МИНОБРНАУКИ РОССИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
«Московский государственный технологический университет «СТАНКИН»
(ФГАОУ ВО «МГТУ «СТАНКИН»)

Институт цифровых интеллектуальных систем
Кафедра компьютерных систем управления

Образовательная программа 15.03.04
«Автоматизация технологических процессов и производств»

«КРЕСТИКИ-НОЛИКИ» С ДОПОЛНИТЕЛЬНЫМ УСЛОВИЕМ

Курсовая работа
по дисциплине «Прикладное программирование»

Выполнил:
студент гр. АДБ-23-07 _____ _____ Ванькаев Д.А.
(дата) (подпись)

Принял:
к.т.н., доцент _____ _____ Пушков Р.Л.
(дата) (подпись)

Москва 2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Задача курсовой работы – разработка модифицированной версии классической игры «Крестики-Нолики» на языке **C++** с использованием библиотеки **MFC (Microsoft Foundation Classes)** для реализации графического интерфейса. Основной задачей является создание игрового процесса с расширенными правилами, направленными на повышение стратегической сложности и динамичности.

Библиотека MFC выбрана для реализации графического интерфейса благодаря интеграции с Windows API, наличию готовых компонентов для отрисовки элементов управления и поддержке обработки событий, необходимых для интерактивности.

Игра «Крестики-нолики» представляет собой соревнование двух игроков (обозначаемых символами X и O) на квадратном поле. В данной реализации используется поле фиксированного размера 10×10 клеток. Цель – первым построить непрерывную линию из 4 одинаковых символов по горизонтали, вертикали или диагонали. Особенность этой версии – инверсия: после каждых четырёх сделанных ходов все X-символы заменяются на O и наоборот, и у игроков меняются игровые символы.

Какие элементы подверглись модификации:

- Размер поля увеличен до 10×10 клеток (вместо классического 3×3).
- Победа достигается при построении непрерывной линии из **4 одинаковых символов** (горизонтальной, вертикальной или диагональной).

Что было добавлено:

- Каждые **4 хода** все символы на поле меняются местами: крестики (X) превращаются в нолики (O) и наоборот.
- Типы символов игроков также инвертируются, чтобы сохранить логику очередности ходов.

ГЛАВА 1. АЛГОРИТМЫ РЕАЛИЗАЦИИ

1.1 ИГРОВОЕ ПОЛЕ

Для работы с игровым полем мы работаем с пятью классами: TicTacToeTestDlg, TicTacBoard, CStartupDlg TicTacToeField и TicTacPlayer.

Классы взаимодействуют так: CStartupDlg используется для настройки игроков перед началом игры «Крестики-нолики». Оно вызывается из основного диалога TicTacToeTestDlg при нажатии кнопки «Запустить» и позволяет ввести имена игроков и выбрать их тип (человек или компьютер). При старте игры CStartupDlg создаёт объект TicTacBoard(10) и двух игроков с символами X и O. Поле (tttField) запоминает родительский диалог и при отрисовке опрашивает board для получения текущих символов. Игроки при своём ходе устанавливают символ на доску через board->SetCell(x,y, currentPlayer->GetCurrentCellType()). После каждого хода диалог вызывает метод TogglePlayer, который учитывает инверсию и переключает активного игрока. Эти данные передаются в TicTacToeTestDlg. CStartupDlg координирует работу всех компонентов: UI-поле, доски и игроков.

TicTacBoard – хранит состояние игрового поля и содержит логику проверки победы. В классе определяется двумерный динамический массив клеток cells, размер поля boardsize и флаг bVictory (факт победы). Методы класса обеспечивают установки символов, проверку допустимости хода и окончание игры.

TicTacPlayer – базовый класс для игрока, содержит имя и текущий символ (X или O). Производные классы TicTacHumanPlayer и TicTacComputerPlayer реализуют разные способы выбора хода. У игроков есть метод SetupPlayer(name, symbol), который задаёт имя и начальный символ игрока. В диалоге игроку присваивается указатель на доску (SetBoard), что позволяет игроку взаимодействовать с состоянием поля.

TicTacToeField – компонент MFC (наследник CStatic или подобного) для отображения поля и приёма нажатий мыши. Он содержит ссылку на диалог CTicTacToeTestDlg как родителя и рисует клетки, используя данные из TicTacBoard. При клике вычисляется координата клетки, вызывается SetCell и далее происходит проверка результата хода. Этот класс отвечает только за пользовательский интерфейс поля.

CTicTacToeTestDlg – главное диалоговое окно приложения. Хранит указатели на TicTacBoard* board, на двух игроков (player1, player2, currentPlayer), счётчик ходов moveCount, флаг bGameInProgress и элемент поля tttField (типа CTicTacToeField). Диалог управляет жизненным циклом игры: создание доски и игроков, запуск/остановка игры, переключение текущего

игрока, инверсия после 4 ходов, а также обновление интерфейса (надпись «Ход игрока: …» и перерисовка поля).

В классе TicTacBoard хранится двумерный массив cells размера boardsize×boardsize типа CellType (CellType_X, CellType_O, CellType_Empty). Конструктор выделяет память под массив и инициализирует все клетки значением CellType_Empty. Деструктор освобождает память, удаляя каждую строку и затем сам массив указателей. Есть метод ClearBoard(), сбрасывающий все клетки в пустые, который может вызываться при перезапуске игры. Метод SetCell(xpos, ypos, ct) устанавливает символ ct в указанную клетку (строка ypos, столбец xpos). Метод CheckLegal(x, y) возвращает true, если координаты в пределах поля и клетка пуста (Empty).

Проверка победы производится методами IsRowMade(row), IsColumnMade(col), IsDiagMade() и IsBoardFull() (поле заполнено).

- **IsRowMade(row)**: по заданной строке перебираются все непрерывные участки длины 4 (от i=0 до boardsize-4). Берётся символ current = cells[row][i]. Если он пустой, пропускаем. Иначе проверяется, совпадают ли четыре последовательные клетки (cells[row][i+j]) с current. Если да, устанавливаем bVictory = true и возвращаем true.
- **IsColumnMade(col)**: аналогично IsRowMade, но по столбцу. Перебираются участки длины 4 по вертикали: берётся cells[i][col] и проверяется блок из четырёх клеток по i+j. При обнаружении выигрыша устанавливается bVictory = true и возвращается true.
- **IsDiagMade()**: проверяет два типа диагоналей. Сначала «слева-направо-вниз»: для каждого возможного старта (i,j) ($0 \le i,j \le$ boardsize-4) берётся cells[i][j]. Если не пустая, проверяются четыре клетки по диагонали cells[i+k][j+k]. Если все равны current, устанавливаем bVictory = true и возвращаем true. Затем аналогично проверяется диагональ «справа-назад-вниз»: стартуя в каждой (i,j) с j от 3 до boardsize-1, проверяется cells[i+k][j-k]. Найденная последовательность из четырёх одинаковых символов по любой диагонали тоже означает победу (здесь bVictory устанавливается и возвращается true).
- **IsBoardFull()**: пробегает по всем клеткам и проверяет, нет ли хотя бы одного пустого (CellType_Empty). Если найдена пустая, возвращает false, иначе true. Она используется, чтобы определить окончание игры ничьей, когда поле полностью заполнено без победных линий.

Метод **CheckEndCondition()** объединяет эти проверки: он сбрасывает bVictory = false, затем для каждой строки и столбца вызывает IsRowMade(i) и IsColumnMade(i). Если хотя бы одна вернула true, останавливаемся и возвращаем true (есть победа). Иначе вызываем IsDiagMade(). Если и там true, возвращаем true. Если ни один выигрыш не найден, возвращаем результат IsBoardFull(). Таким образом, CheckEndCondition() возвращает true либо при победе, либо при заполнении поля (конец игры). Дополнительный метод **IsVictory()** просто возвращает флаг bVictory (факт того, что победа уже обнаружена). В поле bVictory заносится true только при нахождении выигрышной линии в одном из методов проверки.

Инверсия – ключевая особенность данной реализации. Каждый четвертый сделанный ход приводит к тому, что все X символы меняются на O, и наоборот, а у игроков меняются игровые символы. Всё это реализовано в методах диалога **TogglePlayer** и **InvertBoard** в классе TicTacToeTestDlg.

При переключении игрока после каждого хода вызывается TogglePlayer(). В этом методе сначала увеличивается счётчик ходов moveCount++. Далее проверяется условие if (moveCount % 4 == 0): если оно истинно (ходов сделали кратное 4), вызывается InvertBoard(). В методе **InvertBoard()** перебираются все клетки доски (два вложенных цикла for i,j < board->GetSize()). Для каждой клетки берётся текущий символ current = board->GetCell(j,i). Если это X, то в ту же клетку записывается O (SetCell(j,i,CellType_O)), а если это O, то записывается X. Пустые клетки остаются неизменными. Таким образом происходит полная смена всех символов на поле. Затем меняются символы у игроков: у player1 и player2 вызывается SetCellType(...), где текущий символ заменяется на противоположный (X→O, O→X). Наконец, поле tttField инвалидируется (Invalidate()), что вызывает перерисовку доски с новыми символами.

После инверсии в TogglePlayer меняется указатель currentPlayer на противоположного: если сейчас не player1, то делаем currentPlayer = player1, иначе – player2. Затем обновляется текстовая надпись с именем текущего игрока через UpdateName(). Заметим, что при инверсии «текущий игрок» после смены символов фактически оказывается тем же лицом, но с новым символом (т.е. если игрок ходил X, сделал четвертый ход и доска инвертировалась, дальше он уже ходит O). Механизм инверсии обеспечивает выполнение этого условия: после вызова InvertBoard() в том же методе TogglePlayer сразу происходит переключение указателя на игрока.

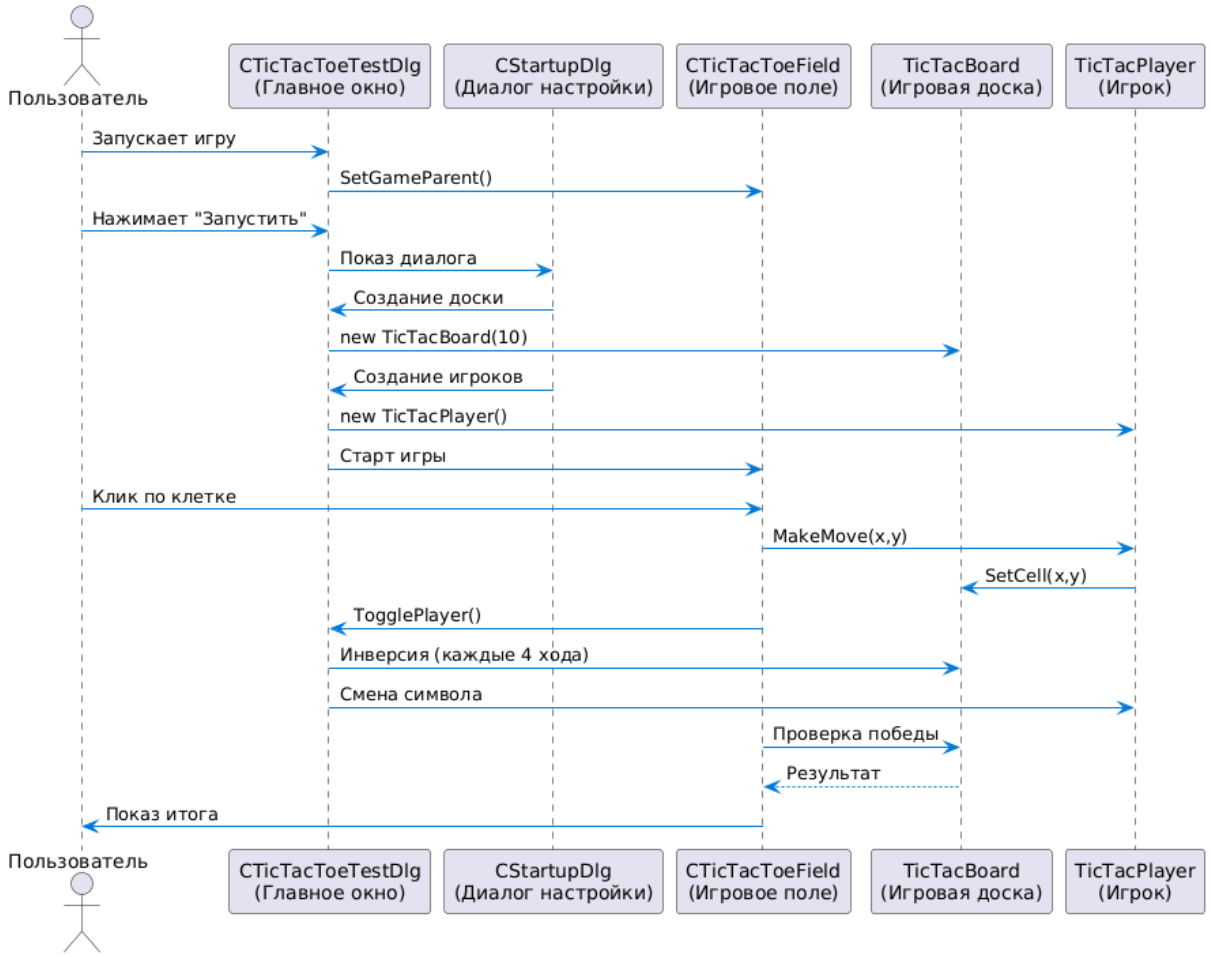

Рис 1. Диаграмма взаимодействия классов при работе с игровым полем.

1.2 КОМПЬЮТЕРНЫЙ ИГРОК (ИИ)

Класс **TicTacComputerPlayer** наследуется от базового **TicTacPlayer** и реализует логику автоматического игрока. Как правило, он содержит идентификатор своего знака (крестик или нолик) и указатель на текущую доску игры. Основной метод этого класса – MakeMove – переопределяется для выбора хода ИИ на основе заданной стратегии. В отличие от обычного игрока, **TicTacComputerPlayer** генерирует ход не по вводу пользователя, а по алгоритму: он анализирует ситуацию на доске, выбирает оптимальный ход (сначала проверяя простые случаи выигрыша или блока, а потом применяя симуляции Монте-Карло) и затем делает этот ход на объекте **TicTacBoard**.

Метод MakeMove класса **TicTacComputerPlayer** проходит несколько этапов, чтобы выбрать наилучший ход:

Поиск немедленной победы: перебираются все пустые клетки доски; для каждой клетке алгоритм условно ставит туда свой знак и проверяет, приводит ли это к победе ИИ. Если такой ход найден, ИИ сразу делает его и заканчивает ход. Такой приём гарантирует быструю победу, если она возможна. Это соответствует общему принципу: сначала проверяем, может ли компьютер выиграть одним ходом; если да – совершаем этот ход.

Блокировка победы противника: если выигрышного хода нет, алгоритм проверяет критические ходы оппонента. Опять перебираются все пустые клетки; ставится туда знак противника, после чего проверяется, выигрывает ли при этом противник. Если обнаруживается клетка, где соперник бы выиграл, то ИИ занимает эту клетку своим знаком, чтобы заблокировать выигрыш противника. Таким образом компьютер не даёт сопернику немедленно победить.

Оценка ходов методом Монте-Карло: если ни выигрышных, ни срочных блокирующих ходов нет, ИИ переходит к более тщательной оценке. Для каждого возможного хода он использует симуляции Монте-Карло: итеративно «доигрывает» партию до конца, начиная с данного хода, случайными ходами (за обоих игроков), и накапливает статистику результатов. Алгоритм подсчитывает, сколько раз после данного хода в итоге выигрывает ИИ, сколько раз проигрывает и сколько раз получается ничья. На основании этих данных вычисляется оценка хода (например, отношение побед к общему числу симуляций, или разность баллов: +1 за победу, −1 за проигрыш, 0 за ничью). Затем ИИ выбирает ход с наибольшей оценкой. Такая стратегия основана на идее, что выигрышная позиция даст высокую долю побед при случайных доигровках.

Случайный ход: если по каким-то причинам ни один ход не выделяется статистически (например, все оценки равны), ИИ в качестве запасного варианта выбирает один из оставшихся ходов случайным образом. Обычно это выполняется так: собирается список свободных клеток, затем случайным образом из него выбирается одна клетка. Такой случайный ход гарантирует, что ИИ всё же сделает ход, если более умная логика не дала результата.

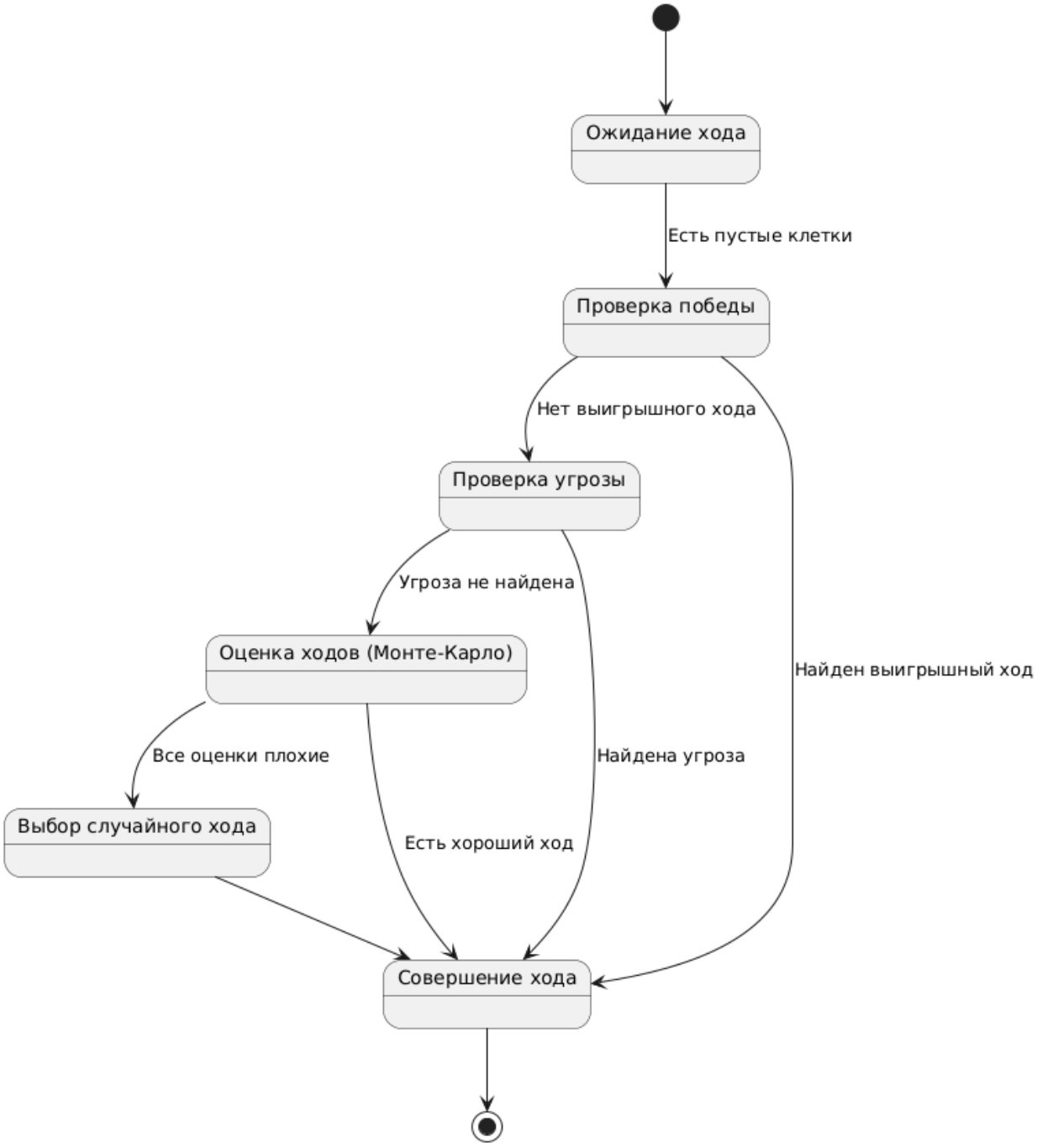

Рис 2. Диаграмма состояний класса TicTacComputerPlayer.

Класс **TicTacBoardMonteCarloEvaluator** отвечает за проведение симуляций Монте-Карло и оценку потенциальных ходов. Его ключевые компоненты и особенности работы включают в себя пункты, как симуляция игр (и их количество), копирование доски, подсчёт результатов и интерпретация результата.

При симуляции игры после применения хода ИИ симулятор несколько раз (заданное число симуляций) «отыгрывает» оставшуюся часть партии. В ходе каждой симуляции оба игрока делают случайные (разрешённые) ходы по очереди до окончания игры (либо до победы одного из игроков, либо до заполнения доски). Как отмечено в описании метода Монте-Карло: «мы просто начинаем играть, совершая случайные ходы за обоих игроков, пока игра не закончится». Обычно при инициализации или в настройках указывается число симуляций для каждого хода (например, 100–1000). Чем больше симуляций, тем надёжнее статистика, но тем больше затрат по времени.

Для каждой симуляции создаётся копия текущего состояния доски и на неё предварительно делается проверяемый ход (двигаем фигуру ИИ в рассматриваемую клетку). Затем остальные ходы в симуляции выполняются случайно. После применения хода ИИ симулятор несколько раз (заданное число симуляций) «сыгрывает» оставшуюся часть партии. В ходе каждой симуляции оба игрока делают случайные (разрешённые) ходы по очереди до окончания игры (либо до победы одного из игроков, либо до заполнения доски). После каждой симуляции определяется результат: выиграл ли ИИ (его знак создал требуемую выигрышную последовательность), выиграл ли соперник, или возникла ничья (нет свободных клеток и победителя нет). В простейшей схеме каждой симуляции даётся балл: +1 за победу ИИ, −1 за победу противника, 0 за ничью. По итогам всех симуляций для данного хода вычисляется суммарный результат или доля побед ИИ.

Интерпретация результата: если доля побед ИИ по данному ходу высока (или суммарный счёт положителен), этот ход считается перспективным. Обратная ситуация – частые проигрыши – говорит о плохом ходе. Класс **TicTacComputerPlayer** затем выбирает ход с наилучшей статистикой.

В итоге, **TicTacBoardMonteCarloEvaluator** многократно случайно «доигрывает» партии от позиции, предполагающей заданный ход, и возвращает оценку ходов на основании частоты побед/поражений. С точки зрения реализации, этот класс может содержать метод вроде EvaluateMove(board, pos, player), который запускает цикл симуляций, используя для каждой симуляции экземпляр доски **TicTacBoard** и случайных игроков.

Также стоит отметить **TicTacRandomPlayer.** Это вспомогательный класс «случайного» игрока, наследник **TicTacPlayer**, который в каждый ход выбирает одну из доступных клеток случайно. Его единственная цель – служить симулятором при Монте-Карло. При симуляции ИИ и его оппонент ходят не по умному алгоритму, а метод **TicTacRandomPlayer** подставляет для каждого из них случайные ходы. Это позволяет быстро эмулировать большое количество партий. **TicTacRandomPlayer** обычно содержит простую логику: получить список пустых клеток доски и выбрать одну из них случайно (например, с помощью функции rand()). Использование случайного игрока нужно лишь для оценки конкретной позиции: он моделирует, как могла бы быть сыграна партия, если дальше оба игрока играют наугад. Это упрощённое моделирование даёт статистически значимую оценку хода ИИ.

Рис 3. Диаграмма деятельности класса TicTacMonteCarloEvaluator.

ГЛАВА 2. ОПИСАНИЕ ИНТЕРФЕЙСА

При запуске программы пользователя встретит главное окно с тремя кнопками. Чтобы начать игру, нужно нажать кнопку «Запустить».

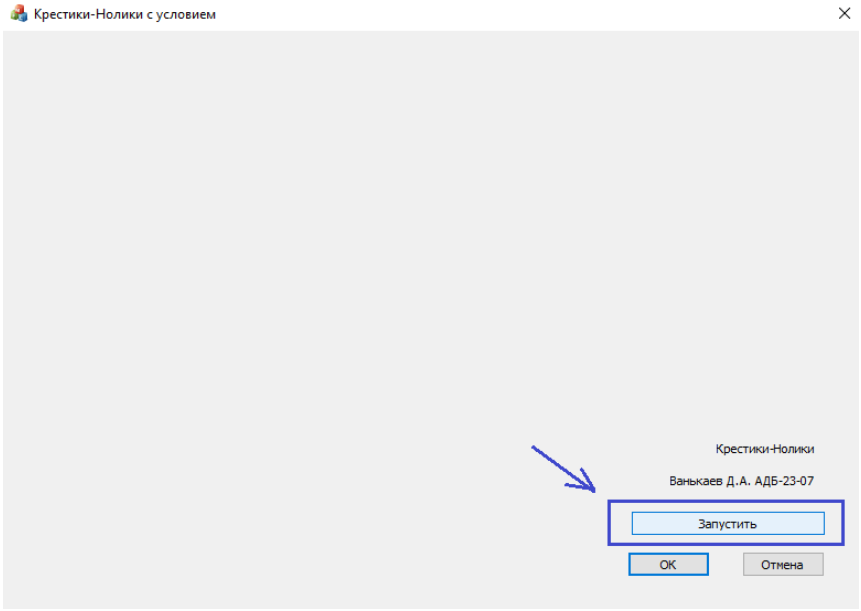

Рис 4. Стартовый экран.

Появится диалоговое окно с параметрами, которые нужно настроить, чтобы начать партию.

- Вписать имя двух игроков.
- Выбрать тип (либо по очереди будут играть два человека, либо человек с компьютером)

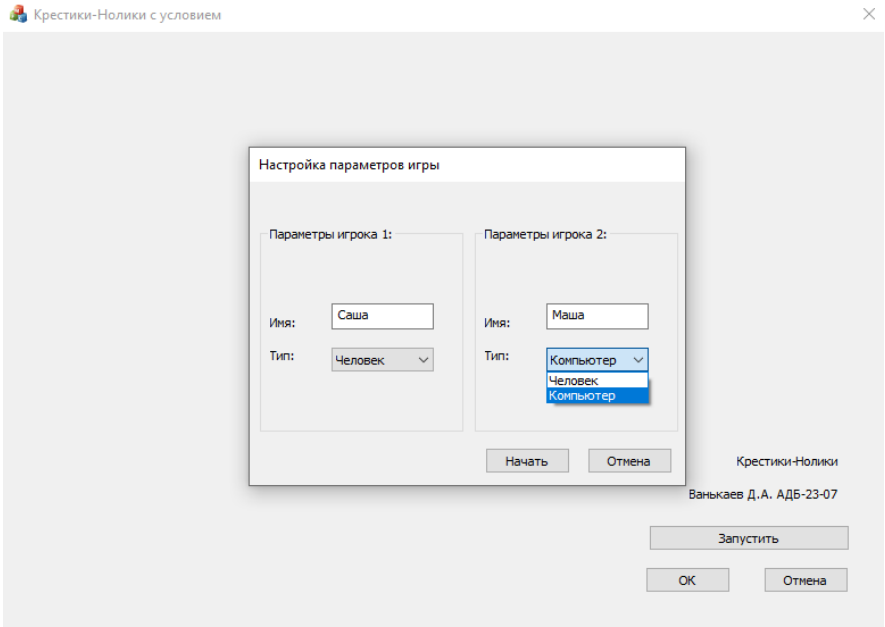

Рис 5. Параметры игры.

Нажав на кнопку «Начать» сгенерируется пустое поле 10x10 и ход предоставится первому игроку. Проводя курсором мыши по клеткам, они будут подсвечиваться голубым цветом. Левой кнопкой мыши можно занять клетку своим знаком.

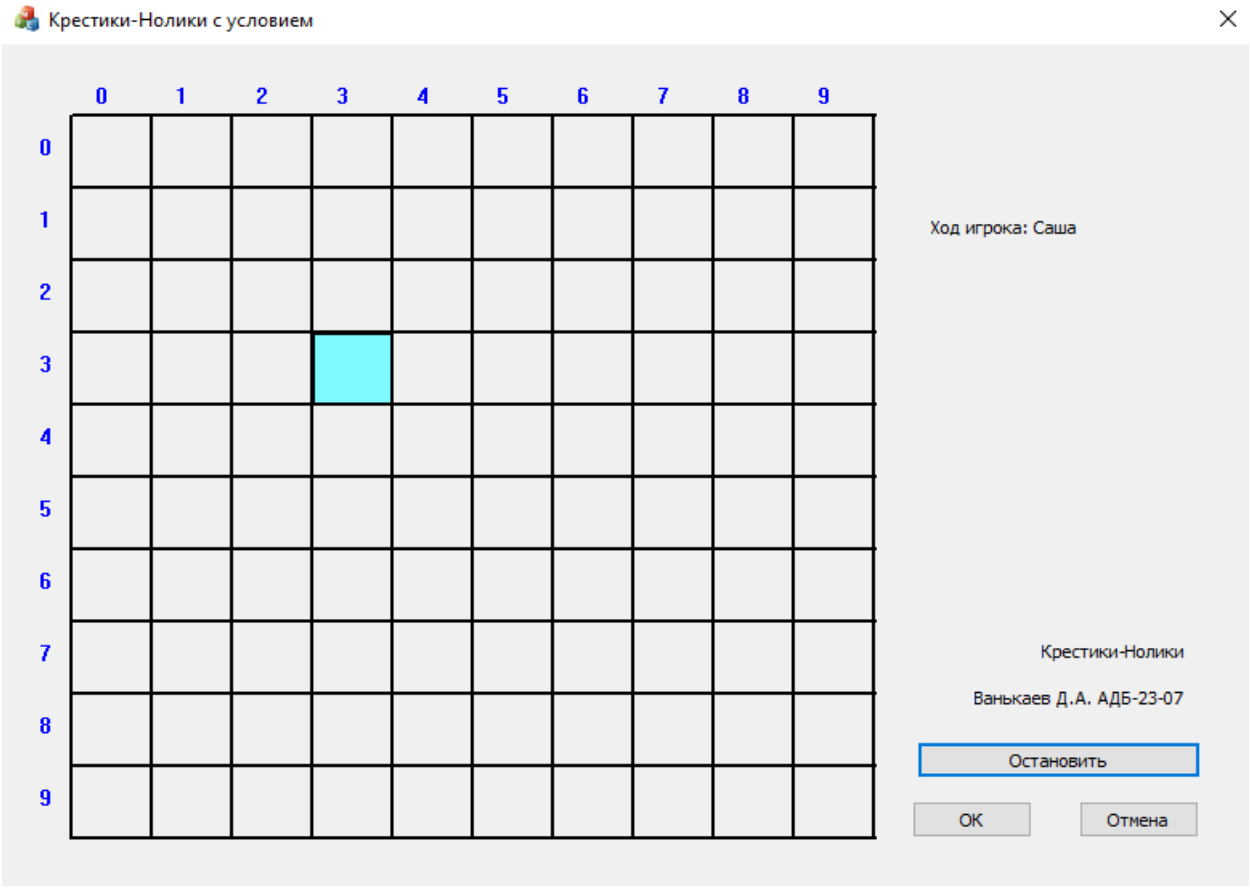

Рис 6. Созданное игровое поле.

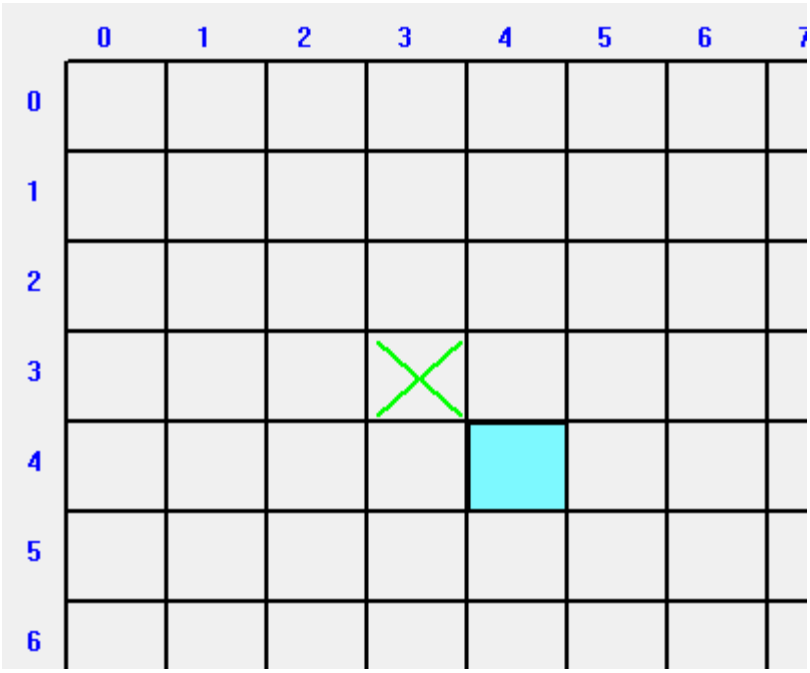

Рис 7. Крестик.

Справа обновляется текст с информацией, какой игрок ходит на данный момент. Не стоит забывать про упомянутый ранее алгоритм инверсии знаков на самом поле. Первым ходил игрок Саша, второй ходила Маша. После третьего хода, Маша ставит нолик на поле и после перехода хода к Саше игровое поле инвертирует знаки. Теперь Саша ставит нолики, а Маша – крестики.

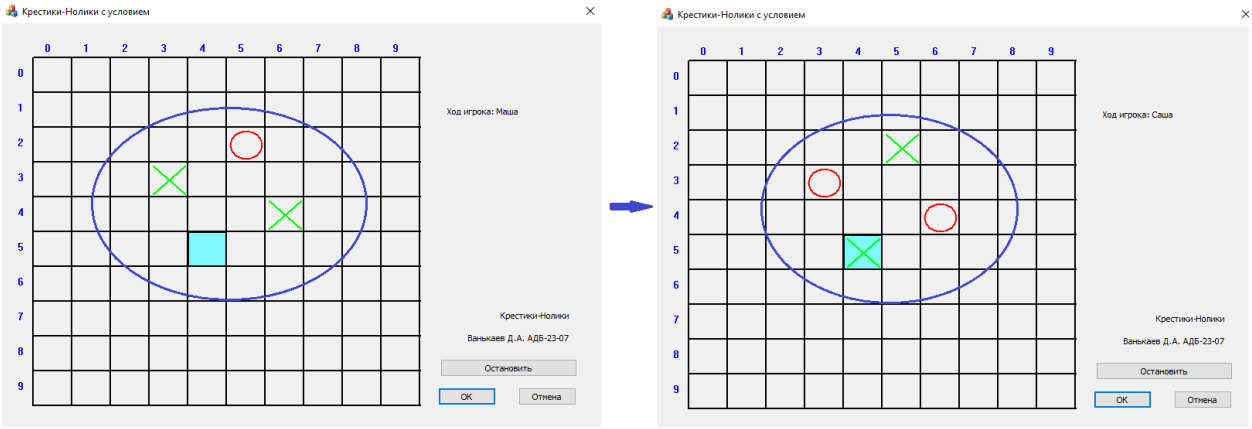

Рис 8. Демонстрация алгоритма инверсии на примере игры.

Для того, чтобы закончить партию, необходимо собрать ряд из **четырёх** знаков. Саша выиграл эту партию, потому появилось новое диалоговое окно.

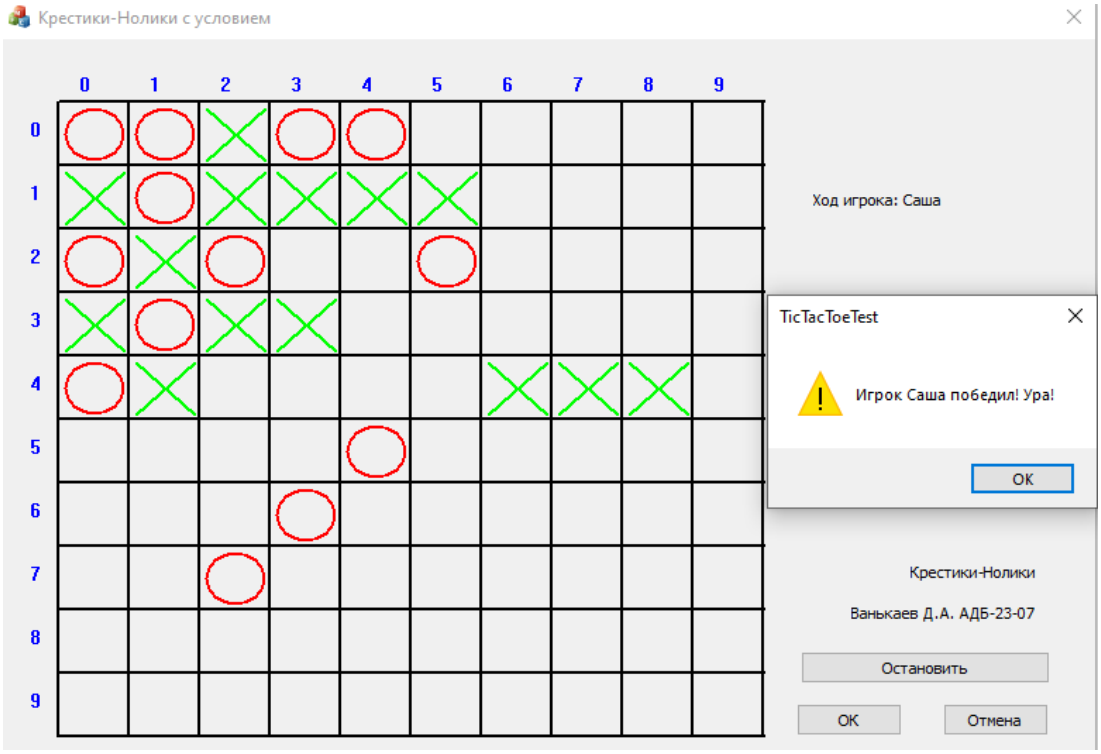

Рис 9. Игрок Саша победил.

Если же поле будет полностью заполнено, то появится окно об окончании игры ничьей.

Рис 10. Демонстрация ничьи.

ВЫВОД

В ходе выполнения курсовой работы была разработана графическая версия игры «Крестики-Нолики» на базе фреймворка MFC с нестандартными правилами и интеллектуальным оппонентом. Был спроектирован и реализован удобный GUI на базе MFC: поле фиксированного размера 10×10, элементы управления расположены логично, предусмотрена обработка всех возможных пользовательских событий.

Победа фиксируется при выстраивании непрерывной линии из четырёх одинаковых символов (X или O). Каждые четыре сделанных хода происходит «инверсия» всего игрового поля: все X превращаются в O, и наоборот. Эта особенность добавляет глубины стратегическому планированию, требуя от игрока учёта будущих перестановок символов. Для управления ходами компьютерного игрока реализован алгоритм Монте-Карло, анализирующий возможные варианты развития партии путём серии случайных симуляций. Эксперименты показали, что при разумном числе итераций (от 5 000 до 10 000 симуляций на ход) компьютер демонстрирует высокий уровень игры, успешно учитывая особенности инверсии поля и эффективно блокируя угрозы противника.

В целом, проделанная работа продемонстрировала эффективность сочетания инструментов MFC для создания GUI и методов статистического моделирования для реализации сильного компьютерного оппонента. Реализованный прототип может служить основой для дальнейших исследований в области алгоритмов принятия решений и разработки игр с нестандартными правилами.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Сидорина Т.Л. - Самоучитель Microsoft Visual Studio C++ и MFC/ Сидорина Т.Л..: БХВ-Петербург, 2009– 842 с.

2. Давыдов В. - Visual C++. Разработка Windows-приложений с помощью MFC и API-функций + Code/ Давыдов В. – БХВ-Петербург. – 576 с.

3. Стивен Прата - Язык программирования C++. Лекции и упражнения/ Стивен Прата (Stephen Prata).: Издательский дом «Вильямс», 2017– 1248 с.

4. Stroustrup B. / Страуструп Б. - Программирование. Принципы и практика с использованием C++ (2е издание) / Страуструп Б, Красикова И. В.. – М.: Издательский дом «Вильямс», 2016 – 1328 с.

5. Васильев, А.Н. Самоучитель C++ с примерами и задачами/ А.Н. Васильев. – СПб.: Наука и Техника, 2010. – 480с.

6. Липпман, С. Язык программирования C++. Вводный курс /С. Липпман, Б. Му, Ж. Лажойе. – М.: Издательский дом «Вильямс», 2007 – 896 с.