Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего профессионального образования

Тамбовский государственный технический университет

Кафедра

Пояснительная записка

к курсовой работе по дисциплине «ООП»   
на тему: Разработка игры «Точки»

Выполнил студент гр.

.

Проверил:.

Тамбов 20

Содержание

1. Введение………………………………………………………………………………….…...3

2. Основной раздел…………………..……………………………………………………….....4

2.1. Обзор источников и аналогов…………………………………………..………….4

2.2. Анализ предметной области………………….…………………………….……...4

2.3. Объектная модель …….………………………………………………….…………5

2.4. Проектирование программного продукта……………………………………….. 6

2.5. Тестирование программного продукта………………………………………..….6

3. Заключение……………………………….…………………………………………………...7

4. Приложения……………………………………………..……………………………….……7

4.1. Листинг программы………………...………………………………………...……..8

1. **Введение**

Программы, написанные объектно-ориентированно имеют множество плюсов. Они разделяются на разные блоки, которые пишутся отдельно и выглядят как маленькие модели, в последствие соединенные вместе. Так же, если программа (игра) представляют собой какой-либо глобальный проект, над его реализацией трудится не один человек, а целая компания. Каждый пишет свой отдельный объект, потом они соединяются, это обеспечивает работу удаленного доступа и экономию времени.

ООП — это и ОО программирование и проектирование. Создано ООП для проектирования/программирования программных продуктов. Не для моделирования процессов. Не для проектирования протоколов, а именно для программных продуктов, для их реализации. Первое что нужно делать — это начать использовать объектное мышление, а это самая большая проблема ООП, научиться мыслить объектно очень сложно.

Цель курсовой работы:

-углубить и расширить теоретические знания по дисциплине «Объектно-ориентированное программирование»;

-формировать навыки исследовательской работы: определение объекта и предмета исследования, постановка целей и задач, умение анализировать, обобщать, делать выводы;

- научиться применять полученные теоретические знания для решения

практических задач.

Задачи курсовой работы:

а). Изучить теоретический материал по теме работы и особенности применения на практике полученных теоретических знаний по дисциплине «Объектно-ориентированное программирование».

б). Решить прикладную задачу из конкретной предметной области, создав программный продукт в одной из объектно-ориентированных инструментальных сред.

Задание на курсовую работу:

Разработать логическую игру «Точки»

Данное приложение является реализацией известной логической игры. В данной

реализации предусматривается игра двух игроков на квадратном поле фиксированного размера. Цель игры. заполнить максимальное количество клеток игрового поля своими фишками. Игроки выполняют ходы по очереди. Ход игрока заключается в произвольной установке линии на границе двух ячеек, причем, если какая-либо ячейка оказывается со всех четырех сторон обрамлена линиями, то она отмечается фишкой данного игрока, а самому игроку предоставляется дополнительный ход, и т.д. Игра заканчивается, когда все

игровое поле оказывается заполнено фишками. Выигрывает тот игрок, чьих фишек на поле больше на момент окончания игры.

Приложение должно обеспечивать начало новой игры на чистом поле, а также

проверку соответствия действий игроков правилам игры и условия окончания игры. Роль одного из игроков (по выбору пользователя) может выполнять компьютер.

1. **Основной раздел**
   1. **Обзор источников и аналогов**

«То́чки» (реже «города», «фазенда», «феодалы») — [игра на клетчатой бумаге](http://ru.wikipedia.org/wiki/%D0%98%D0%B3%D1%80%D1%8B_%D0%BD%D0%B0_%D0%B1%D1%83%D0%BC%D0%B0%D0%B3%D0%B5), как правило, для двух человек.

[Игра](http://ru.wikipedia.org/wiki/%D0%98%D0%B3%D1%80%D0%B0) «точки» возникла в начале 80-х годов ХХ века из-за неправильной трактовки правил игры «[го](http://ru.wikipedia.org/wiki/%D0%93%D0%BE)». Для новой игры чётких правил не создали, и в [Европе](http://ru.wikipedia.org/wiki/%D0%95%D0%B2%D1%80%D0%BE%D0%BF%D0%B0) появилось большое количество вариаций «точек» - разные виды окружения, последовательность ходов, размеры поля. Наиболее устоялись нынешние правила – стандартные и польский вариант ([игра](http://ru.wikipedia.org/wiki/%D0%98%D0%B3%D1%80%D0%B0) на территорию).

В 2006 г. в [Крыму](http://ru.wikipedia.org/wiki/%D0%9A%D1%80%D1%8B%D0%BC) был проведен первый в истории турнир (оффлайн) по точкам, в котором приняло участие 6 игроков, и победил Николай Ермак. Спустя год был организован первый международный онлайн турнир Need2Lead Points Open Cup 2007. В нём приняло участие 32 игрока из России, Украины и Германии, но доигран он не был. Однако этот турнир послужил примером и стал начальным [этапом](http://ru.wikipedia.org/wiki/%D0%AD%D1%82%D0%B0%D0%BF) в проведении различных, более успешных онлайн турниров.

26 декабря 2010 года Иван Гейко создал первую в истории команду - сборную Днепропетровска. А уже зимой 2011 года состоялся первый командный турнир, в котором приняло участие 5 команд и более 25 игроков. Сейчас точки являются достаточно популярной игрой, в основном среди школьников и студентов, и быстро развиваются (создаются новые программы для игры, сайты, проводятся турниры и т. д.). Создано множество различных онлайн точек.

* 1. **Анализ предметной области**

На поле играют два игрока - синий и красный. Ходят по очереди, ставя точку на пересечении линий.

За один ход ставится одна точка и она больше не передвигается. Необходимо окружать точки соперника, не давая окружать собственные. Побеждает игрок, захвативший больше точек.

Игра идет на поле различного размером 39х32 пунктов (основной размер поля, используется на турнирах). Также на zagram.org доступны поля меньшего размера: 20х20 25х25 30х30 пунктов. Пункт - пересечение линий на поле.

Игроки могут ходить в свободные пункты, не занятые точками или не входящие в окружение. Возможности пропускать ход нет.

При создании непрерывной (по вертикали, горизонтали, диагонали) замкнутой линии образуется область. Если внутри нее есть точки противника (при этом могут быть пункты, не занятые чьими-либо точками), то это считается областью окружения. Если точек соперника нет, то область свободная и в нее можно ставить точки. Точки, попавшие в область окружения, далее не участвуют в образовании линий для окружения. Точки, поставленные на краю поля, не окружаются.

Если есть замкнутая область, в которой нет точек соперника, то точка соперника будет окружена автоматически, но только после вашего хода. При этом ход может быть в любое другое место на поле (не обязательно входить в состав той окружающей области). Но только если точка противника своим ходом не окружает ваши точки.

В начале игры на поле есть несколько точек, называемых стартовой позицией. Также

есть правила "на территорию" и "стандартные", которые предусматривают пустое поле

в начале игры без стартовой позиции и включают игру на территорию, распространенную среди польских игроков.

Партия заканчивается в одном из случаев:

- когда не осталось свободных мест на поле;

- по взаимному согласию игроков (ничья или один из них сдается);

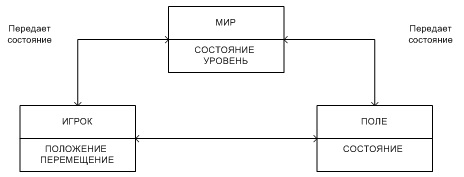
- один из игроков нажимает СТОП, когда он заземляет (их уже нельзя обвести) свои точки;

- у одного из игроков кончается время и он проигрывает.

Среди польских игроков преобладают правила игры на территорию. В конце игры

подсчитывается не только количество захваченных точек, а и захваченная территория.

* 1. **Объектная модель**



* 1. **Проектирование программного продукта**

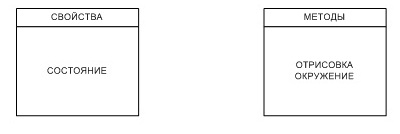
Мир



Игрок



Поле



* 1. **Тестирование программного продукта**

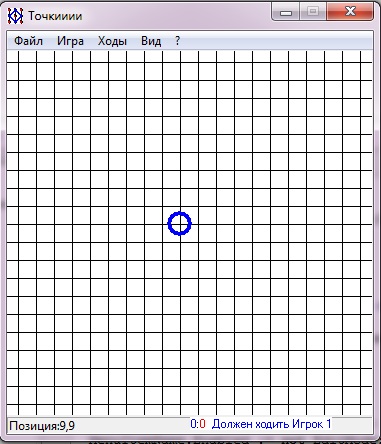


Рис 1. Начало игры

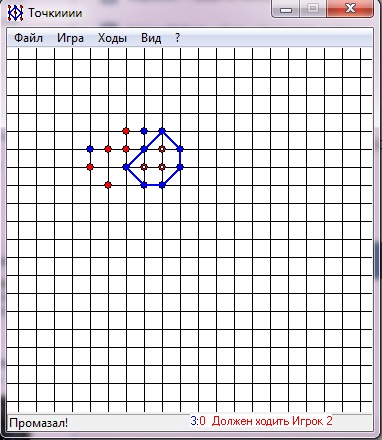


Рис 2. Активный режим игры

1. **Заключение**

Объектно-ориентированные подходы созданы для эффективности и гибкости программ.

Используя объектное мышление, легко можно проектировать сложные системы, оперируя объектами и взаимодействием между ними. Т.е. ООП без объектного мышления не позволит начать использовать всю силу и мощь ООП.

Одной из целей курсовой было, как раз научиться использовать объектное мышление. Так же создание отдельных моделей позволяет: уменьшить сложности, повысить надежность программного обеспечения, обеспечить возможности моделирования и повторное использование отдельных объектов.

1. **Список литературы:** 
   1. Зачем нам ООП и что это такое. Хабрахабр URL: <http://habrahabr.ru/post/148015/> (дата обращения 20.03.2014)
   2. Я не знаю ООП. Хабрахабр URL: <http://habrahabr.ru/post/147927/> (дата обращения 18.02.2014)
   3. Приймак А. Учебник по игре «точки»: онлайн-версия, издание 1-е, февраль 2013, 37с.
2. **Приложение (**Листинг)

unit GamePoint;

//{$define debug}

{$I SVV.INC}

interface

uses

Windows, Messages, SysUtils, Graphics, Forms, ExtCtrls, Dialogs, Menus,

Controls, Classes, ComCtrls, newgamedialog, shellapi, PointUtils, math;

type

TGamePoint = type windows.TCoord; // Координаты точки доски

TGameRect = type windows.TSmallRect; // Ограничивающий прямоугольник для каких-то операций

TPlayerInfo = packed record

Name: string; // Имя игрока

Color: TColor; // Цвет визуального представления точки

Count: Integer; // Число поставленных игроком точек

Gain: Integer; // Число захваченных игроком точек

end;

TPlayersInfo = array[1..2] of TPlayerInfo; // настроийки профилей игроков

TMoveInfo = packed record // исчерпывающая информация об одном конкретном ходе

Point: TGamePoint;

BoardValue: Byte;

end;

TMoveInfoEx = packed record // исчерпывающая информация об одном конкретном ходе, оптимизированная для быстрого отката назад при просчете ходов компьютером

Point: TGamePoint;

Who: Word;

Gain: array[0..1] of Integer;

Count: array[0..1] of Integer;

NumberOfChanges: Integer;

BoardChangesList: array of TMoveInfo;

end;

TGameBoard = array of array of Byte; // собственно доска

TGameStack = array of TGamePoint; // определение вспомогательной структуры, используемой для поиска замкнутых контуров и в алгоритмах заливки(захвата крепостей)

TRollBackStructure = array of TMoveInfoEx; // позволяет делать откаты назад и вперед

TMainForm = class(TForm)

StatusBar: TStatusBar;

BoardView: TImage;

MainMenu: TMainMenu;

MenuGame: TMenuItem;

MenuHelp: TMenuItem;

MenuView: TMenuItem;

MenuMove: TMenuItem;

MenuItemNewGame: TMenuItem;

MenuItemAbout: TMenuItem;

MenuItemHelp: TMenuItem;

MenuItemEditing: TMenuItem;

MenuItemMoveForward: TMenuItem;

MenuItemMoveBack: TMenuItem;

MenuFile: TMenuItem;

MenuItemSaveFile: TMenuItem;

MenuItemLoadFile: TMenuItem;

MenuItemExit: TMenuItem;

SaveDialog: TSaveDialog;

OpenDialog: TOpenDialog;

MenuItemName: TMenuItem;

MenuItemVsComp: TMenuItem;

MenuItemGameLevel: TMenuItem;

MenuItemFileSeparator: TMenuItem;

MenuItemGameSeparator: TMenuItem;

MenuItemShowMove: TMenuItem;

MenuItemMakeMove: TMenuItem;

MenuItemIncWindow: TMenuItem;

MenuItemDecWindow: TMenuItem;

MenuItemMoveSeparator1: TMenuItem;

MenuItemMoveSeparator2: TMenuItem;

MenuItemMoveHalfBack: TMenuItem;

MenuItemMoveHalfForward: TMenuItem;

N1: TMenuItem;

miMarkPoint: TMenuItem;

miMarkPoint2: TMenuItem;

N2: TMenuItem;

miShowCursor: TMenuItem;

miHighlight: TMenuItem;

procedure FormCreate(Sender: TObject);

procedure BoardViewMouseDown(Sender: TObject; Button: TMouseButton;

Shift: TShiftState; X, Y: Integer);

procedure BoardViewMouseMove(Sender: TObject; Shift: TShiftState; X,

Y: Integer);

procedure FormClose(Sender: TObject; var Action: TCloseAction);

procedure CancelFormClose(Sender: TObject; var Action: TCloseAction);

procedure MenuItemNewGameClick(Sender: TObject);

procedure MenuItemAboutClick(Sender: TObject);

procedure MenuItemHelpClick(Sender: TObject);

procedure MenuItemEditingClick(Sender: TObject);

procedure MenuItemExitClick(Sender: TObject);

procedure MenuItemSaveFileClick(Sender: TObject);

procedure MenuItemLoadFileClick(Sender: TObject);

procedure MenuItemNameClick(Sender: TObject);

procedure MenuItemVsCompClick(Sender: TObject);

procedure MenuItemGameLevelClick(Sender: TObject);

procedure MenuItemStopClick(Sender: TObject);

procedure MenuItemMoveBackClick(Sender: TObject);

procedure MenuItemMoveForwardClick(Sender: TObject);

procedure MenuItemShowMoveClick(Sender: TObject);

procedure MenuItemMakeMoveClick(Sender: TObject);

procedure MenuItemIncWindowClick(Sender: TObject);

procedure MenuItemDecWindowClick(Sender: TObject);

procedure MenuItemMoveHalfBackClick(Sender: TObject);

procedure MenuItemMoveHalfForwardClick(Sender: TObject);

procedure FormDestroy(Sender: TObject);

procedure SetCursorPos(const CurPoint: TGamePoint);

function GetCursorRGN(const Point: TGamePoint): HRGN;

procedure HideCursor(const Point: TGamePoint);

procedure ShowCursor(const Point: TGamePoint; const Color: TColor);

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

Shift: TShiftState);

procedure PaintBoardView(var WMPaint: TWMPaint); message WM\_PAINT;

procedure miMarkPointClick(Sender: TObject);

procedure miMarkPoint2Click(Sender: TObject);

procedure miShowCursorClick(Sender: TObject);

procedure SetKbKursorShowState(const Hide: Boolean);

procedure SetHighlight(const NotHihlight: Boolean);

procedure miHighlightClick(Sender: TObject);

private

fCursorPos: TGamePoint;

bHideCursor: Boolean;

fNotHihlightLastMove: Boolean;

published

property CursorPos: TGamePoint read fCursorPos write SetCursorPos;

property HideKbCursor: Boolean read bHideCursor write SetKbKursorShowState;

property NotHighlightLastMove: Boolean read fNotHihlightLastMove write SetHighlight;

private

{Фундаментальные определения}

Board: TGameBoard;

// собственно описание доски - квадратного поля размера (BoardSize+1) X (BoardSize+1)

// каждая клетка может быть точкой какого-то цвета, свободна или захвачена

BoardSize: Integer;

// максимальный индекс по обоим измерениям доски.Доска в данной реализации

// обязательно квадратная, хотя просто сделать и прямоугольную

// Индексация начинается с нуля, так что размер доски и теоретически

// максимально возможное число поставленных точек - это [(BoardSize+1) \* (BoardSize+1)]

ActivePlayer: Integer;

// номер текущего игрока - ActivePlayer=1..2

// Равен занчению , которе имеет клетка доски с поставленной свободной точкой

// данного игрока. Именно поэтому ActivePlayer=1..2 , а не 0..1 , так как значение

// 0 зарезервировано для пустых свободных клеток доски

// Формула перехода к следующему игроку: CurPlayer:=NextPlayer[CurPlayer]

CurGameState: TPlayersInfo;

//состояние игры(число точек, поставленных игроками, счет , текущие настройки

// профилей игроков).СОбственно описание доски находится в массиве Board.

PGamSeq: TRollBackStructure;

// Двусвязный список, описывающий последовательность ходов

// Служит для возможности возврата назад при итерации с конца

// и для возможности перерисовки поля при итерации сначала

// Также используется при просчете ходов для быстрого отката назад

CurrentSeq: Integer;

// Текущая позиция в двусвязном списке PGamSeq

SeqCount: Integer;

// Всего позиций в в двусвязном списке PGamSeq

// Если SeqCount > CurrentSeq , то это значит , что происходил

// откат назад и возможен откат вперед

{Неиспользуемые переменные}

Vassal: Boolean;

// Изменение порядка ведения счета

// , то есть считать многократно окруженные точки захваченными игроком,

// выстроившим первое или последнеее кольцо окружения

// В разных местах принято играть по-разному.

// Я первоначально реализовывал так, как привык сам(т.е. когда Vassal=False).

savedTitle : String;

{Переменные, задающие режим работы}

EditMode: Boolean;

// Устанавливается в True , если текущий режим - режим редактирования позиций

// В режиме редактирования позиций очередность ходов не задана и компьютер не будет ходить автоматически

ComputerGame: Boolean;

// True, если компьютер должен автоматически ходить после хода игрока

GameLevel: Integer;

// Уровень игры компьютера. Чем больше это число, тем сильнее должна играть программы и тем дольше она будет обдумывать ход

// Значение 0 должно приблизительно соотв. почти полному отсутвию просчета ходов(т.е. ход GameLevel=0 должен делаться в основном на основе функции оценки позиции без минимакса)

// В текущей реализации это фактически глубина поиска по дереву. В общем случае при изменении GameLеvel может происходить не только изменение глубины просчета, но и подключение каких-то продвинутых дополнительных алгоритмов.

{Интерфейсные определения}

PixelGridStep: Integer;

// шаг сетки на BoardView - расстояние между соседними линиями сетки на BoardView в пикселях

PixelIndent: Integer;

// отступ от края доски до первой линии сетки на BoardView в пикселях

PixelTolerance: Integer;

//смещение , на которе может удалиться указатель мыши от точки, чтобы нажатие

// на кнопку мыши над BoardView воспринималось как попытка сделать ход, а не как промах мимо точеи пересечения линий доски на BoardView

PixelPointRadius: Integer;

// Радиус точки в пикселях

{Вспомогательные переменные}

LastPrize: Integer;

// Выигрыш игрока в захваченных точках противника за последний совершенный ход

LastEmptySquare: Integer;

// Выигрыш игрока в захваченных свободных точках(площади) за последний совершенный ход

UserBreak: Boolean;

{фундаментальные процедуры игровой логики, напрямую взаимодействующие с интерфейсом}

ComputerTHinking: Boolean;

procedure PrepareGame(SizeOfBoard: Integer);

// подготовка к игре c размером поля (SizeOfBoard+1)

procedure SynchronizeMenuItems();

// Синхронизирует доступность пунктов меню с внутренними переменными

procedure EndGame();

//освобождает все ресуры, связанные с игрой

function TrySetPoint(const NewPoint: TGamePoint; const Player: Byte; const DrawPoint: Boolean): Boolean;

// пытается поставить точку.

// Возвращает True, если это удалось и False, если не удалось

procedure RollBack(const Draw: Boolean);

// откат на один полуход.Если откатываться некуда , то ничего не происходит

procedure RollForward(const Draw: Boolean);

// отказ от отката на один полуход

procedure DrawGrid(); // перерисовка сетки

procedure DrawBoard(); // перерисовка доски целиком

procedure CorrectSize();

// перерисовывает все поле целиком при изменении размера BoardView

procedure KillPoint(const DeadPoint: TGamePoint);

// визуально помечает точку как убитую

procedure DrawNewPoint(const NewPoint: TGamePoint; const Player: Integer);

// отрисовка точки на доске

procedure DrawFortress(const Ring: TGameStack; const Ring\_Depth: Integer; const Player: Byte);

// отрисовка кольца окружения

procedure IndicateGain();

// прорисовка строки статуса

procedure IndicateProgress(const MaxValue, CurValue: Integer);

// прорисовка строки статуса во время хода компьютера

function SaveLoadGameFile(const filename: string; const Load: Boolean): Boolean;

{Вспомогательные процедуры для игровой логики}

function NumberNearGroups(const CheckedPoint: TGamePoint; const Player: Byte): Integer;

// связность точки относительно точек своего же цвета

function NumberNearPoints(const CheckedPoint: TGamePoint; const PLayer: Byte): Integer;

// число соседних точек того же цвета

function IsOnBoard(const TestedPoint: TGamePoint): Boolean; overload;

function IsOnBoard(const x, y: SmallInt): Boolean; overload;

// True , если координаты точки попадают на доску

function MakePoint(const x, y: SmallInt): TGamePoint;

// просто упаковка двух целых в структуру TGamePoint

function IsNear(const P1, P2: TGamePoint): Boolean;

// True, если точки P1, P2: соседи

function IsNearOut(const CheckedPoint: TGamePoint): Boolean;

// True , если CheckedPoint лежит на границе поля

function IsPointInsideRing(

const Point: TGamePoint;

const Ring: TGameStack;

const RingLength: Integer): Boolean;

// True , если данная точка находится внутри данного кольца

function IsEndGame(): Boolean;

// True , если игра закончилась

function CounterAttack(const NewPoint: TGamePoint; const Player: Byte; const Draw: Boolean): Boolean;

// проверка на то, что точка поставлена внутрь чужой пустой крепости

function Attack(const NewPoint: TGamePoint; const Player: Byte; const Draw: Boolean): Boolean;

function NextPoint(const Pred: TGamePoint; const SearchFor, Mask: Byte): TGamePoint;

function CheckNonEmptyFortress(const StackBounds: TGameRect;

const Player: Byte; const Ring: TGameStack; const Ring\_Depth: Integer;

const Draw: Boolean): Boolean;

procedure GetRing(NewPoint: TGamePoint; Player: Byte; var Ring: TGameStack; var Ring\_Depth: Integer; const Mask: Byte);

function DoStep(NewPoint: TGamePoint; const Player: Byte; const DrawPoint: Boolean): Boolean;

procedure EnterChange(const NewPoint: TGamePOint; const Mask: Byte);

procedure SetDefaultBounds(var bounds: TGameRect);

procedure SetNewBounds(var bounds: TGameRect; const NewValues: TGameStack; CountV: Word);

function CaptureFortress(const NewPOint: TGamePOint; const Player: Byte; const Draw: Boolean): Boolean;

procedure CorrectStackBounds(const NewPoint: TGamePoint; var Bounds: TGameRect);

procedure PushNextPoints(var p: TGameStack; var p\_len: Integer; const NextPoint: TGamePoint;

const Player: Byte; const Mask: Byte; var Changes: TGameStack; var Changes\_Depth: Integer);

procedure ResetStack(const Stack: TGameStack; const STack\_Depth: Integer; const Mask: Byte);

procedure ReduceEmptyCycles(const Ring1: TGameStack; const Ring1\_Depth: Integer;

var Ring2: TGameStack; var Ring2\_Depth: Integer);

{функции, обслуживающие алгоритм хода компа}

procedure ComputerMove(var BestPoint: TGamePoint);

// компьютер ходит

function GetBestPosition(var BestPoint: TGamePoint; const BestPoints: TGameStack; const Len\_BestPoints: Integer): Integer;

//возвращает лучшую позицию без просчета ходов

function PositionEstimate(const TestedPoint: TGamePoint): Integer;

// оценка позиции без просчета ходов

function GetMinMaxEstimate(SearchDepth: Integer; const TestedPoint: TGamePoint; const p: Pointer): Integer;

function GetBestPoints(const SearchDepth: Integer; var BestPoints: TGameStack; var Len\_BestPoints: Integer; const p: Pointer): Integer;

// возвращает лучшую позицию с просчетом на глубину SearchDepth полуходов

procedure InitDonePerspectivePoints(var p: Pointer; const WhatToDo: Boolean);

function IsPointPerspective(const SearchDepth: Integer; const p: Pointer; const x, y: integer): Boolean;

{Вспомогательные интерфейсные процедуры для связи координат точек на доске

и их координат на ее изображении}

function BoardViewSpaceToGameSpace(const X: Integer): Integer;

function GameSpaceToBoardViewSpace(const X: Integer): Integer;

function BoardViewSpaceMayTranslateToGameSpace(const X: Integer): Boolean;

function BoardViewCoordToGamePoint(const X, Y: Integer): TGamePoint;

function TryBoardViewCoordToGamePoint(const X, Y: Integer; var GamePoint: TGamePoint): Boolean;

{Вспомогательные интерфейсные процедуры}

procedure DrawCircle(const X, Y, R: Integer; const Color: Integer = -1);

end;

var

MainForm: TMainForm;

implementation

{$R \*.dfm}

resourcestring

AboutText =

' ИГРА ТОЧКИ '#13#10 +

' Автор '#13#10 +

' Курстовая работа по ООП ';

GameOverText = ' Игра окончена ';

MoveBesideText = 'Промазал! ';

MoveOutText = 'Аут! ';

PositionText = 'Позиция:';

HelpFileNotFoundText = 'Файл справки не найден:';

GameLevelText = 'Уровень';

ParamCountErrorText = 'Допустим только один параметр' + #10#13 +

' - имя файла с описанием позиции';

GameFileNotFoundText = 'Файл с описанием позиции не найден ';

FirstPlayerDefNameText = 'Игрок 1';

SecondPlayerDefNameText = 'Игрок 2';

MustMoveText = 'Должен ходить ';

EditModeText = 'Редак.-ходит любой ';

LoadErrorText = 'Неверный формат файла ';

SaveErrorText = 'Ошибка при попытке сохранить игру в файле';

PlayerChangeNameCaption = 'Смена имени игрока ';

PlayerChangeNameText = 'Введите новое имя ';

GameLevelCaption = ' Выбор уровня сложности ';

GameLevelChangeText = 'Введите уровень сложности (от нуля - самый простой)';

ProgressText = 'Прогресс';

PositionNotFreeText = 'Данное поле доски уже занято';

MustMoveAnotherPlayer = 'должен ходить другой игрок';

OrText = ' или ';

UserBreakText = 'Ход не сделан';

ComputerThinkText = 'Идет просчет хода.';

const

debugconst = 2

const

{Битовые маски для определения состояния клеток поля}

brFree = $00; {0000.0000b} // точка свободна

brFirst = $01; {0000.0001b} // точка принадлежит первому игроку

brSecond = $02; {0000.0010b} // точка принадлежит второму игроку

brDead = $04; {0000.0100b} // точка мертва(находится в окружении)

{Битовые маски для пометки обработанных клеток поля в алгоритмах заливки

и поиска замкнутых контуров}

brCheck = $08; {0000.1000b} // эти биты используются

brInside = $10; {0001.0000b} // для пометки обработанных точек доски

brCheck2 = $20; {0010.0000b} // в алгоритмах заливки и

brCheck3 = $40; {0100.0000b} // поисков замкнутых контуров

brCheck4 = $80; {1000.0000b} // и эти тоже там же

const

{Формула перехода от текущего к следующему игроку: CurPlayer:=NextPlayer[CurPlayer]}

NextPlayer: array[0..2] of Byte = (0, brsecond, brfirst);

const

MinGameLevel = 0;

// принятие решения о ходе на основе функции оценки позиции

MaxGameLevel = 4;

// просчет на 4 полухода , т.е. два полных хода

MinBoardSize = 7;

MaxBoardSize = 31;

MinPixelGridStep = 20;

IncPixelStep = 20;

{Обработчики главных событий - взаимодействие пользователя с доской}

procedure TMainForm.BoardViewMouseMove(Sender: TObject; Shift: TShiftState; X,

Y: Integer);

{Показывает координаты в строке статуса}

var

GamePoint: TGamePoint;

begin

if not TryBoardViewCoordToGamePoint(x, y, GamePoint) then

begin

StatusBar.SimpleText := MoveBesideText;

BoardView.Cursor := crArrow;

end

else begin

BoardView.Cursor := crHandPoint;

StatusBar.SimpleText := PositionText + IntToStr(GamePoint.Y) + ',' + IntToStr(GamePoint.X);

{$IFDEF debug}

if IsOnBoard(GamePoint) then

StatusBar.SimpleText := StatusBar.SimpleText

+ ' Ст:' + IntToHex(Board[GamePoint.X, GamePoint.Y], 2) + 'H (' + ByteToBin(Board[GamePoint.X, GamePoint.Y]) + 'b)';

{$ENDIF}

end;

IndicateGain();

//ToDO:не перерисовывать это постоянно.Использовать прямо BoardView и функции GDI для индикации. Тогда статус бар вообще не нужен.

end;

procedure TMainForm.BoardViewMouseDown(Sender: TObject; Button: TMouseButton;

Shift: TShiftState; X, Y: Integer);

var

NewPoint: TGamePoint;

begin

if not TryBoardViewCoordToGamePoint(x, y, NewPoint) then

begin

StatusBar.SimpleText := MoveBesideText;

Exit;

end;

if not IsOnBoard(NewPoint) then

begin

StatusBar.SimpleText := MoveOutText;

Exit;

end;

if not IsEndGame() then

begin

assert((Byte(mbLeft) + 1 = brFirst) and (Byte(mbRight) + 1 = brSecond),

'Программа несовместима с данной версией модуля Controls.pas');

if DoStep(NewPoint, Byte(Button) + 1, True) then

begin

if IsEndGame() then

ShowMessage(GameOverText)

else if ComputerGame then

Self.MenuItemMakeMoveClick(Self);

SynchronizeMenuItems();

end

else

StatusBar.SimpleText := PositionNotFreeText + OrText + MustMoveAnotherPlayer;

end;

end;

{Обработчики начала и завершения работы программы}

procedure TMainForm.FormClose(Sender: TObject; var Action: TCloseAction);

begin

Action := caFree;

end;

procedure TMainForm.CancelFormClose(Sender: TObject; var Action: TCloseAction);

begin

UserBreak := True;

Action := caNone;

end;

procedure TMainForm.FormCreate(Sender: TObject);

begin

Randomize();

Self.DoubleBuffered := True; // Попытка избавиться от мерцания при ходе.

// Оно возникает при изменении активности(Свойства Enabled) пунктов меню

// TODO: загрузка нижележащих значений из реестра или ini-файла

CurGameState[brFirst].Name := FirstPlayerDefNameText;

CurGameState[brSecond].Name := SecondPlayerDefNameText;

CurGameState[brFirst].Color := clBlue;

CurGameState[brSecond].Color := clRed;

Vassal := False;

ComputerGame := True;

EditMode := False;

PrepareGame((MinBoardSize + MaxBoardSize) div 2);

if ParamCount() > 1 then

ShowMessage(ParamCountErrorText)

else

if (ParamCount() = 1) then

if FileExists(ParamStr(1)) then

begin

if not SaveLoadGameFile(ParamStr(1), True) then

ShowMessage(LoadErrorText + ':' + ParamStr(1));

end

else

ShowMessage(GameFileNotFoundText + ParamStr(1));

SynchronizeMenuItems();

end;

procedure TMainForm.FormDestroy(Sender: TObject);

begin

EndGame();

end;

{Обработчики событий нажатия на клавиши}

{Меню "Файл"}

procedure TMainForm.MenuItemSaveFileClick(Sender: TObject);

begin

if not SaveDialog.Execute then Exit;

if not SaveLoadGameFile(SaveDialog.FileName, False) then

ShowMessage(SaveErrorText);

SynchronizeMenuItems();

end;

procedure TMainForm.MenuItemLoadFileClick(Sender: TObject);

begin

if not OpenDialog.Execute then Exit;

if not SaveLoadGameFile(OpenDialog.FileName, True) then

ShowMessage(LoadErrorText);

SynchronizeMenuItems();

end;

procedure TMainForm.MenuItemExitClick(Sender: TObject);

begin

Self.Close();

end;

{Меню "Игра"}

procedure TMainForm.MenuItemShowMoveClick(Sender: TObject);

var

Colors: array[0..1] of Integer;

LastPoint: TGamePoint;

LastPlayer: Integer;

PCurColor: PColor;

I: Integer;

begin

if CurrentSeq < 0 then Exit;

// TODO:поставить таймер или запускать это отдельным потоком,

// чтобы программа могла реагировать на события в это время

LastPoint := PGamSeq[CurrentSeq].Point;

LastPlayer := PGamSeq[CurrentSeq].Who;

PCurColor := @CurGameState[LastPlayer].Color;

Colors[0] := PCurColor^;

Colors[1] := clYellow;

for i := 1 to 8 do // необходимо заканчивать при четном i

begin

PCurColor^ := Colors[i and $01];

DrawNewPoint(LastPoint, LastPlayer);

BoardView.Refresh;

Sleep(200);

end;

if (Board[LastPoint.X, LastPoint.Y] and brDead) <> 0 then KillPoint(LastPoint);

end;

procedure TMainForm.MenuItemNewGameClick(Sender: TObject);

begin

NewGameDlg := TNewGameDlg.Create(Self);

try

if NewGameDlg.ShowDialog(BoardSize, GameLevel, Vassal,

MinGameLevel, MaxGameLevel, MinBoardSize, MaxBoardSize) then

begin

EndGame();

PrepareGame(BoardSize - 1);

end;

finally

NewGameDlg.Free;

end;

SynchronizeMenuItems();

end;

procedure TMainForm.MenuItemMoveForwardClick(Sender: TObject);

begin

RollForward(True);

if ComputerGame then RollForward(True);

SynchronizeMenuItems();

end;

procedure TMainForm.MenuItemEditingClick(Sender: TObject);

begin

EditMode := not EditMode;

if EditMode then ComputerGame := False;

SynchronizeMenuItems();

end;

procedure TMainForm.MenuItemVsCompClick(Sender: TObject);

begin

ComputerGame := not ComputerGame;

if ComputerGame then EditMode := False;

SynchronizeMenuItems();

end;

procedure TMainForm.MenuItemGameLevelClick(Sender: TObject);

var

NewGameLevel: Integer;

begin

while not TryStrToInt(InputBox(GameLevelCaption, GameLevelChangeText +

'(' + IntToStr(MinGameLevel) + '..' + IntToStr(MaxGameLevel) + ')', IntToStr(GameLevel)),

NewGameLevel) or not ((NewGameLevel >= MinGameLevel) and (NewGameLevel <= MaxGameLevel)) do ;

GameLevel := NewGameLevel;

SynchronizeMenuItems();

end;

procedure TMainForm.MenuItemNameClick(Sender: TObject);

begin

if not EditMode then

begin

CurGameState[ActivePlayer].Name := InputBox(

PlayerChangeNameCaption,

PlayerChangeNameText + CurGameState[ActivePlayer].Name,

CurGameState[ActivePlayer].Name);

SynchronizeMenuItems();

end;

end;

{Меню "Ходы"}

procedure TMainForm.MenuItemMoveHalfBackClick(Sender: TObject);

begin

RollBack(True);

SynchronizeMenuItems();

end;

procedure TMainForm.MenuItemMoveHalfForwardClick(Sender: TObject);

begin

RollForward(True);

SynchronizeMenuItems();

end;

procedure TMainForm.MenuItemMoveBackClick(Sender: TObject);

begin

RollBack(True);

if ComputerGame then RollBack(True);

SynchronizeMenuItems();

end;

procedure TMainForm.MenuItemMakeMoveClick(Sender: TObject);

procedure InvertEnabled();

var

i: Integer;

begin

for i := 0 to Self.MainMenu.Items.Count - 1 do

Self.MainMenu.Items[i].Enabled := not Self.MainMenu.Items[i].Enabled;

end;

var

CompNewPoint: TGamePoint;

LastCursor: TCursor;

begin

if IsEndGame() then

ShowMessage(GameOverText)

else

begin

InvertEnabled();

Self.OnClose := Self.CancelFormClose;

Self.BoardView.OnMouseDown := nil;

Self.BoardView.OnMouseMove := nil;

LastCursor := BoardView.Cursor;

BoardView.Cursor := crHourGlass;

UserBreak := False;

ComputerMove(CompNewPoint);

BoardView.Cursor := LastCursor;

if not IsOnBoard(CompNewPoint) then

ShowMessage(UserBreakText)

else

DoStep(CompNewPoint, ActivePlayer, True);

Self.OnClose := Self.FormClose;

Self.BoardView.OnMouseDown := Self.BoardViewMouseDown;

Self.BoardView.OnMouseMove := Self.BoardViewMouseMove;

InvertEnabled();

SynchronizeMenuItems();

end;

end;

{Меню "Вид"}

procedure TMainForm.MenuItemIncWindowClick(Sender: TObject);

begin

if Self.Height + IncPixelStep <= Screen.Height then begin

BoardView.Hide;

BoardView.width := BoardView.width + IncPixelStep;

BoardView.Picture.Bitmap.Width := BoardView.Picture.Bitmap.Width + IncPixelStep;

BoardView.Height := BoardView.Height + IncPixelStep;

BoardView.Picture.Bitmap.Height := BoardView.Picture.Bitmap.Height + IncPixelStep;

BoardView.Show;

if (Self.ClientHeight <> BoardView.Height + StatusBar.Height) or

(Self.ClientWidth <> BoardView.Width) then

begin

CorrectSize();

SynchronizeMenuItems();

end;

end;

end;

procedure TMainForm.MenuItemDecWindowClick(Sender: TObject);

begin

if (BoardView.Width - IncPixelStep) div BoardSize > MinPixelGridStep then begin

BoardView.Hide;

BoardView.width := BoardView.width - IncPixelStep;

BoardView.Picture.Bitmap.Width := BoardView.Picture.Bitmap.Width - IncPixelStep;

BoardView.Height := BoardView.Height - IncPixelStep;

BoardView.Picture.Bitmap.Height := BoardView.Picture.Bitmap.Height - IncPixelStep;

BoardView.Show;

if (Self.ClientHeight <> BoardView.Height + StatusBar.Height) or

(Self.ClientWidth <> BoardView.Width) then

begin

CorrectSize();

SynchronizeMenuItems();

end;

end;

end;

{Меню "Help"}

procedure TMainForm.MenuItemAboutClick(Sender: TObject);

begin

savedTitle := Application.Title;

try

Application.Title := 'О программе';

ShowMessage(AboutText);

finally

Application.Title := savedTitle

end

{TODO:сделать интерактивные гиперссылки в этом окне}

end;

procedure TMainForm.MenuItemHelpClick(Sender: TObject);

var

HelpFileName: string;

begin

HelpFileName := ParamStr(0);

HelpFileName := Copy(HelpFileName, 1, Length(HelpFileName) - 3) + 'txt';

//ToDo:сделать hlp или chm справку

if not FileExists(HelpFileName) then

ShowMessage(HelpFileNotFoundText + HelpFileName)

else

ShellExecute(Self.Handle, nil, PChar(HelpFileName), nil, nil, SW\_SHOW);

end;

{Меню "Стоп"}

procedure TMainForm.MenuItemStopClick(Sender: TObject);

begin

UserBreak := True;

end;

{Конец обработчиков нажатий на пункты меню}

procedure TMainForm.SynchronizeMenuItems();

begin

{Файл}

// Все пункты меню доступны всегда

{Игра}

MenuItemEditing.Checked := EditMode;

MenuItemName.Enabled := not EditMode;

MenuItemVsComp.Checked := ComputerGame;

MenuItemGameLevel.Caption := GameLevelText + '(' + IntToStr(GameLevel) + ')...';

MenuItemGameLevel.Enabled := ComputerGame;

{Ходы}

MenuItemShowMove.Enabled := CurrentSeq >= 0;

MenuItemMoveBack.Enabled := (CurrentSeq >= 1) and not EditMode;

MenuItemMoveHalfBack.Enabled := (CurrentSeq >= 0);

MenuItemMoveForward.Enabled := (SeqCount - CurrentSeq >= 2) and not EditMode;

MenuItemMoveHalfForward.Enabled := (SeqCount - CurrentSeq >= 1);

MenuItemMakeMove.Enabled := not EditMode;

miMarkPoint.Enabled := not bHideCursor;

miMarkPoint2.Enabled := not bHideCursor and EditMode;

{Вид}

MenuItemIncWindow.Enabled := Self.Height + IncPixelStep <= Screen.Height;

MenuItemDecWindow.Enabled := (BoardView.Width - IncPixelStep) div BoardSize > MinPixelGridStep;

miShowCursor.Checked := not bHideCursor;

miHighlight.Checked := not fNotHihlightLastMove;

{Помощь}

// Все пункты меню доступны всегда

IndicateGain(); //отображение счета

end;

{Основные игровые процедуры}

function TMainForm.DoStep(NewPoint: TGamePoint; const Player: Byte; const DrawPoint: Boolean): Boolean;

begin

Result := (Player = ActivePlayer) or EditMode;

if not Result then Exit;

Inc(CurrentSeq);

Result := TrySetPoint(NewPoint, Player, DrawPoint);

if Result then

begin

with PGamSeq[CurrentSeq] do

begin

Who := Player;

Point := NewPoint;

Gain[0] := CurGameState[brFirst].Gain;

Gain[1] := CurGameState[brSecond].Gain;

Count[0] := CurGameState[brFirst].Count;

Count[1] := CurGameState[brSecond].Count;

end;

SeqCount := CurrentSeq;

ActivePlayer := NextPlayer[ActivePlayer];

end

else

Dec(CurrentSeq);

end;

function TMainForm.TrySetPoint(

const NewPoint: TGamePoint;

const Player: Byte;

const DrawPoint: Boolean): Boolean;

begin

Result := Board[NewPoint.X, NewPoint.Y] = brFree;

if not Result then Exit;

Inc(CurGameState[Player].Count);

Board[NewPoint.X, NewPoint.Y] := Player;

if not DrawPoint then

with PGamSeq[CurrentSeq] do

begin

SetLength(BoardChangesList, (BoardSize + 1) \* (BoardSize + 1) \* debugconst);

//очень неэффективный расход памяти!!

//TODO:поставить здесь стек на базе указателей

NumberOfChanges := -1;

end;

if DrawPoint then DrawNewPoint(NewPoint, Player);

LastPrize := 0;

LastEmptySquare := 0;

if NumberNearGroups(NewPoint, Player) >= 2 then

Attack(NewPoint, Player, DrawPoint);

if (LastPrize = 0) and not IsNearOut(NewPoint) then

CounterAttack(NewPoint, Player, DrawPoint);

if not DrawPoint then

with PGamSeq[CurrentSeq] do

begin

SetLength(BoardChangesList, NumberOfChanges + 1);

//TODO:поставить здесь стек на базе указателей

end;

end;

function TMainForm.IsEndGame(): Boolean;

var

i, j: Integer;

begin

for i := 0 to BoardSize do

for j := 0 to BoardSize do

if Board[i, j] = brFree then

begin

Result := False;

Exit;

end;

Result := True;

end;

function TMainForm.NextPoint(const Pred: TGamePoint; const SearchFor, Mask: Byte): TGamePoint;

const

bGetLex: array[0..1, 0..8] of ShortInt =

((-1, -1, -1, 0, 1, 1, 1, 0, -1),

(-1, 0, 1, 1, 1, 0, -1, -1, -1));

var

i: Integer;

begin

for i := 7 downto 0 do

begin

Result.X := Pred.x + bGetLex[0, i];

Result.Y := Pred.y + bGetLex[1, i];

if IsOnBoard(Result) and ((Board[Result.X, Result.Y] and Mask) = SearchFor) then

Exit;

end;

Result.X := -1;

Result.Y := -1;

end;

function TMainForm.IsNear(const p1, p2: TGamePoint): Boolean;

(\* True , если p1, p2 - точки -соседи(разные точки)

цвет и статус не важен\*)

begin

Result := ((p1.x <> p2.X) or (p1.y <> p2.Y)) and

(abs(p1.x - p2.x) <= 1) and (abs(p1.y - p2.Y) <= 1)

end;

procedure TMainForm.ReduceEmptyCycles(const Ring1: TGameStack; const Ring1\_Depth: Integer;

var Ring2: TGameStack; var Ring2\_Depth: Integer);

var

i, j: Integer;

CurPoint: TGamePoint;

begin

Ring2\_Depth := 0;

Ring2[Ring2\_Depth] := Ring1[0];

Board[Ring2[0].x, Ring2[0].y] := Board[Ring2[0].x, Ring2[0].y] or brcheck2;

i := 0;

while (i < Ring1\_Depth) do

begin

CurPoint := Ring1[i];

j := Ring1\_Depth;

if (i = 0) then Dec(j);

while (j > i) and not IsNear(CurPoint, Ring1[j]) do Dec(j);

Inc(Ring2\_Depth);

Ring2[Ring2\_Depth] := Ring1[j];

Board[Ring2[Ring2\_Depth].X, Ring2[Ring2\_Depth].y] :=

Board[Ring2[Ring2\_Depth].X, Ring2[Ring2\_Depth].y] or brcheck2;

i := j;

end;

end;

function TMainForm.Attack(const NewPoint: TGamePoint; const Player: Byte; const Draw: Boolean): Boolean;

var

Ring1, Ring2: TGameStack;

Ring1\_Depth, Ring2\_Depth: Integer;

Target: TGamePoint;

NextPoint: TGamePoint;

ServerStack: TGameStack;

ServerStack\_Depth: Integer;

Stack2Rect: TGameRect;

{$IFDEF debug}

i, j: SmallInt;

{$ENDIF}

begin

Result := False;

target := NewPoint;

NextPoint := Target;

SetLength(Ring1, (BoardSize + 1) \* (BoardSize + 1));

SetLength(Ring2, Length(Ring1));

SetLength(ServerStack, Succ(BoardSize) \* Succ(BoardSize));

ServerStack\_Depth := -1;

Ring1\_Depth := 0;

Ring1[Ring1\_Depth] := Target;

while Ring1\_Depth >= 0 do

begin

Board[NextPoint.X, NextPoint.Y] := Board[NextPoint.X, NextPoint.Y] or brCheck;

Inc(ServerStack\_Depth);

ServerStack[ServerStack\_Depth] := NextPoint;

NextPoint := Ring1[Ring1\_Depth];

NextPoint := Self.NextPoint(NextPoint, Player, $FF);

if NextPoint.X = -1 then

begin

if (Ring1\_Depth >= 3) and IsNear(Ring1[Ring1\_Depth], Target) then

begin

ReduceEmptyCycles(Ring1, Ring1\_Depth, Ring2, Ring2\_Depth);

if Ring2\_Depth >= 3 then

if IsNear(Ring2[Ring2\_Depth], Target) then

begin

SetDefaultBounds(Stack2Rect);

SetNewBounds(Stack2Rect, Ring2, Ring2\_Depth);

if CheckNonEmptyFortress(Stack2Rect, Player, Ring2, Ring2\_Depth, Draw) then

Result := True;

end;

ResetStack(Ring2, Ring2\_Depth, Byte(not brCheck2));

end;

Dec(Ring1\_Depth);

if Ring1\_Depth >= 0 then

NextPoint := Ring1[Ring1\_Depth];

end

else

begin

Inc(Ring1\_Depth);

Ring1[Ring1\_Depth] := NextPoint;

end;

end;

ResetStack(ServerStack, ServerStack\_Depth, Byte(not brCheck));

{$IFDEF debug}

for i := 0 to BoardSize do

for j := 0 to BoardSize do

if (Board[i, j] and (brCheck or brCheck2)) <> 0 then

ShowMessage(Format('Attack:Error:Board[%u,%u]=%u', [j, i, Board[i, j]]));

{$ENDIF}

end;

function TMainForm.CheckNonEmptyFortress(

const StackBounds: TGameRect;

const Player: Byte;

const Ring: TGameStack;

const Ring\_Depth: Integer;

const Draw: Boolean): Boolean;

var

i, j: Integer;

CurPoint: TGamePoint;

begin

Result := False;

if (StackBounds.Right > 0) and (StackBounds.Bottom > 0) then

for i := StackBounds.Left + 1 to StackBounds.Right - 1 do

for j := StackBounds.Top + 1 to StackBounds.Bottom - 1 do

if (Board[i, j] = NextPlayer[Player]) then

begin

CurPoint.X := i;

CurPoint.Y := j;

if IsPointInsideRing(CurPoint, Ring, Ring\_Depth) then

if CaptureFortress(CurPoint, Player, Draw) then Result := True;

end;

end;

function TMainForm.CaptureFortress(

const NewPoint: TGamePoint;

const Player: Byte;

const Draw: Boolean): Boolean;

var

FillingArea, CheckingPoints: TGameStack;

FillingArea\_Depth, CheckingPoints\_Depth: Integer;

TempPoint: TGamePoint;

Ring: TGameStack;

Ring\_Depth: Integer;

i: Integer;

BrCheck4Stack: TGameStack;

Len\_BrCheck4: Integer;

{$IFDEF debug}

x, y: Integer;

{$ENDIF}

begin

SetLength(BrCheck4Stack, (BoardSize + 1) \* (BoardSize + 1) + 2);

Len\_BrCheck4 := -1;

SetLength(FillingArea, (BoardSize + 1) \* (BoardSize + 1) + 2);

SetLength(CheckingPoints, debugconst \* ((BoardSize + 1) \* (BoardSize + 1) + 2));

FillingArea\_Depth := -1;

CheckingPoints\_Depth := -1;

Inc(FillingArea\_Depth);

FillingArea[FillingArea\_Depth] := NewPoint;

Result := False;

while FillingArea\_Depth >= 0 do

begin

TempPoint := FillingArea[FillingArea\_Depth];

Dec(FillingArea\_Depth);

Inc(CheckingPoints\_Depth);

CheckingPoints[CheckingPoints\_Depth] := TempPoint;

if (Board[TempPoint.X, TempPoint.Y] and not brInside) = NextPlayer[Player] then

begin

Inc(CurGameState[Player].Gain);

Inc(LastPrize);

if not Draw then EnterChange(TempPoint, Byte(not (brCheck or BrCheck2 or brCheck3 or brCheck4)));

if Draw then KillPoint(TempPoint);

Board[TempPoint.X, TempPoint.Y] := Board[TempPoint.X, TempPoint.Y] or brDead or brInside;

Result := True;

end

else if not Vassal and ((Board[TempPoint.X, TempPoint.Y] = ((Player or brDead) or brInside))

or (Board[TempPoint.X, TempPoint.Y] = Player or brDead)) then

begin

if not Draw then EnterChange(TempPoint, Byte(not (brCHeck or BrCheck2 or brCheck3 or brCheck4)));

Board[TempPoint.X, TempPoint.Y] := Board[TempPoint.X, TempPoint.Y] and not brDead;

Dec(CurGameState[NextPlayer[Player]].Gain);

Inc(LastPrize);

end;

Board[TempPoint.X, TempPoint.Y] := Board[TempPoint.X, TempPoint.Y] or brCheck3;

PushNextPoints(FillingArea, FillingArea\_Depth, TempPoint, Player, brCheck4, BrCheck4Stack, Len\_BrCheck4);

end;

while CheckingPoints\_Depth >= 0 do

begin

TempPoint := CheckingPoints[CheckingPoints\_Depth];

Dec(CheckingPoints\_Depth);

Board[TempPoint.X, TempPoint.Y] := Board[TempPoint.X, TempPoint.Y] and not brCheck3;

if Result and (Board[TempPoint.X, TempPoint.Y] = brFree) then

begin

if not Draw then EnterChange(TempPoint, Byte(not (brCheck or BrCheck2 or brCheck3 or brCheck4)));

Board[TempPoint.X, TempPoint.Y] := brDead;

Inc(LastEmptySquare);

end;

end;

CheckingPoints := nil;

if Result and Draw then

for i := 0 to NewPoint.x - 1 do

if Board[i, NewPoint.y] and brcheck4 <> 0 then

begin

TempPoint.Y := NewPoint.Y;

TempPoint.X := i;

SetLength(Ring, (BoardSize + 1) \* (BoardSize + 1));

GetRing(TempPoint, Player, Ring, Ring\_Depth, brCheck4);

DrawFortress(Ring, Ring\_Depth, Player);

Ring := nil;

Break;

end;

ResetStack(BrCheck4Stack, Len\_brCheck4, Byte(not brCheck4));

FillingArea := nil;

{$IFDEF debug}

for x := 0 to BoardSize do

for y := 0 to BoardSize do

if (Board[x, y] and (brCheck3 or brCheck4)) <> 0 then

ShowMessage(Format('CaptureFortress:Error:Board[%u,%u]=%u', [y, x, Board[x, y]]));

{$ENDIF}

end;

procedure TMainForm.GetRing(NewPoint: TGamePoint; Player: Byte; var Ring: TGameStack; var Ring\_Depth: Integer; const Mask: Byte);

var

NextPoint: TGamePoint;

begin

NextPoint := NewPoint;

Ring\_Depth := 0;

Ring[Ring\_Depth] := NextPoint;

while Ring\_Depth >= 0 do

begin

Board[NextPoint.X, NextPoint.Y] := Board[NextPoint.X, NextPoint.Y] and not Mask;

NextPoint := Ring[Ring\_Depth];

NextPoint := Self.NextPoint(NextPoint, Mask, Mask);

if (NextPoint.X = -1) then

begin

if (Ring\_Depth >= 3) and IsNear(Ring[Ring\_Depth], NewPoint) then

Break;

Dec(Ring\_Depth);

if Ring\_Depth >= 0 then

NextPoint := Ring[Ring\_Depth];

end

else begin

Inc(Ring\_Depth);

Ring[Ring\_Depth] := NextPoint;

end;

end;

end;

procedure TMainForm.DrawFortress(const Ring: TGameStack; const Ring\_Depth: Integer; const Player: Byte);

var

NextPoint: TGamePoint;

Current: Word;

LastPenWidth, LastPenColor: Integer;

begin

Current := 0;

NextPoint := Ring[Current];

LastPenColor := BoardView.Canvas.Pen.Color;

BoardView.Canvas.Pen.Color := CurGameState[Player].Color;

LastPenWidth := BoardView.Canvas.Pen.Width;

BoardView.Canvas.Pen.Width := 2;

BoardView.Canvas.MoveTo(

GameSpaceToBoardViewSpace(NextPoint.X),

GameSpaceToBoardViewSpace(NextPoint.Y));

while (Current < Ring\_Depth) do

begin

inc(Current);

NextPoint := Ring[Current];

BoardView.Canvas.LineTo(

GameSpaceToBoardViewSpace(NextPoint.X),

GameSpaceToBoardViewSpace(NextPoint.Y));

end;

Current := 0;

NextPoint := Ring[Current];

BoardView.Canvas.LineTo(

GameSpaceToBoardViewSpace(NextPoint.X),

GameSpaceToBoardViewSpace(NextPoint.Y));

BoardView.Canvas.Pen.Color := LastPenColor;

BoardView.Canvas.Pen.Width := LastPenWidth;

end;

function TMainForm.IsPointInsideRing(

const Point: TGamePoint;

const Ring: TGameStack;

const RingLength: Integer): Boolean;

{Для точек ,входящих в кольцо, результат не определен}

const

sNone = $00; {0000.0000b}

sTop = $01; {0000.0001b}

sBottom = $02; {0000.0010b}

sTarget = $04; {0000.0100b}

var

Current, Bound: Integer;

Status: Byte;

Count: Byte;

NextLex: Byte;

SecondPass: Boolean;

function GetNext(): Byte;

begin

if (Point.Y >= Ring[Current].Y) then

case (Point.X - Ring[Current].X) of

-1: Result := sBottom;

0: Result := starget;

1: Result := sTop;

else Result := sNone;

end

else

Result := sNone;

end;

begin

Current := 0;

while Current <= RingLength do

begin

if GetNext() = sNone then Break;

inc(Current);

end;

if (Current = RingLength + 1) then

begin

Result := False;

Exit;

end;

Bound := Current;

Count := 0;

Status := sNone;

SecondPass := False;

while ((Current <> Bound) or not SecondPass) do

begin

if (Current = RingLength) then

Current := 0

else

Inc(Current);

NextLex := GetNext();

case NextLex of

sNone:

Status := sNone;

sBottom:

begin

if (status = (sTop or starget)) then inc(Count);

status := sBottom;

end;

sTop:

begin

if (status = (sBottom or starget)) then inc(Count);

status := sTop;

end;

sTarget:

status := status or sTarget;

else

assert(False, 'Внутреняя ошибка.№4' + #13#10 + 'Закройте программу');

end; {Case}

SecondPass := True;

end;

{Точка входит в объект, описываемый замкнутой кривой

если число пересечений луча , проведенного из этой точки в произвольном

направлении,с кривой - контуром границы нечетно.

Касание не считается пересечением}

Result := (Count and $01) <> 0;

end;

procedure TMainForm.EndGame; //освобождение ресурсов

begin

Board := nil;

PGamSeq := nil;

end;

procedure TMainForm.IndicateProgress(const MaxValue, CurValue: Integer);

begin

StatusBar.SimpleText := Format('%s %s %s %u %s %u',

[ComputerThinkText, ProgressText, ':', CurValue, '/', MaxValue]);

end;

procedure TMainForm.IndicateGain();

var

LastFontColor: Integer;

begin

with StatusBar.Canvas do

begin

LastFontColor := Font.Color;

Font.Color := CurGameState[brFirst].Color;

TextOut(StatusBar.Width div 2, 2, ' ');

TextOut(StatusBar.Width div 2, 2, IntToStr(CurGameState[brFirst].Gain));

Font.Color := clBlack;

TextOut(PenPos.X, PenPos.Y, ':');

Font.Color := CurGameState[brSecond].Color;

TextOut(PenPos.X, PenPos.Y, IntToStr(CurGameState[brSecond].Gain));

case EditMode of

False:

begin

Font.Color := CurGameState[ActivePlayer].Color;

TextOut(PenPos.X + 6, PenPos.Y, MustMoveText + CurGameState[ActivePlayer].Name);

end;

else

begin

Font.Color := clBlack;

TextOut(PenPos.X + 6, PenPos.Y, EditModeText);

end;

end;

Font.Color := LastFontColor;

end;

end;

procedure TMainForm.ResetStack(const Stack: TGameStack; const Stack\_Depth: Integer; const Mask: Byte);

var

i: Integer;

begin

for i := Stack\_Depth downto 0 do

Board[Stack[i].X, Stack[i].Y] := Board[Stack[i].X, Stack[i].Y] and Mask;

end;

{Фундаментальные процедуры}

procedure TMainForm.DrawGrid(); //Отрисовка сетки на BoardView

procedure FillCanvas(const Color: Integer);

var

fl: TRect;

begin

SetRect(fl, 0, 0, BoardView.Width, BoardView.Height);

BoardView.Canvas.Brush.Color := Color;

BoardView.Canvas.FillRect(fl);

end;

var

i, j: SmallInt;

begin

CurGameState[brFirst].Gain := 0;

CurGameState[brSecond].Gain := 0;

CurGameState[brFirst].Count := 0;

CurGameState[brSecond].Count := 0;

for i := 0 to BoardSize do

for j := 0 to BoardSize do

Board[i][j] := 0;

FillCanvas(clWhite); //TODO: загрузка цветовых настроек из реестра или ini-файла

begin

PixelGridStep := (BoardView.Height - 20) div BoardSize;

PixelIndent := (((BoardView.height - 20) mod BoardSize) div 2) + 10;

i := PixelIndent;

while i <= BoardView.height - PixelIndent do

begin

{Горизонтальные линии}

BoardView.Canvas.MoveTo(0, i);

BoardView.Canvas.LineTo(BoardView.Width, i);

i := i + PixelGridStep;

end;

i := PixelIndent;

while i <= BoardView.Width - PixelIndent do

begin

{Вертикальные линии}

BoardView.Canvas.MoveTo(i, 0);

BoardView.Canvas.LineTo(i, BoardView.height);

i := i + PixelGridStep;

end;

end;

PixelPointRadius := (PixelGridStep div 5) + Byte(PixelGridStep < 30) \* (PixelGridStep div 20) +

Byte(PixelGridStep < 20) \* (PixelGridStep div 12);

if PixelPointRadius <= 2 then Inc(PixelPointRadius);

PixelTolerance := PixelGridStep div 3;

end;

procedure TMainForm.RollBack(const Draw: Boolean);

{ Откат на один полуход с изменением BoardView(Draw<>False)

или без изменения (Draw=False) - гораздо более быстрый}

var

i: Integer;

begin

ActivePlayer := PGamSeq[CurrentSeq].Who;

if CurrentSeq < 0 then Exit;

case Draw of

False:

begin

if PGamSeq[CurrentSeq].NumberOfChanges >= 0 then

begin

for I := PGamSeq[CurrentSeq].NumberOfChanges downto 0 do

with PGamSeq[CurrentSeq].BoardChangesList[I] do

Board[Point.x, Point.y] := BoardValue;

end;

Board[PGamSeq[CurrentSeq].Point.X, PGamSeq[CurrentSeq].Point.Y] := brFree;

PGamSeq[CurrentSeq].BoardChangesList := nil;

Dec(CurrentSeq);

SeqCount := CurrentSeq;

if CurrentSeq >= 0 then

begin

CurGameState[brFirst].Gain := PGamSeq[CurrentSeq].Gain[0];

CurGameState[brSecond].Gain := PGamSeq[CurrentSeq].Gain[1];

CurGameState[brFirst].Count := PGamSeq[CurrentSeq].Count[0];

CurGameState[brSecond].Count := PGamSeq[CurrentSeq].Count[1];

end

else

begin

CurGameState[brFirst].Gain := 0;

CurGameState[brSecond].Gain := 0;

CurGameState[brFirst].Count := 0;

CurGameState[brSecond].Count := 0;

end;

end;

else

begin

Dec(CurrentSeq);

DrawGrid();

for i := 0 to CurrentSeq do

TrySetPoint(PGamSeq[i].Point, PGamSeq[i].Who, True);

end;

end; {case Draw of}

end;

procedure TMainForm.DrawBoard();

var

LastActivePlayer: Integer;

begin

LastActivePlayer := ActivePlayer;

Inc(CurrentSeq);

RollBack(True);

ActivePlayer := LastActivePlayer;

end;

procedure TMainForm.RollForward(const Draw: Boolean);

{Медленный возврат на один полуход с изменением BoardView

Применяется при нажатии на кнопку}

begin

assert(Draw, ' Невозможен отказ от отката при прсчете ходов');

if (CurrentSeq < SeqCount) then

begin

Inc(CurrentSeq);

TrySetPoint(PGamSeq[CurrentSeq].Point, PGamSeq[CurrentSeq].Who, Draw);

ActivePlayer := NextPlayer[PGamSeq[CurrentSeq].Who];

end;

end;

function TMainForm.SaveLoadGameFile(

const filename: string;

const Load: Boolean): Boolean;

const

GameFileSignature = '!Tochki v0.1';

BoardSizeSignature = 'BoardSize';

var

InOutFile: System.text;

i, j: Integer;

DelimiterPos: Integer;

Temp: Integer;

LastEditMode: Boolean;

t: string;

function ParseNextLine: Boolean;

var

CommaPos: Integer;

Player: Integer;

begin

Result := True;

ReadLn(InOutFile, t);

if (length(t) = 0) or (t[1] in [';', '#']) then Exit;

DelimiterPos := pos(':', t);

CommaPos := pos(',', t);

Result := False;

if (DelimiterPos < 2)

or (CommaPos < DelimiterPos + 2)

or (CommaPos = length(t))

or not TryStrToInt(Copy(t, 1, DelimiterPos - 1), Player)

or not (Player in [brFirst, brSecond])

or not TryStrToInt(Copy(t, DelimiterPos + 1, CommaPos - 1 - DelimiterPos), j)

or not TryStrToInt(Copy(t, CommaPos + 1, Length(t) - CommaPos), i)

or not IsOnBoard(i, j)

or (Board[i, j] <> brFree) then

Exit;

Result := DoStep(MakePoint(i, j), Player, True);

end;

label

Abort;

begin

AssignFile(InOutFile, FileName);

case Load of

False:

begin

FileMode := fmOpenWrite;

Rewrite(InOutFile);

WriteLn(InOutFile, GameFileSignature);

WriteLn(InOutFile, BoardSizeSignature + '=' + IntToStr(BoardSize));

WriteLn(InOutFile, '#' + CurGameState[brFirst].Name + ' - ' + CurGameState[brSecond].Name);

for i := 0 to CurrentSeq do

WriteLn(InOutFile, IntToStr(PGamSeq[i].Who) + ':' +

IntToStr(PGamSeq[i].Point.Y) + ',' + IntToStr(PGamSeq[i].Point.X));

CloseFile(InOutFile);

Result := True;

end;

else

begin

Result := False;

LastEditMode := EditMode;

FileMode := fmOpenRead;

Reset(InOutFile);

ReadLn(InOutFile, t);

if t <> '!Tochki v0.1' then goto Abort;

ReadLn(InOutFile, t);

DelimiterPos := pos('=', t);

if (DelimiterPos = 0)

or (copy(t, 1, DelimiterPos - 1) <> 'BoardSize')

or (DelimiterPos = length(t))

or not TryStrToInt(Copy(t, DelimiterPos + 1, Length(t) - DelimiterPos), Temp) then

goto Abort;

EndGame();

PrepareGame(Temp);

EditMode := True;

while not eof(InOutFile) do

if not ParseNextLine() then goto Abort;

Result := True;

Abort:

CloseFile(InOutFile);

EditMode := LastEditMode;

end; {else Case}

end; {Case Mode of}

end;

procedure TMainForm.PrepareGame(SizeOfBoard: Integer);

var

BoardCenter: TGamePoint;

begin

BoardSize := SizeOfBoard;

GameLevel := (MinGameLevel + MaxGameLevel) div 2;

SetLength(Board, BoardSize + 1, BoardSize + 1);

SetLength(PGamSeq, (BoardSize + 1) \* (BoardSize + 1));

CurrentSeq := -1;

SeqCount := CurrentSeq;

DrawGrid();

ActivePlayer := brFirst;

BoardCenter.X := BoardSize div 2;

BoardCenter.Y := BoardSize div 2;

CursorPos := BoardCenter;

end;

procedure TMainForm.InitDonePerspectivePoints(var p: Pointer; const WhatToDo: Boolean);

begin

//ToDo:см. вводный комментарий

end;

procedure TMainForm.ComputerMove(var BestPoint: TGamePoint);

var

BestPoints: TGameStack;

Len\_BestPoints: Integer;

P: Pointer;

begin

ComputerTHinking := True;

try

InitDonePerspectivePoints(P, False);

SetLength(BestPoints, (BoardSize + 1) \* (BoardSize + 1));

GetBestPoints(GameLevel, BestPoints, Len\_BestPoints, P);

GetBestPosition(BestPoint, BestPoints, Len\_BestPoints);

InitDonePerspectivePoints(P, True);

if UserBreak then BestPoint := MakePoint(-1, -1);

finally

ComputerTHinking := False;

end;

end;

function TMainForm.GetBestPosition(var BestPoint: TGamePoint; const BestPoints: TGameStack; const Len\_BestPoints: Integer): Integer;

var

i, j: Integer;

CurEstimate: Integer;

CurPoint: TGamePoint;

begin

Result := Low(Result);

BestPoint := MakePoint(-1, -1);

if Len\_BestPoints < 0 then

begin

for i := 0 to BoardSize do

for j := 0 to BoardSize do

if Board[i, j] = brFree then

begin

CurPoint := MakePoint(i, j);

CurEstimate := PositionEstimate(CurPoint);

if (CurEstimate > Result) or ((CurEstimate = Result) and (Random(BoardSize) = 0)) then

begin

Result := CurEstimate;

BestPoint := CurPoint;

end;

end;

end

else

begin

for i := 0 to Len\_BestPoints do

begin

CurPoint := BestPoints[i];

CurEstimate := PositionEstimate(CurPoint);

if (CurEstimate > Result) or ((CurEstimate = Result) and (Random(BoardSize) = 0)) then

begin

Result := CurEstimate;

BestPoint := CurPoint;

end;

end;

end;

end;

function TMainForm.GetBestPoints(const SearchDepth: Integer; var BestPoints: TGameStack; var Len\_BestPoints: Integer; const p: Pointer): Integer;

// возвращает лучшую позицию с просчетом на глубину SearchDepth полуходов

var

i, j: Integer;

CurEstimate: Integer;

CurPoint: TGamePoint;

AllPerspectivePoints: Integer;

ProcessedPoints: Integer;

label

Abort;

begin

Result := Low(Result);

Len\_BestPoints := -1;

if SearchDepth <= 0 then Exit;

{$IFDEF debug}

DrawBoard();

{$ENDIF}

AllPerspectivePoints := 0;

for i := 0 to BoardSize do

for j := 0 to BoardSize do

if (Board[i, j] = brFree) and IsPointPerspective(SearchDepth, p, i, j) then

Inc(AllPerspectivePoints);

ProcessedPoints := 0;

Self.IndicateProgress(AllPerspectivePoints, ProcessedPoints);

for i := 0 to BoardSize do

for j := 0 to BoardSize do

if (Board[i, j] = brFree) and IsPointPerspective(SearchDepth, p, i, j) then

begin

CurPoint := MakePoint(i, j);

CurEstimate := GetMinMaxEstimate(SearchDepth - 1, CurPoint, p);

if UserBreak then goto Abort;

Inc(ProcessedPoints);

Self.IndicateProgress(AllPerspectivePoints, ProcessedPoints);

{$IFDEF debug}

BoardView.Canvas.Brush.Color := clWhite;

BoardView.Canvas.Pen.Color := clBlack;

Self.BoardView.Canvas.TextOut(

GameSpaceToBoardViewSpace(CurPoint.X),

GameSpaceToBoardViewSpace(CurPoint.Y),

IntToStr(CurEstimate));

{$ENDIF}

if (CurEstimate > Result) then

begin

Result := CurEstimate;

Len\_BestPoints := 0;

BestPoints[Len\_BestPoints] := CurPoint;

end

else if (CurEstimate = Result) then

begin

Inc(Len\_BestPoints);

BestPoints[Len\_BestPoints] := CurPoint;

end;

end;

Exit;

Abort:

Len\_BestPoints := -1;

end;

function TMainForm.IsPointPerspective(const SearchDepth: Integer;

const p: Pointer; const x, y: integer): Boolean;

begin

//ToDo:см. вводный комментарий

assert(SearchDepth <= 4, ' IsPointPerspective: функция бессмысленна, если SearchDepth > 4');

Result := (NumberNearPoints(MakePoint(x, y), ActivePlayer) <> 0) or

(NumberNearPoints(MakePoint(x, y), NextPlayer[ActivePlayer]) <> 0)

end;

function TMainForm.GetMinMaxEstimate(SearchDepth: Integer; const TestedPoint: TGamePoint; const p: Pointer): Integer;

var

i, j: Integer;

BestEstimate, CurEstimate: Integer;

{$IFDEF debug}

BoardCopy: TGameBoard;

GameStateCopy: TPlayersInfo;

LastPlayer: Byte;

LastCurrentSeq: Integer;

LastMoveInfoEx: TMoveInfoEx;

function ArraysEqual(): Boolean;

var

i, j: Integer;

begin

Result := False;

for i := 0 to BoardSize do

for j := 0 to BoardSize do

if Board[i, j] <> BoardCopy[i, j] then Exit;

Result := True;

end;

{$ENDIF}

label Abort;

begin

if UserBreak then

begin

Result := 0;

Exit;

end;

{$IFDEF debug}

BoardCopy := Copy(Board);

GameStateCopy := CurGameState;

LastCurrentSeq := CurrentSeq;

LastPlayer := ActivePlayer;

if CurrentSeq >= 0 then LastMoveInfoEx := PGamSeq[CurrentSeq];

{$ENDIF}

DoStep(TestedPoint, ActivePlayer, False);

Result := (LastPrize shl 10) + (LastEmptySquare shl 2);

BestEstimate := Low(BestEstimate);

if SearchDepth > 0 then

for i := 0 to BoardSize do

for j := 0 to BoardSize do

if (Board[i, j] = brFree) and IsPointPerspective(SearchDepth - 1, p, i, j) then

begin

Application.ProcessMessages();

if UserBreak then goto Abort;

CurEstimate := GetMinMaxEstimate(SearchDepth - 1, MakePoint(i, j), p);

if UserBreak then goto Abort;

if (CurEstimate > BestEstimate) then BestEstimate := CurEstimate;

end;

Abort:

if BestEstimate = Low(BestEstimate) then BestEstimate := 0;

Result := Result - BestEstimate;

RollBack(False);

{$IFDEF debug}

if LastCurrentSeq >= 0 then

assert(

ArraysEqual()

and CompareMem(@CurGameState, @GameStateCopy, SizeOf(CurGameState))

and CompareMem(@LastMoveInfoEx, @PGamSeq[CurrentSeq], SizeOf(TMoveInfoEx))

and (LastCurrentSeq = CurrentSeq)

and (ActivePlayer = LastPlayer),

' Неправильно работает FastRollBack!!')

else

assert(

ArraysEqual()

and CompareMem(@CurGameState, @GameStateCopy, SizeOf(CurGameState))

and (LastCurrentSeq = CurrentSeq)

and (ActivePlayer = LastPlayer),

' Неправильно работает FastRollBack!!');

{$ENDIF}

end;

function TMainForm.PositionEstimate(const TestedPoint: TGamePoint): Integer;

{учет позиционных эвристик}

const

cgSumma: array[0..8] of Integer = (-5, -1, 0, 0, 1, 2, 5, 20, 30);

var

g1, g2: Integer;

c1, c2: Integer;

begin

g1 := NumberNearGroups(TestedPoint, ActivePlayer);

g2 := NumberNearGroups(TestedPoint, NextPlayer[ActivePlayer]);

c1 := cgSumma[NumberNearPoints(TestedPoint, ActivePlayer)];

c2 := cgSumma[NumberNearPoints(TestedPoint, NextPlayer[ActivePlayer])];

Result := (g1 \* 3 + g2 \* 2) \* (5 - abs(g1 - g2)) - c1 - c2;

if CurrentSeq > 0 then

Result := Result + Byte(IsNear(PGamSeq[CurrentSeq - 1].Point, TestedPoint)) \* 5;

{эмпирическая формула оценки важности точки при просчете ходов}

end;

procedure TMainForm.EnterChange(const NewPoint: TGamePoint; const Mask: Byte);

begin

with PGamSeq[CurrentSeq] do

begin

Inc(NumberOfChanges);

BoardChangesList[NumberOfChanges].Point := NewPoint;

BoardChangesList[NumberOfChanges].BoardValue := Board[NewPoint.X, NewPOint.Y] and Mask;

end;

end;

procedure TMainForm.SetDefaultBounds(var bounds: TGameRect);

begin

Bounds.Left := BoardSize;

Bounds.Top := BoardSize;

Bounds.Right := 0;

Bounds.Bottom := 0;

end;

procedure TMainForm.CorrectStackBounds(const NewPoint: TGamePoint; var Bounds: TGameRect);

begin

if (NewPoint.x < Bounds.Left) then Bounds.Left := NewPoint.x;

if (NewPoint.x > Bounds.Right) then Bounds.Right := NewPoint.x;

if (NewPoint.y < Bounds.Top) then Bounds.Top := NewPoint.y;

if (NewPoint.y > Bounds.Bottom) then Bounds.Bottom := NewPoint.y;

end;

procedure TMainForm.SetNewBounds(var bounds: TGameRect; const NewValues: TGameStack; CountV: Word);

var

i: SmallInt;

begin

for i := 0 to CountV do

CorrectStackBounds(NewValues[i], Bounds);

end;

function TMainForm.CounterAttack(const NewPoint: TGamePoint; const Player: Byte; const Draw: Boolean): Boolean;

var

FillingArea, CheckingPoints: TGameStack;

FillingArea\_Depth, CheckingPoints\_Depth: Integer;

Ring: TGameSTack;

Ring\_Depth: Integer;

TempPoint: TGamePoint;

i: Integer;

NeedBreak: Boolean;

AnotherPlayer: Byte;

brCheck4Stack: TGameStack;

Len\_brCheck4: Integer;

{$IFDEF debug}

x, y: Integer;

{$ENDIF}

begin

NeedBreak := True;

Result := False;

AnotherPlayer := NextPlayer[Player];

for i := NewPoint.X - 1 downto 0 do

if Board[i, NewPoint.Y] = AnotherPlayer then

begin

NeedBreak := False;

Break;

end

else if (Board[i, NewPoint.y] <> brFree) then

Exit;

if NeedBreak then Exit;

NeedBreak := True;

for i := NewPoint.x + 1 to BoardSize do

if Board[i, NewPoint.Y] = AnotherPlayer then

begin

NeedBreak := False;

Break;

end

else if (Board[i, NewPoint.y] <> brFree) then

Exit;

if NeedBreak then Exit;

NeedBreak := True;

for i := NewPoint.y - 1 downto 0 do if

Board[NewPoint.x, i] = AnotherPlayer then

begin

NeedBreak := False;

Break;

end

else if (Board[NewPoint.x, i] <> brFree) then

Exit;

if NeedBreak then Exit;

NeedBreak := True;

for i := NewPoint.y + 1 to BoardSize do

if Board[NewPoint.x, i] = AnotherPlayer then

begin

NeedBreak := False;

Break;

end

else if (Board[NewPoint.x, i] <> brFree) then

Exit;

if NeedBreak then Exit;

SetLength(brCHeck4Stack, (BoardSize + 1) \* (BoardSize + 1));

Len\_brCheck4 := -1;

SetLength(FillingArea, (BoardSize + 1) \* (BoardSize + 1));

SetLength(CheckingPoints, debugconst \* (BoardSize + 1) \* (BoardSize + 1));

FillingArea\_Depth := -1;

CheckingPoints\_Depth := -1;

Result := True;

Inc(CheckingPoints\_Depth);

CheckingPoints[CheckingPoints\_Depth] := NewPoint;

PushNextPoints(FillingArea, FillingArea\_Depth, NewPoint, AnotherPlayer, brCheck4, brCHeck4STack, Len\_brCheck4);

TempPoint := NewPoint;

while Result and (FillingArea\_Depth >= 0) do

begin

Board[TempPoint.x, TempPoint.y] := Board[TempPoint.x, TempPoint.y] or brCheck3;

TempPoint := FillingArea[FillingArea\_Depth];

Dec(FillingArea\_Depth);

Inc(CheckingPoints\_Depth);

CheckingPoints[CheckingPoints\_Depth] := TempPoint;

if ((Board[TempPoint.x, TempPoint.y] <> AnotherPlayer) and (Board[TempPoint.x, TempPoint.y] <> brFree) and

(Board[TempPoint.x, TempPoint.y] <> brCheck3)) or (IsNearOut(TempPoint)) then

begin

Result := False;

Break;

end;

PushNextPoints(FillingArea, FillingArea\_Depth, TempPoint, AnotherPlayer, brcheck4, brCHeck4Stack, Len\_brCheck4)

end;

case Result of

True:

begin

Inc(CurGameState[AnotherPlayer].Gain);

Dec(LastPrize);

if not Draw then EnterChange(NewPoint, Byte(not (brCheck or BrCheck2 or brCheck3 or brCheck4)));

Board[NewPoint.X, NewPoint.Y] := Player or brDead;

while CheckingPoints\_Depth > 0 do

begin

TempPoint := CheckingPoints[CheckingPoints\_Depth];

Dec(CheckingPoints\_Depth);

Board[TempPoint.X, TempPoint.Y] := Board[TempPoint.X, TempPoint.Y] and not brCheck3;

if not Draw then EnterChange(TempPoint, Byte(not (brCheck or BrCheck2 or brCheck3 or brCheck4)));

Board[TempPoint.X, TempPoint.Y] := brDead;

Dec(LastEmptySquare);

end;

if Draw then

begin

for i := 0 to NewPoint.x - 1 do

if (Board[i, NewPoint.y] and brcheck4) <> 0 then

begin

SetLength(Ring, (BoardSize + 1) \* (BoardSize + 1));

Ring\_Depth := -1;

GetRing(MakePoint(i, NewPoint.y), Player, Ring, Ring\_Depth, brCheck4);

Break;

end;

KillPoint(NewPoint);

DrawFortress(Ring, Ring\_Depth, AnotherPlayer);

Ring := nil;

end;

end;

False:

ResetStack(CheckingPoints, CheckingPoints\_Depth, Byte(not brCheck3));

end;

ResetStack(brCheck4Stack, Len\_brCheck4, Byte(not brCheck4));

FillingArea := nil;

CheckingPoints := nil;

{$IFDEF debug}

for x := 0 to BoardSize do

for y := 0 to BoardSize do

if (Board[x, y] and (brCheck4 or brCheck3)) <> 0 then

ShowMessage(Format('CounterAttack:Error:Board[%u,%u]=%u', [y, x, Board[x, y]]));

{$ENDIF}

end;

procedure TMainForm.PushNextPoints(

var p: TGameStack;

var p\_len: Integer;

const NextPoint: TGamePoint;

const Player: Byte;

const Mask: Byte;

var Changes: TGameStack;

var Changes\_Depth: Integer);

const

Neighbour: array[0..3, 0..1] of Integer = ((0, 1), (1, 0), (0, -1), (-1, 0));

var

i: Integer;

CurPoint: Byte;

begin

for i := 3 downto 0 do

begin

CurPoint := Board[NextPoint.X + Neighbour[i, 0], NextPoint.Y + Neighbour[i, 1]];

if (CurPoint and brcheck3) = 0 then

if ((CurPoint and Player) <> Player) or

(CurPoint = (Player or brDead) or brInside) or

(CurPoint = (Player or brDead)) then

begin

Inc(p\_len);

p[p\_len] := MakePoint(NextPoint.X + Neighbour[i, 0], NextPoint.Y + Neighbour[i, 1]);

end

else

if (CurPoint and Player) <> 0 then

begin

Board[NextPoint.X + Neighbour[i, 0], NextPoint.Y + Neighbour[i, 1]] :=

Board[NextPoint.X + Neighbour[i, 0], NextPoint.Y + Neighbour[i, 1]] or Mask;

Inc(Changes\_Depth);

Changes[Changes\_Depth] := MakePoint(NextPoint.X + Neighbour[i, 0], NextPoint.Y + Neighbour[i, 1]);

end;

end;

end;

function TMainForm.IsNearOut(const CheckedPoint: TGamePoint): Boolean;

begin

Result := (CheckedPoint.X = 0) or (CheckedPoint.Y = 0)

or (CheckedPoint.X = BoardSize) or (CheckedPoint.Y = BoardSize)

end;

function TMainForm.NumberNearGroups(

const CheckedPoint: TGamePoint; const Player: Byte): Integer;

var

x, y: SmallInt;

{Возвращает связность точки - число в интервале 0..4}

const

bNone = $00;

bState = $01;

bGetLex: array[0..1, 0..8] of ShortInt =

((-1, -1, -1, 0, 1, 1, 1, 0, -1),

(-1, 0, 1, 1, 1, 0, -1, -1, -1));

function GetLex(i: ShortInt): Byte;

begin

if not IsOnBoard(x + bGetLex[0, i], y + bGetLex[1, i]) then

GetLex := bNone

else

if Board[x + bGetLex[0, i], y + bGetLex[1, i]] = Player then

GetLex := bState

else

GetLex := bNone;

end;

const

CyclebStates: array[0..1, 0..1, 0..1] of Byte =

(((bNone, bState), (bNone, bState)),

((bNone, bState), (bState, bState)));

var

i, k, n: Byte;

begin

x := CheckedPoint.x;

y := CheckedPoint.y;

Result := 0;

k := GetLex(7);

n := GetLex(0);

k := CyclebStates[1, k, n];

for i := 1 to 8 do

begin

n := GetLex(i);

if (k = bNone) and (n = bState) then inc(Result);

k := CyclebStates[(i and $01) xor $01, k, n];

end;

end;

function TMainForm.IsOnBoard(const x, y: SmallInt): Boolean;

begin

Result := (x >= 0) and (y >= 0) and (x <= BoardSize) and (y <= BoardSize)

end;

function TMainForm.IsOnBoard(const TestedPoint: TGamePoint): Boolean;

begin

Result := (TestedPoint.x >= 0) and (TestedPoint.y >= 0) and (TestedPoint.x <= BoardSize) and (TestedPoint.y <= BoardSize)

end;

procedure TMainForm.CorrectSize();

begin

BoardView.Hide;

Self.ClientHeight := BoardView.Height + StatusBar.Height;

Self.ClientWidth := BoardView.Width;

BoardView.Show;

DrawBoard();

end;

procedure TMainForm.KillPoint(const DeadPoint: TGamePoint);

begin

DrawCircle(

GameSpaceToBoardViewSpace(DeadPoint.Y),

GameSpaceToBoardViewSpace(DeadPoint.X),

PixelPointRadius div 2,

clWhite);

end;

procedure TMainForm.DrawNewPoint(const NewPoint: TGamePoint; const Player: Integer);

begin

DrawCircle(

GameSpaceToBoardViewSpace(NewPoint.Y),

GameSpaceToBoardViewSpace(NewPoint.X),

PixelPointRadius,

CurGameState[Player].Color);

end;

function TMainForm.BoardViewSpaceMayTranslateToGameSpace(const X: Integer): Boolean;

begin

//Result := ((X + PixelGridStep - PixelIndent - PixelTolerance) mod PixelGridStep) <= (PixelTolerance \* 2)//это почему-то не работает , хотя должно. Разобраться!!

Result := (frac(abs(X - PixelIndent) / PixelGridStep) < PixelTolerance / PixelGridStep) or

(frac(abs(X - PixelIndent) / PixelGridStep) > 1 - PixelTolerance / PixelGridStep);

end;

function TMainForm.BoardViewSpaceToGameSpace(const X: Integer): Integer;

begin

Result := (X - PixelIndent + PixelGridStep div 2) div PixelGridStep;

end;

function TMainForm.GameSpaceToBoardViewSpace(const X: Integer): Integer;

begin

Result := PixelIndent + PixelGridStep \* X;

end;

function TMainForm.BoardViewCoordToGamePoint(const X, Y: Integer): TGamePoint;

begin

Result.x := BoardViewSpaceToGameSpace(x);

Result.y := BoardViewSpaceToGameSpace(y);

end;

function TMainForm.TryBoardViewCoordToGamePoint(const X, Y: Integer; var GamePoint: TGamePoint): Boolean;

begin

Result := BoardViewSpaceMayTranslateToGameSpace(x) and BoardViewSpaceMayTranslateToGameSpace(y);

if Result then

GamePoint := BoardViewCoordToGamePoint(x, y);

end;

procedure TMainForm.DrawCircle(const X, Y, R: Integer; const Color: Integer = -1);

begin

if Color <> -1 then BoardView.Canvas.Brush.Color := Color;

BoardView.Canvas.Ellipse(Y - R, X - R, Y + R, X + R);

end;

function TMainForm.MakePoint(const x, y: SmallInt): TGamePoint;

begin

result.X := x;

result.Y := y;

end;

function TMainForm.NumberNearPoints(const CheckedPoint: TGamePoint; const Player: Byte): Integer;

const

bGetLex: array[0..1, 0..8] of ShortInt =

((-1, -1, -1, 0, 1, 1, 1, 0, -1),

(-1, 0, 1, 1, 1, 0, -1, -1, -1));

var

i: Integer;

begin

Result := 0;

for i := 0 to 7 do

if IsOnBoard(CheckedPoint.x + bGetLex[0, i], CheckedPoint.y + bGetLex[1, i]) and

(Board[CheckedPoint.x + bGetLex[0, i], CheckedPoint.y + bGetLex[1, i]] = Player) then

Inc(Result);

end;

procedure TMainForm.HideCursor(const Point: TGamePoint);

var

cRGN: HRGN;

begin

cRGN := GetCursorRGN(Point);

InvalidateRgn(Self.Handle, cRGN, TRUE);

DeleteObject(cRGN);

end;

procedure TMainForm.ShowCursor(const Point: TGamePoint; const Color: TColor);

var

cRGN: HRGN;

PolyBrush: HBRUSH;

begin

cRGN := GetCursorRGN(Point);

PolyBrush := CreateSolidBrush(Color);

FillRgn(Self.Canvas.Handle, cRGN, PolyBrush);

DeleteObject(PolyBrush);

DeleteObject(cRGN);

end;

function TMainForm.GetCursorRGN(const Point: TGamePoint): HRGN;

var

X, Y: Integer;

hExternalCircle, hInternalCircle: HRGN;

ExternalCircleRadius, InternalCircleRadius: Integer;

begin

ExternalCircleRadius := round(PixelGridStep \* 0.7);

InternalCircleRadius := round(ExternalCircleRadius \* 0.7);

X := GameSpaceToBoardViewSpace(Point.X);

Y := GameSpaceToBoardViewSpace(Point.Y);

hExternalCircle := CreateEllipticRgn(X + ExternalCircleRadius, Y + ExternalCircleRadius, X - ExternalCircleRadius, Y - ExternalCircleRadius);

hInternalCircle := CreateEllipticRgn(X + InternalCircleRadius, Y + InternalCircleRadius, X - InternalCircleRadius, Y - InternalCircleRadius);

Win32Check(CombineRgn(hExternalCircle, hExternalCircle, hInternalCircle, RGN\_XOR) = COMPLEXREGION);

Result := hExternalCircle;

DeleteObject(hInternalCircle);

end;

procedure TMainForm.SetCursorPos(const CurPoint: TGamePoint);

begin

if IsOnBoard(CurPoint) then

begin

HideCursor(fCursorPos);

fCursorPos := CurPoint;

end;

end;

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

Shift: TShiftState);

var

NewPoint: TGamePoint;

begin

if not HideKbCursor and (Shift = []) then

begin

NewPoint := CursorPos;

case Key of

VK\_LEFT: Dec(NewPoint.X);

VK\_DOWN: Inc(NewPoint.Y);

VK\_UP: Dec(NewPoint.Y);

VK\_RIGHT: Inc(NewPoint.X);

VK\_HOME:

begin

newPoint.X := BoardSize div 2;

newPoint.Y := BoardSize div 2;

end;

end;

if IsOnBoard(NewPoint) and not CompareMem(@fCursorPos, @NewPoint, SizeOf(TGamePoint)) then

CursorPos := NewPoint;

end;

end;

procedure TMainForm.PaintBoardView(var WMPaint: TWMPaint);

var

LastPoint: TGamePoint;

begin

inherited;

if not ComputerThinking and (NotHighlightLastMove = False) and (CurrentSeq >= 0 )then

begin

LastPoint := PGamSeq[CurrentSeq].Point;

ShowCursor(LastPoint, {LastColor}clYellow);

end;

if not bHideCursor then

ShowCursor(CursorPos, CurGameState[ActivePlayer].Color);

end;

procedure TMainForm.miMarkPointClick(Sender: TObject);

begin

BoardViewMouseDown(Self, TMouseButton(ActivePlayer - 1), [], GameSpaceToBoardViewSpace(CursorPos.X), GameSpaceToBoardViewSpace(CursorPos.Y));

end;

procedure TMainForm.miMarkPoint2Click(Sender: TObject);

begin

BoardViewMouseDown(Self, TMouseButton(NextPlayer[ActivePlayer] - 1), [], GameSpaceToBoardViewSpace(CursorPos.X), GameSpaceToBoardViewSpace(CursorPos.Y));

end;

procedure TMainForm.miShowCursorClick(Sender: TObject);

begin

HideKbCursor := (Sender as TMenuItem).Checked;

end;

procedure TMainForm.SetKbKursorShowState(const Hide: Boolean);

begin

bHideCursor := Hide;

miShowCursor.Checked := not bHideCursor;

if Hide then

HideCursor(CursorPos)

else

ShowCursor(CursorPos, CurGameState[ActivePlayer].Color);

SynchronizeMenuItems;

end;

procedure TMainForm.SetHighlight(const NotHihlight: Boolean);

var

LastPoint: TGamePoint;

begin

Self.fNotHihlightLastMove := NotHihlight;

LastPoint := PGamSeq[CurrentSeq].Point;

if NotHihlight = False then

ShowCursor(LastPoint, {LastColor}clYellow)

else

HideCursor(LastPoint);

SynchronizeMenuItems;

end;

procedure TMainForm.miHighlightClick(Sender: TObject);

begin

NotHighlightLastMove := (Sender as TMenuItem).CHecked;

end;

end.