МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение

высшего профессионального образования

«**Вятский государственный университет**»

**(ФГБОУ ВПО «ВятГУ»)**

Факультет автоматики и вычислительной техники

Кафедра электронных вычислительных машин

РАЗРАБОТКА ИГРЫ “SUFFER, SNAKE!”

Пояснительная записка курсового проекта по дисциплине

«Компьютерная графика»

ТПЖА.230100.074 ПЗ

Разработал студент группы ИВТ-22 \_\_\_\_\_\_\_\_\_\_ /Нечаев А.А./

Руководитель

преподаватель кафедры ЭВМ \_\_\_\_\_\_\_\_\_\_\_\_\_/Вожегов Д.В./

Проект защищен с оценкой «\_\_\_\_\_\_\_\_\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(оценка) (дата)

Члены комиссии \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/

(подпись) (Ф.И.О)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ /\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/

Киров 2013

**Содержание**

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

2 Анализ аналогов…………………………………………………………... 4

2.1 Snake Classics………………………………………………….………….4

2.2 Python………………………………………………….………………….4

2.3 Snake-game………………………………………………….…………….4

2.4 Intelligent Multiplayer Snake..……………………………………….……4

2.5 Сравнительная характеристика..………………………………………..5

3 Обоснование актуальности темы..………………………………………..6

4 Расширенное техническое задание..………………………………………6

4.1 Требования к функциям программы……………………………………6

4.2 Пользовательский интерфейс..……………………………………….…6

4.3 Управление..……………………………………….……………………..7

4.4 Правила игры..……………………………………….…………………...7

4.5 Требования к входным и выходным данным..………………………….8

4.6 Системные требования..……………………………………….………...8

5 Схема взаимодействия функциональных блоков………………………..8

5.1 Пользовательский интерфейс…………………………………………...9

5.2 Работа с файлами…………………………………………………………9

5.3 Логика игрового процесса……………………………………………….9

5.4 Отображение игрового процесса………………………………………..9

5.5 Взаимодействие блоков………………………………………………….9

6 Разработка алгоритмов…………………………………………………...10

6.1 Логика игрового процесса……………………………………………...10

6.2 Отображение игрового процесса………………………………………14

7 Разработка программного обеспечения…………………………………15

7.1 Разработка модульной структуры……………………………………..15

7.2 Пользовательский интерфейс………………………………………….16

7.3 Работа с файлами……………………………………………………….17

7.4 Игровая логика………………………………………………………….18

7.5 Отображение игрового процесса………………………………………19

8 Экранные формы работы…………………………………………………20

9 Заключение………………………………………………………………..22

Приложение А. Код основной части программы…………………………23

Приложение Б. Код модуля, отвечающего за пользовательский

интерфейс…………………………………………………………………...30

Приложение В. Код модуля работы с файлами…………………………..41

Приложение Г. Код модуля игровой логики…………….………………..45

Приложение Д. Код модуля, выполняющего отображение

игрового процесса………………………………………………………….59

10 Список используемых источников…………………………………….71

**1 Введение**

Последнее время наблюдается значительный рост спроса на простые игры с двухмерной графикой. Среди всего прочего стоит выделить отдельную категорию – переделанные старые игры с более приятной глазу внешностью и, иногда, усовершенствованным (но не перегруженным новшествами) игровым процессом. В качестве разрабатываемого приложения была выбрана игра «Змейка», содержащая незначительные дополнения в виде различных препятствий и бонусов на игровом поле, а главное – режимом для двух игроков за одним компьютером.

Приложение может оказаться интересным как тем, кто давно знаком с компьютерными играми и в прошлом много времени потратил на классическую «Змейку», так и более молодой аудитории, по причине динамичного игрового процесса и наблюдающейся популярности игр такого рода.

Область применения данного приложения – развлечение людей, вне зависимости от наличия доступа к сети Интернет, а также с минимальной зависимостью от используемой платформы. Сейчас планируется выпуск только на персональных компьютерах, где в настоящее время происходит заметный рост использования отличных от Windows ОС на домашних компьютерах, а в будущем, вероятно, будет перенос на мобильные платформы.

**2 Анализ аналогов**

Было рассмотрено четыре аналогичных приложения [1-4]. Как оказалось, ни одно из них действительно не вносит никаких существенных дополнений в игровой процесс, и, за исключением рассмотренного в пункте 2.4, ни одно не предоставляет возможности для совместной игры.

**2.1 Snake Classics**

Типичный современная «Змейка»: отсутствуют какие-либо изменения в игровом процессе, несколько улучшена графика. Работает только на платформе Windows.

**2.2 Python**

В отличии от предыдущей игры, добавлено движение змейки в не ортогональных направлениях. Однако, управление производится с клавиатуры, что неудобно. Это – браузерная игра, а значит, работает на любой платформе.

**2.3 Snake-game**

Игра аналогична предыдущей, но управление змейкой производится с помощью мыши. Это гораздо удобнее, однако других новшеств игра не предлагает. Точно так же, она является браузерной.

**2.4 Intelligent Multiplayer Snake**

Свободный проект с возможностью игры по сети, созданием собственных серверов. Написана на языке Java, что позволяет запускать его на любой платформе. Кроме игры по сети с несколькими людьми, никаких усовершенствований в игровой процесс не вносит.

**2.5 Сравнительная характеристика**

В таблице 1 представлена сравнительная характеристика данных игр и разрабатываемого приложения.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Таблица 1 – Сравнительная характеристика аналогов | | | | | |
|  | Snake Classics | Python | Snake-game | Intelligent Multiplayer Snake | Разрабатываемое приложение |
| Наличие режима игры для нескольких человек | - | - | - | Только по сети Интернет | Только режим игры за одним компьютером |
| Возможность игры против искусственного интеллекта | - | - | - | + | + |
| Движение змейки в не ортогональных направлениях | - | + | + | - | - |
| Наличие различных бонусов, вносящих разнообразие в игровой процесс | - | - | - | - | + |
| Кроссплат-форменность | - | + | + | + | + |

Таким образом, разрабатываемое приложение имеет много преимуществ на фоне аналогов, так как предлагает режим игры для нескольких игроков за одним компьютером, а так же вносит дополнения в игровой процесс, не встречающиеся у аналогичных приложений. Возможность движения змейки в не ортогональных направлениях будет лишней, т.к. это сделает неудобным управление с клавиатуры, обязательное для режима двух игроков за одним компьютером.

**3 Обоснование актуальности темы**

В настоящее время практически все игры, реализованные на основе классической «Змейки», однообразны и не предлагают ничего нового, что было показано в разделе 2. Разрабатываемое приложение не только внесет разнообразие в классические правила, но и добавит режим игры для двух игроков за одним компьютером. Данная форма (одна клавиатура для двух человек) была выбрана неслучайно – именно она характерна для старых игр, а в настоящий момент существует большой спрос на них и их адаптации под современные платформы. Для этих же целей графическая часть приложения будет выполнена в пикселизованном стиле, без естественно выглядящих текстур и обилия оттенков одного цвета. Кроме того, в настоящее время растет количество пользователей ОС, основанных на ядре Linux, а количество игр для них до сих пор мало. Таким образом, с учетом современных тенденций, простоты и популярности классической «Змейки», а так же малой зависимости от платформы, разрабатываемое приложение актуально и оправдает себя.

**4 Расширенное техническое задание**

**4.1 Требования к функциям программы**

Должны присутствовать игровые режимы:

* одиночный;
* игрок против игрока;
* игрок против компьютера.

Игра проходит по правилам, описанным в пункте 4.4.

Для одиночного режима ведется таблица рекордов, в которой хранятся парами имя игрока и длина змейки, которую он получил за одну игровую сессию. Имя игрока запрашивается после окончания игры, один человек может присутствовать в таблице несолько раз.

В игре должен присутствовать подробный раздел справки, содержащий все отличия разработанной игры от классической змейки.

Игра принимает уровни из внешних файлов в формате TXT. Файл с таблицей рекордов зашифрован, чтобы его нельзя было менять вручную.

**4.2 Пользовательский интерфейс**

Доступные настройки:

* скорость игры;
* режим игры (классический, игра с другим человеком или компьютером);
* выбрать цвет змейки (змеек);
* игровое поле;
* отображение крови.

Во время игры пользователь должен видеть, какая длина у его змейки, и, в случае игры с другим человеком или компьютером – змейки противника; кроме того, можно вызвать экран паузы.

Таблица рекордов и разделы справки вызываются из главного меню; также, таблица рекордов отображается после окончания игры в одиночном режиме, если игрок сохраняет своё достижение.

* 1. **Управление**

Управление: первый игрок управляет клавишами W,A,S,D, соответствующими направлениям вверх, влево, вниз и вправо, второй – стрелками. Приоритет управления – у первого игрока. Выбор пунктов меню осуществляется на клавишу Enter. Нажатие клавиши ESC во время игры вызывает экран паузы, на котором пользователю будет предоставлен выбор: продолжить игру, или закончить её и вернуться в главное меню.

* 1. **Правила игры**

Начальная длина змейки равна трем. В начале игры змейки появляются в

разных половинах игрового поля, направление движения – вверх. Движение и рост змеек происходит по классическим правилам. Противоположные границы поля связаны: пересечение, например, верхней границы приводит к переносу на нижнюю. В режиме игры для двух игроков или против компьютера, если змейка игрока была пересечена собой или другой змейкой, она не выбывает из игрового процесса, а только теряет часть от места пересечения до хвоста. Столкновение головами приводит к ничье. На карте могут находится различные бонусы и препятствия:

* огонь, сжигающий 5 клеток у змейки за 5 тактов игры;
* замедление на 20 тактов игры;
* клетка-ловушка: если змейка проползает сквозь неё, то когда в ней окажется последняя клетка хвоста, от змейки отрывается 10 клеток;
* замедление змейки противника на 20 тактов игры;
* укорачивание змейки противника на 5 клеток;
* иммунитет к негативным действиям на 20 тактов игры;
* уменьшение необходимой для победы длины на 5 клеток змейке, подобравшей бонус.

В классическом режиме змейка может умереть от огня или ловушки. При этом, или самопересечении, игра заканчивается. В остальных режимах голова всегда остаётся, а конец игры наступает, как только одна из змеек становится заданной длины.

* 1. **Требования к входным и выходным данным**

Игра может принимать на вход уровни в формате TXT, в которых первой строкой написано название уровня, а далее в виде матрицы с соответствующими кодами (0 – пустая клетка, 1-7 – бонусы и препятствия в том же порядке, как в пункте 4.4) задано расположение клеток игрового поля. Текстуры игры хранятся в формате PNG.

Выходные данные – таблица рекордов в каком-либо шифрованном формате; это необходимо для того, чтобы препятствовать её ручному изменению.

* 1. **Системные требования**

Для запуска и работы программы необходимы, как минимум:

* операционная система – Windows XP и выше, либо дистрибутивы Linux на основе Ubuntu 10.04 и выше;
* процессор – любой процессор с тактовой частотой 1.7 ГГц и выше;
* 512 MБ оперативной памяти;
* клавиатура, монитор.

1. **Схема взаимодействия функциональных блоков**

Подлежащее разработке приложение было разбито на несколько функциональных блоков, что показано на рисунке 1. Далее представлено краткое описание каждого из блоков.

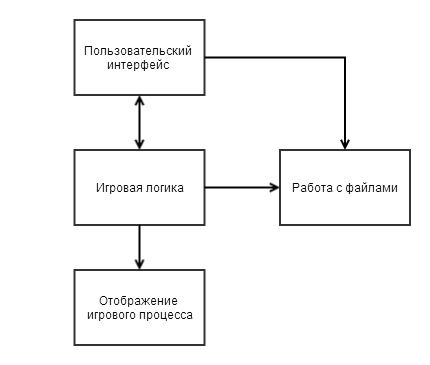


Рисунок 1 – Разбиение программы на функциональные блоки

**5.1 Пользовательский интерфейс**

Все модули этого блока отвечают за отображение каких-либо данных об игре: различных экранов меню, справки, информации о текущей игровой сессии во время игры.

**5.2 Работа с файлами**

Под файлами в данном случае подразумеваются уровни и таблица рекордов. Блок содержит подпрограммы для чтения и записи последней, а так же для загрузки уровней и их проверки в том случае, если пользователь добавил свои.

**5.3 Логика игрового процесса**

Данный блок содержит алгоритмы программного представления и обработки текущей игровой сессии: реакция на действия игроков (а также работа искусственного интеллекта) и взаимодействие с игровым полем.

**5.4 Отображение игрового процесса**

Здесь находятся все части программы, ответственные за рисование игрового поля, змеек, спецэффектов.

**5.5 Взаимодействие блоков**

Блок работы с файлами независим от всех остальных блоков: он только передает в них данные, считанные из файлов. Пользовательский интерфейс использует данные таблицы рекордов для отображения, а также требует информацию о всех уровнях, доступных для загрузки. Из пользовательского интерфейса (меню настройки игры, вызываемое перед стартом) настройки (какой уровень использовать, режим игры и т.д.) передаются в подпрограммы блока игровой логики. Рисование игрового процесса не зависит от блока работы с фалами, т.к. самостоятельно загружает все текстуры, но требует данных об игровом процессе, а также вызывает различные подпрограммы, связанные с пользовательским интерфейсом.

**6 Разработка алгоритмов**

Разработка пользовательского интерфейса проста с алгоритмической точки зрения, поэтому не требует никаких дополнительных пояснений. В подпрограммах блока 5.2 единственный сложный момент – шифровка и дешифровка таблицы рекордов – будет реализован с использованием Java Cryptography Architecture (JCA), что делает их разработку также тривиальной. Поэтому далее рассмотрены только алгоритмы для блоков 5.3 и 5.4.

**6.1 Логика игрового процесса**

Этот раздел – основной в разрабатываемом приложении. Два главных алгоритма здесь – выполнение действий в игровом цикле и работа искусственного интеллекта. Первый показан на рисунках 2 и 3, второй – на рисунке 4 соответственно.

В классическом режиме в игровом цикле следует обрабатывать только три вещи:

* пересечение с клеткой, содержащей еду, бонус или препятствие;
* самопересечение;
* применение каких-либо эффектов, которые длятся несколько тактов (замедление, горение и т.д.).

В режимах, где на поле находятся две змейки, к этому добавляются проверки на:

* пересечение змеек;
* воздействие одной змейки на другую через какие-либо бонусы;
* конец игры в зависимости от длины змеек.

Работа искусственного интеллекта основана на волновом алгоритме. Каждый раз, когда змейке необходимо принять решение, создается соответствующая полю игры матрица, где часть клеток помечена как непроходимые, а часть – как доступные для посещения. Затем из головы змейки пускается волна, в результате чего в каждой клетке записаны либо расстояние до неё, либо метка о недоступности этой клетки. После этого некоторые клетки поля выбираются как приоритетные, и среди них выбирается ближайшая. По заполненной волной матрице восстанавливается обратный путь, и змейка делает ход. Если все приоритетные клетки оказываются недоступными, то змейка продолжает движение в том же направлении, что и раньше.

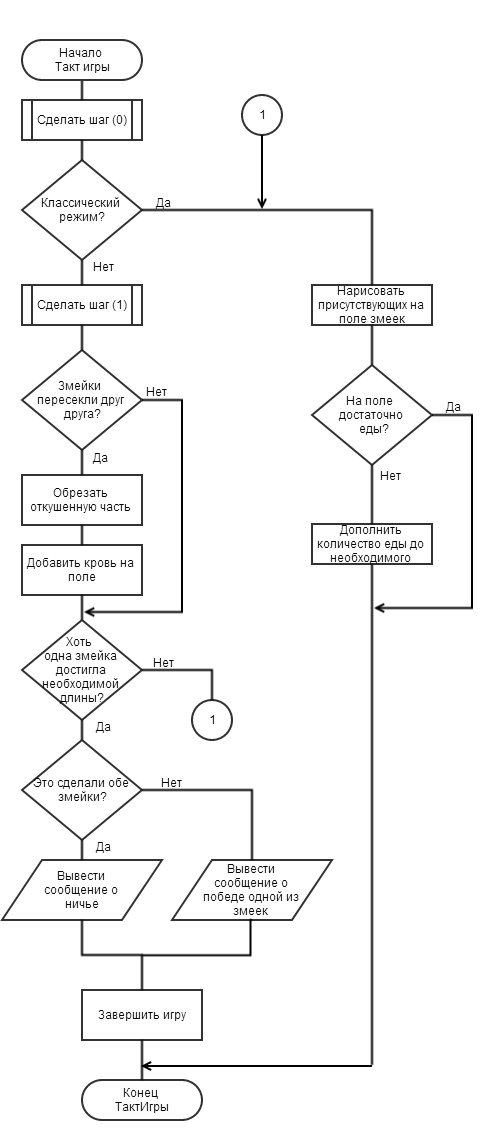


Рисунок 2 – Действия, выполняемые в игровом цикле

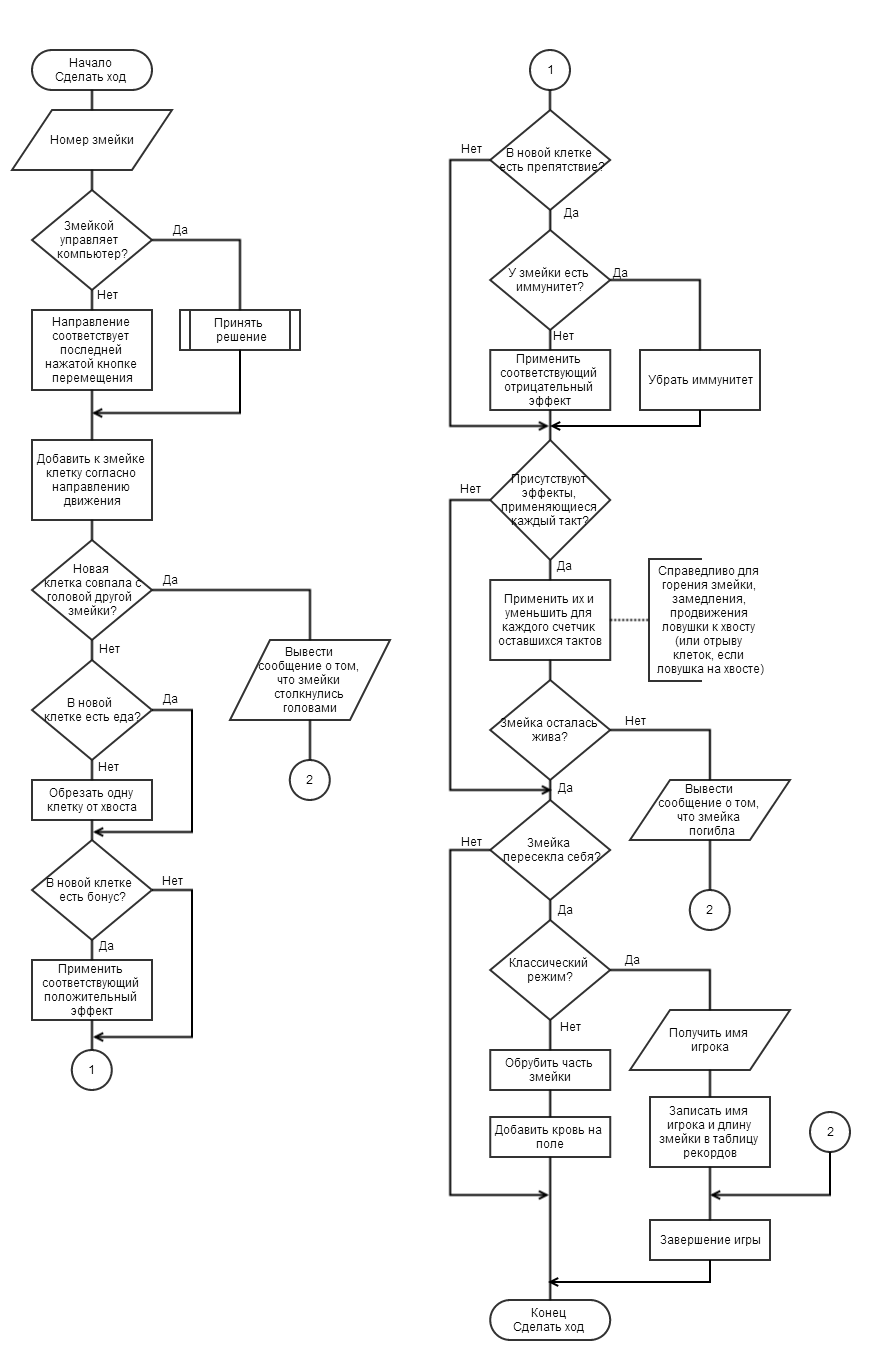


Рисунок 3 – Действия змейки на каждом ходу

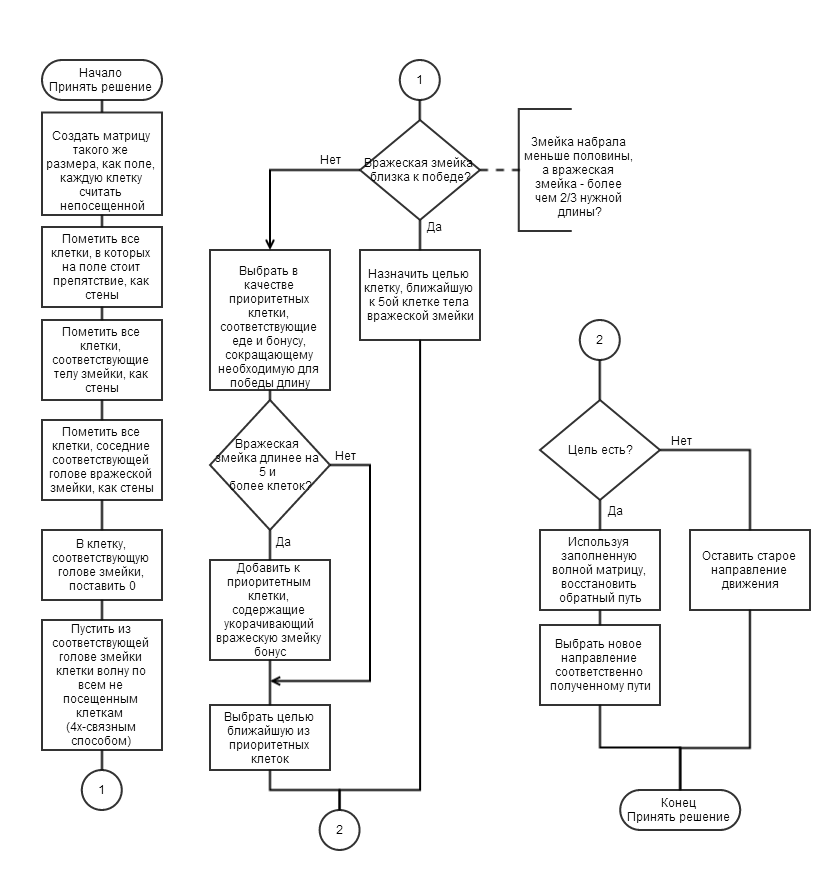


Рисунок 4 – Алгоритм принятия решения искусственным

интеллектом

**6.2 Отображение игрового процесса**

Все клетки будут рисоваться путем простой подстановки необходимых текстур на нужное место игрового поля, причем, чтобы каждый раз не перерисовывать поле, будет использоваться матрица логических переменных, обозначающих, изменилась ли на данном такте игры клетка, т.е. нужно ли её перерисовать. Это не вызывает сложности. Однако, рисование анимированных блоков путем смены заранее заготовленных кадров выглядит достаточно некрасиво, что особенно заметно при большом количестве одинаковых текстур (огня, пятен крови и т.д.). Поэтому огонь и кровь рисуются иначе: создается система частиц, никак не взаимодействующих друг с другом, но меняющих при необходимости положение в пространстве и цвет. Поскольку нет необходимости обрабатывать взаимодействие частиц, алгоритмы в данном случае сводятся к способу выбора новых координат для частицы (цвет выбирается случайным образом из заданного массива цветов).

Огонь:

* фиксированное смещение по высоте; если точка выходит за переделы отведенной клетки, она возвращается в её низ;
* абсцисса выбирается случайным образом в пределах нужной клетки.

Кровь:

* при создании абсцисса выбирается случайным образом в выбранном радиусе;
* ордината при создании также выбирается случайно, но если расстояние получившейся точки от центра пятна оказывается больше заданного радиуса, ордината меняется таким образом, чтобы точка оказалась на окружности;
* при дальнейшем рисовании координаты не меняются.

**7 Разработка программного обеспечения**

В соответствие с разработанными функциональными блоками и алгоритмами была создана модульная структура приложения, а затем согласно ей – полностью код программы. В качестве языка программирования был выбран язык Java, благодаря кроссплатформенности и богатому набору стандартных средств.

**7.1 Разработка модульной структуры**

На рисунке 5 показана модульная структура программы. Над стрелками подписаны подпрограммы, с помощью которых данные передаются из одного модуля в другой.

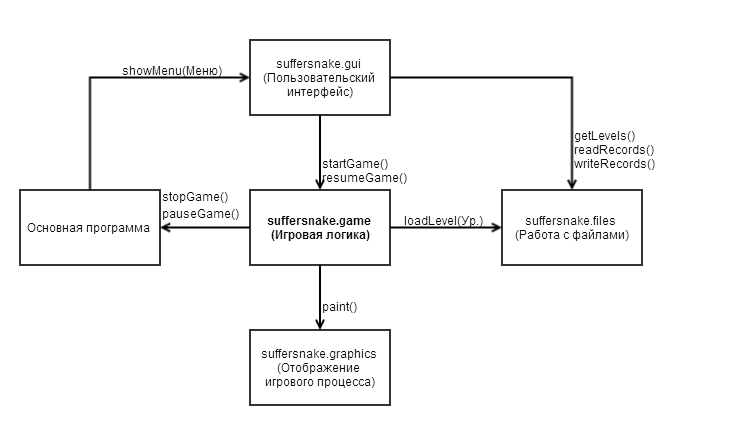
****

Рисунок 5 – Модульная структура программы

Согласно схеме, начало игры, показ меню и дальнейшая работа программы начинается с модуля основной программы. Его код показан в приложении А.

Помимо описанных на схеме разработанных модулей, использовались следующие стандартные библиотеки и пакеты Java [5]:

* AWT – для обработки изображений, событий клавиатуры;
* java.io, javax.imageio – для работы с файлами;
* java.util – для работы с коллекциями объектов, генерации случайных чисел;
* JCA – для шифрования и дешифровки таблицы рекордов;
* Swing – для отображения игрового экрана в стандартном для ОС окне.

**7.2 Пользовательский интерфейс**

При запуске приложения пользователь видит главное меню, содержащее следующие пункты, показанные на рисунке 6.

|  |
| --- |
| 1. Начать игру  * Выбор режима * Выбор цвета змеек * Выбор уровня * Выбор скорости игры * Включение (выключение) рисования крови  1. Рекорды 2. Справка 3. Выход |

Рисунок 6 – Структура меню

Чтобы внешность и управление больше соответствовали старым играм, стандартные компоненты Java Swing не используются. Были разработаны классы MenuLabel, MenuComponent, Menu. Первый реализует текстовую метку, которую умеет только отображаться в определенном месте на экране. MenuComponent – его наследник, и у него появляются новые характеристики: он умеет быть активным и обрабатывать различные посылаемые ему события клавиатуры. Все остальные компоненты (кнопки, списки и т.д.) являются его наследниками. Последний класс – Menu – содержит в себе массивы объектов типа MenuLabel и MenuComponent, и организует взаимодействие между ними. Все меню в игре наследуются от него. Указанная на рисунке 6 подпрограмма showMenu показывает на экране определенное меню, при этом направляя в него все события клавиатуры. Для большей стилизации под старые игры, при рисовании меню используется сторонний свободно распространяемый шрифт.

Разработанный исходный код показан в приложении Б (из созданных наследников класса Menu показан только реализующий главное меню, так как принцип использования созданных компонентов в других меню точно такой же).

**7.3 Работа с файлами**

Две главных задачи, которые выполняет данный модуль – чтение уровней и чтение/запись таблицы рекордов.

Чтение уровней организовано следующим образом: программа просматривает все файлы в определенной заданной в исходном коде папке и пытается получить из них данные уровня по определенному шаблону. В том случае, если при попытке получить уровень из файла не возникает никаких исключений, выбранный файл считается подходящим, и перед началом игры можно выбрать уровень из него.

Таблица рекордов, согласно расширенному техническому заданию, должна хранится в шифрованном виде. Для этого все табличные данные преобразуются в строку, затем шифруются и пишутся в файл. Для чтения используется обратный алгоритм. Поскольку JCA предоставляет богатый API для криптографической защиты данных, в качестве алгоритма было выбрано тройное DES-шифрование. Оно работает быстро на используемом объеме данных и не поддается взлому тем, кто не имеет в своем распоряжении секретный ключ. Разумеется, никакого труда не составляет достать ключ из байт-кода Java и с его помощью дешифровать таблицу рекордов, а затем сохранить её в том же формате, но от рядовых пользователей эта защита более чем достаточна.

Показанные на рисунке 5 подпрограммы вызывают выполнение соответствующих алгоритмов:

* readRecords, writeRecords – чтение и запись таблицы рекордов;
* getLevels – получение списка доступных уровней;
* loadLevel – загрузка выбранного уровня.

В приложении В показан разработанный исходный код модуля.

**7.4 Игровая логика**

Были разработаны четыре класса: Point, Snake, Field и KeyHandler. Первый используется для описания точки поля, а так же содержит в себе статические методы для получения соседних точек и направления перемещения из одной точки в другую (это необходимо, так как противоположные границы поля связаны).

Класс змейки содержит в себе методы, реализующие алгоритмы на рисунках 3 и 4, и различные вспомогательные методы. Тело змейки представлено в виде списка точек.

Класс Field содержит исключительно статические поля и методы (так как значения полей единственны во всей программе, и, кроме того, при такой реализации его проще использовать из других модулей). Он содержит в себе описания всех констант, связанных с содержимым клеток, а главное – три матрицы:

* environment – содержит в себе загружаемые из файла уровня данные (бонусы и препятствия);
* blocks – содержит блоки, которые не зависят от уровня (еда, змейки);
* changed – матрица логического типа, показывающая, изменялись ли клетки на данном такте игры (используется при перерисовке).

Последний класс, KeyHandler, содержит в себе все методы, обрабатывающие нажатия клавиш на клавиатуре.

Метод, реализующий алгоритм на рисунке 2, оставлен в модуле основной программы. Методы startGame, stopGame, pauseGame, resumeGame класса Main выполняют все необходимые действия для начала, остановка, приостановки и возобновления игры соответсвтенно. Исходный код модуля остальной игровой логики представлен в приложении Г.

**7.5 Отображение игрового процесса**

Основной созданный в данном модуле класс – класс Block, соответствующий одной клетке поля и содержащий метод отображения этой клетки на игровом экране. Также разработан класс Painter, содержащий в себе статические (по тем же причинам, что класс Field из модуля 7.4) методы для рисования клеток на игровом поле.

Для рисования огня и крови был создан класс ParticleSystem, реализующий описанные в разделе 6.2 алгоритмы. Единственная возможная проблема здесь – большая нагрузка на процессор (рисование выполняется на ЦП) при большом количестве крови. Поэтому количество пятен крови на поле ограничено пятьюдесятью.

Чтобы избежать моргания экрана, все игровое окно сначала рисуется на изображении в памяти компьюетра, и только потом – на экране.

Текстуры из сторонних источников при разработке не использовались.

Исходный код модуля представлен в приложении Д.

**8 Экранные формы работы**

На рисунках 7, 8 и 9 показаны главное меню, меню настроек игры и игровой процесс соответственно. На рисунке 10 показан запуск игры в Linux Mint 15 (остальные сделаны во время работы приложения в ОС Windows).

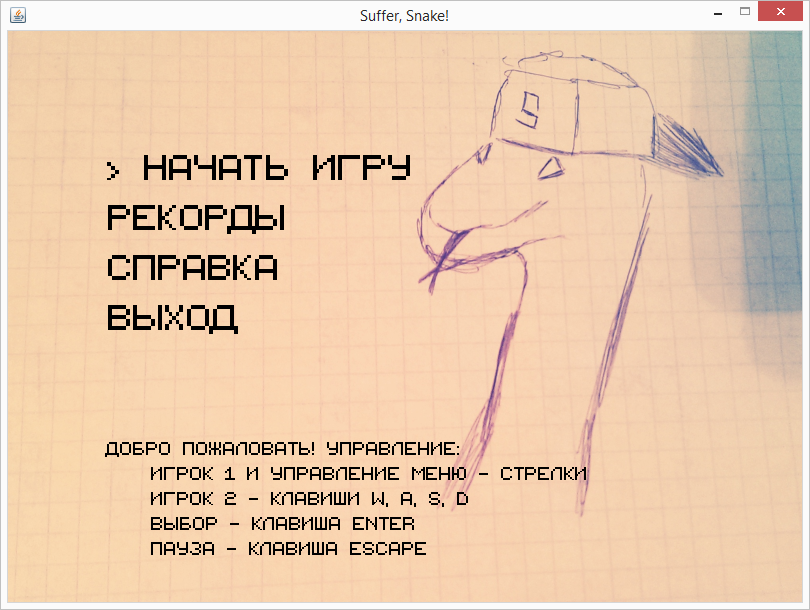


Рисунок 7 – Главное меню

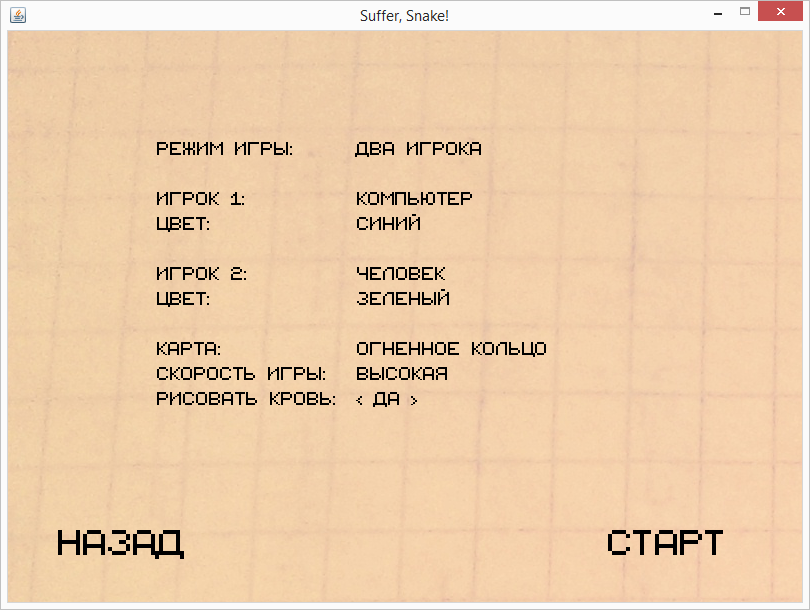


Рисунок 8 – Меню настройки

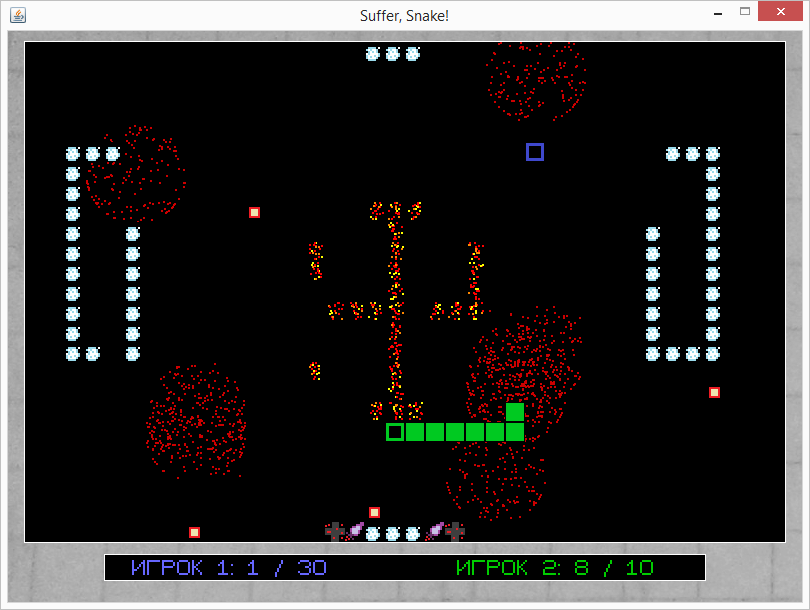


Рисунок 9 – Игровой процесс



Рисунок 10 – Запуск в Linux Mint 15

**9 Заключение**

В начале была поставлена задача по разработке игры «Suffer, snake!», подробно описанная в расширенном техническом задании. В результате анализа аналогичных приложений была обоснована актуальность выбранной темы, а также выявлены преимущества и недостатки аналогов, которые были учтены при разработке. Далее программа была разбита на несколько функциональных блоков, каждый из который отвечает за определённую часть её работы, и, в соответствии с выбранным разбиением, были разработаны необходимые алгоритмы, впоследствии реализованные в определенных программных модулях проекта.

Разработанное приложение полностью соответствует поставленному техническому заданию. Приложение тестировалось на компьютере с двухъядерным (что неважно, так как все выполняется в одном потоке) процессоре частотой 2,2 ГГц, и в наихудшем случае (максимальная скорость игры, компьютер против компьютера, много крови и огня) загрузка составила 35%; потребляемая память – не более 80 Мб. Это отвечает заявленным требованиям, но в будущем можно оптимизировать работу программы ради снижения потребляемых ресурсов. Кроме того, искусственный интеллект оказался, с одной стороны, слишком сильным (не попадает в препятствия никогда, кроме безысходных ситуаций), а с другой стороны – слишком не агрессивным по отношению к сопернику. Также, так как приложение разработано на платформе Java, в будущем его можно будет перенести на мобильные платформы с минимальными временными и денежными затратами, где оно (особенно на планшетах при использовании совместной игры) может оказаться даже более актуальным, чем на стационарных компьютерах.

Приложение А

(обязательное)

**Код основной части программы**

**А.1 Код главного класса Main**

**package** suffersnake;

**import** suffersnake.files.LevelLoader;

**import** suffersnake.files.RecordsIO;

**import** suffersnake.game.Field;

**import** suffersnake.game.KeyHandler;

**import** suffersnake.graphics.Painter;

**import** suffersnake.graphics.Block;

**import** suffersnake.gui.\*;

**import** javax.imageio.ImageIO;

**import** javax.swing.\*;

**import** java.awt.event.ActionEvent;

**import** java.awt.event.ActionListener;

**import** java.awt.image.BufferedImage;

**import** java.io.File;

**import** java.io.IOException;

**public class** Main {

**private static** MainMenu mainMenu; *// Главное меню*

**private static** RecordsMenu recordsMenu; *// Страница отображения рекордов*

**private static** HelpMenu helpMenu; *// Страница справки*

**private static** GameSettingsMenu gameSettingsMenu; *// Меню настройки игры*

**private static** ErrorScreen errorScreen; *// Экран ошибки*

**private static** LoadingScreen loadingScreen; *// Экран загрузки*

**private static** PauseScreen pauseScreen; *// Экран паузы*

**private static** GameOverScreen gameOverScreen; *// Экран окончания игры*

**private static** BufferedImage mainMenuBackground;

**private static** BufferedImage otherMenusBackground;

**private static** MainFrame mainFrame;

**private static boolean** classicMode;

**private static boolean** needToDrawBlood;

**protected static** Timer gameTickTimer;

**private static boolean** gamePaused;

*// Время одного такта перерисовки. Один такт игры (движения змеек)*

*// должен быть кратен этому значению!*

**public static final int** PAINT\_TICK\_TIME = 25;

**private static int** curTickTime;

**private static int** timeFromLastMove;

**public static final int** INITIAL\_FOOD\_COUNT = 4; *// Начальное количество еды*

**public static final int** LENGTH\_TO\_WIN = 30; *// Длина, необходимая для победы*

**public static final int** FIRING\_BLOCKS = 5; *// Количество клеток, сжигаемых одним костром*

**public static final int** SLOWING\_TIME = 20; *// Время замедления, в тактах игры*

**public static final int** TRAPPING\_BLOCKS = 10; *// Количество клеток, убираемых ловушкой/мечом*

**public static final int** IMMUNITY\_TIME = 20; *// Время иммунитета, в тактах игры*

**public static final int** EASIER\_WIN\_DEC = 5; *// Уменьшение необходимой для победы длины бонусом*

**public static void** main(String[] args) {

mainFrame = **new** MainFrame();

**try** {

mainMenuBackground = ImageIO.read(

**new** File(**"resource/img/mainMenuBackground.png"**)

);

otherMenusBackground = ImageIO.read(

**new** File(**"resource/img/otherMenusBackground.png"**)

);

} **catch** (Exception e) {

*// Даже для экрана ошибки нужны фоновые изображения*

e.printStackTrace();

System.exit(0);

}

errorScreen = **new** ErrorScreen(mainFrame.gameScreen, otherMenusBackground, mainFrame);

loadingScreen = **new** LoadingScreen(mainFrame.gameScreen, otherMenusBackground, mainFrame);

loadingScreen.setVisible();

mainFrame.setVisible(**true**);

**try** {

RecordsIO.readRecords();

} **catch** (Exception e) {

errorScreen.setVisible(**true**, **"Не удалось прочитать или создать файл рекордов."**);

**return**;

}

**try** {

LevelLoader.init();

} **catch** (Exception e) {

errorScreen.setVisible(**true**, **"Не удалось загрузить файлы уровней."**);

**return**;

}

**try** {

helpMenu = **new** HelpMenu(mainFrame.gameScreen, otherMenusBackground, mainFrame);

} **catch** (IOException e) {

errorScreen.setVisible(**true**, **"Не удалось загрузить все необходимые ресурсы."**);

**return**;

}

mainMenu = **new** MainMenu(mainFrame.gameScreen, mainMenuBackground, mainFrame);

recordsMenu = **new** RecordsMenu(mainFrame.gameScreen, otherMenusBackground, mainFrame);

gameSettingsMenu = **new** GameSettingsMenu(mainFrame.gameScreen, otherMenusBackground, mainFrame);

pauseScreen = **new** PauseScreen(mainFrame.gameScreen, otherMenusBackground, mainFrame);

gameOverScreen = **new** GameOverScreen(mainFrame.gameScreen, otherMenusBackground, mainFrame);

mainMenu.setVisible();

gameTickTimer = **new** Timer(PAINT\_TICK\_TIME, **new** ActionListener() {

@Override

**public void** actionPerformed(ActionEvent e) {

gameTick();

}

});

}

**public static void** showMainMenu() {

mainMenu.setVisible();

}

**public static void** showRecordsMenu() {

recordsMenu.setVisible();

}

**public static void** showHelpMenu() {

helpMenu.setVisible();

}

**public static void** showGameSettingsMenu() {

gameSettingsMenu.setSelectedComponent(0);

gameSettingsMenu.setVisible();

}

**public static void** startGame(

**boolean** isClassic, *// Если true, то классический режим*

**boolean** isAi1, *// Является ли первая змейка ИИ*

**int** colorIndex1, *// Цвет первой змейки согласно Painter.SNAKE\_COLORS*

**boolean** isAi2, *// Является ли вторая змейка ИИ*

**int** colorIndex2, *// Цвет второй змейки согласно Painter.SNAKE\_COLORS*

**int** level, *// Выбранный уровень согласно LevelLoader.levelNames*

**int** speed, *// Множитель для PAINT\_TICK\_TIME*

**boolean** \_needToDrawBlood *// Если true, то кровь отображается*

) {

loadingScreen.setVisible();

classicMode = isClassic;

needToDrawBlood = \_needToDrawBlood;

**try** {

Block.loadImages();

LevelLoader.loadLevel(level);

} **catch** (Exception e) {

errorScreen.setVisible(**false**, **"Не удалось загрузить ресурсы игры."**);

**return**;

}

*// Создание змеек*

Field.createSnakes(isAi1, isAi2, colorIndex1, colorIndex2);

*// Забыть информацию о предыдущей сессии*

Field.clearBlocks();

Painter.initialize(mainFrame.gameScreen, otherMenusBackground, mainFrame);

*// Убрать кровь из предыдущей сессии*

Painter.clearBlood();

*// Все блоки первый раз нужно нарисовать*

Painter.setNeedToDrawAllBlocks(**true**);

*// Необхдимо, чтобы окно восринимало нажатия клавиш*

KeyHandler.bindKeyListener(mainFrame);

curTickTime = PAINT\_TICK\_TIME \* (speed \* 2 < 3 ? 3 : speed \* 2); *// Множитель 2 - слишком мало*

timeFromLastMove = 0;

Field.generateInitialFood(isClassic ? 1 : INITIAL\_FOOD\_COUNT);

gamePaused = **false**;

gameTickTimer.start();

}

**protected static void** gameTick() {

**if** (gamePaused) {

**return**;

}

timeFromLastMove += PAINT\_TICK\_TIME;

**if** (timeFromLastMove >= curTickTime) {

timeFromLastMove = 0;

*// Забыть инфомрацию о предыдущем положении змеек и получить новые*

Field.clearAllBlocksWoutFood();

Field.snakes[0].makeStep();

**if** (!classicMode) {

Field.snakes[1].makeStep();

**int** eater;

**if** ((eater = Field.processIntersection()) != -1) {

Painter.addBlood(

Field.snakes[eater].getHead().getX(),

Field.snakes[eater].getHead().getY()

);

}

**if** (Field.snakes[0].isHasSlowedAnEnemy()) {

Field.snakes[1].makeSlowed();

} **else if** (Field.snakes[1].isHasSlowedAnEnemy()) {

Field.snakes[0].makeSlowed();

}

**if** (Field.snakes[0].isHasShortedAnEnemy()) {

Field.snakes[1].makeShorter();

} **else if** (Field.snakes[1].isHasShortedAnEnemy()) {

Field.snakes[0].makeShorter();

}

*// Проверка на завершение игры*

**if** (Field.snakes[0].hasWon() && Field.snakes[1].hasWon()) {

stopGame(**"Змейки достигли нужной длины одновременно. Ничья."**, **false**);

**return**;

} **else if** (Field.snakes[0].hasWon()) {

stopGame(String.format(

**"%s игрок победил!"**,

(String)Painter.SNAKE\_COLORS[Field.snakes[0].getColorIndex()][1]

), **false**);

**return**;

} **else if** (Field.snakes[1].hasWon()) {

stopGame(String.format(

**"%s игрок победил!"**,

(String)Painter.SNAKE\_COLORS[Field.snakes[1].getColorIndex()][1]

), **false**);

}

}

*// Добавляет еду на поле, если её не хватает*

Field.normalizeFoodCount();

*// Добавляет тела змеек в массив Field.blocks (для отрисовки)*

Field.dropSnakesToBlocks();

}

**if** (gameTickTimer.isRunning()) {

*// Непустые клетки также необходимо обновить (кроме того,*

*// бонусы или препятсвтия могли исчезнуть на данном такте)*

Field.dropEnvironmentToBlocks();

*// Рисование текущего состояния игры*

Painter.paint();

}

}

**public static void** pause() {

gamePaused = **true**;

pauseScreen.setVisible();

}

**public static void** resume() {

gamePaused = **false**;

pauseScreen.killKeyListeners();

Painter.initialize(mainFrame.gameScreen, otherMenusBackground, mainFrame);

Painter.setNeedToDrawAllBlocks(**true**);

KeyHandler.bindKeyListener(mainFrame);

}

**public static void** stopGame(String message, **boolean** needToAddRecord) {

gameTickTimer.stop();

gameOverScreen.setVisible(message, needToAddRecord);

}

**public static void** stopGame() {

gameTickTimer.stop();

mainMenu.setVisible();

}

**public static boolean** isClassicMode() {

**return** classicMode;

}

**public static boolean** isNeedToDrawBlood() {

**return** needToDrawBlood;

}

**public static void** showSaveRecordError() {

errorScreen.setVisible(**false**, **"Не удалось добавить рекорд."**);

}

}

**A.2 Код класса окна приложения**

**package** suffersnake;

**import** javax.swing.\*;

**import** java.awt.\*;

**import** java.awt.image.BufferedImage;

**import** java.io.File;

**import** java.io.FileInputStream;

**import** java.io.IOException;

**public class** MainFrame **extends** JFrame {

**public** BufferedImage gameScreen;

**public static final int** SCREEN\_WIDTH = 800;

**public static final int** SCREEN\_HEIGHT = 600;

**public static** Font MENU\_FONT;

**public static final float** MENU\_FONT\_SIZE = 18;

**public static final** Color MENU\_FONT\_COLOR = Color.BLACK;

**public** MainFrame() {

**super**();

gameScreen = **new** BufferedImage(

SCREEN\_WIDTH,

SCREEN\_HEIGHT,

BufferedImage.TYPE\_4BYTE\_ABGR

);

Graphics2D g2d = gameScreen.createGraphics();

g2d.setColor(Color.BLACK);

g2d.fillRect(0, 0, SCREEN\_WIDTH, SCREEN\_HEIGHT);

setSize(SCREEN\_WIDTH, SCREEN\_HEIGHT);

setDefaultCloseOperation(WindowConstants.EXIT\_ON\_CLOSE);

setResizable(**false**);

setTitle(**"Suffer, Snake!"**);

setLocationRelativeTo(**null**);

loadMenuFont();

}

**private void** loadMenuFont() {

**try** {

MENU\_FONT = Font.createFont(

Font.TRUETYPE\_FONT,

**new** FileInputStream(**new** File(**"resource/MenuFont.ttf"**))

);

} **catch** (FontFormatException | IOException e) {

MENU\_FONT = UIManager.getFont(**"Label.font"**);

}

MENU\_FONT = MENU\_FONT.deriveFont(MENU\_FONT\_SIZE);

}

@Override

**public void** paint(Graphics g) {

g.drawImage(gameScreen, 0, 0, **null**);

}

}

**А.3 Код класса, содержащего задающие клавиши управления константы**

**package** suffersnake;

**import** java.awt.event.KeyEvent;

**public class** KeyMap {

**public static final int** PLAYER1\_UP = KeyEvent.VK\_UP;

**public static final int** PLAYER1\_RIGHT = KeyEvent.VK\_RIGHT;

**public static final int** PLAYER1\_DOWN = KeyEvent.VK\_DOWN;

**public static final int** PLAYER1\_LEFT = KeyEvent.VK\_LEFT;

**public static final int** PLAYER2\_UP = KeyEvent.VK\_W;

**public static final int** PLAYER2\_RIGHT = KeyEvent.VK\_D;

**public static final int** PLAYER2\_DOWN = KeyEvent.VK\_S;

**public static final int** PLAYER2\_LEFT = KeyEvent.VK\_A;

**public static final int** SELECT = KeyEvent.VK\_ENTER;

**public static final int** PAUSE = KeyEvent.VK\_ESCAPE;

}

Приложение Б

(обязательное)

**Код модуля, отвечающего за пользовательский**

**интерфейс**

**Б.1 Код класса MenuLabel**

*/\*\**

*\* Класс описывает невыделяемую статическую надпись на экане игры*

*\*/*

**package** suffersnake.gui;

**import** suffersnake.MainFrame;

**import** java.awt.\*;

**public class** MenuLabel {

**protected int** x, y;

**protected** String labelText;

**private boolean** visible;

**public** MenuLabel(**int** x, **int** y) {

**this**.x = x;

**this**.y = y;

visible = **true**;

}

**public** MenuLabel(**int** x, **int** y, String labelText) {

**this**.x = x;

**this**.y = y;

visible = **true**;

**this**.labelText = labelText;

}

**public void** drawComponent(Graphics2D gameScreenG2D) {

**if** (!visible) {

**return**;

}

gameScreenG2D.setColor(MainFrame.MENU\_FONT\_COLOR);

gameScreenG2D.setFont(MainFrame.MENU\_FONT);

gameScreenG2D.drawString(labelText, x, y);

}

**public void** setLabelText(String value) {

labelText = value;

}

**public void** setVisible(**boolean** value) {

visible = value;

}

**public boolean** isVisible() {

**return** visible;

}

}

**Б.2 Код класса MenuComponent**

*/\*\**

*\* Класс описывает абстрактный активный компонент меню:*

*\* он может быть выделен и должен уметь обрабатывать некоторые*

*\* события клавиатуры*

*\*/*

**package** suffersnake.gui;

**import** java.awt.event.KeyEvent;

**public abstract class** MenuComponent **extends** MenuLabel {

**public** MenuComponent(**int** x, **int** y) {

**super**(x, y);

}

**public abstract void** setSelected(**boolean** selected);

**public abstract void** handleKeyEvent(KeyEvent e);

}

**Б.3 Код класса MenuButton**

*/\*\**

*\* Класс описывает компонент, выполняющий какое-либо действие,*

*\* если нажать на клавишу выбора, когда компонент выделен*

*\*/*

**package** suffersnake.gui;

**import** suffersnake.KeyMap;

**import** suffersnake.MainFrame;

**import** java.awt.\*;

**import** java.awt.event.KeyEvent;

**public class** MenuButton **extends** MenuComponent {

**private** String

unselectedButtonText,

selectedButtonText;

**private** Menu.MenuAction action;

**public** MenuButton(**int** x, **int** y, String buttonText, Menu.MenuAction action) {

**super**(x, y);

unselectedButtonText = buttonText;

selectedButtonText = String.format(**"> %s"**, buttonText);

**this**.labelText = unselectedButtonText;

**this**.action = action;

}

@Override

**public void** drawComponent(Graphics2D gameScreenG2D) {

gameScreenG2D.setFont(

MainFrame.MENU\_FONT.deriveFont(MainFrame.MENU\_FONT\_SIZE\*2)

);

gameScreenG2D.setColor(MainFrame.MENU\_FONT\_COLOR);

gameScreenG2D.drawString(labelText, x, y);

}

@Override

**public void** setSelected(**boolean** selected) {

labelText = selected ? selectedButtonText : unselectedButtonText;

}

@Override

**public void** handleKeyEvent(KeyEvent e) {

**if** (e.getKeyCode() == KeyMap.SELECT && action != **null**) {

action.execute();

}

}

**public void** setAction(Menu.MenuAction action) {

**this**.action = action;

}

}

**Б.3 Код класса MenuList**

*/\*\**

*\* Класс описывает комопнент, содержащий в себе*

*\* список каких-либо значений и позволяющий выбрать одно из них*

*\*/*

**package** suffersnake.gui;

**import** suffersnake.KeyMap;

**import** suffersnake.MainFrame;

**import** java.awt.\*;

**import** java.awt.event.KeyEvent;

**public class** MenuList **extends** MenuComponent {

**private** String[] values;

**private int** selectedIndex;

**private** Menu parent;

**private boolean** circular = **true**;

**private boolean** selected;

**private** MenuList otherMenuList = **null**;

**public** MenuList(**int** x, **int** y, String[] values, Menu parent) {

**super**(x, y);

**this**.values = values;

**this**.parent = parent;

selectedIndex = 0;

}

*/\*\**

*\* По умолчанию выбор осуществляется циклически, однако*

*\* (например для перелистывания страниц справки) это не*

*\* нужно*

*\*/*

**private void** selectPreviousValue() {

**if** (--selectedIndex < 0) {

**if** (circular) {

selectedIndex = values.length - 1;

} **else** {

selectedIndex++;

**return**;

}

}

**if** (otherMenuList != **null** && selectedIndex == otherMenuList.getSelectedIndex()) {

**if** (--selectedIndex < 0) {

selectedIndex = values.length - 1;

}

}

}

**private void** selectNextValue() {

**if** (++selectedIndex == values.length) {

**if** (circular) {

selectedIndex = 0;

} **else** {

selectedIndex--;

**return**;

}

}

**if** (otherMenuList != **null** && selectedIndex == otherMenuList.getSelectedIndex()) {

**if** (++selectedIndex == values.length) {

selectedIndex = 0;

}

}

}

**public int** getSelectedIndex() {

**return** selectedIndex;

}

**public void** setSelectedIndex(**int** value) {

selectedIndex = value;

}

@Override

**public void** setSelected(**boolean** selected) {

**this**.selected = selected;

}

@Override

**public void** handleKeyEvent(KeyEvent e) {

**switch** (e.getKeyCode()) {

**case** KeyMap.PLAYER1\_LEFT:

selectPreviousValue();

**break**;

**case** KeyMap.PLAYER1\_RIGHT:

selectNextValue();

**break**;

}

parent.drawOnGameScreen();

}

@Override

**public void** drawComponent(Graphics2D gameScreenG2D) {

**if** (!isVisible()) {

**return**;

}

gameScreenG2D.setFont(MainFrame.MENU\_FONT);

gameScreenG2D.setColor(MainFrame.MENU\_FONT\_COLOR);

gameScreenG2D.drawString(

String.format((selected ?

!circular && selectedIndex == 0 ?

**"%s >"** :

!circular && selectedIndex == values.length - 1?

**"< %s"** :

**"< %s > "** :

**"%s"**), values[selectedIndex]), x, y

);

}

**public void** setOtherMenuList(MenuList otherMenuList) {

**this**.otherMenuList = otherMenuList;

}

**public void** setNotCircular() {

circular = **false**;

}

}

**Б.4 Код класса TextField**

*/\*\**

*\* Класс описывает компонент, в который можно*

*\* вводить какой-либо текст*

*\*/*

**package** suffersnake.gui;

**import** suffersnake.MainFrame;

**import** java.awt.\*;

**import** java.awt.event.KeyEvent;

**public class** TextField **extends** MenuComponent {

**private** Menu parent;

**private boolean** selected;

**private static final int** MAX\_LENGTH = 20;

**private** String text;

**public** TextField(**int** x, **int** y, Menu parent) {

**super**(x, y);

**this**.parent = parent;

text = **""**;

}

@Override

**public void** setSelected(**boolean** selected) {

**this**.selected = selected;

}

@Override

**public void** handleKeyEvent(KeyEvent e) {

*// Вводимый текст ограничен буквами латинского алфавита и пробелом*

**int** keyCode = e.getKeyCode();

**if** (keyCode >= KeyEvent.VK\_A && keyCode <= KeyEvent.VK\_Z && text.length() < MAX\_LENGTH) {

text += (**char**)(**'A'** + keyCode - KeyEvent.VK\_A);

} **else if** (keyCode == KeyEvent.VK\_SPACE) {

text += **' '**;

} **else if** (keyCode == KeyEvent.VK\_BACK\_SPACE && text.length() > 0) {

text = text.substring(0, text.length() - 1);

}

parent.drawOnGameScreen();

}

@Override

**public void** drawComponent(Graphics2D gameScreenG2D) {

**if** (!isVisible()) {

**return**;

}

gameScreenG2D.setFont(MainFrame.MENU\_FONT.deriveFont(MainFrame.MENU\_FONT\_SIZE \* 2));

gameScreenG2D.setColor(MainFrame.MENU\_FONT\_COLOR);

gameScreenG2D.drawString(

String.format((selected ? **"%s \_"** : **"%s"**), text), x, y

);

}

**public** String getText() {

**return** text;

}

}

**Б.5 Код класса абстрактного меню**

**package** suffersnake.gui;

**import** suffersnake.KeyMap;

**import** suffersnake.MainFrame;

**import** javax.swing.\*;

**import** java.awt.\*;

**import** java.awt.event.KeyAdapter;

**import** java.awt.event.KeyEvent;

**import** java.awt.event.KeyListener;

**import** java.awt.image.BufferedImage;

**import** java.util.ArrayList;

**public class** Menu {

*// Интерфейс необходим для связывания элементов меню и определенных*

*// частей программы*

**public interface** MenuAction {

**public void** execute();

}

**private** JFrame parent;

**protected** BufferedImage background;

*// Изображние, на котором будет производиться отрисвока*

**protected** BufferedImage gameScreen;

**protected** ArrayList<MenuLabel> labels;

**protected** ArrayList<MenuComponent> components;

**public static final int** SELECT\_NEXT\_COMPONENT\_KEY = KeyMap.PLAYER1\_DOWN;

**public static final int** SELECT\_PREVIOUS\_COMPONENT\_KEY = KeyMap.PLAYER1\_UP;

**private int** selectedComponent;

**public** Menu(BufferedImage gameScreen, BufferedImage background, JFrame parent) {

**this**.gameScreen = gameScreen;

**this**.background = background;

**this**.parent = parent;

labels = **new** ArrayList<>();

components = **new** ArrayList<>();

selectedComponent = -1;

}

**public void** killKeyListeners() {

**for** (KeyListener curKeyListener: parent.getKeyListeners()) {

parent.removeKeyListener(curKeyListener);

}

}

**public void** setVisible() {

*// При переключение в новое меню старые обработчики событий*

*// должны быть убраны*

killKeyListeners();

parent.addKeyListener(**new** KeyAdapter() {

@Override

**public void** keyPressed(KeyEvent e) {

handleKeyEvent(e);

}

});

drawOnGameScreen();

}

*// Рисует все элементы меню на изобрадении, которое в будущем*

*// будет выведено в игровое окно*

**public void** drawOnGameScreen() {

Graphics2D g2d = gameScreen.createGraphics();

**if** (background != **null**) {

g2d.drawImage(background, 0, 0, **null**);

} **else** {

g2d.setColor(Color.WHITE);

g2d.fillRect(0, 0, MainFrame.SCREEN\_WIDTH, MainFrame.SCREEN\_HEIGHT);

}

**for** (MenuComponent curComponent: components) {

curComponent.drawComponent(g2d);

}

**for** (MenuLabel curLabel: labels) {

curLabel.drawComponent(g2d);

}

parent.repaint();

}

*/\*\**

*\* Следующие два метода необходимы, т.к. выбор компонентов осуществеляется*

*\* циклически и только среди видимых компонентов*

*\*/*

**private void** selectNextComponent() {

**if** (components.size() == 0)

**return**;

components.get(selectedComponent).setSelected(**false**);

**do** {

**if** (++selectedComponent == components.size()) {

selectedComponent = 0;

}

} **while** (!components.get(selectedComponent).isVisible());

components.get(selectedComponent).setSelected(**true**);

}

**private void** selectPreviousComponent() {

**if** (components.size() == 0)

**return**;

components.get(selectedComponent).setSelected(**false**);

**do** {

**if** (--selectedComponent == -1) {

selectedComponent = components.size()-1;

}

} **while** (!components.get(selectedComponent).isVisible());

components.get(selectedComponent).setSelected(**true**);

}

**public void** setSelectedComponent(**int** componentNumber) {

**if** (componentNumber >= components.size() || componentNumber < 0) {

**return**;

}

**if** (selectedComponent != componentNumber) {

**if** (selectedComponent != -1) {

components.get(selectedComponent).setSelected(**false**);

}

selectedComponent = componentNumber;

components.get(selectedComponent).setSelected(**true**);

}

}

**public void** handleKeyEvent(KeyEvent e) {

**switch** (e.getKeyCode()) {

**case** SELECT\_NEXT\_COMPONENT\_KEY:

selectNextComponent();

**break**;

**case** SELECT\_PREVIOUS\_COMPONENT\_KEY:

selectPreviousComponent();

**break**;

**default**:

**if** (components.size() != 0) {

components.get(selectedComponent).handleKeyEvent(e);

**return**;

}

}

drawOnGameScreen();

}

}

**Б.6 Код класса главного меню**

**package** suffersnake.gui;

**import** suffersnake.Main;

**import** javax.swing.\*;

**import** java.awt.event.KeyEvent;

**import** java.awt.image.BufferedImage;

**public class** MainMenu **extends** Menu {

**public** MainMenu(BufferedImage gameScreen, BufferedImage background, JFrame parent) {

**super**(gameScreen, background, parent);

labels.add(**new** MenuLabel(100, 450, **"Добро пожаловать! Управление: "**));

labels.add(**new** MenuLabel(100, 475, **" Игрок 1 и управление меню - стрелки"**));

labels.add(**new** MenuLabel(100, 500, **" Игрок 2 - клавиши W, A, S, D"**));

labels.add(**new** MenuLabel(100, 525, **" Выбор - клавиша ENTER"**));

labels.add(**new** MenuLabel(100, 550, **" Пауза - клавиша ESCAPE"**));

components.add(**new** MenuButton(100, 175, **"Начать игру"**, **new** MenuAction() {

@Override

**public void** execute() {

Main.showGameSettingsMenu();

}

}));

components.add(**new** MenuButton(100, 225, **"Рекорды"**, **new** MenuAction() {

@Override

**public void** execute() {

Main.showRecordsMenu();

}

}));

components.add(**new** MenuButton(100, 275, **"Справка"**, **new** MenuAction() {

@Override

**public void** execute() {

Main.showHelpMenu();

}

}));

components.add(**new** MenuButton(100, 325, **"Выход"**, **new** MenuAction() {

@Override

**public void** execute() {

System.exit(0);

}

}));

setSelectedComponent(0);

}

}

Приложение В

(обязательное)  
**Код модуля работы с файлами**

**В.1 Сохранение и запись таблицы рекордов**

*/\*\**

*\* Класс реалиует чтение и запись таблицы рекордов с*

*\* использованием шифрования*

*\*/*

**package** suffersnake.files;

**import** suffersnake.game.Field;

**import** javax.crypto.Cipher;

**import** javax.crypto.SecretKey;

**import** javax.crypto.SecretKeyFactory;

**import** javax.crypto.spec.DESedeKeySpec;

**import** javax.xml.bind.DatatypeConverter;

**import** java.io.\*;

**public class** RecordsIO {

**public static** String[] names;

**public static int**[] scores;

**public static final** String RECORDS\_FILE\_NAME = **"records.sfs"**;

**public static final int** RECORD\_COUNT = 10;

**private static final** String SEPARATOR = **"\u0000"**;

**public static final** String EMPTY\_VALUE = **"-"**;

**private static final** String KEY = **"D938AA357A5BA96BF4C6B915F6CE08D1EA9D2AE7D5FF98CA"**;

**private static** SecretKey desKey;

**private static** Cipher desCipher = **null**;

*// Записывает хранящуюся в памяти таблицу рекордов*

*// в файл*

**public static void** writeRecords() **throws** Exception {

String res = **""**;

**for** (**int** i = 0; i < RECORD\_COUNT; i++) {

**if** (!names[i].equals(EMPTY\_VALUE)) {

res += String.format(**"%s%s%d%s"**, names[i], SEPARATOR, scores[i], SEPARATOR);

} **else** {

res += String.format(**"%s%s%s%s"**, EMPTY\_VALUE, SEPARATOR, EMPTY\_VALUE, SEPARATOR);

}

}

**if** (desCipher == **null**) {

init();

}

desCipher.init(Cipher.ENCRYPT\_MODE, desKey);

**byte**[] resBytes = res.getBytes();

*// Длина входной поледовательности должна быть кратна 64 бит*

**if** (resBytes.length % 8 != 0) {

**byte**[] newResBytes = **new byte**[resBytes.length + 8 - resBytes.length % 8];

System.arraycopy(resBytes, 0, newResBytes, 0, resBytes.length);

resBytes = newResBytes;

}

**byte**[] encryptedRes = desCipher.doFinal(resBytes);

FileOutputStream out = **new** FileOutputStream(RECORDS\_FILE\_NAME);

out.write(encryptedRes);

out.close();

}

*// Читает и дешифрует информацию из файла с рекорлами*

**public static void** readRecords() **throws** Exception {

names = **new** String[RECORD\_COUNT];

scores = **new int**[RECORD\_COUNT];

**try** {

File recordsFile = **new** File(RECORDS\_FILE\_NAME);

**byte**[] recordsFileBytes = **new byte**[(**int**)recordsFile.length()];

DataInputStream dis = **new** DataInputStream(**new** FileInputStream(recordsFile));

dis.readFully(recordsFileBytes);

dis.close();

**if** (desCipher == **null**) {

init();

}

desCipher.init(Cipher.DECRYPT\_MODE, desKey);

**byte**[] decryptedBytes = desCipher.doFinal(recordsFileBytes);

String[] recordsString = **new** String(decryptedBytes).split(SEPARATOR);

**for** (**int** i = 0; i < RECORD\_COUNT; i++) {

names[i] = recordsString[i\*2];

**if** (names[i].equals(EMPTY\_VALUE)) {

scores[i] = 0;

} **else** {

scores[i] = Integer.parseInt(recordsString[i\*2+1]);

}

}

} **catch** (FileNotFoundException e) {

**for** (**int** i = 0; i < RECORD\_COUNT; i++) {

names[i] = EMPTY\_VALUE;

scores[i] = 0;

}

writeRecords();

}

}

*// Создает объект для шифровки и дешифровки*

*// с использованием тройного DES-шифрования*

**private static void** init() **throws** Exception {

**byte**[] desKeyBytes = DatatypeConverter.parseHexBinary(KEY);

DESedeKeySpec desKeySpec = **new** DESedeKeySpec(desKeyBytes);

SecretKeyFactory keyFactory = SecretKeyFactory.getInstance(**"DESede"**);

desKey = keyFactory.generateSecret(desKeySpec);

desCipher = Cipher.getInstance(**"DESede/ECB/NoPadding"**);

}

**private static void** moveDown(**int** pos) {

**if** (pos >= RECORD\_COUNT - 2) {

**return**;

}

names[pos + 1] = names[pos];

scores[pos + 1] = scores[pos];

}

*// Добавляет рекорд в таблицу рекордов, помещая его на нужную позицию*

*// и вызывая перезапись файла*

**public static void** addRecord(String playerName) **throws** Exception {

**int** score = Field.getSnakeLength();

**int** pos = RECORD\_COUNT;

**for** (**int** i = RECORD\_COUNT - 1; i >= 0; i--) {

**if** (scores[i] <= score) {

moveDown(i);

pos = i;

} **else** {

**break**;

}

}

**if** (pos < RECORD\_COUNT) {

names[pos] = playerName;

scores[pos] = score;

writeRecords();

}

}

}

**В.2 Загрузка уровней**

*/\*\**

*\* Класс служит для загрузки информации об*

*\* уровнях и самих уровней*

*\*/*

**package** suffersnake.files;

**import** suffersnake.Main;

**import** suffersnake.game.Field;

**import** java.io.File;

**import** java.util.Scanner;

**import** java.util.Vector;

**public class** LevelLoader {

**private final static** String DIR\_NAME = **"levels"**; *// имя папки с уровнями*

**public static** String[] levelNames;

**private static** File[] levelFiles;

**private static boolean** initialized = **false**;

*// Пытается прочитать файл как уровень, и добавляет его*

*// в список файлов с уровнями, если не возникло исключений*

**private static void** checkFile(File file, Vector<String> levelNames, Vector<File> levelFiles) {

**boolean** goodFile = **true**;

String levelName = **""**;

**try** {

Scanner in = **new** Scanner(file);

levelName = in.nextLine();

**int** value;

**for** (**int** i = 0; i < Field.FIELD\_HEIGHT; i++) {

**for** (**int** j = 0; j < Field.FIELD\_WIDTH; j++) {

value = in.nextInt();

goodFile &= (value >= Field.BLOCK\_TYPE\_EMPTY && value <= Field.BLOCK\_TYPE\_EASIER\_WIN);

}

}

in.close();

} **catch** (Exception e) {

goodFile = **false**;

}

**if** (goodFile) {

levelNames.add(levelName);

levelFiles.add(file);

}

}

*// Проверка всех файлов в выбранной папке и формировнаие*

*// списка уровней*

**public static void** init() **throws** Exception {

File dir = **new** File(DIR\_NAME);

File[] dirFiles = dir.listFiles();

**if** (dirFiles.length == 0) {

**throw new** Exception();

}

Vector<File> levelFilesVector = **new** Vector<>();

Vector<String> levelNamesVector = **new** Vector<>();

**for** (File f: dirFiles) {

checkFile(f, levelNamesVector, levelFilesVector);

}

levelNames = levelNamesVector.toArray(**new** String[0]);

levelFiles = levelFilesVector.toArray(**new** File[0]);

initialized = **true**;

}

*// Загружает выбранный уровень в Field.environment*

**public static void** loadLevel(**int** index) **throws** Exception {

**if** (!initialized) {

init();

}

Scanner in = **new** Scanner(levelFiles[index]);

in.nextLine();

**for** (**int** i = 0; i < Field.FIELD\_HEIGHT; i++) {

**for** (**int** j = 0; j < Field.FIELD\_WIDTH; j++) {

Field.environment[i][j] = in.nextInt();

}

}

**if** (Main.isClassicMode()) {

**for** (**int** i = 0; i < Field.FIELD\_HEIGHT; i++) {

**for** (**int** j = 0; j < Field.FIELD\_WIDTH; j++) {

**switch** (Field.environment[i][j]) {

**case** Field.BLOCK\_TYPE\_EASIER\_WIN:

Field.environment[i][j] = Field.BLOCK\_TYPE\_IMMUNITY;

**break**;

**case** Field.BLOCK\_TYPE\_MAKE\_OTHER\_SHORTER:

Field.environment[i][j] = Field.BLOCK\_TYPE\_TRAP;

**break**;

**case** Field.BLOCK\_TYPE\_SLOW\_OTHER:

Field.environment[i][j] = Field.BLOCK\_TYPE\_SLOW\_URSELF;

**break**;

}

}

}

}

in.close();

}

}

Приложение Г

(обязательное)  
**Код модуля игровой логики**

**Г.1 Код класса, описывающего точку игрового поля**

*/\*\**

*\* Класс описывает объект, содержащий информацию об одной*

*\* точке поля*

*\*/*

**package** suffersnake.game;

**public class** Point {

**private int** x, y;

**public** Point(**int** x, **int** y) {

**this**.x = x;

**this**.y = y;

}

**public int** getX() {

**return** x;

}

**public int** getY() {

**return** y;

}

*// Возвращает соседа точки по указанному направлению, с учетом того,*

*// что противоположные границы связаны*

**public static** Point getNeighbour(Point point, **int** dim) {

**int** tempX = point.getX();

**int** tempY = point.getY();

**switch** (dim) {

**case** KeyHandler.Dimensions.DIMENSION\_UP:

tempY--;

**break**;

**case** KeyHandler.Dimensions.DIMENSION\_RIGHT:

tempX++;

**break**;

**case** KeyHandler.Dimensions.DIMENSION\_DOWN:

tempY++;

**break**;

**case** KeyHandler.Dimensions.DIMENSION\_LEFT:

tempX--;

**break**;

}

**if** (tempX < 0) tempX = Field.FIELD\_WIDTH - 1;

**if** (tempY < 0) tempY = Field.FIELD\_HEIGHT - 1;

**if** (tempX == Field.FIELD\_WIDTH) tempX = 0;

**if** (tempY == Field.FIELD\_HEIGHT) tempY = 0;

**return new** Point(tempX, tempY);

}

**public boolean** equals(Point point) {

**return** (x == point.getX() && y == point.getY());

}

*// Возвращает код направления, в котором нужно двигаться, чтобы*

*// попасть из точки p1 в точку p2*

**public static int** getDimension(Point p1, Point p2) {

**if** (p1.getX() == p2.getX()) {

**if** (p2.equals(getNeighbour(p1, KeyHandler.Dimensions.DIMENSION\_DOWN))) {

**return** KeyHandler.Dimensions.DIMENSION\_DOWN;

} **else** {

**return** KeyHandler.Dimensions.DIMENSION\_UP;

}

} **else** {

**if** (p2.equals(getNeighbour(p1, KeyHandler.Dimensions.DIMENSION\_RIGHT))) {

**return** KeyHandler.Dimensions.DIMENSION\_RIGHT;

} **else** {

**return** KeyHandler.Dimensions.DIMENSION\_LEFT;

}

}

}

}

**Г.2 Код класса, описывающего змейку и её поведение**

*/\*\**

*\* Класс описывает змейку и её поведение*

*\*/*

**package** suffersnake.game;

**import** suffersnake.Main;

**import** suffersnake.graphics.Painter;

**import** java.util.\*;

**public class** Snake {

**private int** number; *// Номер змейки в массиве Field.snakes*

**private int** colorIndex; *// Цвет змейки согласно Painter.SNAKE\_COLORS*

**private** ArrayList<Point> body; *// Тело змейки*

**private int** dimension, tempDimension; *// Текущее и предполагаемое следующее направления движения*

**private** Point head; *// Ссылается на голову*

**private int** lengthToWin; *// Длина, которая нужна для победы*

**private boolean** ai; *// Флаг показывает, управляет ли змейкой ИИ*

**private int** firingCounter = 0; *// Счеткчик блоков, которые нужно сжечь*

**private int** slowCounter = 0; *// Счетчик тактов, которые длися замедление*

**private boolean** needToMakeStep = **true**; *// Делает ли змейка ход на этом такте*

**private boolean** needToBurnTail = **true**; *// Нужно ли на след. ходу сжечь клетку*

*// Учитывается только при ненулевом firingCounter!*

**private int** trapPosition = -1; *// Позиция ловушки*

**private boolean** hasSlowedAnEnemy = **false**; *// Замедлила ли змейка врага*

**private boolean** hasShortedAnEnemy = **false**; *// Укоротила ли змейка врага*

**private int** immunityCounter = 0; *// Счетчик тактов, которые еще будет длится иммунитет*

**public** Snake(Point[] startPosition, **boolean** ai, **int** lengthToWin, **int** colorIndex, **int** number) {

**this**.number = number;

**this**.ai = ai;

**this**.lengthToWin = lengthToWin;

**this**.colorIndex = colorIndex;

body = **new** ArrayList<>();

Collections.addAll(body, startPosition);

head = startPosition[0];

dimension = KeyHandler.Dimensions.DIMENSION\_UP;

}

*// Устанавливает предполагаемое слежующее направление*

*// согласно нажатым клавишам клавиатуры*

**public void** setDimension(**int** dimension) {

**if** (body.size() > 1 && (**this**.dimension == KeyHandler.Dimensions.DIMENSION\_UP && dimension == KeyHandler.Dimensions.DIMENSION\_DOWN ||

**this**.dimension == KeyHandler.Dimensions.DIMENSION\_DOWN && dimension == KeyHandler.Dimensions.DIMENSION\_UP ||

**this**.dimension == KeyHandler.Dimensions.DIMENSION\_LEFT && dimension == KeyHandler.Dimensions.DIMENSION\_RIGHT ||

**this**.dimension == KeyHandler.Dimensions.DIMENSION\_RIGHT && dimension == KeyHandler.Dimensions.DIMENSION\_LEFT)) {

**return**;

}

tempDimension = dimension;

}

*// Принятие решения ИИ (подробно описано на рисунке 4 в ПЗ)*

**public int** makeDecision() {

**final int** WALL = -1, NOT\_VISITED = -2;

**int**[][] grid = **new int**[Field.FIELD\_HEIGHT][Field.FIELD\_WIDTH];

*// Установка стен и клеток для посещения*

**for** (**int** i = 0; i < Field.FIELD\_HEIGHT; i++) {

**for** (**int** j = 0; j < Field.FIELD\_WIDTH; j++) {

**switch** (Field.environment[i][j]) {

**case** Field.BLOCK\_TYPE\_EMPTY:

**case** Field.BLOCK\_TYPE\_MAKE\_OTHER\_SHORTER:

**case** Field.BLOCK\_TYPE\_EASIER\_WIN:

**case** Field.BLOCK\_TYPE\_IMMUNITY:

**case** Field.BLOCK\_TYPE\_SLOW\_OTHER:

grid[i][j] = NOT\_VISITED;

**break**;

**default**:

grid[i][j] = WALL;

}

}

}

ArrayList<Point> enemyBody = Field.snakes[number ^ 1].getBody();

Point p;

**for** (**int** i = 0; i < 4; i++) {

p = Point.getNeighbour(enemyBody.get(0), i);

grid[p.getY()][p.getX()] = WALL;

}

**for** (Point p1: body) {

grid[p1.getY()][p1.getX()] = WALL;

}

Vector<Point> visitedPoints = **new** Vector<>();

visitedPoints.add(head);

grid[head.getY()][head.getX()] = 0;

*// Ход волны*

**boolean** stop;

**int** curDist = 0;

**do** {

stop = **true**;

**for** (**int** i = visitedPoints.size()-1; i >= 0; i--) {

p = visitedPoints.get(i);

**if** (grid[p.getY()][p.getX()] != curDist) {

**break**;

}

**for** (**int** j = 0; j < 4; j++) {

p = Point.getNeighbour(visitedPoints.get(i), j);

**if** (grid[p.getY()][p.getX()] == NOT\_VISITED) {

grid[p.getY()][p.getX()] = curDist + 1;

stop = **false**;

visitedPoints.add(p);

}

}

}

curDist++;

} **while** (!stop);

p = **null**;

**if** (enemyBody.size() >= Main.LENGTH\_TO\_WIN \* 2/3 && body.size() < Main.LENGTH\_TO\_WIN / 2) {

*// Если противник сильно близок к победе, то охота за ним*

p = enemyBody.get(4);

Point p1, min = **null**;

**for** (**int** i = 0; i < 4; i++) {

p1 = Point.getNeighbour(p, i);

**if** (grid[p1.getY()][p1.getX()] > 0) {

**if** (min == **null**) {

min = p1;

} **else if** (grid[min.getY()][min.getX()] > grid[p1.getY()][p1.getX()]) {

min = p1;

}

}

}

p = min;

} **else** {

*// Иначе цель - ближайшая из приоритетных клеток*

Vector<Integer> priorities = **new** Vector<>();

priorities.add(Field.BLOCK\_TYPE\_FOOD);

priorities.add(Field.BLOCK\_TYPE\_EASIER\_WIN);

**if** (enemyBody.size() - body.size() >= Main.TRAPPING\_BLOCKS) {

priorities.add(Field.BLOCK\_TYPE\_MAKE\_OTHER\_SHORTER);

}

**for** (**int** i = 0; i < Field.FIELD\_HEIGHT; i++) {

**for** (**int** j = 0; j < Field.FIELD\_WIDTH; j++) {

**if** (priorities.contains(Field.blocks[i][j]) || priorities.contains(Field.environment[i][j])) {

**if** ((p == **null** || grid[p.getY()][p.getX()] > grid[i][j]) && grid[i][j] > 0) {

p = **new** Point(j, i);

}

}

}

}

}

*// На случай безысходной ситуации*

**if** (p == **null**) {

**return** dimension;

}

*// Восстановление пути до цели для выбора направления*

Point next;

**boolean** nextFound;

**while** (grid[p.getY()][p.getX()] != 1) {

nextFound = **false**;

**for** (**int** i = 0; i < 4; i++) {

next = Point.getNeighbour(p, i);

**if** (grid[next.getY()][next.getX()] == grid[p.getY()][p.getX()] - 1) {

p = next;

nextFound = **true**;

**break**;

}

}

**if** (!nextFound) {

**for** (**int** i = 0; i < 4; i++) {

next = Point.getNeighbour(p, i);

**if** (grid[next.getY()][next.getX()] == grid[p.getY()][p.getX()]) {

p = next;

**break**;

}

}

}

}

**return** Point.getDimension(head, p);

}

*// Действия змейки на каждом такте*

**public void** makeStep() {

dimension = ai ? makeDecision() : tempDimension;

**for** (Point curPoint: body) {

Field.changed[curPoint.getY()][curPoint.getX()] = **true**;

}

**if** (needToMakeStep) {

head = Point.getNeighbour(body.get(0), dimension);

body.add(0, head);

**if** (!Main.isClassicMode() && head.equals(Field.snakes[number ^ 1].getHead())) {

Main.stopGame(**"Змейки столкнулись головами"**, **false**);

**return**;

}

**if** (Field.blocks[head.getY()][head.getX()] != Field.BLOCK\_TYPE\_FOOD) {

body.remove(body.size()-1);

}

*// Добавление бонусов или вредных эффектов*

**switch** (Field.environment[head.getY()][head.getX()]) {

**case** Field.BLOCK\_TYPE\_FIRE:

**if** (immunityCounter == 0) {

firingCounter += Main.FIRING\_BLOCKS;

} **else** {

immunityCounter = 0;

}

**break**;

**case** Field.BLOCK\_TYPE\_SLOW\_URSELF:

**if** (immunityCounter == 0) {

slowCounter += Main.SLOWING\_TIME;

} **else** {

immunityCounter = 0;

}

**break**;

**case** Field.BLOCK\_TYPE\_TRAP:

**if** (immunityCounter == 0) {

trapPosition = 0;

} **else** {

immunityCounter = 0;

}

**break**;

**case** Field.BLOCK\_TYPE\_SLOW\_OTHER:

hasSlowedAnEnemy = **true**;

**break**;

**case** Field.BLOCK\_TYPE\_MAKE\_OTHER\_SHORTER:

hasShortedAnEnemy = **true**;

**break**;

**case** Field.BLOCK\_TYPE\_IMMUNITY:

immunityCounter += Main.IMMUNITY\_TIME;

**break**;

**case** Field.BLOCK\_TYPE\_EASIER\_WIN:

lengthToWin -= Main.EASIER\_WIN\_DEC;

**break**;

}

Field.environment[body.get(0).getY()][body.get(0).getX()] = Field.BLOCK\_TYPE\_EMPTY;

}

*// Применение присутсвтующих на данный момент бонусов или*

*// вредных эффектов*

**if** (immunityCounter != 0) {

immunityCounter--;

}

**if** (firingCounter > 0) {

needToBurnTail ^= **true**;

**if** (needToBurnTail) {

**if** (body.size() > 1) {

body.remove(body.size()-1);

} **else if** (Main.isClassicMode()) {

body.clear();

Main.stopGame(**"Змейка полностью сгорела."**, **false**);

**return**;

}

firingCounter--;

}

}

**if** (slowCounter > 0) {

needToMakeStep ^= **true**;

slowCounter--;

} **else** {

needToMakeStep = **true**;

}

**if** (trapPosition != -1) {

trapPosition++;

**if** (trapPosition == body.size()) {

makeShorter();

trapPosition = -1;

}

}

processSelfIntersection();

}

*// Проверка на самопересечение*

**private void** processSelfIntersection() {

**for** (**int** i = 1; i < body.size(); i++) {

**if** (body.get(i).equals(head)) {

**if** (Main.isClassicMode()) {

Main.stopGame(**"Змейка пересекла себя. Игра окончена."**, **true**);

**return**;

}

body = **new** ArrayList<>(body.subList(0, i));

Painter.addBlood(head.getX(), head.getY());

}

}

}

*// Добавление змейки в массивы класса Field для отрисовки*

**public void** dropToField() {

**for** (**int** i = 0; i < body.size(); i++) {

Field.blocks[body.get(i).getY()][body.get(i).getX()] = Field.BLOCK\_TYPE\_SNAKE;

Field.changed[body.get(i).getY()][body.get(i).getX()] = **true**;

}

}

**public boolean** contains(Point point) {

**for** (Point curPoint: body) {

**if** (curPoint.getX() == point.getX() && curPoint.getY() == point.getY()) {

**return true**;

}

}

**return false**;

}

**public boolean** isHead(Point point) {

**return** (head.getX() == point.getX() && head.getY() == point.getY());

}

**public** Point getHead() {

**return** head;

}

*// Проверка на пересечение с другой змейкой*

**public boolean** slice(Point enemyHead) {

**boolean** res = **false**;

**for** (**int** i = 1; i < body.size(); i++) {

**if** (body.get(i).equals(enemyHead)) {

body = **new** ArrayList<>(body.subList(0, i));

res = **true**;

**break**;

}

}

**return** res;

}

**public boolean** contains(**int** x, **int** y) {

**return** body.contains(**new** Point(x,y));

}

**public boolean** isHasSlowedAnEnemy() {

**boolean** res = hasSlowedAnEnemy;

**if** (res) {

hasSlowedAnEnemy = **false**;

}

**return** res;

}

**public void** makeSlowed() {

slowCounter += Main.SLOWING\_TIME;

}

**public boolean** tailIsFiring() {

**return** (firingCounter > 0 && !needToBurnTail);

}

**public** ArrayList<Point> getBody() {

**return** body;

}

*// Показывает, находится ли в указанной точке*

*// клетка змейки в ловушке*

**public boolean** isTrappedAt(Point p) {

**if** (trapPosition == -1) {

**return false**;

}

Point trappedBlock = body.get(trapPosition);

**return** (trappedBlock.equals(p));

}

**public boolean** isHasShortedAnEnemy() {

**boolean** res = hasShortedAnEnemy;

**if** (res) {

hasShortedAnEnemy = **false**;

}

**return** res;

}

*// Обрубает часть змейки (из-за ловушки или собранного врагом меча)*

**public void** makeShorter() {

**if** (Main.isClassicMode()) {

**if** (body.size() <= Main.TRAPPING\_BLOCKS) {

body.clear();

Main.stopGame(**"Змейка погибла в ловушке."**, **false**);

**return**;

}

}

**int** blocksToDestroyCount = Math.min(body.size() - 1, Main.TRAPPING\_BLOCKS);

Painter.addBlood(

body.get(body.size() - blocksToDestroyCount - 1).getX(),

body.get(body.size() - blocksToDestroyCount - 1).getY()

);

body = **new** ArrayList<>(body.subList(0, body.size() - blocksToDestroyCount));

}

**public boolean** hasImmunity() {

**return** (immunityCounter != 0);

}

**public int** getLengthToWin() {

**return** lengthToWin;

}

**public int** getColorIndex() {

**return** colorIndex;

}

**public boolean** hasWon() {

**return** (body.size() >= lengthToWin);

}

}

**Г.3 Код класса, описывающего действия с игровым полем**

**package** suffersnake.game;

**import** suffersnake.Main;

**import** java.util.Random;

**public class** Field {

*// Константы для блоков в соответствии с правилами игры*

**public static final int** BLOCK\_TYPE\_EMPTY = 0;

**public static final int** BLOCK\_TYPE\_FIRE = 1;

**public static final int** BLOCK\_TYPE\_SLOW\_URSELF = 2;

**public static final int** BLOCK\_TYPE\_TRAP = 3;

**public static final int** BLOCK\_TYPE\_SLOW\_OTHER = 4;

**public static final int** BLOCK\_TYPE\_MAKE\_OTHER\_SHORTER = 5;

**public static final int** BLOCK\_TYPE\_IMMUNITY = 6;

**public static final int** BLOCK\_TYPE\_EASIER\_WIN = 7;

**public static final int** BLOCK\_TYPE\_SNAKE = 8;

**public static final int** BLOCK\_TYPE\_FOOD = 9;

*// Высота и ширина поля в клетках*

**public static final int** FIELD\_HEIGHT = 25;

**public static final int** FIELD\_WIDTH = 38;

*// Блоки уровня*

**public static int**[][] environment = **new int**[FIELD\_HEIGHT][FIELD\_WIDTH];

*// Блоки змеек и еды*

**public static int**[][] blocks = **new int**[FIELD\_HEIGHT][FIELD\_WIDTH];

*// Необходимость перерисовывать блоки*

**public static boolean**[][] changed = **new boolean**[FIELD\_HEIGHT][FIELD\_WIDTH];

**public static** Snake[] snakes;

**private static** Random random = **new** Random();

*// Создает змейки с нужными параметрами в начальных позициях*

**public static void** createSnakes(**boolean** isAi1, **boolean** isAi2, **int** colorIndex1, **int** colorIndex2) {

snakes = **new** Snake[]{

**new** Snake(**new** Point[]{

**new** Point((FIELD\_WIDTH-2) / 4, (FIELD\_HEIGHT-3) / 2),

**new** Point((FIELD\_WIDTH-2) / 4, (FIELD\_HEIGHT-3) / 2 + 1),

**new** Point((FIELD\_WIDTH-2) / 4, (FIELD\_HEIGHT-3) / 2 + 2)

}, isAi1, Main.LENGTH\_TO\_WIN, colorIndex1, 0),

Main.isClassicMode() ? **null** : **new** Snake(**new** Point[]{

**new** Point(FIELD\_WIDTH - (FIELD\_WIDTH-2) / 4, (FIELD\_HEIGHT-3) / 2),

**new** Point(FIELD\_WIDTH - (FIELD\_WIDTH-2) / 4, (FIELD\_HEIGHT-3) / 2 + 1),

**new** Point(FIELD\_WIDTH - (FIELD\_WIDTH-2) / 4, (FIELD\_HEIGHT-3) / 2 + 2)

}, isAi2, Main.LENGTH\_TO\_WIN, colorIndex2, 1)

};

}

*// Перемещает блоки из environment в blocks для отрисовки*

**public static void** dropEnvironmentToBlocks() {

**for** (**int** i = 0; i < FIELD\_HEIGHT; i++) {

**for** (**int** j = 0; j < FIELD\_WIDTH; j++) {

**if** (blocks[i][j] != BLOCK\_TYPE\_SNAKE && blocks[i][j] != BLOCK\_TYPE\_FOOD) {

blocks[i][j] = environment[i][j];

}

changed[i][j] |= blocks[i][j] == BLOCK\_TYPE\_FIRE;

}

}

}

*// Помещает змеек в blocks для отрисовки*

**public static void** dropSnakesToBlocks() {

snakes[0].dropToField();

**if** (!Main.isClassicMode()) {

snakes[1].dropToField();

}

}

*// Убирает из массива blocks все, кроме еды*

**public static void** clearAllBlocksWoutFood() {

**for** (**int** i = 0; i < FIELD\_HEIGHT; i++) {

**for** (**int** j = 0; j < FIELD\_WIDTH; j++) {

**if** (blocks[i][j] != BLOCK\_TYPE\_FOOD) {

blocks[i][j] = BLOCK\_TYPE\_EMPTY;

}

changed[i][j] = **false**;

}

}

}

*// Случайным образом создает новую еду на поле*

**public static void** generateNewFood() {

**int** emptyCellsCount = 0;

**for** (**int** i = 0; i < FIELD\_HEIGHT; i++) {

**for** (**int** j = 0; j < FIELD\_WIDTH; j++) {

**if** (blocks[i][j] == BLOCK\_TYPE\_EMPTY && environment[i][j] == BLOCK\_TYPE\_EMPTY &&

!snakes[0].contains(j,i) && (Main.isClassicMode() || !snakes[1].contains(j,i))) {

emptyCellsCount++;

}

}

}

**if** (emptyCellsCount == 0) {

Main.stopGame(**"Змейка заняла все поле. Вы божественны!"**, **true**);

**return**;

}

**int** nextFoodCell = random.nextInt(emptyCellsCount) + 1;

**int** k = 0;

out: **for** (**int** i = 0; i < FIELD\_HEIGHT; i++) {

**for** (**int** j = 0; j < FIELD\_WIDTH; j++) {

**if** (blocks[i][j] == BLOCK\_TYPE\_EMPTY && environment[i][j] == BLOCK\_TYPE\_EMPTY &&

!snakes[0].contains(j,i) && (Main.isClassicMode() || !snakes[1].contains(j,i))) {

k++;

}

**if** (k == nextFoodCell) {

blocks[i][j] = BLOCK\_TYPE\_FOOD;

changed[i][j] = **true**;

**break** out;

}

}

}

}

*// Вызывает создание еды нужное количество раз*

**public static void** generateInitialFood(**int** count) {

**for** (**int** i = 0; i < count; i++) {

generateNewFood();

}

}

*// Если на поле мало еды, дополняет её до нужного количества*

**public static void** normalizeFoodCount() {

**int** requiredCount = Main.isClassicMode() ? 1 : Main.INITIAL\_FOOD\_COUNT;

**int** foodCount = 0;

**for** (**int** i = 0; i < FIELD\_HEIGHT; i++) {

**for** (**int** j = 0; j < FIELD\_WIDTH; j++) {

**if** (blocks[i][j] == Field.BLOCK\_TYPE\_FOOD) {

foodCount++;

}

}

}

**for** (**int** i = 0; i < requiredCount - foodCount; i++) {

generateNewFood();

}

}

*// Возвращает номер змейки, которая перексекла врага, либо -1,*

*// если пересечений не было*

**public static int** processIntersection() {

**if** (snakes[0].slice(snakes[1].getHead())) {

**return** 1;

} **else if** (snakes[1].slice(snakes[0].getHead())) {

**return** 0;

} **else** {

**return** -1;

}

}

*// Очищает весь массив blocks (необходимо при начале новой игры)*

**public static void** clearBlocks() {

**for** (**int** i = 0; i < FIELD\_HEIGHT; i++) {

**for** (**int** j = 0; j < FIELD\_WIDTH; j++) {

blocks[i][j] = BLOCK\_TYPE\_EMPTY;

}

}

}

*// Возвращает длину змейки в одиночном режиме*

**public static int** getSnakeLength() {

**return** snakes[0].getBody().size();

}

}

**Г.4 Код класса, выполняющего обработку нажатий клавиш**

*/\*\**

*\* Класс осуществяет обработку нажатий клавиш во время игры*

*\*/*

**package** suffersnake.game;

**import** suffersnake.KeyMap;

**import** suffersnake.Main;

**import** suffersnake.MainFrame;

**import** java.awt.event.KeyAdapter;

**import** java.awt.event.KeyEvent;

**import** java.awt.event.KeyListener;

**public class** KeyHandler {

**private static void** setDimensions(KeyEvent e) {

**switch** (e.getKeyCode()) {

**case** KeyMap.PLAYER1\_UP:

Field.snakes[0].setDimension(Dimensions.DIMENSION\_UP);

**break**;

**case** KeyMap.PLAYER1\_RIGHT:

Field.snakes[0].setDimension(Dimensions.DIMENSION\_RIGHT);

**break**;

**case** KeyMap.PLAYER1\_DOWN:

Field.snakes[0].setDimension(Dimensions.DIMENSION\_DOWN);

**break**;

**case** KeyMap.PLAYER1\_LEFT:

Field.snakes[0].setDimension(Dimensions.DIMENSION\_LEFT);

**break**;

**case** KeyMap.PLAYER2\_UP:

Field.snakes[1].setDimension(Dimensions.DIMENSION\_UP);

**break**;

**case** KeyMap.PLAYER2\_RIGHT:

Field.snakes[1].setDimension(Dimensions.DIMENSION\_RIGHT);

**break**;

**case** KeyMap.PLAYER2\_DOWN:

Field.snakes[1].setDimension(Dimensions.DIMENSION\_DOWN);

**break**;

**case** KeyMap.PLAYER2\_LEFT:

Field.snakes[1].setDimension(Dimensions.DIMENSION\_LEFT);

**break**;

**case** KeyMap.PAUSE:

Main.pause();

**break**;

}

}

**public static void** handleKeyEvent(KeyEvent e) {

setDimensions(e);

}

**public static void** bindKeyListener(MainFrame mainFrame) {

**for** (KeyListener curKeyListener: mainFrame.getKeyListeners()) {

mainFrame.removeKeyListener(curKeyListener);

}

mainFrame.addKeyListener(**new** KeyAdapter() {

@Override

**public void** keyPressed(KeyEvent e) {

handleKeyEvent(e);

}

});

}

*// Коды направлений*

**public static class** Dimensions {

**public static final int** DIMENSION\_UP = 0;

**public static final int** DIMENSION\_RIGHT = 1;

**public static final int** DIMENSION\_DOWN = 2;

**public static final int** DIMENSION\_LEFT = 3;

}

}

Приложение Д

(обязательное)  
**Код модуля, выполняющего отображение**

**игрового процесса**

**Д.1 Код класса, описывающего абстрактную клетку поля**

*/\*\**

*\* Класс описывает абстрактную клетку игрового поля*

*\* (в плане отрисовки), которая находится в определенном месте*

*\* и может себя нарисовать на экране*

*\*/*

**package** suffersnake.graphics;

**import** javax.imageio.ImageIO;

**import** java.awt.\*;

**import** java.awt.image.BufferedImage;

**import** java.io.File;

**import** java.io.IOException;

**public abstract class** Block {

**public final static int** BLOCK\_WIDTH = 20;

**public final static int** BLOCK\_HEIGHT = 20;

**protected static** BufferedImage environmentTextures;

**protected static** BufferedImage snakeTextures;

**private int** blockType;

**public static void** loadImages() **throws** IOException {

environmentTextures = ImageIO.read(**new** File(**"resource/img/environmentTextures.png"**));

snakeTextures = ImageIO.read(**new** File(**"resource/img/snakeTextures.png"**));

}

**public** Block(**int** blockType) {

**this**.blockType = blockType;

}

**public abstract void** draw(Graphics2D g2d, **int** blockX, **int** blockY);

**public int** getBlockType() {

**return** blockType;

}

}

**Д.2 Код класса, описывающего абстрактную систему частиц**

**package** suffersnake.graphics;

**import** java.awt.\*;

**import** java.util.Random;

**public class** ParticleSystem {

*/\*\**

*\* Интерфейс, необходимый для передачи в конструктор системы частиц*

*\* уравнений, задающих движение частиц. Параметры:*

*\* startX, startY - начальное положение частицы*

*\* x, y - текущее положение частицы*

*\* random - нужен, чтобы каждый раз не создавать заного*

*\*/*

**public interface** MovementEquation {

*// Получение новой абсциссы*

**public float** getX(**float** startX, **float** startY, **float** x, **float** y, Random random);

*// Получение новой ординаты*

**public float** getY(**float** startX, **float** startY, **float** x, **float** y, Random random);

*// Условие, при котором частицу больше не нужно рисовать*

**public boolean** needToStop(**float** startX, **float** startY, **float** x, **float** y, Random random);

}

**private** Particle[] particles;

**private int** particleCount;

**private** MovementEquation movementEquation;

**private float** startX, startY;

**private** Random random;

**public final static int** COLORSET\_FIRE = 0;

**public final static int** COLORSET\_BLOOD = 1;

**public static final int**[][] COLOR\_SETS\_VALUES = **new int**[][] {

{0xff0000, 0xff0000, 0xff0000, 0xffff00, 0xff8800}, *// Огонь*

{0xcc0000} *// Кровь*

};

**public** ParticleSystem(**final int** colorSet, **final int** particleCount, **final** MovementEquation movementEquation,

**final float** startX, **final float** startY) {

**this**.movementEquation = movementEquation;

**this**.startX = startX;

**this**.startY = startY;

**this**.particleCount = particleCount;

particles = **new** Particle[particleCount];

**for** (**int** i = 0; i < particleCount; i++) {

particles[i] = **new** Particle(startX, startY, colorSet);

}

random = **new** Random();

}

**public void** tick() {

**for** (**int** i = 0; i < particleCount; i++) {

**if** (!particles[i].isAlive()) {

**continue**;

}

particles[i].x = movementEquation.getX(startX, startY, particles[i].x, particles[i].y, random);

particles[i].y = movementEquation.getY(startX, startY, particles[i].x, particles[i].y, random);

**if** (movementEquation.needToStop(startX, startY, particles[i].x, particles[i].y, random)) {

particles[i].kill();

}

}

}

**public void** draw(Graphics2D g2d) {

tick();

**for** (Particle p: particles) {

**if** (p.isAlive()) {

p.draw(g2d);

}

}

}

}

*/\*\**

*\* Класс Particle описывает оду частицу, меняющую цвет в заданных пределах*

*\* каждый такт отрисовки*

*\*/*

**class** Particle {

**public float** x, y;

**private boolean** alive;

**public static final int** SIZE = 2;

**private int** colorSet;

**private** Random random;

**public** Particle(**float** startX, **float** startY, **int** colorSet) {

x = startX;

y = startY;

alive = **true**;

**this**.colorSet = colorSet;

random = **new** Random();

}

**private int** getNewColor() {

**return** ParticleSystem.COLOR\_SETS\_VALUES[colorSet][

random.nextInt(ParticleSystem.COLOR\_SETS\_VALUES[colorSet].length)

];

}

**public void** draw(Graphics2D g2d) {

**if** (x < Painter.LEFT\_CORNER\_X || y < Painter.LEFT\_CORNER\_Y ||

x + SIZE > Painter.LEFT\_CORNER\_X + Painter.FIELD\_TO\_PAINT\_WIDTH ||

y + SIZE > Painter.LEFT\_CORNER\_Y + Painter.FIELD\_TO\_PAINT\_HEIGHT) {

kill();

**return**;

}

g2d.setColor(**new** Color(getNewColor()));

g2d.fillRect(Math.round(x), Math.round(y), SIZE, SIZE);

}

*// После вызова метода частица рисоваться не будет*

**public void** kill() {

alive = **false**;

}

**public boolean** isAlive() {

**return** alive;

}

}

**Д.3 Код класса, описывающего огонь**

*/\*\**

*\* Класс описывает анимированный блок, рисующий огонь, используя*

*\* объект типа ParticleSystem*

*\*/*

**package** suffersnake.graphics;

**import** suffersnake.game.Field;

**import** java.awt.\*;

**import** java.util.Random;

**public class** FireBlock **extends** Block {

*// Смещение по У каждый такт отрисовки*

**private final static int** DY = 1;

*// Количество линий, из которых будут запущены системы частиц*

**private final static int** PARTICLE\_LINES\_COUNT = BLOCK\_HEIGHT / 2;

*// Количество частиц в одной линии*

**private final static int** PARTICLES\_IN\_LINE = 2;

*// Огонь состоит из нескольких систем частиц*

**private** ParticleSystem[] fires = **new** ParticleSystem[PARTICLE\_LINES\_COUNT];

**public** FireBlock(**int** blockX, **final int** blockY) {

**super**(Field.BLOCK\_TYPE\_FIRE);

**for** (**int** i = 0; i < PARTICLE\_LINES\_COUNT; i++) {

fires[i] = **new** ParticleSystem(

ParticleSystem.COLORSET\_FIRE,

PARTICLES\_IN\_LINE,

**new** ParticleSystem.MovementEquation() {

@Override

**public float** getX(**float** startX, **float** startY, **float** x, **float** y, Random random) {

*// Абцисса меняется случайным образом внутри блока*

x += (random.nextInt(3)-1) \* random.nextInt(BLOCK\_WIDTH / 4);

**if** (Math.abs(startX - x) >= BLOCK\_WIDTH / 2 - Particle.SIZE) {

**return** startX;

} **else return** x;

}

@Override

**public float** getY(**float** startX, **float** startY, **float** x, **float** y, Random random) {

*// Ордината меняется каждый такт одинакого, причем выйти за*

*// пределы блока частица огня не может*

y -= DY;

**if** (y - blockY <= DY) {

**return** startY;

} **else return** y;

}

@Override

**public boolean** needToStop(**float** startX, **float** startY, **float** x, **float** y, Random random) {

**return false**;

}

}, blockX + Block.BLOCK\_WIDTH / 2, blockY + BLOCK\_HEIGHT - i\*2 - Particle.SIZE

);

}

}

@Override

**public void** draw(Graphics2D g2d, **int** blockX, **int** blockY) {

g2d.setColor(**new** Color(0x000000));

g2d.fillRect(blockX, blockY, BLOCK\_WIDTH, BLOCK\_HEIGHT);

**for** (**int** i = 0; i < PARTICLE\_LINES\_COUNT; i++) {

fires[i].draw(g2d);

}

}

}

**Д.4 Код класса, описывающего пятна крови**

*/\*\**

*\* Класс описывает объект, рисующий кровь на игровом поле, используя*

*\* ParticleSystem*

*\*/*

**package** suffersnake.graphics;

**import** java.awt.\*;

**import** java.util.Random;

**public class** Blood {

**private** ParticleSystem particleSystem;

*// Количество частиц крови*

**private static final int** PARTICLE\_COUNT = 200;

*// Радиус пятна*

**private static final int** BLOOD\_SPREAD\_RADIUS = 50;

**public** Blood(**float** startX, **float** startY) {

particleSystem = **new** ParticleSystem(

ParticleSystem.COLORSET\_BLOOD,

PARTICLE\_COUNT,

**new** ParticleSystem.MovementEquation() {

@Override

**public float** getX(**float** startX, **float** startY, **float** x, **float** y, Random random) {

*// Абсцисса выбирается случайно в пределаз радиуса при первой отрисовке*

**return** Float.compare(x, startX) == 0 ?

x + random.nextInt(BLOOD\_SPREAD\_RADIUS) \* (random.nextBoolean() ? 1 : -1) :

x;

}

@Override

**public float** getY(**float** startX, **float** startY, **float** x, **float** y, Random random) {

*// Ордината тоже выбирается случайно в пределаз радиуса при первой отрисовке,*

*// но расстояние от цетра до частицы не должно быть более, чем*

*// BLOOD\_SPREAD\_RADIUS*

**if** (Float.compare(y, startY) == 0) {

**float** newY = y + random.nextInt(BLOOD\_SPREAD\_RADIUS) \* (random.nextBoolean() ? 1 : -1);

**if** (Math.sqrt(Math.pow(newY - startY, 2) + Math.pow(x - startX, 2)) > BLOOD\_SPREAD\_RADIUS) {

newY = (startY +

(**float**)Math.sqrt(Math.pow(BLOOD\_SPREAD\_RADIUS, 2) - Math.pow(x - startX, 2))) \*

(random.nextBoolean() ? 1 : -1);

}

y = newY;

}

**return** y;

}

@Override

**public boolean** needToStop(**float** startX, **float** startY, **float** x, **float** y, Random random) {

**return false**;

}

},

startX,

startY

);

}

**public void** draw(Graphics2D g2d) {

particleSystem.draw(g2d);

}

}

**Д.5 Код класса, описывающего статичную клетку игрового поля**

*/\*\**

*\* Класс описывает клетку, у которой статичная текстура, не*

*\* меняющаяся на протяжении игры*

*\*/*

**package** suffersnake.graphics;

**import** suffersnake.game.Field;

**import** java.awt.\*;

**import** java.awt.image.BufferedImage;

**public class** StaticBlock **extends** Block {

**private** BufferedImage image;

**private** Color backgroundColor = **new** Color(0x000000);

**public** StaticBlock(**int** blockType) {

**super**(blockType);

**if** (blockType != Field.BLOCK\_TYPE\_EMPTY) {

image = environmentTextures.getSubimage(

0,

blockType \* BLOCK\_HEIGHT,

BLOCK\_WIDTH,

BLOCK\_HEIGHT

);

} **else** {

image = **new** BufferedImage(BLOCK\_WIDTH, BLOCK\_HEIGHT, BufferedImage.TYPE\_INT\_RGB);

Graphics2D g2d = image.createGraphics();

g2d.setColor(backgroundColor);

g2d.fillRect(0, 0, BLOCK\_WIDTH, BLOCK\_HEIGHT);

}

}

@Override

**public void** draw(Graphics2D g2d, **int** blockX, **int** blockY) {

g2d.drawImage(image, blockX, blockY, **null**);

}

}

**Д.6 Код класса, описывающего клетку змейки**

*/\*\**

*\* Класс описывает клетку змейки, внешний вид*

*\* которой может изменяться в зависимости от состояния змейки*

*\*/*

**package** suffersnake.graphics;

**import** suffersnake.game.Field;

**import** java.awt.\*;

**import** java.awt.image.BufferedImage;

**public class** SnakeBlock **extends** Block {

**private** BufferedImage image;

**public** SnakeBlock(**int** colorIndex, **boolean** isHead, **boolean** tailIsFiring, **boolean** isTrappedHere,

**boolean** hasImmunity) {

**super**(Field.BLOCK\_TYPE\_SNAKE);

*// Выбор нужного вида клетки в зависимости*

*// от состояния змейки*

**int** k = 1;

**if** (isHead) k = 0;

**if** (hasImmunity && isHead) k = 4;

**if** (hasImmunity && !isHead) k = 5;

**if** (tailIsFiring) k = 2;

**if** (isTrappedHere) k = 3;

image = snakeTextures.getSubimage(

k \* BLOCK\_WIDTH,

colorIndex \* BLOCK\_HEIGHT,

BLOCK\_WIDTH,

BLOCK\_HEIGHT

);

}

@Override

**public void** draw(Graphics2D g2d, **int** blockX, **int** blockY) {

g2d.drawImage(image, blockX, blockY, **null**);

}

}

**Д.7 Код класса, выполняющего рисование в окне игры**

*/\*\**

*\* Основной класс, ответственный за отобржаение*

*\* игрового процесса*

*\*/*

**package** suffersnake.graphics;

**import** suffersnake.Main;

**import** suffersnake.MainFrame;

**import** suffersnake.game.\*;

**import** suffersnake.game.Point;

**import** java.awt.\*;

**import** java.awt.color.ColorSpace;

**import** java.awt.image.BufferedImage;

**import** java.awt.image.ColorConvertOp;

**import** java.util.Vector;

**public class** Painter {

*// Заданные в этом массиве константы цветов испольуются*

*// только при рисовании информации об игровой сессии, для отображения*

*// змеек цвета заданы в файле с текстурами*

**public static final** Object[][] SNAKE\_COLORS = **new** Object[][] {

{**new** Color(0x6666ff), **"Синий"**},

{**new** Color(0x00cc00), **"Зеленый"**},

{**new** Color(0xcc00cc), **"Фиолетовый"**},

{**new** Color(0xff8800), **"Оранжевый"**},

{**new** Color(0x888888), **"Серый"**},

{**new** Color(0xffffff), **"Белый"**}

};

**private static** BufferedImage gameScreen; *// Изображение для отрисовки*

**private static** BufferedImage background; *// Фон окна*

**private static** MainFrame mainFrame; *// Родительское окно*

*// Первый раз нужно отрисовать все клетки*

**private static boolean** needToDrawAllBlocks = **true**;

*// Массив клеток для отрисовки*

**private static** Block[][] blocks =

**new** Block[Field.FIELD\_HEIGHT][Field.FIELD\_WIDTH];

*// Массив с системами частиц, соответсующих крови на поле*

**private static** Vector<Blood> blood = **new** Vector<>();

**private static final int** MAX\_BLOOD\_COUNT = 50;

*// Размеры поля для отрисовки (в пикселях)*

**public static final int** FIELD\_TO\_PAINT\_WIDTH = Block.BLOCK\_WIDTH \* Field.FIELD\_WIDTH;

**public static final int** FIELD\_TO\_PAINT\_HEIGHT = Block.BLOCK\_HEIGHT \* Field.FIELD\_HEIGHT;

*// Координаты поля в окне*

**public static final int** LEFT\_CORNER\_X = (MainFrame.SCREEN\_WIDTH - FIELD\_TO\_PAINT\_WIDTH) / 2 ;

**public static final int** LEFT\_CORNER\_Y = (MainFrame.SCREEN\_HEIGHT - 25 - FIELD\_TO\_PAINT\_HEIGHT) / 2;

*// Параметры поля с инфоррмацией о текущей игровой сессии*

**private static final float** STATUS\_BAR\_FONT\_SIZE = 22;

**private static final** Color STATUS\_BAR\_BG\_COLOR = Color.BLACK;

**private static final** Color STATUS\_BAR\_BORDER\_COLOR = Color.WHITE;

**public static void** initialize(BufferedImage \_gameScreen, BufferedImage \_background, MainFrame \_mainFrame) {

gameScreen = \_gameScreen;

mainFrame = \_mainFrame;

**if** (\_background != **null**) {

background = **new** ColorConvertOp(

ColorSpace.getInstance(ColorSpace.CS\_GRAY),

**null**

).filter(\_background, **null**);

}

}

*// Помещает в массив blocks объекты для отрисовки всех клеток,*

*// в зависимости от типа клетки*

**private static void** loadBlocks() {

**int** curSnake;

**for** (**int** i = 0; i < Field.FIELD\_HEIGHT; i++) {

**for** (**int** j = 0; j < Field.FIELD\_WIDTH; j++) {

**if** (!needToDrawAllBlocks && !Field.changed[i][j]) **continue**;

**if** (Field.blocks[i][j] == Field.BLOCK\_TYPE\_FIRE) {

**if** (blocks[i][j] == **null** || blocks[i][j].getBlockType() != Field.BLOCK\_TYPE\_FIRE) {

blocks[i][j] = **new** FireBlock(

LEFT\_CORNER\_X + j \* Block.BLOCK\_WIDTH,

LEFT\_CORNER\_Y + i \* Block.BLOCK\_HEIGHT

);

}

} **else if** (Field.blocks[i][j] == Field.BLOCK\_TYPE\_SNAKE) {

curSnake = Field.snakes[0].contains(**new** Point(j, i)) ? 0 : 1;

blocks[i][j] = **new** SnakeBlock(

Field.snakes[curSnake].getColorIndex(),

Field.snakes[curSnake].isHead(**new** Point(j, i)),

Field.snakes[curSnake].tailIsFiring(),

Field.snakes[curSnake].isTrappedAt(**new** Point(j, i)),

Field.snakes[curSnake].hasImmunity()

);

} **else** {

**if** (blocks[i][j] == **null** || blocks[i][j].getBlockType() != Field.blocks[i][j]) {

blocks[i][j] = **new** StaticBlock(Field.blocks[i][j]);

}

}

}

}

}

*// Рисует текущее состояние игры в игровом окне*

**public static void** paint() {

loadBlocks();

Graphics2D g2d = gameScreen.createGraphics();

*// Начальная зарисовка*

**if** (needToDrawAllBlocks) {

**if** (background != **null**) {

g2d.drawImage(background, 0, 0, **null**);

} **else** {

g2d.setColor(Color.WHITE);

g2d.fillRect(0, 0, MainFrame.SCREEN\_WIDTH, MainFrame.SCREEN\_HEIGHT);

}

g2d.setColor(Color.WHITE);

g2d.drawRect(LEFT\_CORNER\_X -1, LEFT\_CORNER\_Y -1, FIELD\_TO\_PAINT\_WIDTH +1, FIELD\_TO\_PAINT\_HEIGHT +1);

}

*// Отрисовка фона*

**for** (**int** i = 0; i < Field.FIELD\_HEIGHT; i++) {

**for** (**int** j = 0; j < Field.FIELD\_WIDTH; j++) {

**if** (blocks[i][j].getBlockType() != Field.BLOCK\_TYPE\_EMPTY ||

!Field.changed[i][j] && !needToDrawAllBlocks) {

**continue**;

}

blocks[i][j].draw(

g2d,

LEFT\_CORNER\_X + j \* Block.BLOCK\_WIDTH,

LEFT\_CORNER\_Y + i \* Block.BLOCK\_HEIGHT

);

}

}

*// Отрисвока крови*

**for** (Blood b: blood) {

b.draw(g2d);

}

*// Отрисовка меняющихся блоков*

**for** (**int** i = 0; i < Field.FIELD\_HEIGHT; i++) {

**for** (**int** j = 0; j < Field.FIELD\_WIDTH; j++) {

**if** (blocks[i][j].getBlockType() == Field.BLOCK\_TYPE\_EMPTY) {

**continue**;

}

blocks[i][j].draw(

g2d,

LEFT\_CORNER\_X + j \* Block.BLOCK\_WIDTH,

LEFT\_CORNER\_Y + i \* Block.BLOCK\_HEIGHT

);

}

}

drawScores(g2d);

mainFrame.repaint();

**if** (needToDrawAllBlocks) {

needToDrawAllBlocks = **false**;

}

}

*// Добавялет новое пятно крови на поле в указанную клетку*

**public static void** addBlood(**int** x, **int** y) {

**if** (!Main.isNeedToDrawBlood()) {

**return**;

}

blood.add(**new** Blood(

LEFT\_CORNER\_X + x \* Block.BLOCK\_WIDTH + Block.BLOCK\_WIDTH / 2,

LEFT\_CORNER\_Y + y \* Block.BLOCK\_HEIGHT + Block.BLOCK\_HEIGHT / 2

));

**if** (blood.size() > MAX\_BLOOD\_COUNT) {

blood = **new** Vector<>(blood.subList(1, blood.size()));

}

}

**public static void** setNeedToDrawAllBlocks(**boolean** value) {

needToDrawAllBlocks = value;

}

**public static void** clearBlood() {

blood.clear();

}

*// Отображает информацию о текущей игровой сессии*

**private static void** drawScores(Graphics2D g2d) {

g2d.setColor(STATUS\_BAR\_BG\_COLOR);

g2d.fillRect(100, 550, 600, 25);

g2d.setColor(STATUS\_BAR\_BORDER\_COLOR);

g2d.drawRect(99, 549, 601, 26);

g2d.setFont(MainFrame.MENU\_FONT.deriveFont(STATUS\_BAR\_FONT\_SIZE));

**if** (!Main.isClassicMode()) {

g2d.setColor((Color)SNAKE\_COLORS[Field.snakes[0].getColorIndex()][0]);

g2d.drawString(

String.format(**"Игрок 1: %d / %d"**, Field.snakes[0].getBody().size(),

Field.snakes[0].getLengthToWin()),

125, 570

);

g2d.setColor((Color)SNAKE\_COLORS[Field.snakes[1].getColorIndex()][0]);

g2d.drawString(

String.format(**"Игрок 2: %d / %d"**, Field.snakes[1].getBody().size(),

Field.snakes[1].getLengthToWin()),

450, 570

);

} **else** {

g2d.setColor((Color)SNAKE\_COLORS[Field.snakes[0].getColorIndex()][0]);

g2d.drawString(

String.format(**"Длина змейки: %d"**, Field.snakes[0].getBody().size()),

125, 570

);

}

}

}

**10 Список используемых источников**

1. Страница для получения игры Snake Classics [Электронный ресурс] // URL: http://toomkygames.com/ru/games/snake-classics/
2. Страница игры Python [Электронный ресурс] // URL: http://ru.y8.com/games/python
3. Страница игры Snake-game [Электронный ресурс] // URL: http://www.onemotion.com/flash/snake-game/
4. Страница игры Intelligent Multiplayer Snake [Электронный ресурс] // URL: http://isnake.sourceforge.net/
5. Официальная документация Java [Электронный ресурс] // URL: http://docs.oracle.com/javase/7/docs/api/