Министерство образования Республики Беларусь
Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей
Кафедра электронных вычислительных машин
Дисциплина: Программирование на языках высокого уровня

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту

на тему

Игра “Flappy Bird”

БГУИР КП 1–40 02 01 216 ПЗ

Студент: группы 250502,
Копылова Е.С.

Руководитель: ассистент каф. ЭВМ
Богдан Е. В.

Минск 2023

Учреждение образования
«Белорусский Учреждение образования
«Белорусский государственный университет информатики
и радиоэлектроники»

Факультет компьютерных систем и сетей

УТВЕРЖДАЮ
Заведующий кафедрой

_____
(подпись)
_____2023 г.

ЗАДАНИЕ
по курсовому проектированию

Студенту _____*Копыловой Екатерине Сергеевне*_____

Тема проекта_____*Игра "Flappy Bird"*_____

2. Срок сдачи студентом законченного проекта____*15 декабря 2023 г.*

3. Исходные данные к проекту *Язык программирования – C++, дополнительная библиотека - SFML*

4. Содержание расчетно-пояснительной записки (перечень вопросов, которые подлежат разработке)

1. Лист задания.
2. Введение.
3. Обзор литературы.
   3.1. Обзор методов и алгоритмов решения поставленной задачи.
4. Функциональное проектирование.
   4.1. Структура входных и выходных данных.
   4.2. Разработка диаграммы классов.
   4.3. Описание классов.

5. Разработка программных модулей.

5.1. Разработка схем алгоритмов(два наиболее важных метода).

5.2. Разработка алгоритмов (описание алгоритмов по шагам, для двух методов).

6. Результаты работы.

7. Заключение

8. Литература

9. Приложения

5. Перечень графического материала (с точным обозначением обязательных чертежей и графиков)

1. Диаграмма классов.

2. Схема основного хода игры.

3. Схема обновления состояний игры.

4. Ведомость документов.

6. Консультант по проекту (с обозначением разделов проекта) Е. В. Богдан

7. Дата выдачи задания *15.09.2023г.*

8. Календарный график работы над проектом на весь период проектирования (с обозначением сроков выполнения и трудоемкости отдельных этапов):

1. Выбор задания. Разработка содержания пояснительной записки. Перечень графического материала – 15 %;

разделы 2, 3 – 10 %;

разделы 4 к –20 %;

разделы 5 к – 35 %;

раздел 6,7,8 – 5 %;

раздел 9 к – 5%;

оформление пояснительной записки и графического материала к 15.12.22 – 10 %

Защита курсового проекта с 21.12 по 28.12.23г.

РУКОВОДИТЕЛЬ Е.В. Богдан

(подпись)

Задание принял к исполнению_____

(дата и подпись студента)

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В настоящее время игры являются одним из самых популярных видов развлечения. Одной из самых знаменитых и простых игр, которая завоевала миллионы пользователей, является Flappy Bird. В данной курсовой работе мы рассмотрим особенности игры Flappy Bird, ее популярность и механику игрового процесса на языке программирования C++.

C++ - это мощный и универсальный язык программирования, который широко используется при разработке программных приложений, системного программного обеспечения, драйверов устройств и встроенного микропрограммного обеспечения.

С тех пор этот язык эволюционировал и стал одним из самых популярных и широко используемых языков программирования в мире. Его популярность обусловлена его эффективностью, гибкостью и широким спектром применений, для которых он может быть использован. C++ известен своей высокой производительностью, поскольку позволяет выполнять низкоуровневые манипуляции с оборудованием и памятью, что делает его подходящим для разработки ресурсоемких приложений.

1 ПОСТАНОВКА ЗАДАЧИ

«Flappy Bird» представляет собой игру в жанре аркада с удобным пользовательским интерфейсом. В ней необходимо при помощи нажатия на левую кнопку контролировать полёт птицы между рядами труб, не задевая их.

«Flappy Bird» имеет игровой процесс с участием 2D-графики. Цель игры состоит в управлении полётом птицы, которая непрерывно передвигается между рядами зелёных труб. При столкновении с ними происходит завершение игры. Управление производится касанием экрана, при котором птица совершает небольшой рывок вверх. При отсутствии рывков птица падает из-за силы тяжести, и игра также завершается. Очки набираются при каждом успешном перелёте между двумя трубами. Геймплей не имеет изменений на протяжении всей игры.

Для реализации игрового процесса используется SFML. SFML (*Simple and Fast Multimedia Library* — простая и быстрая мультимедийная библиотека) — свободная кроссплатформенная мультимедийная библиотека. Написана на C++, но доступна также
для C, C#, .Net, D, Java, Python, Ruby, OCaml, Go и Rust. Представляет собой объектно-ориентированный аналог SDL. SFML содержит ряд модулей для простого программирования игр и мультимедиа приложений.

Также для реализации игрового процесса используются математические и физические законы для создания движения главного персонажа и труб.

2 ОБЗОР ЛИТЕРАТУРЫ

2.1 Анализ существующих аналогов

Тема курсового проекта была выбрана в первую очередь для получения знаний в области разработки десктопных приложений с использованием ООП. В процессе создания приложения будут затронуты главные принципы ООП, работа с мультимедиа и графикой. Для того, чтобы создать корректно работающее приложение, необходимо иметь представление об уже реализованных аналогах и изучить содержание источника.

2.1.1 Приложение Badland

Badland – это игра, выполненная в сочетании жанров adventure, action и platformer. На протяжении всего прохождения игрок управляет неким существом, которое при нажатии на экран взлетает, а при отпускании его — падает вниз. При помощи комбинирования этих простейших действий игроку предстоит двигаться слева направо в двухмерном мире, перелетая через препятствия, справляться с не очень сложными, но порой довольно хитрыми головоломками, чтобы достичь выхода с уровня. Имеет еще ряд преимуществ: удобный интерфейс для приятного игрового процесса, высокая оптимизация, интуитивно понятное управление ходом игры. На рисунке 2.1 показана часть интерфейса данного приложения.

Рисунок 2.1 – Скриншот Badland

2.1.2 Приложение Fly Catbug Fly!

Fly Catbug Fly - это аркадная игра, в которой игрок управляет персонажем по имени Котбаг, который летит на своем джетпаке сквозь препятствия и собирает монеты. Цель игры - набрать как можно больше очков, не сталкиваясь с препятствиями. Игрок нажимает на экран, чтобы поднять Котбага выше и отпускает экран, чтобы опустить его. Препятствия включают в себя стены и бомбы, которые нужно избегать. Монеты можно собирать, чтобы набрать дополнительные очки. Игра имеет несколько уровней сложности, которые открываются по мере прохождения игроком. Также есть возможность покупки новых джетпаков и улучшений для Котбага за монеты, которые он собирает во время игры. Графика игры выполнена в ярком и красочном стиле, что делает ее привлекательной для игроков всех возрастов. Игра также имеет приятную музыку и звуковые эффекты, которые добавляют атмосферности и увлекательности в игру.. Рисунок 2.2 демонстрирует базовый интерфейс приложения Fly Catbug Fly!.

Рисунок 2.2 – Скриншот Fly Catbug Fly!

2.1.3 Приложение Rocket Romeo

Rocket Romeo - это аркадная игра, в которой игрок управляет космическим кораблем по имени Ромео, который летит сквозь препятствия и собирает звезды. Цель игры - набрать как можно больше очков, не сталкиваясь с препятствиями. Игрок нажимает на экран, чтобы запустить ракетные двигатели корабля и поднять его выше, отпуская экран, чтобы корабль

опустился. Препятствия включают в себя астероиды, космические мины и лазерные лучи, которые нужно избегать. Звезды можно собирать, чтобы набрать дополнительные очки. Игра имеет несколько уровней сложности, которые открываются по мере прохождения игроком. Также есть возможность покупки новых ракетных двигателей и улучшений для корабля за звезды, которые он собирает во время игры. Графика игры выполнена в стиле научной фантастики, что делает ее привлекательной для любителей космических приключений. Игра также имеет приятную музыку и звуковые эффекты, которые добавляют атмосферности и увлекательности в игру. Рисунок 2.3 демонстрирует базовый интерфейс приложения Rocket Romeo.

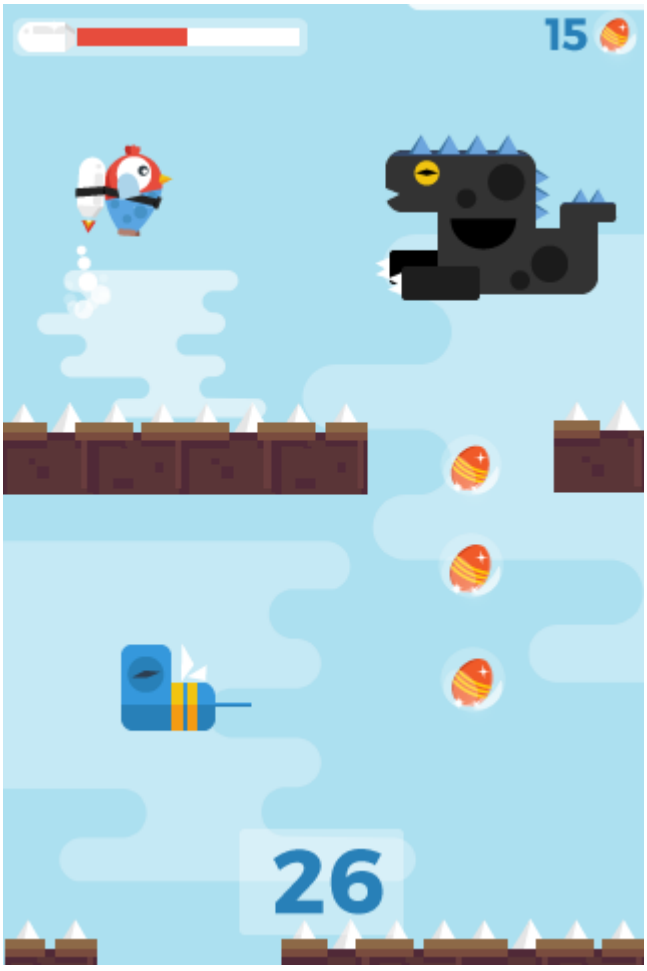

Рисунок 2.3 – Скриншоты Rocket Romeo

2.2 Требования к работе программы

После рассмотрения аналогов данного курсового проекта, становится понятно, что подобного рода приложения, в основном, включают в себя основные функции:

- создание окна игры и отрисовка графики;
- реализация движения главного персонажа и препятствий;
- разработка логики столкновения персонажа с препятствиями;
- реализация системы счёта, отображение на экране и хранение результата в текстовом файле;
- добавление звуковых эффектов и музыки в игру;
- разработка меню игры и возможности перезапуска игры;

Для создания приложения был выбран язык C++, так как он поддерживает использование объектно–ориентированной парадигмы, что делает его удобным для создания больших и сложных систем, разбитых на модули и классы. C++ остается одним из самых популярных и востребованных языков программирования.

3 СИСТЕМНОЕ ПРОЕКТИРОВАНИЕ

После определения требований к функционалу разрабатываемому приложению его следует разбить на функциональные блоки. Такой подход упростит понимание создания приложения, а также позволит легкую оптимизацию.

3.1 Модуль птица

Этот модуль отвечает за хранение и управление информацией о главном персонаже - птице. Его задачи включают:

- Хранение информации о птице: содержит данные о птице, включая ее рисовку;
- Анимация птицы: движение птицы;

3.2 Модуль трубы

Модуль трубы отвечает за хранение и управление информацией о движущихся препятствиях в игре – трубах. Его задачи включают:

- Хранение информации о трубах: содержит данные о трубах, включая рисовку;
- Анимация труб: движение труб;

3.3 Модуль отображения всей игры

Модуль отображения всей игры является главной частью приложения, позволяющей пользователям взаимодействовать с самой игрой.

В модуль входят создание игрового окна, сборка всех элементов игры (птица, трубы, меню игры, звуковые эффекты и музыка) в готовое приложение. Здесь реализуется вся игровая графика: движение труб, движение, падение, подъем птицы при нажатии на правую кнопку мыши, счётчик игровых очков и т.д.

3.4 Модуль хранения результатов игры

Модуль управления полками представляет собой функционал приложения, который отвечает за хранение результатов всех игр в текстовом файле.

4 ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ

В данном разделе описывается функционирование и структура разрабатываемого приложения.

4.1 Входные и выходные данные

Пользователь начинает игру нажатием на кнопку, выходными данными являются движение птицы и труб, отображение анимации объектов, текстовые уведомления пользователя о выигрыше и проигрыше, результат игры, записанный в текстовом файле.

4.2 Разработка диаграммы классов

Диаграмма классов представлена в Приложении А.

4.3 Описание классов

4.3.1 Класс AssetManager

Этот класс AssetManager представляет собой управление действиями для объектов.

Он необходим для загрузки текстур и шрифтов объектов игры.

Поля:

- `std::map<std::string, sf::Texture> _textures` – текстуры объектов;
- `std::map<std::string, sf::Font> _fonts` – шрифты объектов.

Методы:

- `AssetManager()` – конструктор для создания объектов без начальных значений;
- `~AssetManager()` – деструктор класса;
- `void LoadTexture(std::string name, std::string filename)` – загружает текстуру по её имени и имени файла, который ее хранит;
- `sf::Texture &GetTexture(std::string name)` – получает текстуру по её имени;
- `void LoadFont(std::string name, std::string filename)` – загружает шрифт по его имени и имени файла, который его хранит;
- `sf::Texture &GetFont(std::string name)` – получает шрифт по его имени.

4.3.2 Класс Bird

Этот класс представляет собой реализацию функционала главного персонажа – птицы: рисовка птицы, анимация птицы, движение птицы.

Поля:

- `GameDataRef _data` – данные для создания объекта;
- `sf::Sprite _birdSprite` – созданная птица;
- `unsigned int _animationIterator` – итератор анимации;
- `sf::Clock _clock` – прошедшее время;
- `sf::Clock _movementClock` – время движения;
- `std::vector<sf::Texture> _animationFrames` – вектор структур, в котором хранятся кадры для анимации птицы;
- `int _birdState` – состояние птицы;
- `float _rotation` – поворот птицы.

Методы:

- `Bird(GameDataRef data)` – конструктор класса `Bird`, в который передаются данные птицы, которые необходимы для создания объекта птицы;
- `~Bird` () – деструктор класса `Bird`;
- `void Draw()` – метод для рисовки птицы;
- `void Animate(float dt)` – метод для создания анимации объекта птицы, в который передаётся число `dt` для расчёта частоты кадров при движении;
- `void Update(float dt)` – метод для обновления состояния птицы (подъём, спуск, столкновение с препятствием), который передаётся число `dt` для расчёта частоты кадров при движении;
- `void Tap()` – действие при нажатии на кнопку мыши;
- `const sf::Sprite &GetSprite() const` – метод для получения объекта птицы.

4.3.3 Класс Collision

Данный класс предназначен для проверки столкновения объекта с препятствием.

Методы:

- `Collision()` – конструктор класса `Collision`, который выполняет только инициализацию возможного препятствия;
- `~Collision()` – деструктор класса `Collision`;
- `bool CheckSpriteCollision(sf::Sprite sprite1, float scale1, sf::Sprite sprite2, float scale2)` – этот метод проверяет, есть ли столкновение объекта (птицы) с препятсивем(трубами).

4.3.4 Класс Flash

Класс представляет собой «вспышку» при столкновении объекта с препятствием.

Поля:

- `GameDataRef _data` – данные для создания объекта;
- `sf::RectangleShape _shape` – фигура вспышки;
- `bool _flashOn` – состояние включенной вспышки;

Методы:

- `Flash(GameDataRef data)` – конструктор класса `Flash`, в который передаются данные для создания объекта вспышки;
- `~Flash()` – деструктор класса `Flash`;
- `void Show(float dt)` – показать вспышку при столкновении птицы с трубой;
- `void Draw()` – нарисовать вспышку.

4.3.5 Класс Game

Данный класс предназначен для реализации игрового процесса, создания окна приложения.

Поля:

- `const float dt = 1.0f / 60.0f`;
- `sf::Clock _clock` – прошедшее время;
- `GameDataRef _data` – данные, используемые для построения игры.

Методы:

- `Game(int width, int weight, std::string title)` – конструктор класса `Game`, в котором создаётся окно приложения, задаются его размеры и название;
- `void Run()` – запуск игры.

4.3.6 Класс GameOverState

Класс является частью игрового процесса, который используется для вывода сообщения о проигрыше в случае столкновения объекта с препятствием, наследуется от класса `State`.

Поля:

- `GameDataRef _data` – данные для создания объекта;
- `sf::Sprite _background` – фон;
- `sf::Sprite _gameOverTitle`;
- `sf::Sprite _gameOverContainer`;
- `sf::Sprite _retryButton` – кнопка для новой попытки;
- `sf::Sprite _medal` – медаль;
- `sf::Text _scoreText` – результат игры, который выводится на экран в случае проигрыша;

- `sf::Text _highScoreText` – наилучший результат игры, который выводится на экран в случае проигрыша;
- `int _score` – результат игры;
- `int _highScore` - наилучший результат игры.

Методы:

- `GameOverState(GameDataRef data, int score)` – конструктор класса `GameOverState`, в котором создаётся окно, появляющееся в случае проигрыша, результат изначально равен нулю;
- `void Init()` – инициализация;
- `void HandleInput()` – метод, который обрабатывает нажатие на кнопку мыши;
- `void Update(float dt)` – метод для обновления состояния проигрыша;
- `void Draw(float dt)` – нарисовать проигрыш.

4.3.7 Класс GameState

Класс предназначен для отслеживания хода игры и в случае разных ситуаций выполнения соответствующих действий, наследуется от класса `State`.

Поля:

- `GameDataRef _data` – данные для создания объекта;
- `sf::Sprite _background` – фон;
- `Pipe *pipe` – указатель на объект труба;
- `Land *land` – указатель на объект земля;
- `Bird *bird` – указатель на объект птица:
- `Collision *collision` – указатель на столкновение;
- `Flash *flash` – указатель на объект вспышка;
- `HUD *hud`;
- `sf::Clock clock` – прошедшее время;
- `int _gameState` – состояние игры;
- `sf::RectangleShape _gameOverFlash` – фигура проигрыша;
- `bool _flashOn` – состояние включенной вспышки;
- `int _score` – счётчик результата игры;
- `sf::SoundBuffer _hitSoundBuffer` – буфер для звука столкновения объекта с препятствием;
- `sf::SoundBuffer _wingSoundBuffer` – буфер для звука взмаха крыльев объекта птица;
- `sf::SoundBuffer _pointSoundBuffer` – буфер для звука прохождения объекта птица между трубами;
- `sf::Sound _hitSound` – звук столкновения объекта с препятствием;

- `sf::Sound _wingSound` – звук взмаха крыльев объекта птица;
- `sf::Sound _pointSound` – звук прохождения объекта птица между трубами.

Методы:
- `GameState(GameDataRef data)` – конструктор класса `GameState`;
- `void Init()` – инициализация;
- `void HandleInput()` – метод, который обрабатывает нажатие на кнопку мыши;
- `void Update(float dt)` – метод для обновления состояния игры;
- `void Draw(float dt)` – нарисовать состояние игры.

4.3.8 Класс HUD

Этот класс собирает статистику за все игры, которые были совершены.
Поля:
- `GameDataRef _data` – данные для создания объекта;
- `sf::Text _scoreText` – результат игры.

Методы:
- `HUD(GameDataRef data)` – конструктор класса `HUD`;
- `~HUD()` – деструктор класса `HUD`;
- `void Draw()` – нарисовать статистику;
- `void UpdateScore(int score)` – обновить статистику.

4.3.9 Класс InputManager

Класс представляет собой проверку на то, был ли нажат объект кнопкой мыши или нет.
Методы:
- `InputManager()` – конструктор класса `InputManager`;
- `~InputManager()` – деструктор класса `InputManager`;
- `bool IsSpriteClicked(sf::Sprite object, sf::Mouse::Button button, sf::RenderWindow &window)` – проверяет, был ли нажат кнопкой мыши объект или нет.

4.3.10 Класс Land

Класс предназначен для отрисовки земли и создания её анимации.
Поля:
- `GameDataRef _data` – данные для создания объекта;
- `std::vector<sf::Sprite> _landSprites` – вектор с объектами земли.

Методы:

- `Land(GameDataRef data)` – конструктор класса;
- `void MoveLand(float dt)` – движение земли;
- `void DrawLand()` – нарисовать землю;
- `const std::vector<sf::Sprite> &GetSprite() const` – получить объекты.

4.3.11 Класс MainMenuState

Этот класс реализует окно главного меню игры, наследуется от класса `State`.

Поля:

- `GameDataRef _data` – данные для создания объекта;
- `sf::Sprite _background` – фон;
- `sf::Sprite _title` – заголовок;
- `sf::Sprite _playButton` – кнопка «Играть».

Методы:

- `MainMenuState(GameDataRef data)` – конструктор класса;
- `void Init()` – инициализация;
- `void HandleInput()` – метод, который обрабатывает нажатие на кнопку мыши;
- `void Update(float dt)` – метод для обновления состояния главного меню;
- `void Draw(float dt)` – нарисовать главное меню.

4.3.12 Класс Pipe

Класс `Pipe` представляет собой создание объекта труба его анимацию и движение.

Поля:

- `GameDataRef _data` – данные для создания объекта;
- `std::vector<sf::Sprite> pipeSprites` – вектор объектов труба;
- `std::vector<sf::Sprite> scoringPipes` – вектор «пройденных» объектов труба;
- `int _landHeight` – высота трубы;
- `int _pipeSpawnYOffset` – сдвиг труб при появлении по оси Y.

Методы:

- `Pipe(GameDataRef data)` – конструктор класса `Pipe` инициализирует объект класса;

- `void SpawnBottomPipe()` – появление нижних труб;
- `void SpawnTopPipes()` – появление верхних труб;
- `void SpawnInvisiblePipes()` – появление невидимых труб;
- `void MovePipes(float dt)` – движение труб;
- `void DrawPipes()` – нарисовать трубы;
- `void RandomisePipeOffset()` – генерация случайного сдвига труб при появлении по оси Y;
- `const std::vector<sf::Sprite> &GetSprite() const` – получить вектор объектов труба;
- `std::vector<sf::Sprite> &GetScoringSprites()` – получить вектор «пройденных» труб.

4.3.13 Класс SplashState

В данном классе отображается состояние столкновения объекта с препятствием. Класс наследуется от класса `State`.

Поля:

- `GameDataRef _data` – данные для создания объекта;
- `sf::Sprite _background` – фон;
- `sf::Clock _clock` – прошедшее время.

Методы:

- `SplashState(GameDataRef data)` – конструктор класса;
- `void Init()` – инициализация;
- `void HandleInput()` – метод, который обрабатывает нажатие на кнопку мыши;
- `void Update(float dt)` – метод для обновления состояния столкновения;
- `void Draw(float dt)` – нарисовать состояние столкновения.

4.3.14 Класс State

Класс `State` является базовым классом всех состояний игры. В нём реализуются возможности поставить игру на паузу или же продолжить её.

Методы:

- `virtual void Init() = 0;`
- `virtual void HandleInput() = 0;`
- `virtual void Update(float dt) = 0;`
- `virtual void Draw(float dt) = 0;`
- `virtual void Pause();`
- `virtual void Resume().`

4.3.15 Класс StateMachine

Данный класс реализует отображение состояний игры, а также действия с ними.

Поля:

- `std::stack<StateRef> _states` – список состояний;
- `StateRef _newState` – новое состояние игры;
- `bool _isRemoving` – флаг удаления состояния;
- `bool _isAdding` – флаг добавления состояния;
- `bool _isReplacing` – флаг замены состояния.

Методы:

- `StateMachine()` – конструктор класса;
- `~StateMachine()` – деструктор класса ;
- `void AddState(StateRef newState, bool isReplacing = true)` – добавить новое состояние. В этом методе создаётся новое состояние, флаг замены состояния имеет значение истина;
- `void ProcessStateChanges()` – обработка изменений состояния;
- `StateRef &GetActiveState()` – получение активного состояния.

5 РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ

В данном разделе рассмотрены описания алгоритмов, используемых в программе.

5.1 Алгоритм основного хода игры

Для алгоритма по шагам рассмотрены методы класса `Game`.

Шаг 1. Начало метода.

Шаг 2. Объявление переменных newTime, frameTime, interpolation и присваивание им значений 0.

Шаг 3. Получение текущего времени в секундах и присваивание его переменной currentTime.

Шаг 4. Обнуление переменной accumulator.

Шаг 5. Начало цикла, который продолжается до тех пор, пока окно игры открыто, иначе переход к Шагу 17.

Шаг 6. Вызов метода ProcessStateChanges() у объекта класса StateMachine, который обрабатывает изменения состояний игры.

Шаг 7. Получение нового времени в секундах и вычисление разницы между ним и currentTime, присваивание этой разницы переменной frameTime.

Шаг 8. Если значение frameTime больше 0.25, то присваиваем ему значение 0.25.

Шаг 9. Присваивание переменной currentTime значения newTime.

Шаг 10. Увеличение значения accumulator на значение frameTime.

Шаг 11. Начало цикла, который продолжается до тех пор, пока accumulator не станет меньше или равным dt (константа времени обновления игры), иначе перейти к Шагу 17.

Шаг 12. Вызов метода HandleInput() у текущего активного состояния игры для обработки ввода.

Шаг 13. Вызов метода Update() у текущего активного состояния игры для обновления игрового состояния.

Шаг 14. Вычитание значения dt из accumulator.

Шаг 15. Расчет значения interpolation как отношения оставшегося времени в accumulator к dt.

Шаг 16. Вызов метода Draw() у текущего активного состояния игры для отрисовки игровых объектов с учетом interpolation.

Шаг 17. Конец метода.

Схема алгоритма представлена в Приложении Б.

5.2 Алгоритм обновления состояний игры

Для алгоритма по шагам рассмотрен метод `void Update(float dt)` класса `GameState`.

Шаг 1. Начало метода.

Шаг 2. Проверка, что игра не находится в состоянии "Game Over", иначе переход к Шагу 31.

Шаг 3. Если игра не находится в состоянии "Game Over", то анимируем птицу и двигаем землю.

Шаг 4. Если игра находится в состоянии "Playing", то двигаем трубы и проверяем, прошло ли достаточно времени для создания новых труб.

Шаг 5. Если прошло достаточно времени, то создаем невидимую трубу, верхнюю трубу, нижнюю трубу и трубу для подсчета очков.

Шаг 6. Обновление положения птицы.

Шаг 7. Получение списка спрайтов земли.

Шаг 8. Начало цикла, в котором инициализируем переменную i нулём, которая итерируется после каждого шага, действие выполняется, пока i меньше размера списка спрайтов земли, иначе переход к Шагу 13.

Шаг 9. Проверка на столкновение каждого из спрайтов земли с птицей.

Шаг 10. Присваивание состоянию игры состояние "Game Over".

Шаг 11. Перезагрузка времени.

Шаг 12. Воспроизведение звука столкновения птицы с объектом.

Шаг 13. Получение списка спрайтов труб.

Шаг 14. Начало цикла, в котором инициализируем переменную i нулём, которая итерируется после каждого шага, действие выполняется, пока i меньше размера списка спрайтов труб, иначе переход к Шагу 19.

Шаг 15. Проверка на столкновение каждого из спрайтов труб с птицей.

Шаг 16. Присваивание состоянию игры состояние "Game Over".

Шаг 17. Перезагрузка времени.

Шаг 18. Воспроизведение звука столкновения птицы с объектом.

Шаг 19. Проверка состояния игры на состояние "Playing".

Шаг 20. Получение списка спрайтов для подсчета очков.

Шаг 21. Начало цикла, в котором инициализируем переменную i нулём, которая итерируется после каждого шага, действие выполняется, пока i меньше размера списка спрайтов подсчитанных труб, иначе переход к Шагу 26.

Шаг 22. Проверка прохождения подсчитанных труб.

Шаг 23. Итерация счётчика результата.

Шаг 24. Обновление текущего результата.

Шаг 25. Воспроизведение звука прохождения через трубы.

Шаг 26. Проверка игры на состояние "Game Over.

Шаг 27. Демонстрация вспышки.

Шаг 28. Проверка, прошло ли достаточно времени для перехода на экран "Game Over".

Шаг 29. Переход на экран "Game Over".

Шаг 30. Передача количества набранных очков.

Шаг 31. Конец метода.

Схема алгоритма представлена в Приложении Б.

6 РЕЗУЛЬТАТЫ РАБОТЫ

При запуске программы отображается начальное окно (Рисунок 6.1). Пользователь должен нажать на кнопку «play», чтобы начать игру.

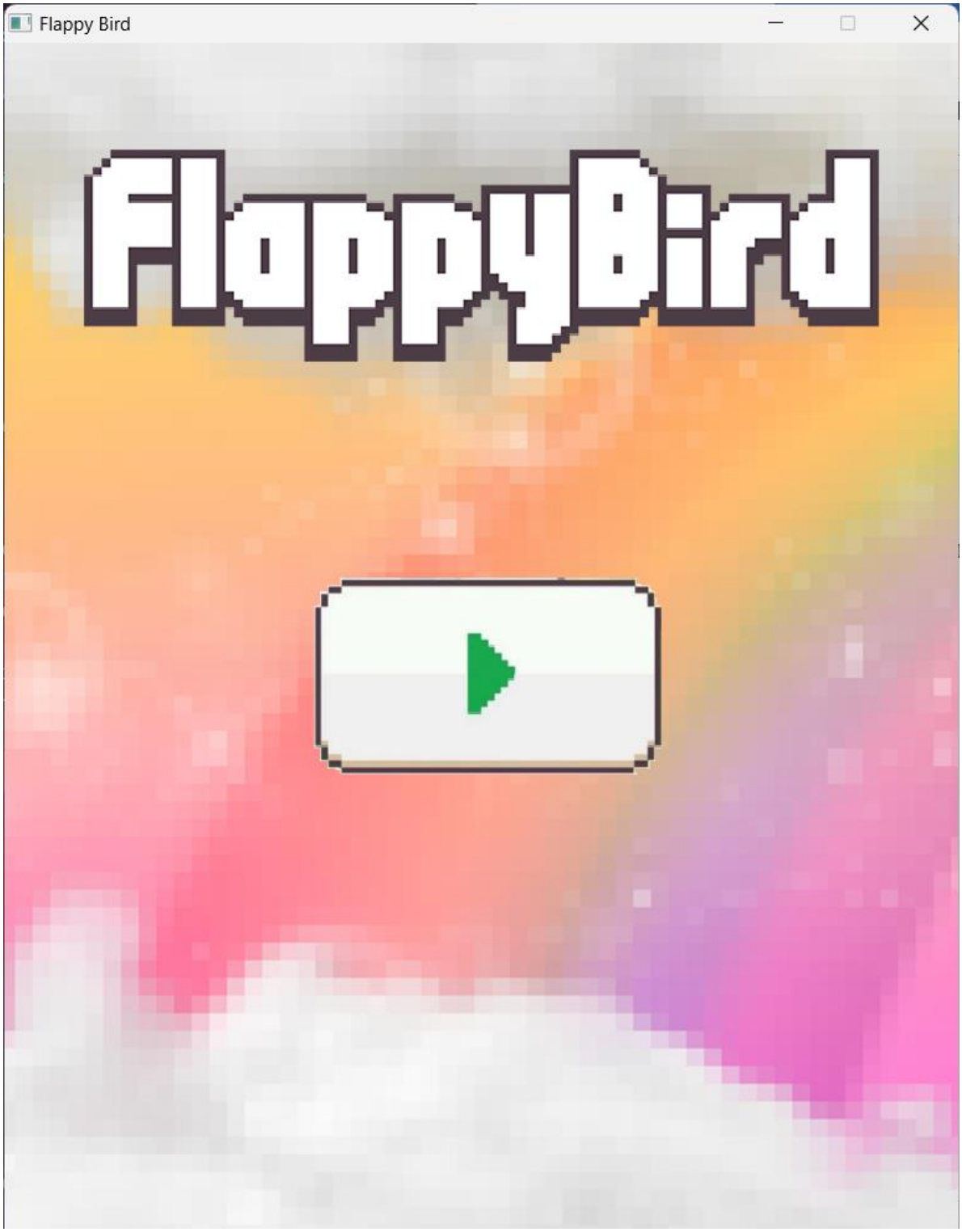

Рисунок 6.1 – Начальное окно.

После нажатия кнопки начала игры данное окно сворачивается и открывается основное игровое окно (Рисунок 6.2). Для того, чтобы птица совершала полёт, необходимо кнопкой мыши, после чего она сначала поднимается вверх и затем опускается вниз до следующего нажатия на кнопку мыши. Это можно увидеть на Рисунке 6.3.

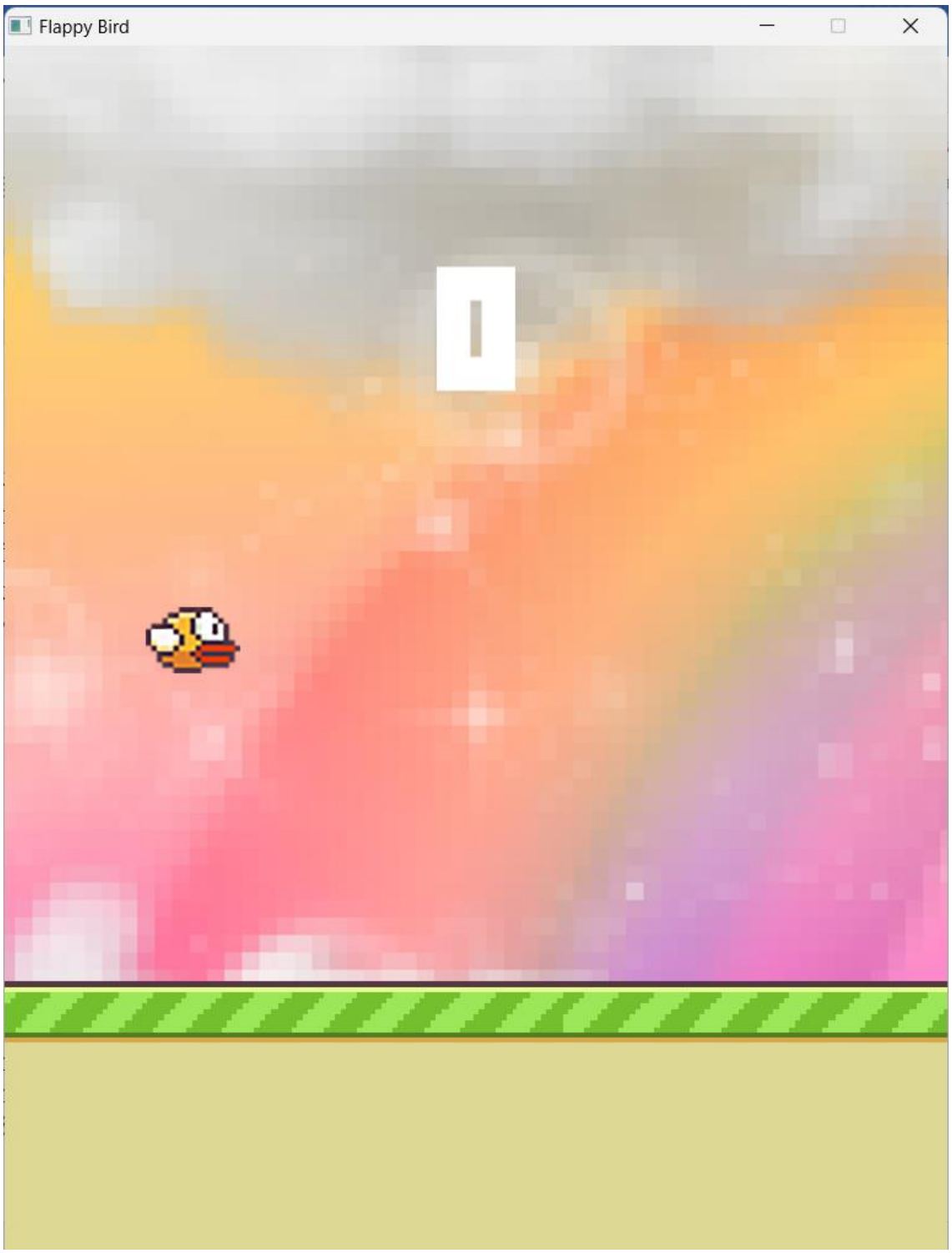

Рисунок 6.2 – Начало игры.

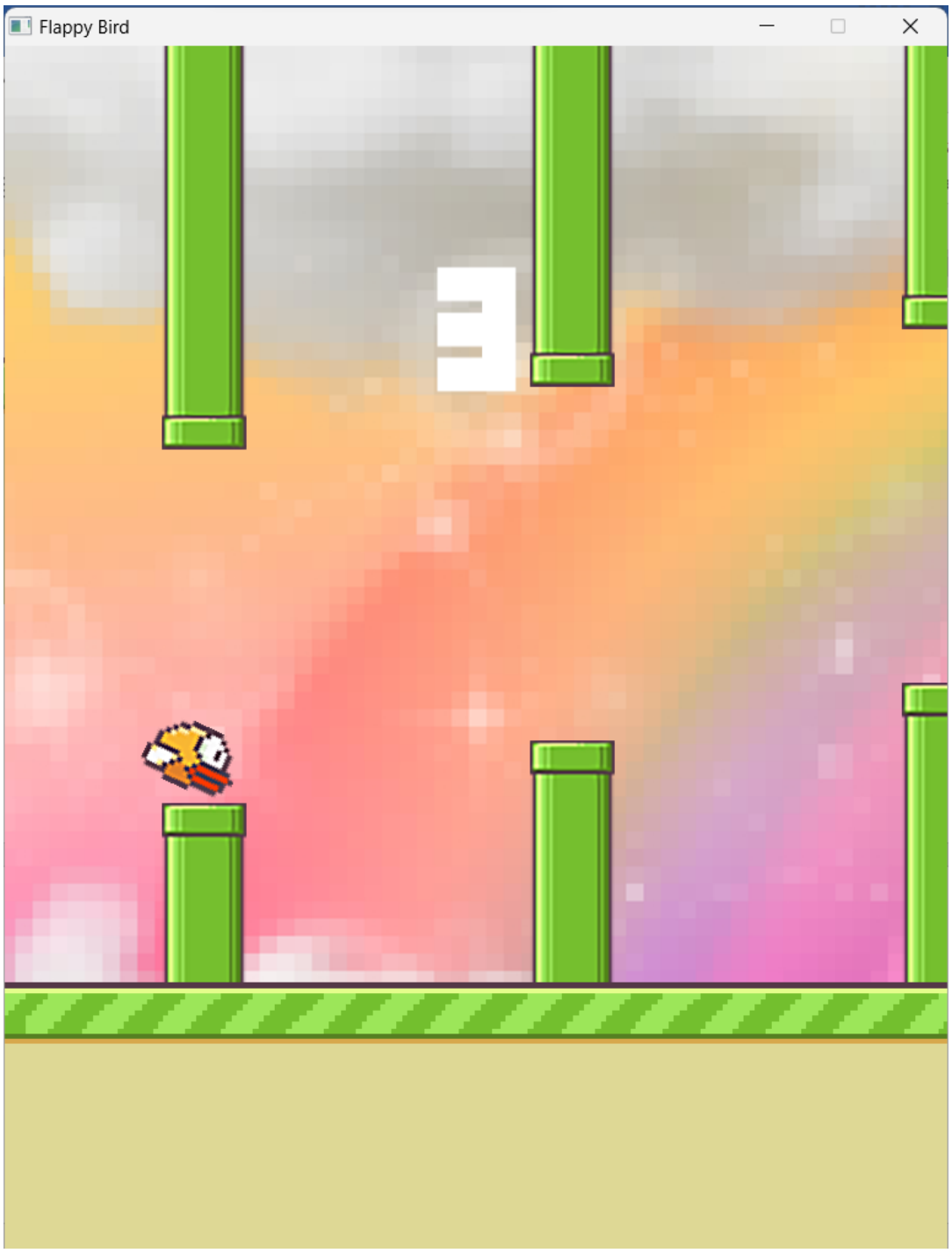

Рисунок 6.3 – Игровой процесс.

При столкновении птицы с трубой либо же при полном падении на землю игра считается законченной и окно игрового процесса сменяется на окно окончания игры. Это можно увидеть на Рисунках 6.4(а), 6.5(б).

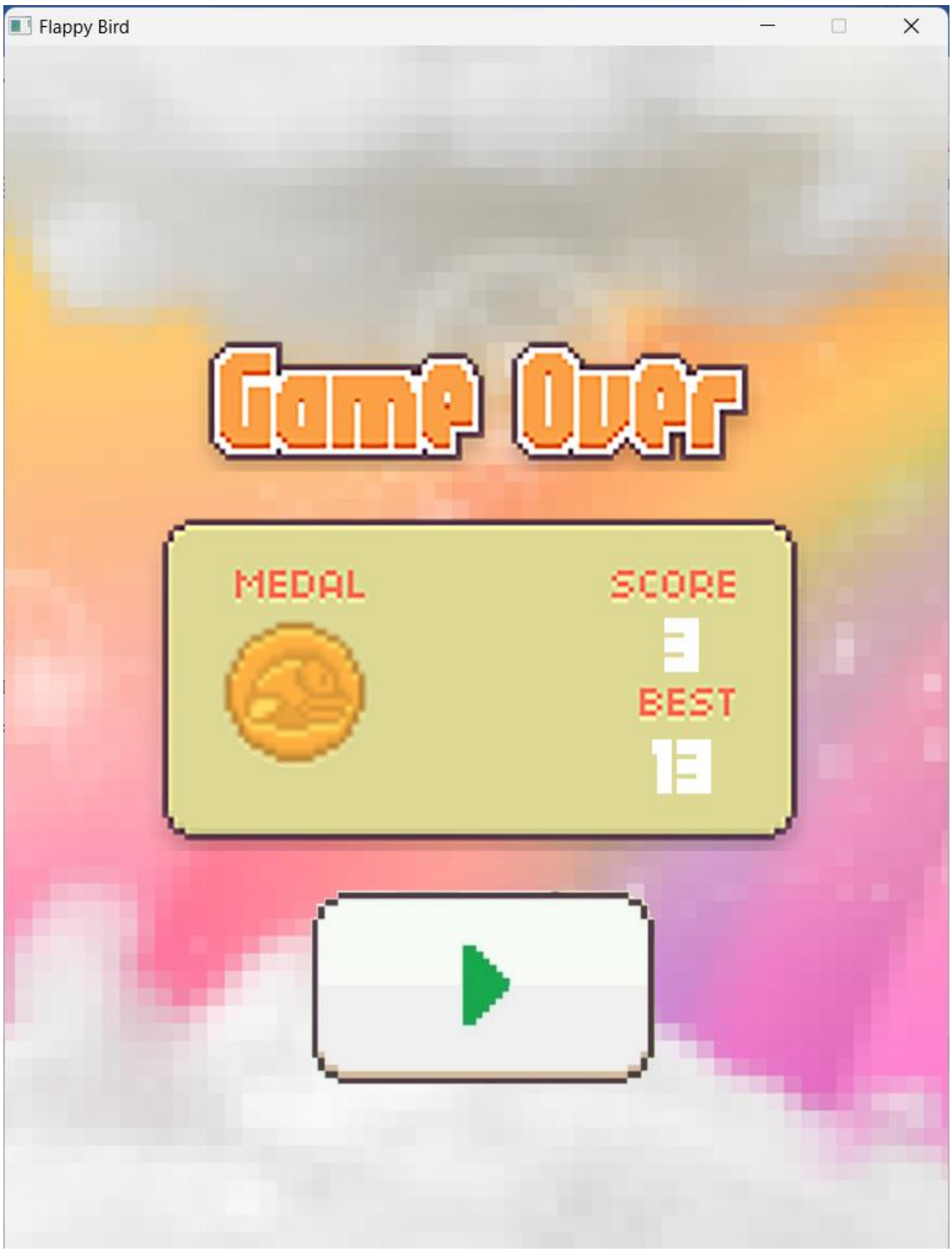

Рисунок 6.4(а) – Конец игры в случае столкновения.

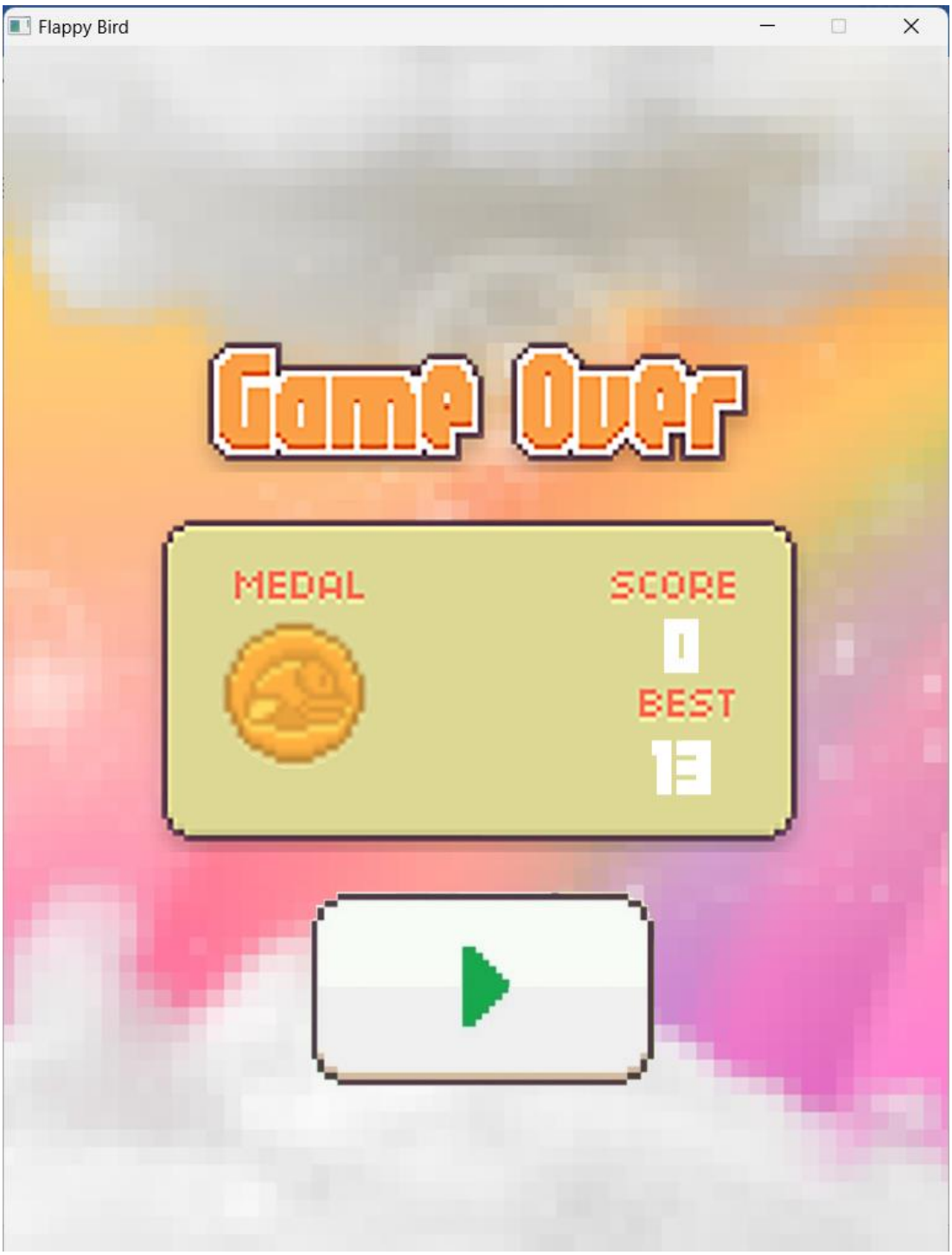

Рисунок 6.4(б) – Конец игры в случае падения.

ЗАКЛЮЧЕНИЕ

Данный курсовой проект представляет собой разработанное приложение, которое обладает широким функционалом, интуитивно понятным пользовательским интерфейсом. В ходе его создания были успешно достигнуты поставленные цели, и весь запланированный функционал был реализован полностью и эффективно.

Для создания программного продукта было проведено детальное изучение библиотеки C++ SFML. Исследования помогли получить глубокое понимание особенностей каждого элемента, что сыграло ключевую роль в успешной реализации проекта.

Основное внимание уделено языку программирования C++, где были усвоены основы объектно-ориентированного программирования (ООП), что позволило эффективно использовать его возможности при создании различных модулей приложения. Опыт написания анимации открыл новые возможности по созданию оконных приложений с приятным пользовательским интерфейсом.

Работа над проектом была разбита на этапы: анализ аналогов и литературных источников, постановка требований, проектирование, конструирование, разработка модулей и тестирование. Благодаря последовательности выполнения каждого этапа был достигнут результат – функциональный и стабильно работающий программный модуль «игра Flappy Bird».

В будущем планируется улучшение текущего функционала. Планируется добавить новые возможности, такие как изменение уровня сложности, добавление новых анимаций и т.д.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Документация библиотеки SFML [Электронный ресурс]. - Электронные данные. -Режим доступа: https://www.sfml-dev.org/documentation/2.6.1/ - Дата доступа: 27.11.2023
2. Язык программирования C++, – Бьерн Страуструп, 1985
3. Язык программирования C. Второе издание, –Деннис Ричи, Брайан Кернигхан, 1978
4. Объектно-ориентированное программирование в C++, – Роберт Лафоре, 2019

ПРИЛОЖЕНИЕ А
(обязательное)

Диаграмма классов

ПРИЛОЖЕНИЕ Б

(обязательное)

Схема алгоритма основного хода игры

ПРИЛОЖЕНИЕ В

(обязательное)

Схема алгоритма изменения состояния игры

ПРИЛОЖЕНИЕ Г
(обязательное)

Листинг кода

Файл AssetManager.h:
```
#pragma once

#include <map>
#include <SFML/Graphics.hpp>

    class AssetManager
    {
    public:
        AssetManager() { }
        ~AssetManager() { }

        void LoadTexture(std::string name, std::string
fileName);
        sf::Texture &GetTexture(std::string name);

        void LoadFont(std::string name, std::string fileName);
        sf::Font &GetFont(std::string name);

    private:
        std::map<std::string, sf::Texture> _textures;
        std::map<std::string, sf::Font> _fonts;
    };
```

Файл AssetManager.cpp:
```
#include <SFML/Graphics.hpp>
#include "AssetManager.h"

    void AssetManager::LoadTexture(std::string name, std::string
fileName)
    {
        sf::Texture tex;

        if (tex.loadFromFile(fileName))
        {
            this->_textures[name] = tex;
        }
    }

    sf::Texture &AssetManager::GetTexture(std::string name)
    {
        return this->_textures.at(name);
    }

    void AssetManager::LoadFont(std::string name, std::string
fileName)
```

```
    {
        sf::Font font;

        if (font.loadFromFile(fileName))
        {
            this->_fonts[name] = font;
        }
    }

    sf::Font &AssetManager::GetFont(std::string name)
    {
        return this->_fonts.at(name);
    }
```

Файл Bird.h:

```
#pragma once

#include <SFML/Graphics.hpp>

#include "DEFINITIONS.h"
#include "Game.h"

#include <vector>

class Bird
    {
    public:
        Bird(GameDataRef data);
        ~Bird();

        void Draw();

        void Animate(float dt);

        void Update(float dt);

        void Tap();

        const sf::Sprite &GetSprite() const;

    private:
        GameDataRef _data;

        sf::Sprite _birdSprite;
        std::vector<sf::Texture> _animationFrames;

        unsigned int _animationIterator;

        sf::Clock _clock;

        sf::Clock _movementClock;
```

```
        int _birdState;

        float _rotation;

    };
```

Файл Bird.cpp:

```
#include "Bird.h"

Bird::Bird(GameDataRef data) : _data(data)
    {
        _animationIterator = 0;

        _animationFrames.push_back(this->_data-
>assets.GetTexture("Bird Frame 1"));
        _animationFrames.push_back(this->_data-
>assets.GetTexture("Bird Frame 2"));
        _animationFrames.push_back(this->_data-
>assets.GetTexture("Bird Frame 3"));
        _animationFrames.push_back(this->_data-
>assets.GetTexture("Bird Frame 4"));


_birdSprite.setTexture(_animationFrames.at(_animationIterator));

        _birdSprite.setPosition((_data->window.getSize().x / 4)
- (_birdSprite.getGlobalBounds().width / 2), (_data-
>window.getSize().y / 2) - (_birdSprite.getGlobalBounds().height
/ 2));

        _birdState = BIRD_STATE_STILL;

        sf::Vector2f origin =
sf::Vector2f(_birdSprite.getGlobalBounds().width / 2,
_birdSprite.getGlobalBounds().height / 2);

        _birdSprite.setOrigin(origin);

        _rotation = 0;
    }

    Bird::~Bird()
    {
    }

    void Bird::Draw()
    {
        _data->window.draw(_birdSprite);
    }

    void Bird::Animate(float dt)
    {
```

```
        if (_clock.getElapsedTime().asSeconds() >
BIRD_ANIMATION_DURATION / _animationFrames.size())
        {
            if (_animationIterator < _animationFrames.size() -
1)
            {
                _animationIterator++;
            }
            else
            {
                _animationIterator = 0;
            }


_birdSprite.setTexture(_animationFrames.at(_animationIterator));

            _clock.restart();
        }
    }

    void Bird::Update(float dt)
    {
        if (BIRD_STATE_FALLING == _birdState)
        {
            _birdSprite.move(0, GRAVITY * dt);

            _rotation += ROTATION_SPEED * dt;

            if (_rotation > 25.0f)
            {
                _rotation = 25.0f;
            }

            _birdSprite.setRotation(_rotation);
        }
        else if (BIRD_STATE_FLYING == _birdState)
        {
            _birdSprite.move(0, -FLYING_SPEED * dt);

            _rotation -= ROTATION_SPEED * dt;

            if (_rotation < -25.0f)
            {
                _rotation = -25.0f;
            }

            _birdSprite.setRotation(_rotation);
        }

        if (_movementClock.getElapsedTime().asSeconds() >
FLYING_DURATION)
        {
```

```
            _movementClock.restart();
            _birdState = BIRD_STATE_FALLING;
        }
    }

    void Bird::Tap()
    {
        _movementClock.restart();
        _birdState = BIRD_STATE_FLYING;
    }

    const sf::Sprite &Bird::GetSprite() const
    {
        return _birdSprite;
    }
```

Файл Collision.h:

```
#pragma once

#include <SFML/Graphics.hpp>

class Collision
    {
    public:
        Collision();
        ~Collision();

        bool CheckSpriteCollision(sf::Sprite sprite1, sf::Sprite
sprite2);
        bool CheckSpriteCollision(sf::Sprite sprite1, float
scale1, sf::Sprite sprite2, float scale2);

    };
```

Файл Collision.cpp:

```
#include "Collision.h"
#include <iostream>

    Collision::Collision()
    {
    }

    Collision::~Collision()
    {
    }

    bool Collision::CheckSpriteCollision(sf::Sprite sprite1,
sf::Sprite sprite2)
    {
        sf::Rect<float> rect1 = sprite1.getGlobalBounds();
        sf::Rect<float> rect2 = sprite2.getGlobalBounds();
```

```
        if (rect1.intersects(rect2))
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    bool Collision::CheckSpriteCollision(sf::Sprite sprite1,
float scale1, sf::Sprite sprite2, float scale2)
    {
        sprite1.setScale(scale1, scale1);
        sprite2.setScale(scale2, scale2);

        sf::Rect<float> rect1 = sprite1.getGlobalBounds();
        sf::Rect<float> rect2 = sprite2.getGlobalBounds();

        if (rect1.intersects(rect2))
        {
            return true;
        }
        else
        {
            return false;
        }
    }
```

Файл DEFINITIONS.h:

```
#pragma once

#define SCREEN_WIDTH 768
#define SCREEN_HEIGHT 1024

#define SPLASH_STATE_SHOW_TIME 0.0

#define SPLASH_SCENE_BACKGROUND_FILEPATH "Resources/res/Splash
Background.png"
#define MAIN_MENU_BACKGROUND_FILEPATH "Resources/res/sky.png"
#define GAME_BACKGROUND_FILEPATH "Resources/res/sky.png"
#define GAME_OVER_BACKGROUND_FILEPATH "Resources/res/sky.png"

#define GAME_TITLE_FILEPATH "Resources/res/title.png"
#define PLAY_BUTTON_FILEPATH "Resources/res/PlayButton.png"

#define PIPE_UP_FILEPATH "Resources/res/PipeUp.png"
#define PIPE_DOWN_FILEPATH "Resources/res/PipeDown.png"

#define LAND_FILEPATH "Resources/res/land.png"

#define BIRD_FRAME_1_FILEPATH "Resources/res/bird-01.png"
```

```
#define BIRD_FRAME_2_FILEPATH "Resources/res/bird-02.png"
#define BIRD_FRAME_3_FILEPATH "Resources/res/bird-03.png"
#define BIRD_FRAME_4_FILEPATH "Resources/res/bird-04.png"

#define SCORING_PIPE_FILEPATH
"Resources/res/InvisibleScoringPipe.png"

#define FLAPPY_FONT_FILEPATH "Resources/fonts/FlappyFont.ttf"

#define GAME_OVER_TITLE_FILEPATH "Resources/res/Game-Over-
Title.png"
#define GAME_OVER_BODY_FILEPATH "Resources/res/Game-Over-
Body.png"

#define BRONZE_MEDAL_FILEPATH "Resources/res/Bronze-Medal.png"
#define SILVER_MEDAL_FILEPATH "Resources/res/Silver-Medal.png"
#define GOLD_MEDAL_FILEPATH "Resources/res/Gold-Medal.png"
#define PLATINUM_MEDAL_FILEPATH "Resources/res/Platinum-
Medal.png"

#define HIT_SOUND_FILEPATH "Resources/audio/Hit.wav"
#define POINT_SOUND_FILEPATH "Resources/audio/Point.wav"
#define WING_SOUND_FILEPATH "Resources/audio/Wing.wav"

#define PIPE_MOVEMENT_SPEED 200.0f
#define PIPE_SPAWN_FREQUENCY 1.5f

#define BIRD_ANIMATION_DURATION 0.95f

#define BIRD_STATE_STILL 1
#define BIRD_STATE_FALLING 2
#define BIRD_STATE_FLYING 3

#define GRAVITY 400.0f
#define FLYING_SPEED 550.0f

#define FLYING_DURATION 0.35f

#define ROTATION_SPEED 400.0f

enum GameStates
{
    eReady,
    ePlaying,
    eGameOver
};

#define FLASH_SPEED 1500.0f

#define TIME_BEFORE_GAME_OVER_APPEARS 1.5f

#define BRONZE_MEDAL_SCORE 0
```

```
#define SILVER_MEDAL_SCORE 5
#define GOLD_MEDAL_SCORE 25
#define PLATINUM_MEDAL_SCORE 100
```

Файл Flash.h:

```
#pragma once

#include <SFML/Graphics.hpp>
#include "Game.h"
#include "DEFINITIONS.h"

    class Flash
    {
    public:
        Flash(GameDataRef data);
        ~Flash();

        void Show(float dt);
        void Draw();

    private:
        GameDataRef _data;

        sf::RectangleShape _shape;

        bool _flashOn;

    };
```

Файл Flash.cpp:

```
#include "Flash.h"

    Flash::Flash(GameDataRef data) : _data(data)
    {
        _shape = sf::RectangleShape(sf::Vector2f(_data-
>window.getSize().x, _data->window.getSize().y));
        _shape.setFillColor(sf::Color(255, 255, 255, 0));

        _flashOn = true;
    }

    Flash::~Flash()
    {
    }

    void Flash::Show(float dt)
    {
        if (_flashOn)
        {
            int alpha = (int)_shape.getFillColor().a +
(FLASH_SPEED * dt);
```

```
            if (alpha >= 255.0f)
            {
                alpha = 255.0f;
                _flashOn = false;
            }

            _shape.setFillColor(sf::Color(255, 255, 255,
alpha));
        }
        else
        {
            int alpha = (int)_shape.getFillColor().a -
(FLASH_SPEED * dt);

            if (alpha <= 0.0f)
            {
                alpha = 0.0f;
            }

            _shape.setFillColor(sf::Color(255, 255, 255,
alpha));
        }
    }

    void Flash::Draw()
    {
        _data->window.draw(_shape);
    }
```

Файл Game.h:

```
#pragma once

#include <memory>
#include <string>
#include <SFML/Graphics.hpp>
#include "StateMachine.h"
#include "AssetManager.h"
#include "InputManager.h"

    struct GameData
    {
        StateMachine machine;
        sf::RenderWindow window;
        AssetManager assets;
        InputManager input;
    };

    typedef std::shared_ptr<GameData> GameDataRef;

    class Game
    {
    public:
```

```
        Game(int width, int height, std::string title);

    private:
        // Updates run at 60 per second.
        const float dt = 1.0f / 60.0f;
        sf::Clock _clock;

        GameDataRef _data = std::make_shared<GameData>();

        void Run();
    };
```

Файл Game.cpp:

```
#include "Game.h"
#include "SplashState.h"

#include <stdlib.h>
#include <time.h>

    Game::Game(int width, int height, std::string title)
    {
        srand(time(NULL));

        _data->window.create(sf::VideoMode(width, height),
title, sf::Style::Close | sf::Style::Titlebar);
        _data->machine.AddState(StateRef(new SplashState(this-
>_data)));

        this->Run();
    }

    void Game::Run()
    {
        float newTime, frameTime, interpolation;

        float currentTime = this-
>_clock.getElapsedTime().asSeconds();
        float accumulator = 0.0f;

        while (this->_data->window.isOpen())
        {
            this->_data->machine.ProcessStateChanges();

            newTime = this->_clock.getElapsedTime().asSeconds();
            frameTime = newTime - currentTime;

            if (frameTime > 0.25f)
            {
                frameTime = 0.25f;
            }

            currentTime = newTime;
```

```
            accumulator += frameTime;

            while (accumulator >= dt)
            {
                this->_data->machine.GetActiveState()-
>HandleInput();
                this->_data->machine.GetActiveState()-
>Update(dt);

                accumulator -= dt;
            }

            interpolation = accumulator / dt;
            this->_data->machine.GetActiveState()-
>Draw(interpolation);
        }
    }
```

Файл GameOverState.h:

```
#pragma once

#include <SFML/Graphics.hpp>
#include "State.h"
#include "Game.h"

    class GameOverState : public State
    {
    public:
        GameOverState(GameDataRef data, int score);

        void Init();

        void HandleInput();
        void Update(float dt);
        void Draw(float dt);

    private:
        GameDataRef _data;

        sf::Sprite _background;

        sf::Sprite _gameOverTitle;
        sf::Sprite _gameOverContainer;
        sf::Sprite _retryButton;
        sf::Sprite _medal;

        sf::Text _scoreText;
        sf::Text _highScoreText;

        int _score;
        int _highScore;
```

```
    };
```

Файл GameOverState.cpp:

```
#include <sstream>
#include "DEFINITIONS.h"
#include "GameOverState.h"
#include "GameState.h"

#include <iostream>
#include <fstream>

    GameOverState::GameOverState(GameDataRef data, int score) :
_data(data), _score(score)
    {

    }

    void GameOverState::Init()
    {
        std::ifstream readFile;
        readFile.open( "Resources/Highscore.txt" );

        if ( readFile.is_open( ) )
        {
            while ( !readFile.eof( ) )
            {
                readFile >> _highScore;
            }
        }

        readFile.close( );

        std::ofstream writeFile( "Resources/Highscore.txt" );

        if ( writeFile.is_open( ) )
        {
            if ( _score > _highScore )
            {
                _highScore = _score;
            }

            writeFile << _highScore;
        }

        writeFile.close( );

        this->_data->assets.LoadTexture("Game Over Background",
GAME_OVER_BACKGROUND_FILEPATH);
        this->_data->assets.LoadTexture("Game Over Title",
GAME_OVER_TITLE_FILEPATH);
        this->_data->assets.LoadTexture("Game Over Body",
GAME_OVER_BODY_FILEPATH);
```

```
        this->_data->assets.LoadTexture("Bronze Medal",
BRONZE_MEDAL_FILEPATH);
        this->_data->assets.LoadTexture("Silver Medal",
SILVER_MEDAL_FILEPATH);
        this->_data->assets.LoadTexture("Gold Medal",
GOLD_MEDAL_FILEPATH);
        this->_data->assets.LoadTexture("Platinum Medal",
PLATINUM_MEDAL_FILEPATH);

        _background.setTexture(this->_data-
>assets.GetTexture("Game Over Background"));
        _gameOverTitle.setTexture(this->_data-
>assets.GetTexture("Game Over Title"));
        _gameOverContainer.setTexture(this->_data-
>assets.GetTexture("Game Over Body"));
        _retryButton.setTexture(this->_data-
>assets.GetTexture("Play Button"));

        _gameOverContainer.setPosition(sf::Vector2f((_data-
>window.getSize().x / 2) -
(_gameOverContainer.getGlobalBounds().width / 2), (_data-
>window.getSize().y / 2) -
(_gameOverContainer.getGlobalBounds().height / 2)));
        _gameOverTitle.setPosition(sf::Vector2f((_data-
>window.getSize().x / 2) -
(_gameOverTitle.getGlobalBounds().width / 2),
_gameOverContainer.getPosition().y -
(_gameOverTitle.getGlobalBounds().height * 1.2)));
        _retryButton.setPosition(sf::Vector2f((_data-
>window.getSize().x / 2) - (_retryButton.getGlobalBounds().width
/ 2), _gameOverContainer.getPosition().y +
_gameOverContainer.getGlobalBounds().height +
(_retryButton.getGlobalBounds().height * 0.2)));

        _scoreText.setFont(this->_data->assets.GetFont("Flappy
Font"));
        _scoreText.setString(std::to_string(_score));
        _scoreText.setCharacterSize(56);
        _scoreText.setFillColor(sf::Color::White);

_scoreText.setOrigin(sf::Vector2f(_scoreText.getGlobalBounds().w
idth / 2, _scoreText.getGlobalBounds().height / 2));
        _scoreText.setPosition(sf::Vector2f(_data-
>window.getSize().x / 10 * 7.25, _data->window.getSize().y /
2.15));

        _highScoreText.setFont(this->_data-
>assets.GetFont("Flappy Font"));
        _highScoreText.setString(std::to_string(_highScore));
        _highScoreText.setCharacterSize(56);
        _highScoreText.setFillColor(sf::Color::White);
```

```
_highScoreText.setOrigin(sf::Vector2f(_highScoreText.getGlobalBo
unds().width / 2, _highScoreText.getGlobalBounds().height / 2));
        _highScoreText.setPosition(sf::Vector2f(_data-
>window.getSize().x / 10 * 7.25, _data->window.getSize().y /
1.78));

        if ( _score >= PLATINUM_MEDAL_SCORE )
        {
            _medal.setTexture( _data->assets.GetTexture(
"Platinum Medal" ) );
        }
        else if ( _score >= GOLD_MEDAL_SCORE )
        {
            _medal.setTexture( _data->assets.GetTexture( "Gold
Medal" ) );
        }
        else if ( _score >= SILVER_MEDAL_SCORE )
        {
            _medal.setTexture( _data->assets.GetTexture( "Silver
Medal" ) );
        }
        else
        {
            _medal.setTexture( _data->assets.GetTexture( "Bronze
Medal" ) );
        }

        _medal.setPosition( 175, 465 );
    }

    void GameOverState::HandleInput()
    {
        sf::Event event;

        while (this->_data->window.pollEvent(event))
        {
            if (sf::Event::Closed == event.type)
            {
                this->_data->window.close();
            }

            if (this->_data->input.IsSpriteClicked(this-
>_retryButton, sf::Mouse::Left, this->_data->window))
            {
                this->_data->machine.AddState(StateRef(new
GameState(_data)), true);
            }
        }
    }

    void GameOverState::Update(float dt)
    {
```

```
    }

    void GameOverState::Draw(float dt)
    {
        this->_data->window.clear(sf::Color::Red);

        this->_data->window.draw(this->_background);

        _data->window.draw(_gameOverTitle);
        _data->window.draw(_gameOverContainer);
        _data->window.draw(_retryButton);
        _data->window.draw(_scoreText);
        _data->window.draw(_highScoreText);

        _data->window.draw( _medal );

        this->_data->window.display();
    }
```

Файл GameState.h:

```
#pragma once

#include <SFML/Graphics.hpp>
#include <SFML/Audio.hpp>

#include "State.h"
#include "Game.h"
#include "Pipe.h"
#include "Land.h"
#include "Bird.h"
#include "Collision.h"
#include "Flash.h"
#include "HUD.h"

    class GameState : public State
    {
    public:
        GameState(GameDataRef data);

        void Init();

        void HandleInput();
        void Update(float dt);
        void Draw(float dt);

    private:
        GameDataRef _data;

        sf::Sprite _background;

        Pipe *pipe;
```

```
        Land *land;
        Bird *bird;
        Collision collision;
        Flash *flash;
        HUD *hud;

        sf::Clock clock;

        int _gameState;

        sf::RectangleShape _gameOverFlash;
        bool _flashOn;

        int _score;

        sf::SoundBuffer _hitSoundBuffer;
        sf::SoundBuffer _wingSoundBuffer;
        sf::SoundBuffer _pointSoundBuffer;

        sf::Sound _hitSound;
        sf::Sound _wingSound;
        sf::Sound _pointSound;

    };
```

Файл GameState.cpp:

```
#pragma once

#include <sstream>
#include "DEFINITIONS.h"
#include "GameState.h"
#include "GameOverState.h"

#include <iostream>

    GameState::GameState(GameDataRef data) : _data(data)
    {

    }

    void GameState::Init()
    {
        if (!_hitSoundBuffer.loadFromFile(HIT_SOUND_FILEPATH))
        {
            std::cout << "Error Loading Hit Sound Effect" <<
std::endl;
        }

        if (!_wingSoundBuffer.loadFromFile(WING_SOUND_FILEPATH))
        {
            std::cout << "Error Loading Wing Sound Effect" <<
std::endl;
```

```
        }

        if
(!_pointSoundBuffer.loadFromFile(POINT_SOUND_FILEPATH))
        {
            std::cout << "Error Loading Point Sound Effect" <<
std::endl;
        }

        _hitSound.setBuffer(_hitSoundBuffer);
        _wingSound.setBuffer(_wingSoundBuffer);
        _pointSound.setBuffer(_pointSoundBuffer);

        this->_data->assets.LoadTexture("Game Background",
GAME_BACKGROUND_FILEPATH);
        this->_data->assets.LoadTexture("Pipe Up",
PIPE_UP_FILEPATH);
        this->_data->assets.LoadTexture("Pipe Down",
PIPE_DOWN_FILEPATH);
        this->_data->assets.LoadTexture("Land", LAND_FILEPATH);
        this->_data->assets.LoadTexture("Bird Frame 1",
BIRD_FRAME_1_FILEPATH);
        this->_data->assets.LoadTexture("Bird Frame 2",
BIRD_FRAME_2_FILEPATH);
        this->_data->assets.LoadTexture("Bird Frame 3",
BIRD_FRAME_3_FILEPATH);
        this->_data->assets.LoadTexture("Bird Frame 4",
BIRD_FRAME_4_FILEPATH);
        this->_data->assets.LoadTexture("Scoring Pipe",
SCORING_PIPE_FILEPATH);
        this->_data->assets.LoadFont("Flappy Font",
FLAPPY_FONT_FILEPATH);

        pipe = new Pipe(_data);
        land = new Land(_data);
        bird = new Bird(_data);
        flash = new Flash(_data);
        hud = new HUD(_data);

        _background.setTexture(this->_data-
>assets.GetTexture("Game Background"));

        _score = 0;
        hud->UpdateScore(_score);

        _gameState = GameStates::eReady;
    }

    void GameState::HandleInput()
    {
        sf::Event event;
```

```
        while (this->_data->window.pollEvent(event))
        {
            if (sf::Event::Closed == event.type)
            {
                this->_data->window.close();
            }

            if (this->_data->input.IsSpriteClicked(this-
>_background, sf::Mouse::Left, this->_data->window))
            {
                if (GameStates::eGameOver != _gameState)
                {
                    _gameState = GameStates::ePlaying;
                    bird->Tap();

                    _wingSound.play();
                }
            }
        }
    }

    void GameState::Update(float dt)
    {
        if (GameStates::eGameOver != _gameState)
        {
            bird->Animate(dt);
            land->MoveLand(dt);
        }

        if (GameStates::ePlaying == _gameState)
        {
            pipe->MovePipes(dt);

            if (clock.getElapsedTime().asSeconds() >
PIPE_SPAWN_FREQUENCY)
            {
                pipe->RandomisePipeOffset();

                pipe->SpawnInvisiblePipe();
                pipe->SpawnBottomPipe();
                pipe->SpawnTopPipe();
                pipe->SpawnScoringPipe();

                clock.restart();
            }

            bird->Update(dt);

            std::vector<sf::Sprite> landSprites = land-
>GetSprites();

            for (int i = 0; i < landSprites.size(); i++)
```

```
            {
                if (collision.CheckSpriteCollision(bird-
>GetSprite(), 0.7f, landSprites.at(i), 1.0f))
                {
                    _gameState = GameStates::eGameOver;

                    clock.restart();

                    _hitSound.play();
                }
            }

            std::vector<sf::Sprite> pipeSprites = pipe-
>GetSprites();

            for (int i = 0; i < pipeSprites.size(); i++)
            {
                if (collision.CheckSpriteCollision(bird-
>GetSprite(), 0.625f, pipeSprites.at(i), 1.0f))
                {
                    _gameState = GameStates::eGameOver;

                    clock.restart();

                    _hitSound.play();
                }
            }

            if (GameStates::ePlaying == _gameState)
            {
                std::vector<sf::Sprite> &scoringSprites = pipe-
>GetScoringSprites();

                for (int i = 0; i < scoringSprites.size(); i++)
                {
                    if (collision.CheckSpriteCollision(bird-
>GetSprite(), 0.625f, scoringSprites.at(i), 1.0f))
                    {
                        _score++;

                        hud->UpdateScore(_score);


scoringSprites.erase(scoringSprites.begin() + i);

                        _pointSound.play();
                    }
                }
            }
        }

        if (GameStates::eGameOver == _gameState)
```

```
        {
            flash->Show(dt);

            if (clock.getElapsedTime().asSeconds() >
TIME_BEFORE_GAME_OVER_APPEARS)
            {
                this->_data->machine.AddState(StateRef(new
GameOverState(_data, _score)), true);
            }
        }
    }

    void GameState::Draw(float dt)
    {
        this->_data->window.clear( sf::Color::Red );

        this->_data->window.draw(this->_background);

        pipe->DrawPipes();
        land->DrawLand();
        bird->Draw();

        flash->Draw();

        hud->Draw();

        this->_data->window.display();
    }
}
```

Файл HUD.h:

```
#pragma once

#include <SFML/Graphics.hpp>

#include "DEFINITIONS.h"
#include "Game.h"

    class HUD
    {
    public:
        HUD(GameDataRef data);
        ~HUD();

        void Draw();
        void UpdateScore(int score);

    private:
        GameDataRef _data;

        sf::Text _scoreText;
```

```
    };
```

Файл HUD.cpp:

```
#include "HUD.h"

#include <string>

    HUD::HUD(GameDataRef data) : _data(data)
    {
        _scoreText.setFont(this->_data->assets.GetFont("Flappy
Font"));

        _scoreText.setString("0");

        _scoreText.setCharacterSize(128);

        _scoreText.setFillColor(sf::Color::White);


_scoreText.setOrigin(sf::Vector2f(_scoreText.getGlobalBounds().w
idth / 2, _scoreText.getGlobalBounds().height / 2));

        _scoreText.setPosition(sf::Vector2f(_data-
>window.getSize().x / 2, _data->window.getSize().y / 5));
    }

    HUD::~HUD()
    {
    }

    void HUD::Draw()
    {
        _data->window.draw(_scoreText);
    }

    void HUD::UpdateScore(int score)
    {
        _scoreText.setString(std::to_string(score));
    }
```

Файл InputManager.h:

```
#pragma once

#include <SFML\Graphics.hpp>

    class InputManager
    {
    public:
        InputManager() {}
        ~InputManager() {}

        bool IsSpriteClicked(sf::Sprite object,
```

```
sf::Mouse::Button button, sf::RenderWindow &window);

        sf::Vector2i GetMousePosition(sf::RenderWindow &window);
    };
```

Файл InputManager.cpp:

```
#pragma once

#include "InputManager.h"

    bool InputManager::IsSpriteClicked(sf::Sprite object,
sf::Mouse::Button button, sf::RenderWindow &window)
    {
        if (sf::Mouse::isButtonPressed(button))
        {
            sf::IntRect playButtonRect(object.getPosition().x,
object.getPosition().y, object.getGlobalBounds().width,
object.getGlobalBounds().height);

            if
(playButtonRect.contains(sf::Mouse::getPosition(window)))
            {
                return true;
            }
        }

        return false;
    }

    sf::Vector2i InputManager::GetMousePosition(sf::RenderWindow
&window)
    {
        return sf::Mouse::getPosition(window);
    }
```

Файл Land.h:

```
#pragma once

#include <SFML/Graphics.hpp>
#include "Game.h"
#include <vector>

    class Land
    {
    public:
        Land(GameDataRef data);

        void MoveLand(float dt);
        void DrawLand();

        const std::vector<sf::Sprite> &GetSprites() const;
```

```
    private:
        GameDataRef _data;

        std::vector<sf::Sprite>_landSprites;

    };
```

Файл Land.cpp:

```
#include "Land.h"
#include "DEFINITIONS.h"

    Land::Land(GameDataRef data) : _data(data)
    {
        sf::Sprite sprite(this->_data-
>assets.GetTexture("Land"));
        sf::Sprite sprite2(this->_data-
>assets.GetTexture("Land"));

        sprite.setPosition(0, this->_data->window.getSize().y -
sprite.getLocalBounds().height);
        sprite2.setPosition(sprite.getLocalBounds().width, this-
>_data->window.getSize().y - sprite2.getLocalBounds().height);

        _landSprites.push_back(sprite);
        _landSprites.push_back(sprite2);
    }

    void Land::MoveLand(float dt)
    {
        for (unsigned short int i = 0; i < _landSprites.size();
i++)
        {
            sf::Vector2f position =
_landSprites.at(i).getPosition();
            float movement = PIPE_MOVEMENT_SPEED * dt;

            _landSprites.at(i).move(-movement, 0.0f);

            if (_landSprites.at(i).getPosition().x < 0 -
_landSprites.at(i).getLocalBounds().width)
            {
                sf::Vector2f position(_data->window.getSize().x,
_landSprites.at(i).getPosition().y);

                _landSprites.at(i).setPosition(position);
            }
        }
    }

    void Land::DrawLand()
    {
        for (unsigned short int i = 0; i < _landSprites.size();
```

```
i++)
        {
            this->_data->window.draw(_landSprites.at(i));
        }
    }

    const std::vector<sf::Sprite> &Land::GetSprites() const
    {
        return _landSprites;
    }
```

Файл main.cpp:

```
#include "Game.h"
#include "DEFINITIONS.h"

int main()
{
    Game(SCREEN_WIDTH, SCREEN_HEIGHT, "Flappy Bird");

    return EXIT_SUCCESS;
}
```

Файл MainMenuState.h:

```
#pragma once

#include <SFML/Graphics.hpp>
#include "State.h"
#include "Game.h"

    class MainMenuState : public State
    {
    public:
        MainMenuState(GameDataRef data);

        void Init();

        void HandleInput();
        void Update(float dt);
        void Draw(float dt);

    private:
        GameDataRef _data;

        sf::Sprite _background;
        sf::Sprite _title;
        sf::Sprite _playButton;
    };
```

Файл MainMenuState.cpp:

```
#pragma once

#include <sstream>
```

```
#include "DEFINITIONS.h"
#include "MainMenuState.h"
#include "GameState.h"

#include <iostream>

    MainMenuState::MainMenuState(GameDataRef data) : _data(data)
    {

    }

    void MainMenuState::Init()
    {
        this->_data->assets.LoadTexture("Main Menu Background",
MAIN_MENU_BACKGROUND_FILEPATH);
        this->_data->assets.LoadTexture("Game Title",
GAME_TITLE_FILEPATH);
        this->_data->assets.LoadTexture("Play Button",
PLAY_BUTTON_FILEPATH);

        _background.setTexture(this->_data-
>assets.GetTexture("Main Menu Background"));
        _title.setTexture(this->_data->assets.GetTexture("Game
Title"));
        _playButton.setTexture(this->_data-
>assets.GetTexture("Play Button"));

        _title.setPosition((SCREEN_WIDTH / 2) -
(_title.getGlobalBounds().width / 2),
_title.getGlobalBounds().height / 2);
        _playButton.setPosition((SCREEN_WIDTH / 2) -
(_playButton.getGlobalBounds().width / 2), (SCREEN_HEIGHT / 2) -
(_playButton.getGlobalBounds().height / 2));
    }

    void MainMenuState::HandleInput()
    {
        sf::Event event;

        while (this->_data->window.pollEvent(event))
        {
            if (sf::Event::Closed == event.type)
            {
                this->_data->window.close();
            }

            if (this->_data->input.IsSpriteClicked(this-
>_playButton, sf::Mouse::Left, this->_data->window))
            {
                // Switch To Main Menu
                this->_data->machine.AddState(StateRef(new
GameState(_data)), true);
```

```
            }
        }
    }

    void MainMenuState::Update(float dt)
    {

    }

    void MainMenuState::Draw(float dt)
    {
        this->_data->window.clear(sf::Color::Red);

        this->_data->window.draw(this->_background);
        this->_data->window.draw(this->_title);
        this->_data->window.draw(this->_playButton);

        this->_data->window.display();
    }
```

Файл Pipe.h:

```
#pragma once

#include <SFML/Graphics.hpp>
#include "Game.h"
#include <vector>

    class Pipe
    {
    public:
        Pipe(GameDataRef data);

        void SpawnBottomPipe();
        void SpawnTopPipe();
        void SpawnInvisiblePipe();
        void SpawnScoringPipe();
        void MovePipes(float dt);
        void DrawPipes();
        void RandomisePipeOffset();

        const std::vector<sf::Sprite> &GetSprites() const;
        std::vector<sf::Sprite> &GetScoringSprites();

    private:
        GameDataRef _data;
        std::vector<sf::Sprite> pipeSprites;
        std::vector<sf::Sprite> scoringPipes;

        int _landHeight;
        int _pipeSpawnYOffset;
```

```
    };
```

Файл Pipe.cpp:

```
#include "Pipe.h"
#include "DEFINITIONS.h"

#include <iostream>

    Pipe::Pipe(GameDataRef data) : _data(data)
    {
        _landHeight = this->_data-
>assets.GetTexture("Land").getSize().y;
        _pipeSpawnYOffset = 0;
    }

    void Pipe::SpawnBottomPipe()
    {
        sf::Sprite sprite(this->_data->assets.GetTexture("Pipe
Up"));

        sprite.setPosition(this->_data->window.getSize().x,
this->_data->window.getSize().y - sprite.getLocalBounds().height
- _pipeSpawnYOffset);

        pipeSprites.push_back(sprite);
    }

    void Pipe::SpawnTopPipe()
    {
        sf::Sprite sprite(this->_data->assets.GetTexture("Pipe
Down"));

        sprite.setPosition(this->_data->window.getSize().x, -
_pipeSpawnYOffset);

        pipeSprites.push_back(sprite);
    }

    void Pipe::SpawnInvisiblePipe()
    {
        sf::Sprite sprite(this->_data->assets.GetTexture("Pipe
Down"));

        sprite.setPosition(this->_data->window.getSize().x, -
_pipeSpawnYOffset);
        sprite.setColor(sf::Color(0, 0, 0, 0));

        pipeSprites.push_back(sprite);
    }

    void Pipe::SpawnScoringPipe()
    {
```

```
        sf::Sprite sprite(this->_data-
>assets.GetTexture("Scoring Pipe"));

        sprite.setPosition(this->_data->window.getSize().x, 0);

        scoringPipes.push_back(sprite);
    }

    void Pipe::MovePipes(float dt)
    {
        for ( int i = 0; i < pipeSprites.size(); i++)
        {
            if (pipeSprites.at(i).getPosition().x < 0 -
pipeSprites.at(i).getLocalBounds().width)
            {
                pipeSprites.erase( pipeSprites.begin( ) + i );
            }
            else
            {
                sf::Vector2f position =
pipeSprites.at(i).getPosition();
                float movement = PIPE_MOVEMENT_SPEED * dt;

                pipeSprites.at(i).move(-movement, 0);
            }
        }

        for (int i = 0; i < scoringPipes.size(); i++)
        {
            if (scoringPipes.at(i).getPosition().x < 0 -
scoringPipes.at(i).getLocalBounds().width)
            {
                scoringPipes.erase(scoringPipes.begin() + i);
            }
            else
            {
                sf::Vector2f position =
scoringPipes.at(i).getPosition();
                float movement = PIPE_MOVEMENT_SPEED * dt;

                scoringPipes.at(i).move(-movement, 0);
            }
        }
    }

    void Pipe::DrawPipes()
    {
        for (unsigned short int i = 0; i < pipeSprites.size();
i++)
        {
            this->_data->window.draw(pipeSprites.at(i));
        }
```

```
    }

    void Pipe::RandomisePipeOffset()
    {
        _pipeSpawnYOffset = rand() % (_landHeight + 1);
    }

    const std::vector<sf::Sprite> &Pipe::GetSprites() const
    {
        return pipeSprites;
    }

    std::vector<sf::Sprite> &Pipe::GetScoringSprites()
    {
        return scoringPipes;
    }
```

Файл SplashState.h:

```
#pragma once

#include <SFML/Graphics.hpp>
#include "State.h"
#include "Game.h"

    class SplashState : public State
    {
    public:
        SplashState(GameDataRef data);

        void Init();

        void HandleInput();
        void Update(float dt);
        void Draw(float dt);

    private:
        GameDataRef _data;

        sf::Clock _clock;

        sf::Sprite _background;
    };
```

Файл SplashState.cpp:

```
#pragma once

#include <sstream>
#include "SplashState.h"
#include "DEFINITIONS.h"
#include "MainMenuState.h"

#include <iostream>
```

```
    SplashState::SplashState(GameDataRef data) : _data(data)
    {

    }

    void SplashState::Init()
    {
        this->_data->assets.LoadTexture("Splash State
Background", SPLASH_SCENE_BACKGROUND_FILEPATH);

        _background.setTexture(this->_data-
>assets.GetTexture("Splash State Background"));
    }

    void SplashState::HandleInput()
    {
        sf::Event event;

        while (this->_data->window.pollEvent(event))
        {
            if (sf::Event::Closed == event.type)
            {
                this->_data->window.close();
            }
        }
    }

    void SplashState::Update(float dt)
    {
        if (this->_clock.getElapsedTime().asSeconds() >
SPLASH_STATE_SHOW_TIME)
        {
            // Switch To Main Menu
            this->_data->machine.AddState(StateRef(new
MainMenuState(_data)), true);
        }
    }

    void SplashState::Draw(float dt)
    {
        this->_data->window.clear(sf::Color::Red);

        this->_data->window.draw( this->_background );

        this->_data->window.display();
    }
```

Файл State.h:

```
#pragma once

    class State
    {
```

```
    public:
        virtual void Init() = 0;

        virtual void HandleInput() = 0;
        virtual void Update(float dt) = 0;
        virtual void Draw(float dt) = 0;

        virtual void Pause() { }
        virtual void Resume() { }
    };
```

Файл StateMachine.h:

```
#pragma once

#include <memory>
#include <stack>

#include "State.h"

    typedef std::unique_ptr<State> StateRef;

    class StateMachine
    {
    public:
        StateMachine() { }
        ~StateMachine() { }

        void AddState(StateRef newState, bool isReplacing =
true);

        void RemoveState();
        // Run at start of each loop in Game.cpp
        void ProcessStateChanges();

        StateRef &GetActiveState();

    private:
        std::stack<StateRef> _states;
        StateRef _newState;

        bool _isRemoving;
        bool _isAdding, _isReplacing;
    };
```

Файл StateMachine.cpp:

```
#include "StateMachine.h"

    void StateMachine::AddState(StateRef newState, bool
isReplacing)
    {
        this->_isAdding = true;
        this->_isReplacing = isReplacing;
```

```
    this->_newState = std::move(newState);
}

void StateMachine::RemoveState()
{
    this->_isRemoving = true;
}

void StateMachine::ProcessStateChanges()
{
    if (this->_isRemoving && !this->_states.empty())
    {
        this->_states.pop();

        if (!this->_states.empty())
        {
            this->_states.top()->Resume();
        }

        this->_isRemoving = false;
    }

    if (this->_isAdding)
    {
        if (!this->_states.empty())
        {
            if (this->_isReplacing)
            {
                this->_states.pop();
            }
            else
            {
                this->_states.top()->Pause();
            }
        }

        this->_states.push(std::move(this->_newState));
        this->_states.top()->Init();
        this->_isAdding = false;
    }
}

StateRef &StateMachine::GetActiveState()
{
    return this->_states.top();
}
```

ПРИЛОЖЕНИЕ Д
(обязательное)

Ведомость документов