unit LoadU;

interface

uses

Winapi.Windows, Winapi.Messages, System.SysUtils, System.Variants, System.Classes, Vcl.Graphics,

Vcl.Controls, Vcl.Forms, Vcl.Dialogs, Vcl.ExtCtrls, Vcl.ComCtrls,

Vcl.Imaging.GIFImg, Vcl.Imaging.pngimage, Vcl.StdCtrls;

type

TLoading\_Screen = class(TForm)

ProgressBar1: TProgressBar;

Timer1: TTimer;

Image1: TImage;

Label1: TLabel;

procedure Timer1Timer(Sender: TObject);

procedure FormCreate(Sender: TObject);

private

{ Private declarations }

public

{ Public declarations }

end;

var

Loading\_Screen: TLoading\_Screen;

implementation

uses Launcher;

{$R \*.dfm}

procedure TLoading\_Screen.FormCreate(Sender: TObject);

begin

Image1.Picture.LoadFromFile('Images\load.png');

PostMessage(ProgressBar1.Handle, $0409, 0, clBlue);

progressbar1.BarColor:=clblue;

end;

procedure TLoading\_Screen.Timer1Timer(Sender: TObject);

begin

Progressbar1.Position:=Progressbar1.Position+25;

if progressbar1.position=100 then

begin

Timer1.Enabled:=false;

frmLauncher.Show;

Loading\_Screen.Hide;

end;

end;

end.

unit Launcher;

interface

uses

Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,

Dialogs, StdCtrls, pngimage, ExtCtrls, PlayerU, EngineUI, Spin,

engineclasses, math, Vcl.Menus, ShellAPI,

Vcl.Buttons, Vcl.MPlayer;

type

TfrmLauncher = class(TForm)

Image1: TImage;

sedMonitor: TSpinEdit;

Label1: TLabel;

Label2: TLabel;

sedWidth: TSpinEdit;

lblHeight: TLabel;

shpPlay: TShape;

Label3: TLabel;

Label4: TLabel;

tmr: TTimer;

Label5: TLabel;

shpClose: TShape;

Image2: TImage;

Label6: TLabel;

shpDisplaySettings: TShape;

MainMenu1: TMainMenu;

N1: TMenuItem;

N2: TMenuItem;

N3: TMenuItem;

Panel1: TPanel;

MediaPlayer1: TMediaPlayer;

VolumeButton: TSpeedButton;

cbxWindowed: TCheckBox;

procedure LaunchGame;

procedure FormCreate(Sender: TObject);

procedure sedWidthChange(Sender: TObject);

procedure tmrTimer(Sender: TObject);

procedure N2Click(Sender: TObject);

procedure N3Click(Sender: TObject);

procedure FormShow(Sender: TObject);

procedure VolumeButtonClick(Sender: TObject);

private

{ Private declarations }

public

isMusicAllowed:boolean;

end;

var

frmLauncher: TfrmLauncher;

showingSettings : boolean = false;

iHeight : integer = 360;

rectPlay, rectSettings : TShape;

const

heightWOSettings = 335;

heightWSettings = 460;

implementation

{$R \*.dfm}

procedure TfrmLauncher.LaunchGame;

begin

ChessForm.PlayerRefresh.Enabled := false;

ChessForm.WindowState := wsNormal;

ChessForm.Show;

if not cbxWindowed.checked then

begin

ChessForm.Top := screen.Monitors[sedMonitor.Value - 1].Top;

ChessForm.Left := screen.Monitors[sedMonitor.Value - 1].Left;

gameHeight := screen.Monitors[sedMonitor.Value - 1].Height;

gameWidth := screen.Monitors[sedMonitor.Value - 1].Width;

ChessForm.WindowState := wsMaximized;

end

else

begin

ChessForm.Top := screen.Monitors[sedMonitor.Value - 1].Top;

ChessForm.Left := screen.Monitors[sedMonitor.Value - 1].Left;

gameHeight := iHeight;

gameWidth := sedWidth.Value;

ChessForm.ClientHeight := gameHeight;

ChessForm.ClientWidth := gameWidth;

ChessForm.roundEdges;

end;

ChessForm.reloadGame;

end;

procedure TfrmLauncher.N2Click(Sender: TObject);

begin

ShellExecute(0, PChar ('Open'), PChar ('NewProject.chm'), nil, nil, SW\_SHOW);

end;

procedure TfrmLauncher.N3Click(Sender: TObject);

begin

Application.Terminate;

end;

procedure TfrmLauncher.FormCreate(Sender: TObject);

var

rgn : HRGN;

begin

isMusicAllowed:=true;

Image1.Picture.LoadFromFile('Images\Launch\_horse.bmp');

Image2.Picture.LoadFromFile('Images\close.png');

label6.Caption := 'П Р И Я Т Н О Й ' + #13 + 'И Г Р Ы';

sedMonitor.MaxValue := screen.MonitorCount;

if screen.MonitorCount = 1 then

sedMonitor.Enabled := false;

sedWidth.MaxValue := screen.Width;

lblHeight.Caption := format('X %d', [iHeight]);

rgn := CreateRoundRectRgn(0,

0,

ClientWidth,

ClientHeight,

20,

20);

SetWindowRgn(Handle, rgn, True);

end;

procedure TfrmLauncher.FormShow(Sender: TObject);

begin

var path:string:=ExtractFilePath((Application.ExeName) );

self.MediaPlayer1.FileName := path + '\Images\TheHappyBride.mp3';

try

self.MediaPlayer1.Open();

self.MediaPlayer1.Play();

except

begin

MessageDlg('Неверный путь к файлу. Возможно его больше не существует. Попробуйте еще.',vcl.Dialogs.mtError, mbOKCancel, 0);

exit;

end;

end;

end;

procedure TfrmLauncher.sedWidthChange(Sender: TObject);

begin

if sedWidth.Text <> '' then

iHeight := ceil(sedWidth.value/(16/9));

lblHeight.Caption := format('X %d', [iHeight]);

end;

procedure TfrmLauncher.tmrTimer(Sender: TObject);

begin

if (mouse.CursorPos.X >= shpPlay.ClientToScreen(Point(0, 0)).X) AND

(mouse.CursorPos.X <= shpPlay.ClientToScreen(Point(shpPlay.Width, 0)).X)

AND (mouse.CursorPos.Y >= shpPlay.ClientToScreen(Point(0, 0)).Y) AND

(mouse.CursorPos.Y <= shpPlay.ClientToScreen(Point(0, shpPlay.Height)).Y)

then

begin

while GETGVALUE(shpPlay.Brush.color) > $BE do

begin

shpPlay.Brush.color := shpPlay.Brush.color - $000100;

Application.ProcessMessages;

end;

if GetKeyState(VK\_LBUTTON) < 0 then

begin

tmr.Enabled := false;

LaunchGame;

end;

end

else

begin

while GETGVALUE(shpPlay.Brush.color) < $DD do

begin

shpPlay.Brush.color := shpPlay.Brush.color + $000100;

Application.ProcessMessages;

end;

end;

if (mouse.CursorPos.X >= shpDisplaySettings.ClientToScreen(Point(0, 0)).X) AND

(mouse.Curso rPos.X <= shpDisplaySettings.ClientToScreen(Point(shpDisplaySettings.Width, 0)).X)

AND (mouse.CursorPos.Y >= shpDisplaySettings.ClientToScreen(Point(0, 0)).Y) AND

(mouse.CursorPos.Y <= shpDisplaySettings.ClientToScreen(Point(0, shpDisplaySettings.Height)).Y)

then

begin

while GETGVALUE(shpDisplaySettings.Brush.color) > $BE do

begin

shpDisplaySettings.Brush.color := shpDisplaySettings.Brush.color - $000100;

Application.ProcessMessages;

cbxWindowed.Color := shpDisplaySettings.Brush.Color;

end;

end

else

begin

while GETGVALUE(shpDisplaySettings.Brush.color) < $DD do

begin

shpDisplaySettings.Brush.color := shpDisplaySettings.Brush.color + $000100;

Application.ProcessMessages;

cbxWindowed.Color := shpDisplaySettings.Brush.Color;

end;

end;

if (mouse.CursorPos.X >= shpClose.ClientToScreen(Point(0, 0)).X) AND

(mouse.CursorPos.X <= shpClose.ClientToScreen(Point(shpClose.Width, 0)).X)

AND (mouse.CursorPos.Y >= shpClose.ClientToScreen(Point(0, 0)).Y) AND

(mouse.CursorPos.Y <= shpClose.ClientToScreen(Point(0, shpClose.Height)).Y)

then

begin

while GETGVALUE(shpClose.Brush.color) > $BE do

begin

shpClose.Brush.color := shpClose.Brush.color - $000100;

Application.ProcessMessages;

end;

if GetKeyState(VK\_LBUTTON) < 0 then

begin

tmr.Enabled := false;

Application.Terminate;

end;

end

else

begin

while GETGVALUE(shpClose.Brush.color) < $DD do

begin

shpClose.Brush.color := shpClose.Brush.color + $000100;

Application.ProcessMessages;

end;

end;

end;

procedure TfrmLauncher.VolumeButtonClick(Sender: TObject);

begin

if self.isMusicAllowed then

begin

self.isMusicAllowed:=false;

self.VolumeButton.Glyph.LoadFromFile(ExtractFilePath(Application.ExeName) +'/Images/mute.bmp');

self.MediaPlayer1.Stop

end

else

begin

self.isMusicAllowed:=true;

self.VolumeButton.Glyph.LoadFromFile(ExtractFilePath(Application.ExeName) +'/Images/speaker.bmp');

try

MediaPlayer1.FileName:=ExtractFilePath(Application.ExeName)+ '\Images\TheHappyBride.mp3';

self.MediaPlayer1.Open();

MediaPlayer1.Play

except

begin

MessageDlg('Неверный путь к файлу. Возможно его больше не существует. Попробуйте еще.',vcl.Dialogs.mtError, mbOKCancel, 0);

exit;

end;

end;

end;

end;

end.

unit PlayerU;

interface

uses

Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,

Dialogs, EngineClasses, jpeg, math, StdCtrls, ImgList, ExtCtrls, pngimage,

Menus, ActnList, EngineUI, System.Actions, System.ImageList, Vcl.Buttons;

type

TChessForm = class(TForm)

PlayerRefresh: TTimer;

lblWhiteTitle: TLabel;

lblBlackTitle: TLabel;

lblWPiecesTook: TLabel;

lblBPiecesTook: TLabel;

highlightblock: TImage;

imgClose: TImage;

imgCloseHover: TImage;

imgCloseDef: TImage;

Settings: TActionList;

setWhiteColor: TAction;

setBlackColor: TAction;

setOutlineColor: TAction;

setBackColor: TAction;

autoDeselect: TAction;

saveDirSet: TAction;

saveSettings: TAction;

resetSettings: TAction;

setAssetsPath: TAction;

setUIScale: TAction;

AssetsList: TImageList;

cldlg: TColorDialog;

procedure FormDestroy(Sender: TObject);

procedure FormKeyDown(Sender: TObject; var Key: Word; Shift: TShiftState);

procedure PlayerRefreshTimer(Sender: TObject);

procedure imgCloseClick(Sender: TObject);

procedure imgCloseMouseEnter(Sender: TObject);

procedure imgCloseMouseLeave(Sender: TObject);

procedure FormCreate(Sender: TObject);

procedure setWhiteColorExecute(Sender: TObject);

procedure setBlackColorExecute(Sender: TObject);

procedure setOutlineColorExecute(Sender: TObject);

procedure setBackColorExecute(Sender: TObject);

procedure autoDeselectExecute(Sender: TObject);

procedure SetSettings;

procedure saveSettingsExecute(Sender: TObject);

procedure showDebugExecute(Sender: TObject);

procedure resetSettingsExecute(Sender: TObject);

procedure reloadGame;

procedure setAssetsPathExecute(Sender: TObject);

procedure roundEdges;

procedure ScaleComponents;

procedure FormMouseDown(Sender: TObject; Button: TMouseButton;

Shift: TShiftState; X, Y: Integer);

procedure setUIScaleExecute(Sender: TObject);

procedure exitButtonClick(Sender: TObject);

function GetData(FilePath : string; Tag : string) : string;

private

{ Private declarations }

public

{ Public declarations }

end;

var

ChessForm: TChessForm;

BoardMannager: TBoardMannager;

highlightblock: TImage;

SaveManager: TSaveManager;

AssetPath: string = 'default';

turnColor : TColor;

selectColor : TColor;

scaleMultplier : real = 1;

implementation

{$R \*.dfm}

procedure TChessForm.autoDeselectExecute(Sender: TObject);

begin

BoardMannager.autoDeselect := not BoardMannager.autoDeselect;

autoDeselect.Checked := BoardMannager.autoDeselect

end;

function TChessForm.GetData(FilePath : string; Tag : string) : string;

var

tS : TextFile;

s : string;

begin

if not fileExists(filepath) then

begin

result := 'default';

exit;

end;

AssignFile(ts, FilePath);

reset(ts);

while (Pos(Tag, s) = 0) AND (not eof(tS)) do

readln(ts, s);

if eof(ts) then begin closeFile(tS); exit end;

delete(s, 1, pos('[', s));

Result := Copy(s, 1, pos(']', s) - 1);

closeFile(tS);

end;

procedure TChessForm.FormCreate(Sender: TObject);

var

tempbm: TBitmap;

settingDat: string;

begin

highlightblock.Picture.LoadFromFile('Images\block.png');

imgClose.Picture.LoadFromFile('Images\close.png');

imgCloseDef.Picture.LoadFromFile('Images\white\_block.png');

imgCloseHover.Picture.LoadFromFile('Images\close.png');

settingDat := getdata('\_SETTINGS.DWCS', 'AssetsDir');

if DirectoryExists(settingDat) then

AssetPath := settingDat

else

AssetPath := 'default';

if AssetPath <> 'default' then

imageSize := StrToInt(getdata(AssetPath + '\\_SETUP.DWCS', 'ImageSize'))

else

imageSize := 32;

tempbm := TBitmap.Create;

with tempbm do

begin

PixelFormat := pf32bit;

Height := imageSize;

Width := Height;

end;

BoardMannager := TBoardMannager.Create(Self);

SaveManager := TSaveManager.Create(Self);

SaveManager.LinkedBoard := BoardMannager;

color := rgb(102, 202, 255);

SetSettings;

if AssetPath = 'default' then

try

AssetsList.Draw(BoardMannager.Bishop.Canvas, 0, 0, 0, true);

AssetsList.Draw(BoardMannager.Castle.Canvas, 0, 0, 1, true);

AssetsList.Draw(BoardMannager.horse.Canvas, 0, 0, 2, true);

AssetsList.Draw(BoardMannager.king.Canvas, 0, 0, 3, true);

AssetsList.Draw(BoardMannager.pawn.Canvas, 0, 0, 4, true);

AssetsList.Draw(BoardMannager.queen.Canvas, 0, 0, 5, true);

finally

BoardMannager.Orientation := orTop\_Bottom;

BoardMannager.InitialDraw;

end;

scalecomponents;

autoDeselect.Checked := BoardMannager.autoDeselect;

end;

procedure TChessForm.FormDestroy(Sender: TObject);

begin

BoardMannager.destroy;

end;

procedure TChessForm.FormKeyDown(Sender: TObject; var Key: Word;

Shift: TShiftState);

begin

case key of

VK\_ESCAPE:

begin

if boardmannager.selected then

begin

boardmannager.selected := false;

if boardmannager.Turn = 1 then

boardmannager.turn := 2

else

boardmannager.turn := 1;

end;

end;

VK\_END:

begin

BoardMannager.Clear;

BoardMannager.InitialDraw;

end;

end;

end;

procedure TChessForm.FormMouseDown(Sender: TObject; Button: TMouseButton;

Shift: TShiftState; X, Y: Integer);

const

SC\_DRAGMOVE = $F012;

begin

if WindowState = wsNormal then

if Button = mbLeft then

begin

ReleaseCapture;

Perform(WM\_SYSCOMMAND, SC\_DRAGMOVE, 0);

end;

end;

procedure TChessForm.imgCloseClick(Sender: TObject);

begin

Application.Terminate;

end;

procedure TChessForm.imgCloseMouseEnter(Sender: TObject);

begin

imgClose.Picture := imgCloseHover.Picture;

end;

procedure TChessForm.imgCloseMouseLeave(Sender: TObject);

begin

imgClose.Picture := imgCloseDef.Picture;

end;

procedure TChessForm.PlayerRefreshTimer(Sender: TObject);

var

sWPT, sBPT : string;

I: Integer;

y, x, newKind: Integer;

begin

turnColor := RGB(GetGValue(Color), GetBValue(Color), GetRValue(Color));

selectColor := RGB(GetBValue(Color), GetRValue(Color), GetGValue(Color));

lblWhiteTitle.Caption := 'БЕЛЫЙ ИГРОК';

lblBlackTitle.Caption := 'ЧЕРНЫЙ ИГРОК';

for I := 0 to boardmannager.getBlackTooklength do

begin

case boardMannager.BlackPiecesTook[i] of

0: ;

1: sBPT := sBPT + 'Пешка' + nl;

2: sBPT := sBPT + 'Ладья' + nl;

3: sBPT := sBPT + 'Слог' + nl;

4: sBPT := sBPT + 'Конь' + nl;

5: sBPT := sBPT + 'Ферзь' + nl;

end;

end;

for I := 0 to boardmannager.getWhiteTooklength do

begin

case boardMannager.WhitePiecesTook[i] of

0: ;

1: sWPT := sWPT + 'Пешка' + nl;

2: sWPT := sWPT + 'Ладья' + nl;

3: sWPT := sWPT + 'Слон' + nl;

4: sWPT := sWPT + 'Конь' + nl;

5: sWPT := sWPT + 'Ферзь' + nl;

end;

end;

lblWPiecesTook.caption := sWPT;

lblBPiecesTook.Caption := sBPT;

if (BoardMannager.turn = 1) AND (Not BoardMannager.Selected) then

begin

lblWhiteTitle.font.Color := turncolor;

lblBlackTitle.font.Color := clblack;

end

else if (BoardMannager.turn = 2) AND (Not BoardMannager.Selected) then

begin

lblWhiteTitle.font.Color := clblack;

lblBlackTitle.font.Color := turncolor;

end

else if (BoardMannager.turn = 2) AND (BoardMannager.Selected) then

begin

lblWhiteTitle.font.Color := selectcolor;

lblBlackTitle.font.Color := clblack;

end

else if (BoardMannager.turn = 1) AND (BoardMannager.Selected) then

begin

lblWhiteTitle.font.Color := clblack;

lblBlackTitle.font.Color := selectcolor;

end;

with boardmannager do

begin

if Orientation = orTop\_Bottom then

begin

for y := 1 to 2 do

for x := 1 to 8 do

if (board[x, y \* 7 -6].kind = 1) then

begin

board[x, y \* 7 -6].Kind := 0;

newKind := pickpawnpromotion;

SetSquareTo(point(x, y\* 7 -6), newKind);

end

else if (board[x, y \* 7 -6].kind = -1) then

begin

board[x, y \* 7 -6].Kind := 0;

newKind := pickpawnpromotion;

SetSquareTo(point(x, y\* 7 -6), -1 \* newKind);

end;

end

else

begin

for x := 1 to 2 do

for y := 1 to 8 do

if board[x \* 7 -6, y].kind = 1 then

begin

board[x \* 7 -6, y].Kind := 0;

newKind := pickpawnpromotion;

SetSquareTo(point(x \* 7 -6, y), newKind);

end

else if board[x \* 7 -6, y].kind = -1 then

begin

board[x \* 7 -6, y].Kind := 0;

newKind := pickpawnpromotion;

SetSquareTo(point(x \* 7 -6, y), -1 \* newKind);

end;

end;

end;

if boardmannager.selected then

begin

highlightblock.Visible := true;

highlightblock.Top := boardmannager.SelectedSqr.Top;

highlightblock.Left := boardmannager.SelectedSqr.left;

end

else

highlightblock.Visible := false;

end;

procedure TChessForm.reloadGame;

begin

SaveManager.SaveToFileOverwrite('\_RESETTEMP.DWCS');

PlayerRefresh.Enabled := false;

BoardMannager.destroy;

BoardMannager := nil;

SaveManager.Destroy;

FormCreate(nil);

SaveManager.LoadFromFile('\_RESETTEMP.DWCS');

DeleteFile('\_RESETTEMP.DWCS');

DeleteFile('\_RESETTEMP.PGN');

PlayerRefresh.Enabled := true;

end;

procedure TChessForm.resetSettingsExecute(Sender: TObject);

var

tS : textfile;

begin

assignfile(ts, '\_SETTINGS.DWCS');

rewrite(ts);

write(tS, 'WhiteColor=[default]'#13#10'BlackColor=[default]'#13#10'OutlineColor=[default]'#13#10'BackColor=[default]'#13#10'SaveDir=[default]'#13#10'AutoDeselect=[default]'#13#10'ShowDebug=[default]'#13#10'AssetsDir=[default]'#13#10'END');

closefile(tS);

reloadGame;

end;

procedure TChessForm.roundEdges;

var

rgn : HRGN;

begin

rgn := CreateRoundRectRgn(0,

0,

chessform.ClientWidth,

chessform.ClientHeight,

40,

40);

SetWindowRgn(chessform.Handle, rgn, True);

end;

procedure TChessForm.saveSettingsExecute(Sender: TObject);

var

tS : textFile;

showDebug, autoDeselect : string;

begin

if NOT BoardMannager.Debug.Visible then

showdebug := 'false'

else

showdebug := 'true';

if NOT BoardMannager.AutoDeselect then

autoDeselect := 'false'

else

autoDeselect := 'true';

assignfile(ts, '\_SETTINGS.DWCS');

rewrite(ts);

write(tS, format(

'WhiteColor=[%d]'#13#10'BlackColor=[%d]'#13#10'OutlineColor=[%d]'#13#10''

+ 'BackColor=[%d]'#13#10'SaveDir=[%s]'#13#10'AutoDeselect=[%s]'#13#10'ShowDebug=[%s]'#13#10'AssetsDir=[%s]'#13#10'END',

[rgb(GetBValue(BoardMannager.WhiteColor), GetGValue(BoardMannager.WhiteColor),GetRValue(BoardMannager.WhiteColor)),

rgb(GetBValue(BoardMannager.BlackColor), GetGValue(BoardMannager.BlackColor),GetRValue(BoardMannager.BlackColor)),

rgb(GetBValue(BoardMannager.OutlineColor), GetGValue(BoardMannager.OutlineColor),GetRValue(BoardMannager.OutlineColor)),

color, savemanager.rootDir, autoDeselect, showdebug, assetPath]));

closefile(tS);

end;

procedure TChessForm.ScaleComponents;

begin

lblWhiteTitle.Top := 8;

lblWhiteTitle.Font.Size := Ceil((20 / (1080/ClientHeight)) \* scaleMultplier);

lblWhiteTitle.Left := 8;

lblBPiecesTook.Left := 8;

lblBPiecesTook.Top := lblWhiteTitle.Top + lblWhiteTitle.Height + 8;

lblBlackTitle.Top := 8;

lblBPiecesTook.Font.Size := Ceil((12 / (1080/ClientHeight))\* scaleMultplier);

lblBlackTitle.Font.Size := Ceil((20 / (1080/ClientHeight))\* scaleMultplier);

lblBlackTitle.Left := BoardMannager.getLastSquareLeft +

BoardMannager.getSquareHeightWidth + 8;

lblWPiecesTook.Top := lblBlackTitle.Top + lblBlackTitle.Height + 8;

lblWPiecesTook.Font.Size := Ceil((12 / (1080/ClientHeight))\* scaleMultplier);

lblWPiecesTook.Left := BoardMannager.getLastSquareLeft +

BoardMannager.getSquareHeightWidth + 8;

highlightblock.BringToFront;

highlightblock.Parent := Self;

highlightblock.Stretch := true;

highlightblock.Visible := false;

highlightblock.Height := BoardMannager.Board[1, 1].Height;

highlightblock.Width := BoardMannager.Board[1, 1].Width;

imgClose.Width := Ceil((45 / (1080/ClientHeight))\* scaleMultplier);

imgClose.Height := Ceil((45 / (1080/ClientHeight))\* scaleMultplier);

imgClose.Left := chessform.Width - imgClose.Width - 8;

BoardMannager.Debug.Font.Size := Ceil((10 / (1080/ClientHeight))\* scaleMultplier);

end;

procedure TChessForm.setAssetsPathExecute(Sender: TObject);

var

prePath : string;

accept : integer;

begin

prePath := AssetPath;

AssetPath := InputBox('', '', AssetPath);

if prePath <> AssetPath then

accept := MessageDlg('' , mtConfirmation, [mbYes, mbNo], 0);

if accept = mrYes then

begin

saveSettingsExecute(nil);

reloadGame;

end;

end;

procedure TChessForm.setBackColorExecute(Sender: TObject);

begin

cldlg.Color := color;

clDlg.Execute();

Color := clDlg.Color;

end;

procedure TChessForm.setBlackColorExecute(Sender: TObject);

begin

clDlg.Color := rgb(GetBValue(BoardMannager.BlackColor), GetGValue(BoardMannager.BlackColor),GetRValue(BoardMannager.BlackColor));

clDlg.Execute();

if clDlg.Color = $000000 then

clDlg.Color := $000001;

if clDlg.Color = $FFFFFF then

clDlg.Color := $FFFFFE;

BoardMannager.BlackColor := clDlg.Color;

end;

procedure TChessForm.setOutlineColorExecute(Sender: TObject);

begin

clDlg.Color := rgb(GetBValue(BoardMannager.OutlineColor), GetGValue(BoardMannager.OutlineColor),GetRValue(BoardMannager.OutlineColor));

clDlg.Execute();

if clDlg.Color = $000000 then

clDlg.Color := $000001;

if clDlg.Color = $FFFFFF then

clDlg.Color := $FFFFFE;

BoardMannager.OutlineColor := clDlg.Color;

end;

procedure TChessForm.SetSettings;

var

settingDat : string;

begin

if fileexists('\_SETTINGS.DWCS') then

begin

settingDat := getdata('\_SETTINGS.DWCS', 'WhiteColor');

if settingDat <> 'default' then

BoardMannager.WhiteColor := StrToInt(settingDat);

settingDat := getdata('\_SETTINGS.DWCS', 'BlackColor');

if settingDat <> 'default' then

BoardMannager.BlackColor := StrToInt(settingDat);

settingDat := getdata('\_SETTINGS.DWCS', 'OutlineColor');

if settingDat <> 'default' then

BoardMannager.OutlineColor := StrToInt(settingDat);

settingDat := getdata('\_SETTINGS.DWCS', 'BackColor');

if settingDat <> 'default' then

chessForm.Color := StrToInt(settingDat);

settingDat := getdata('\_SETTINGS.DWCS', 'SaveDir');

if settingDat <> 'default' then

SaveManager.rootDir := settingDat;

settingDat := getdata('\_SETTINGS.DWCS', 'ShowDebug');

if settingDat <> 'default' then

if settingDat = 'false' then

BoardMannager.Debug.Visible := false;

settingDat := getdata('\_SETTINGS.DWCS', 'AutoDeselect');

if settingDat <> 'default' then

if settingDat = 'false' then

BoardMannager.AutoDeselect := false;

end;

end;

procedure TChessForm.setUIScaleExecute(Sender: TObject);

var

newScaleM : real;

begin

newScaleM := strtofloat(inputbox('Set new UI Scale', 'Enter a scale multiplier [Any real number]', FloatToStr(scaleMultplier)));

scaleMultplier := newscaleM;

ScaleComponents;

end;

procedure TChessForm.setWhiteColorExecute(Sender: TObject);

begin

clDlg.Color := rgb(GetBValue(BoardMannager.WhiteColor), GetGValue(BoardMannager.WhiteColor),GetRValue(BoardMannager.WhiteColor));

clDlg.Execute();

if clDlg.Color = $000000 then

clDlg.Color := $000001;

if clDlg.Color = $FFFFFF then

clDlg.Color := $FFFFFE;

BoardMannager.WhiteColor := clDlg.Color;

SaveManager.SaveToFileOverwrite(SaveManager.rootDir + '\\_TEMPSAVE.DWCS');

SaveManager.LoadFromFile(SaveManager.rootDir + '\\_TEMPSAVE.DWCS');

deleteFile(SaveManager.rootDir + '\\_TEMPSAVE.DWCS');

deletefile(SaveManager.rootDir + '\\_TEMPSAVE.PGN');

end;

procedure TChessForm.showDebugExecute(Sender: TObject);

begin

BoardMannager.Debug.Visible := not BoardMannager.Debug.Visible;

end;

procedure TChessForm.ExitButtonClick(Sender: TObject);

begin

self.Close();

end;

end.

unit EngineUI;

interface

uses

ExtCtrls, Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,

Forms, Dialogs, Math, StdCtrls, pngimage;

function PickPawnPromotion : byte;

implementation

function PickPawnPromotion : byte;

var

frm : TForm;

rgn: HRGN;

Done : boolean;

lbl : TLabel;

sR : array of string;

i, y, i2 : integer;

clickRegions : array of trect;

rects : array of TShape;

begin

Done := false;

frm := TForm.CreateNew(nil, 0);

frm.BorderStyle := bsNone;

frm.AlphaBlend := true;

frm.AlphaBlendValue := 0;

lbl := TLabel.Create(frm);

lbl.Parent := frm;

lbl.Caption := 'П Р О В Е Д Е Н Н А Я П Е Ш К А';

lbl.Font.Name := 'Arial';

lbl.Font.Size := 18;

lbl.Font.Color := $5D2FFF;

lbl.Font.Style := [fsBold];

frm.ClientWidth := lbl.Width + 40;

lbl.left := round((frm.ClientWidth/2) - (lbl.Width/2));

lbl.Top := 20;

i := 0;

for i := 1 to 4 do

begin

SetLength(rects, i);

SetLength(clickRegions, i);

SetLength(sR, i);

rects[i-1] := TShape.Create(frm);

with rects[i-1] do

begin

Parent := frm;

Width := frm.ClientWidth;

brush.Color := $FFDD69;

top := i\*(height+5);

pen.Style := psClear;

end;

case i of

1: sr[i - 1] := 'ЛАДЬЯ';

2: sr[i - 1] := 'СЛОН';

3: sr[i - 1] := 'КОНЬ';

4: sr[i - 1] := 'ФЕРЗЬ';

end;

lbl := TLabel.Create(frm);

with lbl do

begin

parent := frm;

Font.Name := 'Arial';

Font.Size := 18;

Font.Color := rgb(105,97,225);

Font.Style := [fsBold];

Top := round(rects[i-1].Top + (rects[i-1].Height/2) - (Height/2)) ;

Caption := sR[i-1];

Left := round((frm.Width/2) - (lbl.Width/2));

end;

end;

frm.ClientHeight := rects[i-2].Top + rects[i-2].Height + 20;

frm.Position := poScreenCenter;

rgn := CreateRoundRectRgn(0,

0,

frm.ClientWidth,

frm.ClientHeight,

20,

20);

SetWindowRgn(frm.Handle, rgn, True);

frm.Color := rgb(168,244,255);

frm.DoubleBuffered := true;

frm.Show;

i2 := 0;

while i2 < 250 do

begin

inc(i2, 2);

frm.AlphaBlendValue := i2;

Application.ProcessMessages;

end;

for y := 0 to i - 2 do

begin

clickRegions[y].Left := rects[y].ClientToScreen(point(0,0)).x;

clickRegions[y].Top := rects[y].ClientToScreen(point(0,0)).y;

clickRegions[y].Bottom := rects[y].ClientToScreen(point(0,0 + rects[y].height)).y;

clickRegions[y].Right := rects[y].ClientToScreen(point(0 + rects[y].width,0)).x;

end;

while not done do

begin

frm.BringToFront;

for Y := 0 to i - 2 do

begin

if (mouse.CursorPos.X >= clickRegions[y].Left) AND

(mouse.CursorPos.X <= clickRegions[y].Right) AND

(mouse.CursorPos.Y >= clickRegions[y].Top) AND

(mouse.CursorPos.Y <= clickRegions[y].Bottom) then

begin

while GETGVALUE(rects[y].Brush.Color) > $BE do

begin

rects[y].Brush.Color := rects[y].Brush.Color - $000100;

Application.ProcessMessages;

end;

if GetKeyState(VK\_LBUTTON) < 0 then

Done := True;

case sR[y][1] of

'Л' : result := 2;

'С' : result := 3;

'К' : result := 4;

'Ф' : result := 5;

end;

end

else

begin

while GETGVALUE(rects[y].Brush.Color) < $DD do

begin

rects[y].Brush.Color := rects[y].Brush.Color + $000100;

Application.ProcessMessages;

end;

end;

Application.ProcessMessages;

end;

Application.ProcessMessages;

end;

i2 := 250;

while i2 > 2 do

begin

dec(i2, 2);

frm.AlphaBlendValue := i2;

Application.ProcessMessages;

end;

frm.Destroy;

end;

end.

unit EngineClasses;

interface

uses

ExtCtrls, Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,

Forms, Dialogs, Math, StdCtrls, pngimage, EngineUI;

type

TLineAddTrigger = function(const s : string):integer of object;

TDebug = class(TMemo)

published

constructor Create(AOwner:TComponent);override;

public

println : TLineAddTrigger;

end;

TForward = procedure(ASquare : Pointer) of object;

TSquare = class(TImage)

private

FCords: TPoint;

FDebug: TDebug;

FForwardClick: TForward;

FKind: integer;

FColor: integer;

FPreKind: integer;

procedure SetCords(const Value: TPoint);

procedure Click(Sender:TObject);

procedure SetForwardClick(const Value: TForward);

procedure SetKind(const Value: integer);

procedure SetColor(const Value: integer);

procedure SetPreKind(const Value: integer);

published

constructor Create(AOwner : TComponent); override;

property Cords : TPoint read FCords write SetCords;

property PreKind : integer read FPreKind write SetPreKind;

property Kind : integer read FKind write SetKind;

property Color : integer read FColor write SetColor;

property ForwardClick : TForward read FForwardClick write SetForwardClick;

end;

TBoard = array[1..8] of array[1..8] of TSquare;

PDW = ^DWORD;

PSQR = ^TSquare;

TIntArray = array of integer;

TBoardMannager = class

private

FDebug: TDebug;

Fhorse: TBitmap;

Fpawn: TBitmap;

Fknight: TBitmap;

Fking: TBitmap;

Fqueen: TBitmap;

Fbishop: TBitmap;

FSelected: boolean;

FSelectedSqr: TSquare;

FTurn: integer;

InCheck : boolean;

FWhitePiecesTook: array of integer;

FBlackPiecesTook: array of integer;

FOrientation: Integer;

FAutoDeselect: boolean;

FOutlineColor: TColor;

FBlackColor: TColor;

FWhiteColor: TColor;

FPlayerNameWhite: string;

FPlayerNameBlack: string;

procedure SetDebug(const Value: TDebug);

procedure Setbishop(const Value: TBitmap);

procedure Sethorse(const Value: TBitmap);

procedure Setking(const Value: TBitmap);

procedure Setknight(const Value: TBitmap);

procedure Setpawn(const Value: TBitmap);

procedure Setqueen(const Value: TBitmap);

procedure SetSelected(const Value: boolean);

procedure SetSelectedSqr(const Value: TSquare);

procedure SetTurn(const Value: integer);

function GetBlackPiecesTook(index:integer):integer;

function GetWhitePiecesTook(index:integer):integer;

procedure SetBlackPiecesTook(Index: Integer; Value: Integer);

procedure SetWhitePiecesTook(Index: Integer; Value: Integer);

function Move(ASquare: TSquare; Abm : TBitmap) : integer;

procedure TakePiece(ASquare: TSquare; Abm : TBitmap);

procedure SetOrientation(const Value: Integer);

procedure SetAutoDeselect(const Value: boolean);

procedure SetBlackColor(const Value: TColor);

procedure SetOutlineColor(const Value: TColor);

procedure SetWhiteColor(const Value: TColor);

procedure SetPlayerNameBlack(const Value: string);

procedure SetPlayerNameWhite(const Value: string);

published

constructor create(AOwner : TForm);

destructor destroy;

property Debug : TDebug read FDebug write SetDebug;

procedure Click(ASquare : Pointer);

procedure InitialDraw;

procedure DrawBoard;

property AutoDeselect : boolean read FAutoDeselect write SetAutoDeselect;

property pawn : TBitmap read Fpawn write Setpawn;

property king : TBitmap read Fking write Setking;

property castle : TBitmap read Fknight write Setknight;

property queen : TBitmap read Fqueen write Setqueen;

property bishop : TBitmap read Fbishop write Setbishop;

property horse : TBitmap read Fhorse write Sethorse;

property Selected : boolean read FSelected write SetSelected;

property SelectedSqr : TSquare read FSelectedSqr write SetSelectedSqr;

property Orientation : Integer read FOrientation write SetOrientation;

property Turn : integer read FTurn write SetTurn;

property WhiteColor : TColor read FWhiteColor write SetWhiteColor;

property BlackColor : TColor read FBlackColor write SetBlackColor;

property OutlineColor : TColor read FOutlineColor write SetOutlineColor;

property PlayerNameWhite : string read FPlayerNameWhite write SetPlayerNameWhite;

property PlayerNameBlack : string read FPlayerNameBlack write SetPlayerNameBlack;

function getLastSquareLeft : integer;

function getSquareHeightWidth : integer;

function getBlackTookLength : integer;

function getWhiteTookLength : integer;

procedure Clear;

procedure InvalidMove;

procedure SetSquareTo(Location : TPoint; kind : integer);

public

Board : TBoard;

CastlingPossible : array[1..2] of boolean;

property WhitePiecesTook[Index:integer] : integer read GetWhitePiecesTook write SetWhitePiecesTook;

property BlackPiecesTook[Index:integer] : integer read GetBlackPiecesTook write SetBlackPiecesTook;

function CheckDetect : byte; overload;

function CheckDetect(APoint: TPoint; multiplier : integer) : boolean; overload;

end;

TSaveManager = class

private

FrootDir: string;

FLinkedBoard: TBoardMannager;

procedure SetLinkedBoard(const Value: TBoardMannager);

procedure SetrootDir(const Value: string);

published

constructor Create(AOwner:TObject);

property LinkedBoard : TBoardMannager read FLinkedBoard write SetLinkedBoard;

property rootDir : string read FrootDir write SetrootDir;

procedure SaveToFile(filepath : string);

procedure SaveToFileOverwrite(filepath : string);

procedure LoadFromFile(filepath : string);

end;

var

imageSize : integer = 32;

gameWidth, gameHeight : integer;

const

nl = #13#10;

orRight\_Left = 1;

orTop\_Bottom = 2;

implementation

{ TSquare }

procedure TSquare.Click(Sender: TObject);

begin

ForwardClick(Self);

end;

constructor TSquare.Create(AOwner: TComponent);

begin

inherited;

stretch := True;

Height := floor(gameHeight/8);

Width := height;

if AOwner IS TForm then

parent := TForm(AOwner);

kind := 0;

color := 0;

OnClick := Click;

end;

procedure TSquare.SetColor(const Value: integer);

begin

FColor := Value;

end;

procedure TSquare.SetCords(const Value: TPoint);

begin

FCords := Value;

end;

procedure TSquare.SetForwardClick(const Value: TForward);

begin

FForwardClick := Value;

end;

procedure TSquare.SetKind(const Value: integer);

begin

PreKind := FKind;

FKind := Value;

end;

procedure TSquare.SetPreKind(const Value: integer);

begin

FPreKind := Value;

end;

{ TDebug }

constructor TDebug.Create(AOwner: TComponent);

begin

inherited;

if AOwner IS TForm then

parent := TForm(AOwner);

lines.Clear;

width := floor(gamewidth/2) - floor((gamewidth/8) \* 2.25);

height := floor((gameheight/8)\*2);

top := gameHeight - height;

println := lines.Add;

ReadOnly := true;

Enabled := false;

end;

{ TBoardMannager }

function TBoardMannager.CheckDetect: byte;

var

blackCords, whiteCords, searchCords, search : TPoint;

x: Integer;

y: Integer;

i, i2: integer;

multiplier : integer;

bExit : boolean;

pawn1, pawn2 : TPoint;

horseLocation : array[1..8] of TPoint;

begin

result := 0;

for x := 1 to 8 do

for y := 1 to 8 do

if (Board[x,y].Kind = -6) then

blackCords := Point(x, y)

else if Board[x,y].Kind = 6 then

whiteCords := Point(x, y);

for i := 1 to 2 do

begin

if i = 1 then

begin

searchCords := whiteCords;

multiplier := -1;

end

else

begin

searchCords := blackCords;

multiplier := 1;

end;

bExit := false;

search := searchCords;

while (search.x + 1 < 9) AND (search.y - 1 > 0) AND (NOT bExit) do

begin

inc(search.x);

dec(search.y);

if (Board[search.x, search.y].Kind = 5 \* multiplier) or (Board[search.x, search.y].Kind = 3 \* multiplier) then

result := i

else if Board[search.x, search.y].Kind <> 0 then

bExit := true;

end;

bExit := false;

search := searchCords;

while (search.x + 1 < 9) AND (search.y + 1 < 9) AND (NOT bExit) do

begin

inc(search.x);

inc(search.y);

if (Board[search.x, search.y].Kind = 5 \* multiplier) or (Board[search.x, search.y].Kind = 3 \* multiplier) then

result := i

else if Board[search.x, search.y].Kind <> 0 then

bExit := true;

end;

bExit := false;

search := searchCords;

while (search.x - 1 > 0) AND (search.y - 1 > 0) AND (NOT bExit) do

begin

dec(search.x);

dec(search.y);

if (Board[search.x, search.y].Kind = 5 \* multiplier) or (Board[search.x, search.y].Kind = 3 \* multiplier) then

result := i

else if Board[search.x, search.y].Kind <> 0 then

bExit := true;

end;

bExit := false;

search := searchCords;

while (search.x - 1 > 0 ) AND (search.y + 1 < 9) AND (NOT bExit) do

begin

dec(search.x);

inc(search.y);

if (Board[search.x, search.y].Kind = 5 \* multiplier) or (Board[search.x, search.y].Kind = 3 \* multiplier) then

result := i

else if Board[search.x, search.y].Kind <> 0 then

bExit := true;

end;

bExit := false;

search := searchCords;

while (search.X - 1 > 0) AND (NOT bExit) do

begin

dec(search.x);

if (Board[search.x, search.y].Kind = 5 \* multiplier) or (Board[search.x, search.y].Kind = 2 \* multiplier) then

result := i

else if Board[search.x, search.y].Kind <> 0 then

bExit := true;

end;

bExit := false;

search := searchCords;

while (search.X + 1 < 9) AND (NOT bExit) do

begin

inc(search.x);

if (Board[search.x, search.y].Kind = 5 \* multiplier) or (Board[search.x, search.y].Kind = 2 \* multiplier) then

result := i

else if Board[search.x, search.y].Kind <> 0 then

bExit := true;

end;

bExit := false;

search := searchCords;

while (search.Y - 1 > 0) AND (NOT bExit) do

begin

dec(search.Y);

if (Board[search.x, search.y].Kind = 5 \* multiplier) or (Board[search.x, search.y].Kind = 2 \* multiplier) then

result := i

else if Board[search.x, search.y].Kind <> 0 then

bExit := true;

end;

bExit := false;

search := searchCords;

while (search.Y + 1 < 9) AND (NOT bExit) do

begin

inc(search.Y);

if (Board[search.x, search.y].Kind = 5 \* multiplier) or (Board[search.x, search.y].Kind = 2 \* multiplier) then

result := i

else if Board[search.x, search.y].Kind <> 0 then

bExit := true;

end;

bExit := false;

search := searchCords;

if (search.X > 1) AND (search.X < 8) then

begin

pawn1 := Point(search.X - 1, search.Y + multiplier);

pawn2 := Point(search.X + 1, search.Y + multiplier);

end

else if NOT (search.X < 8) then

begin

pawn1 := Point(search.X - 1, search.Y + multiplier);

pawn2 := pawn1;

end

else if NOT (search.X > 1) then

begin

pawn2 := Point(search.X + 1, search.Y + multiplier);

pawn1 := Pawn2;

end;

if (board[Pawn1.X, Pawn1.Y].Kind = 1 \* multiplier) OR

(board[Pawn2.X, Pawn2.Y].Kind = 1 \* multiplier) then

result := i;

horseLocation[1] := point(search.X - 1, search.Y - 2);

horseLocation[2] := point(search.X + 1, search.Y - 2);

horseLocation[3] := point(search.X - 2, search.Y - 1);

horseLocation[4] := point(search.X + 2, search.Y - 1);

horseLocation[5] := point(search.X - 2, search.Y + 1);

horseLocation[6] := point(search.X + 2, search.Y + 1);

horseLocation[7] := point(search.X - 1, search.Y + 2);

horseLocation[8] := point(search.X + 1, search.Y + 2);

for i2 := 1 to 8 do

begin

if (horseLocation[i2].X IN [1..8]) AND (horseLocation[i2].Y IN [1..8]) then

if board[horselocation[i2].x,horselocation[i2].Y].kind = 4 \* multiplier then

result := i;

end;

end;

end;

function TBoardMannager.CheckDetect(APoint: TPoint; multiplier : integer): boolean;

var

searchCords, search : TPoint;

x: Integer;

y: Integer;

i2: integer;

bExit : boolean;

pawn1, pawn2 : TPoint;

horseLocation : array[1..8] of TPoint;

const

i = true;

begin

result := false;

searchCords := APoint;

bExit := false;

search := searchCords;

while (search.x + 1 < 9) AND (search.y - 1 > 0) AND (NOT bExit) do

begin

inc(search.x);

dec(search.y);

if (Board[search.x, search.y].Kind = 5 \* multiplier) or

(Board[search.x, search.y].Kind = 3 \* multiplier) then

result := i

else if Board[search.x, search.y].Kind <> 0 then

bExit := True;

end;

bExit := false;

search := searchCords;

while (search.x + 1 < 9) AND (search.y + 1 < 9) AND (NOT bExit) do

begin

inc(search.x);

inc(search.y);

if (Board[search.x, search.y].Kind = 5 \* multiplier) or

(Board[search.x, search.y].Kind = 3 \* multiplier) then

result := i

else if Board[search.x, search.y].Kind <> 0 then

bExit := True;

end;

bExit := false;

search := searchCords;

while (search.x - 1 > 0) AND (search.y - 1 > 0) AND (NOT bExit) do

begin

dec(search.x);

dec(search.y);

if (Board[search.x, search.y].Kind = 5 \* multiplier) or

(Board[search.x, search.y].Kind = 3 \* multiplier) then

result := i

else if Board[search.x, search.y].Kind <> 0 then

bExit := True;

end;

bExit := false;

search := searchCords;

while (search.x - 1 > 0) AND (search.y + 1 < 9) AND (NOT bExit) do

begin

dec(search.x);

inc(search.y);

if (Board[search.x, search.y].Kind = 5 \* multiplier) or

(Board[search.x, search.y].Kind = 3 \* multiplier) then

result := i

else if Board[search.x, search.y].Kind <> 0 then

bExit := True;

end;

bExit := false;

search := searchCords;

while (search.x - 1 > 0) AND (NOT bExit) do

begin

dec(search.x);

if (Board[search.x, search.y].Kind = 5 \* multiplier) or

(Board[search.x, search.y].Kind = 2 \* multiplier) then

result := i

else if Board[search.x, search.y].Kind <> 0 then

bExit := True;

end;

bExit := false;

search := searchCords;

while (search.x + 1 < 9) AND (NOT bExit) do

begin

inc(search.x);

if (Board[search.x, search.y].Kind = 5 \* multiplier) or

(Board[search.x, search.y].Kind = 2 \* multiplier) then

result := i

else if Board[search.x, search.y].Kind <> 0 then

bExit := True;

end;

bExit := false;

search := searchCords;

while (search.y - 1 > 0) AND (NOT bExit) do

begin

dec(search.y);

if (Board[search.x, search.y].Kind = 5 \* multiplier) or

(Board[search.x, search.y].Kind = 2 \* multiplier) then

result := i

else if Board[search.x, search.y].Kind <> 0 then

bExit := True;

end;

bExit := false;

search := searchCords;

while (search.y + 1 < 9) AND (NOT bExit) do

begin

inc(search.y);

if (Board[search.x, search.y].Kind = 5 \* multiplier) or

(Board[search.x, search.y].Kind = 2 \* multiplier) then

result := i

else if Board[search.x, search.y].Kind <> 0 then

bExit := True;

end;

bExit := false;

search := searchCords;

if (search.x > 1) AND (search.x < 8) then

begin

pawn1 := Point(search.x - 1, search.y + multiplier);

pawn2 := Point(search.x + 1, search.y + multiplier);

end

else if NOT(search.x < 8) then

begin

pawn1 := Point(search.x - 1, search.y + multiplier);

pawn2 := pawn1;

end

else if NOT(search.x > 1) then

begin

pawn2 := Point(search.x + 1, search.y + multiplier);

pawn1 := pawn2;

end;

if (Board[pawn1.x, pawn1.y].Kind = 1 \* multiplier) OR

(Board[pawn2.x, pawn2.y].Kind = 1 \* multiplier) then

result := i;

horseLocation[1] := Point(search.x - 1, search.y - 2);

horseLocation[2] := Point(search.x + 1, search.y - 2);

horseLocation[3] := Point(search.x - 2, search.y - 1);

horseLocation[4] := Point(search.x + 2, search.y - 1);

horseLocation[5] := Point(search.x - 2, search.y + 1);

horseLocation[6] := Point(search.x + 2, search.y + 1);

horseLocation[7] := Point(search.x - 1, search.y + 2);

horseLocation[8] := Point(search.x + 1, search.y + 2);

for i2 := 1 to 8 do

begin

if (horseLocation[i2].x IN [1 .. 8]) AND (horseLocation[i2].y IN [1 .. 8])

then

if Board[horseLocation[i2].x,

horseLocation[i2].y].Kind = 4 \* multiplier then

result := i;

end;

end;

procedure TBoardMannager.Clear;

var

y, x, i: Integer;

t1, t2 : integer;

begin

t1 := GetTickCount;

selected := false;

for I := 0 to getBlackTooklength do

FBlackPiecesTook[i] := 0;

for I := 0 to getWhiteTookLength do

FWhitePiecesTook[i] := 0;

SetLength(fwhitePiecesTook, 1);

SetLength(fblackPiecesTook, 1);

turn := 1;

for y := 1 to 8 do

for x := 1 to 8 do

with Board[x, y] do

Kind := 0;

t2 := GetTickCount;

Debug.lines.Clear;end;

procedure TBoardMannager.Click(ASquare: Pointer);

var

Square : TSquare;

sDebugMSG : string;

difInY, difInX:integer;

difPawnForward, difPawnSide : integer;

bm: TBitmap;

x,y:integer;

pbase, p : PDW;

xMultiplier, yMin, yMax, newKind : integer;

I: Integer;

startcheckX, endcheckx, startchecky, endchecky: integer;

possibleCastling : boolean;

presquareKind : integer;

begin

InCheck := false;

bm := TBitmap.Create;

with bm do

begin

PixelFormat := pf32bit;

height := 1;

width := 1;

end;

Square := TSquare(ASquare);

sDebugMSG := 'Clicked On: X:' + IntToStr(Square.Cords.X) + ' Y:' + IntToStr(Square.Cords.Y);

else

begin

difInY := Square.Cords.Y - SelectedSqr.Cords.Y;

difInX := Square.Cords.X - SelectedSqr.Cords.X;

case SelectedSqr.Color of

1:

bm.Canvas.Pixels[0, 0] := $0;

2:

bm.Canvas.Pixels[0, 0] := $FFFFFF;

end;

case selectedsqr.kind of

-6..-1:

begin

xMultiplier := -1;

yMax := 6;

yMin := 1;

newKind := Selectedsqr.Kind;

end;

1..6:

begin

xMultiplier := 1;

yMax := -1;

yMin := -6;

newKind := Selectedsqr.Kind;

end;

end;

case SelectedSqr.Kind of

1, -1:

begin

case Orientation of

orRight\_Left:

begin

difPawnForward := difInX;

difPawnSide := difInY;

end;

orTop\_Bottom:

begin

difPawnForward := -difInY;

difPawnSide := -difInX;

end;

end;

if difPawnForward = (1 \* xMultiplier) then

begin

if (Square.Kind <> 0) AND (difPawnSide = 0) then

begin

invalidmove;

Exit;

end;

if (difPawnSide = 0) then

begin

move(square, bm);

exit;

end;

if difPawnSide <> 0 then

if (difPawnSide = 1) or (difPawnSide = -1) then

begin

if (Square.Kind >= yMin) AND (Square.Kind <= yMax) then

begin

takepiece(square, bm);

if not ((Square.Kind >= yMin) AND (Square.Kind <= yMax)) then

exit;

end

else

begin

invalidmove;

Exit;

end;

end

else

begin

invalidmove;

Exit;

end;

end

else if (difPawnForward = (2\*xMultiplier)) AND

(board[square.cords.x, square.cords.y + (1 \* xMultiplier)].Kind = 0) AND

((selectedsqr.Cords.Y = 7)or(selectedsqr.Cords.Y = 2)) then

begin

if (difPawnSide = 0) then

begin

move(square, bm);

exit;

end;

end

else

begin

invalidmove;

Exit;

end;

end;

2, -2:

begin

if (difInY <> 0) and (difInX <> 0) then

begin

invalidmove;

Exit;

end

else if difInY <> 0 then

begin

if difInY < 0 then

for I := Square.Cords.y + 1 to SelectedSqr.Cords.y - 1 do

begin

if Board[Square.Cords.x, I].Kind <> 0 then

begin

invalidmove;

Exit;

end;

end

else if difInY > 0 then

for I := SelectedSqr.Cords.y + 1 to Square.Cords.y - 1 do

begin

if Board[Square.Cords.x, I].Kind <> 0 then

begin

invalidmove;

Exit;

end;

end;

if (Square.Kind >= yMin) AND (Square.Kind <= yMax) then

begin

takepiece(square, bm);

if not ((Square.Kind >= yMin) AND (Square.Kind <= yMax)) then

castlingPossible[turn] := false;

exit;

end

else if Square.Kind = 0 then

begin

move(square, bm);

castlingPossible[turn] := false;

exit;

end

else

begin

invalidmove;

Exit;

end;

end

else if difInX <> 0 then

begin

if difInX < 0 then

for I := Square.Cords.X + 1 to SelectedSqr.Cords.X - 1 do

begin

if Board[I, square.Cords.y].Kind <> 0 then

begin

invalidmove;

Exit;

end;

end

else if difInX > 0 then

for I := SelectedSqr.Cords.X + 1 to Square.Cords.X - 1 do

begin

if Board[I, square.Cords.y].Kind <> 0 then

begin

invalidmove;

Exit;

end;

end;

if (Square.Kind >= yMin) AND (Square.Kind <= yMax) then

begin

takepiece(square,bm);

if not ((Square.Kind >= yMin) AND (Square.Kind <= yMax)) then

castlingPossible[turn] := false;

exit;

end

else if Square.Kind = 0 then

begin

move(square, bm);

castlingPossible[turn] := false;

exit;

end

else

begin

invalidmove;

Exit;

end;

end;

end;

3, -3:

begin

if difInX > 0 then

begin

if (difInY = difInX) or (difInY = -difInX) then

begin

i := 0;

for x := SelectedSqr.Cords.x + 1 to Square.Cords.x - 1 do

begin

inc(i);

if difInY = difInX then

y := SelectedSqr.Cords.y + i

else

y := SelectedSqr.Cords.y - i;

if Board[x, y].Kind <> 0 then

begin

invalidmove;

Exit;

end;

end;

if Square.Kind = 0 then

begin

move(square, bm);

exit;

end;

if (Square.Kind >= yMin) AND (Square.Kind <= yMax) then

begin

takepiece(square, bm);

if not ((Square.Kind >= yMin) AND (Square.Kind <= yMax)) then

exit;

end

else

begin

invalidmove;

Exit;

end;

end

else

begin

invalidmove;

Exit;

end;

end

else if difInX < 0 then

begin

if (difInY = difInX) or (difInY = -difInX) then

begin

i := 0;

for x := SelectedSqr.Cords.x - 1 downto Square.Cords.x + 1 do

begin

dec(i);

if difInY = difInX then

y := SelectedSqr.Cords.y + i

else

y := SelectedSqr.Cords.y - i;

if Board[x, y].Kind <> 0 then

begin

invalidmove;

Exit;

end;

end;

if Square.Kind = 0 then

begin

move(square, bm);

exit;

end;

if (Square.Kind >= yMin) AND (Square.Kind <= yMax) then

begin

takepiece(square, bm);

if not ((Square.Kind >= yMin) AND (Square.Kind <= yMax)) then

exit;

end

else

begin

invalidmove;

Exit;

end;

end

else

begin

invalidmove;

Exit;

end;

end

else

begin

invalidmove;

Exit;

end;

end;

4, -4:

begin

if (((difInX = 2) or (difInX = -2)) and ((difInY = 1) or (difInY = -1)))

or (((difInY = 2) or (difInY = -2)) and ((difInX = 1) or (difInX = -1))) then

begin

if Square.Kind = 0 then

begin

move(square, bm);

exit;

end

else if (Square.Kind >= yMin) AND (Square.Kind <= yMax) then

begin

takepiece(square,bm);

if not ((Square.Kind >= yMin) AND (Square.Kind <= yMax)) then

exit;

end

else

begin

invalidmove;

Exit;

end;

end

else

begin

invalidmove;

Exit;

end;

end;

5, -5:

begin

if (difInY = difInX) or (difInY = -difInX) then

begin

if difInX > 0 then

begin

i := 0;

for x := SelectedSqr.Cords.x + 1 to Square.Cords.x - 1 do

begin

inc(i);

if difInY = difInX then

y := SelectedSqr.Cords.y + i

else

y := SelectedSqr.Cords.y - i;

if Board[x, y].Kind <> 0 then

begin

invalidmove;

Exit;

end;

end;

if Square.Kind = 0 then

begin

Move(Square, bm);

Exit;

end;

if (Square.Kind >= yMin) AND (Square.Kind <= yMax) then

begin

TakePiece(Square, bm);

if not ((Square.Kind >= yMin) AND (Square.Kind <= yMax)) then

Exit;

end

else

begin

invalidmove;

Exit;

end;

end

else if difInX < 0 then

begin

i := 0;

for x := SelectedSqr.Cords.x - 1 downto Square.Cords.x + 1 do

begin

dec(i);

if difInY = difInX then

y := SelectedSqr.Cords.y + i

else

y := SelectedSqr.Cords.y - i;

if Board[x, y].Kind <> 0 then

begin

invalidmove;

Exit;

end;

end;

if Square.Kind = 0 then

begin

Move(Square, bm);

Exit;

end;

if (Square.Kind >= yMin) AND (Square.Kind <= yMax) then

begin

TakePiece(Square, bm);

if not ((Square.Kind >= yMin) AND (Square.Kind <= yMax)) then

Exit;

end

else

begin

invalidmove;

Exit;

end;

end

else

begin

invalidmove;

Exit;

end;

end

else if (difInY <> 0) and (difinx = 0) then

begin

if difInY < 0 then

for I := Square.Cords.y + 1 to SelectedSqr.Cords.y - 1 do

begin

if Board[Square.Cords.x, I].Kind <> 0 then

begin

invalidmove;

Exit;

end;

end

else if difInY > 0 then

for I := SelectedSqr.Cords.y + 1 to Square.Cords.y - 1 do

begin

if Board[Square.Cords.x, I].Kind <> 0 then

begin

invalidmove;

Exit;

end;

end;

if (Square.Kind >= yMin) AND (Square.Kind <= yMax) then

begin

takepiece(square,bm);

if not ((Square.Kind >= yMin) AND (Square.Kind <= yMax)) then

exit;

end

else if Square.Kind = 0 then

begin

move(square, bm);

exit;

end

else

begin

invalidmove;

Exit;

end;

end

else if (difInX <> 0) and (difiny = 0) then

begin

if difInX < 0 then

for I := Square.Cords.X + 1 to SelectedSqr.Cords.X - 1 do

begin

if Board[I, square.Cords.y].Kind <> 0 then

begin

invalidmove;

Exit;

end;

end

else if difInX > 0 then

for I := SelectedSqr.Cords.X + 1 to Square.Cords.X - 1 do

begin

if Board[I, square.Cords.y].Kind <> 0 then

begin

invalidmove;

Exit;

end;

end;

if (Square.Kind >= yMin) AND (Square.Kind <= yMax) then

begin

takepiece(square,bm);

if not ((Square.Kind >= yMin) AND (Square.Kind <= yMax)) then

exit;

end

else if Square.Kind = 0 then

begin

move(square, bm);

exit;

end

else

begin

invalidmove;

Exit;

end;

end

else

begin

invalidmove;

Exit;

end;

end;

6, -6:

begin

if ((difInX < 2) AND (difInX > -2)) AND ((difInY < 2) AND (difInY > -2)) then

begin

startcheckX := square.Cords.X - 1;

if startcheckX < 1 then

startcheckx := 1;

endcheckX := square.Cords.X + 1;

if endcheckX > 8 then

endcheckx := 8;

startcheckY := square.Cords.Y - 1;

if startcheckY < 1 then

startcheckY := 1;

endcheckY := square.Cords.Y + 1;

if endcheckY > 8 then

endcheckY := 8;

for y := startcheckY to endcheckY do

for x := startcheckX to endcheckX do

begin

if board[x,y].Kind = selectedsqr.kind \* -1 then

begin

beep;

Exit;

end;

end;

if square.Kind = 0 then

begin

Move(square, bm);

castlingPossible[turn] := false;

exit;

end

else if (Square.Kind >= yMin) AND (Square.Kind <= yMax) then

begin

takepiece(square,bm);

if not ((Square.Kind >= yMin) AND (Square.Kind <= yMax)) then

castlingPossible[turn] := false;

exit;

end

else

begin

invalidmove;

Exit;

end;

end

else if ((((difInX > 1) or (difinx < -1)) and (Orientation = orTop\_Bottom)) and ((square.Kind = 0) AND (castlingpossible[turn]))) then

begin

possibleCastling := false;

if ((board[8, square.Cords.y].kind = 2 \* xmultiplier)) then

possibleCastling := true;

if square.Cords.X = 7 then

if ((board[8, square.Cords.y].kind = 2 \* xmultiplier)) then

possibleCastling := true

else if square.Cords.X = 2 then

if (board[1, square.Cords.y].Kind = 2 \* xmultiplier) then

possibleCastling := true;

if possibleCastling then

if SelectedSqr.Cords.x < Square.Cords.x then

begin

for i := SelectedSqr.Cords.x + 1 to Square.Cords.x - 1 do

if Board[i, Square.Cords.y].Kind <> 0 then

begin

possibleCastling := false;

end;

end

else

for i := SelectedSqr.Cords.x - 1 downto Square.Cords.x + 1 do

if Board[i, Square.Cords.y].Kind <> 0 then

begin

possibleCastling := false;

end;

if possibleCastling then

begin

if SelectedSqr.Cords.x < Square.Cords.x then

if CheckDetect(Point(SelectedSqr.Cords.x + 2, Square.Cords.y),

xMultiplier \* -1) = false then

begin

CastlingPossible[turn] := false;

move(square, bm);

setsquareto(point(8, square.Cords.y), 0);

if board[8, square.Cords.y].Color = 1 then

bm.Canvas.Pixels[0,0] := $0

else

bm.Canvas.Pixels[0,0] := $ffffff;

board[8, square.Cords.y].Picture.Bitmap := bm;

SetSquareTo(point(square.Cords.X - 1, square.Cords.Y), 2 \* xMultiplier);

end;

if SelectedSqr.Cords.x > Square.Cords.x then

begin

if CheckDetect(Point(SelectedSqr.Cords.x - 2, Square.Cords.y),

xMultiplier \* -1) = false then

begin

CastlingPossible[turn] := false;

move(board[SelectedSqr.Cords.x - 2, Square.Cords.y], bm);

setsquareto(point(1, square.Cords.y), 0);

if board[1, square.Cords.y].Color = 1 then

bm.Canvas.Pixels[0,0] := $0

else

bm.Canvas.Pixels[0,0] := $ffffff;

board[1, square.Cords.y].Picture.Bitmap := bm;

SetSquareTo(point(square.Cords.X + 2, square.Cords.Y), 2 \* xMultiplier);

end;

end;

end;

end

else

begin

invalidmove;

Exit;

end;

end;

end;

end;

freeAndNil(bm);

end;

constructor TBoardMannager.create(AOwner: TForm);

var

y, x, firstX: Integer;

bm : TBitmap;

t1, t2 : integer;

begin

BlackColor := $1F2635;

WhiteColor := $BED5FF;

OutlineColor := $505050;

AutoDeselect := true;

incheck := false;

selected := false;

firstX := floor(gameWidth/2) - floor((gameWidth/8) \* 2.25);

SetLength(fwhitePiecesTook, 1);

SetLength(fblackPiecesTook, 1);

turn := 1;

Debug := TDebug.Create(AOwner);

debug.Visible:=false;

for y := 1 to 8 do

for x := 1 to 8 do

begin

Board[x, y] := tsquare.Create(AOwner);

with board[x, y] do

begin

top := (y - 1) \* Height;

left := (x - 1) \* Height + firstX;

Cords := Point(x, y);

ForwardClick := self.Click;

end;

end;

pawn := TBitmap.Create;

with pawn do

begin

PixelFormat := pf32bit;

height := imagesize;

width := height;

end;

bishop := TBitmap.Create;

with bishop do

begin

PixelFormat := pf32bit;

height := imagesize;

width := height;

end;

castle := TBitmap.Create;

with castle do

begin

PixelFormat := pf32bit;

height := imagesize;

width := height;

end;

horse := TBitmap.Create;

with horse do

begin

PixelFormat := pf32bit;

height := imagesize;

width := height;

end;

king := TBitmap.Create;

with king do

begin

PixelFormat := pf32bit;

height := imagesize;

width := height;

end;

queen := TBitmap.Create;

with queen do

begin

PixelFormat := pf32bit;

height := imagesize;

width := height;

end;

Orientation := orRight\_Left;

end;

destructor TBoardMannager.destroy;

var

x, y : integer;

begin

for y := 1 to 8 do

for x := 1 to 8 do

freeandnil(Board[x,y]);

debug.Destroy;

pawn.Destroy;

bishop.Destroy;

castle.Destroy;

horse.Destroy;

king.Destroy;

queen.Destroy;

end;

procedure TBoardMannager.DrawBoard;

var

bm : TBitmap;

x, y : integer;

t1, t2 : integer;

begin

t1 := GetTickCount;

bm := TBitmap.Create;

with bm do

begin

PixelFormat := pf32bit;

height := 1;

width := 1;

end;

for y := 1 to 8 do

for x := 1 to 8 do

begin

with board[x, y] do

begin

if odd(x + y - orientation) then

begin

bm.Canvas.Pixels[0,0] := $000000;

color := 1;

end

else

begin

bm.Canvas.Pixels[0,0] := $ffffff;

color := 2;

end;

picture.Bitmap := bm

end;

end;

bm.Destroy;

t2 := GetTickCount;

end;

function TBoardMannager.GetBlackPiecesTook(index: integer): integer;

begin

result := FBlackPiecesTook[index];

end;

function TBoardMannager.getBlackTookLength: integer;

begin

result := length(FBlackPiecesTook) - 1;

end;

function TBoardMannager.getLastSquareLeft: integer;

begin

result := board[8,1].Left

end;

function TBoardMannager.getSquareHeightWidth: integer;

begin

result := board[1,1].Height;

end;

function TBoardMannager.GetWhitePiecesTook(index: integer): integer;

begin

result := FWhitePiecesTook[index];

end;

function TBoardMannager.getWhiteTookLength: integer;

begin

result := length(FWhitePiecesTook) - 1;

end;

procedure TBoardMannager.InitialDraw;

var

pbase, p : PDW;

y, y1, x, x1, i : integer;

tempbm: TBitmap;

t1, t2 : integer;

begin

drawboard;

castlingPossible[1] := true;

CastlingPossible[2] := true;

t1 := GetTickCount;

if Orientation = orTop\_Bottom then

begin

for x := 1 to 8 do

begin

SetSquareTo(Point(x, 7), 1);

SetSquareTo(Point(x, 2), -1);

end;

for x := 1 to 2 do

begin

SetSquareTo(Point( x \* 7 - 6, 8), 2);

SetSquareTo(Point( x \* 7 - 6, 1), -2);

SetSquareTo(Point( x \* 3,8), 3);

SetSquareTo(Point( x \* 3,1), -3);

SetSquareTo(Point( x \* 5 - 3, 8), 4);

SetSquareTo(Point( x \* 5 - 3, 1), -4);

end;

SetSquareTo(Point(5, 8), 6);

SetSquareTo(Point(5, 1), -6);

SetSquareTo(Point(4, 8), 5);

SetSquareTo(Point(4, 1), -5);

end

else

Begin

for y := 1 to 8 do

begin

SetSquareTo(Point(2, y), 1);

SetSquareTo(Point(7, y), -1);

end;

for y := 1 to 2 do

begin

SetSquareTo(Point(1, y \* 7 - 6), 2);

SetSquareTo(Point(8, y \* 7 - 6), -2);

SetSquareTo(Point(1, y \* 3), 3);

SetSquareTo(Point(8, y \* 3), -3);

SetSquareTo(Point(1, y \* 5 - 3), 4);

SetSquareTo(Point(8, y \* 5 - 3), -4);

end;

SetSquareTo(Point(1, 5), 6);

SetSquareTo(Point(8, 5), -6);

SetSquareTo(Point(1, 4), 5);

SetSquareTo(Point(8, 4), -5);

End;

t2 := GetTickCount;

end;

procedure TBoardMannager.InvalidMove;

begin

beep;

if autoDeselect then

begin

selected := false;

if turn = 1 then

turn := 2

else

turn := 1;

end;

end;

function TBoardMannager.Move(ASquare: TSquare; Abm : TBitmap) : integer;

var

Atempbm: Tbitmap;

pbase, p : PDW;

y,x : integer;

CheckTurn : byte;

reverseSelected, reverseSquare : TSquare;

squarebm : TBitmap;

begin

result := ASquare.kind;

reverseSelected := ASquare;

reverseSquare := SelectedSqr;

squarebm := TBitmap.Create;

with squarebm do

begin

PixelFormat := pf32bit;

Height := imageSize;

Width := Height;

end;

squarebm.Assign(ASquare.Picture.Bitmap);

if Turn = 1 then

CheckTurn := 2

else

CheckTurn := 1;

if asquare.Color <> selectedsqr.color then

begin

Atempbm := TBitmap.Create;

with Atempbm do

begin

PixelFormat := pf32bit;

Height := imageSize;

Width := Height;

end;

Atempbm.Assign(SelectedSqr.picture.Bitmap);

for y := 0 to imageSize - 1 do

for x := 0 to imageSize - 1 do

begin

pbase := Atempbm.ScanLine[y];

p := PDW(DWORD(pbase) + (x shl 2));

case ASquare.Color of

2:

if p^ = $0 then

p^ := $FFFFFF;

1:

if p^ = $FFFFFF then

p^ := $0;

end;

end;

ASquare.picture.Bitmap := Atempbm;

SelectedSqr.picture.Bitmap := Abm;

ASquare.Kind := SelectedSqr.Kind;

SelectedSqr.Kind := 0;

Selected := false;

freeandnil(atempbm);

end

else

begin

ASquare.picture.Bitmap := SelectedSqr.picture.Bitmap;

ASquare.Kind := selectedsqr.Kind;

SelectedSqr.Kind := 0;

SelectedSqr.picture.Bitmap := Abm;

Selected := false;

end;

if CheckDetect = CheckTurn then

begin

if not InCheck then

begin

incheck := true;

showmessage('Шах!');

beep;

SelectedSqr := reverseSelected;

Move(reverseSquare, squarebm);

ASquare.Kind := result;

selectedSqr := reverseSquare;

Turn := CheckTurn;

end

else

InCheck := false;

end

else if CheckDetect <> 0 then

selectedSqr := reverseSquare;

squarebm.Destroy;

end;

procedure TBoardMannager.SetAutoDeselect(const Value: boolean);

begin

FAutoDeselect := Value;

end;

procedure TBoardMannager.Setbishop(const Value: TBitmap);

begin

Fbishop := Value;

end;

procedure TBoardMannager.SetBlackColor(const Value: TColor);

begin

FBlackColor := rgb(GetBValue(value), GetGValue(Value),GetRValue(Value));

end;

procedure TBoardMannager.SetBlackPiecesTook(Index: Integer; Value: Integer);

begin

FBlackPiecesTook[Index] := Value;

end;

procedure TBoardMannager.SetDebug(const Value: TDebug);

begin

FDebug := Value;

end;

procedure TBoardMannager.Sethorse(const Value: TBitmap);

begin

Fhorse := Value;

end;

procedure TBoardMannager.Setking(const Value: TBitmap);

begin

Fking := Value;

end;

procedure TBoardMannager.Setknight(const Value: TBitmap);

begin

Fknight := Value;

end;

procedure TBoardMannager.SetOrientation(const Value: Integer);

begin

FOrientation := Value;

end;

procedure TBoardMannager.SetOutlineColor(const Value: TColor);

begin

FOutlineColor := rgb(GetBValue(value), GetGValue(Value),GetRValue(Value));

end;

procedure TBoardMannager.Setpawn(const Value: TBitmap);

begin

Fpawn := Value;

end;

procedure TBoardMannager.SetPlayerNameBlack(const Value: string);

begin

FPlayerNameBlack := Value;

end;

procedure TBoardMannager.SetPlayerNameWhite(const Value: string);

begin

FPlayerNameWhite := Value;

end;

procedure TBoardMannager.Setqueen(const Value: TBitmap);

begin

Fqueen := Value;

end;

procedure TBoardMannager.SetSelected(const Value: boolean);

begin

FSelected := Value;

end;

procedure TBoardMannager.SetSelectedSqr(const Value: TSquare);

begin

FSelectedSqr := Value;

end;

procedure TBoardMannager.SetSquareTo(Location: TPoint; Kind: integer);

var

tempbm: TBitmap;

x, y: integer;

pbase, p: PDW;

begin

if (Location.x IN [1 .. 8]) AND (Location.y IN [1 .. 8]) then

begin

tempbm := TBitmap.Create;

with tempbm do

begin

PixelFormat := pf32bit;

Height := imageSize;

Width := Height;

end;

case Kind of

1, -1:

tempbm.Assign(pawn);

2, -2:

tempbm.Assign(castle);

3, -3:

tempbm.Assign(bishop);

4, -4:

tempbm.Assign(horse);

5, -5:

tempbm.Assign(queen);

6, -6:

tempbm.Assign(king);

0:

begin

for y := 0 to imageSize - 1 do

for x := 0 to imageSize - 1 do

pbase := tempbm.ScanLine[y];

p := PDW(DWORD(pbase) + (x shl 2));

p^ := $0000FF;

end;

end;

for y := 0 to imageSize - 1 do

for x := 0 to imageSize - 1 do

begin

pbase := tempbm.ScanLine[y];

p := PDW(DWORD(pbase) + (x shl 2));

case p^ of

$0000FF:

if odd(Location.y + Location.x - orientation) then

p^ := $000000

else

p^ := $FFFFFF;

$00FF00:

p^ := outlineColor;

$FF0000:

begin

if Kind > 0 then

p^ := WhiteColor

else

p^ := BlackColor;

end;

end;

end;

if Kind <> 0 then

Board[Location.x, Location.y].picture.Bitmap := tempbm;

Board[Location.x, Location.y].Kind := Kind;

tempbm.Destroy;

end;

end;

procedure TBoardMannager.SetTurn(const Value: integer);

begin

FTurn := Value;

end;

procedure TBoardMannager.SetWhiteColor(const Value: TColor);

begin

FWhiteColor := rgb(GetBValue(value), GetGValue(Value),GetRValue(Value));

end;

procedure TBoardMannager.SetWhitePiecesTook(Index: Integer; Value: Integer);

begin

FWhitePiecesTook[Index] := Value;

end;

procedure TBoardMannager.TakePiece(ASquare: TSquare; Abm: TBitmap);

var

i, oKind: integer;

begin

okind := selectedsqr.Kind;

i := Move(ASquare, Abm);

if ASquare.Kind = oKind then

case i of

1 .. 6:

begin

WhitePiecesTook[ high(FWhitePiecesTook)] := i;

SetLength(FWhitePiecesTook, length(FWhitePiecesTook) + 1);

end;

-6 .. -1:

begin

BlackPiecesTook[ high(FBlackPiecesTook)] := i \* -1;

SetLength(FBlackPiecesTook, length(FBlackPiecesTook) + 1);

end;

end;

end;

{ TSaveManager }

constructor TSaveManager.Create(AOwner: TObject);

begin

LinkedBoard := nil;

rootdir := '';

end;

procedure TSaveManager.LoadFromFile(filepath: string);

var

tS : TextFile;

x: Integer;

y: Integer;

i: Integer;

s: string;

t1, t2 : integer;

PGNPath : string;

begin

PGNPath := filepath;

delete(PGNPath, pos('.', PGNPath), 6);

PGNPath := PGNPath + '.PGN';

if FileExists(PGNPath) then

assignFile(tS, filepath);

reset(tS);

readln(tS, s);

with LinkedBoard do

begin

Clear;

drawboard;

t1 := GetTickCount;

turn := strtoint(s);

for y := 1 to 8 do

begin

readln(tS, s);

for x := 1 to 8 do

begin

SetSquareTo(Point(x,y),strtoint(copy(s,1,2)));

delete(s, 1, 2);

end;

end;

readln(tS, s);

SetLength(FWhitePiecesTook, length(S));

for i := 0 to length(s) - 1 do

begin

WhitePiecesTook[i] := strtoint(copy(s, 1, 1));

delete(s, 1, 1);

end;

readln(tS, s);

SetLength(FBlackPiecesTook, length(S));

for i := 0 to length(s) - 1 do

begin

BlackPiecesTook[i] := strtoint(copy(s, 1, 1));

delete(s, 1, 1);

end;

readln(ts, s);

if s = 'TRUE' then

CastlingPossible[1] := true

else

CastlingPossible[1] := false;

readln(ts, s);

if s = 'TRUE' then

CastlingPossible[2] := true

else

CastlingPossible[2] := false;

readln(tS, s);

PlayerNameWhite := s;

readln(tS, s);

PlayerNameBlack := s;

end;

closefile(ts);

t2 := GetTickCount;

end;

procedure TSaveManager.SaveToFile(filepath: string);

var

tS : TextFile;

x: Integer;

y: Integer;

s: Integer;

pgnpath : string;

begin

PGNPath := filepath;

delete(PGNPath, pos('.', PGNPath), 6);

PGNPath := PGNPath + '.PGN';

assignfile(tS, PGNPath);

rewrite(ts);

closefile(ts);

assignFile(tS, filepath);

rewrite(tS);

with LinkedBoard do

begin

if selected then

begin

selected := false;

if Turn = 1 then

turn := 2

else

turn := 1;

end;

writeln(tS, turn);

for y := 1 to 8 do

begin

for x := 1 to 8 do

begin

if Board[x, y].Kind >= 0 then

write(tS, FormatFloat('00', Board[x, y].Kind))

else

write(tS, Board[x, y].Kind);

end;

write(tS, #13#10);

end;

for s := 0 to getWhiteTookLength do

write(tS ,WhitePiecesTook[s]);

write(tS, #13#10);

for s := 0 to getBlackTookLength do

write(tS, BlackPiecesTook[s]);

write(tS, #13#10);

writeln(ts, CastlingPossible[1]);

writeln(ts, CastlingPossible[2]);

writeln(tS, PlayerNameWhite);

writeln(tS, PlayerNameBlack);

end;

closefile(tS);

assignFile(tS, rootDir + '\\_LOG.DWCS');

if not fileExists(rootDir + '\\_LOG.DWCS') then

rewrite(tS);

Append(tS);

writeLn(tS, filepath);

closefile(tS);

end;

procedure TSaveManager.SaveToFileOverwrite(filepath: string);

var

tS : TextFile;

x: Integer;

y: Integer;

s: Integer;

pgnpath : string;

begin

PGNPath := filepath;

delete(PGNPath, pos('.', PGNPath), 6);

PGNPath := PGNPath + '.PGN';

assignfile(tS, PGNPath);

rewrite(ts);

closefile(ts);

assignFile(tS, filepath);

rewrite(tS);

with LinkedBoard do

begin

if selected then

begin

selected := false;

if Turn = 1 then

turn := 2

else

turn := 1;

end;

writeln(tS, turn);

for y := 1 to 8 do

begin

for x := 1 to 8 do

begin

if Board[x, y].Kind >= 0 then

write(tS, FormatFloat('00', Board[x, y].Kind))

else

write(tS, Board[x, y].Kind);

end;

write(tS, #13#10);

end;

for s := 0 to getWhiteTookLength do

write(tS ,WhitePiecesTook[s]);

write(tS, #13#10);

for s := 0 to getBlackTookLength do

write(tS, BlackPiecesTook[s]);

write(tS, #13#10);

writeln(ts, CastlingPossible[1]);

writeln(ts, CastlingPossible[2]);

writeln(tS, PlayerNameWhite);

writeln(tS, PlayerNameBlack);

end;

closefile(tS);

end;

procedure TSaveManager.SetLinkedBoard(const Value: TBoardMannager);

begin

FLinkedBoard := Value;

end;

procedure TSaveManager.SetrootDir(const Value: string);

begin

if not DirectoryExists(value) then

CreateDir(value);

FrootDir := Value;

end;

end.