**ДЕПАРТАМЕНТ ОБРАЗОВАНИЯ ЯРОСЛАВСКОЙ ОБЛАСТИ**

государственное профессиональное образовательное учреждение   
Ярославской области

Рыбинский полиграфический колледж

**КУРСОВОЙ ПРОЕКТ**

|  |  |
| --- | --- |
| Разработка класса визуального компонента-игры «Змейка» | |
|  | |
| по дисциплине | Объектно-ориентированное программирование |
|  | |

Пояснительная записка

|  |
| --- |
|  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Студент группы | 3-ИС-2 |  |  |  | Кузнецов Н С |
|  | *(Код учебной группы)* |  | *(Подпись, дата)* |  | *(И.О.Фамилия)* |
| Руководитель | преподаватель |  |  |  | Смирнов В Б |
|  | *(Должность, звание)* |  | *(Подпись, дата)* |  | *( И.О.Фамилия)* |
| Нормоконтроль | преподаватель |  |  |  | Смирнов В Б |
|  | *(Должность, звание)* |  | *(Подпись, дата)* |  | *(И.О.Фамилия)* |

г. Рыбинск

2022

содержание

[Введение 3](#_Toc104039610)

[1 Исследовательский раздел 6](#_Toc104039611)

[2 Конструкторский раздел 9](#_Toc104039612)

[2.1 Математическая модель 9](#_Toc104039613)

[2.2 Разработка структуры компонента 11](#_Toc104039614)

[2.3 Разработка графического интерфейса компонента 15](#_Toc104039615)

[2.4 Описание входных и выходных данных компонента 18](#_Toc104039616)

[2.5 Разработка алгоритмов компонента 20](#_Toc104039617)

[3 Технологический раздел 37](#_Toc104039618)

[3.2 Инструкция по установке компонента 41](#_Toc104039619)

[3.3 Инструкция по эксплуатации компонента 47](#_Toc104039620)

[4 Раздел охраны труда 48](#_Toc104039621)

[Заключение 50](#_Toc104039622)

[Список источников 52](#_Toc104039623)

[Приложение А 53](#_Toc104039624)

Введение

В 21 веке вычислительная техника связана почти с каждым человеком на планете. С её помощью достигаются новые вершины в науке, модернизируя её саму. С ней возможно связаться с любым человеком из разных уголков земли, не прилагая особых усилий. Технологии подарили досуг, развлечения, работу, горы информации которая не соберется ни в одной библиотеке мира. Но всегда ли было так? С чего все началось?

Вычислительная техника работает благодаря логике, которая представлена в виде машинного кода, а языки программирования существуют для написания и упрощения программного кода, первым таким языком был ассемблер, созданный по принципу «Одна инструкция – одна строка». Ассемблер — язык программирования низкого уровня. Язык программирования низкого уровня — это язык программирования, который ориентирован на конкретный тип процессора и учитывает его особенности. Появление языка ассемблера значительно облегчило жизнь программистов, так как теперь вместо рябящих в глазах нулей и единиц, они могли писать программу командами, состоящими из символов, приближенных к обычному языку. Но сложность разработки в нём больших программных комплексов привела к появлению высокоуровневых языков программирования.

С развитием возможностей вычислительной техники объём функциональности и сроки реализации кода на ассемблере уже не устраивали. Затраты для написания, тестирования и сопровождения программ росли на порядок быстрее их возможностей. Необходимо было снизить требования от программиста в плане знаний функционирования аппаратуры, дать ему инструмент, позволяющий писать на языках, приближенных к человеческой логике. Перейти к новому уровню типов языков программирования. Предоставить возможность разбивать на разнообразные модули с дальнейшим последовательным вызовом, предоставить различные типы данных с возможностью их конструирования. Одним из первых языков, поддерживающий всё вышеперечисленное был Фортран. Возможность создавать в текстовом виде с описанием логики выполнения используя циклы, ветвления, подпрограммы и оперируя массивами и представляя данные в виде вещественных, целых и комплексных чисел приводила программистов в восторг. Компилятор получает каждую команду и преобразует ее в машинный код. Он позволяет использовать имена (переменные) для представления элементов данных так, что одна и та же программа может быть использована с любыми входными данными. Программы, написанные на языках высокого уровня, более компактны, легче для понимания, а вероятность появления в них ошибок меньше.

В связи с ростом сложности программ и дальнейшим развитием программных средств возникла необходимость увеличить производительность труда программистов, что привело к разработке структурного программирования. С развитием структурного программирования следующим достижением были [процедуры](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D0%B4%D1%83%D1%80%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) и [функции](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)). То есть, если есть задача, которая выполняется несколько раз, то её можно объявить, как функцию или как процедуру и в выполнении программы просто вызывать её. Общий код программы в данном случае становится меньше. Следующим достижением было объединение разнородных данных, которые используются в программе в связке, в структуры. [Структуры](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) — это составные типы данных, построенные с использованием других типов данных. Дальше началось зарождаться объектно-ориентированное программирование. При использовании структур данных в программе вырабатываются и соответствующие им функции для работы с ними. Это привело к мысли их объединить и использовать совместно, так появились классы. Теперь программирование можно было разбить на классы и тестировать не всю программу, состоящую из 10 000 строк кода, а разбить программу на 100 классов, и тестировать каждый класс. Объектно-ориентированное программирование сочетает лучшие принципы структурного программирования с новыми концепциями [инкапсуляции](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%BA%D0%B0%D0%BF%D1%81%D1%83%D0%BB%D1%8F%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), [полиморфизма подтипов](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D0%B8%D0%BC%D0%BE%D1%80%D1%84%D0%B8%D0%B7%D0%BC_%D0%BF%D0%BE%D0%B4%D1%82%D0%B8%D0%BF%D0%BE%D0%B2) и [наследования](https://ru.wikipedia.org/wiki/%D0%9D%D0%B0%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), информация из работы [1].

В рамках данного курсового проекта планируется разработка игры «Змейка». Змейку лучше всего реализовывать с помощью языков программирования с дальнейшей эксплуатацией при помощи вычислительной техники. Такой подход обеспечит наибольшую эффективность, поскольку он позволяет быстро спроектировать и реализовать проект. Распространение проекта будет более целесообразно в цифровом формате нежели в физическом представлении данной игры. «Змейка» является, возможно, одной из самых популярных игр, которые вложили свою долю в основу игрового сегмента. Эта легендарная игра на одном уровне с Марио и Тетрисом, которая безусловно успела подарить много различных эмоций людям, которым удалось познакомиться с ней. Даже в настоящее время к ней не потеряли интерес, и продолжают играть в неё для получения новых эмоций.

«Змейка» – это хороший вариант для обычного пользователя, чтобы сократить время в ожидании чего-либо или просто развлечь себя. Для прикладного программиста «Змейка» – это хороший опыт, который может помочь при написании новых программ и разработке игр.

1 Исследовательский раздел

При выборе сред разработки были рассмотрены Visual Studio и Project Rider. Visual Studio – это удобная интегрированная среда разработки (IDE) от Microsoft, позволяющая быстро и эффективно создавать, и разрабатывать проект, выбрав для этого все необходимое. Среда использует платформы разработки программного обеспечения Microsoft: Windows API, Windows Forms, Windows Presentation Foundation, Windows Store и Microsoft Silverlight. Так же она принимает плагины, которые расширяют функциональные возможности практически на каждом уровне, включая добавление поддержки систем управления исходным кодом (таких как Subversion) и обеспечивает стандартный для Windows вид окон приложения. Единственным минусом можно считать сложность освоения данной среды разработки из-за её большого количества различных функций, спрятанных в подразделах меню. Информация из работы [2].

Project Rider – это среда от JetBrains для работы с платформой .NET. Она обладает поддержкой полного цикла. Фирменная черта продуктов JetBrains, воплощенная и в Project Rider. С Project Rider появиться возможность организовать весь цикл создания программного обеспечения: от идеи до поддержки. Функциональность Project Rider позволяет подключить MSBuild и XBuild, работать с CLI-проектами и организовать отладку приложений .NET and Mono. Множество опций для быстрого создания кода улучшает производительность. Кроссплатформенность Project Rider работает с Windows, Linux и MacOS. Из минусов можно выделить её молодость. Часть функциональности еще в разработке, не все стартовые ошибки исправлены. Так же можно отметить её стоимость. Самая дешевая версия Project Rider обойдется в 139 долларов за первый год использования. Но есть триал-версия и специальные предложения для студентов и непрофильных организаций. Информация из работы [3].

Из этих двух сред разработки был выбран Visual Studio, так как он обладает всем необходимым функционалом для реализации проекта, а также она является бесплатной и дольше находится на рынке труда.

В проекте в качестве родительского класса был выбран класс Control, отвечающий за визуальное представление компонента в форме. Класс Control имеет метод Invalidate отвечающий за перерисовку компонента, а также метод OnPaint, он не реализует функциональность рисования, но вызывает делегаты событий, зарегистрированные в событии [Paint](https://docs.microsoft.com/ru-ru/dotnet/api/system.windows.forms.control.paint). Класс Control даёт весь необходимый функционал для реализации проекта. Так же проект должен соответствовать требованиям графического интерфейса операционной системы Microsoft Windows поскольку это существенно облегчат жизнь программисту, упрощая взаимодействие с программой.

В интернете существует большое изобилие клонов игры «Змейки». Одним из таких примеров можно ознакомиться по адресу [4]. Данная версия игры – это пример классической игры «Змейка». В этой версии змейка может выйти за границы поля и выходить из его противоположной стороны в отличии от проекта, который будет реализован. Так же здесь нет сложной анимации, только змейка, фрукты и счет. Присутствует и фиксация лучшего счёта игрока. Эта версия максимальна проста в своем исполнении и нельзя выделить каких-то выраженных плюсов. В отличии от проекта, эта версия зависима от интернета и не может использоваться в офлайн режиме, что может являться существенным минусом для пользователя. Стоит заметить отсутствие системы жизней, которые позволят продолжить игру в случаи ошибки пользователя. Поскольку игру сложно пройти до конца, пользователю не будет интересно постоянно начинать сначала после неудачи, интереснее было бы сохранить результат и дать пользователю возможность продолжить игру с сохранением имеющегося результата.

В проекте будет реализована вся базовая логика - это генерация игрового поля при помощи метода OnPaint, направление движения змейкой при помощи событий нажатия клавиш на клавиатуре. Будет реализован метод генерации фруктов, ведения счета и фиксация лучшего результата. Так же будут реализованы методы слежения выхода змейки за границы поля и столкновения её головы с хвостом. Будет добавлен метод генерирующий хвост змейки и реализация логики системы жизни у змейки, при потери которой, змейка будет переходить в состояние «Призрака» на 3 секунды и сможет передвигаться по своему хвосту. Анимация будет реализована при помощи таймера, при вызове которого будет начинаться игра, и осуществляться передвижение змейки. При старте и завершении игры будут выводиться соответствующие сообщения. Будет реализована логика адаптивности компонента при изменении размера компонента.

Основным процессом в проекте будет являться реализация анимации передвижения змейки по игровому полю, включая функции управления игроком данным процессом.

2 Конструкторский раздел

2.1 Математическая модель

Модель основного процесса движения змейки будет происходить благодаря таймеру, который будет вызывать методы проверок взаимодействия змейки с фруктом, полем и хвостом, а также будет содержать в себе метод, отвечающий за передвижение змейки.

Скорость змейки на поле будет зависеть от скорости обновления таймера. Чем быстрее обновляется таймер, тем быстрее передвигается змейка. Вектор её направления будет завесить от нажатой клавиши на клавиатуре пользователем. Метод, который будет вызывать таймер будет создавать новый поток, внутри которого будет метод invoke выполняющий делегат со следующими методами внутри: метод передвижения змейки, метод проверки выхода головы за пределы границ, метод проверки съедания фрукта головой змейки, метод проверки вхождения головы змейки в свой хвост, в конце метод перерисовки invalidate. Метод передвижения змейки будет вызывать в себе методы передвижения головы и хвоста змейки, а также метод инвертирования направления змейки при определённом условии.

Изначально змейка будет появляться в центре поля. Она будет двигаться без остановки в направлении, которое задается с помощью двух переменных, отвечающих за оси x и y. Значения этих переменных, будут задаваться в обработчике событий, отвечающих за нажатую клавишу на клавиатуре пользователем. Каждая клавиша будет отвечать за свой вектор направления, всего их будет четыре. Внутри метода передвижения змейки будет находится условие, которое будет контролировать перемещение змейки. В первой ветке будет проверка счета на ноль, в этом случае будет вызываться метод перемещения головы змейки внутри которого будет переопределение позиции головы в соответствие с вектором направления. Во второй ветке будет проверка счета на то чтобы он был больше ноля, а также проверка перемещения головы змейки. Если голова не сдвинется в обратном направлении по отношению к своему хвосту, то будет вызван метод передвижения хвоста змейки, а также метод передвижения её головы. Метод передвижения хвоста будет реализован при помощи цикла, в котором перебираются все элементы хвоста змейки и принимают позицию своего соседа по направлению к голове. Иначе если оба условия не будут выполнены, вызовется метод инвертирования направления, а после него метод передвижения хвоста и головы змейки. Метод инвертирования направления будет содержать в себе два условия для каждого вектора, внутри которых будет меняться их значение на противоположное, при условии, что их значение не было равно 0. Это сделано для того, чтобы не допустить возможности передвижения головы змейки в обратном направлении по отношению к своему хвосту.

2.2 Разработка структуры компонента

Для описания метода проектирования компонента будет использоваться путь проведения серии декомпозиций, начиная с «черной сферы». Черная сфера представляет собой систему, внутреннее устройство которой не важно. В эту систему подаются входные данные, а на выходе из системы поступают выходные данные. Черная сфера представлена на рисунке 2.1.

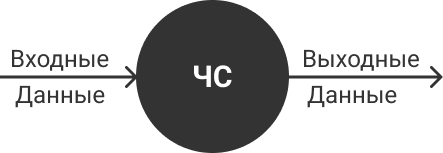


Рисунок 2.1 – Модель «Черная сфера»

Представим наш класс в виде черной сферы. В компоненте будут присутствовать такие выходные данные, как цвета для объектов компонента, его размеры и управляющее воздействие пользователя. На выходе из компонента будет поступать счёт, лучший счёт на момент игры и количество жизней, оставшихся у пользователя, а также изображение на экране. Цвета будут использоваться для таких объектов как игровое поле, голова змейки, хвост змейки и фрукты. Размеры будут поступать после изменения размеров компонента с помощью манипулятора типа «мышь». Управляющее воздействие будет происходить с помощью клавиатуры. Счёт, лучший счёт и количество жизней будут доступны через свойства, а изображение компонента будет отображаться на экране. Черная сфера с перечисленными входными и выходными параметрами представлена на рисунке 2.2.

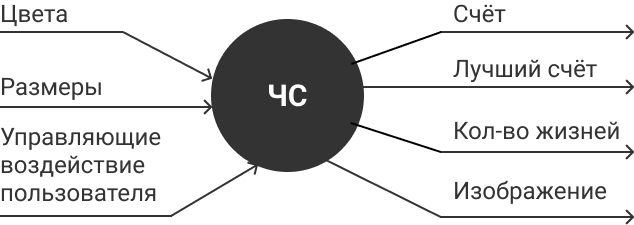


Рисунок 2.2 – Черная сфера с параметрами

Для выявления основных параметров и процессов компонента проведем несколько этапов декомпозиции. Декомпозиция – операция [мышления](https://ru.wikipedia.org/wiki/%D0%9C%D1%8B%D1%88%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5" \o "Мышление), состоящая в разделении целого на части. Также декомпозицией называется общий приём, применяемый при решении [проблем](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B1%D0%BB%D0%B5%D0%BC%D0%B0" \o "Проблема), состоящий в разделении проблемы на множество частных проблем, а также [задач](https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%B4%D0%B0%D1%87%D0%B0" \o "Задача), не превосходящих суммарно по [сложности](https://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D0%BE%D0%B6%D0%BD%D0%BE%D1%81%D1%82%D1%8C" \o "Сложность) исходную проблему, с помощью объединения решений которых, можно сформировать решение исходной проблемы в целом.

Проведем первый этап декомпозиции с выявлением основных внутренних структур данных, необходимых для хранения информации, и процессов компонента. Размеры всех составных частей компонента (карта, фрукты, змейка т.д.) будут напрямую зависеть от его размеров. Обрабатывать изменения будет метод, фиксирующий изменения размеров компонента и корректирующий размеры его составных частей таким образом, чтобы компонент стал полностью адаптивным. При изменении цвета элементов, компонент будет перерисовываться в соответствии с заданными цветами. Обработка нажатия клавиш пользователем будет осуществляться событием, реагирующим на нажатия и изменяя в соответствии с клавишей вектор направления змейки. События, которые будут реагировать на изменения счета и количества жизней будут выводить соответствующие данные на экран. Результат первого этапа декомпозиции представлен на рисунке 2.3.



Рисунок 2.3 – Результат первого этапа декомпозиции

На первом этапе декомпозиции были получены основные структуры данных, необходимых для хранения информации, и процессов компонента, которые позволят построить базовую логику. Для получения более точных структур данных и процессов компонента проведем второй этап декомпозиции.

Управляющее воздействие пользователя будет влиять на следующие методы: запуск таймера, остановка таймера, изменение направление по нажатию клавиш. Метод запуска таймера изменит состояние игры и запустит таймер, при этом вызовется метод установки начальных позиций и размеров для змейки и фрукта, а также произойдет обновление экрана. Метод остановки таймера также изменит состояние игры и остановит таймер, удалит змейку. При остановке таймер произойдет событие записи лучшего счёта и обновление экрана. При обновлении таймера происходит перемещение змейки, проверка на её столкновение с хвостом и выход за границы. Так же проверка на столкновение с фруктом после которого произойдет обновление счёта и генерация нового фрукта. При выходе за границы вызовется метод остановки таймера, а при столкновении с хвостом в зависимости от количества жизней, либо также вызовется метод остановки таймера, либо переход змейки в режим призрака. В состоянии призрака змейка станет синего цвета и получит возможность беспрепятственно проходить по своему хвосту на 3 секунды. По истечению времени змейка вернется в прежней режим и количество жизней уменьшится. При изменении цвета элементов, компонент будет перерисовываться в соответствии с заданными цветами. Размеры компонента и размеры его элементов (карта, фрукты, змейка т.д.) будут зависеть от изменения размеров окна формы пользователем. Обрабатывать изменения будет метод, фиксирующий изменения размеров формы и вызывающий метод, корректирующий размеры элементов компонента таким образом, чтобы компонент соответствовал размерам формы. Результат второго этапа декомпозиции представлен на рисунке 2.4. и на плакате КП.0902,09.000000.01 ПЛ.

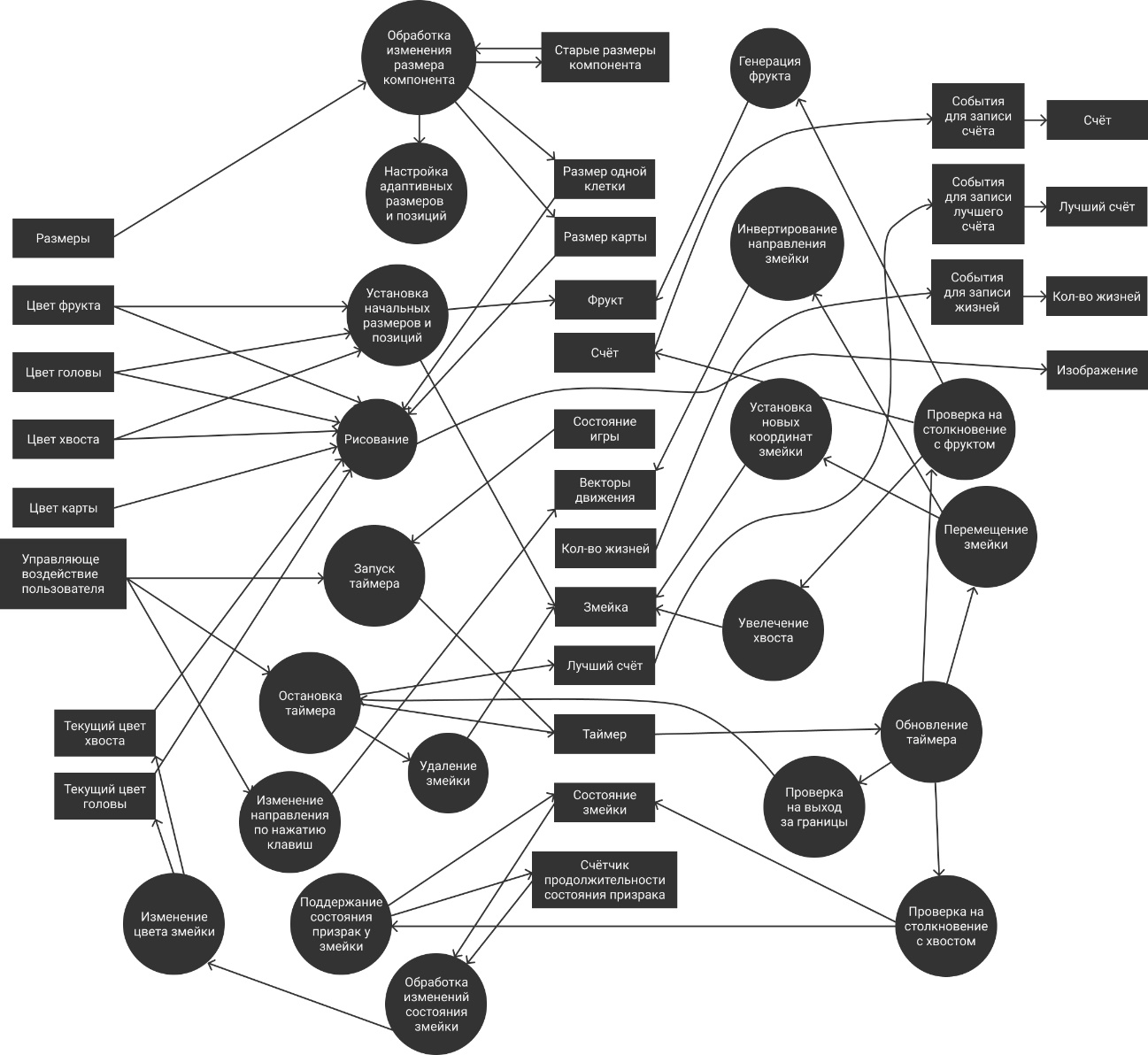


Рисунок 2.4 – Результат второго этапа декомпозиции

2.3 Разработка графического интерфейса компонента

Пользовательские интерфейсы современных программ строятся по технологии WIMP: Windows (окна), Icons (пиктограммы), Mouse (мышь), Pop-up (всплывающие или выпадающие меню). Таким образом, основными элементами графических интерфейсов являются окна, пиктограммы, компоненты ввода-вывода, мышь, которую используют в качестве указывающего устройства и устройства прямого манипулирования объектами на экране. Окно – это ограниченная рамкой область экрана, которая может менять размеры и местоположение в пределах экрана. Все окна можно разделить на пять категорий: основные окна, дочерние окна, окна диалога, информационные окна, окна меню. Пиктограмма – это небольшое окно с графическим изображением.

Различают следующие пиктограммы: программные пиктограммы (свернутое в пиктограмму окно приложения), пиктограммы дочерних окон, пиктограммы панели инструментов (дублируют команды меню для быстрого вызова), пиктограммы объектов (используются для прямого манипулирования объектами).

Прямое манипулирование объектами - это возможность замены команды воздействия на некоторый объект физическим действием в интерфейсе, осуществляемым с помощью мыши. При этом любая область экрана рассматривается как адресат, который может быть активизирован при подведении курсора и нажатии клавиши мыши. По реакции на воздействие различают следующие типы адресатов: указание и выбор (развертывание пиктограммы, определение активного окна), буксировка и «резиновая нить» (перенос объекта и его границ), экранные кнопки. Для реализации диалогов, управляемых пользователем, применяют меню различных видов. Для реализации диалогов, управляемых системой, обычно используют диалоговые окна.

Можно проследить за историей развития графических интерфейсов для Windows начиная с Windows 1.0. Особенностью этой операционной системы было наличие окон, которые можно было расположить друг около друга и возможность управления мышкой. С выходом Windows 2.0 окна уже могут перекрывать друг друга, в отличии от версии 1.0. Для отображения интерфейса используется 16 цветов, и к окнам впервые применены термины «свернуть» и «развернуть». В Windows 3.0 пользователь мог любоваться 256-ю цветами, что однозначно делало работу более приятной, хотя основные элементы интерфейса остались без существенных изменений, информация из работы [5]. Windows 95-98-98SE-2000-ME-XP-Vista-7 закрепляют успех и позиции Microsoft на рынке операционных систем. В этом ряду стоит выделить, наверное, Windows Vista – в плане развития графического интерфейса она не выделяется из общего ряда, но прохладно принята общественностью. Windows 8 – пионер операционных систем с оконным интерфейсом, «тайлами» и возможностью управления с помощью мыши, клавиатуры и сенсорного экрана (пальцами), информация из работы [6].

Разрабатываемый визуальный компонент должен выдерживать концепцию графических образов и набор приемов управления в соответствие с теми, что используются в операционной системе, чтобы конечный пользователь программ, опираясь на имеющийся у него опыт работы с элементами графического интерфейса операционной системы, без дополнительных инструкций смог начать работать с ним и правильно управлять его настройками. Это позволит пользователю сэкономить время и нервы при работе с визуальным компонентом.

Очень важно обеспечить возможность использования манипулятора типа «мышь» для управления настройками и параметрами компонент, так как управление посредством манипулятора «мышь» делает компонент более простым в использовании и понятным для конечного пользователя.

Компонент будет выглядеть в виде клетчатого квадратного поля 15x15 с составом из фрукта занимающий одну клетку и змейки, каждый элемент которой будет также занимать одну клетку.

Компонент будет адаптивным и иметь публичные свойства изменения цвета для таких элементов как поле, голова змейки, хвост змейки, фрукт.

2.4 Описание входных и выходных данных компонента

В результатах раздела «Разработка структуры компонента» получена структурная схема компонента, из которой следует необходимость присутствия тех или иных переменных и событий для полноценной работы компонента. Для удобства все свойства и события сведены в табличном виде. Свойства класса представлены в таблице 2.1.

Таблица 2.1 – Свойства класса

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Имя свойства | Тип данных | Тип доступа | Метод доступа на чтение | Метод доступа на запись |
| Score | Int | чтение/запись | Вернёт значение переменной \_score | Используется механизм обновления значения переменной \_score, вызов события RecordScore и изменение значения свойства BestScore |
| BestScore | Int | чтение/запись | Вернёт значение переменной \_bestScore | Используется механизм обновления значения переменной \_ bestScore |
| Lives | Int | чтение/запись | Вернёт значение переменной \_lives | Используется механизм обновления значения переменной \_ lives и вызов события RecordLives |
| MapColor | Color | чтение/запись | Вернёт значение переменной \_mapColor | Используется механизм обновления значения переменной \_ mapColor и обновление картинки на экране |
| FructColor | Color | чтение/запись | Вернёт значение переменной \_ fructColor | Используется механизм обновления значения переменной \_ fructColor и обновление картинки на экране |

Окончание таблицы 2.1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Имя свойства | Тип данных | Тип доступа | Метод доступа на чтение | Метод доступа на запись |
| HeadColor | Color | чтение/запись | Вернёт значение переменной \_ headColor | Используется механизм обновления значения переменной \_ headColor и обновление картинки на экране |
| TailColor | Color | чтение/запись | Вернёт значение переменной \_ tailColor | Используется механизм обновления значения переменной \_ tailColor и обновление картинки на экране |
| GameStatus | Bool | чтение | Вернёт значение переменной \_gameStatus |  |

События класса представлена в таблице 2.2.

Таблица 2.2

|  |  |  |  |
| --- | --- | --- | --- |
| Имя события | Тип данных | Тип доступа | Метод доступа на чтение |
| RecordScore | EventHandler | чтение | Событие изменения счёта. Необходим для изменения счёта на экране |
| RecordBestScore | EventHandler | чтение | Событие изменения лучшего счёта. Необходим для изменения лучшего счёта на экране |
| RecordLives | EventHandler | чтение | Событие изменения количества жизней. Необходим для изменения количества жизней на экране |

2.5 Разработка алгоритмов компонента

В результатах подраздела 2.2, получена структурная схема компонента, из которой следует необходимость присутствия тех или иных методов для полноценной работы компонента. В соответствие с этой структурной схемой в данном разделе будет рассмотрена разработка алгоритмов всех методов.

SetStartPositonObjects – метод отвечающий за инициализацию и размещение головы и первого элемента хвоста змейки, а также за инициализацию и размещение фрукта. Размещение фрукта происходит при помощи метода GenerateFruit. Входные и выходные параметры отсутствуют. Блок-схема метода SetStartPositonObjects приведена на рисунке 2.5.



Рисунок 2.5 – Метод SetStartPositonObjects

GenerateFruit – метод отвечающий за размещение фрукта на экране. В этом методе задаются случайные координаты для фрукта, проверяем что бы они не совпадали с координатами змейки, после чего размещаем фрукт на форме. Входные и выходные параметры отсутствуют. Блок-схема метода GenerateFruit приведена на рисунок 2.6.

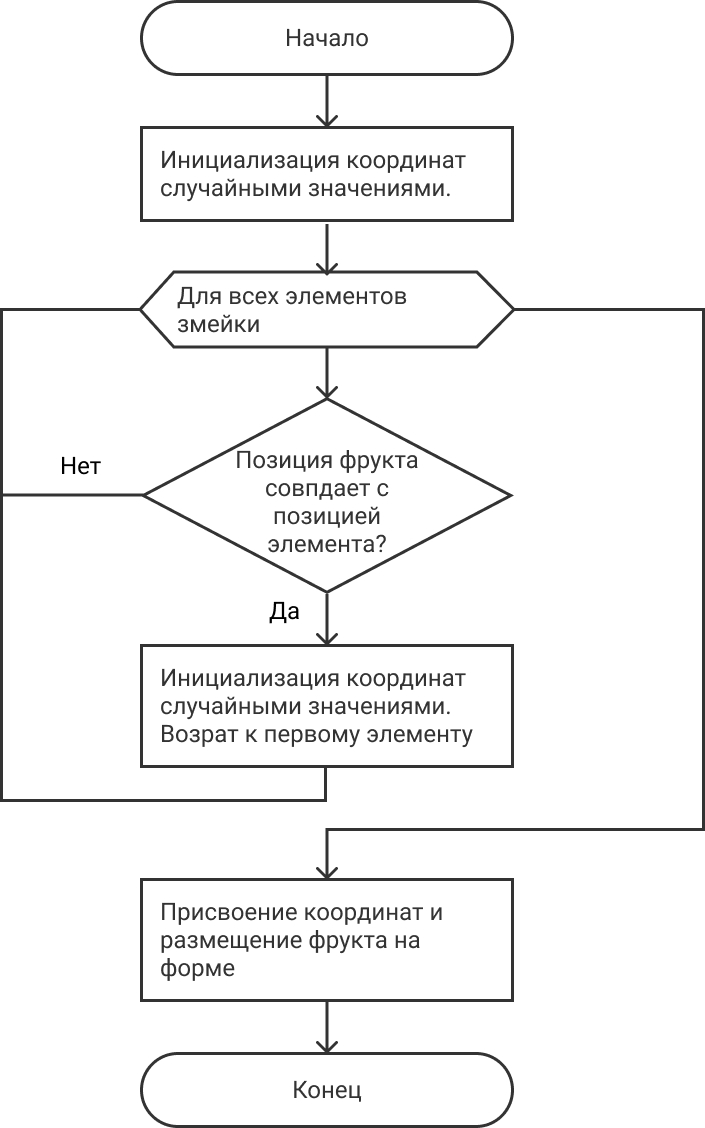


Рисунок 2.6 – Метод GenerateFruit

ClearSnake – метод отвечающий за полное удаление змейки с экрана. Входные и выходные параметры отсутствуют. Блок-схема метода ClearSnake приведена на рисунке 2.7.

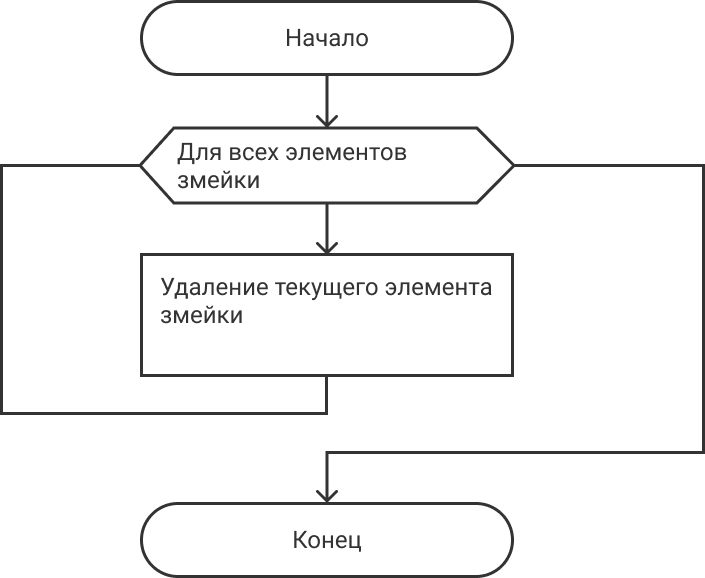


Рисунок 2.7 – Метод ClearSnake

CreateSnakeTailElement – метод инициализации нового элемента змейки. Входные и выходные параметры отсутствуют. Блок-схема метода CreateSnakeTailElement приведена на рисунке 2.8.



Рисунок 2.8 – Метод CreateSnakeTailElement

EatFruit – метод обрабатывающий пересечение змейки с фруктом. Если голова змейки пересекается с фруктом инкрементируется счет, после чего проверяем чтобы счет не превышал 224. При истинном условии вызывается метод CreateSnakeTailElement и GenerateFruit. Входные и выходные параметры отсутствуют. Блок-схема метода EatFruit приведена на рисунке 2.9.

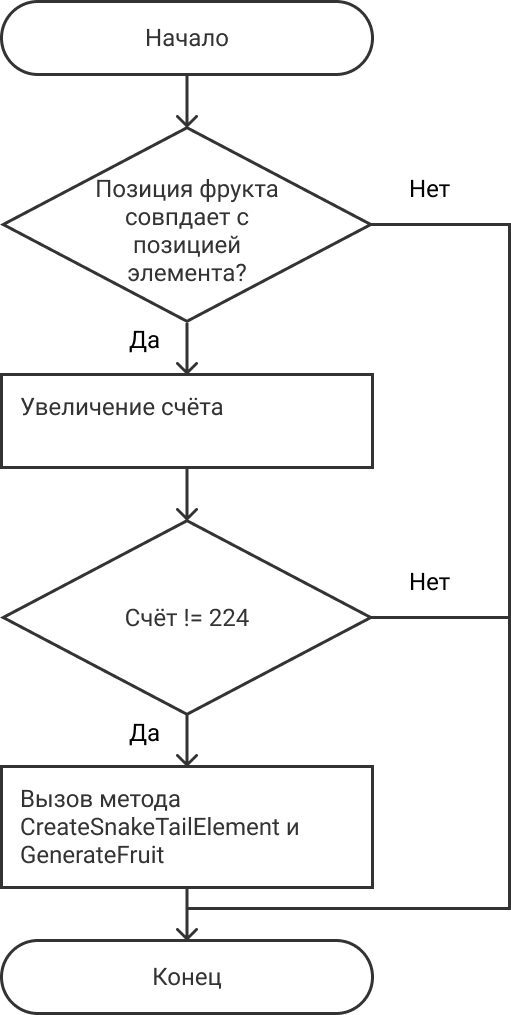


Рисунок 2.9 – Метод EatFruit

InverseDirection – метод меняющий вектор направления на противоположный. Если какой-то из векторов не равен 0, то меняется его значение на противоположное. Входные и выходные параметры отсутствуют. Блок-схема метода InverseDirection приведена на на рисунке 2.10.

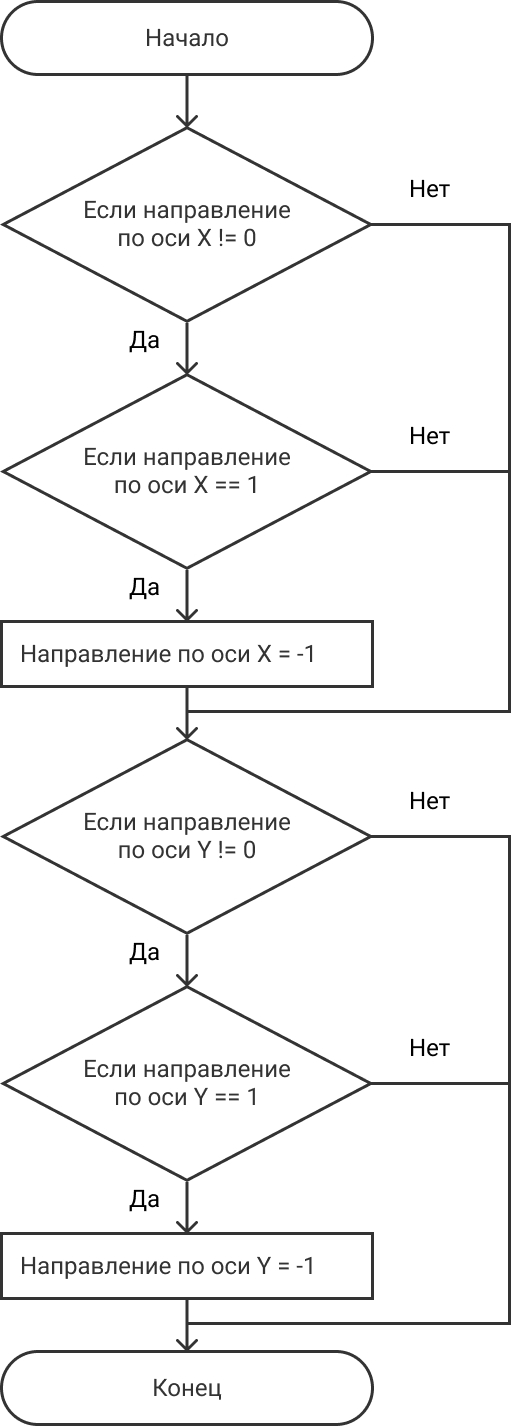


Рисунок 2.10 – Метод InverseDirection

SetNewSnakePosition – метод передвигающий змейку на новую позицию. Цикл проходит по всем элементам начиная с конца хвоста до первого элемента и присваивает значение следующего элемента. После цикла вычисляется новая позиция для головы змейки. Входные и выходные параметры отсутствуют. Блок-схема метода SetNewSnakePosition приведена на рисунке 2.11.

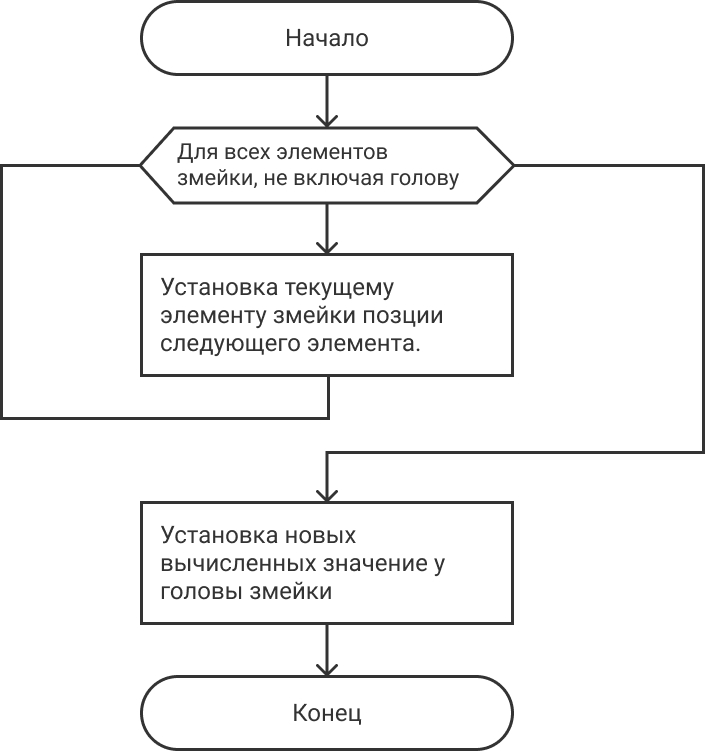


Рисунок 2.11 – Метод SetNewSnakePosition

MoveSnake – метод осуществляющий передвижение змейки. Если пользователь будет пытаться войти во внутрь хвоста, тогда направление змейки измениться на противоположное и сделает шаг, иначе змейка только сделает шаг. Входные и выходные параметры отсутствуют. Блок-схема метода MoveSnake приведена на рисунке 2.12.

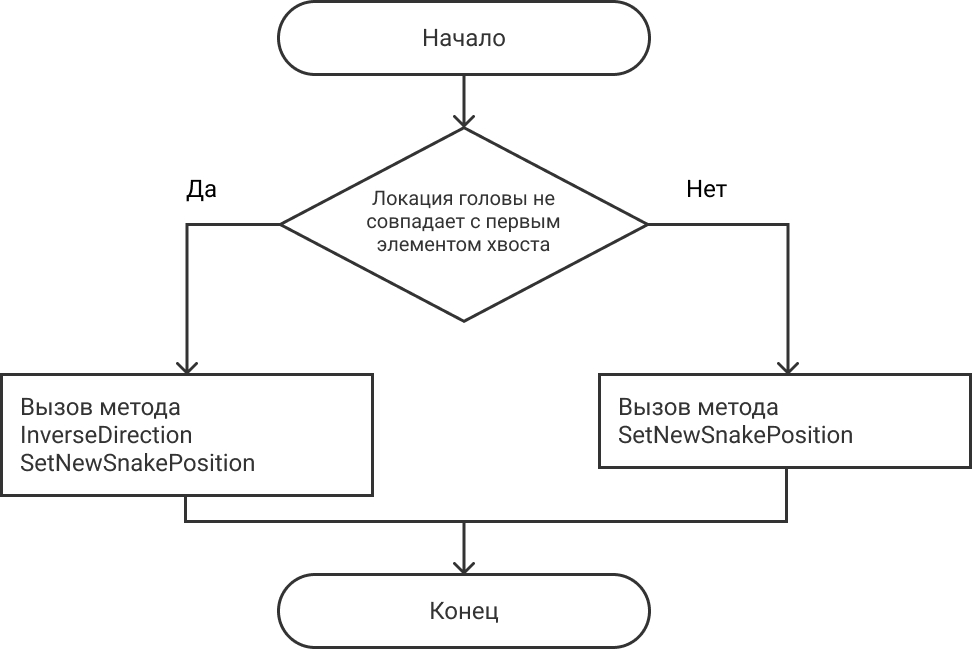


Рисунок 2.12 – Метод MoveSnake

GhostSnake – метод меняющий состояние змейки. Если голова змейки столкнулась с хвостом, змейка переходит в режим «призрак», по истечению 3 секунд она возвращается в прежнее состояние. Входные и выходные параметры отсутствуют. Блок-схема метода GhostSnake приведена на рисунке 2.13.

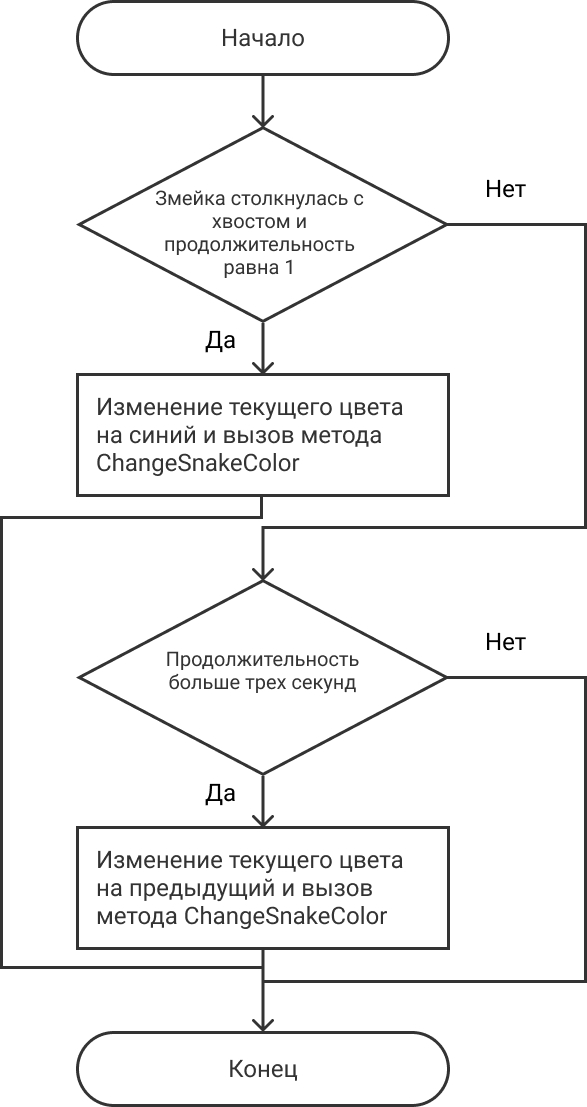


Рисунок 2.13 – Метод GhostSnake

ChangeSnakeColor – метод меняющий цвет змейки на заданный. Входные параметры head и tail типа Color, выходные параметры отсутствуют. Блок-схема метода ChangeSnakeColor приведена на рисунке 2.14.

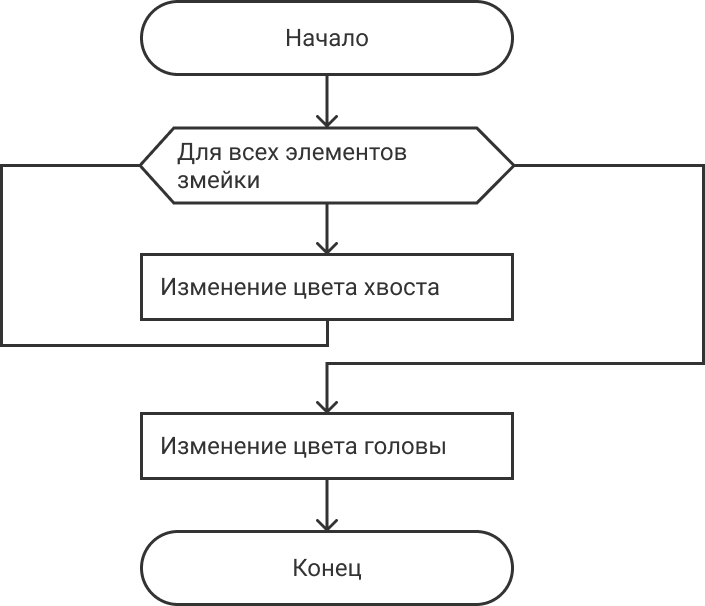


Рисунок 2.14 – Метод ChangeSnakeColor

EndProcess – метод вызывающийся при завершении игры. Он останавливает таймер, удаляет змейку и фрукты и меняет текущий цвет на цвет обычного состояния змейки, а также меняет значение флага состояния игры. В конце вызывается метод Invalidate который обновляет экран. Входные и выходные параметры отсутствуют. Блок-схема метода EndProcess приведена на рисунке 2.15.

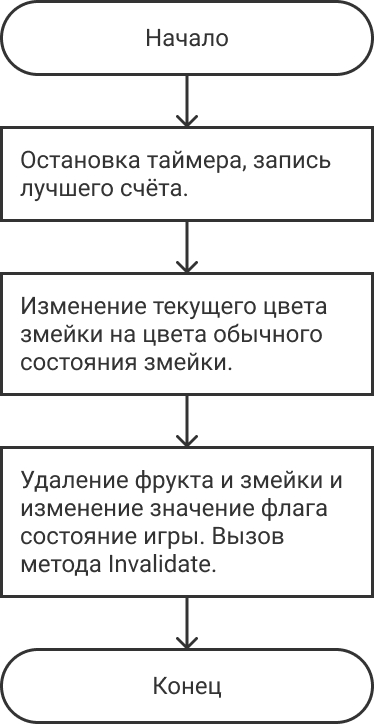


Рисунок 2.15- Метод EndProcess

EatItself – метод обрабатывающий вхождение головы змейки в свой хвост. При условии вхождения головы змейки в свой хвост и состояние змейки не находится в состоянии призрака, проверяется количество жизней. Если количество жизней больше 0, тогда состояние змейки переходит в состояние призрака и количество жизней декрементируется. Иначе вызывается метод EndProcess. В конце вызываются методы DurationGhost и GhostSnake. Входные и выходные параметры отсутствуют. Блок-схема метода EatItself приведена на рисунке 2.16.

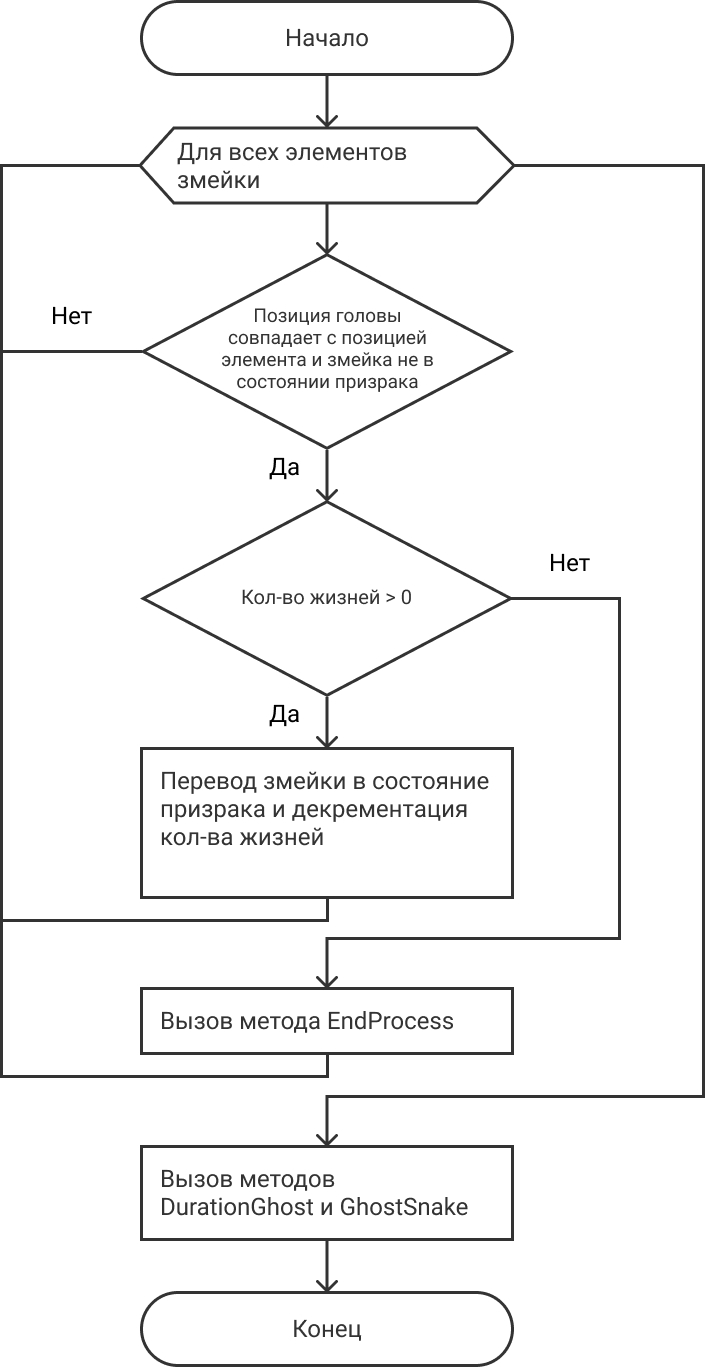


Рисунок 2.16 – Метод EatItself

StartProcess – метод вызывающийся при запуске игры. Он запускает таймер и переводит фокус на компонент. При условии, что игра находится в завершенном состоянии, подготавливаем начальные настройки. В конце обновляем экран. Входные и выходные параметры отсутствуют. Блок-схема метода StartProcess приведена на рисунке 2.17.



Рисунок 2.17 – Метод StartProcess

CheckBorders – метод проверяющий выход змейки за границы поля. Входные и выходные параметры отсутствуют. Блок-схема метода CheckBorders приведена на рисунке 2.18.

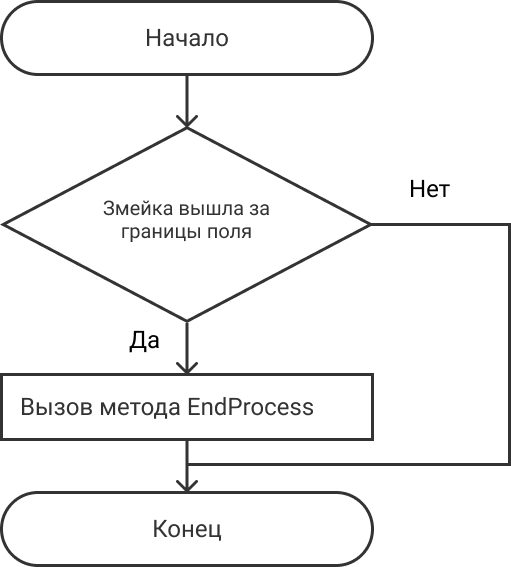


Рисунок 2.18 – Метод CheckBorders

SetBoundsCore – метод отвечающий за регулирование границ компонента. Метод не позволяет задать размер меньше 301 и держит форму квадрата для компонента. Входные параметры x, y, width, height типа Int и specified типа BoundsSpecified, выходные параметры отсутствуют. Блок-схема метода SetBoundsCore приведена на рисунке 2.19.

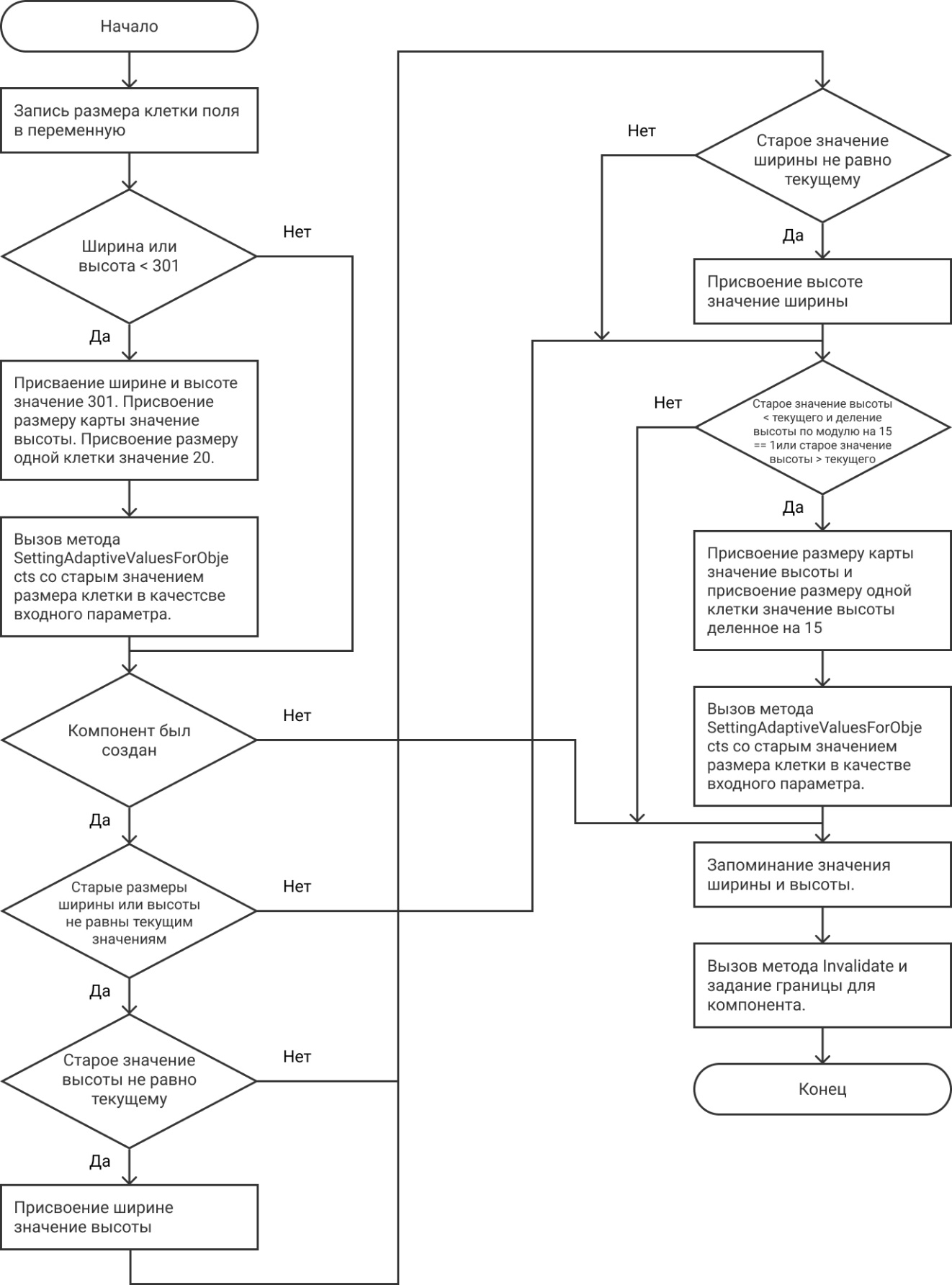


Рисунок 2.19 – Метод SetBoundsCore

SettingAdaptiveValuesForObjects – метод отвечающий за регулирование размеров и положения объектов внутри компонента. Метод принимает во входные параметры старое значение размера одной клетки поля и исходя из этого значения адаптирует размер и положение объектов к новым размерам. Входные параметры oldSizeOfSides типа Int, выходные параметры отсутствуют. Блок-схема метода SettingAdaptiveValuesForObjects приведена на рисунке 2.20.

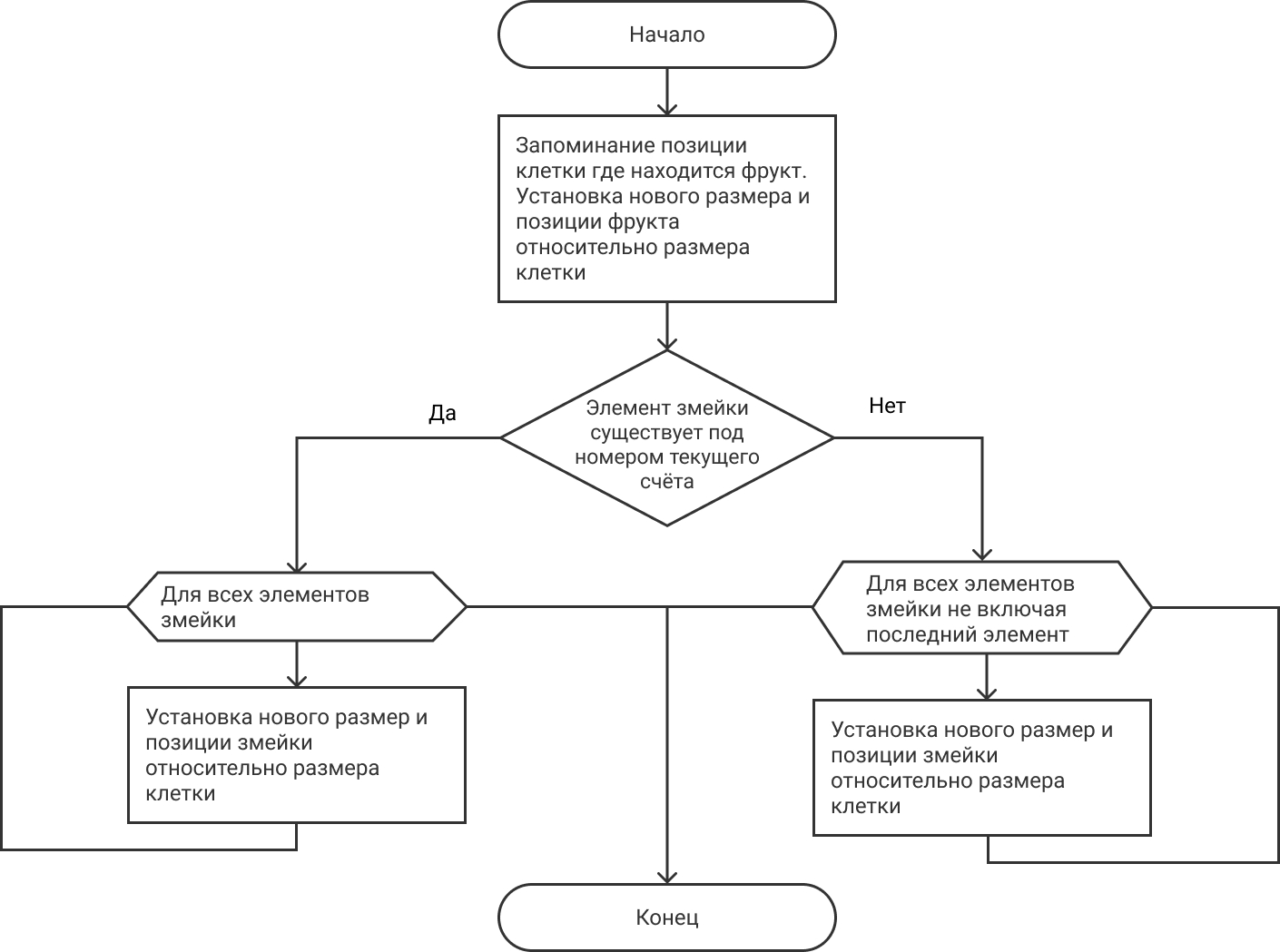


Рисунок 2.20 – Метод SettingAdaptiveValuesForObjects

OKP – метод задающий вектор направления змейки в соответствие с нажатой клавишей на клавиатуре. Входные параметры sender типа object и e типа KeyEventArgs, выходные параметры отсутствуют. Блок-схема метода OKP приведена на рисунке 2.21.

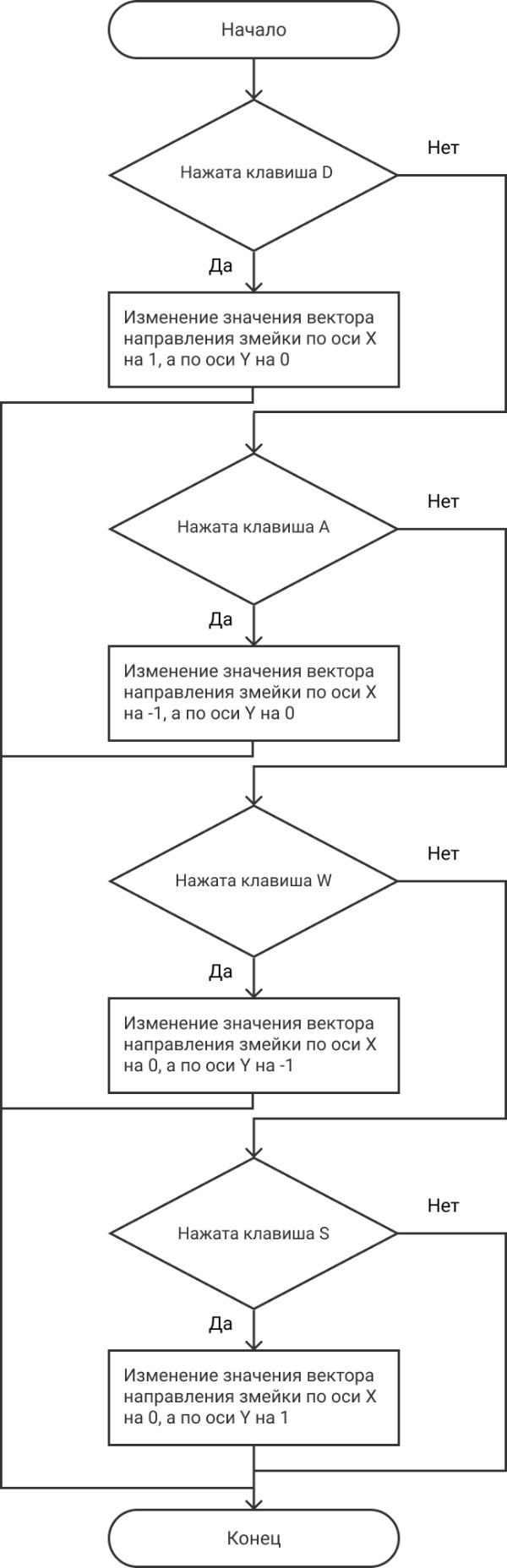


Рисунок 2.21 – Метод OKP

OnPaint – метод отвечающий за прорисовку карты, змейки и фрукта в компоненте. Входные параметры e типа PaintEventArgs, выходные параметры отсутствуют. Блок-схема метода OnPaint приведена на рисунке 2.22.

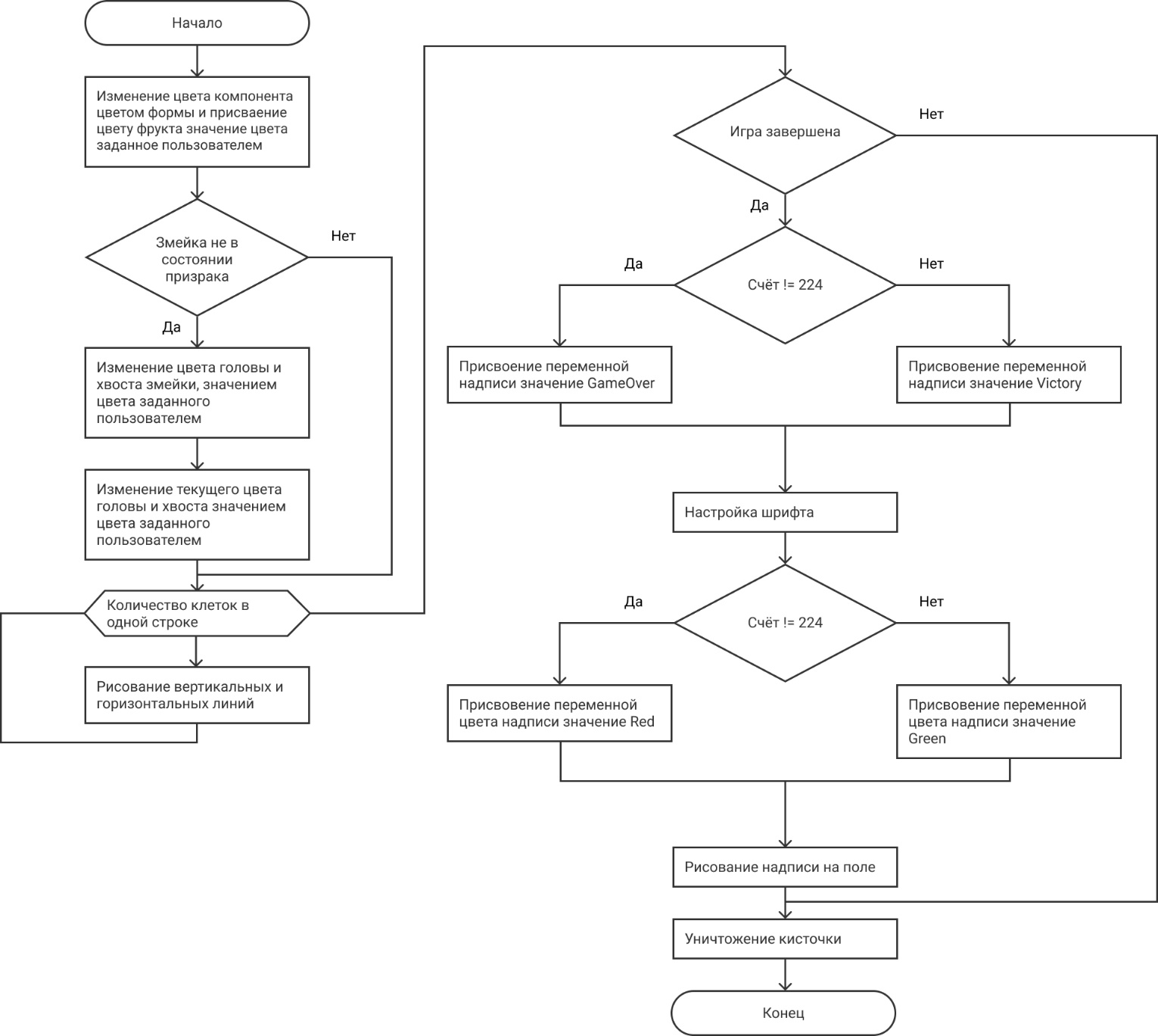


Рисунок 2.22 – Метод OnPaint

Update – метод отвечающий за выполнение основной логики за каждый тик таймера. Входные параметры source типа Object и e типа ElapsedEventArgs выходные параметры отсутствуют. Блок-схема метода Update приведена на рисунке 2.23.

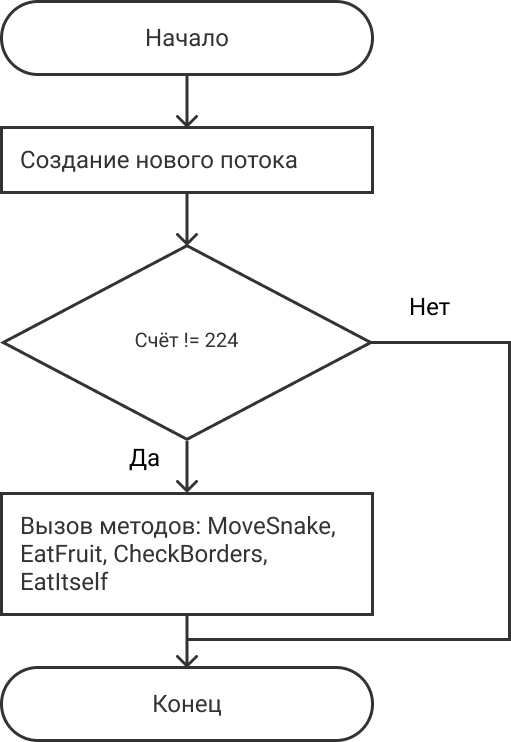


Рисунок 2.23 – Метод Update

Согласно разработанным алгоритмам всех методов был написан программный код, представленный в (приложение А).

3 Технологический раздел

3.1 Отладка компонента

Отладка — этап разработки компьютерной программы, на котором обнаруживают, локализуют и устраняют ошибки, информация из работы [7]. В связи с тем, что почти невозможно составить реальную программу без ошибок, и почти невозможно для достаточно сложной программы быстро найти и устранить все имеющиеся в ней ошибки. Разумно уже при разработке программы на этапах алгоритмизации и программирования готовиться к обнаружению ошибок на стадии отладки принимать профилактические меры по их предупреждению, информация из работы [8].

Тестирование будет происходить через тест кейсы. Тест кейс – это артефакт, описывающий совокупность шагов, конкретных условий и параметров, необходимых для проверки реализации тестируемой функции или её части. При передаче тестировщику тест-кейсов, он должен пройтись по всем его пунктам и выполнить описанные действия, которые должны привести к определенным результатам. информация из работы [9]. Тест кейс для свойств представлен в таблице 3.1.

Таблица 3.1 – Тест-кейс для свойств

|  |  |  |
| --- | --- | --- |
| Имя свойства | Управляющее воздействие | Результат воздействия |
| Score | Изменить значение свойства путём поглощения фрукта змейкой | Меняется значение переменной \_score и Срабатывает событие RecordScore и изменяется значения свойства BestScore, если Score больше текущего BestScore |
| Score | Достигнуть Score равного 224 путём прохождения игры | Меняется значение переменной \_score, происходит вызов метода EndProcess |
| BestScore | Изменить значение свойства путём изменения значения Score в большую сторону чем BestScore | Меняется значение переменной \_bestScore на наибольшее значение счёта |

Окончание таблицы 3.1

|  |  |  |
| --- | --- | --- |
| Имя свойства | Управляющее воздействие | Результат воздействия |
| Lives | Изменить значение свойства при срабатывание метода EatItself | Меняется значение переменной \_lives на заданное значение. Срабатывает событие RecordLives |
| MapColor | Изменение цвета через интерфейс при выделении компонента в форме | Цвет меняется на выбранный |
| FructColor | Изменение цвета через интерфейс при выделении компонента в форме | Цвет меняется на выбранный |
| HeadColor | Изменение цвета через интерфейс при выделении компонента в форме | Цвет меняется на выбранный |
| TailColor | Изменение цвета через интерфейс при выделении компонента в форме | Цвет меняется на выбранный |
| Имя события | Управляющее воздействие | Результат воздействия |
| RecordScore | Изменение значения Score | Запись нового значения счёта в поле на экране |
| RecordBestScore | Завершение игры | Запись нового значения лучшего счёта в поле на экране |
| RecordLives | Изменение значения Lives | Запись нового значения количества жизней в поле на экране |

Тест кейс для методов представлен в таблице 3.2.

Таблица 3.2 – Тест-кейс для методов

|  |  |  |
| --- | --- | --- |
| Имя метода | Управляющее воздействие | Результат воздействия |
| SetBoundsCore | Изменение размеров компонента | Изменение ширины и высоты компонента с сохранением квадратной формы |
| SetStartPositionObjects | Вызвать метод StartProcess | Змейка преместиться в центр компонента, а фрукт займет случайную позицию |

Продолжение таблицы 3.2

|  |  |  |
| --- | --- | --- |
| Имя метода | Управляющее воздействие | Результат воздействия |
| SettingAdaptiveValuesForObjects | Изменение размеров компонента | Изменение размеров объектов компонента относительно новых размеров с сохранением их старых позиций на карте |
| GenerateFruit | Поглощение фрукта змейкой или создании компонента | Генерация фрукта в случайной позиции на карте кроме позиций нахождения змейки |
| ClearSnake | При выходе за границы карты. Столкновении головы змейки с хвостом, при условии что количество жизней равно 0. Присвоение счёту значения 224 | Удаление змейки |
| CheckBorders | Выход змейки за границы карты | Игра завершится |
| EatItself | Столкновении головы змейки с хвостом | Если количество жизней больше 0, то уменьшение количества жизней и изменение состояния змейки. Иначе завершение игры |
| CreateSnakeTailElement | Поглощение фрукта змейкой | Создание нового элемента хвоста |
| EatFruit | Поглощение фрукта змейкой | Увеличение счёта и если счёт не равен 224, создание нового фрукта и увеличение змейки |
| InverseDirection | Изменения вектора движения во внутрь хвоста змейки | Изменение вектора движения на противоположный |

Продолжение таблицы 3.2

|  |  |  |
| --- | --- | --- |
| Имя метода | Управляющее воздействие | Результат воздействия |
| SetNewSnakePosition | Программно вызвать метод | Сдвигает змейку относительно вектора движения |
| MoveSnake | Программно вызвать метод | Движение змейки по заданному вектору движения или неизменная траектория движения при попытке войти во внутрь змейки |
| DurationGhost | Программно вызвать метод 20 раз | Поддерживает состояние призрака 3 секунды |
| ChangeSnakeColor | Войти во хвост змейки | Изменение цвета змейки на заданный |
| GhostSnake | Программно вызвать метод | В зависимости от состояния змейки меняет цвет змейки |
| Update | Программно вызвать метод | Передвижение змейки и проверка выхода за границы и входа головы змейки во хвост. Проверка на съедания фрукта |
| StartProcess | Программно вызвать метод | Переводит фокус на компонент. Размещает змейку по центру и создает фрукт в случайно позиции |
| EndProcess | При выходе за границы карты. Столкновении головы змейки с хвостом, при условии что количество жизней равно 0. Дойти до конца игры | Удаляет фрукт и змейку. Выводи сообщение об окончании игры |
| OKP | При нажатии клавиш (w,a,s,d) на клавиатуре | При нажатии «w» змейка движется вверх, «a» движется в право, «s» движется вниз, «d» в лево |

Окончание таблицы 3.2

|  |  |  |
| --- | --- | --- |
| Имя метода | Управляющее воздействие | Результат воздействия |
| OnPaint | При изменении свойств цвета или размеров компонента.  Остановке и запуске игры | Задает цвет для фрукта, и задает текущий цвет и цвет для головы и хвоста змейки при условии, что змейка не в состоянии призрака.  Прорисовывает карту и по завершении игры выводит соответствующее сообщение в зависимости от результата игры(победа или проигрыш) |

3.2 Инструкция по установке компонента

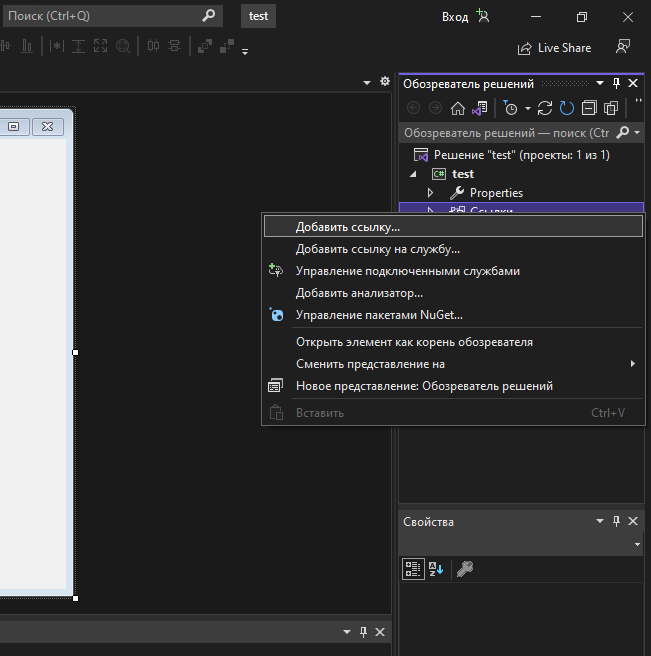
Для установки компонента в проект необходимо в обозревателе решений щелчком правой кнопки мыши на ветке «Ссылки» в структуре текущего проекта вызвать контекстное меню. В появившемся меню требуется выбрать команду «добавить ссылку» из группы команд «ссылки», как показано на рисунке 3.1  


Рисунок 3.1 – Контекстное меню

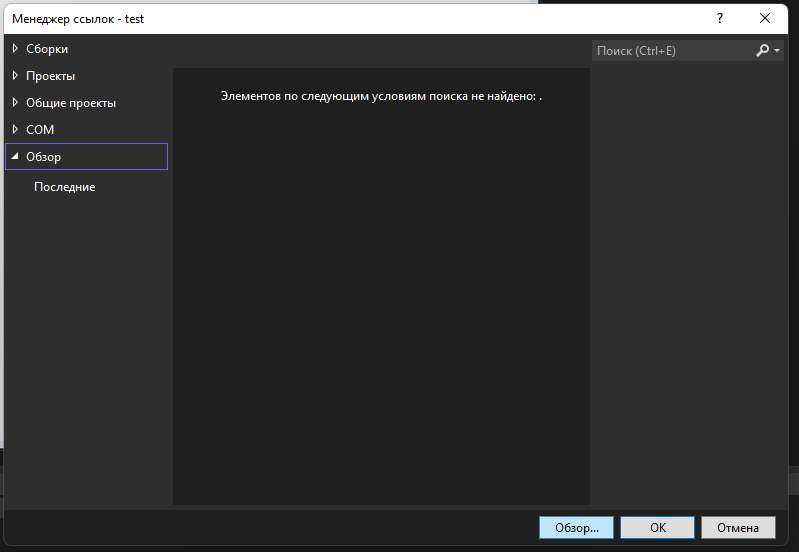
После описанных действий на экране должно появиться окно Менеджера ссылок. Его внешний вид приведен на рисунке 3.2. В нем, с левой стороны необходимо выделить раздел настроек «Обзор».

Рисунок 3.2 – Менеджер ссылок

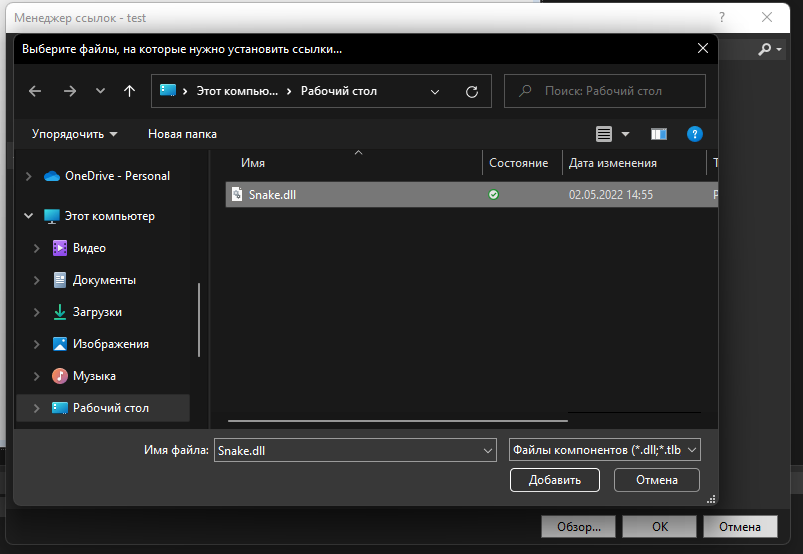
В появившемся на экране окне, представленном на рисунке 3.3, предназначенном для выбора файлов, требуется найти файловый пакет с компонентом и выбрать его.  


Рисунок 3.3 – Файловый пакет в проводнике

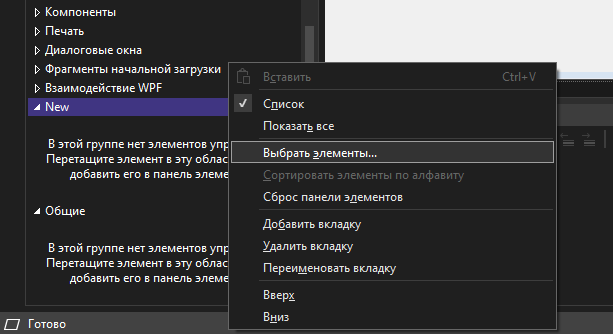
После выбора файлового пакета в панели элементов выбирается новая вкладка, и вызывается на ней контекстное меню, где выбирается команда «Выбрать элементы», как на рисунке 3.4  


Рисунок 3.4 – Панель элементов

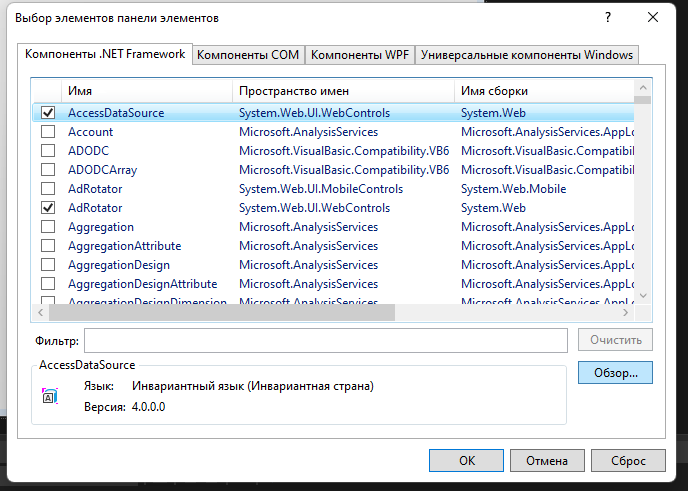
В новом появившемся окне «Выбор элементов панели элементов», в правом углу выбираем команду «Обзор» как на рисунке 3.5  


Рисунок 3.5 – Окно «Выбор элементов панели элементов»

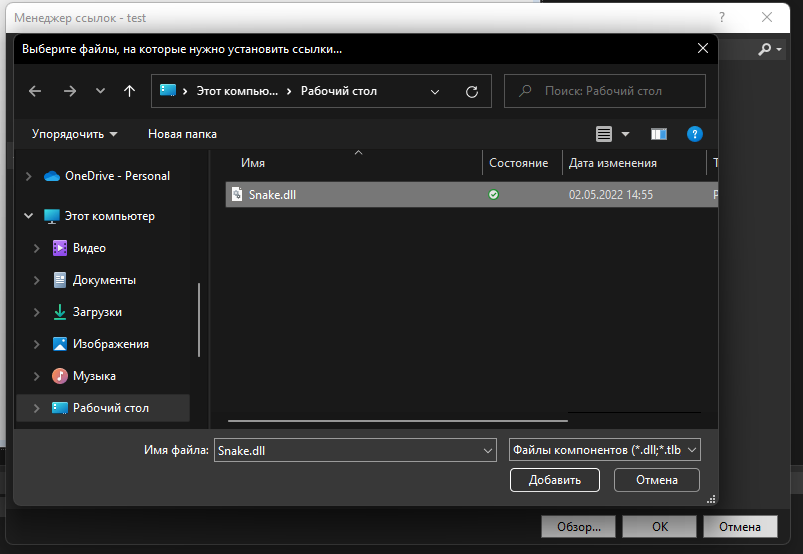
Далее в появившемся окне проводника, необходимо найти файловый пакет с компонентом и выбрать его, как показано на рисунке 3.6.  


Рисунок 3.6 – Файловый пакет в проводнике

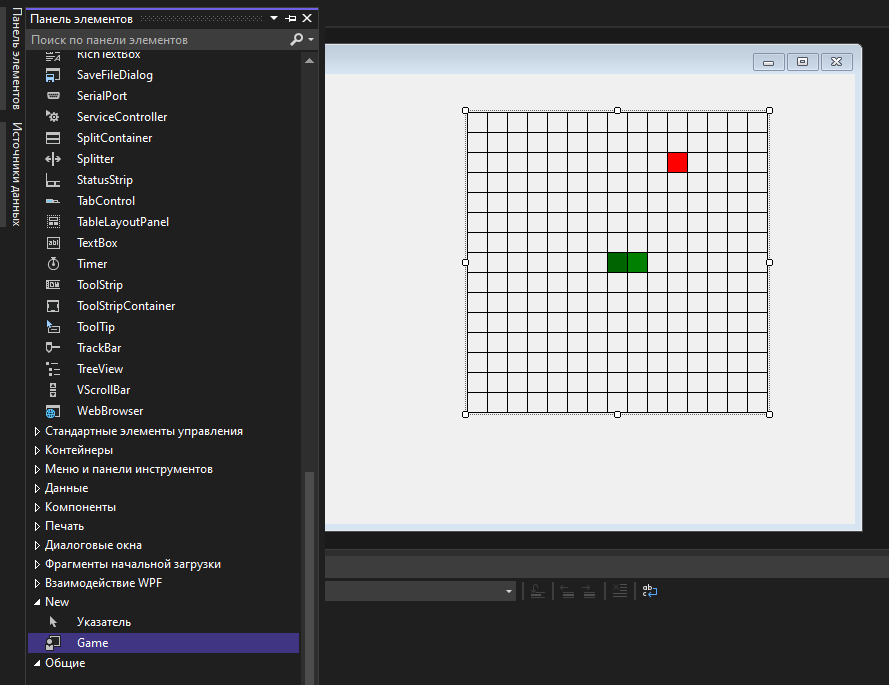
В панели элементов выбирается вкладка «New», откуда перемещаем компонент на форму. Результат представлен на рисунке 3.7  


Рисунок 3.7 – Панель элементов

3.3 Инструкция по эксплуатации компонента

Для удобства вся инструкция по эксплуатации визуального компонента сведена в табличный вид. Все свойства компонента перечислены в таблице 3.1

Таблица 3.1 – Свойства компонента

|  |  |  |  |
| --- | --- | --- | --- |
| Имя свойства | Тип данных | Тип доступа | Назначение |
| Score (Счёт игры) | Int | чтение | Свойство необходимо для отображения текущего счёта |
| BestScore (Лучший счёт за игру) | Int | чтение | Свойство необходимо для отображения лучшего счёта |
| Lives (Кол-во жизней) | Int | чтение/запись | Свойство необходимо для отображения кол-ва жизней |
| MapColor (Цвет бордюров) | Color | чтение/запись | Свойство необходимо для задания цвета бордюров |
| FructColor (Цвет фрукта) | Color | чтение/запись | Свойство необходимо для задания цвета фрукта |
| HeadColor (Цвет головы змейки) | Color | чтение/запись | Свойство необходимо для задания цвета головы змейки |
| TailColor (Цвет хвоста змейки) | Color | чтение/запись | Свойство необходимо для задания цвета хвоста змейки |
| GameStatus (Статус игры) | Bool | чтение | Свойство необходимо для отображения текущего статуса игры (игра идет или закончена) |

События компонента представлены в таблице 3.2.

Таблица 3.2 – События компонента

|  |  |
| --- | --- |
| Имя события | Назначение |
| RecordScore | Событие срабатывает при изменении счёта |
| RecordBestScore | Событие срабатывает при завершении игры |
| RecordLives | Событие срабатывает при изменении количества жизней |

Публичные методы компонента представлены в таблице 3.3.

Таблица 3.3 – Публичные методы компонента

|  |  |  |
| --- | --- | --- |
| Имя события | Входные и выходные параметры | Назначение |
| EndProcess | Нет входных и выходных параметров | Метод служит для принудительного завершения игры |
| StartProcess | Нет входных и выходных параметров | Метод служит для старта игры |

4 Раздел охраны труда

Охрана труда – это целая система законодательных и нормативно-правовых актов, технических, гигиенических, лечебно-профилактических мероприятий и средств, которые обеспечивают безопасность, сохранение здоровья и работоспособности человека в процессе труда. В наши дни труд стал более интенсивным и требует огромных затрат умственной, эмоциональной и физической нагрузок.

На рабочем месте программист осуществляет трудовую деятельность и проводит большую часть рабочего времени. Правильная организация рабочего места программиста повышает производительность труда от 8 до 20%. Следуя рекомендациям ГОСТ 12.2.032-78, необходимо организовать рабочее место таким образом, чтобы взаимное расположение всех его элементов соответствовало физическим и психологическим требованиям. Главные элементы рабочего места программиста – это письменный стол и кресло. Рабочее место организуется в соответствии с ГОСТ 12.2.032-78, информация из работы [10].

Площадь рабочего места с компьютером с жидкокристаллическим или плазменным экраном должна быть не менее 4,5 кв. м, а расстояние между столами с мониторами (от тыла одного монитора до экрана другого) не менее 2 м. Монитор должен располагаться на расстоянии 50-70 см от глаз программиста. Параметры рабочего стола сотрудника: возможность регулировки высоты рабочего стола, или точная высота — 72,5 см, ширина — 80, 100, 120 или 140 см, глубина рабочего стола 80 или 100 см, высота и ширина пространства под столешницей (для ног) – не менее 50 см, глубина на уровне колен не менее 45 см, а на уровне вытянутых ног не менее 65 см.

Правильное освещение рабочего места – это очень важный момент в трудовой деятельности человека, влияющий на эффективность труда, при этом такой момент предупреждает травматизм и профессиональные заболевания. При недостаточном освещении приходится напрягать зрение, при этом ослабляется внимание и это приводит к наступлению преждевременной утомленности. Слишком яркое освещение тоже плохо, так как оно вызывает ослепление, раздражение и резь в глазах. При искусственном освещении, источниками света служат два вида ламп: лампы накаливания и люминесцентные.

Известно, что шум ухудшает условия труда и оказывает вредное воздействие на организм человека. Согласно ГОСТ 12.1.003-88 «Шум для помещений расчетчиков и программистов, уровни шума не должны превышать соответственно: 71, 61, 54, 49, 45, 42, 40, 38 дБ», информация из работы [11].

При работе компьютерной техники выделяется много тепла, что может привести к пожароопасной ситуации. Источниками зажигания так же могут служить приборы, применяемые для технического обслуживания, устройства электропитания, кондиционеры воздуха. Серьёзную опасность представляют различные электроизоляционные материалы, используемые для защиты от механических воздействий отдельных радиодеталей. В связи с этим, участки, на которых используется компьютерная техника, по пожарной опасности относятся к категории пожароопасных “В”. При пожаре люди должны покинуть помещение в течение минимального времени. В помещениях с компьютерной техникой, недопустимо применение воды и пены ввиду опасности повреждения или полного выхода из строя дорогостоящего электронного оборудования. Для тушения пожаров необходимо применять углекислотные и порошковые огнетушители, которые обладают высокой скоростью тушения, большим временем действия, возможностью тушения электроустановок, высокой эффективностью борьбы с огнем. Воду разрешено применять только во вспомогательных помещениях, информация из работы [12].

Заключение

{Заключение не успел проверить перечитаю позднее!}

По итогу работы был разработан компонент для игры в змейку. Компонент является адаптивным и представляет из себя карту 15х15 со змейкой и фруктом. Для карты, змейки и фрукта пользователь может задать любые цвета на свое усмотрение. В компоненте присутствует система жизней, при которой игрок теряет одну жизнь при столкновении змейки с хвостом, и переходит в состояние призрака, в котором змейка на протяжении 3 секунд может беспрепятственно двигаться по своему хвосту. По истечению жизней игра будет завершена при следующем столкновении с хвостом. При выходе змейки за границы карты игра завершается. В зависимости от результата игры в окне будет появляться сообщение, сигнализирующее о том, что игра была завершена с определенным результатом. Если игрок достиг цели игры и набрал необходимое количество очков ему покажут сообщение об успешном прохождении игры. В противном случае его уведомят о том, что игра была завершена без достижения главной цели игры. В конце каждый игры сохраняется лучший счёт.

Главным достоинством разработанного компонента можно считать систему жизней, которая может облегчить игровой процесс пользователя и дать перспективы в дальнейшем для реализации новых различных игровых механика на её основе. К примеру жизнями можно манипулировать для задания различных уровней сложности, которые может выбрать для себя пользователь.

У компонента есть перспективы развития в плане визуализации какой-либо анимации в сопровождении с звуковыми эффектами, а также в реализации системы уровней с разными картами, где будет передвигаться змейка.

Змейку можно использовать в развлекательных целях, чтобы сократить время в ожидании чего-либо или просто развлечь себя. Так же её можно использовать в учебных целях. Змейка может проиллюстрировать различные методы рисования, объектно-ориентированный подход, работу с таймером и потоками. В ней используются различного рода механики, на которые можно опираться при разработке других компонентов.

Список источников

1. <https://ru.wikipedia.org/wiki/История_языков_программирования>
2. <https://ruprogi.ru/software/visual-studio>
3. <https://gb.ru/posts/c_sharp_ides>
4. <https://playsnake.net/snake-game/>
5. <https://studme.org/223367/informatika/osnovnye_komponenty_interfeysov>
6. <https://total.kz/ru/news/biznes/istoriya_graficheskih_interfeyso>
7. <https://studopedia.ru/22_29871_neobhodimost-otladki-programmnogo-produkta.html>
8. <https://infopedia.su/4x1ec5.html>
9. <https://sergeygavaga.gitbooks.io/kurs-lektsii-testirovanie-programnogo-obespecheni/content/lektsiya-4-ch3.html>
10. <https://www.retail.ru/rbc/pressreleases/tsentr-povysheniya-kvalifikatsii-lider-organizatsiya-rabochego-mesta-ofisnogo-rabotnika/>
11. <https://xn--d1aux.xn--p1ai/opisanie-rabochego-mesta-programmista-na-predpriyatii/>
12. <https://studopedia.ru/8_107307_osveshchenie-pomeshcheniy-vichislitelnih-tsentrov.html>

Приложение А

Программный кодкласса Snake

using System;

using System.Collections.Generic;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading;

using System.Threading.Tasks;

using System.Timers;

using System.Windows.Forms;

namespace Snake

{

public class Game : Control

{

protected int \_lives = 1;

protected bool \_snakeStatus = true;

protected bool \_gameStatus = false;

protected int \_currentDurationGhost = 0;

protected PictureBox \_fruit;

protected PictureBox[] \_snake = new PictureBox[225];

protected int \_dirX = -1;

protected int \_dirY = 0;

protected int \_mapSize = 301;

protected int \_sizeOfSides = 20;

protected int \_score = 1;

protected int \_bestScore = 0;

private int \_oldWidth = 0;

private int \_oldHeight = 0;

protected Color \_mapColor = Color.Black;

protected Color \_fructColor = Color.Red;

protected Color \_headColor = Color.DarkGreen;

protected Color \_tailColor = Color.Green;

protected Color \_currentHeadColor;

protected Color \_currentTailColor;

protected System.Timers.Timer \_timer;

protected event EventHandler \_recordScore;

protected event EventHandler \_recordBestScore;

protected event EventHandler \_recordLives;

public event EventHandler RecordScore

{

add { \_recordScore += value; }

remove { \_recordScore -= value; }

}

protected void OnRecordScore()

{

\_recordScore?.Invoke(this, new EventArgs());

}

public event EventHandler RecordBestScore

{

add { \_recordBestScore += value; }

remove { \_recordBestScore -= value; }

}

protected void OnRecordBestScore()

{

\_recordBestScore?.Invoke(this, new EventArgs());

}

public event EventHandler RecordLives

{

add { \_recordLives += value; }

remove { \_recordLives -= value; }

}

protected void OnRecordLives()

{

\_recordLives?.Invoke(this, new EventArgs());

}

public int Score

{

get { return \_score; }

private set

{

if (\_score != value)

{

\_score = value;

OnRecordScore();

BestScore = (BestScore < \_score) ? \_score : BestScore;

}

if (value == 224)

{

EndProcess();

}

}

}

public int BestScore

{

get { return \_bestScore; }

private set

{

if (\_bestScore != value)

{

\_bestScore = value;

}

}

}

public int Lives

{

get { return \_lives; }

private set

{

if (\_lives != value)

{

\_lives = value;

OnRecordLives();

}

}

}

public Color MapColor

{

get

{

return \_mapColor;

}

set

{

if (\_mapColor != value)

{

\_mapColor = value;

Invalidate();

}

}

}

public Color FructColor

{

get

{

return \_fructColor;

}

set

{

if (\_fructColor != value)

{

\_fructColor = value;

Invalidate();

}

}

}

public Color HeadColor

{

get

{

return \_headColor;

}

set

{

if (\_headColor != value)

{

\_headColor = value;

Invalidate();

}

}

}

public Color TailColor

{

get

{

return \_tailColor;

}

set

{

if (\_tailColor != value)

{

\_tailColor = value;

Invalidate();

}

}

}

public bool GameStatus

{

get

{

return \_gameStatus;

}

}

//Метод отвечающий за регулирование границ компонента

protected override void SetBoundsCore(int x, int y, int width, int height, BoundsSpecified specified)

{

//Запоминаем размеры одной клетки

int oldSizeOfSides = \_sizeOfSides;

//Проверка минимальных значений высоты и ширины

if (width < 301 || height < 301)

{

width = 301;

height = 301;

\_mapSize = height;

\_sizeOfSides = 20;

SettingAdaptiveValuesForObjects(oldSizeOfSides);

}

//Установка адаптивных значений

if (Created)

{

if (\_oldWidth == width || \_oldHeight == height)

{

if (\_oldHeight != height)

{

width = height;

}

if (\_oldWidth != width)

{

height = width;

}

}

if ((\_oldHeight < height && height % 15 == 1) || \_oldHeight > height)

{

\_mapSize = height;

\_sizeOfSides = height / 15;

SettingAdaptiveValuesForObjects(oldSizeOfSides);

}

}

//Запоминаем значения ширины и высоты

\_oldWidth = width;

\_oldHeight = height;

Invalidate();

base.SetBoundsCore(x, y, width, height, specified);

}

public Game() : base()

{

\_currentHeadColor = HeadColor;

\_currentTailColor = TailColor;

SetStartPositionObjects();

\_timer = new System.Timers.Timer(150);

\_timer.AutoReset = true;

\_timer.Elapsed += Update;

\_timer.Enabled = false;

this.KeyDown += new KeyEventHandler(OKP);

}

//Метод отвечающий за инициализацию и размещение головы и первого эле-мента хвоста змейки, а также за инициализацию и размещение фрукта

protected void SetStartPositionObjects()

{

\_snake[0] = new PictureBox

{

Location = new Point(\_sizeOfSides \* 7 + 1, \_sizeOfSides \* 7 + 1),

Size = new Size(\_sizeOfSides - 1, \_sizeOfSides - 1),

BackColor = HeadColor

};

this.Controls.Add(\_snake[0]);

\_snake[1] = new PictureBox

{

Location = new Point(\_sizeOfSides \* 7 + 1 + \_sizeOfSides, \_sizeOfSides \* 7 + 1),

Size = new Size(\_sizeOfSides - 1, \_sizeOfSides - 1),

BackColor = TailColor

};

this.Controls.Add(\_snake[1]);

\_fruit = new PictureBox

{

BackColor = FructColor,

Size = new Size(\_sizeOfSides - 1, \_sizeOfSides - 1)

};

GenerateFruit();

}

//Метод отвечающий за регулирование размеров и положения объектов внутри компонента

private void SettingAdaptiveValuesForObjects(int oldSizeOfSides)

{

int fruitLocX = \_fruit.Location.X / oldSizeOfSides;

int fruitLocY = \_fruit.Location.Y / oldSizeOfSides;

\_fruit.Size = new Size(\_sizeOfSides - 1, \_sizeOfSides - 1);

\_fruit.Location = new Point(\_sizeOfSides \* fruitLocX + 1, \_sizeOfSides \* fruitLocY + 1);

if (\_snake[Score] != null)

{

for (int i = 0; i <= Score; i++)

{

\_snake[i].Size = new Size(\_sizeOfSides - 1, \_sizeOfSides - 1);

\_snake[i].Location = new Point(\_sizeOfSides \* (\_snake[i].Location.X / oldSizeOfSides) + 1,

\_sizeOfSides \* (\_snake[i].Location.Y / oldSizeOfSides) + 1);

}

}

else //В случае если новый объект хвоста змейки не успел создаться, обновляем свойства змейки без его учета

{

for (int i = 0; i < Score; i++)

{

\_snake[i].Size = new Size(\_sizeOfSides - 1, \_sizeOfSides - 1);

\_snake[i].Location = new Point(\_sizeOfSides \* (\_snake[i].Location.X / oldSizeOfSides) + 1,

\_sizeOfSides \* (\_snake[i].Location.Y / oldSizeOfSides) + 1);

}

}

}

//Метод отвечающий за размещение фрукта на экране

protected void GenerateFruit()

{

int posFructX, posFructY;

Random r = new Random();

posFructX = r.Next(0, \_mapSize - \_sizeOfSides);

posFructX = posFructX / \_sizeOfSides \* \_sizeOfSides;

posFructX++;

posFructY = r.Next(0, \_mapSize - \_sizeOfSides);

posFructY = posFructY / \_sizeOfSides \* \_sizeOfSides;

posFructY++;

//Пробегаеся по всей змейки

for (int i = 0; i <= Score; i++)

{

//Если поизиция фрукта совпадает с позициями змейки генерируем новые координаты и повторяем процедуру

if (\_snake[i].Location.X == posFructX && \_snake[i].Location.Y == posFructY)

{

posFructX = r.Next(0, \_mapSize - \_sizeOfSides);

posFructX = posFructX / \_sizeOfSides \* \_sizeOfSides;

posFructX++;

posFructY = r.Next(0, \_mapSize - \_sizeOfSides);

posFructY = posFructY / \_sizeOfSides \* \_sizeOfSides;

posFructY++;

i = 0;

}

}

\_fruit.Location = new Point(posFructX, posFructY);

this.Controls.Add(\_fruit);

}

//Метод отвечающий за полное удаление змейки с экрана

protected void ClearSnake()

{

for (int i = 0; i <= Score; i++)

this.Controls.Remove(\_snake[i]);

}

//Метод проверяющий выход змейки за границы поля

protected void CheckBorders()

{

if ((\_snake[0].Location.X < 0) || (\_snake[0].Location.X > \_mapSize - \_sizeOfSides) ||

(\_snake[0].Location.Y < 0) || (\_snake[0].Location.Y > \_mapSize - \_sizeOfSides))

{

EndProcess();

}

}

//Метод обрабатывающий вхождение головы змейки в свой хвост

protected void EatItself()

{

for (int i = 2; i < Score; i++)

{

if (\_snake[0].Location == \_snake[i].Location && \_snakeStatus)

{

if (Lives > 0)

{

\_snakeStatus = false;

--Lives;

} else

{

EndProcess();

}

}

}

DurationGhost();

GhostSnake();

}

//Метод инициализации нового элемента змейки

protected void CreateSnakeTailElement()

{

\_snake[Score] = new PictureBox

{

Location = new Point(\_snake[Score - 1].Location.X - \_sizeOfSides \* \_dirX,

\_snake[Score - 1].Location.Y - \_sizeOfSides \* \_dirY),

Size = new Size(\_sizeOfSides - 1, \_sizeOfSides - 1),

BackColor = \_currentTailColor

};

this.Controls.Add(\_snake[Score]);

}

//Метод обрабатывающий пересечение змейки с фруктом

protected void EatFruit()

{

if (\_snake[0].Location.X == \_fruit.Location.X && \_snake[0].Location.Y == \_fruit.Location.Y)

{

Score++;

//Если игра закончена не нужно создавать новые объекты

if (Score != 224)

{

CreateSnakeTailElement();

GenerateFruit();

}

}

}

//Метод меняющий вектор направления на противоположный

protected void InverseDirection()

{

if (\_dirX != 0)

{

\_dirX = (\_dirX == 1) ? -1 : 1;

}

if (\_dirY != 0)

{

\_dirY = (\_dirY == 1) ? -1 : 1;

}

}

//Метод передвигающий змейку на новую позицию

protected void SetNewSnakePosition()

{

for (int i = Score; i >= 1; i--)

{

\_snake[i].Location = \_snake[i - 1].Location;

}

var snakeLocationX = \_snake[0].Location.X + \_dirX \* \_sizeOfSides;

var snakeLocationY = \_snake[0].Location.Y + \_dirY \* \_sizeOfSides;

\_snake[0].Location = new Point(snakeLocationX, snakeLocationY);

}

//Метод осуществляющий передвижение змейки

protected void MoveSnake()

{

if ((\_snake[0].Location.X + \_dirX \* \_sizeOfSides != \_snake[1].Location.X) ||

(\_snake[0].Location.Y + \_dirY \* \_sizeOfSides != \_snake[1].Location.Y))

{

SetNewSnakePosition();

}

else

{

InverseDirection();

SetNewSnakePosition();

}

}

//Метод меняющий состояние змейки

protected void DurationGhost()

{

if (!\_snakeStatus && \_currentDurationGhost <= 20)

{

++\_currentDurationGhost;

}

else

{

\_currentDurationGhost = 0;

\_snakeStatus = true;

}

}

//Метод меняющий цвет змейки на заданный

protected void ChangeSnakeColor(Color head, Color tail)

{

for (int i = 0; i <= \_score; i++)

{

\_snake[i].BackColor = tail;

}

\_snake[0].BackColor = head;

}

protected void GhostSnake()

{

if (!\_snakeStatus && \_currentDurationGhost == 1)

{

\_currentHeadColor = Color.DarkBlue;

\_currentTailColor = Color.Blue;

ChangeSnakeColor(\_currentHeadColor, \_currentTailColor);

} else if (\_currentDurationGhost > 20)

{

\_currentHeadColor = HeadColor;

\_currentTailColor = TailColor;

ChangeSnakeColor(HeadColor, TailColor);

}

}

//Метод отвечающий за выполнение основной логики за каждый тик таймера

protected void Update(Object source, ElapsedEventArgs e)

{

Thread thread = new Thread(() =>

{

Invoke((Action)(() =>

{

//Если игра закончена не нужно делать лишних действий

if (Score != 224)

{

MoveSnake(); EatFruit(); CheckBorders(); EatItself();

}

}));

});

thread.Start();

}

//Метод вызывающийся при запуске игры

public void StartProcess()

{

\_timer.Enabled = true;

this.Focus();

if (\_gameStatus)

{

\_gameStatus = false;

Score = 1;

Lives = 1;

SetStartPositionObjects();

}

Invalidate();

}

//Метод вызывающийся при завершении игры

public void EndProcess()

{

\_timer.Stop();

OnRecordBestScore();

\_currentHeadColor = HeadColor;

\_currentTailColor = TailColor;

ClearSnake();

Controls.Remove(\_fruit);

\_gameStatus = true;

Invalidate();

}

//Метод задающий вектор направления змейки в соответствие с нажатой клави-шей на клавиатуре

protected void OKP(object sender, KeyEventArgs e)

{

switch (e.KeyCode.ToString())

{

case "D":

\_dirX = 1;

\_dirY = 0;

break;

case "A":

\_dirX = -1;

\_dirY = 0;

break;

case "W":

\_dirY = -1;

\_dirX = 0;

break;

case "S":

\_dirY = 1;

\_dirX = 0;

break;

}

}

//Метод отвечающий за прорисовку карты, змейки и фрукта в компоненте

protected override void OnPaint(PaintEventArgs e)

{

Brush b = new SolidBrush(BackColor);

e.Graphics.FillRectangle(b, ClientRectangle);

b = new SolidBrush(MapColor);

\_fruit.BackColor = FructColor;

if (\_snakeStatus)

{

\_snake[0].BackColor = HeadColor;

\_snake[1].BackColor = TailColor;

\_currentHeadColor = HeadColor;

\_currentTailColor = TailColor;

}

//Прорисовка карты

Pen p = new Pen(MapColor, 1);

for (int i = 0; i <= \_mapSize / \_sizeOfSides; i++)

{

e.Graphics.DrawLine(p, 0, \_sizeOfSides \* i, \_mapSize - 1, \_sizeOfSides \* i);

e.Graphics.DrawLine(p, \_sizeOfSides \* i, 0, \_sizeOfSides \* i, \_mapSize - 1);

}

//Вывод сообщение о результате игры

if (\_gameStatus)

{

string s = (\_score != 224) ? "Game Over" : "Victory";

int fontSize = (int)(Math.Round((\_mapSize - 6) \* 10 / e.Graphics.DpiY));

Font font = new Font("Arial", fontSize);

SolidBrush fontB = (\_score != 224) ? new SolidBrush(Color.Red) : new SolidBrush(Color.Green);

StringFormat sf = new StringFormat

{

Alignment = StringAlignment.Center,

LineAlignment = StringAlignment.Center

};

RectangleF rect = new RectangleF(0, 0, \_mapSize, \_mapSize);

e.Graphics.DrawString(s, font, fontB, rect, sf);

}

b.Dispose();

}

protected override CreateParams CreateParams

{

get

{

CreateParams cp = base.CreateParams;

cp.ExStyle |= 0x02000000;

return cp;

}

}

}

}