|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ДЕПАРТАМЕНТ ОБРАЗОВАНИЯ ИВАНОВСКОЙ ОБЛАСТИ  Областное государственное бюджетное профессиональное образовательное учреждение  «Ивановский промышленно-экономический колледж» | | | | | | |
|  | | | | | | |
| **КУРСОВОЙ ПРОЕКТ** | | | | | | |
|  | | | | | | |
| **Разработка виртуального псевдо-ретро компьютера** | | | | | | |
| **ИВПЭК. 09.02.07.** | | | | | | |
|  | | | | | | |
|  | | **Специальность:** | | | **09.02.07 Информационные системы и программирование, базовая подготовка** | |
|  | | | | | | |
|  | | | | | | |
| Руководитель курсового проекта | | |  | | | В.И. Кукушкин |
| Выполнил обучающийся группы 407а | | |  | | | Д.Д. Груздев |
|  | | |  | | |  |
|  | | | | | | |
|  | | | | | | |
| Курсовой проект выполнен и защищен с оценкой\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | | | | | | |
|  | | | | | | |
| «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2024 г. | | | | | | |
|  |  | | | | | |
|  | | | |  | |  |
|  | | | |  | |  |
|  | Иваново 2024 | | | | |  |

Содержание

[Введение 3](#_Toc191733515)

[1 Концептуальное проектирование 5](#_Toc191733516)

[1.1 Теоретический вопрос 5](#_Toc191733517)

[2. Техно-рабочий проект 12](#_Toc191733518)

[2.1 Моделирование с использованием UML-диаграмм 12](#_Toc191733519)

[3.Рабочая документация 17](#_Toc191733520)

[3.1 Описание разработки приложения 17](#_Toc191733521)

[3.2 Рабочая докуметация. 26](#_Toc191733522)

[Заключение 27](#_Toc191733523)

[Список используемых источников 28](#_Toc191733524)

[Приложение А 30](#_Toc191733525)

Введение

Программирование является одной из ключевых дисциплин в современном мире информационных технологий. Оно играет важную роль в различных областях, начиная от разработки программного обеспечения и веб-приложений и заканчивая созданием игр и решением задач искусственного интеллекта.

В последние годы наблюдается возрождение интереса к ретро-технологиям, включая старые компьютеры и операционные системы. Разработка виртуального псевдо-ретро компьютера предоставляет уникальную возможность погрузиться в атмосферу прошлых десятилетий, изучить основы работы с устаревшими системами и понять, как развивались технологии. Такой проект может быть не только увлекательным, но и образовательным, позволяя пользователям развивать аналитические способности, навыки программирования и понимание компьютерной архитектуры.

Целью данного проекта является создание виртуального псевдо-ретро компьютера, который будет эмулировать работу классических систем, таких как Commodore 64 или ZX Spectrum. Виртуальный компьютер будет разработан на языке программирования Rust, что обеспечит высокую производительность и безопасность, а также позволит использовать современные подходы к разработке программного обеспечения.

Приложение будет включать основные функциональные возможности для изучения работы с ретро-компьютерами, такие как:

* эмуляция работы классических операционных систем и программ;
* возможность написания и запуска простых программ на языках программирования, популярных в то время, таких как BASIC;
* интерактивные уроки и задания для закрепления полученных знаний о компьютерной архитектуре и программировании.

Актуальность данной работы заключается в том, что разработка мобильных приложений для образовательных целей набирает популярность, и изучение ретро-технологий, являясь интересным и полезным навыком, может стать предметом увлекательного и познавательного обучения. Виртуальный псевдо-ретро компьютер не только позволит пользователям окунуться в атмосферу прошлого, но и поможет развить навыки, которые будут полезны в современном мире информационных технологий.

# 1 Концептуальное проектирование

1.1 Теоретический вопрос

Концептуальное проектирование играет ключевую роль в разработке программного обеспечения. Этот этап представляет собой процесс определения общих принципов, концепций и структуры проекта перед его реализацией. В его основе лежит формулировка общего видения будущего программного продукта. На этом этапе определяются общие принципы и концепции программы, такие как механики, архитектура программы и пользовательский интерфейс. Для моего проекта концептуальное проектирование устанавливает следующие задачи:

**Разработка сценариев использования**: на этом этапе определяются основные сценарии взаимодействия пользователя с виртуальным псевдо-ретро компьютером, включая сценарии запуска эмуляции, написания и выполнения программ, а также настройки параметров системы. **Определение интерфейса пользователя**: концептуальное проектирование включает разработку пользовательского интерфейса, который будет обеспечивать комфортное взаимодействие пользователя с эмулятором, включая отображение информации о состоянии системы, редакторе кода и меню действий. **Определение требований к аудиовизуальному контенту**: определение требований к аудиовизуальным элементам, таким как звуковые сигналы и визуальные подсказки, которые помогут пользователям лучше понять работу виртуального компьютера и его программного обеспечения. Концептуальное проектирование способствует созданию эффективных и устойчивых программных решений, позволяя разработчикам четко представить цели и основные принципы проекта перед началом его реализации.

1.1.2 Анализ предметной области и постановка задач

Виртуальный псевдо-ретро компьютер — это эмуляция классических систем, таких как Commodore 64 или ZX Spectrum, которая позволяет пользователям погрузиться в атмосферу прошлых десятилетий и изучить основы работы с устаревшими системами. Такие системы были разработаны для обеспечения простоты и надежности работы с программами, что делает их идеальными для образовательных целей.

Изучение работы с ретро-компьютерами требует понимания основ компьютерной архитектуры и программирования, что может быть непростой задачей для новичков. Традиционные методы обучения, такие как книги и статьи, часто оказываются неэффективными, так как они не обеспечивают должного уровня интерактивности и возможности практического применения. В то же время современные технологии позволяют создать мобильное приложение, которое поддерживает интерактивное обучение и предоставляет пользователю гибкие инструменты для тренировки.

Формулировка задач проекта:

**Обучение основам работы с ретро-компьютерами**: приложение будет использовать аудиовизуальные средства, чтобы помочь пользователю быстрее освоить принципы работы с эмулируемой системой и программами.

**Предоставление интерактивных упражнений для закрепления знаний**: приложение включает упражнения, такие как написание и выполнение простых программ на языках, популярных в то время, что позволяет пользователю самостоятельно оценить прогресс и улучшить свои навыки.

**Повышение доступности ретро-технологий для широкой аудитории**: разработка мобильного приложения позволит сделать обучение работе с ретро-компьютерами доступным для широкой аудитории за счет использования технологий, которые уже находятся в повседневном пользовании у большинства людей.

**Тестирование и отладка**: проведение тестирования программы на различных этапах разработки для выявления и устранения ошибок и недочетов.

Таким образом, данное приложение не только решает проблему удобного и интерактивного изучения работы с ретро-компьютерами, но и устраняет основные недостатки существующих методов обучения. Оно позволяет использовать мобильное устройство для эффективного изучения и закрепления материала, предлагая интуитивный интерфейс и персонализированный подход к процессу обучения.

1.1.3 Анализ аналогов и существующих решений

На рынке существует ряд эмуляторов и приложений, предлагающих пользователям уникальный опыт работы с ретро-компьютерами. Некоторые из них стоит упомянуть:

**VICE (Windows, macOS, Linux)**: Эмулятор Commodore 64, который предлагает широкий выбор функций, включая поддержку различных форматов файлов и возможность настройки параметров эмуляции. Однако интерфейс может быть сложен для новичков, и отсутствуют обучающие модули.

**ZSNES (Windows, Linux)**: Эмулятор Super Nintendo, который позволяет пользователям играть в классические игры. Однако он не предлагает образовательных материалов или интерактивных упражнений, что ограничивает его использование для обучения.

Мой проект имеет ряд инноваций и особенностей, которые могут выделить его среди аналогов:

**Интерактивные образовательные модули**: в отличие от многих аналогов, наше приложение будет включать обучающие м одули, которые охватывают базовые теоретические знания и практические советы по работе с ретро-компьютерами. Это поможет пользователям изучить материал более систематично и не "потеряться" на начальных этапах.

**Аудиовизуальная поддержка и обратная связь**: в приложении будут использованы как звуковые, так и визуальные элементы, что ускорит восприятие и запоминание принципов работы с эмулируемой системой. Применение обратной связи (например, подсказок и коррекций при ошибках) поможет пользователю быстрее освоить материал.

**Система тестирования и проверки навыков**: в дополнение к учебным модулям будет включена система тестирования, которая позволит пользователям проверять свои знания и навыки работы с ретро-компьютерами в режиме реального времени. Тесты будут нацелены на разные аспекты (восприятие, запоминание, программирование), что обеспечит всестороннюю проверку знаний.

В целом, мой проект имеет потенциал привлечь внимание пользователей своими инновациями, увлекательным учебным процессом и качественным исполнением, что делает его более эффективным и привлекательным для целевой аудитории.

1.2Техническое задание

1.2.1 Функциональные требования

Идентификация функциональности

1. Система должна предоставлять возможность создания виртуального псевдо-ретро компьютера.
2. Система должна обеспечивать возможность загрузки и запуска программ, совместимых с псевдо-ретро архитектурой.
3. Система должна предоставлять интерфейс для настройки параметров виртуального компьютера (например, объем оперативной памяти, тип процессора).
4. Система должна поддерживать сохранение и загрузку состояния виртуального компьютера.
5. Система должна обеспечивать возможность взаимодействия с файловой системой хоста для загрузки и сохранения файлов.

Описание функциональности

1. **Создание виртуального компьютера**: Пользователь должен иметь возможность выбрать параметры виртуального компьютера (например, тип процессора, объем оперативной памяти) и создать его. Система должна инициализировать виртуальную машину с заданными параметрами.
2. **Загрузка и запуск программ**: После создания виртуального компьютера пользователь должен иметь возможность загружать и запускать программы, которые были разработаны для работы на псевдо-ретро архитектуре. Система должна обеспечивать корректное выполнение программ и отображение результатов на экране.
3. **Настройка параметров**: Пользователь должен иметь возможность изменять параметры виртуального компьютера через графический интерфейс. Система должна обновлять настройки и перезапускать виртуальную машину при необходимости.
4. **Сохранение и загрузка состояния**: Пользователь должен иметь возможность сохранять текущее состояние виртуального компьютера и загружать его позже. Система должна сохранять все данные, включая состояние оперативной памяти и содержимое дисков.
5. **Взаимодействие с файловой системой**: Система должна предоставлять интерфейс для доступа к файловой системе хоста, позволяя пользователю загружать файлы в виртуальный компьютер и сохранять результаты работы программ.

Взаимодействие функций:

* после создания виртуального компьютера пользователь может сразу же перейти к загрузке программ, что подразумевает, что функция создания компьютера должна завершаться успешным переходом к функции загрузки программ;
* настройки виртуального компьютера могут быть изменены в любой момент, но для применения изменений может потребоваться перезапуск виртуальной машины;
* сохранение состояния виртуального компьютера должно быть доступно в любой момент, но для загрузки состояния необходимо, чтобы виртуальный компьютер был остановлен.

Ограничения функциональности:

* создание и настройка виртуального компьютера доступны только авторизованным пользователям;
* загрузка и запуск программ ограничены только теми, которые совместимы с псевдо-ретро архитектурой;
* сохранение состояния виртуального компьютера может быть ограничено по объему в зависимости от доступного пространства на хосте.

1.2.2 Нефункциональные требования

1. **Производительность**: Виртуальный компьютер должен обеспечивать приемлемую скорость выполнения программ, не превышающую 2 секунд на запуск и 5 секунд на выполнение стандартных задач.
2. Безопасность: Доступ к функциональности системы должен быть защищен паролем, а данные пользователя должны быть зашифрованы.
3. **Масштабируемость**: Система должна поддерживать возможность добавления новых функций и расширения функциональности без значительных изменений в архитектуре.
4. **Совместимость**: Система должна работать на основных операционных системах (Windows, macOS, Linux) без необходимости установки дополнительных библиотек.

Ограничения и предположения:

* предполагается, чтыыо пользователи имеют базовые навыки работы с компьютером и могут взаимодействовать с графическим интерфейсом;
* ограничение на использование только совместимых программ может повлиять на выбор программного обеспечения для загрузки;
* предполагается, что у пользователей достаточно ресурсов на хосте для запуска виртуального компьютера.

1.2.3 Взаимодействие с концептуальным проектированием

Функциональные требования, сформулированные в данном техническом задании, соответствуют концептуальному проектированию, которое акцентировало внимание на создании интуитивно понятного интерфейса и обеспечении высокой производительности. Основные функции, такие как создание виртуального компьютера и загрузка программ, отражают ключевые аспекты концепции, направленной на удобство использования и доступность. Взаимодействие между функциями также соответствует принципам модульности и интеграции, что позволяет пользователю легко переходить от одной функции к другой.

# 2. Техно-рабочий проект

2.1 Моделирование с использованием UML-диаграмм

UML (Unified Modeling Language) — это стандартизированный язык визуального моделирования, используемый для проектирования, визуализации и документирования программных систем. UML предоставляет набор графических нотаций, которые помогают разработчикам и проектировщикам описывать структуру, поведение и взаимодействие компонентов системы.

Цели использования UML в проекте разработки псевдо-ретро компьютера

**Визуализация архитектуры системы**: UML-диаграммы помогут наглядно представить структуру системы, включая её основные компоненты и их взаимодействие.

**Документирование**: UML позволяет создать подробную документацию, которая будет полезна для понимания системы как на этапе разработки, так и в будущем.

**Коммуникация между разработчиками**: UML-диаграммы служат универсальным языком для обмена идеями и концепциями между членами команды.

**Анализ и проектирование**: UML помогает выявить возможные проблемы на ранних этапах проектирования и спроектировать систему с учетом всех требований.

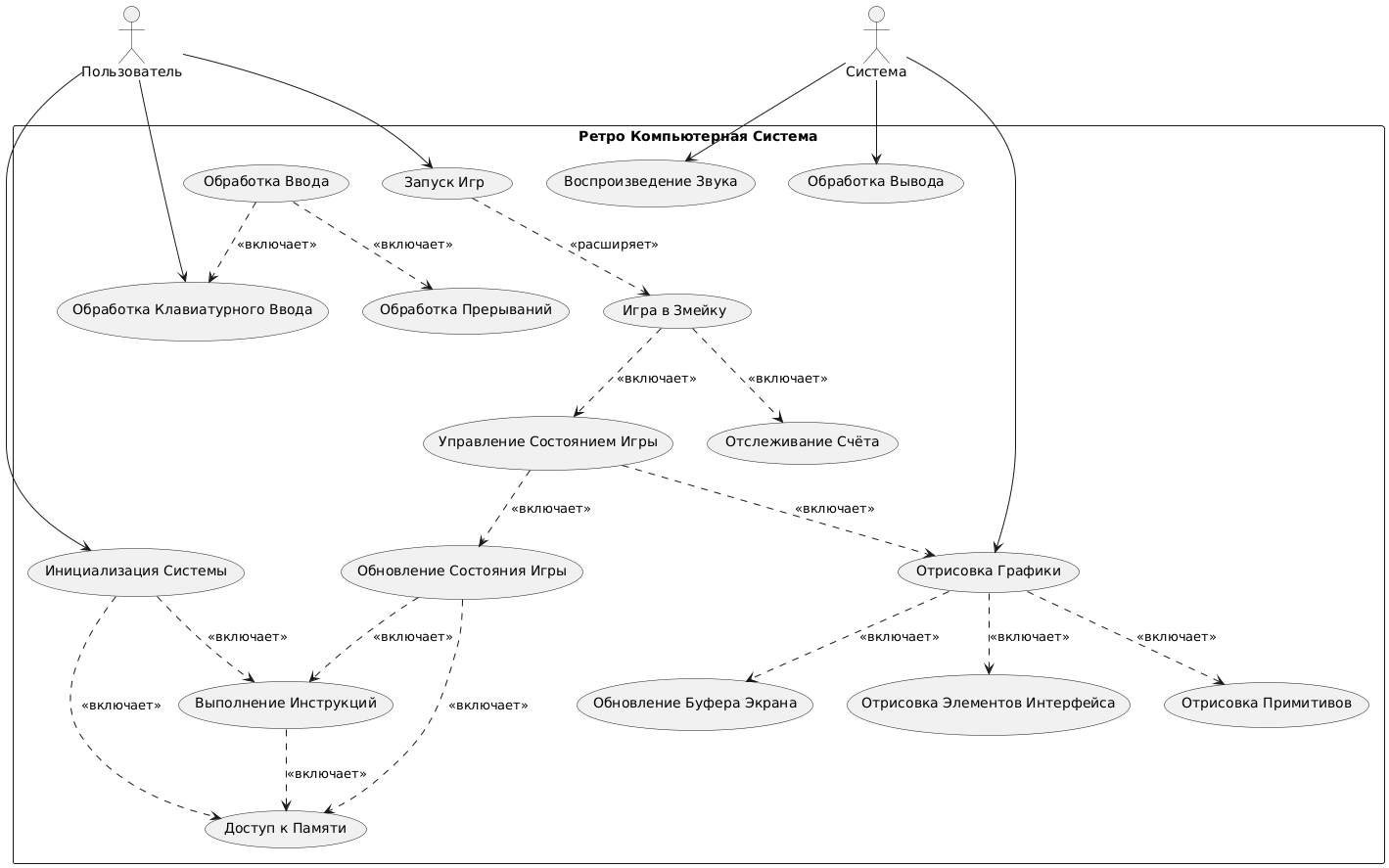


Рисунок 1- UML диаграмма прецедентов

1. **Обработка Ввода**

**Описание**: Этот компонент отвечает за обработку пользовательского ввода, такого как нажатия клавиш или использование джойстика.

**Значение**: В ретро-компьютерах ввод часто ограничивается простыми устройствами, такими как клавиатура или джойстик. Реализация этого компонента требует создания драйверов для обработки сигналов от этих устройств.

1. **Запуск Игр**

**Описание**: Отвечает за запуск игровых приложений.

**Значение**: В ретро-компьютерах игры часто запускаются с кассет или дискет. Этот компонент должен обеспечивать загрузку и запуск игр из соответствующих носителей.

1. **Воспроизведение Звука**

**Описание**: Включает воспроизведение звуковых эффектов и музыки.

**Значение**: Звук в ретро-компьютерах обычно простой, но важный для атмосферы. Этот компонент может включать генерацию звуковых сигналов через встроенный динамик или звуковую карту.

1. **Обработка Вывода**

**Описание**: Отвечает за вывод информации на экран или другие устройства вывода.

**Значение**: В ретро-компьютерах вывод часто осуществляется на CRT-мониторы. Этот компонент должен управлять отрисовкой изображения на экране.

1. **Управление Состоянием Игры**

**Описание**: Управляет текущим состоянием игры, например, паузой, завершением или переходом между уровнями.

**Значение**: Этот компонент важен для обеспечения плавного игрового процесса и управления логикой игры.

1. **Отслеживание Счёта**

**Описание**: Отслеживает и обновляет счет игрока.

**Значение**: В ретро-играх счет часто является ключевым элементом. Этот компонент должен обеспечивать корректное отображение и обновление счета.

1. **Инициализация Системы**

**Описание**: Отвечает за начальную загрузку и настройку системы.

**Значение**: Этот компонент важен для подготовки системы к работе, включая загрузку операционной системы и инициализацию периферийных устройств.

1. **Обновление Состояния Игры**

**Описание**: Обновляет состояние игры на основе ввода пользователя и других факторов.

**Значение**: Этот компонент обеспечивает динамику игры, обновляя позиции объектов, проверяя коллизии и т.д.

1. **Отрисовка Графики**

**Описание**: Отвечает за отрисовку графики на экране.

**Значение**: В ретро-компьютерах графика часто пиксельная и ограниченная по цветам. Этот компонент должен эффективно управлять видеопамятью и отрисовывать изображение.

1. **Отрисовка Элементов Интерфейса**

**Описание**: Отвечает за отрисовку элементов пользовательского интерфейса, таких как меню и кнопки.

**Значение**: Интерфейс должен быть простым и интуитивно понятным, что характерно для ретро-систем.

1. **Выполнение Инструкций**

**Описание**: Выполняет машинные инструкции и команды.

**Значение**: Этот компонент является ядром системы, отвечающим за выполнение программного кода.

1. **Обновление Буфера Экрана**

**Описание**: Обновляет буфер экрана, который затем отображается на мониторе.

**Значение**: Буфер экрана используется для временного хранения графических данных перед их отображением.

1. **Отрисовка Элементов Примитивов**

**Описание**: Отвечает за отрисовку простых графических примитивов, таких как линии и круги.

**Значение**: Эти примитивы используются для создания более сложных графических элементов.

# 3.Рабочая документация

## 3.1 Описание разработки приложения

Начнем с того что, создаем новую папку memory. В этой папке мы создаем три файла это RAM, ROM. В файле RAM создаем структуру, которая представляет собой модель оперативной памяти. Эта структура содержит поле memory, которое является вектором (динамическим массивом) байтов (u8). Вектор инициализируется с размером 65536 элементов, что соответствует 64 КБ памяти.В реализации структуры RAM мы создаем метод new, который отвечает за создание нового экземпляра RAM. Этот метод инициализирует память нулями, что означает, что при создании новой RAM все ячейки памяти будут иметь значение 0. Метод read позволяет читать данные из памяти по указанному адресу. Он принимает параметр address типа u16 (16-битное беззнаковое целое число), который указывает на ячейку памяти, из которой нужно прочитать данные. Метод возвращает значение типа u8 (8-битное беззнаковое целое число), которое находится по указанному адресу. Метод write позволяет записывать данные в память по указанному адресу. Он принимает два параметра: address типа u16, который указывает на ячейку памяти, куда нужно записать данные, и value типа u8, которое представляет собой значение, которое нужно записать. Метод изменяет состояние памяти, записывая значение в указанную ячейку. Дальше Структура ROM ROMпредставляет собой модель постоянной памяти, которая может только читать данные, но не позволяет их изменять. Она содержит поле memory, которое является вектором байтов (Vec<u8>). Этот вектор хранит данные, которые были загружены в ROM при его создании. Метод new отвечает за создание нового экземпляра ROM. Он принимает параметр data типа Vec<u8>, который представляет собой данные, которые будут храниться в ROM. Эти данные передаются в структуру при её создании и сохраняются в поле memory.Метод read позволяет читать данные из ROM по указанному адресу. Он принимает параметр address типа u16 (16-битное беззнаковое целое число), который указывает на ячейку памяти, из которой нужно прочитать данные. Метод возвращает значение типа u8 (8-битное беззнаковое целое число), которое находится по указанному адресу.

Основные отличия ROM от RAM

**Изменяемость**:

В RAM данные можно как читать, так и записывать (с помощью метода write).

В ROM данные можно только читать. Запись данных в ROM невозможна, так как это противоречит самой идее постоянной памяти.

**Инициализация**:

В RAM память инициализируется нулями при создании.

В ROM память инициализируется переданными данными (data: Vec<u8>), которые остаются неизменными на протяжении всего времени жизни структуры.

После этого создаем следующую папку Graphics, в этой папке мы создаем файлы

Animation, Primitives, Pseude retro ui, video adapter. Структура Animation представляет собой модель анимации, которая состоит из последовательности кадров. Каждый кадр — это двумерный массив пикселей, представленный в виде одномерного вектора (Vec<u32>), где каждый элемент вектора соответствует цвету пикселя.

**Поля структуры**:

* frames :Вектор кадров анимации. Каждый кадр — это вектор пикселей;
* current\_frame :Индекс текущего кадра, который отображается;
* frame\_duration :Длительность отображения одного кадра;
* last\_update :Время последнего обновления кадра;
* widt и height :Ширина и высота кадра.

Метод new отвечает за создание нового экземпляра Animation. Он принимает три параметра:

* width: Ширина кадра;
* height: Высота кадра;
* frame\_duration: Длительность отображения одного кадра.

Метод add\_frame позволяет добавлять новый кадр в анимацию. Он принимает параметр frame типа Vec<u32>, который представляет собой одномерный вектор пикселей. Перед добавлением кадра выполняется проверка на соответствие размера кадра заданным ширине и высоте. Метод update отвечает за обновление текущего кадра анимации. Он проверяет, прошло ли достаточно времени для смены кадра, и если да, переходит к следующему кадру. Метод get\_current\_frame возвращает текущий кадр анимации. Это срез (&[u32]) вектора пикселей текущего кадра.

Структура Primitives не содержит полей, так как она используется как набор статических методов для рисования. Это утилитарный класс, который предоставляет функции для работы с графикой. Метод draw\_line этот метод рисует линию на буфере пикселей, используя алгоритм Брезенхэма. Он принимает следующие параметры:

* buffer: Буфер пикселей, представленный как одномерный массив u32;
* width: Ширина буфера (количество пикселей в строке);
* x0, y0: Координаты начальной точки линии;
* x1, y1: Координаты конечной точки линии;
* color: Цвет линии в формате u32.

Метод draw\_rect этот метод рисует прямоугольник на буфере пикселей. Он принимает следующие параметры:

* buffer: Буфер пикселей;
* width: Ширина буфера;
* x,y: Координаты верхнего левого угла прямоугольника;
* w, h: Ширина и высота прямоугольника;
* color: Цвет прямоугольника.

Метод draw\_circle этот метод рисует окружность на буфере пикселей, используя алгоритм Брезенхэма для окружностей. Он принимает следующие параметры:

* buffer: Буфер пикселей;
* width: Ширина буфера;
* x0, y0: Координаты центра окружности;
* radius: Радиус окружности;
* color: Цвет окружности.

Вспомогательный метод plot\_pixel этот метод устанавливает пиксель в буфере, если его координаты находятся в пределах буфера. Метод проверяет, находятся ли координаты пикселя в пределах буфера, прежде чем установить его. Структура PseudoRetroUI, которая представляет собой основной интерфейс для работы с псевдо-ретро компьютером. Функция run в PseudoRetroUI — это основной цикл приложения, который продолжает работать, пока окно открыто и не нажата клавиша Escape. В Rust основной цикл обновляет состояние приложения, обрабатывает ввод, рендерит графику и управляет временем кадров. Функция handle\_input в Rust отвечает за обработку пользовательского ввода. В Rust вы можете добавить логику для обработки нажатий клавиш или других событий ввода. Функция update в Rust отвечает за обновление состояния приложения. В Rust вы обновляете состояние процессора (CPU), выполняете инструкции и обновляете буферы. Функция render в Rust отвечает за отрисовку графики. В Rust вы используете два буфера (front и back) для рендеринга. Если буфер изменился, он обновляется на экране. В Rust вы используете два буфера (front и back) для рендеринга. В Rust вы проверяете, изменился ли буфер, и если да, то обновляете его на экране. В Rust вы также управляете аудио и анимацией через структуры AudioDevice и Animation. Структура VideoAdapter, которая отвечает за управление графическим интерфейсом и отрисовку элементов на экране. Этот класс использует библиотеку minifb для создания окна и управления его содержимым.

Основные компоненты:

Константы:

VRAM\_START, VRAM\_SIZE, SCREEN\_WIDTH, SCREEN\_HEIGHT — определяют начало видеопамяти, её размер и размеры экрана.

FONT\_WIDTH, FONT\_HEIGHT— размеры шрифта.

FONT\_CHARS— массив, содержащий битовые маски для отрисовки символов (цифры, буквы и некоторые специальные символы).

UI\_BORDER\_COLOR, UI\_BACKGROUND\_COLOR, UI\_TEXT\_COLOR, UI\_HIGHLIGHT\_COLOR, UI\_PADDING — константы для оформления интерфейса.

Структуры:

Container — представляет собой прямоугольную область на экране с координатами, размерами и возможностью отрисовки границы.

Button — кнопка, которая содержит контейнер, текст и состояние (выбрана или нет).

ListItem — элемент списка, содержащий текст и значение.

List — список элементов, который содержит контейнер, массив элементов и индекс выбранного элемента.

VideoAdapter:

* window — окно, созданное с помощью minifb;
* vram — видеопамять, представленная массивом пикселей;
* dirty — флаг, указывающий, нужно ли обновлять экран.

Основные функции:

Инициализация:

VideoAdapter::new(title: &str) — создает новое окно с заданным заголовком и инициализирует видеопамять.

Работа с видеопамятью:

* write\_vram(addr: u16, value: u32) — записывает значение в видеопамять по указанному адресу;
* read\_vram(addr: u16) — читает значение из видеопамяти по указанному адресу;
* clear\_screen(color: u32) — заполняет весь экран указанным цветом;
* draw\_pixel(x: usize, y: usize, color: u32) — рисует пиксель на экране по указанным координатам.

Отрисовка элементов интерфейса:

* draw\_text(text: &str, x: usize, y: usize, color: u32) — отрисовывает текст на экране, используя битовые маски из FONT\_CHARS;
* draw\_char(char\_data: u64, x: usize, y: usize, color: u32) — отрисовывает символ по его битовой маске;
* draw\_rectangle(x: usize, y: usize, width: usize, height: usize, color: u32) — рисует прямоугольник на экране;
* draw\_border(x: usize, y: usize, width: usize, height: usize, thickness: usize, color: u32) — рисует границу вокруг прямоугольника;
* draw\_container(container: &Container) — отрисовывает контейнер с фоном и границей (если она есть);
* draw\_button(button: &Button) — отрисовывает кнопку с текстом, фоном и границей. Если кнопка выбрана, её фон меняется на выделенный цвет;
* draw\_list(list: &List) — отрисовывает список элементов с возможностью выбора одного из них. После этого я создаю папку GPU и в этой папки мы создаем файлы EXECUTION, INSTRUCTION, REGISTER. В коде execution создается структура CPU, которая представляет собой центральный процессор. Она содержит регистры (Registers) и аудиоустройство (AudioDevice). Функция execute отвечает за выполнение инструкций, которые передаются в процессор.

Функция execute принимает инструкцию (Instruction), буфер для графики (buffer), ширину экрана (width) и оперативную память (RAM). В зависимости от типа инструкции, процессор выполняет различные действия. Каждая инструкция вызывает определенное действие. Например:

* instruction::LDA(value) загружает значение в аккумулятор;
* instruction::ADD(value) добавляет значение к аккумулятору;
* instruction::DrawLine(x0, y0, x1, y1, color) рисует линию на экране;
* instruction::PlayTone(frequency, duration) воспроизводит звук;

Работа с памятью (RAM) позволяет процессору читать и записывать данные. реализованы инструкции для работы с графикой и звуком:

* drawLine, drawRect, drawCircle — рисуют примитивы на экране;
* playTone — воспроизводит звук;

реализованы инструкции для работы со стеком и вызовами функций:

* push, pop — работают со стеком;
* call, ret — позволяют вызывать функции и возвращаться из них.

Instruction моделирует набор инструкций для процессора. Каждая инструкция может выполнять определенные действия, такие как загрузка значения в аккумулятор, сохранение значения в памяти, выполнение арифметических операций, управление потоком выполнения программы и даже отрисовка графических элементов. Instruction — это перечисление, которое представляет собой набор инструкций для процессора. Каждая инструкция может иметь свои параметры, такие как числовые значения, адреса памяти или строки. Каждая инструкция, такая как LDA, STA, ADD, и т.д., выполняет определенную операцию. Например, LDA(u8) загружает значение в аккумулятор, а DrawLine(u16, u16, u16, u16, u32) рисует линию на экране. Метод decode, который принимает опкод (код операции) и операнд (данные для операции) и возвращает соответствующую инструкцию. Например, если опкод равен 0x01, то будет создана инструкция LDA с переданным операндом. Если опкод неизвестен, то по умолчанию возвращается инструкция NOP (нет операции)

Начнем с того, что создаем структуру Registers, которая представляет собой набор регистров, используемых в эмуляторе или виртуальной машине. Эта структура содержит четыре поля:

* accumulator: u8 — аккумулятор, который используется для хранения промежуточных результатов арифметических и логических операций. Это 8-битный регистр;
* index: u8 — индексный регистр, который часто используется для хранения индексов или адресов в памяти. Это также 8-битный регистр;
* stack\_pointer: u8 — указатель стека, который указывает на текущую вершину стека. В данном случае он инициализируется значением 0xFF, что может означать, что стек начинается с конца памяти и растет вниз;
* program\_counter: u16 — счетчик команд, который хранит адрес следующей команды, которую нужно выполнить. Это 16-битный регистр, что позволяет адресовать больше памяти;

Метод new() используется для создания нового экземпляра структуры Registers. Он инициализирует все регистры начальными значениями:

* accumulator и index устанавливаются в 0, что означает, что они начинают с нулевого значения;
* stack\_pointer устанавливается в 0xFF, что может указывать на начало стека в памяти;
* program\_counter устанавливается в 0, что означает, что выполнение программы начнется с первой команды.

Метод reset() позволяет сбросить все регистры к их начальным значениям. Это полезно, например, при перезапуске эмулятора или виртуальной машины. Этот метод изменяет текущий экземпляр структуры, сбрасывая все поля к значениям, аналогичным тем, которые устанавливаются в методе new(). После того как разработал виртуальный псевдо-ретро компьютер создаем для него любое приложение мой выбор - это игра змейка.

**Создание структуры** nakeGame: в коде создается структура SnakeGame, которая является основным классом игры. Она объединяет все компоненты игры, такие как процессор (CPU), оперативная память (RAM), видеокарта (VideoAdapter), и управляет состоянием игры (GameState).

Внутри структуры SnakeGame инициализируются начальные значения для змейки, еды, направления движения и состояния игры. Это похоже на то, как в примере создается главное меню и настраиваются его параметры.

Функция run: Эта функция является основным игровым циклом. Она проверяет, открыто ли окно игры, и в зависимости от текущего состояния игры (GameState) вызывает соответствующие методы для обработки ввода, обновления состояния игры и отрисовки.

Обработка ввода (handle\_input): В этой функции обрабатываются нажатия клавиш в зависимости от текущего состояния игры. Например, в состоянии меню (GameState:Menu) обрабатываются нажатия клавиш "Вверх" и "Вниз" для выбора пунктов меню, а также "Enter" для подтверждения выбора.

Обновление состояния игры (update):

В этой функции обновляется положение змейки, проверяется столкновение с едой или стенками, а также обновляется состояние игры (например, переход в состояние "Game Over" при столкновении).

Отрисовка игры (render): В зависимости от текущего состояния игры (GameState), вызываются соответствующие функции для отрисовки меню, игрового поля, паузы или экрана завершения игры.

Отрисовка меню (render\_menu): В этой функции создается контейнер для меню, отрисовывается заголовок и кнопки. Кнопки "Start Game" и "Exit" создаются с помощью структуры Button, которая принимает текст и состояние выбора.

Отрисовка игрового поля (render\_game): В этой функции отрисовывается игровое поле, включая границы, змейку и еду. Змейка отрисовывается с использованием данных из оперативной памяти, где хранятся координаты сегментов змейки.

Отрисовка экрана паузы и завершения игры: В этих функциях отрисовываются соответствующие экраны с текстом и кнопками. Например, на экране паузы отображается кнопка "Resume", а на экране завершения игры — кнопка "Restart".

Генерация еды (spawn\_food): В этой функции случайным образом генерируются координаты для еды, которые затем записываются в оперативную память.

Сброс игры (reset\_game): в этой функции сбрасываются все параметры игры (положение змейки, еды, направление и состояние игры) к начальным значениям.

3.2 Руководство пользователя.

Главное меню (Рисунок - 2):

1. **Запуск игры**:

При запуске игры вы увидите главное меню с двумя кнопками:

* **"Exit"** (Выход);
* **"Start game"** (Начать игру);
* Для навигации по меню используйте **стрелки вверх и вниз**;
* Чтобы выбрать кнопку, нажмите **пробел**.

1. **Действия в меню**:

* Если вы выбрали **"Exit"**, игра закроется;
* Если вы выбрали **"Start game"**, начнется игра "Змейка".

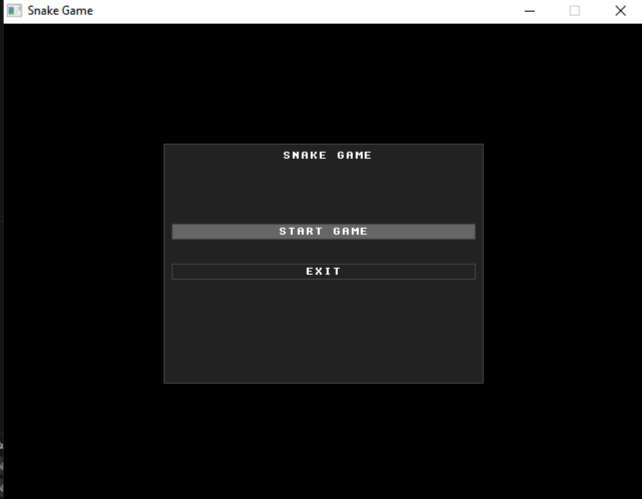


Рисунок 2 - Меню игры

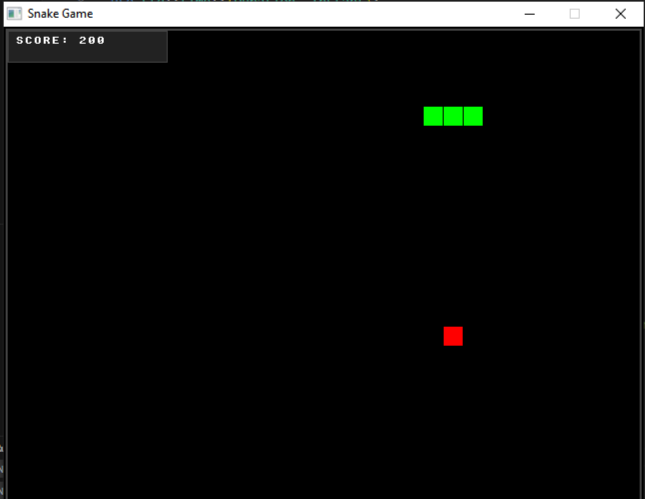


Рисунок 3 - Игра змейка

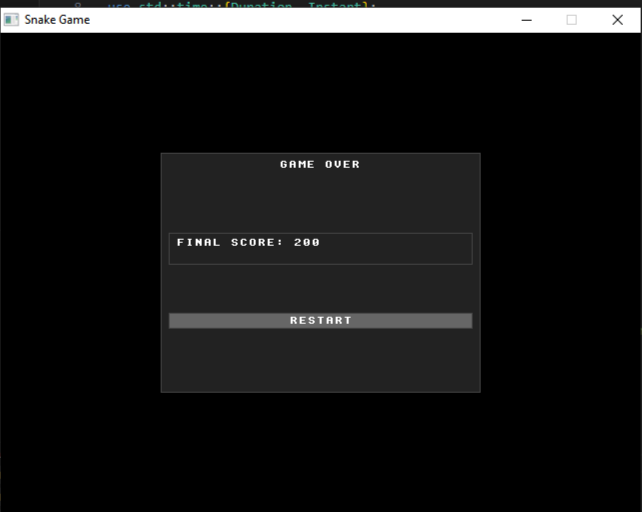


Рисунок 4 - Рестарт игры змейка

Управление игрыы "Змейка"(Рисунок - 3):

1. **Основное управление**:

* Для управления змейкой используйте **стрелки вверх, вниз, влево и вправо**.
* Змейка будет двигаться в выбранном направлении.

1. **Цель игры**:

* Собирайте **яблоки**, которые появляются на игровом поле.
* Каждое съеденное яблоко увеличивает длину змейки и добавляет очки к вашему счету.

1. **Завершение игры**:

Игра завершится, если змейка:

* Коснется **границ игрового поля**.
* Коснется **своего хвоста**.

После завершения игры вы увидите экран с вашим **финальным счетом** и кнопкой **"Рестарт"(Рисунок - 4)**.

1. **Рестарт игры**:

* Чтобы начать игру заново, выберите кнопку **"Рестарт"** с помощью стрелок и нажмите **пробел**.

1. Счет:

* Ваш счет отображается на экране во время игры.
* За каждое съеденное яблоко вы получаете **+1 очко**.

1. Увеличение длины змейки:

* С каждым съеденным яблоком длина змейки увеличивается, что усложняет игру.

Заключение

В рамках данного курсового проекта был разработан псевдо-ретро компьютер, имитирующий архитектуру и функциональность классических вычислительных устройств 80-х годов. Целью проекта было создание программно-аппаратной платформы, которая воспроизводит особенности работы ретро-компьютеров, включая их процессоры, память, графику и звук. В процессе разработки были выполнены следующие задачи:

* была разработана концепция псевдо-ретро компьютера, включающая функциональные и нефункциональные требования. Архитектура проекта включает в себя эмуляцию процессора, оперативной памяти, видеокарты и звуковой карты, что позволяет воспроизводить работу классических компьютеров;
* определены основные компоненты системы: процессор с набором инструкций, память для хранения данных и программ, видеокарта для отображения графики и звуковая карта для генерации звуковых эффектов;
* реализована оперативная память, которая хранит данные и программы. Память разделена на сегменты для хранения кода, данных и стека;
* создана видеокарта, которая отображает графику в текстовом режиме и поддерживает простые графические примитивы, такие как линии и прямоугольники.

В результате проделанной работы был создан псевдо-ретро компьютер, который позволяет пользователю погрузиться в атмосферу классических вычислительных устройств. Проект демонстрирует возможности современных технологий для эмуляции ретро-систем и предоставляет платформу для изучения низкоуровневого программирования.

Список используемых источников

1. Официальная документация Rust [Электронный ресурс] - <https://doc.rust-lang.org/book/>. 2025Официальное руководство по языку программирования Rust, включающее основы языка, работу с памятью, многопоточность и другие ключевые аспекты.
2. Сообщество разработчиков Rust на Reddit [Электронный ресурс] - <https://www.reddit.com/r/rust/>. 2025Активное сообщество разработчиков, где обсуждаются вопросы, связанные с разработкой на Rust, включая создание эмуляторов и низкоуровневых систем.
3. Форум Stack Overflow для разработчиков [Электронный ресурс] - <https://stackoverflow.com/>. 2025Популярный форум для разработчиков, где можно найти ответы на вопросы, связанные с Rust, эмуляцией и низкоуровневым программированием.
4. Руководства и статьи по Rust на Rust by Example [Электронный ресурс] - <https://doc.rust-lang.org/rust-by-example/>. 2025Практические примеры и руководства по использованию Rust для решения различных задач, включая создание эмуляторов.
5. Официальная документация по эмуляции и низкоуровневому программированию [Электронный ресурс] - <https://rust-embedded.github.io/book/>. 2025. Руководство по встраиваемым системам и низкоуровневому программированию на Rust, включая работу с памятью и аппаратными компонентами.
6. «Programming Rust: Fast, Safe Systems Development» Джим Бланди, Джейсон Орендорф, 2021Книга, посвящённая разработке безопасных и производительных систем на Rust. Включает разделы по работе с памятью, многопоточности и низкоуровневому программированию.
7. «Rust in Action» Тим Макнамара, 2021. Практическое руководство по использованию Rust для создания системного программного обеспечения, включая эмуляторы и виртуальные машины.
8. «The Rust Programming Language» (The Book) Стив Клабник, Кэрол Николс, 2023. Официальная книга по Rust, которая охватывает все аспекты языка, от основ до продвинутых тем, таких как многопоточность и работа с памятью.
9. Руководства по эмуляции и ретро-программированию [Электронный ресурс] - <https://www.emulator101.com/>, 2025. Сайт, посвящённый созданию эмуляторов для классических компьютеров. Включает примеры и объяснения, которые могут быть полезны для разработки псевдо-ретро компьютера.

# Приложение А

use crate::cpu::{CPU, instructions::Instruction};

use crate::memory::RAM;

use crate::graphics::video\_adapter::{

Container, Button, List, ListItem, VideoAdapter,

FONT\_HEIGHT, FONT\_WIDTH,

UI\_PADDING, UI\_TEXT\_COLOR, UI\_BACKGROUND\_COLOR, UI\_HIGHLIGHT\_COLOR

};

use std::time::{Duration, Instant};

use minifb::Key;

const GRID\_SIZE: usize = 20;

const CELL\_SIZE: usize = 20;

const WIDTH: usize = 640;

const HEIGHT: usize = 480;

const GAME\_SPEED: Duration = Duration::from\_millis(100);

// Memory map constants

const SNAKE\_HEAD\_X: u16 = 0x0000;

const SNAKE\_HEAD\_Y: u16 = 0x0001;

const FOOD\_X: u16 = 0x0002;

const FOOD\_Y: u16 = 0x0003;

const DIRECTION: u16 = 0x0004;

const GAME\_STATE: u16 = 0x0005;

const SNAKE\_LENGTH: u16 = 0x0006;

const SNAKE\_SEGMENTS\_START: u16 = 0x0100; // Snake segments array starts here

// Add UI constants

const SCORE\_X: usize = 10;

const SCORE\_Y: usize = 10;

const BORDER\_THICKNESS: usize = 2;

const BORDER\_COLOR: u32 = 0x444444;

const SCORE\_COLOR: u32 = 0xFFFFFF;

// UI States

#[derive(PartialEq)]

enum GameState {

Menu,

Playing,

Paused,

GameOver,

}

pub struct SnakeGame {

cpu: CPU,

ram: RAM,

video: VideoAdapter,

last\_update: Instant,

game\_state: GameState,

menu\_selected: usize,

}

impl SnakeGame {

pub fn new() -> Self {

let mut game = SnakeGame {

cpu: CPU::new(),

ram: RAM::new(),

video: VideoAdapter::new("Snake Game"),

last\_update: Instant::now(),

game\_state: GameState::Menu,

menu\_selected: 0,

};

// Initialize game state in RAM

game.ram.write(SNAKE\_HEAD\_X, 5);

game.ram.write(SNAKE\_HEAD\_Y, 5);

game.ram.write(FOOD\_X, 10);

game.ram.write(FOOD\_Y, 10);

game.ram.write(DIRECTION, 3); // Right

game.ram.write(GAME\_STATE, 1); // Active

game.ram.write(SNAKE\_LENGTH, 1);

// Initialize first snake segment

game.ram.write(SNAKE\_SEGMENTS\_START, 5);

game.ram.write(SNAKE\_SEGMENTS\_START + 1, 5);

game

}

pub fn run(&mut self) {

while self.video.is\_open() {  // Only check if window is open

self.handle\_input();

let now = Instant::now();

if now.duration\_since(self.last\_update) >= GAME\_SPEED {

match self.game\_state {

GameState::Playing => self.update(),

\_ => {}  // Don't update game state in other modes

}

self.last\_update = now;

}

self.render();

self.video.update\_screen();

}

}

fn handle\_input(&mut self) {

// Store key states in local variables first

let up\_pressed = self.video.get\_window().is\_key\_down(Key::Up);

let down\_pressed = self.video.get\_window().is\_key\_down(Key::Down);

let left\_pressed = self.video.get\_window().is\_key\_down(Key::Left);

let right\_pressed = self.video.get\_window().is\_key\_down(Key::Right);

let enter\_pressed = self.video.get\_window().is\_key\_down(Key::Enter);

let escape\_pressed = self.video.get\_window().is\_key\_down(Key::Escape);

match self.game\_state {

GameState::Menu => {

if up\_pressed && self.menu\_selected > 0 {

self.menu\_selected -= 1;

}

if down\_pressed && self.menu\_selected < 1 {

self.menu\_selected += 1;

}

if enter\_pressed {

