currentPlayer"/> of this <see cref="Game"/>
        /// </summary>
        public void updateCurrentPlayer()
        {
            if(currentPlayer == player1)
            {
                currentPlayer = player2;
                board.setCurrentPlayer(player2);
            }
            else
            {
                currentPlayer = player1;
                board.setCurrentPlayer(player1);
            }
        }

        /// <summary>
        /// Returns the <see cref="currentPlayer"/> of this <see cref="Game"/>
        /// </summary>
        /// <returns><see cref="currentPlayer"/></returns>
        public Player getCurrentPlayer()
        {
            return currentPlayer;
        }

        /// <summary>
        /// Get the <see cref="player1"/> <see cref="Player"/>
        /// </summary>
        /// <returns><see cref="player1"/></returns>
        public Player getPlayer1()
        {
            return player1;
        }

        /// <summary>
        /// Get the <see cref="player2"/> <see cref="Player"/>
        /// </summary>
        /// <returns><see cref="player2"/></returns>
        public Player getPlayer2()
        {
            return player2;
        }

        /// <summary>
        /// Returns the <see cref="turnCount"/> of this <see cref="Game"/>
        /// </summary>
        /// <returns><see cref="turnCount"/></returns>
        public int getTurnCount()
        {
            return turnCount;
        }

        /// <summary>
        /// Move a <see cref="Piece"/> on the <see cref="board"/>
        /// </summary>
        /// <param name="destination">The destination <see cref="Square"/> to move the <see 
cref="Piece"/> to.</param>
        /// <param name="origin">The origin <see cref="Square"/> to move the <see cref="Piece"/> 
from.</param>
        /// <param name="selectedPiece">The selected <see cref="Piece"/> to move.</param>
        public void movePiece(Square destination, Square origin, Piece selectedPiece)
        {
            // If there is a valid piece
            if (selectedPiece != null && (selectedPiece.getColor() == currentPlayer.getColor()))
            {
                pieceCaptured = false;
                capturedPiece = null;
                List<Square> selectedPieceMoves = selectedPiece.generateLegalMoves(board);
                foreach (Square move in selectedPieceMoves)
                {
                    Square m = move;

                    // If the destination square is a valid move
                    if (destination.getFile().Equals(m.getFile()) && 
destination.getRank().Equals(m.getRank()))
                    {
                        // Remove the piece from the origin square
                        origin.setPiece(null);
                        // Set the location of the selected piece to the location of the destination 
square
                        selectedPiece.setLocation(destination.getRank(), destination.getFile());
                        // Set the destination square's piece to the selected piece
                        destination.setPiece(selectedPiece);
                        // Check if any captures occured
                        capturePiece(destination, selectedPiece);
                        checkKingMoves(selectedPiece, destination);
                        updateTurnCount();
                        // If a piece has been captured
                        if (pieceCaptured)
                        {
                            if(!simulated)
                            {
                                PlaySound("piece_captured.wav");
                            }
                            // Reflect this in the move notation (e.g. A8-B8xC8)
                            moveNotation = turnCount + ". " + origin.getFile() + (int)origin.getRank() 
+ "-" + destination.getFile() + (int)destination.getRank() + "x" + capturedPiece.getFile() + 
(int)capturedPiece.getRank() + "\n";
                        }
                        // Otherwise, just output the standard notation (e.g. A8 - B8)
                        else
                        {
                            if (!simulated)
                            {
                                PlaySound("piece_place.wav");
                            }
                            moveNotation = turnCount + ". " + origin.getFile() + (int)origin.getRank()
+ "-" + destination.getFile() + (int)destination.getRank() + "\n";
                        }
                        if (!simulated)
                        {
                            currentPlayer.updateMoveList(moveNotation);
                        }
                        updateCurrentPlayer();
                        Board newState = new Board(board);
                        stateHistory.Add(turnCount, newState);
                    }
                }

            }
        }

        /// <summary>
        /// Move a <see cref="Piece"/> on the <see cref="board"/>
        /// </summary>
        /// <param name="destination">The destination <see cref="Square"/> to move the <see
cref="Piece"/> to.</param>
        /// <param name="origin">The origin <see cref="Square"/> to move the <see cref="Piece"/>
from.</param>
        /// <param name="selectedPiece">The selected <see cref="Piece"/> to move.</param>
        /// <returns><see cref="board"/></returns>
        public Board movePieceAndGetBoard(Square destination, Square origin, Piece selectedPiece)
        {
            // If there is a valid piece
            if (selectedPiece != null && (selectedPiece.getColor() == currentPlayer.getColor()))
            {
                pieceCaptured = false;
                capturedPiece = null;
                List<Square> selectedPieceMoves = selectedPiece.generateLegalMoves(board);
                foreach (Square move in selectedPieceMoves)
                {
                    Square m = move;

                    // If the destination square is a valid move
                    if (destination.getFile().Equals(m.getFile()) &&
destination.getRank().Equals(m.getRank()))
                    {
                        // Remove the piece from the origin square
                        origin.setPiece(null);
                        // Set the location of the selected piece to the location of the destination
square
                        selectedPiece.setLocation(destination.getRank(), destination.getFile());
                        // Set the destination square's piece to the selected piece
                        destination.setPiece(selectedPiece);
                        // Check if any captures occured
                        capturePiece(destination, selectedPiece);
                        checkKingMoves(selectedPiece, destination);
                        updateTurnCount();
                        // If a piece has been captured
                        if (pieceCaptured)
                        {
                            if (!simulated)
                            {
                                PlaySound("piece_captured.wav");
                            }
                            // Reflect this in the move notation (e.g. A8-B8xC8)
                            moveNotation = turnCount + ". " + origin.getFile() + (int)origin.getRank()
+ "-" + destination.getFile() + (int)destination.getRank() + "x" + capturedPiece.getFile() +
(int)capturedPiece.getRank() + "\n";
                        }
                        // Otherwise, just output the standard notation (e.g. A8 - B8)
                        else
                        {
                            if (!simulated)
                            {
                                PlaySound("piece_place.wav");
                            }
                            moveNotation = turnCount + ". " + origin.getFile() + (int)origin.getRank()
+ "-" + destination.getFile() + (int)destination.getRank() + "\n";
                        }
                        if (!simulated)
                        {
                            currentPlayer.updateMoveList(moveNotation);
                        }
                        updateCurrentPlayer();
                        Board newState = new Board(board);
                        stateHistory.Add(turnCount, newState);
                    }
                }
            }
            return board;
        }

        /// <summary>
        /// Update the <see cref="turnCount"/> for this <see cref="Game"/>
        /// </summary>
        public void updateTurnCount()
        {
            ++turnCount;
        }

        /// <summary>
        /// Checks to see if the King has moved to an edge.
        /// </summary>
        /// <param name="selectedPiece">The <see cref="Piece"/> to check with.</param>
        /// <param name="destination">The <see cref="Square"/> to check against.</param>
        public void checkKingMoves(Piece selectedPiece, Square destination)
        {
            if(selectedPiece.getType() == Enums.PieceType.KING)
            {
                for (int k = 1; k < board.GetSize() - 2; ++k)
                {
                    // Top
                    if (board.GetSquare(1, k).Equals(destination))
                    {
                        kingEscaped = true;
                    }
                    // Left
                    else if (board.GetSquare(k, 1).Equals(destination))
                    {
                        kingEscaped = true;
                    }
                    // Vottom
                    else if (board.GetSquare(board.GetSize() - 2, k).Equals(destination))
                    {
                        kingEscaped = true;
                    }
                    // Right
                    else if (board.GetSquare(k, board.GetSize() - 2).Equals(destination))
                    {
                        kingEscaped = true;
                    }
                }
            }
        }

        /// <summary>
        /// Undo the last move
        /// </summary>
        /// <returns>The previous <see cref="Board"/> obtained from the <see
cref="stateHistory"/></returns>
        public Board undoMove()
        {
            if(turnCount > 0)
            {
                stateHistory.Remove(turnCount);
                --turnCount;
                moveNotation = moveNotation.Remove(moveNotation.Count() - 1);
                updateCurrentPlayer();
                currentPlayer.removeFromMoveList();
                board = new Board(stateHistory[turnCount]);
            }
            else
            {
                //return History[turnCount];
            }
            return board;

        }

        /// <summary>
        /// Checks to see if a <see cref="Player"/> has won or not.
        /// </summary>
        /// <returns><see cref="gameOver"/></returns>
        public bool checkWin()
        {
            // If the king has been captured, player 1 wins.
            if (kingCaptured)
            {
                player1.setWin(true);
                gameOver = true;
            }
            // If the king has escaped, player 2 wins.
            else if (kingEscaped)
            {
                player2.setWin(true);
                gameOver = true;
            }
            return gameOver;
        }

        /// <summary>
        /// Checks to see if the king is surrounded
        /// </summary>
        /// <param name="kingSquare">The <see cref="Square"/> that the king <see cref="Piece"/> is
on.</param>
        /// <returns>A <see cref="bool"/> representing if the king is surrounded or not.</returns>
        public bool checkKingCapture(Square kingSquare)
        {
            Boolean isNull = true;
            Square[] kingNeighbours = kingSquare.getNeighbours();

            // Ensure that there is a piece beside the king in every direction.
            if(kingNeighbours[0] != null && kingNeighbours[0].getPiece() != null &&
                kingNeighbours[1] != null && kingNeighbours[1].getPiece() != null &&
                kingNeighbours[2] != null && kingNeighbours[2].getPiece() != null &&
                kingNeighbours[3] != null && kingNeighbours[3].getPiece() != null)
            {
                isNull = false;
            }
            else
            {
                isNull = true;
            }

            // If there is a piece beisde the king in every direction and they are all a different
color to the king, the king can be captured.
            if((!isNull) &&
               kingNeighbours[0].getPiece().getColor() != kingSquare.getPiece().getColor() &&
               kingNeighbours[1].getPiece().getColor() != kingSquare.getPiece().getColor() &&
               kingNeighbours[2].getPiece().getColor() != kingSquare.getPiece().getColor() &&
               kingNeighbours[3].getPiece().getColor() != kingSquare.getPiece().getColor())
            {
                return true;
            }
            else
            {
                return false;
            }
        }

        /// <summary>
        /// Checks if there is a <see cref="Piece"/> that can be captured in the passed in direction.
        /// </summary>
        /// <param name="direction">The direction to check.</param>
        /// <param name="neighbourSquares">The neighbouring <see cref="Square"/> <see
cref="Array"/></param>
        /// <param name="selectedPiece">The <see cref="Piece"/> to check with.</param>
        public void checkCapture(int direction, Square[] neighbourSquares, Piece selectedPiece)
        {
            // Check for a Left Capture, ensure that there is a square to the left.
            if (neighbourSquares[direction] != null)
            {
                // Get the neighbours of the square
                Square[] NeighboursNeighbours = neighbourSquares[direction].getNeighbours();
                // Get the neighbour of the destination square in the specified direction
                Square neighbour = neighbourSquares[direction];
                // Ensure that the square is not null
                if (NeighboursNeighbours[direction] != null)
                {
                    // Get the neighbour of the nieghbour square
                    Square neighboursNeighbour = NeighboursNeighbours[direction];

                    // If the square contains a piece and it is a different color to the piece in the
destination square
                    if ((neighbour.getPiece() != null) && (neighbour.getPiece().getType() !=
Enums.PieceType.KING) && (neighbour.getPiece().getColor() != selectedPiece.getColor()))
                    {
                        // If the nneighbour's neighbour square has a piece and it is the same color
as the piece in the destination square
                        if ((neighboursNeighbour.getPiece() != null) &&
(neighboursNeighbour.getPiece().getColor() == selectedPiece.getColor()))
                        {
                            capturedPiece = neighbour.getPiece();
                            if (capturedPiece.getColor().Equals(player1.getColor()))
                            {
                                blackPieces--;
                                board.decrementBlackCount();
                            }
                            else
                            {
                                whitePieces--;
                                board.decrementWhiteCount();
                            }
                            // remove the piece from the board
                            neighbour.setPiece(null);
                            // Update the board
                            pieceCaptured = true;
                        }
                        // Otherwise, if the neighbour's square is a corner square or a throne square,
                        else if ((neighboursNeighbour.getPiece() == null) &&
((neighboursNeighbour.getSquareType() == Enums.SquareType.CORNER) ||
(neighboursNeighbour.getSquareType() == Enums.SquareType.THRONE)))
                        {
                            capturedPiece = neighbour.getPiece();
                            if (capturedPiece.getColor() == player1.getColor())
                            {
                                blackPieces--;
                            }
                            else
                            {
                                whitePieces--;
                            }

                            // remove the piece on the left square
                            neighbour.setPiece(null);
                            // Update the board
                            pieceCaptured = true;
                        }
                    }
                    // If the neighbour contains a King
                    else if (neighbourSquares[direction].getPiece() != null &&
neighbourSquares[direction].getPiece().getType() == Enums.PieceType.KING)
                    {
                        // If the king is surrounded
                        if (checkKingCapture(neighbourSquares[direction]))
                        {
                            capturedPiece = neighbourSquares[direction].getPiece();

                            neighbourSquares[direction].setPiece(null);
                            pieceCaptured = true;
                            kingCaptured = true;
                        }
                    }
                }
            }
        }

        /// <summary>
        /// Loops through each of the nieghbours of the destination <see cref="Square"/> and checks if
any piece has been captured.
        /// </summary>
        /// <param name="destination">The <see cref="Square"/> to check the neighbours of.</param>
        /// <param name="selectedPiece">The <see cref="Piece"/> capturing.</param>
        public void capturePiece(Square destination, Piece selectedPiece)
        {
            // Get the neighbours of the destination square
            Square[] neighbourSquares = destination.getNeighbours();

            for(int i = 0; i < 4; ++i)
            {
                checkCapture(i, neighbourSquares, selectedPiece);
            }
        }

        /// <summary>
        /// Plays a sound
        /// </summary>
        /// <param name="wavName">The name of the soundfile to play.</param>
        public async void PlaySound(string wavName)
        {
            MediaElement sound = new MediaElement();
            Windows.Storage.StorageFolder folder = await
Windows.ApplicationModel.Package.Current.InstalledLocation.GetFolderAsync("Assets\\Sounds");
            Windows.Storage.StorageFile file = await folder.GetFileAsync(wavName);
            var stream = await file.OpenAsync(Windows.Storage.FileAccessMode.Read);
            sound.SetSource(stream, file.ContentType);
            sound.Play();
            await Task.Delay(TimeSpan.FromSeconds(.5));
        }
    }
}
```

2.1.2. Player.cs

2.1.2.1. Description

Represents a Player in a game of Viking Chess

2.1.2.2. Code

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace VikingChess.Model
{
    /// <summary>
    /// The <see cref="Player"/> class represents a player in the <see cref="Game"/>
    /// </summary>
    public class Player
    {
        /// <summary>
        /// The <see cref="Enums.Color"/> of the <see cref="Player"/>
        /// </summary>
        private Enums.Color color;
        /// <summary>
        /// A <see cref="List{T}"/> containing the moves made by the player over the course of a <see
cref="Game"/>
        /// </summary>
        private List<String> moveList;
        /// <summary>
        /// The name of the <see cref="Player"/>
        /// </summary>
        private string name = "";
        /// <summary>
        /// The <see cref="Enums.PlayerType"/> of the <see cref="Player"/>
        /// </summary>
        private Enums.PlayerType type;
        /// <summary>
        /// A <see cref="bool"/> representing whether or not the <see cref="Player"/> has won or not.
        /// </summary>
        private Boolean hasWon = false;

        #region Constructors

        /// <summary>
        /// Simple Constructor
        /// </summary>
        public Player()
        {
            moveList = new List<String>();
        }

        /// <summary>
        /// Constructs a new <see cref="Player"/> object and sets its color.
        /// </summary>
        /// <param name="color"><see cref="color"/></param>
        public Player(Enums.Color color, Enums.PlayerType type)
        {
            setColor(color);
            setType(type);
            moveList = new List<String>();
        }

        /// <summary>
        /// Main Constructor
        /// </summary>
        /// <param name="color"><see cref="color"/></param>
        /// <param name="name"><see cref="name"/></param>
        /// <param name="type"><see cref="type"/></param>
        public Player(Enums.Color color, string name, Enums.PlayerType type)
        {
            setColor(color);
            setType(type);
            setName(name);
            moveList = new List<String>();
        }

        /// <summary>
        /// Copy Constructor
        /// </summary>
        /// <param name="player">The <see cref="Player"/> to copy</param>
        public Player(Player player)
        {
            setColor(player.getColor());
            setType(player.getType());
            setName(player.getName());
            moveList = player.getMoveList();
            hasWon = player.hasWon;
        }

        #endregion

        #region Getters & Setters

        /// <summary>
        /// Returns the <see cref="Enums.Color"/> of the <see cref="Player"/>.
        /// </summary>
        /// <returns><see cref="color"/></returns>
        public Enums.Color getColor()
        {
            return color;
        }

        /// <summary>
        /// Returns the <see cref="name"/> of the <see cref="Player"/>.
        /// </summary>
        /// <returns><see cref="name"/></returns>
        public string getName()
        {
            return name;
        }

        /// <summary>
        /// Returns the <see cref="Enums.PlayerType"/> of the <see cref="Player"/>
        /// </summary>
        /// <returns><see cref="type"/></returns>
        public Enums.PlayerType getType()
        {
            return type;
        }

        /// <summary>
        /// Returns the win status of the <see cref="Player"/>
        /// </summary>
        /// <returns><see cref="hasWon"/></returns>
        public Boolean getWin()
        {
            return hasWon;
        }

        /// <summary>
        /// Returns the <see cref="moveList"/> of the <see cref="Player"/>.
        /// </summary>
        /// <returns><see cref="moveList"/></returns>
        public List<String> getMoveList()
        {
            return moveList;
        }

        /// <summary>
        /// Sets the <see cref="Enums.Color"/> of the <see cref="Player"/>.
        /// </summary>
        /// <param name="color"><see cref="color"/></param>
        public void setColor(Enums.Color color)
        {
            this.color = color;
        }

        /// <summary>
        /// Sets the <see cref="name"/> of the <see cref="Player"/>
        /// </summary>
        /// <param name="name"><see cref="name"/></param>
        public void setName(string name)
        {
            this.name = name;
        }

        /// <summary>
        /// Sets the <see cref="Enums.PlayerType"/> of <see cref="Player"/>
        /// </summary>
        /// <param name="type"><see cref="type"/></param>
        public void setType(Enums.PlayerType type)
        {
            this.type = type;
        }

        /// <summary>
        /// Sets the win status of the <see cref="Player"/>.
        /// </summary>
        /// <param name="win"><see cref="hasWon"/></param>
        public void setWin(bool win)
        {
            hasWon = win;
        }

        #endregion

        #region Miscellaneous methods

        /// <summary>
        /// Adds a new entry to the <see cref="moveList"/> of the <see cref="Player"/>
        /// </summary>
        /// <param name="moveNotation">A string containing the move notation</param>
        public void updateMoveList(string moveNotation)
        {
            moveList.Add(moveNotation);
        }

        /// <summary>
        /// Removes an entry from the <see cref="moveList"/> of the <see cref="Player"/>.
        /// </summary>
        public void removeFromMoveList()
        {
            moveList.RemoveAt(moveList.Count - 1);
        }

        /// <summary>
        /// Use the MCTS algorithm to make a move
        /// </summary>
        /// <param name="board">The <see cref="Board"/> to use for the move</param>
        /// <returns>A <see cref="Tuple{T1, T2, T3}"/> containing the origin <see cref="Square"/>,
destination <see cref="Square"/> and <see cref="Piece"/> to move.</returns>
        public Tuple<Square, Piece, Square> makeMCTSMove(Board board)
        {
            MCTS mctsTree = new MCTS(board);

            Node node = mctsTree.makeMove(board);
            Tuple<Square, Piece, Square> moveTuple = node.GetLastMove();
            return moveTuple;
        }


        /// <summary>
        /// Returns a <see cref="List{T}"/> containing all <see cref="Square"/> objects that have <see
cref="Piece"/> objects that
        /// can be moved.
        /// </summary>
        /// <param name="board">The <see cref="Board"/> to get all <see cref="Square"/> objects
from.</param>
        /// <returns><see cref="List{T}"/> of <see cref="Square"/> objects containing <see
cref="Piece"/> objects.</returns>
        public List<Square> GetAllPieceSquares(Board board)
        {
            List<Square> squareList = new List<Square>();

            // Loop through the board and get all squares with moveable pieces
            for (int i = 0; i < board.GetSize(); ++i)
            {
                for (int j = 0; j < board.GetSize(); ++j)
                {
                    Square currSq = board.GetSquare(i, j);

                    if (currSq != null && currSq.getPiece() != null)
                    {
                        Piece currPiece = currSq.getPiece();

                        if (currPiece.getColor().Equals(this.getColor()) &&
currPiece.generateLegalMoves(board).Count != 0)
                        {
                            squareList.Add(currSq);
                        }
                    }
                }
            }
            return squareList;
        }
        /// <summary>
        /// Select the King <see cref="Piece"/> from the <see cref="List{T}"/> of moveable pieces
        /// </summary>
        /// <param name="squareList">A <see cref="List{T}"/> cntaining all <see cref="Square"/>
objects that have <see cref="Piece"/> objects.</param>
        /// <param name="board">The <see cref="Board"/> to use.</param>
        /// <returns>The <see cref="Square"/> containing the King <see cref="Piece"/></returns>
        private Square SelectKingSquare(List<Square> squareList, Board board)
        {
            //Loop through all moveable pieces
            for (int i = 0; i < squareList.Count; ++i)
            {
                Square currSq = squareList[i] as Square;
                Piece currPiece = currSq.getPiece();

                if (currPiece.getType().Equals(Enums.PieceType.KING))
                {
                    List<Square> kingMoves = currPiece.generateLegalMoves(board);

                    for (int j = 0; j < kingMoves.Count; ++j)
                    {
                        Square moveSq = kingMoves[j] as Square;
                        Tuple<Enums.Rank,Enums.File> sqLoc = moveSq.getLocation();

                        for(int k = 1; k < board.GetSize()-2; ++k)
                        {
                            Square[,] boardArr = board.GetBoard();

                            // If the king can move to an edge square
                            if(boardArr[1,k].getLocation().Equals(sqLoc) ||
boardArr[k,1].getLocation().Equals(sqLoc) ||
                                boardArr[board.GetSize()-2, k].getLocation().Equals(sqLoc) ||
boardArr[k, board.GetSize() - 2].getLocation().Equals(sqLoc))
                            {
                                return currSq;
                            }
                        }
                    }
                }
            }
            return null;
        }

        /// <summary>
        /// Select a <see cref="Piece"/> that can capture another <see cref="Piece"/>.
        /// </summary>
        /// <param name="squareList"></param>
        /// <param name="board"></param>
        /// <returns></returns>
        private Square SelectCaptureSquare(List<Square> squareList, Board board)
        {
            // Select a piece that can capture another piece
            for (int i = 0; i < squareList.Count; ++i)
            {
                Square currSq = squareList[i] as Square;
                List<Square> currMoves = currSq.getPiece().generateLegalMoves(board);

                // Get the neighbours of the current square
                for (int j = 0; j < currMoves.Count; ++j)
                {
                    Square moveSquare = currMoves[j] as Square;
                    // Get the location of this square
                    Tuple<Enums.Rank, Enums.File> sqLoc = moveSquare.getLocation();

                    //Get the neighbours of the current move
                    Square[] neighbours = moveSquare.getNeighbours();

                    //Check the neighbouring squares for a piece of the opposite color
                    for (int k = 0; k < neighbours.Length; ++k)
                    {
                        Square neighbour = neighbours[k] as Square;
                        if (neighbour != null && neighbour.getPiece() != null)
                        {
                            //If the piece on the neighbouring square is different to the player's
color
                            if (!(neighbour.getPiece().getColor().Equals(getColor())))
                            {
                                // Get the neighbours of this square
                                Square[] neighboursNeighbours = neighbour.getNeighbours();

                                for (int l = 0; l < neighboursNeighbours.Length; ++l)
                                {
                                    Square neighboursNeighbour = neighboursNeighbours[l] as Square;
                                    if (neighboursNeighbour != null && neighboursNeighbour.getPiece()
!= null)
                                    {
                                        // If the location of the nieghboursNeighbour is different to
the location of the current move square but has the same rank or file as the moveSquare
                                        if (!(neighboursNeighbour.getLocation().Equals(sqLoc)) &&
(neighboursNeighbour.getRank().Equals(moveSquare.getRank()) ||
neighboursNeighbour.getFile().Equals(moveSquare.getFile())))
                                            if
(neighboursNeighbour.getPiece().getColor().Equals(getColor()))
                                                return currSq;

                                    }
                                    else if (getColor().Equals(Enums.Color.BLACK) &&
neighbour.getPiece().Equals(Enums.PieceType.KING))
                                    {
                                        return currSq;
                                    }

                                }
                            }
                        }
                    }
                }
            }
            return null;
        }

        /// <summary>
        /// Selects a <see cref="Square"/> containing a valid <see cref="Piece"/> for the CPU <see
cref="Player"/>
        /// </summary>
        /// <param name="board">The current state of the <see cref="Board"/></param>
        /// <returns>A <see cref="Square"/> containing a valid <see cref="Piece"/> for the
CPU</returns>
        public Square selectPieceSquare(Board board)
        {
            List<Square> squareList = new List<Square>();
            Square pieceSquare = null;

            // Get moveable pieces
            squareList = GetAllPieceSquares(board);

            if(squareList.Count == 0)
            {
                Debug.WriteLine("Error: squareList.Count == 0");
                return null;
            }
            if(pieceSquare == null)
            {
                pieceSquare = SelectKingSquare(squareList, board);
            }
            if(pieceSquare == null)
            {
                // Otherwise return a random square
                Random rnd = new Random();
                int index = rnd.Next(0, squareList.Count);
                pieceSquare = squareList[index] as Square;
            }
            return pieceSquare;
        }

        /// <summary>
        /// Returns a <see cref="Tuple{T1, T2, T3}"/> containing the origin <see cref="Square"/>,
destination <see cref="Square"/> and <see cref="Piece"/> necessary to move.
        /// </summary>
        /// <param name="board">The <see cref="Board"/> to use when obtaining the move tuple</param>
        /// <returns>A <see cref="Tuple{T1, T2, T3}"/> containing the origin <see cref="Square"/>,
destination <see cref="Square"/> and <see cref="Piece"/> necessary to move.</returns>
        public Tuple<Square,Piece,Square> getMoveTuple(Board board)
        {
            Square originSquare = selectPieceSquare(board);
            Piece selectedPiece = originSquare.getPiece();
            Square destinationSquare;

            List<Square> moveList = selectedPiece.generateLegalMoves(board);

            if (moveList.Count == 0)
            {
                Debug.WriteLine("Error: squareList.Count == 0");
                return null;
            }

            Tuple<Square, Piece, Square> moveTuple;

            // Otherwise, pick a random destination
            Random rnd = new Random();
            int index = rnd.Next(0, moveList.Count);
            destinationSquare = moveList[index] as Square;

            moveTuple = new Tuple<Square, Piece,
Square>(originSquare,selectedPiece,destinationSquare);

            return moveTuple;
        }

        /// <summary>
        /// Tries to return a <see cref="Tuple{T1, T2, T3}"/> containing an origin <see
cref="Square"/>, destination <see cref="Square"/> and <see cref="Piece"/>
        /// that will capture another <see cref="Piece"/>
        /// </summary>
        /// <param name="board">The <see cref="Board"/> to use.</param>
        /// <returns>A <see cref="Tuple{T1, T2, T3}"/> containing an origin <see cref="Square"/>,
destination <see cref="Square"/> and <see cref="Piece"/></returns>
        public Tuple<Square, Piece, Square> capturePiece(Board board)
        {
            Square originSquare = selectPieceSquare(board);
            Piece selectedPiece = originSquare.getPiece();
            Square destinationSquare;

            List<Square> moveList = selectedPiece.generateLegalMoves(board);
            Tuple<Square, Piece, Square> moveTuple;

            // If the selected piece can capture another piece, move to capture
            foreach (Square m in moveList.ToList())
            {
                Square moveSquare = new Square(m);
                // Get the location of this square
                Tuple<Enums.Rank, Enums.File> sqLoc = moveSquare.getLocation();

                //Get the neighbours of the current move
                Square[] neighbours = moveSquare.getNeighbours();

                //Check the neighbouring squares for a piece of the opposite color
                for (int j = 0; j < neighbours.Length; ++j)
                {
                    Square neighbour = neighbours[j] as Square;
                    if (neighbour != null && neighbour.getPiece() != null)
                    {
                        //If the piece on the neighbouring square is different to the player's color
                        if ( !neighbour.getPiece().getColor().Equals(getColor()) )
                        {
                            // Get the neighbours of this square
                            Square[] neighboursNeighbours = neighbour.getNeighbours();

                            for (int k = 0; k < neighboursNeighbours.Length; ++k)
                            {
                                Square neighboursNeighbour = neighboursNeighbours[k] as Square;
                                if (neighboursNeighbour != null && neighboursNeighbour.getPiece() != 
null)
                                {
                                    // If the location of the nieghboursNeighbour is different to the 
location of the current move square
                                    if ((!neighboursNeighbour.getLocation().Equals(sqLoc)) && ( 
neighboursNeighbour.getRank().Equals(moveSquare.getRank()) || 
neighboursNeighbour.getFile().Equals(moveSquare.getFile()) ) )
                                    {
                                        if 
(neighboursNeighbour.getPiece().getColor().Equals(getColor()))
                                        {
                                            destinationSquare = moveSquare;
                                            moveTuple = new Tuple<Square, Piece, Square>(originSquare, 
selectedPiece, destinationSquare);

                                            return moveTuple;
                                        }
                                    }
                                }
                               else if (getColor().Equals(Enums.Color.BLACK) && 
neighbour.getPiece().Equals(Enums.PieceType.KING))
                                {
                                    destinationSquare = moveSquare;
                                    moveTuple = new Tuple<Square, Piece, Square>(originSquare, 
selectedPiece, destinationSquare);

                                    return moveTuple;
                                }

                            }
                        }
                    }
                }
            }
            return null;
        }

        /// <summary>
        /// Tries to return a <see cref="Tuple{T1, T2, T3}"/> containing an origin <see 
cref="Square"/>, destination <see cref="Square"/> and <see cref="Piece"/> 
        /// that will capture the King <see cref="Piece"/>
        /// </summary>
        /// <param name="board">The <see cref="Board"/> to use.</param>
        /// <param name="squareList">A <see cref="List{T}"/> containing all <see cref="Square"/> 
objects that have <see cref="Piece"/> objects on them.</param>
        /// <returns>A <see cref="Tuple{T1, T2, T3}"/> containing an origin <see cref="Square"/>, 
destination <see cref="Square"/> and <see cref="Piece"/></returns>
        public Tuple<Square, Piece, Square> captureKing(Board board, List<Square> squareList)
        {
            // Check if the King can be captured
            for (int i = 1; i < board.GetSize()-2; ++i)
            {
                for (int j = 1; j < board.GetSize()-2; ++j)
                {
                    Square currSquare = board.GetSquare(i, j);

                    if (currSquare != null)
                    {
                        Piece currPiece = currSquare.getPiece();
                        if (currPiece != null && currPiece.getType().Equals(Enums.PieceType.KING))
                        {
                            Square kingSquare = currSquare;
                            Tuple<int, int> kingIndex = board.GetPosition(new Tuple<Enums.Rank, 
Enums.File>(kingSquare.getRank(), kingSquare.getFile()));
                            Square[] kingNeighbours = kingSquare.getNeighbours();
                            List<Square> kingMoves = kingSquare.getPiece().generateLegalMoves(board);

                            for (int k = 0; k < squareList.Count; ++k)
                            {
                                Square selectedSquare = squareList[k] as Square;
                                Piece selectedPiece = selectedSquare.getPiece();
                                List<Square> moves = selectedPiece.generateLegalMoves(board);
                                if (selectedPiece.getColor().Equals(Enums.Color.BLACK))
                                {
                                    Square blockingSquare = null;

                                    // First see if the King can escape and find the square to block
it.
                                    for (int l = 0; l < kingMoves.Count; ++l)
                                    {
                                        for (int m = 1; m < board.GetSize() - 2; ++m)
                                        {
                                            Square currMove = kingMoves[l] as Square;
                                            // Top
                                            if (board.GetSquare(1, m).Equals(currMove))
                                            {
                                                blockingSquare = board.GetSquare(kingIndex.Item1 - 1,
m);
                                            }
                                            // Left
                                            else if (board.GetSquare(m, 1).Equals(currSquare))
                                            {
                                                blockingSquare = board.GetSquare(m, kingIndex.Item2 -
1);
                                            }
                                            // Bottom
                                            else if (board.GetSquare(board.GetSize() - 2,
m).Equals(currSquare))
                                            {
                                                blockingSquare = board.GetSquare(kingIndex.Item1 + 1,
m);
                                            }
                                            // Right
                                            else if (board.GetSquare(m, board.GetSize() -
2).Equals(currSquare))
                                            {
                                                blockingSquare = board.GetSquare(m, kingIndex.Item2 +
1);
                                            }
                                        }

                                    }

                                    Square destination = null;

                                    for (int l = 0; l < moves.Count; ++l)
                                    {
                                        // If you can block the king then do so
                                        if (blockingSquare != null)
                                        {
                                            if (moves[l].Equals(blockingSquare))
                                            {
                                                Tuple<Square, Piece, Square> moveTuple = new
Tuple<Square, Piece, Square>(selectedSquare, selectedPiece, blockingSquare);

                                                return moveTuple;
                                            }
                                        }
                                        else
                                        {
                                            for (int m = 0; m < kingNeighbours.Length; ++m)
                                            {
                                                if (moves[l].Equals(kingNeighbours[m]))
                                                {
                                                    destination = moves[l] as Square;

                                                    Tuple<Square, Piece, Square> moveTuple = new
Tuple<Square, Piece, Square>(selectedSquare, selectedPiece, destination);

                                                    return moveTuple;
                                                }
                                            }
                                        }
                                    }
                                }
                            }
```

```
                        }

                    }
                }
            }
            return null;
        }

        /// <summary>
        /// Tries to return a <see cref="Tuple{T1, T2, T3}"/> containing an origin <see
cref="Square"/>, destination <see cref="Square"/> and <see cref="Piece"/>
        /// that will allow the King to escape
        /// </summary>
        /// <param name="board">The <see cref="Board"/> to use.</param>
        /// <param name="squareList">A <see cref="List{T}"/> containing all <see cref="Square"/>
objects that have <see cref="Piece"/> objects on them.</param>
        /// <returns>A <see cref="Tuple{T1, T2, T3}"/> containing an origin <see cref="Square"/>,
destination <see cref="Square"/> and <see cref="Piece"/></returns>
        public Tuple<Square, Piece, Square> escapeKing(Board board, List<Square> squareList)
        {
            Square destination = null;

            for(int i = 0; i < squareList.Count; ++ i)
            {
                Square selectedSquare = squareList[i] as Square;

                if(selectedSquare != null &&
selectedSquare.getPiece().getType().Equals(Enums.PieceType.KING))
                {
                    Square kingSquare = selectedSquare;
                    Piece kingPiece = selectedSquare.getPiece();

                    List<Square> moveList = kingPiece.generateLegalMoves(board);

                    for(int j = 0; j < moveList.Count; ++j)
                    {
                        for(int k = 1; k < board.GetSize()-2; ++k)
                        {
                            Square currSquare = moveList[j] as Square;
                            // Top
                            if (board.GetSquare(1, k).Equals(currSquare))
                            {
                                destination = currSquare;
                            }
                            // Left
                            else if(board.GetSquare(k, 1).Equals(currSquare))
                            {
                                destination = currSquare;
                            }
                            // Bottom
                            else if (board.GetSquare(board.GetSize()-2, k).Equals(currSquare))
                            {
                                destination = currSquare;
                            }
                            // Right
                            else if (board.GetSquare(k, board.GetSize()-2).Equals(currSquare))
                            {
                                destination = currSquare;
                            }
                            if(destination != null)
                            {
                                Tuple<Square, Piece, Square> moveTuple = new Tuple<Square, Piece,
Square>(kingSquare, kingPiece, destination);

                                return moveTuple;
                            }
                        }

                    }
                }
            }
            return null;
        }
        /// <summary>
        /// Tries to return a <see cref="Tuple{T1, T2, T3}"/> containing an origin <see
cref="Square"/>, destination <see cref="Square"/> and <see cref="Piece"/>
        /// that will result in a win.
        /// </summary>
        /// <param name="board">The <see cref="Board"/> to use</param>
        /// <returns>A <see cref="Tuple{T1, T2, T3}"/> containing an origin <see cref="Square"/>,
destination <see cref="Square"/> and <see cref="Piece"/></returns>
        public Tuple<Square, Piece, Square> GetWinningMove(Board board)
        {
            List<Square> squareList = GetAllPieceSquares(board);
            Tuple<Square, Piece, Square> moveTuple = null;

            if (getColor().Equals(Enums.Color.BLACK))
            {
                moveTuple = captureKing(board, squareList);
            }
            else if (getColor().Equals(Enums.Color.WHITE))
            {
                moveTuple = escapeKing(board, squareList);
            }
            return moveTuple;
        }

        /// <summary>
        /// <see cref="ToString"/> override
        /// </summary>
        /// <returns>A <see cref="String"/> representation of the <see cref="Player"/></returns>
        public override string ToString()
        {
            string theString = "";

            theString += "\n | Color : " + getColor().ToString();
            theString += "\n | Type  : " + getType().ToString();
            theString += "\n | Name  : " + getName();
            theString += "\n | Moves : " + getMoveList().ToString();

            return theString;
        }

        #endregion
    }
}
```

2.1.3. Piece.cs

2.1.3.1. Description

Represents a game piece in a game of Viking Chess

2.1.3.2. Code

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Diagnostics;

namespace VikingChess.Model
{
    /// <summary>
    /// The Piece class represents a gamepiece on the <see cref="Board"/>
    /// </summary>
    public class Piece
    {
        /// <summary>
        /// The <see cref="Enums.Color"/> of the <see cref="Piece"/>
        /// </summary>
        private Enums.Color color;
        /// <summary>
        /// The <see cref="Enums.PieceType"/> of the <see cref="Piece"/>
        /// </summary>
        private Enums.PieceType type;
        /// <summary>
        /// The <see cref="Piece"/>'s <see cref="Enums.Rank"/> on the <see cref="Board"/>
        /// </summary>
        private Enums.Rank rank;
        /// <summary>
        /// The <see cref="Piece"/>'s <see cref="Enums.File"/> on the <see cref="Board"/>
        /// </summary>
        private Enums.File file;
        /// <summary>
        /// An <see cref="List{T}"/> containing all legal moves for the <see cref="Piece"/> from its
current location on the <see cref="Board"/>
        /// </summary>
        private List<Square> legalMoves = new List<Square>();

        #region Constructors

        /// <summary>
        /// Blank Constructor.
        /// </summary>
        public Piece()
        {

        }

        /// <summary>
        /// Basic Constructor
        /// </summary>
        /// <param name="color"><see cref="color"/></param>
        /// <param name="type"><see cref="type"/></param>
        public Piece(Enums.Color color, Enums.PieceType type)
        {
            setColor(color);
            setType(type);
        }

        /// <summary>
        /// Main Constructor
        /// </summary>
        /// <param name="color"><see cref="color"/></param>
        /// <param name="type"><see cref="type"/></param>
        /// <param name="rank"><see cref="rank"/></param>
        /// <param name="file"><see cref="file"/></param>
        public Piece(Enums.Color color, Enums.PieceType type, Enums.Rank rank, Enums.File file)
        {
            setColor(color);
            setType(type);
            setLocation(rank, file);
        }

        /// <summary>
        /// Copy Constructor
        /// </summary>
        /// <param name="previousPiece">The <see cref="Piece"/> to copy</param>
        public Piece(Piece previousPiece)
        {
            if (previousPiece != null)
            {
                color = previousPiece.color;
                type = previousPiece.type;
                file = previousPiece.file;
                rank = previousPiece.rank;
            }
        }

        #endregion
        #region Getters and Setters

        /// <summary>
        /// Returns the <see cref="color"/> of the <see cref="Piece"/>
        /// </summary>
        /// <returns><see cref="color"/></returns>
        public Enums.Color getColor()
        {
            return color;
        }
        /// <summary>
        /// Returns the <see cref="type"/> of the <see cref="Piece"/>
        /// </summary>
        /// <returns><see cref="type"/></returns>
        public Enums.PieceType getType()
        {
            return type;
        }

        /// <summary>
        /// Returns the rank of the <see cref="Piece"/> on the board.
        /// </summary>
        /// <returns><see cref="rank"/></returns>
        public Enums.Rank getRank()
        {
            return rank;
        }

        /// <summary>
        /// Returns the file of the <see cref="Piece"/> on the board.
        /// </summary>
        /// <returns><see cref="file"/></returns>
        public Enums.File getFile()
        {
            return file;
        }

        /// <summary>
        /// Returns a <see cref="Tuple{T1, T2}"/> containing the rank and file of the <see
cref="Piece"/>
        /// </summary>
        /// <returns>A <see cref="Tuple{T1, T2}"/> containing the rank and file of the <see
cref="Piece"/></returns>
        public Tuple<Enums.Rank, Enums.File> getLocation()
        {
            return new Tuple<Enums.Rank, Enums.File>(rank, file);
        }

        /// <summary>
        /// Sets the rank of the <see cref="Piece"/>
        /// </summary>
        /// <param name="rank"><see cref="rank"/></param>
        public void setRank(Enums.Rank rank)
        {
            this.rank = rank;
        }

        /// <summary>
        /// Sets the file of the <see cref="Piece"/>
        /// </summary>
        /// <param name="file"><see cref="file"/></param>
        public void setFile(Enums.File file)
        {
            this.file = file;
        }

        /// <summary>
        /// Sets the rank and file of this <see cref="Piece"/>
        /// </summary>
        /// <param name="rank"><see cref="rank"/></param>
        /// <param name="file"><see cref="file"/></param>
        public void setLocation(Enums.Rank rank, Enums.File file)
        {
            setRank(rank);
            setFile(file);
        }

        /// <summary>
        /// Sets the <see cref="color"/> of the <see cref="Piece"/>.
        /// </summary>
        /// <param name="color">The <see cref="color"/> of the <see cref="Piece"/>. Can be either <see
cref="Enums.Color.BLACK"/> or <see cref="Enums.Color.WHITE"/>.</param>
        public void setColor(Enums.Color color)
        {
            this.color = color;
        }

        /// <summary>
        /// Sets the <see cref="type"/> of the <see cref="Piece"/>.
        /// </summary>
        /// <param name="type">The <see cref="type"/> of the <see cref="Piece"/>. Can be either <see 
cref="Enums.PieceType.PAWN"/> or <see cref="Enums.PieceType.KING"/>.</param>
        public void setType(Enums.PieceType type)
        {
            this.type = type;
        }

        #endregion
        #region Additional Methods

        /// <summary>
        /// Returns true or false if the <see cref="Piece"/> can move to the <see cref="Square"/>
        /// </summary>
        /// <param name="square">The <see cref="Square"/> to check against</param>
        /// <returns>true/false</returns>
        public Boolean canMoveTo(Square square)
        {
            // If the piece is a KING
            if (getType().Equals(Enums.PieceType.KING) )
            {
                // Return true if the square is empty, false otherwise
                return square.getPiece() == null ? true : false;
            }

            // If the square is NOT a CORNER or a THRONE and the piece is a PAWN
            if (!(square.getSquareType().Equals(Enums.SquareType.CORNER)) && 
!(square.getSquareType().Equals(Enums.SquareType.THRONE)) && (getType().Equals(Enums.PieceType.PAWN)))
            {
                // Return true if the square is empty, false otherwise
                return square.getPiece() == null ? true : false;
            }
            else return false;
        }

        /// <summary>
        /// Returns an List containing all legal Moves for the <see cref="Piece"/> from its current 
<see cref="Square"/>.
        /// </summary>
        /// <param name="board">The <see cref="Board"/> that represents the state to check 
from.</param>
        /// <returns>An <see cref="List{T}"/> containing all <see cref="Square"/>s that the <see 
cref="Piece"/> can move to.</returns>
        public List<Square> generateLegalMoves(Board board)
        {
            // Get a tuple containing the integer co-ordinates of the Piece on the board
            Tuple<int, int> position = board.GetPosition(getLocation());
            int row = position.Item1;
            int column = position.Item2;

            // Reset the List to ensure no outdated legal moves exist.
            legalMoves.Clear();

            // Generate Up Moves
            for (int i = row - 1; i >= 0; --i)
            {
                Square square = board.GetSquare(i, column);
                if(square != null)
                {
                    if (canMoveTo(square))
                    {
                        legalMoves.Add(square);
                    }
                    else
                    {
                        break;
                    }
                }
            }
            row = position.Item1;
            column = position.Item2;
            // Generate Down Moves
            for (int i = row + 1; i < board.GetSize(); ++i)
            {
                Square square = board.GetSquare(i, column);
                if (square != null)
                {
                    if (canMoveTo(square))
                    {
                        legalMoves.Add(square);
                    }
                    else
                    {
                        break;
                    }
                }
            }

            row = position.Item1;
            column = position.Item2;
            // Generate Left Moves
            for (int i = column - 1; i >= 0; --i)
            {
                Square square = board.GetSquare(row, i);
                if (square != null)
                {
                    if (canMoveTo(square))
                    {
                        legalMoves.Add(square);
                    }
                    else
                    {
                        break;
                    }
                }
            }

            row = position.Item1;
            column = position.Item2;
            // Generate Right Moves
            for (int i = column + 1; i < board.GetSize(); ++i)
            {
                Square square = board.GetSquare(row, i);
                if (square != null)
                {
                    if (canMoveTo(square))
                    {
                        legalMoves.Add(square);
                    }
                    else
                    {
                        break;
                    }
                }
            }
            return legalMoves;
        }

        /// <summary>
        /// <see cref="ToString"/> override
        /// </summary>
        /// <returns>A <see cref="String"/> representation of the <see cref="Piece"/></returns>
        public override string ToString()
        {
            string theString = "";

            theString += "\n | Color : " + getColor().ToString();
            theString += "\n | Type  : " + getType().ToString();
            theString += "\n | Location  : " + getFile() + (int)getRank();

            return theString;
        }
        #endregion
    }
}
```

2.1.4. Square.cs

2.1.4.1. Description

Represents a square on the game board in a game of Viking Chess

2.1.4.2. Code

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace VikingChess.Model
{
    /// <summary>
    /// The <see cref="Square"/> class represents a <see cref="Square"/> on the <see cref="Board"/>
    /// </summary>
    public class Square
    {
        /// <summary>
        /// The <see cref="Enums.File"/> of the <see cref="Square"/>
        /// </summary>
        Enums.File file;
        /// <summary>
        /// The <see cref="Enums.Rank"/> of the <see cref="Square"/>
        /// </summary>
        Enums.Rank rank;
        /// <summary>
        /// The <see cref="Piece"/> (if any) on the <see cref="Square"/>
        /// </summary>
        Piece piece;
        /// <summary>
        /// The <see cref="Enums.SquareType"/> of the <see cref="Square"/>
        /// </summary>
        Enums.SquareType type;
        /// <summary>
        /// A <see cref="Square"/> <see cref="Array"/> of the neighbours of the <see cref="Square"/>
        /// </summary>
        Square[] neighbours = new Square[4];

        /// <summary>
        /// Blank Constructor
        /// </summary>
        public Square()
        {

        }

        /// <summary>
        /// Main Constructor
        /// </summary>
        /// <param name="rank">The <see cref="rank"/> of this <see cref="Square"/> on the <see
cref="Board"/>.</param>
        /// <param name="file">The <see cref="file"/> of this <see cref="Square"/> on the <see
cref="Board"/>.</param>
        /// <param name="piece">The <see cref="Piece"/> (if any) occupying this <see
cref="Square"/>.</param>
        /// <param name="type">The <see cref="type"/> of this <see cref="Square"/>.</param>
        public Square(Enums.Rank rank, Enums.File file, Piece piece, Enums.SquareType type)
        {
            setLocation(rank, file);
            setPiece(piece);
            setType(type);
        }

        /// <summary>
        /// Copy Constructor
        /// </summary>
        /// <param name="previousSquare">The <see cref="Square"/> to copy</param>
        public Square(Square previousSquare)
        {
            file = previousSquare.file;
            rank = previousSquare.rank;
            piece = new Piece(previousSquare.getPiece());
            type = previousSquare.type;
            neighbours = previousSquare.neighbours;
        }

        /// <summary>
        /// Returns the <see cref="Enums.Rank"/> of the <see cref="Square"/> on the <see
cref="Board"/>.
        /// </summary>
        /// <returns><see cref="rank"/></returns>
        public Enums.Rank getRank()
        {
            return rank;
        }

        /// <summary>
        /// Returns the <see cref="Enums.File"/> of the <see cref="Square"/> on the <see
cref="Board"/>.
        /// </summary>
        /// <returns><see cref="file"/></returns>
        public Enums.File getFile()
        {
            return file;
        }

        /// <summary>
        /// Returns the <see cref="Piece"/> occupying the <see cref="Square"/>.
        /// </summary>
        /// <returns><see cref="piece"/></returns>
        public Piece getPiece()
        {
            if (piece != null)
            {
                return piece;
            }
            else return null;

        }

        /// <summary>
        /// Returns the <see cref="Enums.SquareType"/> of <see cref="Square"/>.
        /// </summary>
        /// <returns><see cref="type"/></returns>
        public Enums.SquareType getSquareType()
        {
            return type;
        }

        /// <summary>
        /// Returns a <see cref="Tuple{T1, T2}"/> with the <see cref="rank"/> and <see cref="file"/>
of the <see cref="Square"/>
        /// </summary>
        /// <returns>A <see cref="Tuple{T1, T2}"/> with the <see cref="rank"/> and <see cref="file"/>
of the <see cref="Square"/></returns>
        public Tuple<Enums.Rank, Enums.File> getLocation()
        {
            return new Tuple<Enums.Rank, Enums.File>(rank, file);
        }

        /// <summary>
        /// Returns an <see cref="Array"/> containing the neighbours of the <see cref="Square"/>.
        /// </summary>
        /// <returns><see cref="neighbours"/></returns>
        public Square[] getNeighbours()
        {
            return neighbours;
        }

        /// <summary>
        /// Sets the <see cref="rank"/> of the <see cref="Square"/>
        /// </summary>
        /// <param name="rank"><see cref="rank"/></param>
        public void setRank(Enums.Rank rank)
        {
            this.rank = rank;
        }

        /// <summary>
        /// Sets the <see cref="file"/> of the <see cref="Square"/>
        /// </summary>
        /// <param name="file"><see cref="file"/></param>
        public void setFile(Enums.File file)
        {
            this.file = file;
        }

        /// <summary>
        /// Sets the <see cref="rank"/> and <see cref="file"/> of the <see cref="Square"/>
        /// </summary>
        /// <param name="rank"><see cref="rank"/></param>
        /// <param name="file"><see cref="file"/></param>
        public void setLocation(Enums.Rank rank, Enums.File file)
        {
            setRank(rank);
            setFile(file);
        }

        /// <summary>
        /// Sets the <see cref="Piece"/> occupying the <see cref="Square"/>.
        /// </summary>
        /// <param name="piece"><see cref="piece"/></param>
        public void setPiece(Piece piece)
        {
            this.piece = piece;
        }

        /// <summary>
        /// Sets the <see cref="type"/> of the <see cref="Square"/>.
        /// </summary>
        /// <param name="type"><see cref="type"/></param>
        public void setType(Enums.SquareType type)
        {
            this.type = type;
        }

        /// <summary>
        /// Sets the neighbour squares of the <see cref="Square"/>
        /// </summary>
        /// <param name="leftSquare">The left neighbour <see cref="Square"/></param>
        /// <param name="rightSquare">The right neighbour <see cref="Square"/></param>
        /// <param name="topSquare">The top neighbour <see cref="Square"/></param>
        /// <param name="bottomSquare">The bottom neighbour <see cref="Square"/></param>
        public void setNeighbours(Square leftSquare, Square rightSquare, Square topSquare, Square
bottomSquare)
        {
            neighbours[0] = leftSquare;
            neighbours[1] = rightSquare;
            neighbours[2] = topSquare;
            neighbours[3] = bottomSquare;
        }

        /// <summary>
        /// ToString method
        /// </summary>
        /// <returns>A String representation of the <see cref="Square"/></returns>
        public override string ToString()
        {
            string theString = "";

            theString += "\n | Location  : " + getFile() + (int)getRank();
            if (getPiece() != null)
            {
                theString += "\n | Piece : " + getPiece().ToString();
            }
            else
            {
                theString += "\n | Piece : null ";
            }
            theString += "\n | Type  : " + getSquareType().ToString();

            return theString;
        }
    }
}
```

2.1.5. Board.cs

2.1.5.1. Description

Represents the game board in a game of Viking Chess

2.1.5.2. Code

```
using System;
using System.Linq;

namespace VikingChess.Model
{
    /// <summary>
    /// The <see cref="Board"/> class represents the play board for the game.
    /// </summary>
    public class Board
    {
        /// <summary>
        /// The array holding the board data
        /// </summary>
        private Square[,] board;
        /// <summary>
        /// The size of the board.
        /// </summary>
        private int size;
        /// <summary>
        /// The player who moves next
        /// </summary>
        private Player currentPlayer;
        /// <summary>
        /// The type of <see cref="Game"/> this is.
        /// </summary>
        private String gameType = null;
        /// <summary>
        /// The number of black pieces on this board.
        /// </summary>
        private int blackPieces = 0;
        /// <summary>
        /// The number of white pieces on this board.
        /// </summary>
        private int whitePieces = 0;

        /// <summary>
        /// Constructor. Makes a new board object of the specified <see cref="size"/>
        /// </summary>
        /// <param name="size">The size of the <see cref="Board"/>.</param>
        /// <param name="gameType"><see cref="gameType"/></param>
        public Board(int size, String gameType)
        {
            this.size = size + 2;
            board = new Square[this.size, this.size];
            this.gameType = gameType;
            InitializeBoard();
            SetPieces();
            SetSpecialSquares();
        }

        /// <summary>
        /// Copy Constructor
        /// </summary>
        /// <param name="prevBoard">The <see cref="Board"/> to copy.</param>
        public Board(Board prevBoard)
        {
            this.size = prevBoard.GetSize();
            this.board = new Square[this.size, this.size];
            this.currentPlayer = new Player(prevBoard.getCurrentPlayer());
            this.gameType = prevBoard.gameType;
            InitializeBoard();
            for(int i=1; i < size-1; ++i)
            {
                for(int j=1; j < size-1; ++j)
                {
                    if(prevBoard.GetSquare(i, j).getPiece() != null)
                    {
                        board[i, j].setPiece(new Piece(prevBoard.GetSquare(i, j).getPiece()));
                    }
                    else
                    {
                        board[i, j].setPiece(null);
                    }
                }
            }
            blackPieces = prevBoard.countPieces(Enums.Color.BLACK);
            whitePieces = prevBoard.countPieces(Enums.Color.WHITE);
            SetSpecialSquares();
        }

        /// <summary>
        /// Decrements the <see cref="blackPieces"/> on this board.
        /// </summary>
        public void decrementBlackCount()
        {
            --blackPieces;
        }

        /// <summary>
        /// Decrements the <see cref="whitePieces"/> on this board.
        /// </summary>
        public void decrementWhiteCount()
        {
            --whitePieces;
        }

        /// <summary>
        /// Creates new <see cref="Square"/> objects at the appropriate location in the board array
and defines the neighbours
        /// for each square.
        /// </summary>
        public void InitializeBoard()
        {
            Enums.Rank[] ranks = (Enums.Rank[])Enum.GetValues(typeof(Enums.Rank));
            Enums.File[] files = (Enums.File[])Enum.GetValues(typeof(Enums.File));
            Enums.SquareType[] squareTypes =
(Enums.SquareType[])Enum.GetValues(typeof(Enums.SquareType));
            int r;
            int f = 0;

            if (gameType.Equals("Hnefatafl"))
            {
                r = ranks.Count();
                blackPieces = 24;
                whitePieces = 12;
            }
            else if (gameType.Equals("Brandubh"))
            {
                r = ranks.Count() - 4;
                blackPieces = 8;
                whitePieces = 4;
            }
            else if (gameType.Equals("Ard Ri"))
            {
                r = ranks.Count() - 4;
                blackPieces = 16;
                whitePieces = 8;
            }
            else if (gameType.Equals("Tablut"))
            {
                r = ranks.Count() - 2;
                blackPieces = 16;
                whitePieces = 8;
            }
            else if (gameType.Equals("Tawlbwrdd"))
            {
                r = ranks.Count();
                blackPieces = 24;
                whitePieces = 12;
            }
            else
            {
                r = ranks.Count();
            }

            for (int i = 0; i < size; ++i)
            {
                for (int j = 0; j < size; ++j)
                {
                    // For border squares of the array, set the squares to null
                    if ((i == 0) || (j == 0) || (i == size - 1) || (j == size - 1))
                    {
                        SetSquare(null, i, j);
                    }
                    // Otherwise, create a new square at the specified location
                    else
                    {
                        SetSquare(new Square(ranks[r], files[f], null, squareTypes[2]), i, j);
                        ++f;
                    }
                }
                f = 0;
                --r;
            }

            // Set the neighbours for each square
            for (int i = 0; i < size; ++i)
            {
                for (int j = 0; j < size; ++j)
                {
                    if ((i == 0) || (j == 0) || (i == size - 1) || (j == size - 1))
                    {

                    }
                    else
                    {
                        SetNeighbours(board[i, j]);
                    }
                }
            }
        }

        /// <summary>
        /// Sets the current <see cref="Player"/> for this <see cref="Board"/>.
        /// </summary>
        /// <param name="player">The <see cref="Player"/> to set as the <see cref="currentPlayer"/> of
this <see cref="Board"/>.</param>
        public void setCurrentPlayer(Player player)
        {
            currentPlayer = player;
        }

        /// <summary>
        /// gets the current <see cref="Player"/> for this <see cref="Board"/>
        /// </summary>
        /// <returns><see cref="currentPlayer"/></returns>
        public Player getCurrentPlayer()
        {
            return currentPlayer;
        }

        /// <summary>
        /// Returns the <see cref="gameType"/> of this <see cref="Board"/>
        /// </summary>
        /// <returns><see cref="gameType"/></returns>
        public String getGameType()
        {
            return gameType;
        }

        /// <summary>
        /// Gets the total <see cref="Piece"/> count of a particular <see cref="Enums.Color"/> on this
<see cref="Board"/>
        /// </summary>
        /// <param name="color">The <see cref="Enums.Color"/> to count for.</param>
        /// <returns>The <see cref="Piece"/> count of the passed in <see cref="Enums.Color"/> for this
<see cref="Board"/>.</returns>
        public int countPieces(Enums.Color color)
        {
            int count = 0;

            for (int i = 0; i < size; ++i)
            {
                for (int j = 0; j < size; ++j)
                {
                    if ((i == 0) || (j == 0) || (i == size - 1) || (j == size - 1))
                    {

                    }
                    else
                    {
                        if(board[i,j].getPiece() != null && board[i,
j].getPiece().getColor().Equals(color))
                        {
                            count++;
                        }
                    }
                }
            }
            return count;
        }

        /// <summary>
        /// Returns the array containing the board data.
        /// </summary>
        /// <returns><see cref="board"/></returns>
        public Square[,] GetBoard()
        {
            return board;
        }

        /// <summary>
        /// Returns the index position in the <see cref="board"/> array corresponding to the passed in
rankfile location.
        /// </summary>
        /// <param name="location">A <see cref="Tuple"/> containing the <see cref="Enums.Rank"/> and
<see cref="Enums.File"/> of a square on the board.</param>
        /// <returns>A <see cref="Tuple"/> containing the index location of a position on the
board.</returns>
        public Tuple<int, int> GetPosition(Tuple<Enums.Rank,Enums.File> location)
        {
            int i = 0;
            int j = 0;
            Tuple<int, int> position = new Tuple<int, int>(0, 0);
            for(i = 1; i < size-1; ++i)
            {
                for (j = 1; j < size - 1; ++j)
                {
                    if (board[i, j].getLocation().Equals(location))
                    {
                        position = new Tuple<int, int>(i, j);
                        return position;
                    }
                }
            }
            return position;
        }

        /// <summary>
        /// Returns the <see cref="Square"/> at the specified location
        /// </summary>
        /// <param name="i">The row that the <see cref="Square"/> is on.</param>
        /// <param name="j">The column that the <see cref="Square"/> is on.</param>
        /// <returns></returns>
        public Square GetSquare(int i, int j)
        {
            return board[i, j];
        }

        /// <summary>
        /// Sets the passed in location in the <see cref="board"/> array to the <see cref="Square"/>
passed in.
        /// </summary>
        /// <param name="square">The <see cref="Square"/> object to set the location to.</param>
        /// <param name="i">The row of the <see cref="board"/> array.</param>
        /// <param name="j">The column of the <see cref="board"/> array.</param>
        public void SetSquare(Square square, int i, int j)
        {
            board[i, j] = square;
        }

        /// <summary>
        /// Sets and defines the neighbour locations of a <see cref="Square"/> in the <see
cref="board"/> array.
        /// </summary>
        /// <param name="square">The <see cref="Square"/> to set the neighbours for.</param>
        public void SetNeighbours(Square square)
        {
            Tuple<int, int> squarePosition = GetPosition(square.getLocation());
            Square leftSquare;
            Square rightSquare;
            Square topSquare;
            Square bottomSquare;

            if(squarePosition.Item2-1 > 0)
            {
                leftSquare = GetSquare(squarePosition.Item1, squarePosition.Item2 - 1);
            }
            else
            {
                leftSquare = null;
            }

            if((squarePosition.Item2 + 1 < size - 1)){
                rightSquare = GetSquare(squarePosition.Item1, squarePosition.Item2 + 1);
            }
            else
            {
                rightSquare = null;
            }

            if ((squarePosition.Item1 - 1 > 0))
            {
                topSquare = GetSquare(squarePosition.Item1 - 1, squarePosition.Item2);
            }
            else
            {
                topSquare = null;
            }

            if ((squarePosition.Item1 + 1 < size - 1))
            {
                bottomSquare = GetSquare(squarePosition.Item1 +1, squarePosition.Item2);
            }
            else
            {
                bottomSquare = null;
            }

            square.setNeighbours(leftSquare, rightSquare, topSquare, bottomSquare);
        }

        /// <summary>
        /// Sets the special <see cref="Enums.SquareType"/>s of a <see cref="Square"/> on the <see
cref="Board"/>.
        /// </summary>
        public void SetSpecialSquares()
        {
            // Set Corners
            board[1, 1].setType(Enums.SquareType.CORNER);
            board[1, size - 2].setType(Enums.SquareType.CORNER);
            board[size - 2, 1].setType(Enums.SquareType.CORNER);
            board[size - 2, size - 2].setType(Enums.SquareType.CORNER);

            // Set Throne
            board[(size / 2), (size / 2)].setType(Enums.SquareType.THRONE);
        }

        /// <summary>
        /// Returns the size of the <see cref="board"/> array.
        /// </summary>
        /// <returns></returns>
        public int GetSize()
        {
            return size;
        }

        /// <summary>
        /// Sets the position of the <see cref="Piece"/> objects on the <see cref="Board"/> depending
on the gametype
        /// </summary>
        public void SetPieces()
        {
            switch (gameType)
            {
                case "Brandubh":
                    setBrandubhPieces();
                    break;
                case "Hnefatafl":
                    setHnefataflPieces();
                    break;
                case "Ard Ri":
                    setArdRiPieces();
                    break;
                case "Tablut":
                    setTablutPieces();
                    break;
                case "Tawlbwrdd":
                    setTawlbwrddPieces();
                    break;
                default:
                    break;
            }
        }

        /// <summary>
        /// Sets the starting position for the Hnefatafl pieces.
        /// </summary>
        private void setHnefataflPieces()
        {
            // Top Pawns Black
            board[1, 4].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[1,
4].getRank(), board[1, 4].getFile()));
            board[1, 5].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[1,
5].getRank(), board[1, 5].getFile()));
            board[1, 6].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[1,
6].getRank(), board[1, 6].getFile()));
            board[1, 7].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[1,
7].getRank(), board[1, 7].getFile()));
            board[1, 8].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[1,
8].getRank(), board[1, 8].getFile()));
            board[2, 6].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[2,
6].getRank(), board[2, 6].getFile()));
```

```
            // Left Pawns Black
            board[4, 1].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[4,
1].getRank(), board[4, 1].getFile()));
            board[5, 1].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[5,
1].getRank(), board[5, 1].getFile()));
            board[6, 1].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[6,
1].getRank(), board[6, 1].getFile()));
            board[7, 1].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[7,
1].getRank(), board[7, 1].getFile()));
            board[8, 1].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[8,
1].getRank(), board[8, 1].getFile()));
            board[6, 2].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[6,
2].getRank(), board[6, 2].getFile()));

            // Bottom Pawns Black
            board[11, 4].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[11,
4].getRank(), board[11, 4].getFile()));
            board[11, 5].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[11,
5].getRank(), board[11, 5].getFile()));
            board[11, 6].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[11,
6].getRank(), board[11, 6].getFile()));
            board[11, 7].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[11,
7].getRank(), board[11, 7].getFile()));
            board[11, 8].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[11,
8].getRank(), board[11, 8].getFile()));
            board[10, 6].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[10,
6].getRank(), board[10, 6].getFile()));

            // Right Pawns Black
            board[4, 11].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[4,
11].getRank(), board[4, 11].getFile()));
            board[5, 11].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[5,
11].getRank(), board[5, 11].getFile()));
            board[6, 11].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[6,
11].getRank(), board[6, 11].getFile()));
            board[7, 11].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[7,
11].getRank(), board[7, 11].getFile()));
            board[8, 11].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[8,
11].getRank(), board[8, 11].getFile()));
            board[6, 10].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[6,
10].getRank(), board[6, 10].getFile()));

            // White Pawns
            board[4, 6].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[4,
6].getRank(), board[4, 6].getFile()));
            board[5, 5].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[5,
5].getRank(), board[5, 5].getFile()));
            board[5, 6].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[5,
6].getRank(), board[5, 6].getFile()));
            board[5, 7].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[5,
7].getRank(), board[5, 7].getFile()));
            board[6, 4].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[6,
4].getRank(), board[6, 4].getFile()));
            board[6, 5].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[6,
5].getRank(), board[6, 5].getFile()));
            board[6, 7].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[6,
7].getRank(), board[6, 7].getFile()));
            board[6, 8].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[6,
8].getRank(), board[6, 8].getFile()));
            board[7, 5].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[7,
5].getRank(), board[7, 5].getFile()));
            board[7, 6].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[7,
6].getRank(), board[7, 6].getFile()));
            board[7, 7].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[7,
7].getRank(), board[7, 7].getFile()));
            board[8, 6].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[8,
6].getRank(), board[8, 6].getFile()));

            // White King
            board[6, 6].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.KING, board[6,
6].getRank(), board[6, 6].getFile()));
        }

        /// <summary>
        /// Sets the starting position for the Tawlbwrdd pieces.
        /// </summary>
        private void setTawlbwrddPieces()
        {
            // Top Pawns Black
            board[2, 5].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[2,
5].getRank(), board[2, 5].getFile()));
            board[1, 5].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[1,
5].getRank(), board[1, 5].getFile()));
            board[1, 6].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[1,
6].getRank(), board[1, 6].getFile()));
            board[1, 7].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[1,
7].getRank(), board[1, 7].getFile()));
            board[2, 7].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[2,
7].getRank(), board[2, 7].getFile()));
            board[3, 6].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[3,
6].getRank(), board[3, 6].getFile()));

            // Left Pawns Black
            board[5, 2].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[5,
2].getRank(), board[5, 2].getFile()));
            board[5, 1].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[5,
1].getRank(), board[5, 1].getFile()));
            board[6, 1].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[6,
1].getRank(), board[6, 1].getFile()));
            board[7, 1].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[7,
1].getRank(), board[7, 1].getFile()));
            board[7, 2].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[7,
2].getRank(), board[7, 2].getFile()));
            board[6, 3].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[6,
3].getRank(), board[6, 3].getFile()));

            // Bottom Pawns Black
            board[10, 5].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[10,
5].getRank(), board[10, 5].getFile()));
            board[11, 5].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[11,
5].getRank(), board[11, 5].getFile()));
            board[11, 6].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[11,
6].getRank(), board[11, 6].getFile()));
            board[11, 7].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[11,
7].getRank(), board[11, 7].getFile()));
            board[10, 7].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[10,
7].getRank(), board[10, 7].getFile()));
            board[9, 6].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[9,
6].getRank(), board[9, 6].getFile()));

            // Right Pawns Black
            board[5, 10].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[5,
10].getRank(), board[5, 10].getFile()));
            board[5, 11].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[5,
11].getRank(), board[5, 11].getFile()));
            board[6, 11].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[6,
11].getRank(), board[6, 11].getFile()));
            board[7, 11].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[7,
11].getRank(), board[7, 11].getFile()));
            board[7, 10].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[7,
10].getRank(), board[7, 10].getFile()));
            board[6, 9].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[6,
9].getRank(), board[6, 9].getFile()));

            // White Pawns
            board[4, 6].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[4,
6].getRank(), board[4, 6].getFile()));
            board[5, 5].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[5,
5].getRank(), board[5, 5].getFile()));
            board[5, 6].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[5,
6].getRank(), board[5, 6].getFile()));
            board[5, 7].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[5,
7].getRank(), board[5, 7].getFile()));
            board[6, 4].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[6,
4].getRank(), board[6, 4].getFile()));
            board[6, 5].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[6,
5].getRank(), board[6, 5].getFile()));
```

```
            board[6, 7].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[6,
7].getRank(), board[6, 7].getFile()));
            board[6, 8].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[6,
8].getRank(), board[6, 8].getFile()));
            board[7, 5].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[7,
5].getRank(), board[7, 5].getFile()));
            board[7, 6].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[7,
6].getRank(), board[7, 6].getFile()));
            board[7, 7].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[7,
7].getRank(), board[7, 7].getFile()));
            board[8, 6].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[8,
6].getRank(), board[8, 6].getFile()));

            // White King
            board[6, 6].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.KING, board[6,
6].getRank(), board[6, 6].getFile()));
        }

        /// <summary>
        /// Sets the starting position for the Tablut pieces.
        /// </summary>
        private void setTablutPieces()
        {
            // Top Pawns Black
            board[1, 4].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[1,
4].getRank(), board[1, 4].getFile()));
            board[1, 5].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[1,
5].getRank(), board[1, 5].getFile()));
            board[1, 6].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[1,
6].getRank(), board[1, 6].getFile()));
            board[2, 5].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[2,
5].getRank(), board[2, 5].getFile()));

            // Left Pawns Black
            board[4, 1].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[4,
1].getRank(), board[4, 1].getFile()));
            board[5, 1].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[5,
1].getRank(), board[5, 1].getFile()));
            board[6, 1].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[6,
1].getRank(), board[6, 1].getFile()));
            board[5, 2].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[5,
2].getRank(), board[5, 2].getFile()));

            // Bottom Pawns Black
            board[8, 5].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[8,
5].getRank(), board[8, 5].getFile()));
            board[9, 4].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[9,
4].getRank(), board[9, 4].getFile()));
            board[9, 5].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[9,
5].getRank(), board[9, 5].getFile()));
            board[9, 6].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[9,
6].getRank(), board[9, 6].getFile()));

            // Right Pawns Black
            board[5, 8].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[5,
8].getRank(), board[5, 8].getFile()));
            board[4, 9].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[4,
9].getRank(), board[4, 9].getFile()));
            board[5, 9].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[5,
9].getRank(), board[5, 9].getFile()));
            board[6, 9].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[6,
9].getRank(), board[6, 9].getFile()));

            // White Pawns
            board[3, 5].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[3,
5].getRank(), board[3, 5].getFile()));
            board[4, 5].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[4,
5].getRank(), board[4, 5].getFile()));
            board[5, 3].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[5,
3].getRank(), board[5, 3].getFile()));
            board[5, 4].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[5,
4].getRank(), board[5, 4].getFile()));
            board[5, 6].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[5,
6].getRank(), board[5, 6].getFile()));
```

```
            board[5, 7].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[5,
7].getRank(), board[5, 7].getFile()));
            board[6, 5].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[6,
5].getRank(), board[6, 5].getFile()));
            board[7, 5].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[7,
5].getRank(), board[7, 5].getFile()));

            // White King
            board[5, 5].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.KING, board[5,
5].getRank(), board[5, 5].getFile()));
        }

        /// <summary>
        /// Sets the starting position for the Brandubh pieces.
        /// </summary>
        private void setBrandubhPieces()
        {
            // Top Pawns Black
            board[1, 4].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[1,
4].getRank(), board[1, 4].getFile()));
            board[2, 4].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[2,
4].getRank(), board[2, 4].getFile()));

            // Left Pawns Black
            board[4, 1].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[4,
1].getRank(), board[4, 1].getFile()));
            board[4, 2].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[4,
2].getRank(), board[4, 2].getFile()));

            // Right Pawns Black
            board[4, 7].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[4,
7].getRank(), board[4, 7].getFile()));
            board[4, 6].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[4,
6].getRank(), board[4, 6].getFile()));

            // Bottom Pawns Black
            board[7, 4].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[7,
4].getRank(), board[7, 4].getFile()));
            board[6, 4].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[6,
4].getRank(), board[6, 4].getFile()));

            // White Pawns
            board[4, 3].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[4,
3].getRank(), board[4, 3].getFile()));
            board[4, 5].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[4,
5].getRank(), board[4, 5].getFile()));
            board[3, 4].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[3,
4].getRank(), board[3, 4].getFile()));
            board[5, 4].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[5,
4].getRank(), board[5, 4].getFile()));

            // White King
            board[4, 4].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.KING, board[4,
4].getRank(), board[4, 4].getFile()));
        }

        /// <summary>
        /// Sets the starting position for the Ard Rí pieces.
        /// </summary>
        private void setArdRiPieces()
        {
            // Top Pawns Black
            board[1, 3].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[1,
3].getRank(), board[1, 3].getFile()));
            board[1, 4].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[1,
4].getRank(), board[1, 4].getFile()));
            board[1, 5].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[1,
5].getRank(), board[1, 5].getFile()));
            board[2, 4].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[2,
4].getRank(), board[2, 4].getFile()));

            // Left Pawns Black
            board[3, 1].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[3,
1].getRank(), board[3, 1].getFile()));
```

```
            board[4, 1].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[4,
1].getRank(), board[4, 1].getFile()));
            board[5, 1].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[5,
1].getRank(), board[5, 1].getFile()));
            board[4, 2].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[4,
2].getRank(), board[4, 2].getFile()));

            // Right Pawns Black
            board[3, 7].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[3,
7].getRank(), board[3, 7].getFile()));
            board[4, 6].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[4,
6].getRank(), board[4, 6].getFile()));
            board[5, 7].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[5,
7].getRank(), board[5, 7].getFile()));
            board[4, 7].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[4,
7].getRank(), board[4, 7].getFile()));

            // Bottom Pawns Black
            board[7, 3].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[7,
3].getRank(), board[7, 3].getFile()));
            board[7, 4].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[7,
4].getRank(), board[7, 4].getFile()));
            board[7, 5].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[7,
5].getRank(), board[7, 5].getFile()));
            board[6, 4].setPiece(new Piece(Enums.Color.BLACK, Enums.PieceType.PAWN, board[6,
4].getRank(), board[6, 4].getFile()));

            // White Pawns
            board[4, 3].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[4,
3].getRank(), board[4, 3].getFile()));
            board[4, 5].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[4,
5].getRank(), board[4, 5].getFile()));
            board[3, 3].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[3,
3].getRank(), board[3, 3].getFile()));
            board[3, 4].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[3,
4].getRank(), board[3, 4].getFile()));
            board[3, 5].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[3,
5].getRank(), board[3, 5].getFile()));
            board[5, 3].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[5,
3].getRank(), board[5, 3].getFile()));
            board[5, 4].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[5,
4].getRank(), board[5, 4].getFile()));
            board[5, 5].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.PAWN, board[5,
5].getRank(), board[5, 5].getFile()));

            // White King
            board[4, 4].setPiece(new Piece(Enums.Color.WHITE, Enums.PieceType.KING, board[4,
4].getRank(), board[4, 4].getFile()));
        }

        /// <summary>
        /// Prints a representation of the <see cref="board"/>
        /// </summary>
        /// <returns>A <see cref="string"/> <see cref="Array"/> representing the <see cref="board"/>
</returns>
        public string[] PrintBoard()
        {
            string[] boardString = new string[size];
            int k = 0;
            for(int i = 0; i < size; i++)
            {
                boardString[k] = "";
                for(int j = 0; j < size; j++)
                {

                    if(board[i,j] == null)
                    {
                        boardString[k] += "|---";
                    }
                    else
                    {
                        if (board[i, j].getSquareType() == Enums.SquareType.CORNER)
                        {
                            boardString[k] += "|-X-";
                        }
                        else if(board[i,j].getSquareType() == Enums.SquareType.REGULAR)
                        {
                            if(board[i, j].getPiece() != null)
                            {
                                if(board[i, j].getPiece().getColor().Equals(Enums.Color.BLACK))
                                {
                                    boardString[k] += "|-B-";
                                }
                                else if (board[i, j].getPiece().getColor().Equals(Enums.Color.WHITE))
                                {
                                    boardString[k] += "|-W-";
                                }

                            }
                            else
                            {
                                boardString[k] += "|-0-";
                            }
                        }
                        else if(board[i,j].getSquareType() == Enums.SquareType.THRONE)
                        {
                            if (board[i, j].getPiece() != null)
                            {
                                boardString[k] += "|-K-";
                            }
                            else
                            {
                                boardString[k] += "|-T-";
                            }
                        }
                    }

                }
                boardString[k] += "|";
                boardString[k] += "\n";
                k++;
            }
            return boardString;
        }

        /// <summary>
        /// A <see cref="String"/> representation of this <see cref="Board"/>.
        /// </summary>
        /// <returns>A <see cref="String"/> representation of this <see cref="Board"/>.</returns>
        public override String ToString()
        {
            String theString = "";
            int size = GetSize();

            theString += "\n | Size : " + size;
            theString += "\n | Current Player  : " + getCurrentPlayer().getColor();
            theString += "\n | Game Type  : " + getGameType();
            theString += "\n | Black Pieces : " + countPieces(Enums.Color.BLACK);
            theString += "\n | White Pieces : " + countPieces(Enums.Color.WHITE);

            return theString;
        }
    }
}
```

2.1.6. Player.cs

2.1.6.1. Description

Represents a Player in a game of Viking Chess.

2.1.6.2. Code

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace VikingChess.Model
{
    /// <summary>
    /// The <see cref="Player"/> class represents a player in the <see cref="Game"/>
    /// </summary>
    public class Player
    {
        /// <summary>
        /// The <see cref="Enums.Color"/> of the <see cref="Player"/>
        /// </summary>
        private Enums.Color color;
        /// <summary>
        /// A <see cref="List{T}"/> containing the moves made by the player over the course of a <see
cref="Game"/>
        /// </summary>
        private List<String> moveList;
        /// <summary>
        /// The name of the <see cref="Player"/>
        /// </summary>
        private string name = "";
        /// <summary>
        /// The <see cref="Enums.PlayerType"/> of the <see cref="Player"/>
        /// </summary>
        private Enums.PlayerType type;
        /// <summary>
        /// A <see cref="bool"/> representing whether or not the <see cref="Player"/> has won or not.
        /// </summary>
        private Boolean hasWon = false;

        #region Constructors

        /// <summary>
        /// Simple Constructor
        /// </summary>
        public Player()
        {
            moveList = new List<String>();
        }

        /// <summary>
        /// Constructs a new <see cref="Player"/> object and sets its color.
        /// </summary>
        /// <param name="color"><see cref="color"/></param>
        public Player(Enums.Color color, Enums.PlayerType type)
        {
            setColor(color);
            setType(type);
            moveList = new List<String>();
        }

        /// <summary>
        /// Main Constructor
        /// </summary>
        /// <param name="color"><see cref="color"/></param>
        /// <param name="name"><see cref="name"/></param>
        /// <param name="type"><see cref="type"/></param>
        public Player(Enums.Color color, string name, Enums.PlayerType type)
        {
            setColor(color);
            setType(type);
            setName(name);
            moveList = new List<String>();
        }

        /// <summary>
        /// Copy Constructor
        /// </summary>
        /// <param name="player">The <see cref="Player"/> to copy</param>
        public Player(Player player)
        {
            setColor(player.getColor());
            setType(player.getType());
            setName(player.getName());
            moveList = player.getMoveList();
            hasWon = player.hasWon;
        }

        #endregion

        #region Getters & Setters

        /// <summary>
        /// Returns the <see cref="Enums.Color"/> of the <see cref="Player"/>.
        /// </summary>
        /// <returns><see cref="color"/></returns>
        public Enums.Color getColor()
        {
            return color;
        }

        /// <summary>
        /// Returns the <see cref="name"/> of the <see cref="Player"/>.
        /// </summary>
        /// <returns><see cref="name"/></returns>
        public string getName()
        {
            return name;
        }

        /// <summary>
        /// Returns the <see cref="Enums.PlayerType"/> of the <see cref="Player"/>
        /// </summary>
        /// <returns><see cref="type"/></returns>
        public Enums.PlayerType getType()
        {
            return type;
        }

        /// <summary>
        /// Returns the win status of the <see cref="Player"/>
        /// </summary>
        /// <returns><see cref="hasWon"/></returns>
        public Boolean getWin()
        {
            return hasWon;
        }

        /// <summary>
        /// Returns the <see cref="moveList"/> of the <see cref="Player"/>.
        /// </summary>
        /// <returns><see cref="moveList"/></returns>
        public List<String> getMoveList()
        {
            return moveList;
        }

        /// <summary>
        /// Sets the <see cref="Enums.Color"/> of the <see cref="Player"/>.
        /// </summary>
        /// <param name="color"><see cref="color"/></param>
        public void setColor(Enums.Color color)
        {
            this.color = color;
        }

        /// <summary>
        /// Sets the <see cref="name"/> of the <see cref="Player"/>
        /// </summary>
        /// <param name="name"><see cref="name"/></param>
        public void setName(string name)
        {
            this.name = name;
        }
        /// <summary>
        /// Sets the <see cref="Enums.PlayerType"/> of <see cref="Player"/>
        /// </summary>
        /// <param name="type"><see cref="type"/></param>
        public void setType(Enums.PlayerType type)
        {
            this.type = type;
        }

        /// <summary>
        /// Sets the win status of the <see cref="Player"/>.
        /// </summary>
        /// <param name="win"><see cref="hasWon"/></param>
        public void setWin(bool win)
        {
            hasWon = win;
        }

        #endregion

        #region Miscellaneous methods

        /// <summary>
        /// Adds a new entry to the <see cref="moveList"/> of the <see cref="Player"/>
        /// </summary>
        /// <param name="moveNotation">A string containing the move notation</param>
        public void updateMoveList(string moveNotation)
        {
            moveList.Add(moveNotation);
        }

        /// <summary>
        /// Removes an entry from the <see cref="moveList"/> of the <see cref="Player"/>.
        /// </summary>
        public void removeFromMoveList()
        {
            moveList.RemoveAt(moveList.Count - 1);
        }

        /// <summary>
        /// Use the MCTS algorithm to make a move
        /// </summary>
        /// <param name="board">The <see cref="Board"/> to use for the move</param>
        /// <returns>A <see cref="Tuple{T1, T2, T3}"/> containing the origin <see cref="Square"/>,
destination <see cref="Square"/> and <see cref="Piece"/> to move.</returns>
        public Tuple<Square, Piece, Square> makeMCTSMove(Board board)
        {
            MCTS mctsTree = new MCTS(board);

            Node node = mctsTree.makeMove(board);
            Tuple<Square, Piece, Square> moveTuple = node.GetLastMove();
            return moveTuple;
        }


        /// <summary>
        /// Returns a <see cref="List{T}"/> containing all <see cref="Square"/> objects that have <see
cref="Piece"/> objects that
        /// can be moved.
        /// </summary>
        /// <param name="board">The <see cref="Board"/> to get all <see cref="Square"/> objects
from.</param>
        /// <returns><see cref="List{T}"/> of <see cref="Square"/> objects containing <see
cref="Piece"/> objects.</returns>
        public List<Square> GetAllPieceSquares(Board board)
        {
            List<Square> squareList = new List<Square>();

            // Loop through the board and get all squares with moveable pieces
            for (int i = 0; i < board.GetSize(); ++i)
            {
                for (int j = 0; j < board.GetSize(); ++j)
                {
                    Square currSq = board.GetSquare(i, j);
                    if (currSq != null && currSq.getPiece() != null)
                    {
                        Piece currPiece = currSq.getPiece();

                        if (currPiece.getColor().Equals(this.getColor()) &&
currPiece.generateLegalMoves(board).Count != 0)
                        {
                            squareList.Add(currSq);
                        }
                    }
                }
            }
            return squareList;
        }

        /// <summary>
        /// Select the King <see cref="Piece"/> from the <see cref="List{T}"/> of moveable pieces
        /// </summary>
        /// <param name="squareList">A <see cref="List{T}"/> cntaining all <see cref="Square"/>
objects that have <see cref="Piece"/> objects.</param>
        /// <param name="board">The <see cref="Board"/> to use.</param>
        /// <returns>The <see cref="Square"/> containing the King <see cref="Piece"/></returns>
        private Square SelectKingSquare(List<Square> squareList, Board board)
        {
            //Loop through all moveable pieces
            for (int i = 0; i < squareList.Count; ++i)
            {
                Square currSq = squareList[i] as Square;
                Piece currPiece = currSq.getPiece();

                if (currPiece.getType().Equals(Enums.PieceType.KING))
                {
                    List<Square> kingMoves = currPiece.generateLegalMoves(board);

                    for (int j = 0; j < kingMoves.Count; ++j)
                    {
                        Square moveSq = kingMoves[j] as Square;
                        Tuple<Enums.Rank,Enums.File> sqLoc = moveSq.getLocation();

                        for(int k = 1; k < board.GetSize()-2; ++k)
                        {
                            Square[,] boardArr = board.GetBoard();

                            // If the king can move to an edge square
                            if(boardArr[1,k].getLocation().Equals(sqLoc) ||
boardArr[k,1].getLocation().Equals(sqLoc) ||
                                boardArr[board.GetSize()-2, k].getLocation().Equals(sqLoc) ||
boardArr[k, board.GetSize() - 2].getLocation().Equals(sqLoc))
                            {
                                return currSq;
                            }
                        }
                    }
                }
            }
            return null;
        }

        /// <summary>
        /// Select a <see cref="Piece"/> that can capture another <see cref="Piece"/>.
        /// </summary>
        /// <param name="squareList"></param>
        /// <param name="board"></param>
        /// <returns></returns>
        private Square SelectCaptureSquare(List<Square> squareList, Board board)
        {
            // Select a piece that can capture another piece
            for (int i = 0; i < squareList.Count; ++i)
            {
                Square currSq = squareList[i] as Square;
                List<Square> currMoves = currSq.getPiece().generateLegalMoves(board);

                // Get the neighbours of the current square
                for (int j = 0; j < currMoves.Count; ++j)
                {
                    Square moveSquare = currMoves[j] as Square;
                    // Get the location of this square
                    Tuple<Enums.Rank, Enums.File> sqLoc = moveSquare.getLocation();

                    //Get the neighbours of the current move
                    Square[] neighbours = moveSquare.getNeighbours();

                    //Check the neighbouring squares for a piece of the opposite color
                    for (int k = 0; k < neighbours.Length; ++k)
                    {
                        Square neighbour = neighbours[k] as Square;
                        if (neighbour != null && neighbour.getPiece() != null)
                        {
                            //If the piece on the neighbouring square is different to the player's 
color
                            if (!(neighbour.getPiece().getColor().Equals(getColor())))
                            {
                                // Get the neighbours of this square
                                Square[] neighboursNeighbours = neighbour.getNeighbours();

                                for (int l = 0; l < neighboursNeighbours.Length; ++l)
                                {
                                    Square neighboursNeighbour = neighboursNeighbours[l] as Square;
                                    if (neighboursNeighbour != null && neighboursNeighbour.getPiece() 
!= null)
                                    {
                                        // If the location of the nieghboursNeighbour is different to 
the location of the current move square but has the same rank or file as the moveSquare
                                        if (!(neighboursNeighbour.getLocation().Equals(sqLoc)) && 
(neighboursNeighbour.getRank().Equals(moveSquare.getRank()) || 
neighboursNeighbour.getFile().Equals(moveSquare.getFile())))
                                            if 
(neighboursNeighbour.getPiece().getColor().Equals(getColor()))
                                                return currSq;

                                    }
                                    else if (getColor().Equals(Enums.Color.BLACK) && 
neighbour.getPiece().Equals(Enums.PieceType.KING))
                                    {
                                        return currSq;
                                    }

                                }
                            }
                        }
                    }
                }
            }
            return null;
        }

        /// <summary>
        /// Selects a <see cref="Square"/> containing a valid <see cref="Piece"/> for the CPU <see 
cref="Player"/>
        /// </summary>
        /// <param name="board">The current state of the <see cref="Board"/></param>
        /// <returns>A <see cref="Square"/> containing a valid <see cref="Piece"/> for the 
CPU</returns>
        public Square selectPieceSquare(Board board)
        {
            List<Square> squareList = new List<Square>();
            Square pieceSquare = null;

            // Get moveable pieces
            squareList = GetAllPieceSquares(board);

            if(squareList.Count == 0)
            {
                Debug.WriteLine("Error: squareList.Count == 0");
                return null;
            }
            if(pieceSquare == null)
            {
                pieceSquare = SelectKingSquare(squareList, board);
            }
            if(pieceSquare == null)
            {
                // Otherwise return a random square
                Random rnd = new Random();
                int index = rnd.Next(0, squareList.Count);
                pieceSquare = squareList[index] as Square;
            }
            return pieceSquare;
        }

        /// <summary>
        /// Returns a <see cref="Tuple{T1, T2, T3}"/> containing the origin <see cref="Square"/>,
destination <see cref="Square"/> and <see cref="Piece"/> necessary to move.
        /// </summary>
        /// <param name="board">The <see cref="Board"/> to use when obtaining the move tuple</param>
        /// <returns>A <see cref="Tuple{T1, T2, T3}"/> containing the origin <see cref="Square"/>,
destination <see cref="Square"/> and <see cref="Piece"/> necessary to move.</returns>
        public Tuple<Square,Piece,Square> getMoveTuple(Board board)
        {
            Square originSquare = selectPieceSquare(board);
            Piece selectedPiece = originSquare.getPiece();
            Square destinationSquare;

            List<Square> moveList = selectedPiece.generateLegalMoves(board);

            if (moveList.Count == 0)
            {
                Debug.WriteLine("Error: squareList.Count == 0");
                return null;
            }

            Tuple<Square, Piece, Square> moveTuple;

            // Otherwise, pick a random destination
            Random rnd = new Random();
            int index = rnd.Next(0, moveList.Count);
            destinationSquare = moveList[index] as Square;

            moveTuple = new Tuple<Square, Piece,
Square>(originSquare,selectedPiece,destinationSquare);

            return moveTuple;
        }

        /// <summary>
        /// Tries to return a <see cref="Tuple{T1, T2, T3}"/> containing an origin <see
cref="Square"/>, destination <see cref="Square"/> and <see cref="Piece"/>
        /// that will capture another <see cref="Piece"/>
        /// </summary>
        /// <param name="board">The <see cref="Board"/> to use.</param>
        /// <returns>A <see cref="Tuple{T1, T2, T3}"/> containing an origin <see cref="Square"/>,
destination <see cref="Square"/> and <see cref="Piece"/></returns>
        public Tuple<Square, Piece, Square> capturePiece(Board board)
        {
            Square originSquare = selectPieceSquare(board);
            Piece selectedPiece = originSquare.getPiece();
            Square destinationSquare;

            List<Square> moveList = selectedPiece.generateLegalMoves(board);
            Tuple<Square, Piece, Square> moveTuple;

            // If the selected piece can capture another piece, move to capture
            foreach (Square m in moveList.ToList())
            {
                Square moveSquare = new Square(m);
                // Get the location of this square
                Tuple<Enums.Rank, Enums.File> sqLoc = moveSquare.getLocation();

                //Get the neighbours of the current move
                Square[] neighbours = moveSquare.getNeighbours();
                //Check the neighbouring squares for a piece of the opposite color
                for (int j = 0; j < neighbours.Length; ++j)
                {
                    Square neighbour = neighbours[j] as Square;
                    if (neighbour != null && neighbour.getPiece() != null)
                    {
                        //If the piece on the neighbouring square is different to the player's color
                        if ( !neighbour.getPiece().getColor().Equals(getColor()) )
                        {
                            // Get the neighbours of this square
                            Square[] neighboursNeighbours = neighbour.getNeighbours();

                            for (int k = 0; k < neighboursNeighbours.Length; ++k)
                            {
                                Square neighboursNeighbour = neighboursNeighbours[k] as Square;
                                if (neighboursNeighbour != null && neighboursNeighbour.getPiece() !=
null)

                                {
                                    // If the location of the nieghboursNeighbour is different to the
location of the current move square
                                    if ((!neighboursNeighbour.getLocation().Equals(sqLoc)) && (
neighboursNeighbour.getRank().Equals(moveSquare.getRank()) ||
neighboursNeighbour.getFile().Equals(moveSquare.getFile()) ) )
                                    {
                                        if
(neighboursNeighbour.getPiece().getColor().Equals(getColor()))
                                        {
                                            destinationSquare = moveSquare;
                                            moveTuple = new Tuple<Square, Piece, Square>(originSquare,
selectedPiece, destinationSquare);

                                            return moveTuple;
                                        }
                                    }
                                }
                               else if (getColor().Equals(Enums.Color.BLACK) &&
neighbour.getPiece().Equals(Enums.PieceType.KING))
                                {
                                    destinationSquare = moveSquare;
                                    moveTuple = new Tuple<Square, Piece, Square>(originSquare,
selectedPiece, destinationSquare);

                                    return moveTuple;
                                }

                            }
                        }
                    }
                }
            }
            return null;
        }

        /// <summary>
        /// Tries to return a <see cref="Tuple{T1, T2, T3}"/> containing an origin <see
cref="Square"/>, destination <see cref="Square"/> and <see cref="Piece"/>
        /// that will capture the King <see cref="Piece"/>
        /// </summary>
        /// <param name="board">The <see cref="Board"/> to use.</param>
        /// <param name="squareList">A <see cref="List{T}"/> containing all <see cref="Square"/>
objects that have <see cref="Piece"/> objects on them.</param>
        /// <returns>A <see cref="Tuple{T1, T2, T3}"/> containing an origin <see cref="Square"/>,
destination <see cref="Square"/> and <see cref="Piece"/></returns>
        public Tuple<Square, Piece, Square> captureKing(Board board, List<Square> squareList)
        {
            // Check if the King can be captured
            for (int i = 1; i < board.GetSize()-2; ++i)
            {
                for (int j = 1; j < board.GetSize()-2; ++j)
                {
                    Square currSquare = board.GetSquare(i, j);

                    if (currSquare != null)
                    {
                        Piece currPiece = currSquare.getPiece();
                        if (currPiece != null && currPiece.getType().Equals(Enums.PieceType.KING))
                        {
                            Square kingSquare = currSquare;
                            Tuple<int, int> kingIndex = board.GetPosition(new Tuple<Enums.Rank,
Enums.File>(kingSquare.getRank(), kingSquare.getFile()));
                            Square[] kingNeighbours = kingSquare.getNeighbours();
                            List<Square> kingMoves = kingSquare.getPiece().generateLegalMoves(board);

                            for (int k = 0; k < squareList.Count; ++k)
                            {
                                Square selectedSquare = squareList[k] as Square;
                                Piece selectedPiece = selectedSquare.getPiece();
                                List<Square> moves = selectedPiece.generateLegalMoves(board);
                                if (selectedPiece.getColor().Equals(Enums.Color.BLACK))
                                {
                                    Square blockingSquare = null;

                                    // First see if the King can escape and find the square to block
it.
                                    for (int l = 0; l < kingMoves.Count; ++l)
                                    {
                                        for (int m = 1; m < board.GetSize() - 2; ++m)
                                        {
                                            Square currMove = kingMoves[l] as Square;
                                            // Top
                                            if (board.GetSquare(1, m).Equals(currMove))
                                            {
                                                blockingSquare = board.GetSquare(kingIndex.Item1 - 1,
m);
                                            }
                                            // Left
                                            else if (board.GetSquare(m, 1).Equals(currSquare))
                                            {
                                                blockingSquare = board.GetSquare(m, kingIndex.Item2 -
1);
                                            }
                                            // Bottom
                                            else if (board.GetSquare(board.GetSize() - 2,
m).Equals(currSquare))
                                            {
                                                blockingSquare = board.GetSquare(kingIndex.Item1 + 1,
m);
                                            }
                                            // Right
                                            else if (board.GetSquare(m, board.GetSize() -
2).Equals(currSquare))
                                            {
                                                blockingSquare = board.GetSquare(m, kingIndex.Item2 +
1);
                                            }
                                        }

                                    }

                                    Square destination = null;

                                    for (int l = 0; l < moves.Count; ++l)
                                    {
                                        // If you can block the king then do so
                                        if (blockingSquare != null)
                                        {
                                            if (moves[l].Equals(blockingSquare))
                                            {
                                                Tuple<Square, Piece, Square> moveTuple = new
Tuple<Square, Piece, Square>(selectedSquare, selectedPiece, blockingSquare);

                                                return moveTuple;
                                            }
                                        }
                                        else
                                        {
                                            for (int m = 0; m < kingNeighbours.Length; ++m)
                                            {
                                                if (moves[l].Equals(kingNeighbours[m]))
                                                {
                                                    destination = moves[l] as Square;

                                                    Tuple<Square, Piece, Square> moveTuple = new
Tuple<Square, Piece, Square>(selectedSquare, selectedPiece, destination);

                                                    return moveTuple;
                                                }
                                            }
                                        }
                                    }
                                }
                            }
                        }

                    }
                }
            }
            return null;
        }

        /// <summary>
        /// Tries to return a <see cref="Tuple{T1, T2, T3}"/> containing an origin <see
cref="Square"/>, destination <see cref="Square"/> and <see cref="Piece"/>
        /// that will allow the King to escape
        /// </summary>
        /// <param name="board">The <see cref="Board"/> to use.</param>
        /// <param name="squareList">A <see cref="List{T}"/> containing all <see cref="Square"/>
objects that have <see cref="Piece"/> objects on them.</param>
        /// <returns>A <see cref="Tuple{T1, T2, T3}"/> containing an origin <see cref="Square"/>,
destination <see cref="Square"/> and <see cref="Piece"/></returns>
        public Tuple<Square, Piece, Square> escapeKing(Board board, List<Square> squareList)
        {
            Square destination = null;

            for(int i = 0; i < squareList.Count; ++ i)
            {
                Square selectedSquare = squareList[i] as Square;

                if(selectedSquare != null &&
selectedSquare.getPiece().getType().Equals(Enums.PieceType.KING))
                {
                    Square kingSquare = selectedSquare;
                    Piece kingPiece = selectedSquare.getPiece();

                    List<Square> moveList = kingPiece.generateLegalMoves(board);

                    for(int j = 0; j < moveList.Count; ++j)
                    {
                        for(int k = 1; k < board.GetSize()-2; ++k)
                        {
                            Square currSquare = moveList[j] as Square;
                            // Top
                            if (board.GetSquare(1, k).Equals(currSquare))
                            {
                                destination = currSquare;
                            }
                            // Left
                            else if(board.GetSquare(k, 1).Equals(currSquare))
                            {
                                destination = currSquare;
                            }
                            // Bottom
                            else if (board.GetSquare(board.GetSize()-2, k).Equals(currSquare))
                            {
                                destination = currSquare;
                            }
                            // Right
                            else if (board.GetSquare(k, board.GetSize()-2).Equals(currSquare))
                            {
                                destination = currSquare;
                            }
                            if(destination != null)
                            {
                                Tuple<Square, Piece, Square> moveTuple = new Tuple<Square, Piece,
Square>(kingSquare, kingPiece, destination);

                                return moveTuple;
                            }
                        }

                    }
                }
            }
            return null;
        }

        /// <summary>
        /// Tries to return a <see cref="Tuple{T1, T2, T3}"/> containing an origin <see
cref="Square"/>, destination <see cref="Square"/> and <see cref="Piece"/>
        /// that will result in a win.
        /// </summary>
        /// <param name="board">The <see cref="Board"/> to use</param>
        /// <returns>A <see cref="Tuple{T1, T2, T3}"/> containing an origin <see cref="Square"/>,
destination <see cref="Square"/> and <see cref="Piece"/></returns>
        public Tuple<Square, Piece, Square> GetWinningMove(Board board)
        {
            List<Square> squareList = GetAllPieceSquares(board);
            Tuple<Square, Piece, Square> moveTuple = null;

            if (getColor().Equals(Enums.Color.BLACK))
            {
                moveTuple = captureKing(board, squareList);
            }
            else if (getColor().Equals(Enums.Color.WHITE))
            {
                moveTuple = escapeKing(board, squareList);
            }
            return moveTuple;
        }

        /// <summary>
        /// <see cref="ToString"/> override
        /// </summary>
        /// <returns>A <see cref="String"/> representation of the <see cref="Player"/></returns>
        public override string ToString()
        {
            string theString = "";

            theString += "\n | Color : " + getColor().ToString();
            theString += "\n | Type  : " + getType().ToString();
            theString += "\n | Name  : " + getName();
            theString += "\n | Moves : " + getMoveList().ToString();

            return theString;
        }

        #endregion
    }
}
```

2.1.7. MCTS.cs

2.1.7.1. Description

The implementation of the MCTS algorithm. Builds a game tree using the current state of the game board as the root.

2.1.7.2. Code

```
using System;
using System.Collections.Generic;
using System.Diagnostics;

namespace VikingChess.Model
{

    /// <summary>
    /// The MCTS Class contains the implementation for the Monte Carlo Tree Search algorithm. This
implementation
    /// uses the UCT formula for the selection step after all initial nodes for the CPU player have
been generated.
    /// This formula is defined in the <see cref="Node"/> class which holds data about the current
<see cref="Board"/>
    /// state.
    /// </summary>
    class MCTS
    {
        /// <summary>
        /// The number of points a node receives if the simulation results in a win for the CPU
player.
        /// </summary>
        private const double WIN_POINTS = 100.0;
        /// <summary>
        /// The number of points a node receives if the simulation results in a loss for the CPU
player.
        /// </summary>
        private const double LOSS_POINTS = -100;
        /// <summary>
        /// The number of times the steps of the MCTS algorithm run before the best node is chosen.
        /// Increasing this value may result in a better decision by the CPU at the cost of an
        /// increase in the time taken to produce a result and a higher memory cost.
        /// </summary>
        private const double PLAYOUTS = 1000;
        /// <summary>
        /// The number of moves to process during the simulation step.
        /// </summary>
        private const int SIM_TURNCOUNT = 150;
        /// <summary>
        /// The alphaPlayer is the CPU player running the MCTS algorithm and score are generated in
favour of this player.
        /// </summary>
        private Player alphaPlayer;
        /// <summary>
        /// The root node of the game tree.
        /// </summary>
        private Node root;

        /// <summary>
        /// Constructor for the MCTS class
        /// </summary>
        /// <param name="board"><The <see cref="Board"/> to use as a base for this <see cref="MCTS"/>
instance</param>
        public MCTS(Board board)
        {
            Board mctsBoard = new Board(board);
            root = new Node(mctsBoard);
            alphaPlayer = new Player(mctsBoard.getCurrentPlayer());
            //generateNodes(mctsBoard);
        }

        /// <summary>
        /// Generate all possible move states for the alpha player.
        /// </summary>
        /// <param name="board">The base <see cref="Board"/> to use when generating nodes.</param>
        public void generateNodes(Board board)
        {
            alphaPlayer = new Player(board.getCurrentPlayer());
            Board tempBoard = new Board(board);

            // Get all pieces that can be moved by the current player
            List<Square> squareList = alphaPlayer.GetAllPieceSquares(tempBoard);
            Tuple<Square, Piece, Square> moveTuple = null;
            Game newGame = null;
            Board newBoard = null;

            if (alphaPlayer.getColor().Equals(Enums.Color.BLACK))
            {
                moveTuple = alphaPlayer.captureKing(tempBoard, squareList);
                if(moveTuple != null)
                {
                    // Create a new game using the board for the move
                    newGame = new Game(tempBoard, true);
                    // Move the piece on the board
                    newGame.movePiece(moveTuple.Item3, moveTuple.Item1, moveTuple.Item2);
                    if (newGame.getCurrentPlayer().getColor().Equals(Enums.Color.WHITE))
                    {
                        newGame.updateCurrentPlayer();
                    }
                    // Set the current player of the game to the alpha player
                    tempBoard.setCurrentPlayer(newGame.getCurrentPlayer());
                    root.AddChild(tempBoard, moveTuple);
                }
            }
            else if (alphaPlayer.getColor().Equals(Enums.Color.WHITE))
            {
                moveTuple = alphaPlayer.escapeKing(tempBoard, squareList);
                if(moveTuple != null)
                {
                    // Create a new game using the board for the move
                    newGame = new Game(tempBoard, true);
                    // Move the piece on the board
                    newGame.movePiece(moveTuple.Item3, moveTuple.Item1, moveTuple.Item2);
                    if (newGame.getCurrentPlayer().getColor().Equals(Enums.Color.WHITE))
                    {
                        newGame.updateCurrentPlayer();
                    }
                    // Set the current player of the game to the alpha player
                    tempBoard.setCurrentPlayer(newGame.getCurrentPlayer());
                    root.AddChild(tempBoard, moveTuple);
                }
            }
            else
            {
                // Loop through the list of squares
                foreach (Square sq in squareList.ToArray())
                {
                    if (sq.getPiece() != null)
                    {
                        // Get a square with a moveable piece on it
                        Square origin = new Square(sq);
                        Piece selectedPiece = new Piece(sq.getPiece());

                        // Generate the possible moves for that piece
                        List<Square> moveList = sq.getPiece().generateLegalMoves(tempBoard);

                        // Loop through the move list
                        foreach (Square m in moveList.ToArray())
                        {

                            // Set the destination square to the current move
                            Square destination = new Square(m);

                            // Create a new board to hold the move
                            newBoard = new Board(board);

                            // Create a new game using the board for the move
                            newGame = new Game(newBoard, true);
                            // Move the piece on the board
                            newGame.movePiece(destination, origin, selectedPiece);
                            if (newGame.getCurrentPlayer().getColor().Equals(Enums.Color.WHITE))
                            {
                                newGame.updateCurrentPlayer();
                            }
                            // Set the current player of the game to the alpha player
                            newBoard.setCurrentPlayer(newGame.getCurrentPlayer());
                            moveTuple = new Tuple<Square, Piece, Square>(sq, sq.getPiece(),
destination);

                            root.AddChild(newBoard, moveTuple);
                            newBoard = null;
                            newGame = null;
                        }
                    }
                }
            }
        }

        /// <summary>
        /// Performs the MCTS algorithm.
        /// </summary>
        /// <param name="board">The <see cref="Board"/> to perform the algorithm on.</param>
        /// <returns>The best <see cref="Node"/> to obtain a move from.</returns>
        public Node makeMove(Board board)
        {
            for(int i=0; i < PLAYOUTS; ++i)
            {
                PerformSteps();
            }

            Node bestNode = root.getBestChild();

            return bestNode;
        }

        /// <summary>
        /// Performs te four steps of the MCTS algorithm:
        /// 1. Selection.
        /// 2. Expansion.
        /// 3. Simulation.
        /// 4. Backpropogation
        /// </summary>
        public void PerformSteps()
        {
            // Step 1. Selection
            List<Node> visitedNodes = new List<Node>();
            Node curr = root;
            Node bestChild = null;
            double score = 0;
            visitedNodes.Add(curr);

            // Find the best child node to expand
            while (curr.isLeaf().Equals(false))
            {
                curr = curr.UCTSelectBest();
                visitedNodes.Add(curr);
            }
            if (curr.getWin().Equals(false))
            {
                // Step 2. Expansion
                curr.ExpandNode();
                bestChild = curr.UCTSelectBest();
            }
            if (bestChild == null)
            {
                // Best child is terminal state
                bestChild = curr;
            }
            else
            {
                visitedNodes.Add(bestChild);
            }
            if (bestChild.getWin().Equals(false))
            {
                // Step 3. Simulation
                score = SimulateGame(bestChild);
            }

            // Step 4. Backpropogation
            foreach(Node n in visitedNodes)
            {
                n.updateStats(score);
            }
        }

        /// <summary>
        /// Simulates a <see cref="Game"/> between two <see cref="Player"/> then scores the resulting
<see cref="Board"/>.
        /// </summary>
        /// <param name="node">The <see cref="Node"/> to simulate the <see cref="Game"/> from.</param>
        /// <returns>The <see cref="Node.score"/> for this simulation/></returns>
        public double SimulateGame(Node node)
        {
            Board gameBoard = new Board(node.GetBoard());
            Game game = new Game(gameBoard, true);

            try
            {
                // Then for the specified turn count, play out the game.
                while (game.getTurnCount() < SIM_TURNCOUNT)
                {
                    if (game.checkWin())
                    {
                        node.setWin(true);
                        return evaluateBoard(gameBoard, game);
                    }

                    Tuple<Square, Piece, Square> cpuMovesTuple = null;
                    // Try to find a winning move.
                    cpuMovesTuple = game.getCurrentPlayer().GetWinningMove(gameBoard);
                    // If no winning move was found
                    if (cpuMovesTuple == null)
                    {
                        // Try to capture a piece
                        cpuMovesTuple = game.getCurrentPlayer().capturePiece(gameBoard);
                    }
                    if (cpuMovesTuple == null)
                    {
                        // Find a general move
                        cpuMovesTuple = game.getCurrentPlayer().getMoveTuple(gameBoard);
                    }
                    Piece selectedCPUPiece = cpuMovesTuple.Item2;
                    Square destination = cpuMovesTuple.Item3;
                    Square origin = cpuMovesTuple.Item1;

                    // Move the piece
                    game.movePieceAndGetBoard(destination, origin, selectedCPUPiece);
                    //game.updateCurrentPlayer();
                }

                return evaluateBoard(gameBoard, game);
            }
            catch(NullReferenceException e)
            {
                return 0;
            }

        }

        /// <summary>
        /// Scores a <see cref="Board"/> on the following parameters: Win/Loss, <see cref="Piece"/>
count and <see cref="Board"/> Control.
        /// </summary>
        /// <param name="board">The <see cref="Board"/> to score for.</param>
        /// <param name="game">The <see cref="Game"/> the <see cref="Board"/> was used in.</param>
        /// <returns>A score for the <see cref="Board"/> that was passed in.</returns>
        private double evaluateBoard(Board board, Game game)
        {
            double score = 0;

            // If player 1 has won and it is the alpha player
            if (game.getPlayer1().getWin() && alphaPlayer.getColor().Equals(Enums.Color.BLACK))
            {
                // Score positive
                score += WIN_POINTS;

                // Get a count of pieces
                score += scorePieces(board, game);

                // Evaluate control of rank and file
                score += evaluateControl(board, game);
            }
            // Else if player 2 has won and it is the alpha player
            else if (game.getPlayer2().getWin() && alphaPlayer.getColor().Equals(Enums.Color.WHITE))
            {
                // Score positive
                score += WIN_POINTS;

                // Get a count of pieces
                score += scorePieces(board, game);

                // Evaluate control of rank and file
                score += evaluateControl(board, game);
            }
            // Else if player 1 has won and it is not the alpha player
            else if (game.getPlayer1().getWin() && alphaPlayer.getColor().Equals(Enums.Color.WHITE))
            {
                // Score negative
                score += LOSS_POINTS;

                // Get a count of pieces
                score += scorePieces(board, game);

                // Evaluate control of rank and file
                score += evaluateControl(board, game);
            }
            // Else if player 2 has won and it is not the alpha player
            else if (game.getPlayer2().getWin() && alphaPlayer.getColor().Equals(Enums.Color.BLACK))
            {
                // Score negative
                score += LOSS_POINTS;

                // Get a count of pieces
                score += scorePieces(board, game);

                // Evaluate control of rank and file
                score += evaluateControl(board, game);
            }
            else
            {
                // Get a count of pieces
                score += scorePieces(board, game);

                // Evaluate control of rank and file
                score += evaluateControl(board, game);
            }

            // Check the king's proximity to the edges
            return score;
        }

        /// <summary>
        /// Finds the outermost pieces on the board and scores the board appropriately.
        /// The reason for this is that control of the outer squares in an advantage to both players.
        /// </summary>
        /// <param name="board">The <see cref="Board"/> to score for.</param>
        /// <param name="game">The <see cref="Game"/> the <see cref="Board"/> was used in.</param>
        /// <returns>A score for the <see cref="Board"/> that was passed in.</returns>
        private double evaluateControl(Board board, Game game)
        {
            Piece leftmostPiece = null;
            Piece rightmostPiece = null;
            Piece topmostPiece = null;
            Piece bottommostPiece = null;
            double blackScore = 0;
            double whiteScore = 0;
            double score = 0;
            int i;
            int j;

            // Check the files for outermost pieces
            for (i = 1; i < board.GetSize()-2; ++i)
            {
                for(j = 1; j < board.GetSize()-2; ++j)
                {
                    if(board.GetSquare(i,j) != null)
                    {
                        if(board.GetSquare(i,j).getPiece() != null)
                        {
                            // On the first iteration, set all pieces
                            if(leftmostPiece == null)
                            {
                                leftmostPiece = board.GetSquare(i, j).getPiece();
                                rightmostPiece = board.GetSquare(i, j).getPiece();
                            }
                            // If the current piece's file is less than the current leftmostPiece,
make it the leftmostPiece
                            if (board.GetSquare(i, j).getPiece().getFile() < leftmostPiece.getFile())
                            {
                                leftmostPiece = board.GetSquare(i, j).getPiece();
                            }
                            // If the current piece's file is greater than the current rightmostPiece,
make it the rightmostPiece
                            if (board.GetSquare(i, j).getPiece().getFile() > rightmostPiece.getFile())
                            {
                                rightmostPiece = board.GetSquare(i, j).getPiece();
                            }
                        }
                    }
                }
                if(leftmostPiece != null)
                {
                    // Score for leftmostPiece
                    if (leftmostPiece.getColor().Equals(Enums.Color.BLACK))
                    {
                        ++blackScore;
                    }
                    else if (leftmostPiece.getColor().Equals(Enums.Color.WHITE))
                    {
                        ++whiteScore;
                    }
                }
                if(rightmostPiece != null)
                {
                    // Score for rightmostPiece
                    if (rightmostPiece.getColor().Equals(Enums.Color.BLACK))
                    {
                        ++blackScore;
                    }
                    else if (rightmostPiece.getColor().Equals(Enums.Color.WHITE))
                    {
                        ++whiteScore;
                    }
                }
                leftmostPiece = null;
                rightmostPiece = null;
            }
            // Check the ranks for outermost pieces
            for (i = 1; i < board.GetSize() - 2; ++i)
            {
                for (j = 1; j < board.GetSize() - 2; ++j)
                {
                    if (board.GetSquare(j, i) != null)
                    {
                        if (board.GetSquare(j, i).getPiece() != null)
                        {
                            // On the first iteration, set all pieces
                            if (topmostPiece == null)
                            {
                                topmostPiece = board.GetSquare(j, i).getPiece();
                                bottommostPiece = board.GetSquare(j, i).getPiece();
                            }
                            // If the current piece's file is less than the current leftmostPiece,
make it the leftmostPiece
                            if (board.GetSquare(j, i).getPiece().getFile() < topmostPiece.getFile())
                            {
                                topmostPiece = board.GetSquare(j, i).getPiece();
                            }
                            // If the current piece's file is greater than the current rightmostPiece,
make it the rightmostPiece
                            if (board.GetSquare(j, i).getPiece().getFile() >
bottommostPiece.getFile())
                            {
                                bottommostPiece = board.GetSquare(j, i).getPiece();
                            }
                        }

                    }
                }
                if(topmostPiece != null)
                {
                    // Score for topmostPiece
                    if (topmostPiece.getColor().Equals(Enums.Color.BLACK))
                    {
                        ++blackScore;
                    }
                    else if (topmostPiece.getColor().Equals(Enums.Color.WHITE))
                    {
                        ++whiteScore;
                    }
                }
                if(bottommostPiece != null)
                {
                    // Score for bottommostPiece
                    if (bottommostPiece.getColor().Equals(Enums.Color.BLACK))
                    {
                        ++blackScore;
                    }
                    else if (bottommostPiece.getColor().Equals(Enums.Color.WHITE))
                    {
                        ++whiteScore;
                    }
                }
                topmostPiece = null;
                bottommostPiece = null;
            }

            // score for the alphaPlayer
            if (alphaPlayer.getColor().Equals(Enums.Color.BLACK))
            {
                score += blackScore;
                score -= whiteScore;
            }
            else if (alphaPlayer.getColor().Equals(Enums.Color.WHITE))
            {
                score += whiteScore;
                score -= blackScore;
            }

            return score;
        }

        /// <summary>
        /// Count the number of pieces on the board and score the board appropriately
        /// </summary>
        /// <param name="board">The <see cref="Board"/> to score for.</param>
        /// <param name="game">The <see cref="Game"/> the <see cref="Board"/> was used in.</param>
        /// <returns>A score for the <see cref="Board"/> that was passed in.</returns>
        private int scorePieces(Board board, Game game)
        {
            int score = 0;

            int blackCount = board.countPieces(Enums.Color.BLACK);
            int whiteCount = board.countPieces(Enums.Color.WHITE);

            // If the AlphaPlayer is Attacking
            if (alphaPlayer.getColor().Equals(Enums.Color.BLACK))
            {
                // Reflect the importance of a low defender count by penalizing boards with a high
defender count
                score += (blackCount);
                score -= (whiteCount);
            }
            // If the AlphaPlayer is Defending
            else if (alphaPlayer.getColor().Equals(Enums.Color.WHITE))
            {
                // Reflect the importance of a high defender count by rewarding boards with a high
defender count;
                score -= (blackCount);
                score += (whiteCount);
            }

            return score;
        }
    }
}
```

2.1.8. Node.cs

2.1.8.1. Description

Represents a node in an MCTS Game Tree.

2.1.8.2. Code

```
using System;
using System.Collections.Generic;
using System.Diagnostics;

namespace VikingChess.Model
{
    /// <summary>
    /// The <see cref="Node"/> class defines a node within an <see cref="MCTS"/> tree.
    /// </summary>
    class Node
    {
        /// <summary>
        /// The number of times this <see cref="Node"/> has been visited.
        /// </summary>
        private int visitCount = 0;
        /// <summary>
        /// The score of this <see cref="Node"/>
        /// </summary>
        private double score = 0;
        /// <summary>
        /// The <see cref="Board"/> object representing the game board and position of pieces.
        /// </summary>
        private Board board;
        /// <summary>
        /// The parent <see cref="Node"/> of this <see cref="Node"/>
        /// </summary>
        private Node parent;
        /// <summary>
        /// An <see cref="List"/> containing any and all child <see cref="States"/> of this <see
cref="Node"/>
        /// </summary>
        private List<Node> children;
        /// <summary>
        /// Defines whether or not the board contained within this node is a terminal state or not.
        /// </summary>
        private bool hasWon = false;
        /// <summary>
        /// A constant to protect against divide by zero errors in the UCT algorithm.
        /// </summary>
        private const double EPSILON = 10e-6;
        /// <summary>
        /// The last move made that resulted in the <see cref="board"/> of this node.
        /// </summary>
        private Tuple<Square, Piece, Square> lastMoveTuple;

        /// <summary>
        /// Constructs a new <see cref="Node"/>. This constructor is used to form the root of the <see
cref="Node"/> tree.
        /// </summary>
        /// <param name="board">The <see cref="Board"/> object to store.</param>
        public Node(Board board)
        {
            parent = null;
            this.board = board;

            children = new List<Node>();
        }

        /// <summary>
        /// Constructs a new <see cref="Node"/>. This constructor is used to create a child <see
cref="Node"/>.
        /// </summary>
        /// <param name="board">The <see cref="Board"/> object to store.</param>
        /// <param name="parent">The parent <see cref="Node"/> of this <see cref="Node"/></param>
        public Node(Board board, Node parent)
        {
            this.parent = parent;
            this.board = board;

            children = new List<Node>();
        }

        /// <summary>
        /// Copy Constructor
        /// </summary>
        /// <param name="node">The <see cref="Node"/> to copy</param>
        public Node(Node node)
        {
            parent = node.parent;
            board = node.board;
            lastMoveTuple = node.lastMoveTuple;

            children = new List<Node>(node.children);
        }

        /// <summary>
        /// Returns the <see cref="Board"/> object contained in this <see cref="Node"/>.
        /// </summary>
        /// <returns>The <see cref="Board"/> contained in this <see cref="Node"/></returns>
        public Board GetBoard()
        {
            return board;
        }

        /// <summary>
        /// Returns the <see cref="parent"/> of this <see cref="Node"/>.
        /// </summary>
        /// <returns>The <see cref="parent"/> of this <see cref="Node"/></returns>
        public Node GetParent()
        {
            return parent;
        }

        /// <summary>
        /// Returns the <see cref="children"/> of this <see cref="Node"/>.
        /// </summary>
        /// <returns>A <see cref="List{T}"/> containing the <see cref="children"/> of this <see
cref="Node"/>/></returns>
        public List<Node> GetChildren()
        {
            return children;
        }

        /// <summary>
        ///  Returns the number of times this <see cref="Node"/> has been visited.
        /// </summary>
        /// <returns><see cref="visitCount"/></returns>
        public int GetVisits()
        {
            return visitCount;
        }

        /// <summary>
        /// Returns the win status of this <see cref="Node"/>
        /// </summary>
        /// <returns><see cref="hasWon"/></returns>
        public bool getWin()
        {
            return hasWon;
        }

        /// <summary>
        /// Sets the win status of this <see cref="Node"/>
        /// </summary>
        /// <param name="winStatus">The <see cref="bool"/> win status to set.</param>
        public void setWin(bool winStatus)
        {
            hasWon = winStatus;
        }

        /// <summary>
        /// Returns the <see cref="score"/> of this <see cref="Node"/>.
        /// </summary>
        /// <returns><see cref="score"/></returns>
        public double GetScore()
        {
            return score;
        }

        /// <summary>
        /// Returns the chance that this <see cref="Node"/> will lead to a win.
        /// </summary>
        /// <returns>The chance that this node will lead to a win. This is defined as <see
cref="score"/> / <see cref="visitCount"/></returns>
        public double getWinChance()
        {
            if(visitCount == 0)
            {
                return 0;
            }
            else
            {
                return score / visitCount;
            }
        }

        /// <summary>
        /// Finds the best child <see cref="Node"/> of this <see cref="Node"/>
        /// </summary>
        /// <returns>The best child <see cref="Node"/> of this <see cref="Node"/></returns>
        public Node getBestChild()
        {
            Node bestChild = children[0];
            for (int i = 1; i < children.Count; ++i)
            {
                Node currChild = children[i];
                if (bestChild.getWinChance() < currChild.getWinChance())
                {
                    bestChild = currChild;
                }
            }
            return bestChild;
        }
        /// <summary>
        /// Returns the <see cref="lastMoveTuple"/> of this <see cref="Node"/>
        /// </summary>
        /// <returns><see cref="lastMoveTuple"/></returns>
        public Tuple<Square, Piece, Square> GetLastMove()
        {
            return lastMoveTuple;
        }

        /// <summary>
        /// Sets the <see cref="lastMoveTuple"/> of this <see cref="Node"/>.
        /// </summary>
        /// <param name="moveTuple">The move that results in the <see cref="Board"/> contained in this 
<see cref="Node"/></param>
        public void setLastMove(Tuple<Square, Piece, Square> moveTuple)
        {
            lastMoveTuple = moveTuple;
        }

        /// <summary>
        /// Checks to see if this is a leaf <see cref="Node"/>.
        /// </summary>
        /// <returns>A <see cref="bool"/> representing whether this <see cref="Node"/> is a leaf node 
or not.</returns>
        public bool isLeaf()
        {
            if(children.Count <= 0)
            {
                return true;
            }
            else
            {
                return false;
            }
        }

        /// <summary>
        /// Updates the <see cref="visitCount"/> of this <see cref="Node"/>
        /// </summary>
        public void updateVisitCount()
        {
            ++visitCount;
        }

        /// <summary>
        /// Updates the <see cref="score"/> of this <see cref="Node"/>.
        /// </summary>
        /// <param name="value"></param>
        public void updateScore(double value)
        {
            score += value;
        }

        /// <summary>
        /// A method that updates the <see cref="score"/> and <see cref="visitCount"/> of this <see 
cref="Node"/>
        /// </summary>
        /// <param name="score"></param>
        public void updateStats(double score)
        {
            updateVisitCount();
            updateScore(score);
        }

        /// <summary>
        /// Adds a new child <see cref="Node"/> to this <see cref="Node"/>.
        /// </summary>
        /// <param name="board"></param>
        public void AddChild(Board board, Tuple<Square, Piece, Square> moveTuple)
        {
            Node state = new Node(board, this);
            state.setLastMove(moveTuple);
            children.Add(state);
        }
        /// <summary>
        /// Uses the UCT formula to select the most promising child <see cref="Node"/> to explore.
        /// <code>// UCT Formula
        /// uctVal = (child.GetScore() / (child.GetVisits() + EPSILON)) + Math.Sqrt(2) *
Math.Sqrt((Math.Log(GetVisits() + 1)) / (child.GetVisits() + EPSILON)) + x + EPSILON;</code>
        /// </summary>
        /// <returns>The most promising child <see cref="Node"/></returns>
        public Node UCTSelectBest()
        {
            if (children.Count == 0)
            {
                return null;
            }

            Node selected = null;
            double bestValue = Double.NegativeInfinity;

            // Calculate the UCT value for each child node
            foreach (Node child in children)
            {
                // A small random number to aid in offsetting ties between nodes
                Random r = new Random();
                int x = r.Next(0, 2);
                double uctVal;

                // UCT Formula
                uctVal = (child.GetScore() / (child.GetVisits() + EPSILON)) +
                Math.Sqrt(2) * Math.Sqrt((Math.Log(GetVisits() + 1)) / (child.GetVisits() + EPSILON))
+ x + EPSILON;

                if (uctVal > bestValue)
                {
                    selected = child;
                    bestValue = uctVal;
                }
            }
            return selected;
        }

        /// <summary>
        /// Expands this <see cref="Node"/> by creating new child <see cref="Node"/>
        /// </summary>
        public void ExpandNode()
        {
            Game game = new Game(board, true);
            Board tempBoard = new Board(board);
            Player betaPlayer = board.getCurrentPlayer();
            Tuple<Square, Piece, Square> moveTuple = null;

            try
            {
                // Make winning move for opponent if possible
                moveTuple = betaPlayer.GetWinningMove(board);
                if (moveTuple != null)
                {
                    tempBoard = new Board(board);
                    tempBoard.setCurrentPlayer(betaPlayer);
                    Game tempGame = new Game(tempBoard, true);
                    tempGame.movePiece(moveTuple.Item3, moveTuple.Item1, moveTuple.Item2);
                    if (tempGame.getCurrentPlayer().getColor().Equals(betaPlayer.getColor()))
                    {
                        tempGame.updateCurrentPlayer();
                    }
                    tempBoard.setCurrentPlayer(tempGame.getCurrentPlayer());
                    AddChild(tempBoard, moveTuple);
                    tempBoard = null;
                    tempGame = null;
                    return;
                }
                moveTuple = betaPlayer.capturePiece(board);
                if (moveTuple != null)
                {
                    // Make alternative move
                    tempBoard = new Board(board);
                    tempBoard.setCurrentPlayer(betaPlayer);
                    Game tempGame = new Game(tempBoard, true);
                    tempGame.movePiece(moveTuple.Item3, moveTuple.Item1, moveTuple.Item2);
                    if (tempGame.getCurrentPlayer().getColor().Equals(betaPlayer.getColor()))
                    {
                        tempGame.updateCurrentPlayer();
                    }
                    tempBoard.setCurrentPlayer(tempGame.getCurrentPlayer());
                    AddChild(tempBoard, moveTuple);
                    tempBoard = null;
                    tempGame = null;
                    return;
                }
                else
                {
                    // Get all pieces that can be moved by the current player
                    List<Square> squareList = betaPlayer.GetAllPieceSquares(tempBoard);
                    // Loop through the list of squares
                    foreach (Square sq in squareList.ToArray())
                    {
                        // Get a square with a moveable piece on it
                        Square origin = new Square(sq);
                        Piece selectedPiece = new Piece(sq.getPiece());

                        // Generate the possible moves for that piece
                        List<Square> moveList = sq.getPiece().generateLegalMoves(tempBoard);

                        // Loop through the move list
                        foreach (Square m in moveList.ToArray())
                        {
                            // Set the destination square to the current move
                            Square destination = new Square(m);

                            // Create a new board to hold the move
                            Board newBoard = new Board(board);
                            // Set the current player of the game to the alpha player
                            newBoard.setCurrentPlayer(betaPlayer);
                            // Create a new game using the board for the move
                            Game tempGame = new Game(newBoard, true);

                            // Move the piece on the board
                            tempGame.movePiece(destination, origin, selectedPiece);
                            if (tempGame.getCurrentPlayer().getColor().Equals(betaPlayer.getColor()))
                            {
                                tempGame.updateCurrentPlayer();
                            }
                            newBoard.setCurrentPlayer(tempGame.getCurrentPlayer());

                            moveTuple = new Tuple<Square, Piece, Square>(sq, sq.getPiece(), m);

                            AddChild(newBoard, moveTuple);
                            newBoard = null;
                            tempGame = null;
                        }
                    }
                }
            }
            catch(NullReferenceException e)
            {
                //
            }

        }

        /// <summary>
        /// <see cref="ToString"/> override
        /// </summary>
        /// <returns>A <see cref="String"/> represntation of this <see cref="Node"/></returns>
        public override string ToString()
        {
            string theString = "";

            theString += "\n | Score      : " + score + " ";
            theString += "\n | Visits     : " + visitCount;
            theString += "\n | Has Won    : " + hasWon;
            theString += "\n | Children   : " + children.Count;
            theString += "\n | UCT Value  : " + UCTSelectBest();

            return theString;
        }

    }
}
```

2.1.9. Enums

2.1.9.1. Description

Contains a collection of enumerators for use in other classes.

2.1.9.2. Code

```
namespace VikingChess.Model
{
    /// <summary>
    /// A helper class containing enumerators.
    /// </summary>
    public class Enums
    {
        /// <summary>
        /// Blank Constructor
        /// </summary>
        private Enums()
        {

        }

        /// <summary>
        /// Represents the File of a <see cref="Piece"/> on the <see cref="Board"/>
        /// </summary>
        public enum File { A, B, C, D, E, F, G, H, I, J, K }

        /// <summary>
        /// Represents the Rank of a <see cref="Piece"/> on the <see cref="Board"/>
        /// </summary>
        public enum Rank { None, One, Two, Three, Four, Five, Six, Seven, Eight, Nine, Ten, Eleven }

        /// <summary>
        /// Represents the color of a <see cref="Piece"/> or a <see cref="Player"/>
        /// </summary>
        public enum Color { WHITE, BLACK }

        /// <summary>
        /// Represents the type of a <see cref="Piece"/>
        /// </summary>
        public enum PieceType { PAWN, KING }

        /// <summary>
        /// Represents the type of a <see cref="Square"/>
        /// </summary>
        public enum SquareType { CORNER, THRONE, REGULAR }

        /// <summary>
        /// Represents the type of a <see cref="Player"/>
        /// </summary>
        public enum PlayerType { HUMAN, CPU }
    }
}
```

3. XAML Files & Associated Code-Behinds

3.1. Game Page

3.1.1. Description

Defines the layout of the Game UI. Note that the listed page is one of five game pages. All game pages are identical but for the number of rectangles bound, the size of the board and the name of the gametype. All other game pages have been excluded for this reason.

3.1.2. Code

3.1.2.1. XAML

```
<Page
    x:Class="VikingChess.ArdRiPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:VikingChess"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">
    <Page.Resources>

        <Style x:Key="ButtonStyle1" TargetType="Button">
            <Setter Property="Background" Value="{ThemeResource
SystemControlBackgroundBaseLowBrush}"/>
            <Setter Property="Foreground" Value="{ThemeResource
SystemControlForegroundBaseHighBrush}"/>
            <Setter Property="BorderBrush" Value="{ThemeResource
SystemControlForegroundTransparentBrush}"/>
            <Setter Property="BorderThickness" Value="{ThemeResource ButtonBorderThemeThickness}"/>
            <Setter Property="Padding" Value="8,4,8,4"/>
            <Setter Property="HorizontalAlignment" Value="Left"/>
            <Setter Property="VerticalAlignment" Value="Center"/>
            <Setter Property="FontFamily" Value="{ThemeResource ContentControlThemeFontFamily}"/>
            <Setter Property="FontWeight" Value="Normal"/>
            <Setter Property="FontSize" Value="{ThemeResource ControlContentThemeFontSize}"/>
            <Setter Property="UseSystemFocusVisuals" Value="True"/>
            <Setter Property="Template">
                <Setter.Value>
                    <ControlTemplate TargetType="Button">
                        <Grid x:Name="RootGrid" Background="{TemplateBinding Background}">
                            <VisualStateManager.VisualStateGroups>
                                <VisualStateGroup x:Name="CommonStates">
                                    <VisualState x:Name="Normal">
                                        <Storyboard>
                                            <PointerUpThemeAnimation
Storyboard.TargetName="RootGrid"/>
                                        </Storyboard>
                                    </VisualState>
                                    <VisualState x:Name="PointerOver">
                                        <Storyboard>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="BorderBrush" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="#FFFFA200"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Foreground" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlHighlightBaseHighBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <PointerUpThemeAnimation
Storyboard.TargetName="RootGrid"/>
                                        </Storyboard>
                                    </VisualState>
                                    <VisualState x:Name="Pressed">
                                        <Storyboard>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Foreground" Storyboard.TargetName="ContentPresenter">
```

```
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlHighlightBaseHighBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <PointerDownThemeAnimation
Storyboard.TargetName="RootGrid"/>
                                        </Storyboard>
                                    </VisualState>
                                    <VisualState x:Name="Disabled">
                                        <Storyboard>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Background" Storyboard.TargetName="RootGrid">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlBackgroundBaseLowBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Foreground" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlDisabledBaseLowBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="BorderBrush" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlDisabledTransparentBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                        </Storyboard>
                                    </VisualState>
                                </VisualStateGroup>
                            </VisualStateManager.VisualStateGroups>
                            <ContentPresenter x:Name="ContentPresenter"
AutomationProperties.AccessibilityView="Raw" BorderBrush="{TemplateBinding BorderBrush}"
BorderThickness="{TemplateBinding BorderThickness}" ContentTemplate="{TemplateBinding
ContentTemplate}" ContentTransitions="{TemplateBinding ContentTransitions}" Content="{TemplateBinding
Content}" HorizontalContentAlignment="{TemplateBinding HorizontalContentAlignment}"
Padding="{TemplateBinding Padding}" VerticalContentAlignment="{TemplateBinding
VerticalContentAlignment}"/>
                        </Grid>
                    </ControlTemplate>
                </Setter.Value>
            </Setter>
        </Style>
        <Style x:Key="topbarButtonStyle" TargetType="Button">
            <Setter Property="Background" Value="{ThemeResource
SystemControlBackgroundBaseLowBrush}"/>
            <Setter Property="Foreground" Value="{ThemeResource
SystemControlForegroundBaseHighBrush}"/>
            <Setter Property="BorderBrush" Value="{ThemeResource
SystemControlForegroundTransparentBrush}"/>
            <Setter Property="BorderThickness" Value="{ThemeResource ButtonBorderThemeThickness}"/>
            <Setter Property="Padding" Value="8,4,8,4"/>
            <Setter Property="HorizontalAlignment" Value="Left"/>
            <Setter Property="VerticalAlignment" Value="Center"/>
            <Setter Property="FontFamily" Value="{ThemeResource ContentControlThemeFontFamily}"/>
            <Setter Property="FontWeight" Value="Normal"/>
            <Setter Property="FontSize" Value="{ThemeResource ControlContentThemeFontSize}"/>
            <Setter Property="UseSystemFocusVisuals" Value="True"/>
            <Setter Property="Template">
                <Setter.Value>
                    <ControlTemplate TargetType="Button">
                        <Grid x:Name="RootGrid" Background="{TemplateBinding Background}">
                            <VisualStateManager.VisualStateGroups>
                                <VisualStateGroup x:Name="CommonStates">
                                    <VisualState x:Name="Normal">
                                        <Storyboard>
                                            <PointerUpThemeAnimation
Storyboard.TargetName="RootGrid"/>
                                        </Storyboard>
                                    </VisualState>
                                    <VisualState x:Name="PointerOver">
                                        <Storyboard>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Foreground" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlHighlightBaseHighBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <PointerUpThemeAnimation
Storyboard.TargetName="RootGrid"/>
                                        </Storyboard>
                                    </VisualState>
                                    <VisualState x:Name="Pressed">
                                        <Storyboard>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Foreground" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlHighlightBaseHighBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <PointerDownThemeAnimation
Storyboard.TargetName="RootGrid"/>
                                        </Storyboard>
                                    </VisualState>
                                    <VisualState x:Name="Disabled">
                                        <Storyboard>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Background" Storyboard.TargetName="RootGrid">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlBackgroundBaseLowBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Foreground" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlDisabledBaseLowBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="BorderBrush" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlDisabledTransparentBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                        </Storyboard>
                                    </VisualState>
                                </VisualStateGroup>
                            </VisualStateManager.VisualStateGroups>
                            <ContentPresenter x:Name="ContentPresenter"
AutomationProperties.AccessibilityView="Raw" BorderBrush="{TemplateBinding BorderBrush}"
BorderThickness="{TemplateBinding BorderThickness}" ContentTemplate="{TemplateBinding
ContentTemplate}" ContentTransitions="{TemplateBinding ContentTransitions}" Content="{TemplateBinding
Content}" HorizontalContentAlignment="{TemplateBinding HorizontalContentAlignment}"
Padding="{TemplateBinding Padding}" VerticalContentAlignment="{TemplateBinding
VerticalContentAlignment}"/>
                        </Grid>
                    </ControlTemplate>
                </Setter.Value>
            </Setter>
        </Style>
    </Page.Resources>

    <Grid>
        <Grid.Background>
            <ImageBrush Stretch="UniformToFill" ImageSource="Assets/bg-wood.png"/>
        </Grid.Background>
        <RelativePanel Height="922" Margin="524,62,474,0" VerticalAlignment="Top"
RenderTransformOrigin="0.5,0.5">
            <RelativePanel.Background>
                <ImageBrush ImageSource="Assets/BackboardBrandubh.png"/>
            </RelativePanel.Background>
            <RelativePanel.RenderTransform>
                <CompositeTransform ScaleY="0.8" ScaleX="0.8"/>
            </RelativePanel.RenderTransform>
            <Grid Margin="2,2,-2,-2" RenderTransformOrigin="0.5,0.5">
                <Grid.RenderTransform>
                    <CompositeTransform ScaleY="0.95999997854232788" ScaleX="0.95999997854232788"/>
                </Grid.RenderTransform>
                <Grid.RowDefinitions>
                    <RowDefinition Height="117*"/>
                    <RowDefinition Height="103*"/>
                </Grid.RowDefinitions>

                <Image Width="880" Height="880" Source="Assets/ArdRiBoard.png" Margin="20,20,-20,0"
Grid.RowSpan="2" />
```

```
                <Rectangle x:Name="A7" HorizontalAlignment="Left" Height="126" Margin="20,19,0,0"
VerticalAlignment="Top" Width="125" />
                <Rectangle x:Name="B7" HorizontalAlignment="Left" Height="125" Margin="145,20,0,0"
VerticalAlignment="Top" Width="125"/>
                <Rectangle x:Name="C7" HorizontalAlignment="Left" Height="125" Margin="270,20,0,0"
VerticalAlignment="Top" Width="125"/>
                <Rectangle x:Name="D7" HorizontalAlignment="Left" Height="125" Margin="395,20,0,0"
VerticalAlignment="Top" Width="128"/>
                <Rectangle x:Name="E7" HorizontalAlignment="Left" Height="125" Margin="523,20,0,0"
VerticalAlignment="Top" Width="125"/>
                <Rectangle x:Name="F7" HorizontalAlignment="Left" Height="125" Margin="648,20,0,0"
VerticalAlignment="Top" Width="126"/>
                <Rectangle x:Name="G7" HorizontalAlignment="Left" Height="125" Margin="774,20,-20,0"
VerticalAlignment="Top" Width="126"/>

                <!-- Row 6 -->
                <Rectangle x:Name="A6" HorizontalAlignment="Left" Height="126" Margin="20,145,0,0"
VerticalAlignment="Top" Width="125"/>
                <Rectangle x:Name="B6" HorizontalAlignment="Left" Height="126" Margin="145,145,0,0"
VerticalAlignment="Top" Width="125"/>
                <Rectangle x:Name="C6" HorizontalAlignment="Left" Height="126" Margin="270,145,0,0"
VerticalAlignment="Top" Width="125"/>
                <Rectangle x:Name="D6" HorizontalAlignment="Left" Height="126" Margin="395,145,0,0"
VerticalAlignment="Top" Width="128"/>
                <Rectangle x:Name="F6" HorizontalAlignment="Left" Height="126" Margin="648,145,0,0"
VerticalAlignment="Top" Width="126"/>
                <Rectangle x:Name="G6" HorizontalAlignment="Left" Height="126" Margin="774,145,-20,0"
VerticalAlignment="Top" Width="126"/>
                <Rectangle x:Name="E6" HorizontalAlignment="Left" Margin="523,145,0,197" Width="125"/>

                <!-- Row 5 -->
                <Rectangle x:Name="A5" HorizontalAlignment="Left" Margin="20,271,0,71" Width="125"/>
                <Rectangle x:Name="B5" HorizontalAlignment="Left" Margin="145,271,0,71" Width="125"/>
                <Rectangle x:Name="C5" HorizontalAlignment="Left" Margin="270,271,0,71" Width="125"/>
                <Rectangle x:Name="D5" HorizontalAlignment="Left" Margin="395,271,0,71" Width="128"/>
                <Rectangle x:Name="E5" HorizontalAlignment="Left" Margin="523,271,0,71" Width="125"/>
                <Rectangle x:Name="F5" HorizontalAlignment="Left" Margin="648,271,0,71" Width="126"/>
                <Rectangle x:Name="G5" HorizontalAlignment="Left" Margin="774,271,-20,71"
Width="126"/>

                <!-- Row 4 -->
                <Rectangle x:Name="A4" HorizontalAlignment="Left" Height="126" Margin="20,0,0,377"
VerticalAlignment="Bottom" Width="125" Grid.RowSpan="2"/>
                <Rectangle x:Name="B4" HorizontalAlignment="Left" Height="126" Margin="145,397,0,0"
VerticalAlignment="Top" Width="125" Grid.RowSpan="2"/>
                <Rectangle x:Name="C4" HorizontalAlignment="Left" Height="126" Margin="270,0,0,377"
VerticalAlignment="Bottom" Width="125" Grid.RowSpan="2"/>
                <Rectangle x:Name="D4" HorizontalAlignment="Left" Height="126" Margin="395,0,0,377"
VerticalAlignment="Bottom" Width="128" Grid.RowSpan="2"/>
                <Rectangle x:Name="E4" HorizontalAlignment="Left" Height="126" Margin="523,0,0,377"
VerticalAlignment="Bottom" Width="125" Grid.RowSpan="2"/>
                <Rectangle x:Name="F4" HorizontalAlignment="Left" Height="126" Margin="648,397,0,0"
VerticalAlignment="Top" Width="126" Grid.RowSpan="2"/>
                <Rectangle x:Name="G4" HorizontalAlignment="Left" Height="126" Margin="774,0,-20,377"
VerticalAlignment="Bottom" Width="126" Grid.RowSpan="2"/>

                <!-- Row 3 -->
                <Rectangle x:Name="A3" HorizontalAlignment="Left" Height="125" Margin="20,55,0,0"
VerticalAlignment="Top" Width="125" Grid.Row="1"/>
                <Rectangle x:Name="B3" HorizontalAlignment="Left" Height="125" Margin="145,0,0,252"
VerticalAlignment="Bottom" Width="125" Grid.Row="1"/>
                <Rectangle x:Name="C3" HorizontalAlignment="Left" Height="125" Margin="270,55,0,0"
VerticalAlignment="Top" Width="125" Grid.Row="1"/>
                <Rectangle x:Name="D3" HorizontalAlignment="Left" Height="125" Margin="395,55,0,0"
VerticalAlignment="Top" Width="128" Grid.Row="1"/>
                <Rectangle x:Name="E3" HorizontalAlignment="Left" Height="125" Margin="523,55,0,0"
VerticalAlignment="Top" Width="125" Grid.Row="1"/>
                <Rectangle x:Name="F3" HorizontalAlignment="Left" Height="125" Margin="648,55,0,0"
VerticalAlignment="Top" Width="126" Grid.Row="1"/>
                <Rectangle x:Name="G3" HorizontalAlignment="Left" Height="125" Margin="774,55,-20,0"
VerticalAlignment="Top" Width="126" Grid.Row="1"/>
                <!-- Row 2 -->
                <Rectangle x:Name="A2" HorizontalAlignment="Left" Height="126" Margin="20,0,0,126"
VerticalAlignment="Bottom" Width="125" Grid.Row="1"/>
```

```
                <Rectangle x:Name="B2" HorizontalAlignment="Left" Height="126" Margin="145,0,0,126"
VerticalAlignment="Bottom" Width="125" Grid.Row="1"/>
                <Rectangle x:Name="C2" HorizontalAlignment="Left" Height="126" Margin="270,0,0,126"
VerticalAlignment="Bottom" Width="125" Grid.Row="1"/>
                <Rectangle x:Name="D2" HorizontalAlignment="Left" Height="126" Margin="395,0,0,126"
VerticalAlignment="Bottom" Width="128" Grid.Row="1"/>
                <Rectangle x:Name="E2" HorizontalAlignment="Left" Height="126" Margin="523,0,0,126"
VerticalAlignment="Bottom" Width="125" Grid.Row="1"/>
                <Rectangle x:Name="F2" HorizontalAlignment="Left" Height="126" Margin="648,0,0,126"
VerticalAlignment="Bottom" Width="126" Grid.Row="1"/>
                <Rectangle x:Name="G2" HorizontalAlignment="Left" Height="124" Margin="774,180,-20,0"
VerticalAlignment="Top" Width="126" Grid.Row="1"/>
                <!-- Row 1 -->
                <Rectangle x:Name="A1" HorizontalAlignment="Left" Height="126" Margin="20,306,0,0"
VerticalAlignment="Top" Width="125" Grid.Row="1"/>
                <Rectangle x:Name="B1" HorizontalAlignment="Left" Height="126" Margin="145,306,0,0"
VerticalAlignment="Top" Width="125" Grid.Row="1"/>
                <Rectangle x:Name="C1" HorizontalAlignment="Left" Height="126" Margin="270,306,0,0"
VerticalAlignment="Top" Width="125" Grid.Row="1"/>
                <Rectangle x:Name="D1" HorizontalAlignment="Left" Height="126" Margin="395,306,0,0"
VerticalAlignment="Top" Width="128" Grid.Row="1"/>
                <Rectangle x:Name="E1" HorizontalAlignment="Left" Height="126" Margin="523,306,0,0"
VerticalAlignment="Top" Width="125" Grid.Row="1"/>
                <Rectangle x:Name="F1" HorizontalAlignment="Left" Height="126" Margin="648,306,0,0"
VerticalAlignment="Top" Width="126" Grid.Row="1"/>
                <Rectangle x:Name="G1" HorizontalAlignment="Left" Height="126" Margin="774,306,-20,0"
VerticalAlignment="Top" Width="126" Grid.Row="1"/>
            </Grid>

        </RelativePanel>
        <Rectangle StrokeThickness="5" HorizontalAlignment="Left" Margin="135,196,0,435" Width="365">
            <Rectangle.Fill>
                <ImageBrush ImageSource="Assets/OldPaperV2.png" Stretch="UniformToFill"/>
            </Rectangle.Fill>
        </Rectangle>
        <Rectangle StrokeThickness="5" HorizontalAlignment="Left" Margin="1475,196,0,435" Width="365">
            <Rectangle.Fill>
                <ImageBrush ImageSource="Assets/OldPaperV2.png" Stretch="UniformToFill"/>
            </Rectangle.Fill>
        </Rectangle>
        <Button x:Name="resignBtn1" Style="{StaticResource ButtonStyle1}" Click="resignBtn1_tapped"
Content="" Height="60" Margin="180,0,0,220" VerticalAlignment="Bottom" Width="120"
ToolTipService.ToolTip="Resign">
            <Button.Background>
                <ImageBrush Stretch="Fill" ImageSource="Assets/mini-btn-resign.png"/>
            </Button.Background>
        </Button>
        <Button x:Name="UndoBtn1" Style="{StaticResource ButtonStyle1}" Click="undoBtn_tapped"
Content="" Height="60" Margin="340,0,0,220" VerticalAlignment="Bottom" Width="120"
ToolTipService.ToolTip="Undo Move">
            <Button.Background>
                <ImageBrush Stretch="Fill" ImageSource="Assets/mini-btn-undo.png"/>
            </Button.Background>
        </Button>
        <Button x:Name="resignBtn2" Style="{StaticResource ButtonStyle1}" Click="resignBtn2_tapped"
Content="" Height="60" Margin="0,0,280,220" VerticalAlignment="Bottom" Width="120"
HorizontalAlignment="Right" ToolTipService.ToolTip="Resign">
            <Button.Background>
                <ImageBrush Stretch="UniformToFill" ImageSource="Assets/mini-btn-resign.png"/>
            </Button.Background>
        </Button>
        <Button x:Name="UndoBtn2" Style="{StaticResource ButtonStyle1}" Click="undoBtn_tapped"
Content="" Height="60" Margin="0,0,120,220" VerticalAlignment="Bottom" Width="120"
HorizontalAlignment="Right" FontFamily="Old English Text MT" Foreground="White" FontSize="26.667"
ToolTipService.ToolTip="Undo Move">
            <Button.Background>
                <ImageBrush Stretch="Fill" ImageSource="Assets/mini-btn-undo.png"/>
            </Button.Background>
        </Button>
        <Rectangle x:Name="player1Rectangle" StrokeThickness="5" HorizontalAlignment="Left"
Height="140" Margin="160,0,0,300" VerticalAlignment="Bottom" Width="320">
            <Rectangle.Fill>
                <ImageBrush Stretch="Uniform" ImageSource="Assets/playerPanel.png"/>
            </Rectangle.Fill>
        </Rectangle>
        <Rectangle  x:Name="player2Rectangle" StrokeThickness="5" HorizontalAlignment="Right"
Height="140" Margin="0,0,100,300" VerticalAlignment="Bottom" Width="320">
            <Rectangle.Fill>
                <ImageBrush Stretch="Uniform" ImageSource="Assets/playerPanel.png"/>
            </Rectangle.Fill>
        </Rectangle>
        <TextBlock x:Name="player1Text" HorizontalAlignment="Left" TextAlignment="Center" Height="38"
Margin="284,0,0,386" TextWrapping="Wrap"  VerticalAlignment="Bottom" Width="181" FontSize="18.667"
Text="" FontFamily="Assets/Fonts/VIKING-N.TTF#Viking-Normal" Foreground="White"/>
        <TextBlock x:Name="player2Text" HorizontalAlignment="Right" TextAlignment="Center" Height="37"
Margin="0,0,117,388" TextWrapping="Wrap" VerticalAlignment="Bottom" Width="181" FontSize="18.667"
Text="" Foreground="White" FontFamily="Assets/Fonts/VIKING-N.TTF#Viking-Normal"/>
        <TextBlock x:Name="p1TurnIndicator" HorizontalAlignment="Left" Height="45"
Margin="284,0,0,336" TextWrapping="Wrap" Text="" VerticalAlignment="Bottom" Width="181"
FontFamily="Assets/Fonts/VIKING-N.TTF#Viking-Normal" FontSize="21.333" TextAlignment="Center"
Foreground="White"/>
        <TextBlock x:Name="p2TurnIndicator" HorizontalAlignment="Right" Height="44"
Margin="0,0,117,339" TextWrapping="Wrap" Text="" VerticalAlignment="Bottom" Width="181"
FontSize="21.333" FontFamily="Assets/Fonts/VIKING-N.TTF#Viking-Normal" TextAlignment="Center"
Foreground="White"/>
        <ScrollViewer x:Name="scrollViewerLeft"  Margin="179,222,0,461" HorizontalAlignment="Left"
Width="263" HorizontalScrollMode="Disabled" VerticalScrollMode="Enabled">
            <TextBlock x:Name="p1MoveListText" HorizontalAlignment="Left" TextWrapping="Wrap" Text=""
FontSize="26.667" TextAlignment="Center" FontFamily="Segoe Script" FontWeight="Bold"
ManipulationMode="None" Width="263" Foreground="#FF3E2B09"/>
        </ScrollViewer>
        <ScrollViewer x:Name="scrollViewerRight"  Margin="1520,222,0,461" HorizontalAlignment="Left"
Width="263" HorizontalScrollMode="Disabled" VerticalScrollMode="Enabled">
            <TextBlock x:Name="p2MoveListText" IsColorFontEnabled="true"  HorizontalAlignment="Left"
TextWrapping="Wrap" Text="" FontSize="26.667" TextAlignment="Center" FontFamily="Segoe Script"
FontWeight="Bold" ManipulationMode="None" Width="263" Foreground="#FF3E2B09"/>
        </ScrollViewer>


        <Image x:Name="image" HorizontalAlignment="Left" Height="103" Margin="178,0,0,321"
VerticalAlignment="Bottom" Width="101" Source="Assets/PieceBlack.png"/>
        <Image x:Name="image_Copy" HorizontalAlignment="Right" Height="104" Margin="0,0,303,321"
VerticalAlignment="Bottom" Width="102" Source="Assets/PieceWhite.png"/>
        <TextBlock x:Name="winText" IsHitTestVisible="False" HorizontalAlignment="Left"
Margin="640,80,0,0" TextWrapping="Wrap" VerticalAlignment="Top" Height="40" Width="640"
TextAlignment="Center" FontSize="32" FontFamily="Assets/Fonts/VIKING-N.TTF#Viking-Normal"
Foreground="#FFFFA200" IsDoubleTapEnabled="False" IsHoldingEnabled="False" IsTapEnabled="False"
IsRightTapEnabled="False" ManipulationMode="None" d:LayoutOverrides="VerticalAlignment"/>
        <Rectangle Height="20" StrokeThickness="10" VerticalAlignment="Top" Fill="#FF6B2509" />
        <Button x:Name="fullscreen_btn" Style="{StaticResource topbarButtonStyle}" Content=""
Click="fullscreen_btn_clicked" HorizontalAlignment="Right" Height="18" VerticalAlignment="Top"
Width="18" Margin="0,1,10,0">
            <Button.Background>
                <ImageBrush ImageSource="Assets/fullscreen-arrows.png" Stretch="Uniform"/>
            </Button.Background>
        </Button>
        <Button x:Name="back_btn" Style="{StaticResource topbarButtonStyle}" Content=""
Click="back_btn_clicked" HorizontalAlignment="Left" Height="20" VerticalAlignment="Top" Width="20">
            <Button.Background>
                <ImageBrush ImageSource="Assets/back-arrow.png" Stretch="Uniform"/>
            </Button.Background>
        </Button>
        <TextBlock x:Name="timer_text" IsHitTestVisible="False" HorizontalAlignment="Left"
Margin="665,939,0,0" TextWrapping="Wrap" VerticalAlignment="Top" Height="40" Width="640"
TextAlignment="Center" FontSize="32" FontFamily="Assets/Fonts/VIKING-N.TTF#Viking-Normal"
Foreground="#FFFFA200" IsDoubleTapEnabled="False" IsHoldingEnabled="False" IsTapEnabled="False"
IsRightTapEnabled="False" ManipulationMode="None"/>
        <TextBlock x:Name="player_timer_text" IsHitTestVisible="False" HorizontalAlignment="Left"
Margin="160,906,0,0" TextWrapping="Wrap" VerticalAlignment="Top" Height="40" Width="320"
TextAlignment="Center" FontSize="32" FontFamily="Assets/Fonts/VIKING-N.TTF#Viking-Normal"
Foreground="#FFFFA200" IsDoubleTapEnabled="False" IsHoldingEnabled="False" IsTapEnabled="False"
IsRightTapEnabled="False" ManipulationMode="None"/>
        <TextBlock x:Name="cpu_timer_text" IsHitTestVisible="False" HorizontalAlignment="Left"
Margin="1500,906,0,0" TextWrapping="Wrap" VerticalAlignment="Top" Height="40" Width="320"
TextAlignment="Center" FontSize="32" FontFamily="Assets/Fonts/VIKING-N.TTF#Viking-Normal"
Foreground="#FFFFA200" IsDoubleTapEnabled="False" IsHoldingEnabled="False" IsTapEnabled="False"
IsRightTapEnabled="False" ManipulationMode="None"/>
```

```
    </Grid>
</Page>
```

3.1.2.2. Code-Behind

```
using System;
using System.Collections;
using System.Threading.Tasks;
using Windows.UI;
using Windows.UI.ViewManagement;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Shapes;
using Windows.UI.Xaml.Media.Imaging;

using VikingChess.Model;
using Windows.UI.Xaml.Navigation;
using System.Collections.Generic;
using System.Diagnostics;
using System.ComponentModel;

namespace VikingChess
{
    /// <summary>
    /// The page responsible for displaying and controlling the Ard Rí gametype
    /// </summary>
    public sealed partial class ArdRiPage : Page
    {
        private Brush stationaryStrokeBrush;
        private Brush transformingStrokeBrush = new SolidColorBrush(Color.FromArgb(0xFF, 0x57, 0x13,
0x01));
        private double strokeThickness;
        private double transformingStrokeThickness = 2;
        private ImageBrush blackImageBrush = new ImageBrush();
        private ImageBrush whiteImageBrush = new ImageBrush();
        private ImageBrush kingImageBrush = new ImageBrush();
        private Board board = new Board(7, "Ard Ri");
        private Rectangle[] rectArr;
        private Rectangle lastTappedRect;
        private Game game;
        private List<String> p1MoveList;
        private List<String> p2MoveList;
        private bool checkMate = false;
        private Tuple<string, string, string, string, string> parameters;
        private Tuple<Square, Piece, Square> cpuMovesTuple;
        private Board mctsBoard;

        private double msCount = 0;
        private double secondCount = 0;
        private double minuteCount = 0;

        private double p_msCount = 0;
        private double p_secondCount = 0;
        private double p_minuteCount = 0;

        private double c_msCount = 0;
        private double c_secondCount = 0;
        private double c_minuteCount = 0;

        private DateTime gameStarted;
        private DispatcherTimer gameTimer = new DispatcherTimer();
        private DateTime playerOneStarted = DateTime.UtcNow;
        private DispatcherTimer playerOneTimer = new DispatcherTimer();
        private DateTime playerTwoStarted = DateTime.UtcNow;
        private DispatcherTimer playerTwoTimer = new DispatcherTimer();

        private BackgroundWorker bw = new BackgroundWorker();

        /// <summary>
        /// Constructor for ArdRiPage. Initializes the component, background worker and timers
        /// </summary>
        public ArdRiPage()
        {
            InitializeComponent();
            // Initalize the background worker.
            bw.DoWork += new DoWorkEventHandler(bw_DoWork);
            bw.WorkerSupportsCancellation = true;
            bw.RunWorkerCompleted += new RunWorkerCompletedEventHandler(bw_RunWorkerCompleted);

            // Initialize Game Timer
            gameTimer.Interval = new TimeSpan(0, 0, 0, 0, 1); ;
            gameTimer.Tick += new EventHandler<object>(gameTimer_Tick);

            // Initialize Player 1 timer
            playerOneTimer.Interval = new TimeSpan(0, 0, 0, 0, 1); ;
            playerOneTimer.Tick += new EventHandler<object>(playerOneTimer_Tick);

            // Initialize Player 2 timer
            playerTwoTimer.Interval = new TimeSpan(0, 0, 0, 0, 1); ;
            playerTwoTimer.Tick += new EventHandler<object>(playerTwoTimer_Tick);

            // Set the Start time of the Game Timer and Player 1 Timer
            gameStarted = DateTime.UtcNow;
            playerOneStarted = DateTime.UtcNow;

            // Start the Game Timer and Player 1 Timer
            gameTimer.Start();
            playerOneTimer.Start();
        }

        /// <summary>
        /// Sets up the base parameters for the game
        /// </summary>
        /// <param name="e"></param>
        protected override void OnNavigatedTo(NavigationEventArgs e)
        {
            // get configuration and create a new game with this configuration
            parameters = e.Parameter as Tuple<string, string, string, string, string>;
            game = new Game(board, parameters);

            // Set Player names
            player1Text.Text = game.getPlayer1().getName();
            player2Text.Text = game.getPlayer2().getName();

            // Hide the scrollbars for the movelists
            scrollViewerLeft.VerticalScrollBarVisibility = ScrollBarVisibility.Hidden;
            scrollViewerRight.VerticalScrollBarVisibility = ScrollBarVisibility.Hidden;

            // Get all rectangles on the board
            rectArr = new Rectangle[] { A1, A2, A3, A4, A5, A6, A7,
                                        B1, B2, B3, B4, B5, B6, B7,
                                        C1, C2, C3, C4, C5, C6, C7,
                                        D1, D2, D3, D4, D5, D6, D7,
                                        E1, E2, E3, E4, E5, E6, E7,
                                        F1, F2, F3, F4, F5, F6, F7,
                                        G1, G2, G3, G4, G5, G6, G7
                                        };

            // Apply the event handlers to each rectangle
            for (int i = 0; i < rectArr.Length; ++i)
            {
                rectArr[i].Tapped += rectangle_tapped;
            }

            // Bind the Rectangles to their corresponding squares
            bindRectangles();
            // Render the pieces on screen
            displayPieces(rectArr, board);
            displayMoveNotation();
            notifyPlayer();

            // Make a CPU move if first player is CPU
            if (bw.IsBusy != true)
            {
                mctsBoard = new Board(board);
                bw.RunWorkerAsync();
            }
        }

        /// <summary>
        /// The DoWork method for the <see cref="BackgroundWorker"/>. It runs the MCTS algorithm in
the background and
        /// retrieves the moveTuple for the CPU player for their current turn.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void bw_DoWork(object sender, DoWorkEventArgs e)
        {
            if (bw.CancellationPending)
            {
                e.Cancel = true;
            }
            else
            {
                // If the current player is a CPU player then get the move tuple for their current
turn.
                if (game.getCurrentPlayer().getType().Equals(Enums.PlayerType.CPU) && game.checkWin()
== false)
                {
                    cpuMovesTuple = null;
                    // Make CPU Move
                    cpuMovesTuple = game.getCurrentPlayer().makeMCTSMove(mctsBoard);
                }
            }
        }

        /// <summary>
        /// Runs once the backgroundWorker completes and processes the rest of the CPU player's move.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void bw_RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)
        {
            if (game.getCurrentPlayer().getType().Equals(Enums.PlayerType.CPU) && game.checkWin() ==
false)
            {
                moveCPU(game.getCurrentPlayer(), cpuMovesTuple);
            }
        }

        /// <summary>
        /// Controls the updating and display of the game timer.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        void gameTimer_Tick(object sender, object e)
        {
            // If the ms counter reaches 999, reset it and update the second count
            if (msCount >= 999)
            {
                msCount = 0;
                gameStarted = DateTime.UtcNow;
                secondCount++;
                // If the CPU is currently making their move, update the winText to reflect that.
                if (game.getCurrentPlayer() != null &&
game.getCurrentPlayer().getType().Equals(Enums.PlayerType.CPU))
                {
                    if (winText.Text.Equals(""))
                    {
                        winText.Text = game.getCurrentPlayer().getName() + " is thinking .";
                    }
                    else if (winText.Text.Equals(game.getCurrentPlayer().getName() + " is thinking
."))
                    {
                        winText.Text = game.getCurrentPlayer().getName() + " is thinking . .";
                    }
                    else if (winText.Text.Equals(game.getCurrentPlayer().getName() + " is thinking .
."))
                    {
                        winText.Text = game.getCurrentPlayer().getName() + " is thinking . . .";
                    }
                    else if (winText.Text.Equals(game.getCurrentPlayer().getName() + " is thinking . .
."))
                    {
                        winText.Text = game.getCurrentPlayer().getName() + " is thinking .";
                    }
                }
                else
                {
                    winText.Text = "";
                }
            }
            else
            {
                msCount = (int)(DateTime.UtcNow - gameStarted).TotalMilliseconds;
            }
            if (secondCount == 60)
            {
                secondCount = 0;
                minuteCount++;
            }
            String timeString = minuteCount + " : " + secondCount + " : " + (int)(msCount / 100);
            timer_text.Text = timeString;
        }

        /// <summary>
        /// Controls the updating and display of the player one timer
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void playerOneTimer_Tick(object sender, object e)
        {
            if (p_msCount >= 999)
            {
                p_msCount = 0;
                playerOneStarted = DateTime.UtcNow;
                p_secondCount++;
            }
            else
            {
                p_msCount = (int)(DateTime.UtcNow - playerOneStarted).TotalMilliseconds;
            }
            if (p_secondCount == 60)
            {
                p_secondCount = 0;
                p_minuteCount++;
            }
            String timeString = p_minuteCount + " : " + p_secondCount + " : " + (int)(p_msCount /
100);
            player_timer_text.Text = timeString;
        }

        /// <summary>
        /// Controls the updating and display of the player two timer
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void playerTwoTimer_Tick(object sender, object e)
        {
            if (c_msCount >= 999)
            {
                c_msCount = 0;
                playerTwoStarted = DateTime.UtcNow;
                c_secondCount++;
            }
            else
            {
                c_msCount = (int)(DateTime.UtcNow - gameStarted).TotalMilliseconds;
            }
            if (c_secondCount == 60)
            {
                c_secondCount = 0;
                c_minuteCount++;
            }
            String timeString = c_minuteCount + " : " + c_secondCount + " : " + (int)(c_msCount /
100);
            cpu_timer_text.Text = timeString;
        }

        /// <summary>
        /// Resets the PlayerOneTimer
        /// </summary>
        private void resetPlayerOneTimer()
        {
            p_msCount = 0;
            p_secondCount = 0;
            p_minuteCount = 0;
            playerOneStarted = DateTime.UtcNow;
            player_timer_text.Text = "";
        }

        /// <summary>
        /// Resets the PlayerTwoTimer
        /// </summary>
        private void resetPlayerTwoTimer()
        {
            c_msCount = 0;
            c_secondCount = 0;
            c_minuteCount = 0;
            playerTwoStarted = DateTime.UtcNow;
            cpu_timer_text.Text = "";

        }

        /// <summary>
        /// Sets the DataContext of the Rectangles to the corresponding squares
        /// </summary>
        private void bindRectangles()
        {

            int row = board.GetSize()-2;
            int col = 1;
            int i = 0;

            // Loops through each square on the board and binds each rectangle to its corresponding
square
            for (col = 1; col <= board.GetSize() - 2; ++col)
            {
                for (row = board.GetSize() - 2; row > 0; --row)
                {
                    rectArr[i].DataContext = board.GetSquare(row, col);
                    ++i;
                }
            }
        }

        #region display methods

        /// <summary>
        /// Refreshes the display of the board, displaying the appropriate image for each square
        /// </summary>
        /// <param name="rectArr">The array containing all rectangles</param>
        /// <param name="board">The current state of the <see cref="Board"/></param>
        private void displayPieces(Rectangle[] rectArr, Board board)
        {
            int row = board.GetSize() - 2;
            int col = 1;
            int i = 0;

            blackImageBrush.ImageSource = new BitmapImage(new Uri(this.BaseUri,
"/Assets/PieceBlack.png"));
            whiteImageBrush.ImageSource = new BitmapImage(new Uri(this.BaseUri,
"/Assets/PieceWhite.png"));
            kingImageBrush.ImageSource = new BitmapImage(new Uri(this.BaseUri,
"/Assets/PieceKing.png"));

            // Loops through each square on the board and binds each rectangle to its corresponding
square
```

```
            for (col = 1; col <= board.GetSize() - 2; ++col)
            {
                for (row = board.GetSize() - 2; row > 0; --row)
                {
                    Square square = board.GetSquare(row, col);

                    // If there's a piece on the square at the row and column
                    if (square.getPiece() != null)
                    {
                        Piece piece = board.GetSquare(row, col).getPiece();

                        if (piece.getType().Equals(Enums.PieceType.PAWN))
                        {
                            if (piece.getColor().Equals(Enums.Color.BLACK))
                            {
                                if (rectArr[i].Fill != blackImageBrush)
                                    rectArr[i].Fill = blackImageBrush;
                            }
                            else if (piece.getColor().Equals(Enums.Color.WHITE))
                            {
                                if (rectArr[i].Fill != whiteImageBrush)
                                    rectArr[i].Fill = whiteImageBrush;
                            }
                        }
                        // If the piece is a King
                        else if (piece.getType().Equals(Enums.PieceType.KING))
                        {
                            if (rectArr[i].Fill != kingImageBrush)
                                rectArr[i].Fill = kingImageBrush;
                        }
                    }
                    // If there's no piece on the square
                    else if (square.getPiece() == null)
                    {
                        rectArr[i].Fill = null;

                    }
                    rectArr[i].Stroke = null;
                    rectArr[i].RadiusX = 0;
                    rectArr[i].RadiusY = 0;
                    rectArr[i].Opacity = 1;
                    ++i;
                }
            }
        }

        /// <summary>
        /// Applies a stroke to the current player's pieces to highlight them.
        /// </summary>
        private void highlightPieces()
        {
            Brush highlightStrokeBrush = new SolidColorBrush(Color.FromArgb(0xFF, 0xFF, 0xA2, 0x00));
            double strokeThickness = 2;

            int row = board.GetSize() - 2;
            int col = 1;
            int i = 0;

            for (col = 1; col <= board.GetSize() - 2; ++col)
            {
                for (row = board.GetSize() - 2; row > 0; --row)
                {
                    Piece piece = board.GetSquare(row, col).getPiece();
                    Player currentPlayer = game.getCurrentPlayer();

                    if (piece != null && currentPlayer.getColor().Equals(piece.getColor()))
                    {
                        rectArr[i].Stroke = highlightStrokeBrush;
                        rectArr[i].StrokeThickness = strokeThickness;
                        rectArr[i].RadiusX = 100;
                        rectArr[i].RadiusY = 100;
                    }
                    ++i;
                }
            }
        }

        /// <summary>
        /// Displays to the current player that it is their turn to move.
        /// </summary>
        private void notifyPlayer()
        {
            // If the current player is the attacker (Player 1)
            if (game.getCurrentPlayer().getColor() == Enums.Color.BLACK)
            {
                // Set the active colours for Player 1
                player1Rectangle.Stroke = new SolidColorBrush(Color.FromArgb(0xFF, 0xFF, 0xA2, 0x00));
                p1TurnIndicator.Text = "Your Turn";

                // Reset the colors for Player 2
                player2Rectangle.Stroke = null;
                p2TurnIndicator.Text = "";

                highlightPieces();
            }
            // Otherwise, the current player is the defender (Player 2)
            else
            {
                // Set the active colors for Player 2
                player2Rectangle.Stroke = new SolidColorBrush(Color.FromArgb(0xFF, 0xFF, 0xA2, 0x00));
                p2TurnIndicator.Text = "Your Turn";

                // Reset the colors for Player 1
                player1Rectangle.Stroke = null;
                p1TurnIndicator.Text = "";

                highlightPieces();
            }
        }

        /// <summary>
        /// Displays the squares that a piece can move to
        /// </summary>
        /// <param name="rect">The tapped rectangle</param>
        private void displayLegalMoves(Rectangle rect)
        {
            Square originSquare = (Square)rect.DataContext;
            Piece piece = originSquare.getPiece();
            Player currentPlayer = game.getCurrentPlayer();

            // If there's a piece on the the tapped rectangle and it matches the color of the current
player
            if (piece != null && currentPlayer.getColor().Equals(piece.getColor()))
            {
                // Generate the legal moves for that piece
                List<Square> legalMoves = piece.generateLegalMoves(board);

                int row = board.GetSize() - 2;
                int col = 1;
                int i = 0;

                // Loop through each square on the board and color the squares the piece can move to
                for (col = 1; col <= board.GetSize() - 2; ++col)
                {
                    for (row = board.GetSize() - 2; row > 0; --row)
                    {
                        foreach (var move in legalMoves)
                        {
                            if (board.GetSquare(row, col) == move)
                            {
                                rectArr[i].Fill = new SolidColorBrush(Color.FromArgb(0xFF, 0x57, 0x13,
0x01));

                                rectArr[i].RadiusX = 100;
                                rectArr[i].RadiusY = 100;
                                rectArr[i].StrokeThickness = 2;
                                rectArr[i].Stroke = new SolidColorBrush(Color.FromArgb(0xFF, 0xFF,
0xA2, 0x00));

                                rectArr[i].Opacity = 0.6;
                            }
                        }
                        ++i;
                    }
                }
            }
        }

        /// <summary>
        /// Displays the list of moves taken by each player
        /// </summary>
        private void displayMoveNotation()
        {
            p1MoveListText.Text = "History\n";
            p2MoveListText.Text = "History\n";

            p1MoveList = game.getPlayer1().getMoveList();
            p2MoveList = game.getPlayer2().getMoveList();

            foreach (var move in p1MoveList)
            {
                p1MoveListText.Text += move;
            }
            foreach (var move in p2MoveList)
            {
                p2MoveListText.Text += move;
            }
        }

        #endregion

        #region input methods

        /// <summary>
        /// Ends the game with Player 2 as the winner.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private async void resignBtn1_tapped(object sender, RoutedEventArgs e)
        {
            playerOneTimer.Stop();
            playerTwoTimer.Stop();
            bw.CancelAsync();
            // If the current player is player 1
            if (game.getCurrentPlayer().getColor() == Enums.Color.BLACK)
            {
                game.PlaySound("btn_click.wav");
                game.getPlayer2().setWin(true);
                winText.Text = "Player 2 Wins";
                game.PlaySound("victory_fanfare.wav");
                await Task.Delay(TimeSpan.FromSeconds(5));
                this.Frame.Navigate(typeof(MainPage));
            }
        }

        /// <summary>
        /// Returns the user to the previous page
        /// </summary>
        /// <param name="sender"><see cref="back_btn"/></param>
        /// <param name="e"></param>
        private void back_btn_clicked(object sender, RoutedEventArgs e)
        {
            bw.CancelAsync();
            if (this.Frame.CanGoBack)
            {
                this.Frame.GoBack();
            }
        }

        /// <summary>
        /// Enables and disables fullscreen mode
        /// </summary>
        /// <param name="sender"><see cref="fullscreen_btn"/></param>
        /// <param name="e"></param>
```

```
        private void fullscreen_btn_clicked(object sender, RoutedEventArgs e)
        {
            ApplicationView view = ApplicationView.GetForCurrentView();
            ImageBrush fullscreenImg = new ImageBrush();

            bool isInFullScreenMode = view.IsFullScreenMode;

            if (isInFullScreenMode)
            {
                fullscreenImg.ImageSource = new BitmapImage(new Uri(this.BaseUri, "/Assets/fullscreen-
arrows.png"));
                view.ExitFullScreenMode();
            }
            else
            {
                fullscreenImg.ImageSource = new BitmapImage(new Uri(this.BaseUri, "/Assets/windowed-
arrows.png"));
                view.TryEnterFullScreenMode();
            }
            this.fullscreen_btn.Background = fullscreenImg;
        }

        /// <summary>
        /// Ends the game with Player 1 as the winner
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private async void resignBtn2_tapped(object sender, RoutedEventArgs e)
        {
            playerOneTimer.Stop();
            playerTwoTimer.Stop();
            bw.CancelAsync();
            // If the current player is player 2
            if (game.getCurrentPlayer().getColor().Equals(Enums.Color.WHITE))
            {
                game.PlaySound("btn_click.wav");
                game.getPlayer1().setWin(true);
                winText.Text = "Player 1 Wins";
                game.PlaySound("victory_fanfare.wav");
                await Task.Delay(TimeSpan.FromSeconds(5));
                this.Frame.Navigate(typeof(MainPage));
            }
        }

        /// <summary>
        /// Undoes the last move taken
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void undoBtn_tapped(object sender, RoutedEventArgs e)
        {
            if (game.getCurrentPlayer().getColor().Equals(Enums.Color.BLACK))
            {
                playerOneTimer.Stop();
                resetPlayerTwoTimer();
                playerTwoTimer.Start();

            }
            else
            {
                playerTwoTimer.Stop();
                resetPlayerOneTimer();
                playerOneTimer.Start();
            }
            bw.CancelAsync();
            game.PlaySound("btn_click.wav");
            board = game.undoMove();
            bindRectangles();
            displayPieces(rectArr, board);
            displayMoveNotation();
            notifyPlayer();
            lastTappedRect = null;
        }
        /// <summary>
        /// Performs an action based on the rectangle that was tapped.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private async void rectangle_tapped(object sender, RoutedEventArgs e)
        {
            displayPieces(rectArr, board);
            Rectangle rect = sender as Rectangle;
            Square tappedSquare = (Square)rect.DataContext;
            Piece piece = tappedSquare.getPiece();
            Player currentPlayer = game.getCurrentPlayer();


            if (currentPlayer.getType().Equals(Enums.PlayerType.HUMAN) && !game.checkWin())
            {
                // If a rectangle was previously tapped
                if (rect != lastTappedRect && lastTappedRect != null)
                {
                    // and the rectangle's bound square contains a piece of the same color as the 
current player
                    if (piece != null && currentPlayer.getColor().Equals(piece.getColor()))
                    {
                        displayPieces(rectArr, board);
                        // Reset the stroke of the previously tapped square
                        lastTappedRect.Stroke = stationaryStrokeBrush;
                        lastTappedRect.StrokeThickness = strokeThickness;
                        stationaryStrokeBrush = rect.Stroke;
                        strokeThickness = rect.StrokeThickness;

                        // Set the stroke of the currently tapped square
                        rect.StrokeThickness = transformingStrokeThickness;
                        rect.Stroke = transformingStrokeBrush;
                        // Display any legal moves for the currently selected piece
                        displayLegalMoves(rect);
                        lastTappedRect = rect;

                        // Play a sound notifying the player that a piece has been selected.
                        game.PlaySound("piece_select.wav");
                    }
                    // If the square is not a corner or throne square and there is no piece on the 
tapped square
                    else if (piece == null && 
!(tappedSquare.getSquareType().Equals(Enums.SquareType.CORNER) || 
tappedSquare.getSquareType().Equals(Enums.SquareType.THRONE)))
                    {
                        // Set the square containing the selected piece equal to the last tapped 
square
                        Square originSquare = (Square)lastTappedRect.DataContext;
                        // Set the selected piece equal to the piece on the selected square
                        Piece selectedPiece = originSquare.getPiece();

                        if (selectedPiece != null)
                        {
                            // Make Human Move
                            processMove(tappedSquare, originSquare, selectedPiece);
                            await Task.Delay(TimeSpan.FromSeconds(2));
                            currentPlayer = game.getCurrentPlayer();

                            if (bw.IsBusy != true)
                            {
                                mctsBoard = new Board(board);
                                bw.RunWorkerAsync();
                            }

                        }
                    }
                }
                // If no rectangle was previously tapped
                else
                {
                    // If the tapped square contains a piece and it is of the same color as the 
current player.
                    if (piece != null && currentPlayer.getColor().Equals(piece.getColor()))
                    {
                        // Backup the default stroke for the square.
                        stationaryStrokeBrush = rect.Stroke;
                        strokeThickness = rect.StrokeThickness;

                        // Set the stroke of the square
                        rect.StrokeThickness = transformingStrokeThickness;
                        rect.Stroke = transformingStrokeBrush;
                        // Display the legal moves for the piece on that square
                        displayLegalMoves(rect);
                        // Set this square as the last tapped rectangle
                        lastTappedRect = rect;
                        // Play a sound indicating that a piece has been selected.
                        game.PlaySound("piece_select.wav");
                    }
                }
            }
        }

        /// <summary>
        /// Processes a move for the CPU player
        /// </summary>
        /// <param name="currentPlayer">The current <see cref="Player"/> of the <see
cref="Game"/></param>
        /// <param name="cpuMovesTuple">A <see cref="Tuple{T1, T2, T3}"/> containing the origin <see
cref="Square"/>,
        /// destination <see cref="Square"/> and selected <see cref="Piece"/> to move.</param>
        private async void moveCPU(Player currentPlayer, Tuple<Square, Piece, Square> cpuMovesTuple)
        {
            Square origin = getRectSquare(cpuMovesTuple.Item1);
            Piece selectedCPUPiece = origin.getPiece();
            Square destination = getRectSquare(cpuMovesTuple.Item3);

            Rectangle rect = getCPURect(origin);
            // Set the stroke of the currently tapped square
            rect.StrokeThickness = transformingStrokeThickness;
            rect.Stroke = transformingStrokeBrush;
            // Play a sound notifying the player that a piece has been selected.
            game.PlaySound("piece_select.wav");
            // Display any legal moves for the currently selected piece
            displayLegalMoves(rect);
            await Task.Delay(TimeSpan.FromSeconds(2));
            processMove(destination, origin, selectedCPUPiece);
        }

        /// <summary>
        /// Gets the <see cref="Rectangle"/> Square that matches the <see cref="Square"/> passed in.
        /// </summary>
        /// <param name="square">The <see cref="Square"/> to match against.</param>
        /// <returns></returns>
        private Square getRectSquare(Square square)
        {
            for (int i = 0; i < rectArr.Length; ++i)
            {
                Square rectSquare = rectArr[i].DataContext as Square;
                if (rectSquare.getFile().Equals(square.getFile()) &&
rectSquare.getRank().Equals(square.getRank()))
                {
                    return rectSquare;
                }
            }
            return null;
        }

        /// <summary>
        /// Checks the <see cref="Game"/> to see if it has been one and displays the status to the
<see cref="Player"/>
        /// </summary>
        private async void checkWinAndDisplay()
        {
            // Check for a win
            if (game.checkWin() || checkMate)
            {
                // And display the appropriate win message if a player has won.
                if (game.getPlayer1().getWin())
                {
                    winText.Text = game.getPlayer1().getName() + " Wins!";
                }
                else if (game.getPlayer2().getWin())
                {
                    winText.Text = game.getPlayer2().getName() + " Wins!";
                }
                // Play a jingle to signal a win.
                game.PlaySound("victory_fanfare.wav");
                // Wait 5 seconds to allow the jingle to play.
                await Task.Delay(TimeSpan.FromSeconds(5));
                // Return to the previous screen.
                this.Frame.Navigate(typeof(MainPage));
            }
        }

        /// <summary>
        /// Checks to see if the King can reach one or more board edges on the next turn
        /// </summary>
        /// <param name="piece">The <see cref="Piece"/> passed into check against.</param>
        private async void canEscape(Piece piece)
        {
            int edgeCount = 0;
            List<Square> kingsMoves = piece.generateLegalMoves(board);
            for (int i = 0; i < kingsMoves.Count; ++i)
            {
                Square currSq = (Square)kingsMoves[i];
                if (currSq.getFile().Equals(Enums.File.A) || currSq.getFile().Equals(Enums.File.K) ||
currSq.getRank().Equals(Enums.Rank.One) || currSq.getRank().Equals(Enums.Rank.Eleven))
                {
                    ++edgeCount;
                }
            }
            if (edgeCount == 1)
            {
                winText.Text = "Check!";
                await Task.Delay(TimeSpan.FromSeconds(2));
                winText.Text = "";
            }
            else if (edgeCount > 1)
            {
                winText.Text = "Checkmate!";
                await Task.Delay(TimeSpan.FromSeconds(2));
                winText.Text = "";
                checkMate = true;
                checkWinAndDisplay();
            }
        }

        /// <summary>
        /// Returns the <see cref="Rectangle"/> that matches the <see cref="Square"/> passed in
        /// </summary>
        /// <param name="square">The square to compare against.</param>
        /// <returns></returns>
        private Rectangle getCPURect(Square square)
        {
            for (int i = 0; i < rectArr.Length; ++i)
            {
                if (rectArr[i].DataContext.Equals(square))
                {
                    return rectArr[i];
                }
            }
            return null;
        }

        /// <summary>
        /// Processes the moves made by a player
        /// </summary>
        /// <param name="destination">The desintation <see cref="Square"/> to move the <see
cref="Piece"/> to.</param>
        /// <param name="origin">The origin <see cref="Square"/> the <see cref="Piece"/> is moving
from.</param>
```

```
        /// <param name="piece">The <see cref="Piece"/> being moved.</param>
        private void processMove(Square destination, Square origin, Piece piece)
        {
            displayPieces(rectArr, board);
            // Move the piece to the currently tapped square
            game.movePiece(destination, origin, piece);
            // Check to see if the king can escape
            if (piece.getType().Equals(Enums.PieceType.KING))
            {
                canEscape(piece);
            }
            // Update the display of the pieces on the board
            displayPieces(rectArr, board);
            // Update the display of the move notation.
            displayMoveNotation();
            // Check for a win
            checkWinAndDisplay();
            // Notify the next player that the turn is completed.
            notifyPlayer();
            if (game.getCurrentPlayer().getColor().Equals(Enums.Color.WHITE))
            {
                playerOneTimer.Stop();
                resetPlayerTwoTimer();
                playerTwoTimer.Start();
            }
            else if (game.getCurrentPlayer().getColor().Equals(Enums.Color.BLACK))
            {
                playerTwoTimer.Stop();
                resetPlayerOneTimer();
                playerOneTimer.Start();
            }
        }
    }
    //////////////////
    /// End Class ///
    /////////////////
    #endregion
}
```

3.2. Main Page

3.2.1. Description

The Main Page is the entry point of the game.

3.2.2. Code

3.2.2.1. XAML

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:VikingChess"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:Model="using:VikingChess.Model"
    x:Class="VikingChess.MainPage"
    mc:Ignorable="d">

    <Page.Resources>
        <Style x:Key="ButtonStyle1" TargetType="Button">
            <Setter Property="Background" Value="{ThemeResource
SystemControlBackgroundBaseLowBrush}"/>
            <Setter Property="Foreground" Value="{ThemeResource
SystemControlForegroundBaseHighBrush}"/>
            <Setter Property="BorderBrush" Value="{ThemeResource
SystemControlForegroundTransparentBrush}"/>
            <Setter Property="BorderThickness" Value="{ThemeResource ButtonBorderThemeThickness}"/>
            <Setter Property="Padding" Value="8,4,8,4"/>
            <Setter Property="HorizontalAlignment" Value="Left"/>
            <Setter Property="VerticalAlignment" Value="Center"/>
            <Setter Property="FontFamily" Value="{ThemeResource ContentControlThemeFontFamily}"/>
            <Setter Property="FontWeight" Value="Normal"/>
            <Setter Property="FontSize" Value="{ThemeResource ControlContentThemeFontSize}"/>
            <Setter Property="UseSystemFocusVisuals" Value="True"/>
            <Setter Property="Template">
                <Setter.Value>
                    <ControlTemplate TargetType="Button">
                        <Grid x:Name="RootGrid" Background="{TemplateBinding Background}">
                            <VisualStateManager.VisualStateGroups>
                                <VisualStateGroup x:Name="CommonStates">
                                    <VisualState x:Name="Normal">
                                        <Storyboard>
                                            <PointerUpThemeAnimation 
Storyboard.TargetName="RootGrid"/>
                                        </Storyboard>
                                    </VisualState>
                                    <VisualState x:Name="PointerOver">
                                        <Storyboard>
                                            <ObjectAnimationUsingKeyFrames 
Storyboard.TargetProperty="BorderBrush" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0" Value="#e89300"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames 
Storyboard.TargetProperty="Foreground" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0" 
Value="{ThemeResource SystemControlHighlightBaseHighBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <PointerUpThemeAnimation 
Storyboard.TargetName="RootGrid"/>
                                        </Storyboard>
                                    </VisualState>
                                    <VisualState x:Name="Pressed">
                                        <Storyboard>
                                            <ObjectAnimationUsingKeyFrames 
Storyboard.TargetProperty="Foreground" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0" 
Value="{ThemeResource SystemControlHighlightBaseHighBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <PointerDownThemeAnimation 
Storyboard.TargetName="RootGrid"/>
                                        </Storyboard>
                                    </VisualState>
                                    <VisualState x:Name="Disabled">
                                        <Storyboard>
                                            <ObjectAnimationUsingKeyFrames 
Storyboard.TargetProperty="Background" Storyboard.TargetName="RootGrid">
                                                <DiscreteObjectKeyFrame KeyTime="0" 
Value="{ThemeResource SystemControlBackgroundBaseLowBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames 
Storyboard.TargetProperty="Foreground" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0" 
Value="{ThemeResource SystemControlDisabledBaseLowBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames 
Storyboard.TargetProperty="BorderBrush" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0" 
Value="{ThemeResource SystemControlDisabledTransparentBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                        </Storyboard>
                                    </VisualState>
                                </VisualStateGroup>
                            </VisualStateManager.VisualStateGroups>
                            <ContentPresenter x:Name="ContentPresenter" 
AutomationProperties.AccessibilityView="Raw" BorderBrush="{TemplateBinding BorderBrush}" 
BorderThickness="{TemplateBinding BorderThickness}" ContentTemplate="{TemplateBinding 
ContentTemplate}" ContentTransitions="{TemplateBinding ContentTransitions}" Content="{TemplateBinding 
Content}" HorizontalContentAlignment="{TemplateBinding HorizontalContentAlignment}" 
Padding="{TemplateBinding Padding}" VerticalContentAlignment="{TemplateBinding 
VerticalContentAlignment}"/>
                        </Grid>
                    </ControlTemplate>
                </Setter.Value>
            </Setter>
        </Style>
        <Style x:Key="topbarButtonStyle" TargetType="Button">
            <Setter Property="Background" Value="{ThemeResource
SystemControlBackgroundBaseLowBrush}"/>
            <Setter Property="Foreground" Value="{ThemeResource
SystemControlForegroundBaseHighBrush}"/>
            <Setter Property="BorderBrush" Value="{ThemeResource
SystemControlForegroundTransparentBrush}"/>
            <Setter Property="BorderThickness" Value="{ThemeResource ButtonBorderThemeThickness}"/>
            <Setter Property="Padding" Value="8,4,8,4"/>
            <Setter Property="HorizontalAlignment" Value="Left"/>
            <Setter Property="VerticalAlignment" Value="Center"/>
            <Setter Property="FontFamily" Value="{ThemeResource ContentControlThemeFontFamily}"/>
            <Setter Property="FontWeight" Value="Normal"/>
            <Setter Property="FontSize" Value="{ThemeResource ControlContentThemeFontSize}"/>
            <Setter Property="UseSystemFocusVisuals" Value="True"/>
            <Setter Property="Template">
                <Setter.Value>
                    <ControlTemplate TargetType="Button">
                        <Grid x:Name="RootGrid" Background="{TemplateBinding Background}">
                            <VisualStateManager.VisualStateGroups>
                                <VisualStateGroup x:Name="CommonStates">
                                    <VisualState x:Name="Normal">
                                        <Storyboard>
                                            <PointerUpThemeAnimation
Storyboard.TargetName="RootGrid"/>
                                        </Storyboard>
                                    </VisualState>
                                    <VisualState x:Name="PointerOver">
                                        <Storyboard>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Foreground" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlHighlightBaseHighBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <PointerUpThemeAnimation
Storyboard.TargetName="RootGrid"/>
                                        </Storyboard>
                                    </VisualState>
                                    <VisualState x:Name="Pressed">
                                        <Storyboard>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Foreground" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlHighlightBaseHighBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <PointerDownThemeAnimation
Storyboard.TargetName="RootGrid"/>
                                        </Storyboard>
                                    </VisualState>
                                    <VisualState x:Name="Disabled">
                                        <Storyboard>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Background" Storyboard.TargetName="RootGrid">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlBackgroundBaseLowBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Foreground" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlDisabledBaseLowBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="BorderBrush" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlDisabledTransparentBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                        </Storyboard>
                                    </VisualState>
                                </VisualStateGroup>
                            </VisualStateManager.VisualStateGroups>
                            <ContentPresenter x:Name="ContentPresenter"
AutomationProperties.AccessibilityView="Raw" BorderBrush="{TemplateBinding BorderBrush}"
BorderThickness="{TemplateBinding BorderThickness}" ContentTemplate="{TemplateBinding
ContentTemplate}" ContentTransitions="{TemplateBinding ContentTransitions}" Content="{TemplateBinding
Content}" HorizontalContentAlignment="{TemplateBinding HorizontalContentAlignment}"
Padding="{TemplateBinding Padding}" VerticalContentAlignment="{TemplateBinding
VerticalContentAlignment}"/>
                        </Grid>
                    </ControlTemplate>
                </Setter.Value>
            </Setter>
        </Style>
    </Page.Resources>
    <Grid x:Name="grid" RenderTransformOrigin="0.5,0.5" >
        <Grid.RenderTransform>
            <CompositeTransform/>
        </Grid.RenderTransform>
        <VisualStateManager.VisualStateGroups>
            <VisualStateGroup x:Name="VisualStateGroup">
                <VisualStateGroup.Transitions>
                    <VisualTransition GeneratedDuration="0:0:1">
                        <VisualTransition.GeneratedEasingFunction>
                            <BackEase EasingMode="EaseOut"/>
                        </VisualTransition.GeneratedEasingFunction>
                    </VisualTransition>
                    <VisualTransition GeneratedDuration="0:0:1" To="_1024x768">
                        <VisualTransition.GeneratedEasingFunction>
                            <BackEase EasingMode="EaseOut"/>
                        </VisualTransition.GeneratedEasingFunction>
                    </VisualTransition>
                    <VisualTransition From="_1024x768" GeneratedDuration="0:0:1">
                        <VisualTransition.GeneratedEasingFunction>
                            <BackEase EasingMode="EaseOut"/>
                        </VisualTransition.GeneratedEasingFunction>
                    </VisualTransition>
                    <VisualTransition GeneratedDuration="0:0:1" To="_1920x1080">
                        <VisualTransition.GeneratedEasingFunction>
                            <BackEase EasingMode="EaseOut"/>
                        </VisualTransition.GeneratedEasingFunction>
                    </VisualTransition>
                    <VisualTransition From="_1920x1080" GeneratedDuration="0:0:1">
                        <VisualTransition.GeneratedEasingFunction>
                            <BackEase EasingMode="EaseOut"/>
                        </VisualTransition.GeneratedEasingFunction>
                    </VisualTransition>
                    <VisualTransition GeneratedDuration="0:0:1" To="_1280x720">
                        <VisualTransition.GeneratedEasingFunction>
                            <BackEase EasingMode="EaseOut"/>
                        </VisualTransition.GeneratedEasingFunction>
                    </VisualTransition>
                    <VisualTransition From="_1280x720" GeneratedDuration="0:0:1">
                        <VisualTransition.GeneratedEasingFunction>
                            <BackEase EasingMode="EaseOut"/>
                        </VisualTransition.GeneratedEasingFunction>
                    </VisualTransition>
                </VisualStateGroup.Transitions>
                <VisualState x:Name="_1024x768">
                    <VisualState.Setters>
                        <Setter Target="newGameBtn.(FrameworkElement.Margin)">
                            <Setter.Value>
                                <Thickness>40,80,0,0</Thickness>
                            </Setter.Value>
                        </Setter>
                        <Setter Target="howtoBtn.(FrameworkElement.Margin)">
                            <Setter.Value>
                                <Thickness>40,290,0,0</Thickness>
                            </Setter.Value>
                        </Setter>
                        <Setter Target="statsBtn.(FrameworkElement.Margin)">
                            <Setter.Value>
                                <Thickness>40,499,0,0</Thickness>
                            </Setter.Value>
                        </Setter>
                        <Setter Target="newGameBtn.(FrameworkElement.Width)" Value="942"/>
                        <Setter Target="newGameBtn.(FrameworkElement.Height)" Value="81"/>
                        <Setter Target="howtoBtn.(FrameworkElement.Width)" Value="942"/>
                        <Setter Target="howtoBtn.(FrameworkElement.Height)" Value="80"/>
                        <Setter Target="statsBtn.(FrameworkElement.Width)" Value="942"/>
                        <Setter Target="statsBtn.(FrameworkElement.Height)" Value="81"/>
                    </VisualState.Setters>
                    <VisualState.StateTriggers>
                        <AdaptiveTrigger MinWindowHeight="0" MinWindowWidth="768"/>
                    </VisualState.StateTriggers>
                </VisualState>
                <VisualState x:Name="_1920x1080">
                    <VisualState.StateTriggers>
                        <AdaptiveTrigger MinWindowWidth="1080"/>
                    </VisualState.StateTriggers>
                </VisualState>
                <VisualState x:Name="_1280x720">
                    <VisualState.Setters>
                        <Setter Target="newGameBtn.(FrameworkElement.Margin)">
                            <Setter.Value>
                                <Thickness>40,105,0,0</Thickness>
                            </Setter.Value>
                        </Setter>
                        <Setter Target="howtoBtn.(FrameworkElement.Margin)">
                            <Setter.Value>
                                <Thickness>40,285,0,0</Thickness>
                            </Setter.Value>
                        </Setter>
                        <Setter Target="statsBtn.(FrameworkElement.Margin)">
                            <Setter.Value>
                                <Thickness>40,480,0,0</Thickness>
                            </Setter.Value>
                        </Setter>
                    </VisualState.Setters>
                    <VisualState.StateTriggers>
                        <AdaptiveTrigger MinWindowHeight="0" MinWindowWidth="720"/>
                    </VisualState.StateTriggers>
                </VisualState>
                <VisualState x:Name="_1280x800">
                    <VisualState.Setters>
                        <Setter Target="newGameBtn.(FrameworkElement.Margin)">
                            <Setter.Value>
                                <Thickness>50,80,0,0</Thickness>
                            </Setter.Value>
                        </Setter>
                        <Setter Target="newGameBtn.(FrameworkElement.Width)" Value="920"/>
                        <Setter Target="newGameBtn.(FrameworkElement.Height)" Value="85"/>
                        <Setter Target="howtoBtn.(FrameworkElement.Margin)">
                            <Setter.Value>
                                <Thickness>50,246,0,0</Thickness>
                            </Setter.Value>
                        </Setter>
                        <Setter Target="howtoBtn.(FrameworkElement.Width)" Value="920"/>
                        <Setter Target="howtoBtn.(FrameworkElement.Height)" Value="85"/>
                        <Setter Target="statsBtn.(FrameworkElement.Margin)">
                            <Setter.Value>
                                <Thickness>50,410,0,0</Thickness>
                            </Setter.Value>
                        </Setter>
                        <Setter Target="statsBtn.(FrameworkElement.Width)" Value="920"/>
                        <Setter Target="statsBtn.(FrameworkElement.Height)" Value="86"/>
                    </VisualState.Setters>
                </VisualState>
            </VisualStateGroup>
        </VisualStateManager.VisualStateGroups>
        <Grid.Background>
            <ImageBrush ImageSource="Assets/bg-wood.png"/>
        </Grid.Background>
        <RelativePanel>
            <Button x:Name="newGameBtn" Style="{StaticResource ButtonStyle1}" Click="newGameBtn_Click"
HorizontalAlignment="Left" Margin="360,0,0,-720" VerticalAlignment="Bottom" Width="1200" FontSize="48"
FontFamily="Sylfaen" BorderBrush="#FF000105" BorderThickness="5" Foreground="{x:Null}"
FontStretch="Expanded" FontWeight="Bold" RenderTransformOrigin="0.5,0.5" Height="100"
d:LayoutOverrides="VerticalAlignment">
                <Button.RenderTransform>
                    <CompositeTransform/>
                </Button.RenderTransform>
                <Button.Background>
                    <ImageBrush Stretch="Uniform" ImageSource="Assets/play-btn.png"/>
                </Button.Background>
            </Button>
            <Button x:Name="howtoBtn" Style="{StaticResource ButtonStyle1}" Click="howToBtn_Click"
HorizontalAlignment="Left" Margin="360,740,0,0" VerticalAlignment="Top" Width="1200" FontSize="48"
FontFamily="Sylfaen" BorderBrush="#FF000105" BorderThickness="5" Foreground="{x:Null}"
FontStretch="Expanded" FontWeight="Bold" Height="100" RenderTransformOrigin="0.5,0.5">
                <Button.RenderTransform>
                    <CompositeTransform/>
                </Button.RenderTransform>
                <Button.Background>
                    <ImageBrush Stretch="Uniform" ImageSource="Assets/howto-btn.png"/>
                </Button.Background>
            </Button>
            <Button x:Name="aboutBtn" Style="{StaticResource ButtonStyle1}" Click="aboutBtn_Click"
HorizontalAlignment="Left" Margin="360,860,0,0" VerticalAlignment="Top" Width="1200" FontSize="48"
FontFamily="Sylfaen" BorderBrush="#FF000105" BorderThickness="5" Foreground="{x:Null}"
FontStretch="Expanded" FontWeight="Bold" Height="100">
                <Button.Background>
                    <ImageBrush Stretch="Uniform" ImageSource="Assets/AboutButton.png"/>
                </Button.Background>
            </Button>
        </RelativePanel>
        <Rectangle Height="20" StrokeThickness="10" VerticalAlignment="Top" Fill="#FF6B2509"
Width="1920" />
        <Button x:Name="fullscreen_btn" Style="{StaticResource topbarButtonStyle}" Content=""
Click="fullscreen_btn_clicked" HorizontalAlignment="Right" Height="18" VerticalAlignment="Top"
Width="18" Margin="0,1,10,0">
            <Button.Background>
                <ImageBrush ImageSource="Assets/fullscreen-arrows.png" Stretch="Uniform"/>
            </Button.Background>
        </Button>
        <Button x:Name="back_btn" Style="{StaticResource topbarButtonStyle}" Content=""
Click="back_btn_clicked"  HorizontalAlignment="Left" Height="20" VerticalAlignment="Top" Width="20">
            <Button.Background>
                <ImageBrush ImageSource="Assets/back-arrow.png" Stretch="Uniform"/>
            </Button.Background>
        </Button>

        <ProgressBar Margin="360,60,360,0" VerticalAlignment="Top" Height="100" x:Name="LoadingBar"
Visibility="Collapsed" IsEnabled="False" IsIndeterminate="true" HorizontalAlignment="Stretch"/>
        <Rectangle Height="360" Margin="780,40,780,0" VerticalAlignment="Top">
            <Rectangle.Fill>
                <ImageBrush Stretch="Uniform" ImageSource="Assets/VikingLogoMainPage.png"/>
            </Rectangle.Fill>
        </Rectangle>
        <Rectangle Margin="440,400,440,480" d:LayoutOverrides="TopPosition, BottomPosition">
            <Rectangle.Fill>
                <ImageBrush Stretch="Uniform" ImageSource="Assets/VikingChessLogoMainPage.png"/>
            </Rectangle.Fill>
        </Rectangle>
    </Grid>
</Page>
```

3.2.2.2. Code-Behind

```
using System;
using System.Threading.Tasks;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.ViewManagement;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Media.Imaging;

using VikingChess.Model;
using Windows.UI;

namespace VikingChess
{
    /// <summary>
    /// The main page of the Viking Chess Application
    /// </summary>
    public sealed partial class MainPage : Page
    {
        Game game;

        /// <summary>
        /// The MainPage Constructor
        /// </summary>
        public MainPage()
        {
            this.InitializeComponent();
            this.NavigationCacheMode = Windows.UI.Xaml.Navigation.NavigationCacheMode.Enabled;
            game = new Game();
        }

        /// <summary>
        /// Processes the button click for the newGameBtn
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private async void newGameBtn_Click(object sender, RoutedEventArgs e)
        {
            game.PlaySound("btn_click.wav");
            await Task.Delay(TimeSpan.FromSeconds(1));
            this.Frame.Navigate(typeof(SettingsPage));
        }

        /// <summary>
        /// Processes the button click for the howToBtn
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private async void howToBtn_Click(object sender, RoutedEventArgs e)
        {
            game.PlaySound("btn_click.wav");
            await Task.Delay(TimeSpan.FromSeconds(1));
            this.Frame.Navigate(typeof(RulesPage));
        }

        /// <summary>
        /// Processes the button click for the howToBtn
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private async void aboutBtn_Click(object sender, RoutedEventArgs e)
        {
            game.PlaySound("btn_click.wav");
            await Task.Delay(TimeSpan.FromSeconds(1));
            this.Frame.Navigate(typeof(AboutPage));
        }

        /// <summary>
        /// Returns the user to the main page
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void back_btn_clicked(object sender, RoutedEventArgs e)
        {
            if (this.Frame.CanGoBack)
            {
                this.Frame.GoBack();
            }
        }

        /// <summary>
        /// Puts the application into fullscreen
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void fullscreen_btn_clicked(object sender, RoutedEventArgs e)
        {
            ApplicationView view = ApplicationView.GetForCurrentView();
            ImageBrush fullscreenImg = new ImageBrush();

            bool isInFullScreenMode = view.IsFullScreenMode;

            if (isInFullScreenMode)
            {
                fullscreenImg.ImageSource = new BitmapImage(new Uri(this.BaseUri, "/Assets/fullscreen-
arrows.png"));
                view.ExitFullScreenMode();
                this.fullscreen_btn.Background = fullscreenImg;
            }
            else
            {
                fullscreenImg.ImageSource = new BitmapImage(new Uri(this.BaseUri, "/Assets/windowed-
arrows.png"));
                view.TryEnterFullScreenMode();
                this.fullscreen_btn.Background = fullscreenImg;
            }
        }

    }
}
```

3.3. Settings Page

3.3.1. Description

The Settings Page allows for the configuration of game options before a new game begins.

3.3.2. Code

3.3.2.1. XAML

```
<Page
    x:Class="VikingChess.SettingsPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:VikingChess"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">
    <Page.Resources>
        <Style x:Key="ToggleSwitchStyle1" TargetType="ToggleSwitch">
            <Setter Property="Foreground" Value="{ThemeResource
SystemControlForegroundBaseHighBrush}"/>
            <Setter Property="HorizontalAlignment" Value="Left"/>
            <Setter Property="VerticalAlignment" Value="Center"/>
            <Setter Property="HorizontalContentAlignment" Value="Left"/>
            <Setter Property="FontFamily" Value="{ThemeResource ContentControlThemeFontFamily}"/>
            <Setter Property="FontSize" Value="{ThemeResource ControlContentThemeFontSize}"/>
            <Setter Property="MinWidth" Value="154"/>
            <Setter Property="ManipulationMode" Value="System,TranslateX"/>
            <Setter Property="UseSystemFocusVisuals" Value="True"/>
            <Setter Property="Template">
                <Setter.Value>
                    <ControlTemplate TargetType="ToggleSwitch">
                        <Grid BorderBrush="{TemplateBinding BorderBrush}"
BorderThickness="{TemplateBinding BorderThickness}" Margin="0,0,-23,0">
                            <Grid.ColumnDefinitions>
                                <ColumnDefinition Width="Auto"/>
                                <ColumnDefinition MaxWidth="12" Width="12"/>
                                <ColumnDefinition Width="Auto"/>
                                <ColumnDefinition Width="*"/>
                            </Grid.ColumnDefinitions>
                            <Grid.RowDefinitions>
                                <RowDefinition Height="Auto"/>
                                <RowDefinition Height="10"/>
                                <RowDefinition Height="Auto"/>
                                <RowDefinition Height="10"/>
                            </Grid.RowDefinitions>
                            <VisualStateManager.VisualStateGroups>
                                <VisualStateGroup x:Name="CommonStates">
                                    <VisualState x:Name="Normal"/>
                                    <VisualState x:Name="PointerOver">
                                        <Storyboard>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="StrokeThickness" Storyboard.TargetName="RightPosition">
```

```
                                                <DiscreteObjectKeyFrame KeyTime="0" Value="0"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Stroke" Storyboard.TargetName="RightPosition">
                                                <DiscreteObjectKeyFrame KeyTime="0" Value="#ffa200"/>
                                            </ObjectAnimationUsingKeyFrames>
                                        </Storyboard>
                                    </VisualState>
                                    <VisualState x:Name="Pressed">
                                        <Storyboard>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="StrokeThickness" Storyboard.TargetName="LeftPosition">
                                                <DiscreteObjectKeyFrame KeyTime="0" Value="0"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Fill" Storyboard.TargetName="RightPosition">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlHighlightBaseMediumBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Stroke" Storyboard.TargetName="RightPosition">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlHighlightBaseMediumBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                        </Storyboard>
                                    </VisualState>
                                    <VisualState x:Name="Disabled">
                                        <Storyboard>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Foreground" Storyboard.TargetName="OffContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlDisabledBaseLowBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Foreground" Storyboard.TargetName="OnContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlDisabledBaseLowBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Stroke" Storyboard.TargetName="LeftPosition">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlDisabledBaseLowBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Fill" Storyboard.TargetName="RightPosition">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlDisabledBaseLowBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Stroke" Storyboard.TargetName="RightPosition">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlDisabledBaseLowBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Fill" Storyboard.TargetName="SwitchKnobOn">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlPageBackgroundBaseLowBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                        </Storyboard>
                                    </VisualState>
                                </VisualStateGroup>
                                <VisualStateGroup x:Name="ToggleStates">
                                    <VisualStateGroup.Transitions>
                                        <VisualTransition x:Name="DraggingToOnTransition"
From="Dragging" GeneratedDuration="0" To="On">
                                            <Storyboard>
                                                <RepositionThemeAnimation
FromHorizontalOffset="{Binding TemplateSettings.KnobCurrentToOnOffset, RelativeSource={RelativeSource
Mode=TemplatedParent}}" TargetName="SwitchKnob"/>
                                                <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Opacity" Storyboard.TargetName="RightPosition">
                                                    <DiscreteObjectKeyFrame KeyTime="0" Value="1"/>
                                                </ObjectAnimationUsingKeyFrames>
                                                <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Opacity" Storyboard.TargetName="LeftPosition">
                                                    <DiscreteObjectKeyFrame KeyTime="0" Value="0"/>
                                                </ObjectAnimationUsingKeyFrames>
                                                <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Opacity" Storyboard.TargetName="SwitchKnobOn">
                                                    <DiscreteObjectKeyFrame KeyTime="0" Value="1"/>
                                                </ObjectAnimationUsingKeyFrames>
                                                <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Opacity" Storyboard.TargetName="SwitchKnobOff">
                                                    <DiscreteObjectKeyFrame KeyTime="0" Value="0"/>
                                                </ObjectAnimationUsingKeyFrames>
                                            </Storyboard>
                                        </VisualTransition>
                                        <VisualTransition x:Name="DraggingToOffTransition"
From="Dragging" GeneratedDuration="0" To="Off">
                                            <Storyboard>
                                                <RepositionThemeAnimation
FromHorizontalOffset="{Binding TemplateSettings.KnobCurrentToOffOffset, RelativeSource={RelativeSource
Mode=TemplatedParent}}" TargetName="SwitchKnob"/>
                                            </Storyboard>
                                        </VisualTransition>
                                        <VisualTransition x:Name="OnToOffTransition" From="On"
GeneratedDuration="0" To="Off">
                                            <Storyboard>
                                                <RepositionThemeAnimation
FromHorizontalOffset="{Binding TemplateSettings.KnobOnToOffOffset, RelativeSource={RelativeSource
Mode=TemplatedParent}}" TargetName="SwitchKnob"/>
                                            </Storyboard>
                                        </VisualTransition>
                                        <VisualTransition x:Name="OffToOnTransition" From="Off"
GeneratedDuration="0" To="On">
                                            <Storyboard>
                                                <RepositionThemeAnimation
FromHorizontalOffset="{Binding TemplateSettings.KnobOffToOnOffset, RelativeSource={RelativeSource
Mode=TemplatedParent}}" TargetName="SwitchKnob"/>
                                                <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Opacity" Storyboard.TargetName="RightPosition">
                                                    <DiscreteObjectKeyFrame KeyTime="0" Value="1"/>
                                                </ObjectAnimationUsingKeyFrames>
                                                <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Opacity" Storyboard.TargetName="LeftPosition">
                                                    <DiscreteObjectKeyFrame KeyTime="0" Value="0"/>
                                                </ObjectAnimationUsingKeyFrames>
                                                <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Opacity" Storyboard.TargetName="SwitchKnobOn">
                                                    <DiscreteObjectKeyFrame KeyTime="0" Value="1"/>
                                                </ObjectAnimationUsingKeyFrames>
                                                <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Opacity" Storyboard.TargetName="SwitchKnobOff">
                                                    <DiscreteObjectKeyFrame KeyTime="0" Value="0"/>
                                                </ObjectAnimationUsingKeyFrames>
                                            </Storyboard>
                                        </VisualTransition>
                                    </VisualStateGroup.Transitions>
                                    <VisualState x:Name="Dragging"/>
                                    <VisualState x:Name="Off"/>
                                    <VisualState x:Name="On">
                                        <Storyboard>
                                            <!-- Control the length that the switch travels -->
                                            <DoubleAnimation Duration="0" To="56"
Storyboard.TargetProperty="X" Storyboard.TargetName="KnobTranslateTransform"/>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Opacity" Storyboard.TargetName="RightPosition">
                                                <DiscreteObjectKeyFrame KeyTime="0" Value="1"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Opacity" Storyboard.TargetName="LeftPosition">
                                                <DiscreteObjectKeyFrame KeyTime="0" Value="0"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Opacity" Storyboard.TargetName="SwitchKnobOn">
                                                <DiscreteObjectKeyFrame KeyTime="0" Value="1"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Opacity" Storyboard.TargetName="SwitchKnobOff">
                                                <DiscreteObjectKeyFrame KeyTime="0" Value="0"/>
                                            </ObjectAnimationUsingKeyFrames>
                                        </Storyboard>
                                    </VisualState>
                                </VisualStateGroup>
                                <VisualStateGroup x:Name="ContentStates">
                                    <VisualState x:Name="OffContent">
                                        <Storyboard>
                                            <DoubleAnimation Duration="0" To="1"
Storyboard.TargetProperty="Opacity" Storyboard.TargetName="OffContentPresenter"/>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="IsHitTestVisible" Storyboard.TargetName="OffContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0">
                                                    <DiscreteObjectKeyFrame.Value>
                                                        <x:Boolean>True</x:Boolean>
                                                    </DiscreteObjectKeyFrame.Value>
                                                </DiscreteObjectKeyFrame>
                                            </ObjectAnimationUsingKeyFrames>
                                        </Storyboard>
                                    </VisualState>
                                    <VisualState x:Name="OnContent">
                                        <Storyboard>
                                            <DoubleAnimation Duration="0" To="1"
Storyboard.TargetProperty="Opacity" Storyboard.TargetName="OnContentPresenter"/>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="IsHitTestVisible" Storyboard.TargetName="OnContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0">
                                                    <DiscreteObjectKeyFrame.Value>
                                                        <x:Boolean>True</x:Boolean>
                                                    </DiscreteObjectKeyFrame.Value>
                                                </DiscreteObjectKeyFrame>
                                            </ObjectAnimationUsingKeyFrames>
                                        </Storyboard>
                                    </VisualState>
                                </VisualStateGroup>
                            </VisualStateManager.VisualStateGroups>
                            <ContentPresenter x:Name="OffContentPresenter"
AutomationProperties.AccessibilityView="Raw" ContentTemplate="{TemplateBinding OffContentTemplate}"
Grid.Column="2" Foreground="{TemplateBinding Foreground}" HorizontalAlignment="Center"
IsHitTestVisible="False" Opacity="0" Grid.Row="1" Grid.RowSpan="3" VerticalAlignment="{TemplateBinding
VerticalContentAlignment}"/>
                            <ContentPresenter x:Name="OnContentPresenter"
AutomationProperties.AccessibilityView="Raw" ContentTemplate="{TemplateBinding OnContentTemplate}"
Grid.Column="2" Foreground="{TemplateBinding Foreground}" HorizontalAlignment="Center"
IsHitTestVisible="False" Opacity="0" Grid.Row="1" Grid.RowSpan="3" VerticalAlignment="{TemplateBinding
VerticalContentAlignment}"/>
                            <Grid Control.IsTemplateFocusTarget="True" Margin="0,5" Grid.Row="1"
Grid.RowSpan="3"/>
                            <Rectangle x:Name="LeftPosition" RadiusX="5" RadiusY="5" Height="50"
Grid.Row="2" Stroke="#ffa200" StrokeThickness="0" Width="100">
                                <Rectangle.Fill>
                                    <ImageBrush Stretch="Uniform"
ImageSource="Assets/toggleSwitchBgRed.png"/>
                                </Rectangle.Fill>
                            </Rectangle>
                            <Rectangle x:Name="RightPosition" RadiusX="5" RadiusY="5" Height="50"
Opacity="0" Grid.Row="2" Stroke="#ffa200" StrokeThickness="0" Width="100">
                                <Rectangle.Fill>
                                    <ImageBrush Stretch="Uniform"
ImageSource="Assets/toggleSwitchBgGreen.png"/>
                                </Rectangle.Fill>
                            </Rectangle>
                            <Grid x:Name="SwitchKnob" HorizontalAlignment="Left" Height="50"
Grid.Row="2" Width="40" Margin="2,0,0,2">
                                <Grid.RenderTransform>
                                    <TranslateTransform x:Name="KnobTranslateTransform"/>
                                </Grid.RenderTransform>
                                <Rectangle x:Name="SwitchKnobOn" Height="40" Opacity="0" Width="40">
                                    <Rectangle.Fill>
                                        <ImageBrush Stretch="Uniform"
ImageSource="Assets/toggleSwitchBtn.png"/>
                                    </Rectangle.Fill>
                                </Rectangle>
                                <Rectangle x:Name="SwitchKnobOff" Height="40" Width="40">
                                    <Rectangle.Fill>
                                        <ImageBrush Stretch="Uniform"
ImageSource="Assets/toggleSwitchBtn.png"/>
                                    </Rectangle.Fill>
                                </Rectangle>
                            </Grid>
                            <Thumb x:Name="SwitchThumb" AutomationProperties.AccessibilityView="Raw"
Grid.Row="1" Grid.RowSpan="3" Margin="0,0,0,-10"  Background="{x:Null}" Foreground="White">
                                <Thumb.Template>
                                    <ControlTemplate TargetType="Thumb">
                                        <Rectangle Fill="Transparent"/>
                                    </ControlTemplate>
                                </Thumb.Template>
                            </Thumb>
                        </Grid>
                    </ControlTemplate>
                </Setter.Value>
            </Setter>
        </Style>
        <Style x:Key="ToggleSwitchStyle2" TargetType="ToggleSwitch">
            <Setter Property="Foreground" Value="{ThemeResource
SystemControlForegroundBaseHighBrush}"/>
            <Setter Property="HorizontalAlignment" Value="Left"/>
            <Setter Property="VerticalAlignment" Value="Center"/>
            <Setter Property="HorizontalContentAlignment" Value="Left"/>
            <Setter Property="FontFamily" Value="{ThemeResource ContentControlThemeFontFamily}"/>
            <Setter Property="FontSize" Value="{ThemeResource ControlContentThemeFontSize}"/>
            <Setter Property="MinWidth" Value="154"/>
            <Setter Property="ManipulationMode" Value="System,TranslateX"/>
            <Setter Property="UseSystemFocusVisuals" Value="True"/>
            <Setter Property="Template">
                <Setter.Value>
                    <ControlTemplate TargetType="ToggleSwitch">
                        <Grid BorderBrush="{TemplateBinding BorderBrush}"
BorderThickness="{TemplateBinding BorderThickness}" Margin="0,0,-23,0">
                            <Grid.ColumnDefinitions>
                                <ColumnDefinition Width="Auto"/>
                                <ColumnDefinition MaxWidth="12" Width="12"/>
                                <ColumnDefinition Width="Auto"/>
                                <ColumnDefinition Width="*"/>
                            </Grid.ColumnDefinitions>
                            <Grid.RowDefinitions>
                                <RowDefinition Height="Auto"/>
                                <RowDefinition Height="10"/>
                                <RowDefinition Height="Auto"/>
                                <RowDefinition Height="10"/>
                            </Grid.RowDefinitions>
                            <VisualStateManager.VisualStateGroups>
                                <VisualStateGroup x:Name="CommonStates">
                                    <VisualState x:Name="Normal"/>
                                    <VisualState x:Name="PointerOver">
                                        <Storyboard>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="StrokeThickness" Storyboard.TargetName="RightPosition">
                                                <DiscreteObjectKeyFrame KeyTime="0" Value="0"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Stroke" Storyboard.TargetName="RightPosition">
                                                <DiscreteObjectKeyFrame KeyTime="0" Value="#ffa200"/>
                                            </ObjectAnimationUsingKeyFrames>
                                        </Storyboard>
                                    </VisualState>
                                    <VisualState x:Name="Pressed">
                                        <Storyboard>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="StrokeThickness" Storyboard.TargetName="LeftPosition">
                                                <DiscreteObjectKeyFrame KeyTime="0" Value="0"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Fill" Storyboard.TargetName="RightPosition">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlHighlightBaseMediumBrush}"/>
```

```
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Stroke" Storyboard.TargetName="RightPosition">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlHighlightBaseMediumBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                        </Storyboard>
                                    </VisualState>
                                    <VisualState x:Name="Disabled">
                                        <Storyboard>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Foreground" Storyboard.TargetName="OffContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlDisabledBaseLowBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Foreground" Storyboard.TargetName="OnContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlDisabledBaseLowBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Stroke" Storyboard.TargetName="LeftPosition">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlDisabledBaseLowBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Fill" Storyboard.TargetName="RightPosition">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlDisabledBaseLowBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Stroke" Storyboard.TargetName="RightPosition">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlDisabledBaseLowBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Fill" Storyboard.TargetName="SwitchKnobOn">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlPageBackgroundBaseLowBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                        </Storyboard>
                                    </VisualState>
                                </VisualStateGroup>
                                <VisualStateGroup x:Name="ToggleStates">
                                    <VisualStateGroup.Transitions>
                                        <VisualTransition x:Name="DraggingToOnTransition"
From="Dragging" GeneratedDuration="0" To="On">
                                            <Storyboard>
                                                <RepositionThemeAnimation
FromHorizontalOffset="{Binding TemplateSettings.KnobCurrentToOnOffset, RelativeSource={RelativeSource
Mode=TemplatedParent}}" TargetName="SwitchKnob"/>
                                                <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Opacity" Storyboard.TargetName="RightPosition">
                                                    <DiscreteObjectKeyFrame KeyTime="0" Value="1"/>
                                                </ObjectAnimationUsingKeyFrames>
                                                <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Opacity" Storyboard.TargetName="LeftPosition">
                                                    <DiscreteObjectKeyFrame KeyTime="0" Value="0"/>
                                                </ObjectAnimationUsingKeyFrames>
                                                <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Opacity" Storyboard.TargetName="SwitchKnobOn">
                                                    <DiscreteObjectKeyFrame KeyTime="0" Value="1"/>
                                                </ObjectAnimationUsingKeyFrames>
                                                <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Opacity" Storyboard.TargetName="SwitchKnobOff">
                                                    <DiscreteObjectKeyFrame KeyTime="0" Value="0"/>
                                                </ObjectAnimationUsingKeyFrames>
                                            </Storyboard>
                                        </VisualTransition>
                                        <VisualTransition x:Name="DraggingToOffTransition"
From="Dragging" GeneratedDuration="0" To="Off">
                                            <Storyboard>
```

```
                                                <RepositionThemeAnimation
FromHorizontalOffset="{Binding TemplateSettings.KnobCurrentToOffOffset, RelativeSource={RelativeSource
Mode=TemplatedParent}}" TargetName="SwitchKnob"/>
                                            </Storyboard>
                                        </VisualTransition>
                                        <VisualTransition x:Name="OnToOffTransition" From="On"
GeneratedDuration="0" To="Off">
                                            <Storyboard>
                                                <RepositionThemeAnimation
FromHorizontalOffset="{Binding TemplateSettings.KnobOnToOffOffset, RelativeSource={RelativeSource
Mode=TemplatedParent}}" TargetName="SwitchKnob"/>
                                            </Storyboard>
                                        </VisualTransition>
                                        <VisualTransition x:Name="OffToOnTransition" From="Off"
GeneratedDuration="0" To="On">
                                            <Storyboard>
                                                <RepositionThemeAnimation
FromHorizontalOffset="{Binding TemplateSettings.KnobOffToOnOffset, RelativeSource={RelativeSource
Mode=TemplatedParent}}" TargetName="SwitchKnob"/>
                                                <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Opacity" Storyboard.TargetName="RightPosition">
                                                    <DiscreteObjectKeyFrame KeyTime="0" Value="1"/>
                                                </ObjectAnimationUsingKeyFrames>
                                                <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Opacity" Storyboard.TargetName="LeftPosition">
                                                    <DiscreteObjectKeyFrame KeyTime="0" Value="0"/>
                                                </ObjectAnimationUsingKeyFrames>
                                                <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Opacity" Storyboard.TargetName="SwitchKnobOn">
                                                    <DiscreteObjectKeyFrame KeyTime="0" Value="1"/>
                                                </ObjectAnimationUsingKeyFrames>
                                                <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Opacity" Storyboard.TargetName="SwitchKnobOff">
                                                    <DiscreteObjectKeyFrame KeyTime="0" Value="0"/>
                                                </ObjectAnimationUsingKeyFrames>
                                            </Storyboard>
                                        </VisualTransition>
                                    </VisualStateGroup.Transitions>
                                    <VisualState x:Name="Dragging"/>
                                    <VisualState x:Name="Off"/>
                                    <VisualState x:Name="On">
                                        <Storyboard>
                                            <!-- Control the length that the switch travels -->
                                            <DoubleAnimation Duration="0" To="56"
Storyboard.TargetProperty="X" Storyboard.TargetName="KnobTranslateTransform"/>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Opacity" Storyboard.TargetName="RightPosition">
                                                <DiscreteObjectKeyFrame KeyTime="0" Value="1"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Opacity" Storyboard.TargetName="LeftPosition">
                                                <DiscreteObjectKeyFrame KeyTime="0" Value="0"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Opacity" Storyboard.TargetName="SwitchKnobOn">
                                                <DiscreteObjectKeyFrame KeyTime="0" Value="1"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Opacity" Storyboard.TargetName="SwitchKnobOff">
                                                <DiscreteObjectKeyFrame KeyTime="0" Value="0"/>
                                            </ObjectAnimationUsingKeyFrames>
                                        </Storyboard>
                                    </VisualState>
                                </VisualStateGroup>
                                <VisualStateGroup x:Name="ContentStates">
                                    <VisualState x:Name="OffContent">
                                        <Storyboard>
                                            <DoubleAnimation Duration="0" To="1"
Storyboard.TargetProperty="Opacity" Storyboard.TargetName="OffContentPresenter"/>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="IsHitTestVisible" Storyboard.TargetName="OffContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0">
                                                    <DiscreteObjectKeyFrame.Value>
                                                        <x:Boolean>True</x:Boolean>
                                                    </DiscreteObjectKeyFrame.Value>
                                                </DiscreteObjectKeyFrame>
                                            </ObjectAnimationUsingKeyFrames>
                                        </Storyboard>
                                    </VisualState>
                                    <VisualState x:Name="OnContent">
                                        <Storyboard>
                                            <DoubleAnimation Duration="0" To="1" 
Storyboard.TargetProperty="Opacity" Storyboard.TargetName="OnContentPresenter"/>
                                            <ObjectAnimationUsingKeyFrames 
Storyboard.TargetProperty="IsHitTestVisible" Storyboard.TargetName="OnContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0">
                                                    <DiscreteObjectKeyFrame.Value>
                                                        <x:Boolean>True</x:Boolean>
                                                    </DiscreteObjectKeyFrame.Value>
                                                </DiscreteObjectKeyFrame>
                                            </ObjectAnimationUsingKeyFrames>
                                        </Storyboard>
                                    </VisualState>
                                </VisualStateGroup>
                            </VisualStateManager.VisualStateGroups>
                            <ContentPresenter x:Name="OffContentPresenter" 
AutomationProperties.AccessibilityView="Raw" ContentTemplate="{TemplateBinding OffContentTemplate}" 
Grid.Column="2" Foreground="{TemplateBinding Foreground}" HorizontalAlignment="Center" 
IsHitTestVisible="False" Opacity="0" Grid.Row="1" Grid.RowSpan="3" VerticalAlignment="{TemplateBinding 
VerticalContentAlignment}"/>
                            <ContentPresenter x:Name="OnContentPresenter" 
AutomationProperties.AccessibilityView="Raw" ContentTemplate="{TemplateBinding OnContentTemplate}" 
Grid.Column="2" Foreground="{TemplateBinding Foreground}" HorizontalAlignment="Center" 
IsHitTestVisible="False" Opacity="0" Grid.Row="1" Grid.RowSpan="3" VerticalAlignment="{TemplateBinding 
VerticalContentAlignment}"/>
                            <Grid Control.IsTemplateFocusTarget="True" Margin="0,5" Grid.Row="1" 
Grid.RowSpan="3"/>
                            <Rectangle x:Name="LeftPosition" RadiusX="5" RadiusY="5" Height="50" 
Grid.Row="2" Stroke="#ffa200" StrokeThickness="0" Width="100">
                                <Rectangle.Fill>
                                    <ImageBrush Stretch="Uniform" 
ImageSource="Assets/toggleSwitchBg2Yellow.png"/>
                                </Rectangle.Fill>
                            </Rectangle>
                            <Rectangle x:Name="RightPosition" RadiusX="5" RadiusY="5" Height="50" 
Opacity="0" Grid.Row="2" Stroke="#ffa200" StrokeThickness="0" Width="100">
                                <Rectangle.Fill>
                                    <ImageBrush Stretch="Uniform" 
ImageSource="Assets/toggleSwitchBg2Blue.png"/>
                                </Rectangle.Fill>
                            </Rectangle>
                            <Grid x:Name="SwitchKnob" HorizontalAlignment="Left" Height="50" 
Grid.Row="2" Width="40" Margin="2,0,0,2">
                                <Grid.RenderTransform>
                                    <TranslateTransform x:Name="KnobTranslateTransform"/>
                                </Grid.RenderTransform>
                                <Rectangle x:Name="SwitchKnobOn" Height="40" Opacity="0" Width="40">
                                    <Rectangle.Fill>
                                        <ImageBrush Stretch="Uniform" 
ImageSource="Assets/toggleSwitchBtn.png"/>
                                    </Rectangle.Fill>
                                </Rectangle>
                                <Rectangle x:Name="SwitchKnobOff" Height="40" Width="40">
                                    <Rectangle.Fill>
                                        <ImageBrush Stretch="Uniform" 
ImageSource="Assets/toggleSwitchBtn.png"/>
                                    </Rectangle.Fill>
                                </Rectangle>
                            </Grid>
                            <Thumb x:Name="SwitchThumb" AutomationProperties.AccessibilityView="Raw" 
Grid.Row="1" Grid.RowSpan="3" Margin="0,0,0,-10"  Background="{x:Null}" Foreground="White">
                                <Thumb.Template>
                                    <ControlTemplate TargetType="Thumb">
                                        <Rectangle Fill="Transparent"/>
                                    </ControlTemplate>
                                </Thumb.Template>
                            </Thumb>
                        </Grid>
```

```
                    </ControlTemplate>
                </Setter.Value>
            </Setter>
        </Style>
        <Style x:Key="ButtonStyle1" TargetType="Button">
            <Setter Property="Background" Value="{ThemeResource
SystemControlBackgroundBaseLowBrush}"/>
            <Setter Property="Foreground" Value="{ThemeResource
SystemControlForegroundBaseHighBrush}"/>
            <Setter Property="BorderBrush" Value="{ThemeResource
SystemControlForegroundTransparentBrush}"/>
            <Setter Property="BorderThickness" Value="{ThemeResource ButtonBorderThemeThickness}"/>
            <Setter Property="Padding" Value="8,4,8,4"/>
            <Setter Property="HorizontalAlignment" Value="Left"/>
            <Setter Property="VerticalAlignment" Value="Center"/>
            <Setter Property="FontFamily" Value="{ThemeResource ContentControlThemeFontFamily}"/>
            <Setter Property="FontWeight" Value="Normal"/>
            <Setter Property="FontSize" Value="{ThemeResource ControlContentThemeFontSize}"/>
            <Setter Property="UseSystemFocusVisuals" Value="True"/>
            <Setter Property="Template">
                <Setter.Value>
                    <ControlTemplate TargetType="Button">
                        <Grid x:Name="RootGrid" Background="{TemplateBinding Background}">
                            <VisualStateManager.VisualStateGroups>
                                <VisualStateGroup x:Name="CommonStates">
                                    <VisualState x:Name="Normal">
                                        <Storyboard>
                                            <PointerUpThemeAnimation
Storyboard.TargetName="RootGrid"/>
                                        </Storyboard>
                                    </VisualState>
                                    <VisualState x:Name="PointerOver">
                                        <Storyboard>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="BorderBrush" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0" Value="#e89300"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Foreground" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlHighlightBaseHighBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <PointerUpThemeAnimation
Storyboard.TargetName="RootGrid"/>
                                        </Storyboard>
                                    </VisualState>
                                    <VisualState x:Name="Pressed">
                                        <Storyboard>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Foreground" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlHighlightBaseHighBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <PointerDownThemeAnimation
Storyboard.TargetName="RootGrid"/>
                                        </Storyboard>
                                    </VisualState>
                                    <VisualState x:Name="Disabled">
                                        <Storyboard>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Background" Storyboard.TargetName="RootGrid">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlBackgroundBaseLowBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Foreground" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlDisabledBaseLowBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="BorderBrush" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlDisabledTransparentBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                        </Storyboard>
                                    </VisualState>
                                </VisualStateGroup>
                            </VisualStateManager.VisualStateGroups>
                            <ContentPresenter x:Name="ContentPresenter"
AutomationProperties.AccessibilityView="Raw" BorderBrush="{TemplateBinding BorderBrush}"
BorderThickness="{TemplateBinding BorderThickness}" ContentTemplate="{TemplateBinding
ContentTemplate}" ContentTransitions="{TemplateBinding ContentTransitions}" Content="{TemplateBinding
Content}" HorizontalContentAlignment="{TemplateBinding HorizontalContentAlignment}"
Padding="{TemplateBinding Padding}" VerticalContentAlignment="{TemplateBinding
VerticalContentAlignment}"/>
                        </Grid>
                    </ControlTemplate>
                </Setter.Value>
            </Setter>
        </Style>
        <Style x:Key="topbarButtonStyle" TargetType="Button">
            <Setter Property="Background" Value="{ThemeResource
SystemControlBackgroundBaseLowBrush}"/>
            <Setter Property="Foreground" Value="{ThemeResource
SystemControlForegroundBaseHighBrush}"/>
            <Setter Property="BorderBrush" Value="{ThemeResource
SystemControlForegroundTransparentBrush}"/>
            <Setter Property="BorderThickness" Value="{ThemeResource ButtonBorderThemeThickness}"/>
            <Setter Property="Padding" Value="8,4,8,4"/>
            <Setter Property="HorizontalAlignment" Value="Left"/>
            <Setter Property="VerticalAlignment" Value="Center"/>
            <Setter Property="FontFamily" Value="{ThemeResource ContentControlThemeFontFamily}"/>
            <Setter Property="FontWeight" Value="Normal"/>
            <Setter Property="FontSize" Value="{ThemeResource ControlContentThemeFontSize}"/>
            <Setter Property="UseSystemFocusVisuals" Value="True"/>
            <Setter Property="Template">
                <Setter.Value>
                    <ControlTemplate TargetType="Button">
                        <Grid x:Name="RootGrid" Background="{TemplateBinding Background}">
                            <VisualStateManager.VisualStateGroups>
                                <VisualStateGroup x:Name="CommonStates">
                                    <VisualState x:Name="Normal">
                                        <Storyboard>
                                            <PointerUpThemeAnimation
Storyboard.TargetName="RootGrid"/>
                                        </Storyboard>
                                    </VisualState>
                                    <VisualState x:Name="PointerOver">
                                        <Storyboard>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Foreground" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlHighlightBaseHighBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <PointerUpThemeAnimation
Storyboard.TargetName="RootGrid"/>
                                        </Storyboard>
                                    </VisualState>
                                    <VisualState x:Name="Pressed">
                                        <Storyboard>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Foreground" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlHighlightBaseHighBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <PointerDownThemeAnimation
Storyboard.TargetName="RootGrid"/>
                                        </Storyboard>
                                    </VisualState>
                                    <VisualState x:Name="Disabled">
                                        <Storyboard>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Background" Storyboard.TargetName="RootGrid">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlBackgroundBaseLowBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Foreground" Storyboard.TargetName="ContentPresenter">
```

```
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlDisabledBaseLowBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="BorderBrush" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlDisabledTransparentBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                        </Storyboard>
                                    </VisualState>
                                </VisualStateGroup>
                            </VisualStateManager.VisualStateGroups>
                            <ContentPresenter x:Name="ContentPresenter"
AutomationProperties.AccessibilityView="Raw" BorderBrush="{TemplateBinding BorderBrush}"
BorderThickness="{TemplateBinding BorderThickness}" ContentTemplate="{TemplateBinding
ContentTemplate}" ContentTransitions="{TemplateBinding ContentTransitions}" Content="{TemplateBinding
Content}" HorizontalContentAlignment="{TemplateBinding HorizontalContentAlignment}"
Padding="{TemplateBinding Padding}" VerticalContentAlignment="{TemplateBinding
VerticalContentAlignment}"/>
                        </Grid>
                    </ControlTemplate>
                </Setter.Value>
            </Setter>
        </Style>
    </Page.Resources>

    <Grid>
        <Grid.Background>
            <ImageBrush Stretch="UniformToFill" ImageSource="Assets/bg-wood.png"/>
        </Grid.Background>
        <RelativePanel>
            <ToggleSwitch x:Name="vsToggleSwitch" Header="ToggleSwitch" Height="61" Margin="319,157,-
3,-5" VerticalAlignment="Top" Width="147" Foreground="White" Background="#FFAC0303"
Style="{StaticResource ToggleSwitchStyle1}" RenderTransformOrigin="0.5,0.5">
                <ToggleSwitch.RenderTransform>
                    <CompositeTransform ScaleX="2.5" ScaleY="2.5"/>
                </ToggleSwitch.RenderTransform>
            </ToggleSwitch>
            <TextBlock x:Name="vsSwitchLabel" HorizontalAlignment="Left" Height="20" Margin="256,120,-
23,-20" TextWrapping="Wrap" VerticalAlignment="Top" Width="167" Foreground="#FFFFA200"
FontFamily="Assets/Fonts/VIKING-N.TTF#Viking-Normal" FontSize="13.333" TextAlignment="Center"
Text="Select Opponent" RenderTransformOrigin="0.5,0.5">
                <TextBlock.RenderTransform>
                    <CompositeTransform ScaleX="2" ScaleY="2"/>
                </TextBlock.RenderTransform>
            </TextBlock>
            <TextBlock x:Name="vsHumanLabel" HorizontalAlignment="Left" Height="20"
Margin="102,187,0,-12" TextWrapping="Wrap" VerticalAlignment="Top" Width="73" Foreground="#FFFFA200"
FontFamily="Assets/Fonts/VIKING-N.TTF#Viking-Normal" FontSize="13.333" TextAlignment="Center"
Text="Human" RenderTransformOrigin="0.5,0.5">
                <TextBlock.RenderTransform>
                    <CompositeTransform ScaleX="2" ScaleY="2"/>
                </TextBlock.RenderTransform>
            </TextBlock>
            <TextBlock x:Name="vsCpuLabel" HorizontalAlignment="Left" Height="20" Margin="468,191,-
30,-15" TextWrapping="Wrap" VerticalAlignment="Top" Width="40" Foreground="#FFFFA200"
FontFamily="Assets/Fonts/VIKING-N.TTF#Viking-Normal" FontSize="13.333" TextAlignment="Center"
Text="CPU" RenderTransformOrigin="0.5,0.5">
                <TextBlock.RenderTransform>
                    <CompositeTransform ScaleX="2" ScaleY="2"/>
                </TextBlock.RenderTransform>
            </TextBlock>
            <ToggleSwitch x:Name="colorSwitch" Header="ToggleSwitch" Height="62" Margin="321,319,-4,-
4" VerticalAlignment="Top" Width="160" Foreground="White" Background="#FFAC0303"
Style="{StaticResource ToggleSwitchStyle2}" RenderTransformOrigin="0.5,0.5">
                <ToggleSwitch.RenderTransform>
                    <CompositeTransform ScaleX="2.4" ScaleY="2.4"/>
                </ToggleSwitch.RenderTransform>
            </ToggleSwitch>
            <TextBlock x:Name="colorSwitchLabel" HorizontalAlignment="Left" Height="20"
Margin="276,291,0,-1" TextWrapping="Wrap" VerticalAlignment="Top" Width="112" Foreground="#FFFFA200"
FontFamily="Assets/Fonts/VIKING-N.TTF#Viking-Normal" FontSize="13.333" TextAlignment="Center"
Text="Choose Side" RenderTransformOrigin="0.5,0.5">
                <TextBlock.RenderTransform>
```

```
                    <CompositeTransform ScaleX="2" ScaleY="2"/>
                </TextBlock.RenderTransform>
            </TextBlock>
            <TextBlock x:Name="colorWhite" HorizontalAlignment="Left" Height="20" Margin="61,357,0,-2"
TextWrapping="Wrap" VerticalAlignment="Top" Width="103" Foreground="#FFFFA200"
FontFamily="Assets/Fonts/VIKING-N.TTF#Viking-Normal" FontSize="13.333" TextAlignment="Center"
Text="Attacker" RenderTransformOrigin="0.5,0.5">
                <TextBlock.RenderTransform>
                    <CompositeTransform ScaleX="2" ScaleY="2"/>
                </TextBlock.RenderTransform>
            </TextBlock>
            <TextBlock x:Name="colorBlack" HorizontalAlignment="Left" Height="20" Margin="496,356,-
16,-1" TextWrapping="Wrap" VerticalAlignment="Top" Width="97" Foreground="#FFFFA200"
FontFamily="Assets/Fonts/VIKING-N.TTF#Viking-Normal" FontSize="13.333" TextAlignment="Center"
Text="Defender" RenderTransformOrigin="0.5,0.5">
                <TextBlock.RenderTransform>
                    <CompositeTransform ScaleX="2" ScaleY="2"/>
                </TextBlock.RenderTransform>
            </TextBlock>
            <TextBlock x:Name="p1NameLabel" HorizontalAlignment="Left" Height="20" Margin="261,472,-
6,-5" TextWrapping="Wrap" VerticalAlignment="Top" Width="145" Foreground="#FFFFA200"
FontFamily="Assets/Fonts/VIKING-N.TTF#Viking-Normal" FontSize="13.333" TextAlignment="Center"
Text="Player 1 Name" RenderTransformOrigin="0.5,0.5">
                <TextBlock.RenderTransform>
                    <CompositeTransform ScaleX="2" ScaleY="2"/>
                </TextBlock.RenderTransform>
            </TextBlock>
            <TextBox x:Name="p1NameTextBox" HorizontalAlignment="Left" Height="20" Margin="232,504,-
1,-4" TextWrapping="Wrap" Text="" Background="#ffffa200" VerticalAlignment="Top" Width="200"
TextAlignment="Center" FontSize="16" FontFamily="Assets/Fonts/VIKING-N.TTF#Viking-Normal"/>
            <TextBlock x:Name="p2NameLabel" HorizontalAlignment="Left" Height="18" Margin="259,0,-8,-
609" TextWrapping="Wrap" VerticalAlignment="Bottom" Width="149" Foreground="#FFFFA200"
FontFamily="Assets/Fonts/VIKING-N.TTF#Viking-Normal" FontSize="13.333" TextAlignment="Center"
Text="Player 2 Name" RenderTransformOrigin="0.5,0.5">
                <TextBlock.RenderTransform>
                    <CompositeTransform ScaleX="2" ScaleY="2"/>
                </TextBlock.RenderTransform>
            </TextBlock>
            <TextBox x:Name="p2NameTextBox" HorizontalAlignment="Left" Height="20" Margin="232,0,0,-
658" TextWrapping="Wrap" Text="" Background="#ffffa200" VerticalAlignment="Bottom" Width="200"
TextAlignment="Center" FontSize="16" FontFamily="Assets/Fonts/VIKING-N.TTF#Viking-Normal"/>
            <TextBlock x:Name="BoardSelectionLabel" HorizontalAlignment="Left" Height="20"
Margin="1335,190,-1082,-90" TextWrapping="Wrap" VerticalAlignment="Top" Width="150"
Foreground="#FFFFA200" FontFamily="Assets/Fonts/VIKING-N.TTF#Viking-Normal" FontSize="13.333"
TextAlignment="Center" RenderTransformOrigin="0.5,0.5">
                <TextBlock.RenderTransform>
                    <CompositeTransform ScaleX="2" ScaleY="2"/>
                </TextBlock.RenderTransform>
                <Run Text="Select Game"/>
                <Run Text="        "/>
            </TextBlock>
            <Button x:Name="BrandubhBtn" Style="{StaticResource ButtonStyle1}"
Click="BrandubhBtn_Clicked" HorizontalAlignment="Left" Height="300" VerticalAlignment="Top"
Width="300" Margin="1600,540,-1833,-811" FontSize="26.667" FontFamily="Assets/Fonts/VIKING-
N.TTF#Viking-Normal" Foreground="White">
                <Button.Background>
                    <ImageBrush Stretch="Fill" ImageSource="Assets/BrandubhLayout.png"/>
                </Button.Background>
            </Button>
            <Button x:Name="HnefataflBtn" Style="{StaticResource ButtonStyle1}"
Click="HnefataflBtn_Clicked" HorizontalAlignment="Left" Height="300" VerticalAlignment="Top"
Width="300" Margin="1600,180,-1833,-451" Foreground="{x:Null}">
                <Button.Background>
                    <ImageBrush Stretch="Fill" ImageSource="Assets/HnefataflLayout.png"/>
                </Button.Background>
            </Button>
            <TextBlock x:Name="BrandubhLabel" HorizontalAlignment="Left" Height="20"
Margin="1675,855,-1425,-755" TextWrapping="Wrap" VerticalAlignment="Top" Width="150"
Foreground="#FFFFA200" FontFamily="Assets/Fonts/VIKING-N.TTF#Viking-Normal" FontSize="12"
TextAlignment="Center" Text="Brandubh (7x7)" RenderTransformOrigin="0.5,0.5">
                <TextBlock.RenderTransform>
                    <CompositeTransform ScaleX="2" ScaleY="2"/>
                </TextBlock.RenderTransform>
            </TextBlock>
            <TextBlock x:Name="HnefataflLabel" HorizontalAlignment="Left" Height="20"
Margin="1675,495,-1425,-395" TextWrapping="Wrap" VerticalAlignment="Top" Width="150"
Foreground="#FFFFA200" FontFamily="Assets/Fonts/VIKING-N.TTF#Viking-Normal" FontSize="12"
TextAlignment="Center" Text="Hnefatafl (11x11)" RenderTransformOrigin="0.5,0.5">
                <TextBlock.RenderTransform>
                    <CompositeTransform ScaleX="2" ScaleY="2"/>
                </TextBlock.RenderTransform>
            </TextBlock>
            <Button x:Name="ArdRiBtn" Style="{StaticResource ButtonStyle1}" Click="ArdRiBtn_Clicked"
HorizontalAlignment="Left" Height="300" VerticalAlignment="Top" Width="300" Margin="920,540,-1153,-
811" FontSize="26.667" FontFamily="Assets/Fonts/VIKING-N.TTF#Viking-Normal" Foreground="White">
                <Button.Background>
                    <ImageBrush Stretch="Fill" ImageSource="Assets/ArdRiLayout.png"/>
                </Button.Background>
            </Button>
            <TextBlock x:Name="ArdRiLabel" HorizontalAlignment="Left" Height="20" Margin="997,855,-
749,-755" TextWrapping="Wrap" VerticalAlignment="Top" Width="152" Foreground="#FFFFA200"
FontFamily="Assets/Fonts/VIKING-N.TTF#Viking-Normal" FontSize="12" TextAlignment="Center" Text="Ard Rí
(7x7)" RenderTransformOrigin="0.5,0.5">
                <TextBlock.RenderTransform>
                    <CompositeTransform ScaleX="2" ScaleY="2"/>
                </TextBlock.RenderTransform>
            </TextBlock>
            <Button x:Name="TablutBtn" Style="{StaticResource ButtonStyle1}" Click="TablutBtn_Clicked"
HorizontalAlignment="Left" Height="300" VerticalAlignment="Top" Width="300" Margin="1261,360,-1494,-
631" FontSize="26.667" FontFamily="Assets/Fonts/VIKING-N.TTF#Viking-Normal" Foreground="White">
                <Button.Background>
                    <ImageBrush Stretch="Fill" ImageSource="Assets/TablutLayout.png"/>
                </Button.Background>
            </Button>
            <TextBlock x:Name="TablutLabel" HorizontalAlignment="Left" Height="20" Margin="1336,675,-
1086,-575" TextWrapping="Wrap" VerticalAlignment="Top" Width="150" Foreground="#FFFFA200"
FontFamily="Assets/Fonts/VIKING-N.TTF#Viking-Normal" FontSize="13.333" TextAlignment="Center"
Text="Tablut (9x9)" RenderTransformOrigin="0.578,0.575">
                <TextBlock.RenderTransform>
                    <CompositeTransform ScaleX="2" ScaleY="2" TranslateX="11.544" TranslateY="1.5"/>
                </TextBlock.RenderTransform>
            </TextBlock>
            <Button x:Name="TawlbwrddBtn" Style="{StaticResource ButtonStyle1}"
Click="TawlbwrddBtn_Clicked" HorizontalAlignment="Left" Height="300" VerticalAlignment="Top"
Width="300" Margin="920,180,-1153,-451" FontSize="26.667" FontFamily="Assets/Fonts/VIKING-
N.TTF#Viking-Normal" Foreground="White">
                <Button.Background>
                    <ImageBrush Stretch="Fill" ImageSource="Assets/TawlbwrddLayout.png"/>
                </Button.Background>
            </Button>
            <TextBlock x:Name="TawlbwrddLabel" HorizontalAlignment="Left" Height="20"
Margin="995,495,-745,-395" TextWrapping="Wrap" VerticalAlignment="Top" Width="150"
Foreground="#FFFFA200" FontFamily="Assets/Fonts/VIKING-N.TTF#Viking-Normal" FontSize="10.667"
TextAlignment="Center" Text="Tawlbwrdd (11x11)" RenderTransformOrigin="0.5,0.5">
                <TextBlock.RenderTransform>
                    <CompositeTransform ScaleX="2" ScaleY="2"/>
                </TextBlock.RenderTransform>
            </TextBlock>
        </RelativePanel>
        <Rectangle Height="20" StrokeThickness="10" VerticalAlignment="Top" Fill="#FF6B2509"
Width="1920" />
        <Button x:Name="fullscreen_btn" Style="{StaticResource topbarButtonStyle}" Content=""
Click="fullscreen_btn_clicked" HorizontalAlignment="Right" Height="18" VerticalAlignment="Top"
Width="18" Margin="0,1,10,0">
            <Button.Background>
                <ImageBrush ImageSource="Assets/fullscreen-arrows.png" Stretch="Uniform"/>
            </Button.Background>
        </Button>
        <Button x:Name="back_btn" Style="{StaticResource topbarButtonStyle}" Content=""
Click="back_btn_clicked"  HorizontalAlignment="Left" Height="20" VerticalAlignment="Top" Width="20">
            <Button.Background>
                <ImageBrush ImageSource="Assets/back-arrow.png" Stretch="Uniform"/>
            </Button.Background>
        </Button>
    </Grid>
</Page>
```

3.3.2.2. Code-Behind

```
using System;
using VikingChess.Model;
using Windows.UI.ViewManagement;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Media.Imaging;

namespace VikingChess
{
    /// <summary>
    /// The SettingsPage allows for the setting of game parameters such as playing locally against a
human or against the computer, setting player names and choosing the game type.
    /// </summary>
    public sealed partial class SettingsPage : Page
    {
        Game game = new Game();

        /// <summary>
        /// SettingsPage Constructor
        /// </summary>
        public SettingsPage()
        {
            this.InitializeComponent();
            this.NavigationCacheMode = Windows.UI.Xaml.Navigation.NavigationCacheMode.Enabled;
        }

        /// <summary>
        /// Processes the button click for the backBtn
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void back_btn_clicked(object sender, RoutedEventArgs e)
        {
            if (this.Frame.CanGoBack)
            {
                this.Frame.GoBack();
            }
        }

        /// <summary>
        /// Processes the button click for the fullscreenBtn
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void fullscreen_btn_clicked(object sender, RoutedEventArgs e)
        {
            ApplicationView view = ApplicationView.GetForCurrentView();
            ImageBrush fullscreenImg = new ImageBrush();

            bool isInFullScreenMode = view.IsFullScreenMode;

            if (isInFullScreenMode)
            {
                fullscreenImg.ImageSource = new BitmapImage(new Uri(this.BaseUri, "/Assets/fullscreen-
arrows.png"));
                view.ExitFullScreenMode();
                this.fullscreen_btn.Background = fullscreenImg;
            }
            else
            {
                fullscreenImg.ImageSource = new BitmapImage(new Uri(this.BaseUri, "/Assets/windowed-
arrows.png"));
                view.TryEnterFullScreenMode();
                this.fullscreen_btn.Background = fullscreenImg;
            }
        }

        /// <summary>
        /// Gets the parameters for the <see cref="Game"/>
        /// </summary>
        /// <returns><see cref="Tuple{T1, T2, T3, T4, T5}"/></returns>
        private Tuple<string, string, string, string> getParameters()
        {
            string versusChoice;
            string sideChoice;
            string p1Name;
            string p2Name;

            if (vsToggleSwitch.IsOn)
            {
                versusChoice = "CPU";
            }
            else
            {
                versusChoice = "HUMAN";
            }
            if (colorSwitch.IsOn)
            {
                sideChoice = "DEFENDER";
            }
            else
            {
                sideChoice = "ATTACKER";
            }

            p1Name = p1NameTextBox.Text;
            p2Name = p2NameTextBox.Text;

            return new Tuple<string, string, string, string>(versusChoice, sideChoice, p1Name,
p2Name);
        }

        /// <summary>
        /// Starts a game of Hnefatafl
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void HnefataflBtn_Clicked(object sender, RoutedEventArgs e)
        {
            game.PlaySound("btn_click.wav");

            Tuple<string, string, string, string> parameters = getParameters();

            Tuple<string, string, string, string, string> gameSettings = new Tuple<string, string,
string, string, string>(parameters.Item1, parameters.Item2, parameters.Item3, parameters.Item4,
"Hnefatafl");

            Frame.Navigate(typeof(HnefataflPage), gameSettings);
        }

        /// <summary>
        /// Starts a game of Brandubh
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void BrandubhBtn_Clicked(object sender, RoutedEventArgs e)
        {
            game.PlaySound("btn_click.wav");

            Tuple<string, string, string, string> parameters = getParameters();

            Tuple<string, string, string, string, string> gameSettings = new Tuple<string, string,
string, string, string>(parameters.Item1, parameters.Item2, parameters.Item3, parameters.Item4,
"Brandubh");

            Frame.Navigate(typeof(BrandubhPage), gameSettings);
        }

        /// <summary>
        /// Starts a game of Ard Rí
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void ArdRiBtn_Clicked(object sender, RoutedEventArgs e)
        {
            game.PlaySound("btn_click.wav");

            Tuple<string, string, string, string> parameters = getParameters();

            Tuple<string, string, string, string, string> gameSettings = new Tuple<string, string,
string, string, string>(parameters.Item1, parameters.Item2, parameters.Item3, parameters.Item4, "Ard
Ri");

            Frame.Navigate(typeof(ArdRiPage), gameSettings);
        }

        /// <summary>
        /// Starts a game of Tablut
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void TablutBtn_Clicked(object sender, RoutedEventArgs e)
        {
            game.PlaySound("btn_click.wav");

            Tuple<string, string, string, string> parameters = getParameters();

            Tuple<string, string, string, string, string> gameSettings = new Tuple<string, string,
string, string, string>(parameters.Item1, parameters.Item2, parameters.Item3, parameters.Item4,
"Tablut");

            Frame.Navigate(typeof(TablutPage), gameSettings);
        }

        /// <summary>
        /// Starts a game of Tawlbwrdd
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void TawlbwrddBtn_Clicked(object sender, RoutedEventArgs e)
        {
            game.PlaySound("btn_click.wav");

            Tuple<string, string, string, string> parameters = getParameters();

            Tuple<string, string, string, string, string> gameSettings = new Tuple<string, string,
string, string, string>(parameters.Item1, parameters.Item2, parameters.Item3, parameters.Item4,
"Tawlbwrdd");

            Frame.Navigate(typeof(TawlbwrddPage), gameSettings);
        }
    }
}
```

3.4. Rules Page

3.4.1. Description

The Rules Page displays the rules of the game to the user.

3.4.2. Code

3.4.2.1. XAML

```
<Page
    x:Class="VikingChess.RulesPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:VikingChess"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:gif="using:XamlAnimatedGif"
    mc:Ignorable="d">
    <Page.Resources>
        <Style x:Key="topbarButtonStyle" TargetType="Button">
            <Setter Property="Background" Value="{ThemeResource
SystemControlBackgroundBaseLowBrush}"/>
```

```
            <Setter Property="Foreground" Value="{ThemeResource
SystemControlForegroundBaseHighBrush}"/>
            <Setter Property="BorderBrush" Value="{ThemeResource
SystemControlForegroundTransparentBrush}"/>
            <Setter Property="BorderThickness" Value="{ThemeResource ButtonBorderThemeThickness}"/>
            <Setter Property="Padding" Value="8,4,8,4"/>
            <Setter Property="HorizontalAlignment" Value="Left"/>
            <Setter Property="VerticalAlignment" Value="Center"/>
            <Setter Property="FontFamily" Value="{ThemeResource ContentControlThemeFontFamily}"/>
            <Setter Property="FontWeight" Value="Normal"/>
            <Setter Property="FontSize" Value="{ThemeResource ControlContentThemeFontSize}"/>
            <Setter Property="UseSystemFocusVisuals" Value="True"/>
            <Setter Property="Template">
                <Setter.Value>
                    <ControlTemplate TargetType="Button">
                        <Grid x:Name="RootGrid" Background="{TemplateBinding Background}">
                            <VisualStateManager.VisualStateGroups>
                                <VisualStateGroup x:Name="CommonStates">
                                    <VisualState x:Name="Normal">
                                        <Storyboard>
                                            <PointerUpThemeAnimation
Storyboard.TargetName="RootGrid"/>
                                        </Storyboard>
                                    </VisualState>
                                    <VisualState x:Name="PointerOver">
                                        <Storyboard>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Foreground" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlHighlightBaseHighBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <PointerUpThemeAnimation
Storyboard.TargetName="RootGrid"/>
                                        </Storyboard>
                                    </VisualState>
                                    <VisualState x:Name="Pressed">
                                        <Storyboard>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Foreground" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlHighlightBaseHighBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <PointerDownThemeAnimation
Storyboard.TargetName="RootGrid"/>
                                        </Storyboard>
                                    </VisualState>
                                    <VisualState x:Name="Disabled">
                                        <Storyboard>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Background" Storyboard.TargetName="RootGrid">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlBackgroundBaseLowBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Foreground" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlDisabledBaseLowBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="BorderBrush" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlDisabledTransparentBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                        </Storyboard>
                                    </VisualState>
                                </VisualStateGroup>
                            </VisualStateManager.VisualStateGroups>
                            <ContentPresenter x:Name="ContentPresenter"
AutomationProperties.AccessibilityView="Raw" BorderBrush="{TemplateBinding BorderBrush}"
BorderThickness="{TemplateBinding BorderThickness}" ContentTemplate="{TemplateBinding
ContentTemplate}" ContentTransitions="{TemplateBinding ContentTransitions}" Content="{TemplateBinding
Content}" HorizontalContentAlignment="{TemplateBinding HorizontalContentAlignment}"
Padding="{TemplateBinding Padding}" VerticalContentAlignment="{TemplateBinding
VerticalContentAlignment}"/>
```

```
                        </Grid>
                    </ControlTemplate>
                </Setter.Value>
            </Setter>
        </Style>
        <Style x:Key="ButtonStyle1" TargetType="Button">
            <Setter Property="Background" Value="{ThemeResource
SystemControlBackgroundBaseLowBrush}"/>
            <Setter Property="Foreground" Value="{ThemeResource
SystemControlForegroundBaseHighBrush}"/>
            <Setter Property="BorderBrush" Value="{ThemeResource
SystemControlForegroundTransparentBrush}"/>
            <Setter Property="BorderThickness" Value="{ThemeResource ButtonBorderThemeThickness}"/>
            <Setter Property="Padding" Value="8,4,8,4"/>
            <Setter Property="HorizontalAlignment" Value="Left"/>
            <Setter Property="VerticalAlignment" Value="Center"/>
            <Setter Property="FontFamily" Value="{ThemeResource ContentControlThemeFontFamily}"/>
            <Setter Property="FontWeight" Value="Normal"/>
            <Setter Property="FontSize" Value="{ThemeResource ControlContentThemeFontSize}"/>
            <Setter Property="UseSystemFocusVisuals" Value="True"/>
            <Setter Property="Template">
                <Setter.Value>
                    <ControlTemplate TargetType="Button">
                        <Grid x:Name="RootGrid" Background="{TemplateBinding Background}">
                            <VisualStateManager.VisualStateGroups>
                                <VisualStateGroup x:Name="CommonStates">
                                    <VisualState x:Name="Normal">
                                        <Storyboard>
                                            <PointerUpThemeAnimation
Storyboard.TargetName="RootGrid"/>
                                        </Storyboard>
                                    </VisualState>
                                    <VisualState x:Name="PointerOver">
                                        <Storyboard>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="BorderBrush" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0" Value="#e89300"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Foreground" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlHighlightBaseHighBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <PointerUpThemeAnimation
Storyboard.TargetName="RootGrid"/>
                                        </Storyboard>
                                    </VisualState>
                                    <VisualState x:Name="Pressed">
                                        <Storyboard>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Foreground" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlHighlightBaseHighBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <PointerDownThemeAnimation
Storyboard.TargetName="RootGrid"/>
                                        </Storyboard>
                                    </VisualState>
                                    <VisualState x:Name="Disabled">
                                        <Storyboard>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Background" Storyboard.TargetName="RootGrid">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlBackgroundBaseLowBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Foreground" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlDisabledBaseLowBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="BorderBrush" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlDisabledTransparentBrush}"/>
```

```
                                            </ObjectAnimationUsingKeyFrames>
                                        </Storyboard>
                                    </VisualState>
                                </VisualStateGroup>
                            </VisualStateManager.VisualStateGroups>
                            <ContentPresenter x:Name="ContentPresenter"
AutomationProperties.AccessibilityView="Raw" BorderBrush="{TemplateBinding BorderBrush}"
BorderThickness="{TemplateBinding BorderThickness}" ContentTemplate="{TemplateBinding
ContentTemplate}" ContentTransitions="{TemplateBinding ContentTransitions}" Content="{TemplateBinding
Content}" HorizontalContentAlignment="{TemplateBinding HorizontalContentAlignment}"
Padding="{TemplateBinding Padding}" VerticalContentAlignment="{TemplateBinding
VerticalContentAlignment}"/>
                        </Grid>
                    </ControlTemplate>
                </Setter.Value>
            </Setter>
        </Style>
    </Page.Resources>

    <Grid>
        <Grid.Background>
            <ImageBrush Stretch="UniformToFill" ImageSource="Assets/bg-wood.png"/>
        </Grid.Background>
        <Rectangle Height="20" StrokeThickness="10" VerticalAlignment="Top" Fill="#FF6B2509"
Width="1920" />
        <Button x:Name="fullscreen_btn" Style="{StaticResource topbarButtonStyle}" Content=""
Click="fullscreen_btn_clicked" HorizontalAlignment="Right" Height="18" VerticalAlignment="Top"
Width="18" Margin="0,1,10,0">
            <Button.Background>
                <ImageBrush ImageSource="Assets/fullscreen-arrows.png" Stretch="Uniform"/>
            </Button.Background>
        </Button>
        <Button x:Name="back_btn" Style="{StaticResource topbarButtonStyle}" Content=""
Click="back_btn_clicked"  HorizontalAlignment="Left" Height="20" VerticalAlignment="Top" Width="20">
            <Button.Background>
                <ImageBrush ImageSource="Assets/back-arrow.png" Stretch="Uniform"/>
            </Button.Background>
        </Button>
        <RelativePanel Height="922" Margin="940,-35,58,0" VerticalAlignment="Top"
RenderTransformOrigin="0.5,0.5" d:LayoutOverrides="LeftPosition, RightPosition">
            <RelativePanel.Background>
                <ImageBrush ImageSource="Assets/BackboardBrandubh.png"/>
            </RelativePanel.Background>
            <RelativePanel.RenderTransform>
                <CompositeTransform ScaleY="0.75" ScaleX="0.75"/>
            </RelativePanel.RenderTransform>
            <Grid Margin="2,2,-2,-2" RenderTransformOrigin="0.5,0.5">
                <Grid.RenderTransform>
                    <CompositeTransform ScaleY="0.959999997854232788" ScaleX="0.959999997854232788"/>
                </Grid.RenderTransform>
                <Grid.RowDefinitions>
                    <RowDefinition Height="117*"/>
                    <RowDefinition Height="103*"/>
                </Grid.RowDefinitions>

                <Image Width="880" Height="880" Source="Assets/ArdRiBoard.png" Margin="20,20,-20,0"
Grid.RowSpan="2" />
                <Image x:Name="gifImg" gif:AnimationBehavior.SourceUri=""
gif:AnimationBehavior.RepeatBehavior="Forever" Width="880" Height="880" Source="Assets/ArdRiBoard.png"
Margin="20,20,-20,0" Grid.RowSpan="2" />
            </Grid>

        </RelativePanel>
        <RelativePanel>

            <Rectangle HorizontalAlignment="Right" Height="200" Margin="0,800,-1760,-25"
StrokeThickness="10" VerticalAlignment="Top" Width="720" RenderTransformOrigin="0.5,0.5"
d:LayoutOverrides="HorizontalAlignment">
                <Rectangle.Fill>
                    <ImageBrush ImageSource="Assets/RulesText.png"/>
                </Rectangle.Fill>
            </Rectangle>
            <Image x:Name="textGif" gif:AnimationBehavior.SourceUri=""
gif:AnimationBehavior.RepeatBehavior="Forever" Source="" Margin="0,0,-1760,-1000"
```

```
HorizontalAlignment="Right" Width="720" Height="200" VerticalAlignment="Bottom"
d:LayoutOverrides="HorizontalAlignment, VerticalAlignment" />
        </RelativePanel>
        <RelativePanel>
            <Button x:Name="movementBtn" Style="{StaticResource ButtonStyle1}"
Click="movementBtn_clicked" HorizontalAlignment="Left" Margin="80,80,0,0" VerticalAlignment="Top"
Width="880" FontSize="48" FontFamily="Sylfaen" BorderBrush="#FF000105" BorderThickness="5"
Foreground="{x:Null}" FontStretch="Expanded" FontWeight="Bold" RenderTransformOrigin="0.5,0.5"
Height="80" d:LayoutOverrides="HorizontalAlignment">
                <Button.Background>
                    <ImageBrush Stretch="Uniform" ImageSource="Assets/MovementButton.png"/>
                </Button.Background>
            </Button>
            <Button x:Name="captureBtn" Style="{StaticResource ButtonStyle1}"
Click="captureBtn_clicked" HorizontalAlignment="Left" Margin="80,200,0,0" VerticalAlignment="Top"
Width="880" FontSize="48" FontFamily="Sylfaen" BorderBrush="#FF000105" BorderThickness="5"
Foreground="{x:Null}" FontStretch="Expanded" FontWeight="Bold" RenderTransformOrigin="0.5,0.5"
Height="80" d:LayoutOverrides="HorizontalAlignment">
                <Button.Background>
                    <ImageBrush Stretch="Uniform" ImageSource="Assets/CapturingButton.png"/>
                </Button.Background>
            </Button>
            <Button x:Name="winAttackBtn" Style="{StaticResource ButtonStyle1}"
Click="winAttackBtn_clicked" HorizontalAlignment="Left" Margin="80,320,0,0" VerticalAlignment="Top"
Width="880" FontSize="48" FontFamily="Sylfaen" BorderBrush="#FF000105" BorderThickness="5"
Foreground="{x:Null}" FontStretch="Expanded" FontWeight="Bold" RenderTransformOrigin="0.5,0.5"
Height="80">
                <Button.Background>
                    <ImageBrush Stretch="Uniform" ImageSource="Assets/WinAttackersButton.png"/>
                </Button.Background>
            </Button>
            <Button x:Name="winDefendBtn" Style="{StaticResource ButtonStyle1}"
Click="winDefBtn_clicked" HorizontalAlignment="Left" Margin="80,440,0,0" VerticalAlignment="Top"
Width="880" FontSize="48" FontFamily="Sylfaen" BorderBrush="#FF000105" BorderThickness="5"
Foreground="{x:Null}" FontStretch="Expanded" FontWeight="Bold" RenderTransformOrigin="0.5,0.5"
Height="80">
                <Button.Background>
                    <ImageBrush Stretch="Uniform" ImageSource="Assets/WinDefendersButton.png"/>
                </Button.Background>
            </Button>
            <Button x:Name="stratBlockBtn" Style="{StaticResource ButtonStyle1}"
Click="blockBtn_clicked" HorizontalAlignment="Left" Margin="80,0,0,-760" VerticalAlignment="Bottom"
Width="880" FontSize="48" FontFamily="Sylfaen" BorderBrush="#FF000105" BorderThickness="5"
Foreground="{x:Null}" FontStretch="Expanded" FontWeight="Bold" RenderTransformOrigin="0.5,0.5"
d:LayoutOverrides="VerticalAlignment" Height="80">
                <Button.Background>
                    <ImageBrush Stretch="Uniform" ImageSource="Assets/BlockadeButton.png"/>
                </Button.Background>
            </Button>
            <Button x:Name="stratTowerBtn" Style="{StaticResource ButtonStyle1}"
Click="towerBtn_clicked" HorizontalAlignment="Left" Margin="80,0,0,-640" VerticalAlignment="Bottom"
Width="880" FontSize="48" FontFamily="Sylfaen" BorderBrush="#FF000105" BorderThickness="5"
Foreground="{x:Null}" FontStretch="Expanded" FontWeight="Bold" RenderTransformOrigin="0.5,0.5"
d:LayoutOverrides="VerticalAlignment" Height="80">
                <Button.Background>
                    <ImageBrush Stretch="Uniform" ImageSource="Assets/TowerButton.png"/>
                </Button.Background>
            </Button>
        </RelativePanel>
    </Grid>
</Page>
```

3.4.2.2. Code-Behind

```
using System;
using VikingChess.Model;
using Windows.UI.ViewManagement;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Media.Imaging;
using XamlAnimatedGif;
```

```
namespace VikingChess
{
    /// <summary>
    /// The RulesPage displays information on how to play the game.
    /// </summary>
    public sealed partial class RulesPage : Page
    {
        Game game;

        /// <summary>
        /// Constructor for RulesPage
        /// </summary>
        public RulesPage()
        {
            this.InitializeComponent();
            game = new Game();

        }

        /// <summary>
        /// Returns the user to the main page
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void back_btn_clicked(object sender, RoutedEventArgs e)
        {
            if (this.Frame.CanGoBack)
            {
                this.Frame.GoBack();
            }
        }

        /// <summary>
        /// Puts the application into fullscreen
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void fullscreen_btn_clicked(object sender, RoutedEventArgs e)
        {
            ApplicationView view = ApplicationView.GetForCurrentView();
            ImageBrush fullscreenImg = new ImageBrush();

            bool isInFullScreenMode = view.IsFullScreenMode;

            if (isInFullScreenMode)
            {
                fullscreenImg.ImageSource = new BitmapImage(new Uri(this.BaseUri, "/Assets/fullscreen-
arrows.png"));
                view.ExitFullScreenMode();
                this.fullscreen_btn.Background = fullscreenImg;
            }
            else
            {
                fullscreenImg.ImageSource = new BitmapImage(new Uri(this.BaseUri, "/Assets/windowed-
arrows.png"));
                view.TryEnterFullScreenMode();
                this.fullscreen_btn.Background = fullscreenImg;
            }
        }

        /// <summary>
        /// Displays the movement text and animation
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void movementBtn_clicked(object sender, RoutedEventArgs e)
        {
            game.PlaySound("btn_click.wav");
            AnimationBehavior.SetSourceUri(gifImg, new Uri(this.BaseUri, "/Assets/Moving.gif"));
            AnimationBehavior.SetSourceUri(textGif, new Uri(this.BaseUri,
"/Assets/MovementText.gif"));
        }
        /// <summary>
        /// Displays the capture text and animation
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void captureBtn_clicked(object sender, RoutedEventArgs e)
        {
            game.PlaySound("btn_click.wav");
            AnimationBehavior.SetSourceUri(gifImg, new Uri(this.BaseUri, "/Assets/Capturing.gif"));
            AnimationBehavior.SetSourceUri(textGif, new Uri(this.BaseUri, 
"/Assets/CapturingText.gif"));
        }

        /// <summary>
        /// Displays the attacker's win condition text and animation
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void winAttackBtn_clicked(object sender, RoutedEventArgs e)
        {
            game.PlaySound("btn_click.wav");
            AnimationBehavior.SetSourceUri(gifImg, new Uri(this.BaseUri, "/Assets/WinAttackers.gif"));
            AnimationBehavior.SetSourceUri(textGif, new Uri(this.BaseUri, 
"/Assets/WinAttackerText.gif"));
        }

        /// <summary>
        /// Displays the defender's win condition text and animation
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void winDefBtn_clicked(object sender, RoutedEventArgs e)
        {
            game.PlaySound("btn_click.wav");
            AnimationBehavior.SetSourceUri(gifImg, new Uri(this.BaseUri, "/Assets/WinDefenders.gif"));
            AnimationBehavior.SetSourceUri(textGif, new Uri(this.BaseUri, 
"/Assets/WinDefenderText.gif"));
        }

        /// <summary>
        /// Displays the Blockade text and animation
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void blockBtn_clicked(object sender, RoutedEventArgs e)
        {
            game.PlaySound("btn_click.wav");
            AnimationBehavior.SetSourceUri(gifImg, new Uri(this.BaseUri, 
"/Assets/StrategyBlockade.gif"));
            AnimationBehavior.SetSourceUri(textGif, new Uri(this.BaseUri, 
"/Assets/BlockadeText.gif"));
        }

        /// <summary>
        /// Displays the Tower text and animation
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void towerBtn_clicked(object sender, RoutedEventArgs e)
        {
            game.PlaySound("btn_click.wav");
            AnimationBehavior.SetSourceUri(gifImg, new Uri(this.BaseUri, 
"/Assets/StrategyTower.gif"));
            AnimationBehavior.SetSourceUri(textGif, new Uri(this.BaseUri, "/Assets/TowerText.gif"));
        }
    }
}
```

3.5. About Page

3.5.1. Description

The About Page details information on the game and its developer.

3.5.2. Code

3.5.2.1. XAML

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:VikingChess"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:XamlAnimatedGif="using:XamlAnimatedGif"
    x:Class="VikingChess.AboutPage"
    mc:Ignorable="d">

    <Page.Resources>
        <Style x:Key="topbarButtonStyle" TargetType="Button">
            <Setter Property="Background" Value="{ThemeResource
SystemControlBackgroundBaseLowBrush}"/>
            <Setter Property="Foreground" Value="{ThemeResource
SystemControlForegroundBaseHighBrush}"/>
            <Setter Property="BorderBrush" Value="{ThemeResource
SystemControlForegroundTransparentBrush}"/>
            <Setter Property="BorderThickness" Value="{ThemeResource ButtonBorderThemeThickness}"/>
            <Setter Property="Padding" Value="8,4,8,4"/>
            <Setter Property="HorizontalAlignment" Value="Left"/>
            <Setter Property="VerticalAlignment" Value="Center"/>
            <Setter Property="FontFamily" Value="{ThemeResource ContentControlThemeFontFamily}"/>
            <Setter Property="FontWeight" Value="Normal"/>
            <Setter Property="FontSize" Value="{ThemeResource ControlContentThemeFontSize}"/>
            <Setter Property="UseSystemFocusVisuals" Value="True"/>
            <Setter Property="Template">
                <Setter.Value>
                    <ControlTemplate TargetType="Button">
                        <Grid x:Name="RootGrid" Background="{TemplateBinding Background}">
                            <VisualStateManager.VisualStateGroups>
                                <VisualStateGroup x:Name="CommonStates">
                                    <VisualState x:Name="Normal">
                                        <Storyboard>
                                            <PointerUpThemeAnimation
Storyboard.TargetName="RootGrid"/>
                                        </Storyboard>
                                    </VisualState>
                                    <VisualState x:Name="PointerOver">
                                        <Storyboard>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Foreground" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlHighlightBaseHighBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <PointerUpThemeAnimation
Storyboard.TargetName="RootGrid"/>
                                        </Storyboard>
                                    </VisualState>
                                    <VisualState x:Name="Pressed">
                                        <Storyboard>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Foreground" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlHighlightBaseHighBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <PointerDownThemeAnimation
Storyboard.TargetName="RootGrid"/>
                                        </Storyboard>
                                    </VisualState>
                                    <VisualState x:Name="Disabled">
                                        <Storyboard>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Background" Storyboard.TargetName="RootGrid">
```

```
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlBackgroundBaseLowBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Foreground" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlDisabledBaseLowBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="BorderBrush" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlDisabledTransparentBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                        </Storyboard>
                                    </VisualState>
                                </VisualStateGroup>
                            </VisualStateManager.VisualStateGroups>
                            <ContentPresenter x:Name="ContentPresenter"
AutomationProperties.AccessibilityView="Raw" BorderBrush="{TemplateBinding BorderBrush}"
BorderThickness="{TemplateBinding BorderThickness}" ContentTemplate="{TemplateBinding
ContentTemplate}" ContentTransitions="{TemplateBinding ContentTransitions}" Content="{TemplateBinding
Content}" HorizontalContentAlignment="{TemplateBinding HorizontalContentAlignment}"
Padding="{TemplateBinding Padding}" VerticalContentAlignment="{TemplateBinding
VerticalContentAlignment}"/>
                        </Grid>
                    </ControlTemplate>
                </Setter.Value>
            </Setter>
        </Style>
        <Style x:Key="ButtonStyle1" TargetType="Button">
            <Setter Property="Background" Value="{ThemeResource
SystemControlBackgroundBaseLowBrush}"/>
            <Setter Property="Foreground" Value="{ThemeResource
SystemControlForegroundBaseHighBrush}"/>
            <Setter Property="BorderBrush" Value="{ThemeResource
SystemControlForegroundTransparentBrush}"/>
            <Setter Property="BorderThickness" Value="{ThemeResource ButtonBorderThemeThickness}"/>
            <Setter Property="Padding" Value="8,4,8,4"/>
            <Setter Property="HorizontalAlignment" Value="Left"/>
            <Setter Property="VerticalAlignment" Value="Center"/>
            <Setter Property="FontFamily" Value="{ThemeResource ContentControlThemeFontFamily}"/>
            <Setter Property="FontWeight" Value="Normal"/>
            <Setter Property="FontSize" Value="{ThemeResource ControlContentThemeFontSize}"/>
            <Setter Property="UseSystemFocusVisuals" Value="True"/>
            <Setter Property="Template">
                <Setter.Value>
                    <ControlTemplate TargetType="Button">
                        <Grid x:Name="RootGrid" Background="{TemplateBinding Background}">
                            <VisualStateManager.VisualStateGroups>
                                <VisualStateGroup x:Name="CommonStates">
                                    <VisualState x:Name="Normal">
                                        <Storyboard>
                                            <PointerUpThemeAnimation
Storyboard.TargetName="RootGrid"/>
                                        </Storyboard>
                                    </VisualState>
                                    <VisualState x:Name="PointerOver">
                                        <Storyboard>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="BorderBrush" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0" Value="#e89300"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Foreground" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlHighlightBaseHighBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <PointerUpThemeAnimation
Storyboard.TargetName="RootGrid"/>
                                        </Storyboard>
                                    </VisualState>
                                    <VisualState x:Name="Pressed">
                                        <Storyboard>
```

```
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Foreground" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlHighlightBaseHighBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <PointerDownThemeAnimation
Storyboard.TargetName="RootGrid"/>
                                        </Storyboard>
                                    </VisualState>
                                    <VisualState x:Name="Disabled">
                                        <Storyboard>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Background" Storyboard.TargetName="RootGrid">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlBackgroundBaseLowBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="Foreground" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlDisabledBaseLowBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                            <ObjectAnimationUsingKeyFrames
Storyboard.TargetProperty="BorderBrush" Storyboard.TargetName="ContentPresenter">
                                                <DiscreteObjectKeyFrame KeyTime="0"
Value="{ThemeResource SystemControlDisabledTransparentBrush}"/>
                                            </ObjectAnimationUsingKeyFrames>
                                        </Storyboard>
                                    </VisualState>
                                </VisualStateGroup>
                            </VisualStateManager.VisualStateGroups>
                            <ContentPresenter x:Name="ContentPresenter"
AutomationProperties.AccessibilityView="Raw" BorderBrush="{TemplateBinding BorderBrush}"
BorderThickness="{TemplateBinding BorderThickness}" ContentTemplate="{TemplateBinding
ContentTemplate}" ContentTransitions="{TemplateBinding ContentTransitions}" Content="{TemplateBinding
Content}" HorizontalContentAlignment="{TemplateBinding HorizontalContentAlignment}"
Padding="{TemplateBinding Padding}" VerticalContentAlignment="{TemplateBinding
VerticalContentAlignment}"/>
                        </Grid>
                    </ControlTemplate>
                </Setter.Value>
            </Setter>
        </Style>
    </Page.Resources>

    <Grid>
        <Grid.Background>
            <ImageBrush Stretch="Fill" ImageSource="Assets/bg-wood.png"/>
        </Grid.Background>

        <Rectangle Height="20" StrokeThickness="10" VerticalAlignment="Top" Fill="#FF6B2509"
Width="1920" />
        <Button x:Name="fullscreen_btn" Style="{StaticResource topbarButtonStyle}" Content=""
Click="fullscreen_btn_clicked" HorizontalAlignment="Right" Height="18" VerticalAlignment="Top"
Width="18" Margin="0,1,10,0">
            <Button.Background>
                <ImageBrush ImageSource="Assets/fullscreen-arrows.png" Stretch="Uniform"/>
            </Button.Background>
        </Button>
        <Button x:Name="back_btn" Style="{StaticResource topbarButtonStyle}" Content=""
Click="back_btn_clicked"  HorizontalAlignment="Left" Height="20" VerticalAlignment="Top" Width="20">
            <Button.Background>
                <ImageBrush ImageSource="Assets/back-arrow.png" Stretch="Uniform"/>
            </Button.Background>
        </Button>
        <RelativePanel>
            <Rectangle HorizontalAlignment="Right" Height="640" Margin="0,0,-1840,-960"
StrokeThickness="10" VerticalAlignment="Bottom" Width="1760">
                <Rectangle.Fill>
                    <ImageBrush ImageSource="Assets/AboutText.png"/>
                </Rectangle.Fill>
            </Rectangle>
        </RelativePanel>
        <Rectangle Height="230" Margin="80,80,0,0" VerticalAlignment="Top" HorizontalAlignment="Left"
Width="230">
```

```
            <Rectangle.Fill>
                <ImageBrush Stretch="Uniform" ImageSource="Assets/VikingLogoMainPage.png"/>
            </Rectangle.Fill>
        </Rectangle>
        <Rectangle Margin="245,120,0,0" HorizontalAlignment="Left" Height="97" VerticalAlignment="Top"
Width="505">
            <Rectangle.Fill>
                <ImageBrush Stretch="Uniform" ImageSource="Assets/VikingChessLogoMainPage.png"/>
            </Rectangle.Fill>
        </Rectangle>
        <TextBlock x:Name="textBlock" HorizontalAlignment="Center" TextWrapping="Wrap" Text="Version:
1.0.0" VerticalAlignment="Center" Margin="1724,1040,5,5" Height="35" Width="191"
Foreground="#FFFFA200" FontFamily="Assets/Fonts/VIKING-N.TTF#Viking-Normal" FontSize="18"
TextAlignment="Center"/>

    </Grid>
</Page>
```

3.5.2.2. Code-Behind

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Runtime.InteropServices.WindowsRuntime;
using Windows.Foundation;
using Windows.Foundation.Collections;
using Windows.UI.ViewManagement;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Controls.Primitives;
using Windows.UI.Xaml.Data;
using Windows.UI.Xaml.Input;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Media.Imaging;
using Windows.UI.Xaml.Navigation;


namespace VikingChess
{
    /// <summary>
    /// The AboutPage details information about the development of this application
    /// </summary>
    public sealed partial class AboutPage : Page
    {
        /// <summary>
        /// Constructor for AboutPage
        /// </summary>
        public AboutPage()
        {
            this.InitializeComponent();
        }

        /// <summary>
        /// Returns the user to the main page
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void back_btn_clicked(object sender, RoutedEventArgs e)
        {
            if (this.Frame.CanGoBack)
            {
                this.Frame.GoBack();
            }
        }

        /// <summary>
        /// Puts the application into fullscreen
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void fullscreen_btn_clicked(object sender, RoutedEventArgs e)
        {
            ApplicationView view = ApplicationView.GetForCurrentView();
            ImageBrush fullscreenImg = new ImageBrush();

            bool isInFullScreenMode = view.IsFullScreenMode;

            if (isInFullScreenMode)
            {
                fullscreenImg.ImageSource = new BitmapImage(new Uri(this.BaseUri, "/Assets/fullscreen-
arrows.png"));
                view.ExitFullScreenMode();
                this.fullscreen_btn.Background = fullscreenImg;
            }
            else
            {
                fullscreenImg.ImageSource = new BitmapImage(new Uri(this.BaseUri, "/Assets/windowed-
arrows.png"));
                view.TryEnterFullScreenMode();
                this.fullscreen_btn.Background = fullscreenImg;
            }
        }
    }
}
```