**СОДЕРЖАНИЕ**

ВВЕДЕНИЕ 4

1 ТЕХНИЧЕСКИЕ СРЕДСТВА РАЗРАБОТКИ ИГРОВЫХ ПРИЛОЖЕНИЙ 5

1.1 Графическая библиотека DirectX 5

1.2 Графическая библиотека OpenGL 6

1.3 Отличия графических библиотек OpenGL и DirectX 8

1.4 Игровой жанр «Платформер» 9

1.5 Обзор существующих игр жанра «Платформер» 9

1.6 Технические требования к курсовому проекту 14

2 АРХИТЕКТУРА ИГРОВОГО ПРИЛОЖЕНИЯ «ПЛАТФОРМЕР» 15

2.1 Архитектура игрового приложения 15

2.2 Структура игрового приложения «Платформер» 16

2.3 Обзор взаимодействия классов игрового приложения 19

3 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ ПРИЛОЖЕНИЯ «ПЛАТФОРМЕР» 23

3.1 Алгоритмическая реализация игрового приложения 23

3.2 Тестирование игрового приложения 24

3.3 Опытная эксплуатация игрового приложения 25

ЗАКЛЮЧЕНИЕ 32

Список используемых источников 33

Приложение А Листинг программы «Платформер» 34

Приложение Б Руководство пользователя 120

Приложение В Руководство программиста 122

Приложение Г Руководство системного программиста 123

Приложение Д Внешний вид окон интерфейса программы 123

Приложение Е Диаграмма вариантов использования приложения 123

## ВВЕДЕНИЕ

Актуальность игровых приложений уже не требует никаких доказательств, а их популярность растет с каждым днём. Едва ли в современном мире найдется хоть один человек, который не знал бы об этом чуде технического прогресса.

Индустрия компьютерных игр – одна из наиболее востребованных и быстро развивающихся отраслей компьютерных технологий. Разумеется, что с возникновением новых технологий, творческий потенциал игры становится возможным реализовать. Таким образом, с каждым годом на рынок выходит всё больше креативных и реалистичных видеоигр.

В основном, игровая индустрия считается развлекательной сферой, однако её развитие поспособствовало появлению важных симуляторов для обучения профессиям, спасающим жизни людей.

Казалось, что компьютерные игры всегда присутствовали в жизни людей, и трудно представить, что они появились только около 50-ти лет назад.

Первые игры считались уникальными, неповторимыми, их было достаточно просто отличить от других. Но со временем стало появляться много аналогов и схожих видеоигр, поэтому их принято классифицировать.

Существует большое количество подходов к классификации игр: по используемой платформе, по визуальному представлению, по количеству игроков, множество авторских подходов, а также по жанру.

Жанровая классификация игр стала активно развиваться в 90-е годы с появлением целых серий видеоигр. Одними из наиболее популярных считаются игры в жанрах:

* аркады;
* головоломки;
* симуляторы;
* стратегии;
* action;
* tower defense;
* платформеры;
* RPG;
* песочницы.

Из вышесказанного следует, что разработка компьютерных игр является актуальным и перспективным родом деятельности, а специалисты в этой области очень востребованными. В связи с этим, будет разрабатываться игровое приложение в одном из популярных жанров «Платформер» на языке программирования С# с использованием объектно-ориентированного подхода и графической библиотеки OpenGL.

## ТЕХНИЧЕСКИЕ СРЕДСТВА РАЗРАБОТКИ ИГРОВЫХ ПРИЛОЖЕНИЙ

## 1.1 Графическая библиотека *DirectX*

Microsoft DirectX – набор API функций для реализация разнообразных задач связанных с программированием, который позволяет создавать визуальные и звуковые эффекты в мультимедийных приложениях. Широко используется для разработки игровых приложений. За счёт данного набора API функций увеличивается качество и производительность обработки игр, использующих 3D графику, анимацию, стереозвук и другие элементы мультимедиа, а также увеличивается безопасность и производительность системы в общем [1].

Современные технологии прогрессируют достаточно быстро. С каждой новой версией DirectX становится возможным добавление новых элементов в игры и оптимизация старых.

По сути, Microsoft DirectX представляет собой набор программных библиотек. В случае их отсутствия в системных папках, какие-то эффекты в играх не будут отображаться, а возможно и такое, что игра вообще не запустится. На данный момент, минимальный набор этих файлов присутствует в установленной Windows. А если вдруг нет, то его можно скачать бесплатно на официальном сайте Microsoft.

DirectX поддерживается видеокартами по такому принципу: чем новее видеокарта, тем выше версию графической библиотеки она поддерживает. Следует отметить, что все предыдущие версии всё так же будут поддерживаться. Однако нужно понимать, что от версии Windows тоже зависит, какая версия DirectX будет поддерживаться.

Практически все части DirectX API представляют наборы «объектной модели компонентов».

В целом, Microsoft DirectX подразделяется на следующие компоненты:

* Direct3D – это компонент, предоставляющий возможность отображает трехмерную графику;
* Direct2D – интерфейс вывода двухмерной графики;
* DirectDraw – компонент для отображения двухмерных визуальных эффектов;
* DirectSound – компонент для увеличения обработки звуковых эффектов в фильмах и компьютерных играх;
* DirectInput – интерфейс для обработки данных, поступающих с клавиатуры, мыши и джойстика;
* DirectPlay – интерфейс сетевой коммуникации игр;
* DirectMusic – интерфейс для воспроизведения музыки;
* DirectShow – интерфейс, используемый для ввода/вывода аудио и/или видео данных;
* DirectX Instruments – технология для синтеза звука;
* DirectSetup – компонент, который отвечает за установку DirectX;
* DirectX Media Objects – компонент, который реализует функциональную поддержку потоковых объектов.

В основе данной графической библиотеки функции для отрисовки, которые запускают графический конвейер. На рисунке 1.1 показаны стандартные блоки конвейера.

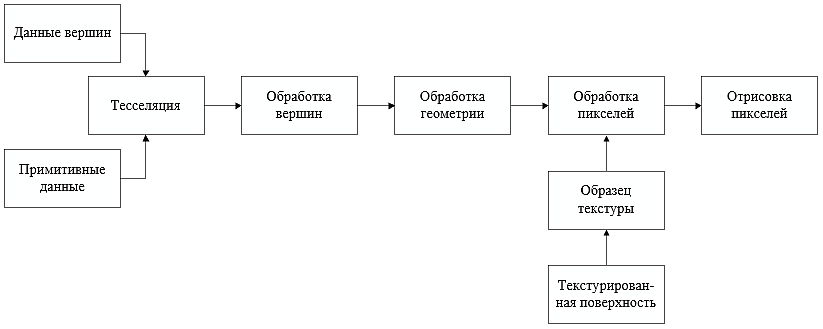


Рисунок 1.1 – Архитектура DirectX

Графический конвейер позволяет эффективно обрабатывать и отображать сцены Direct3D на дисплее, используя преимущества доступного оборудования.

## 1.2 Графическая библиотека *OpenGL*

OpenGL – спецификация, которой определяется программный интерфейс API для разработки приложений в области двумерной и трехмерной графики. Она включает в себя более 300 функций для отрисовки сложных сцен из простых примитивов. Графическая библиотека OpenGL используется для создания видеоигр, виртуальной реальности, а также для визуализации научных исследований [2].

Название библиотеки расшифровывается, как Open Graphics Library, что в переводе на русский означает «Открытая графическая библиотека». Ее открытость обосновывается тем, что она доступна для пользователя и разработчика без лицензионных отчислений.

OpenGL можно использовать с наиболее удобным языком, так для большинства языков есть привязки к этой графической библиотеке. Например, привязки OpenGL есть у C#, C++, Python, Java и у других.

Основные возможности библиотеки OpenGL:

* геометрические и растровые примитивы;
* использование В-сплайнов;
* видовые и модельные преобразования;
* работа с цветом (RGBA или выбор цвета из палитры);
* удаление невидимых линий и поверхностей;
* z-буферизация;
* одинарная и двойная буферизация;
* наложение текстуры;
* сглаживание пикселей;
* возможность задать источники света;
* атмосферные эффекты для наибольшей реалистичности, такие как дым, туман и другие;
* прозрачность объектов;
* использование списков изображений.

В основе OpenGL функции для отрисовки, которые запускают графический конвейер. На рисунке 1.2 показаны стандартные блоки конвейера.

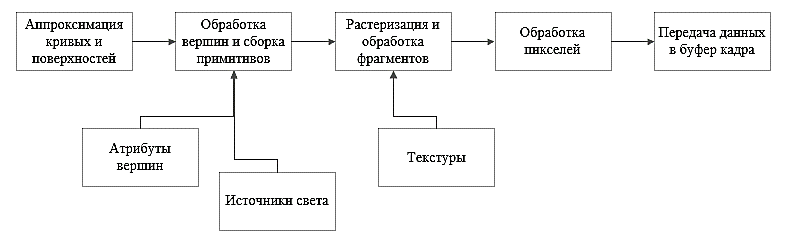


Рисунок 1.2 – Архитектура OpenGL

Функции OpenGL реализованы в модели клиент-сервер. Приложение выступает в роли клиента, так как оно обрабатывает команды, а сервер OpenGL интерпретирует и выполняет их.

На данный момент графическая библиотека OpenGL поддерживается на большинстве популярных аппаратных и программных платформах.

## 1.3 Отличия графических библиотек *OpenGL* и *DirectX*

Основные отличия рассматриваемых графических библиотек приведены в таблице 1.1.

Таблица 1.1 – Сравнение OpenGL и DirectX

|  |  |  |
| --- | --- | --- |
| Признаки для сравнения | OpenGL | DirectX |
| Операционная система | Кроссплатформенная библиотека, можно использовать в большинстве популярных ОС. | Операционная система Windows. |
| Библиотеки | У OpenGL только графическая библиотека. | DirectX содержит сразу несколько библиотек, а не только графическую, как OpenGL. В состав DirectX: основная графическая библиотека, вспомогательная графическая библиотека, библиотека для работы со звуком и другое. |
| Поддержка расширений | По сути, практически весь функционал OpenGL – расширения. Реализация OpenGL позволяет определять расширения к основной спецификации. | DirectX фиксирован в пределах одной мажорной версии. Какие-либо изменения или дополнения происходят только при выпуске следующей версии. |

Из данной таблицы следует, что обе библиотеки имеют свои преимущества и недостатки, но они не существенны. Выбор библиотеки нужно осуществлять, в зависимости от сферы применения API.

В разрабатываемом игровом приложении будет выбрана графическая библиотека OpenGL соответственно заданию.

## 1.4 Игровой жанр «Платформер»

Игры жанра «Платформер» характеризуются наличием разнообразных опор, называющиеся платформами, которые позволяют персонажу передвигаться по игровому миру. Как правило, герою игры нужно совершать прыжки, перескакивать от платформы к платформе, собирать предметы или сражаться с врагами, чтобы достигнуть конца уровня.

Большинство игр жанра «Платформер» обладают нереалистичной, мультипликационной графикой, поэтому чаще всего героями данного игрового жанра чаще всего являются персонажи из сказок и мультфильмов.

Начало 1980-х – период появления первых платформеров. Тогда игровые консоли были недостаточно мощными, чтобы отображать трехмерную графику. Такие игры в 3D появились только к 1990-м годам.

В играх данного жанра встречаются предметы, которые называются power-up, которые наделяют персонажа какими-либо временными способностями, например, ускорение, увеличение высоты прыжка и другое. Так же возможно получение коллекционных предметов и оружия. Противники могут быть многообразными в одной игре, различных типов, с различными способностями. Созданы они, разумеется, для интереса прохождений более сложных уровней. Чем выше уровень, тем сильнее противники. Игрок же должен уклоняться от их нападений, иногда даже защищаться в ответ. Если же игрок не станет уклоняться или защищаться, то его здоровье будет падать, или он вовсе умрёт.

Как правило, уровень пройти возможно при нахождении скрытого выхода, если выход не скрыт, тогда при нахождении загадочного ключа от этого самого выхода. Это всё так же создано для разнообразия и подогрева интереса игрока.

## 1.5 Обзор существующих игр жанра «Платформер»

Считается, что родоначальниками игр жанра «Платформер» является игры Space Panic и Apple Panic. Space Panic лишен возможности прыжка. Вместо этого персонаж роет ямы, в которые ему необходимо заманить пришельцев и нанести удар, чтобы выкинуть их из ямы и экрана, сохранив запас кислорода. Apple Panic – неавторизованная версия Space Panic для Apple II. Внешний вид игры Space Panic представлен на рисунке 1.3.



Рисунок 1.3 – Внешний вид игры Space Panic

Внешний вид игры Apple Panic представлен на рисунке 1.4.



Рисунок 1.4 – Внешний вид игры Apple Panic

За ними последовала игра Donkey Kong (1981 год), аркадная игра, которая была создана фирмой «Nintendo». Вскоре процесс прохождения уровня, в основном, стал горизонтальным с длинным прокручивающимся миром. Считается, что начало этому положила игра Pitfall!, выпущенная фирмой Activision в 1982 году для консолей Atari 2600. Внешний вид игры с новым прокручивающимся миром представлен на рисунке 1.5.



Рисунок 1.5 – Внешний вид игры Pitfall!

Наиболее популярной и переворотной игрой для фирмы «Nintendo» стал платформер Super Mario Bros (1985 год). Игра была выпущена для приставки Nintendo Entertainment System. Она наполнена большими и сложными уровнями, став примером для последующих создателей игр. Внешний вид данной игры представлен нп рисунке 1.6.



Рисунок 1.6 – Внешний вид игры Super Mario Bros

Термин «трёхмерный платформер» обозначает или геймплей, включающий три измерения, или использование 3D полигонов в реальном времени для отрисовки уровней и героев, или и то и другое. Появление трёхмерных платформеров изменило конечные цели некоторых платформеров. В большинстве 2D платформеров игроку нужно достичь одной цели на уровне. Однако во многих 3D платформерах в каждом уровне нужно собирать множество пазлов, разгадывая головоломки, собирать звезды, части ключей и другие предметы. Это сделано для наиболее эффективного использования трёхмерных областей и вознаграждения игрока за тщательное исследование уровней. Однако некоторые игроки не оценили это, считая это очень скучным занятием, особенно, когда разработчики придумывали очень сложные головоломки или места, где спрятаны части ключей или других предметов. Однако не все трёхмерные платформеры были такими. Например, игра Crash Bandicoot оставалось верной традиции двумерных платформеров, в ней использовались достаточно плоские уровни, в конце которых располагалась игровая цель[6]. Внешний вид данной игры представлен на рисунке 1.7.



Рисунок 1.7 – Внешний вид игры Crash Bandicoot

Также существуют изометрические платформеры, которые являются поджанром 2D и 3D платформеров. Такие платформеры отображают трехмерную сцену при помощи двумерной графики. Игровой мир отображает с жёстко ориентированной камерой без учёта перспективы. Ранними примерами изометрических платформеров являются Congo Bongo (1983 год) и 3D Ant Attack (1983 год). Внешний вид игры Congo Bongo представлен на рисунке 1.8.



Рисунок 1.8 – Внешний вид игры Congo Bongo

Внешний вид игры 3D Ant Attack представлен на рисунке 1.9.



Рисунок 1.9 – Внешний вид игры 3D Ant Attack

Так же ожидается появление первого четырехмерного платформера Miegakure. Способ отображения четырёх измерений – 3D-сечения. Игровой процесс в принципе схож с классическим 3D платформером, однако по нажатию кнопки одно из измерений меняется местами с её четвертым измерением, осуществляя перемещение в четвёртом измерении. Внешний вид будущей игры представлен на рисунке 1.10.



Рисунок 1.10 – Внешний вид игры Miegakure

## 1.6 Технические требования к курсовому проекту

В ходе аналитического обзора технических средств решения данной задачи и рассмотрения существующих аналогов игр жанра «Платформер» будет создано игровое приложение на языке программирования C# WPF с использованием графической библиотеки OpenGL, реализующее традиционные концепции данного жанра. Приложение будет разработано для одного игрока, и он будет считаться победителем, достигнув конца уровня.

Для разнообразия и сложности прохождения уровня будут также реализованы противники с различными типами атаки при помощи паттерна объектно-ориентированного программирования «Декоратор». При попадании врагов в главного героя игры, у него будут сниматься очки здоровья. Если они достигнут нуля, то игрок будет считаться проигравшим. Игрок классически будет прыгать по платформам, которые в свою очередь будут реализованы при помощи паттерна «Фабричный метод». Также у игрока будут сниматься очки здоровья при падении с платформ, находящихся на большой высоте, и при попадании на объект «шипы».

Внешний вид игры будет разработан при помощи OpenTK – расширенной низкоуровневой библиотеки, упрощающей работу с OpenGL, использующейся для создания игр, научных приложений или других проектов, требующих 2D и 3D графики, аудио или вычислительных функций.

## АРХИТЕКТУРА ИГРОВОГО ПРИЛОЖЕНИЯ «ПЛАТФОРМЕР»

## 2.1 Архитектура игрового приложения

При проектировании архитектуры игрового приложения был использован метод декомпозиции. Метод декомпозиции – метод разделения целого на части для наиболее наглядного видения того, что требуется сделать для решения общей задачи. То есть для проектирования архитектуры данного приложения главная задача была разделена на подзадачи, с помощью которых в итоге была решена главная задача проектирования. Разумеется, каждый элемент декомпозиции является уникальным для возможности сформировать и отличить составляющие объекта один от другого. Элементы, которые были получены в результате декомпозиции, связаны с функциями отдельных подзадач единой общей задачей.

Обобщенная функциональная схема игрового приложения представлена на рисунке 2.1.



Рисунок 2.1 – Обобщенная схема приложения

Для реализации игровой логики необходимо создать собственный игровой движок. Игровой движок – базовая программная реализация функционала игрового приложения. Основная задача, которую должен выполнять игровой движок с графической стороны, это умение работать со спрайтовой графикой, то есть он должен уметь отображать спрайты и двухмерные анимации. Со стороны функционала приложения основной задачей игрового движка является умение просчитывать физику движения игровых объектов. Для этих целей будет создана библиотека классов GameEngine.

## 2.2 Структура игрового приложения «Платформер»

Игровое приложение должно предоставлять следующий функционал:

* удобное и понятное меню игры;
* передвижение игрока по игровому полю;
* реализация 3-х типов врагов, снимающих очки здоровья у игрока;
* реализация объекта «шипы», снимающих жизнь;
* снятие очков здоровья при падении с высоких платформ;
* переход на следующий уровень;
* возможность пройти игру заново при потере всех жизней;
* завершение игры при победе игрока.

Для реализации игровой логики с использованием игрового движка будет создана новая библиотека классов GameLogic.

Потом будет создан основной проект приложения Windows Presentation Foundation. Он будет обеспечивать реализацию удобного меню приложения.

Так же при написании игрового приложения будут использованы паттерны объектно-ориентированного программирования: паттерн «Декоратор» для реализации различных типов врагов и паттерн «Фабричный метод» для реализации платформ. Данные паттерны будут так же реализованы в библиотеке классов GameLogic.

Паттерн «Декоратор» – это структурный паттерн, который предоставляет возможность динамически добавлять объектам дополнительную функциональность, помещая их в объекты-обертки. Оборачивать объекты можно сколько угодно раз, благодаря общему интерфейсу, как для объектов, так и для оберток. Помимо расширения функциональности, декоратор используется и для реализации обязанностей, которые могут быть сняты с объекта. Однако снятие декораций должно происходить в порядке, обратном наложению [4].

Схема реализации паттерна «Декоратор» представлена на рисунке 2.2.

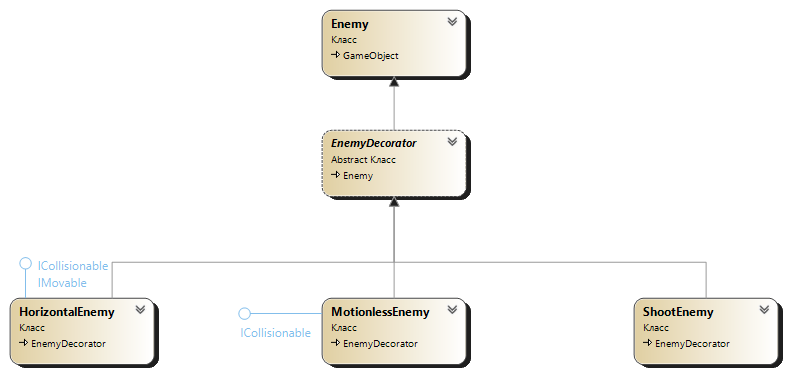


Рисунок 2.2 – Схема паттерна «Декоратор»

Класс Enemy содержит в себе все основные характеристики для врагов, а также виртуальный метод Update, который переопределяется в классах-наследниках от декоратора.

Паттерн «Фабричный метод» – это порождающий паттерн, который определяет интерфейс для создания объектов некоторого класса, но непосредственное решение о том, объект какого класса создавать происходит в подклассах. То есть паттерн предполагает, что базовый класс делегирует создание объектов классам-наследникам [5].

Паттерн применяется в случаях:

* когда классу заранее неизвестно, объекты каких классов ему нужно создавать;
* когда класс спроектирован так, чтобы объекты, которые он создает, специфици­ровались подклассами;
* когда класс делегирует свои обязанности одному из нескольких вспомогательных подклассов, и планируется локализовать знание о том, какой класс при­нимает эти обязанности на себя [4].

Схема реализации паттерна «Фабричный метод» представлена на рисунке 2.3.

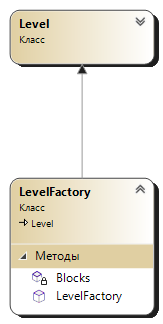


Рисунок 2.3 – Схема паттерна «Фабричный метод»

В данном проекте реализация паттерна «Фабричный метод» имеет неклассическую реализацию, в следствие нестандартной генерации уровней. Уровни проектировались в специальной для этой цели программе, бесплатном графическом редакторе тайловых карт Tiled Map Editor, а она в свою очередь генерировала xml-файл, в котором находились данные о том, на каком месте в матрице блоков находится тот или иной блок, обозначенный своей цифрой. Фабричный метод Blocks считывает и определяет какой тип блока находится под данной цифрой. Так как в xml-файле находятся данные не только о блоках, то класс LevelFactory отвечает полностью за обработку этого файла.

Tiled Map Editor – это редактор тайловых карт общего назначения для всех тайловых игр, таких как ролевые игры, боковые или с видом сверху платформеры.

Tiled очень гибкий. Его можно использовать для создания карт любого размера без ограничений по размеру плитки или количеству слоев или плиток, которые можно использовать. Картам, слоям, плиткам и объектам можно назначать произвольные свойства. Формат карты Tiled (.tmx) прост для понимания и позволяет использовать несколько наборов тайлов на любой карте. Также наборы тайлов можно редактировать в любое время процесса проектирования.

Проектирование одного из уровней в данной программе представлено на рисунке 2.4.

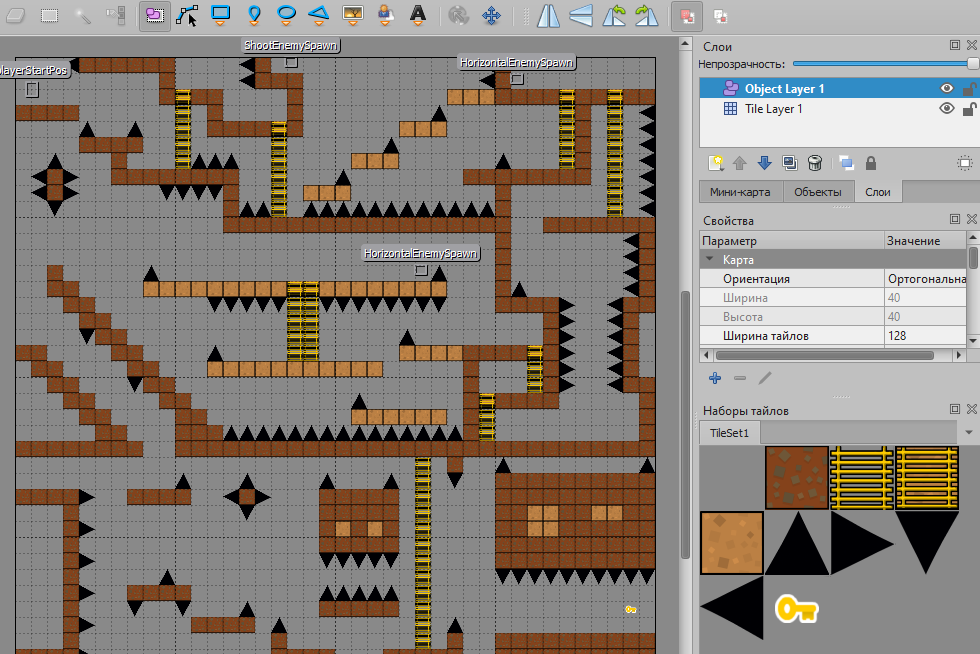


Рисунок 2.4 – Окно редактора Tiled Map Editor

## 2.3 Обзор взаимодействия классов игрового приложения

В процессе разработки игрового приложения была сформирована иерархическая структура классов, которые описывают все возможности своих сущностей.

Для лучшей читаемости кода и удобства были разработаны две библиотеки классов GameEngine и GameLogic. Библиотека классов GameEngine содержит основные классы для функционирования игрового движка, а библиотека классов GameLogic содержит реализацию игровых объектов, паттернов и другие классы игровой логики.

Далее рассматривается библиотека классов GameEngine и взаимодействие её классов.

Классы библиотеки GameEngine:

1. Класс GameObject – базовый абстрактный класс для всех игровых объектов, как движущихся, так и статических. В нем находятся поля position и size, которые определяют положение и размер объектов на игровой сцене;
2. Класс GameComponent – базовый абстрактный класс для всех компонентов игрового движка;
3. Класс Input – класс для работы с событиями клавиатуры;
4. Класс ContentPipe – класс для создания текстуры на основе добавленного изображения и вывода класса Texture2D со всеми данными о текстуре;
5. Класс Spritebatch – класс для работы с отрисовкой текстур. Содержит статический метод Draw для рендеринга спрайтов;
6. Класс Texture2D – класс, содержащий все свойства текстуры, а именно номер, высоту и ширину текстуры;
7. Класс View – класс для работы с камерой, в котором содержится метод перемещения камеры по игровому миру и другие необходимые математические методы.

Схема игрового движка представлена на рисунке 2.5.

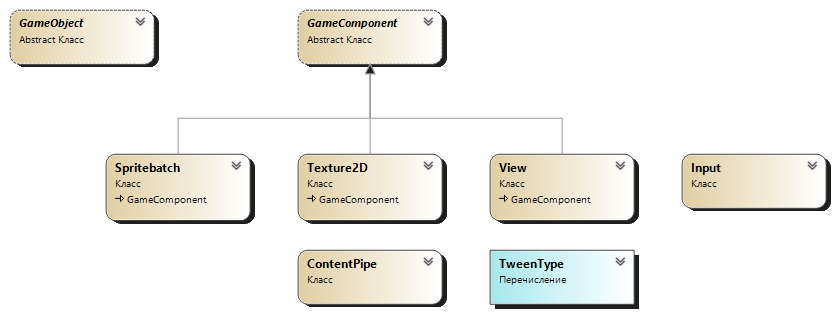


Рисунок 2.5 – Схема игрового движка

Следующей рассматривается библиотека классов GameLogic и взаимодействие её классов.

В данной библиотеке содержатся паттерны объектно-ориентированного программирования «Декоратор» и «Фабричный метод», а также логика игровых объектов.

Паттерн «Декоратор» используется для генерации различных типов врагов для дальнобойной и ближней атаки. В данном случае реализованы 3 типа врагов: HorizontalEnemy – враг с ближней атакой, двигающийся горизонтально, ShootEnemy – враг, атакующий игрока пулями с дальнего расстояния и MotionlessEnemy – не двигающийся враг, размеры которого меняются со временем – увеличиваются и уменьшаются.

Паттерн «Фабричный метод» используется для чтения различных типов блоков, а именно Solid – твердый непроходимый блок, Empty – пустота, Platform – блок-платформа, Ladder – блок-лестница, LadderPlatform – блок-лестница с платформой, SpikeUp, SpikeDown, SpikeLeft, SpikeRight – блоки «шипы», направленные в различные стороны, Key – блок-ключ, являющийся переходом на новый уровень.

Классы библиотеки GameLogic:

1. Класс Player – класс для главного персонажа. В нем содержатся методы для управления персонажем, прописана физика столкновения игрока с другими объектами на игровом поле, отрисовка различных спрайтов при движении и при нахождении персонажа на лестнице, то есть своего рода анимация;
2. Класс Enemy – общий класс для реализации врагов. В нем содержатся общие для всех врагов характеристики;
3. Класс EnemyDecorator – сам декоратор, реализуется посредством абстрактного класса и имеет тот же базовый класс, что и декорируемые объекты. Наследники данного класса представляют дополнительный функционал, которым должен быть расширен объект Enemy;
4. Класс HorizontalEnemy – класс, представляющий дополнительный функционал для Enemy в качестве горизонтального движения от блока к блоку;
5. Класс ShootEnemy – класс для врага, стреляющего в игрока, дополнительный функционал отсутствует помимо загрузки нового спрайта;
6. Класс MotionlessEnemy – класс, представляющий дополнительный функционал для Enemy в качестве увеличения размеров в зависимости от времени;
7. Класс Block – класс для работы с блоками, где содержаться свойства и метод для определения типа блока;
8. Класс Bullet – класс для пули, которой стреляет один из врагов. В нем содержатся методы для реализации движения пули, физики столкновения пули с игроком и другими игровыми объектами;
9. Класс Level – общий класс для генерации всех уровней;
10. Класс LevelFactory – сама фабрика для различных блоков.

Схема взаимодействия объектов игровой логики представлена на рисунке 2.6.

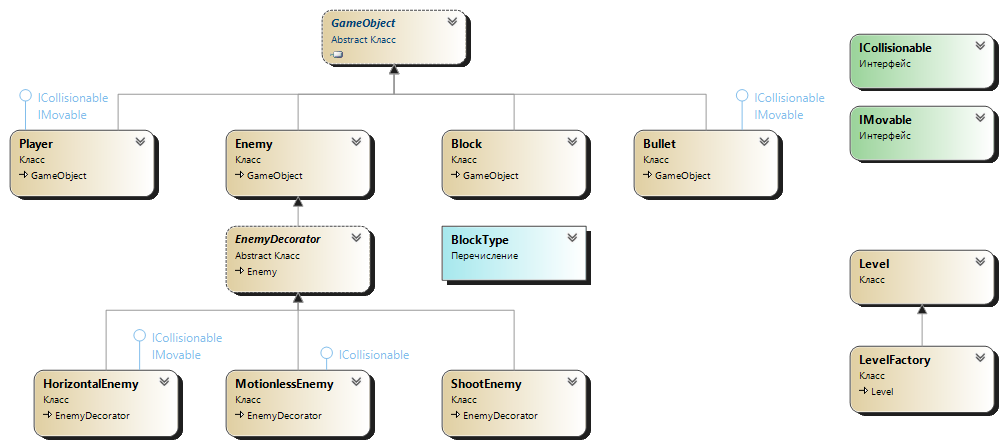


Рисунок 2.6 – Схема взаимодействия объектов игровой логики

Для реализации игрового меню создан проект Windows Presentation Foundation, под названием Game. В данном проекте находится три формы: MainWindow – собственно главное меню игрового приложения, AuthorWindow – дополнительное окно, в котором излагается краткая информация об авторе данного курсового проекта, HelpWindow – так же дополнительное окно, которое дает информацию об управлении персонажем.

Также в данном проекте содержится класс Platformer, который является одним из самых главных классов игрового приложения. Он наследует класс GameWindow, который в свою очередь является классом графической библиотеки-обертки OpenTK. Этот класс отвечает за создание игрового окна и отображение игровой сцены со всеми её составляющими. Класс Platformer использует ссылки на библиотеки классов GameEngine и GameLogic. В нем переопределены самые необходимые методы класса GameWindow.

## ПРОГРАММНАЯ РЕАЛИЗАЦИЯ ПРИЛОЖЕНИЯ «ПЛАТФОРМЕР»

## 3.1 Алгоритмическая реализация игрового приложения

В данном подпункте рассматривается алгоритм, реализующий игровое приложение «Платформер».

При запуске игры открывается проект Windows Presentation Foundation с пользовательским интерфейсом. Перед игроком открывается удобное и стильное меню, соответствующее жанру игры.

Главное меню содержит четыре кнопки: «Начать игру», «Об авторе», «Помощь» и «Выйти». Нажав на кнопку «Об авторе», можно увидеть информацию об авторе и теме курсовой работы, а если нажать на кнопку «Помощь», то откроется окно, где наглядно представлены кнопки управления персонажем. При нажатии на кнопку «Начать игру» открывается окно игры, которое предоставлено библиотекой-оберткой OpenTK. После нажатия на кнопку, класс Platformer считывает информацию о размере, положении и других характеристиках игрового окна. В инициализаторе класса Platformer происходит включение вертикальной синхронизации, отключается видимость курсора на игровом поле, учитывается альфа-канал для текстур всех объектов, создается камера, а также инициализируется клавиатура для игрока.

Далее в переопределенном методе OnLoad библиотеки OpenTK происходит инициализация игровых ресурсов, а именно происходит загрузка текстуры набора тайлов, загружается уровень, создается игрок и враги для него.

После того, как весь необходимый контент для игры был подгружен, начинают свою работу самые главные методы класса Platformer: метод OnUpdateFrame для учета обновлений кадров и метод OnRenderFrame для отрисовки кадров. Метод OnRenderFrame покадрово отображает объекты на игровой сцене, а в методе OnUpdateFrame взаимодействуют все остальные методы класса, обеспечивая функционирование игрового приложения.

За управление персонажем отвечает метод Move, который находится в классе Player.

Основная физика персонажа – это движение и столкновение с другими объектами на игровом поле. За это отвечает класс игрока Player. На различные объекты игрок реагирует по-разному. Например, игрок двигается на обычных блоках набирая скорость, а на лестнице равномерно и небыстро. Через твердые блоки он пройти не может, а на блоке-платформе он может и стоять, и при необходимости пройти сквозь неё. Блоки-шипы же полностью отнимают игроку жизнь, то есть целых сто очков здоровья. Блок-ключ является переходом на новый уровень. При попадании пули в персонажа у него снимаются очки здоровья. При касании любого врага, у игрока также снимаются очки здоровья.

В случае, когда игрок собрал все три ключа, игра завершается с победным всплывающим окном. В случае, когда игрок потерял все жизни, игра завершается. Однако для удержания игрока, ему поступает предложение сыграть заново, но и даётся возможность выйти из игры.

## 3.2 Тестирование игрового приложения

Одним из важнейших этапов разработки игрового приложения является его тестирование. Это необходимо для проверки правильности взаимодействия всех элементов задачи, а также их работоспособность и поведение. Тестирование приложения позволяет выявить и оперативно устранить ошибки при необходимости, оценить качество разрабатываемого продукта и получить информацию, необходимую для принятия дальнейший решений относительно данного программного продукта. Данные, полученные в результате тестирования крайне важны при планировании последующей стратегии развития приложения. Это считается своего рода диагностикой, которая повлияет на дальнейшие действия.

Для тестирования игрового приложения был создан тестовый проект PlatformerGameTests. Проект PlatformerGameTests тестирует классы библиотек GameEngine и GameLogic. В нём были разработаны модульные тесты для классов игрового движка приложения и класса, участвующего в реализации паттерна «Фабричный метод».

Для тестирования используется класс Assert, где определен набор статических методов для тестов. Этот класс находится в пространстве имен Microsoft.VisualStudio.TestTools.UnitTesting вместе с дополнительными классами, полезными для настройки и тестирования различных условий в модульных тестах.

Проект для модульных тестов PlatformerGameTests содержит в себе тесты для важных и сложных по функционалу классов.

Данный проект позволяет проверить, корректно ли работают разработанные классы и их методы, и свойства.

На рисунке 3.1 представлен результат прохождения модульных тестов проекта PlatformerGameTests.

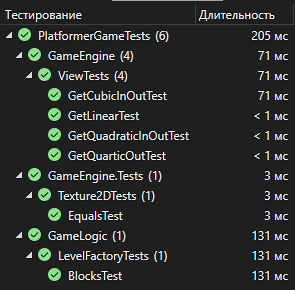


Рисунок 3.1 – Результат прохождения модульных тестов

Описание модульных тестов библиотеки PlatformerGameTests:

* EqualsTest – тестовый метод класса Texture2D, проверяющий работу с характеристиками текстур.
* GetLinearTest – тестовый метод класса View, проверяющий корректную работу линейной функции.
* GetQuadraticInOutTest – тестовый метод класса View, проверяющий корректную работу квадратичной функции.
* GetCubicInOutTest – тестовый метод класса View, проверяющий корректную работу кубической функции.
* GetQuarticOutTest – тестовый метод класса View, проверяющий корректную работу суперкубической функции.
* BlocksTest – тестовый метод класса LevelFactory, проверяющий корректную работу фабричного метода.

## 3.3 Опытная эксплуатация игрового приложения

После запуска разработанного игрового приложения появляется окно главного меню, которое содержит приятный глазу фон, название игры и четыре кнопки: «Начать игру», «Об авторе», «Помощь» и «Выйти».

Окно главного меню представлено на рисунке Д.1 в приложении Д.

После нажатия на кнопку «Об авторе» открывается окно, где содержится информация с именем, фамилией, отчеством, группой автора данной игры, а также тема курсового проекта. Это окно представлено на рисунке Д.2 в приложении Д.

После нажатия на кнопку «Помощь» открывается окно, которое содержит визуализированные кнопки для управления персонажем. Это окно представлено на рисунке Д.3 в приложении Д.

После нажатия на кнопку «Выход» происходит выход из программы.

После нажатия на кнопку «Начать игру» открывается окно библиотеки OpenTK с загруженным первым уровнем игрового приложения. На первом уровне главные цели для игрока – освоить управление персонажем, познакомиться с врагами, а также понаблюдать за что отнимаются очки здоровья и получить свой первый ключ. Ключ является блоком-переходом на следующий уровень.

Сцена игрового приложения при прохождении первого уровня представлена на рисунке 3.1.

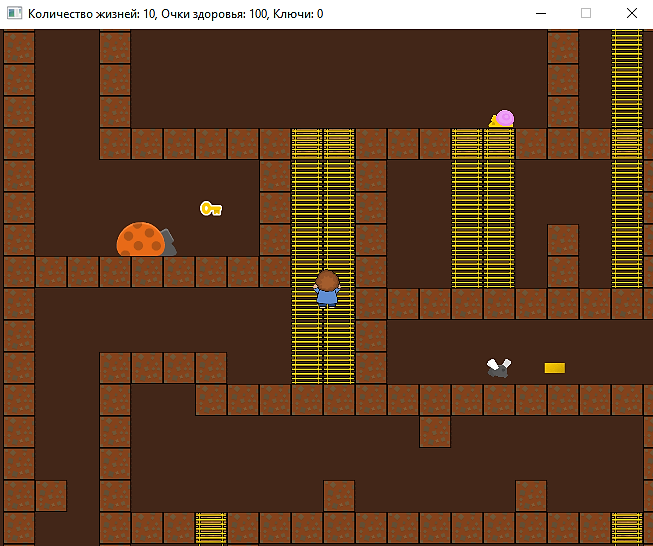


Рисунок 3.1 – Сцена прохождения первого уровня игры

Стоит отметить, что уровень целиком невозможно увидеть. Для сохранения интриги и интереса игрока есть область видимости. Область видимости уровня составляет примерно десять блоков со всех сторон. Этого хватает, чтобы уклониться от нападения врагов и увидеть препятствия дальше.

Количество жизней, очки здоровья и количество ключей отображается в названии окна.

После прохождения первого уровня игрока встречает второй. На втором уровне сложность повышается, так как появляется объект «шипы», который снимает жизнь целиком.

Сцена игрового приложения при прохождении второго уровня представлена на рисунке 3.2.

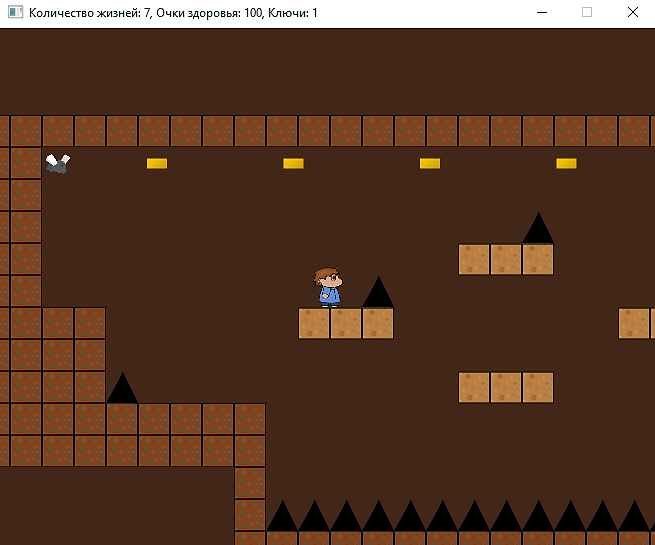


Рисунок 3.2 – Сцена прохождения второго уровня игры

На данном этапе стоит отметить то, что на каждом уровне количество жизней и очки здоровья обновляются.

На втором уровне игрока встречают, помимо блоков-шипов, блоки-платформы. На этих блоках можно как стоять, так и проходить сквозь них, если это потребуется.

Так же можно наблюдать изменения на панели вверху окна. Теперь у игрока есть один ключ, он упорно движется к победе. Однако если у игрока закончатся жизни, то его будет ждать всплывающее окно с предложением пройти игру заново. При нажатии на кнопку «Да», игра начнется заново с первого уровня, а если нажать на кнопку «Нет», тогда произойдет выход из игры.

Всплывающее окно при проигрыше представлено на рисунке 3.3.

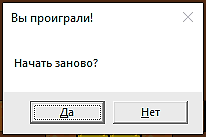


Рисунок 3.3 – Всплывающее окно при проигрыше

Но если удалось пройти второй уровень без проигрыша, тогда игрока встречает самый сложный завершающий третий уровень.

Сцена игрового приложения при прохождении третьего уровня представлена на рисунке 3.4.

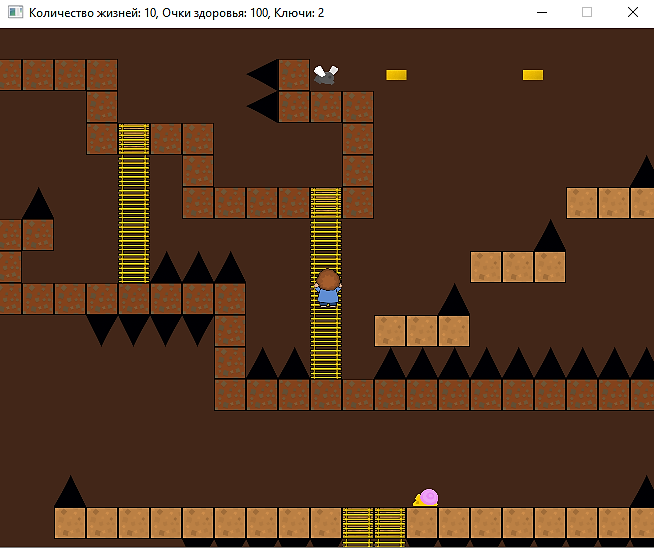


Рисунок 3.4 – Сцена прохождения третьего уровня игры

Данный уровень отличается своей максимальной сложностью, поэтому для его прохождения стоит аккуратно рассчитывать каждый шаг и прыжок.

На третьем уровне также можно наблюдать изменение количества ключей в заголовке. Теперь у игрока есть два ключа, и для победы осталось получить третий ключ.

Далее рассматриваются начальные характеристики персонажа при появлении на новом уровне.

Начальные характеристики персонажа:

* количество жизней – 10 единиц;
* количество очков здоровья – 100 единиц;
* количество ключей – (n-1) штук, где n – номер уровня;
* скорость – 0.

Клавиши на клавиатуре, отвечающие за управление персонажем представлены в таблице 3.1.

Таблица 3.1 – Управление персонажем

|  |  |
| --- | --- |
| Действие | Клавиша |
| Движение вправо | Right |
| Движение влево | Left |
| Подниматься по лестнице | Up |
| Опускаться по лестнице | Down |
| Прыжок | Space |

Спрайты главного персонажа, которые были использованы в приложении представлены на рисунке 3.5.

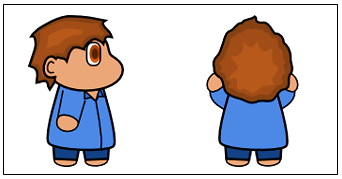


Рисунок 3.5 – Внешний вид игрового персонажа

Типы блоков:

* твёрдый блок – непроходимый;
* блок-платформа – с возможностью прохождения через него;
* блок-лестница – возможно только спускаться и подниматься;
* блок-лестница с платформой – с возможностью стоять на нём;
* блоки-шипы – отнимают жизнь;
* блок-ключ – переход на новый уровень.

Типы игровых блоков представлены на рисунке 3.6.

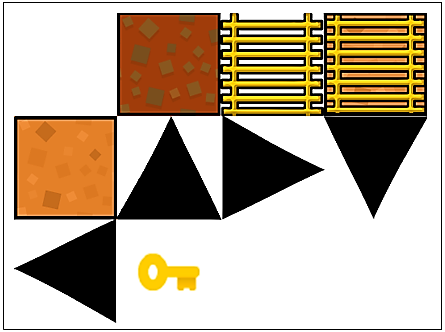


Рисунок 3.6 – Внешний вид всех игровых блоков

Целью игрока также является выживание. В этом ему препятствуют враги. Внешний вид всех врагов представлен на рисунке 3.7.



Рисунок 3.7 – Внешний вид врагов

Первый враг на рисунке – это горизонтально движущийся враг, второй – изменяющий свои размеры, а третий – стреляющий.

При касании любого из врагов отнимается одно очко здоровья. Общее же количество снятый очков здоровья будет зависеть от длительности касания с врагом.

Стреляющий враг имеет в своем арсенале бесконечное количество пуль, которыми стреляет примерно раз в 1.5 секунды. При попадании пуль в игрока у него снимается 10 очков здоровья.

Внешний вид пули представлен на рисунке 3.8.



Рисунок 3.8 – Внешний вид пули

Падение игрока с платформ реализовано таким образом, что если время падения персонажа больше 5/6 секунды, то отнимается (t-40) очков здоровья, где t – время падения с блока.

При удачном прохождении всех уровней и получении последнего ключа, игрока будет встречать победное всплывающее окно с поздравлением.

Всплывающее окно при победе представлено на рисунке 3.9.

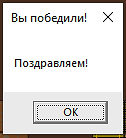


Рисунок 3.9 – Всплывающее окно при победе

При нажатии на кнопку «ОК», программа завершится.

**ЗАКЛЮЧЕНИЕ**

В результате проделанной работы было разработано игровое приложение жанра «Платформер» для одного игрока. В соответствие с заданием курсового проекта игра написана с использованием средств языка программирования C# Windows Presentation Foundation и паттернов объектно-ориентированного программирования. Для отображения объектов на игровом поле была использована спрайтовая графика и средства графической библиотеки OpenGL.

При выполнении курсового проекта первоначально был проведен аналитический обзор литературы в области разработки игровых приложений, а также изучена предметная область и средства её реализации.

Во время разработки игрового приложения был проведен анализ подходов к решению поставленной задачи, были выделены основные сущности игрового мира, а также их взаимосвязи и составлена иерархия классов.

При реализации игрового проекта были использованы шаблоны проектирования: паттерн «Декоратор» – для реализации различных типов врагов и паттерн «Фабричный метод» – для реализации генерации платформ. Данные паттерны придали библиотеке классов такое свойство, как масштабируемость, что в будущем упростит добавление нового функционала и поспособствует улучшению и развитию игрового приложения.

Отдельные разработанные классы были протестированы модульными тестами, что гарантирует корректную работу приложения и обеспечивает минимизацию возможности получения ошибки пользователем.

Конечным этапом стало создание пользовательского интерфейса, который обеспечивает удобное использование игрового приложения.

Курсовая работа выполнена самостоятельно, проверена в системе «Атиплагиат». Процент оригинальности составляет 97,84%. Цитирования обозначены ссылками на публикации, указанные в «Списке использованных источников».

**СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ**

1. Миллер, Т. *Managed DirectX* 9 с управляемым кодом. Программирование игр и графика / Т. Миллер. – М.:КомБук, 2005. – 401 с.
2. *OpenGL* Библиотека программиста. 4-е издание / Д. Шрайнер [и др.]. – СПб: Питер, 2018. – 311с.
3. Как создать игру жанра 2*d* платформер? [Электронный ресурс]. – 2017. – Режим доступа: *https://kychka-pc.ru/sfml/kak-sozdat-igru-zhanra-2d-platformer.html*. – Дата доступа: 15.03.2022.
4. Приемы объектно-ориентированного программирования. Паттерны проектирования / Э. Гамма [и др.]. – СПб: Питер, 2015. – 368 с.
5. Фабричный метод [Электронный ресурс]. – 2015. – Режим доступа: *https://metanit.com/sharp/patterns/2.1.php*. – Дата доступа: 10.04.2022.
6. *The evolution of platform games in* 9 *steps* [Электронный ресурс]. – 2013. – Режим доступа: *https://www.redbull.com/in-en/evolution-of-platformers*. – Дата доступа: 10.04.2022.
7. Декоратор [Электронный ресурс]. – 2015. – Режим доступа: *https://metanit.com/sharp/patterns/4.1.php*. – Дата доступа: 11.04.2022.

**ПРИЛОЖЕНИЯ**

**ПРИЛОЖЕНИЕ А**

(обязательное)

**Листинг программы «Платформер»**

Листинг класса MainWindow.xaml.cs:

using OpenTK;

using System.Windows;

namespace Game

{

/// <summary>

/// Логика взаимодействия для MainWindow.xaml

/// </summary>

public partial class MainWindow : Window

{

/// <summary>

/// Окно главного меню.

/// </summary>

public MainWindow()

{

InitializeComponent();

WindowStartupLocation = WindowStartupLocation.CenterScreen;

}

private void StartButton\_Click(object sender, RoutedEventArgs e)

{

this.Close();

using (Platformer gameWindow = new Platformer())

{

gameWindow.WindowBorder = WindowBorder.Fixed;

gameWindow.Run(60, 60);

}

}

private void AutorButton\_Click(object sender, RoutedEventArgs e)

{

AutorWindow autorWindow = new AutorWindow();

autorWindow.Show();

}

private void ExitButton\_Click(object sender, RoutedEventArgs e)

{

Close();

}

private void HelpButton\_Click(object sender, RoutedEventArgs e)

{

HelpWindow helpWindow = new HelpWindow();

helpWindow.Show();

}

}

}

Листинг класса Platformer.cs:

using GameEngine;

using GameLogic;

using OpenTK;

using OpenTK.Graphics.OpenGL;

using System;

using System.Collections.Generic;

using System.Drawing;

namespace Game

{

/// <summary>

/// Работа с игровым окном.

/// </summary>

public class Platformer : GameWindow

{

Texture2D tileset;

View view;

Level level;

Player player;

List<Enemy> enemyList = new List<Enemy>();

int levelNum = 0;

List<string> lvlNames = new List<string>()

{

"FirstLevel.tmx",

"SecondLevel.tmx",

"ThirdLevel.tmx"

};

private int livesCount, health, keys;

private int GRIDSIZE = 32, TILESIZE = 128;

/// <summary>

/// Конструктор с параметрами окна.

/// </summary>

public Platformer() : base(

640, 480, new OpenTK.Graphics.GraphicsMode(32, 24, 0, 8), "Platformer",

GameWindowFlags.FixedWindow,

DisplayDevice.Default

)

{

CursorVisible = false;

VSync = VSyncMode.On;

GL.Enable(EnableCap.Texture2D);

//Для учёта альфа-канала

GL.Enable(EnableCap.Blend);

GL.BlendFunc(BlendingFactor.SrcAlpha, BlendingFactor.OneMinusSrcAlpha);

view = new View(Vector2.Zero, 0.0, 1.0f);

Input.Initialize(this);

}

/// <summary>

/// Инициализация ресурсов.

/// </summary>

/// <param name="e"></param>

protected override void OnLoad(EventArgs e)

{

base.OnLoad(e);

LoadNewLvl();

}

private void LoadNewLvl()

{

enemyList.Clear();

tileset = ContentPipe.LoadTexture("FullTilesSet.png");

level = new LevelFactory(20, 20, $"Content/{lvlNames[levelNum]}");

player = new Player(new Vector2(level.playerStartPos.X + 0.5f,

level.playerStartPos.Y + 0.5f) \* GRIDSIZE, new Vector2(0, 0.5f), keys + 1);

AddEnemies();

}

private void AddEnemies()

{

Enemy enemy = new Enemy(Vector2.Zero);

foreach (Point p in level.enemiesHorSpawn)

{

enemyList.Add(new HorizontalEnemy(enemy, new Vector2(p.X + 0.5f, p.Y + 0.5f) \* GRIDSIZE));

}

foreach (Point p in level.enemiesShootSpawn)

{

enemyList.Add(new ShootEnemy(enemy, new Vector2(p.X + 0.5f, p.Y + 0.5f) \* GRIDSIZE));

}

foreach (Point p in level.enemiesMotionlessSpawn)

{

enemyList.Add(new MotionlessEnemy(enemy, new Vector2(p.X + 0.5f, p.Y + 0.5f) \* GRIDSIZE));

}

}

/// <summary>

/// Проверка изменений расширений экрана.

/// </summary>

/// <param name="e"></param>

protected override void OnResize(EventArgs e)

{

base.OnResize(e);

}

/// <summary>

/// Учет обновлений кадров.

/// </summary>

/// <param name="e"></param>

protected override void OnUpdateFrame(FrameEventArgs e)

{

base.OnUpdateFrame(e);

ImplemOfMEnemies();

ImplemOfFalling();

IfBulsIntersectWithPlayer();

ImplemOfShooting();

RememberLastUpdate();

IsIntersectsWithEnemies();

IsHealthAboveZero();

UpdatePlayerEnemiesBullets();

IfPlayerDead();

IfCollectedAllKeys();

ImplemOfKeys();

view.SetPosition(player.position, TweenType.QuarticOut, 15);

IfRespawnButPressed();

Input.Update();

view.Update();

Title = player.ToString();

}

private void UpdatePlayerEnemiesBullets()

{

foreach (Enemy he in enemyList)

he.Update(ref level);

foreach (Bullet b in level.bullets)

b.Update(ref level);

int n = level.bullets.Count;

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

{

if (level.bullets[i].IsBumped)

{

level.bullets.Remove(level.bullets[i]);

n--;

i--;

}

}

player.Update(ref level);

}

private void IfCollectedAllKeys()

{

if (player.Key == 1 + lvlNames.Count)

{

System.Windows.MessageBox.Show("Поздравляем!", "Вы победили!");

this.Close();

}

}

private void IfPlayerDead()

{

if (livesCount != player.LivesCount)

{

level.CountOfJumpingTime = 0;

ForcedRespawn();

player.Health = 100;

}

if (livesCount - player.LivesCount > 1)

player.LivesCount = player.LivesCount + (livesCount - player.LivesCount - 1);

if (player.LivesCount <= 0)

{

Defeat();

player.LivesCount = 10;

}

}

private void RememberLastUpdate()

{

keys = player.Key;

livesCount = player.LivesCount;

health = player.Health;

}

private void Defeat()

{

System.Windows.Forms.DialogResult result =

(System.Windows.Forms.DialogResult)System.Windows.MessageBox.Show("Начать заново?",

"Вы проиграли!", System.Windows.MessageBoxButton.YesNo);

if (result == System.Windows.Forms.DialogResult.Yes)

{

this.Close();

Platformer platformer = new Platformer();

platformer.Run(60, 60);

}

if (result == System.Windows.Forms.DialogResult.No)

{

this.Close();

}

}

private void IfBulsIntersectWithPlayer()

{

foreach (Bullet b in level.bullets)

{

if (Math.Abs(player.position.X - b.position.X) < 20 &&

Math.Abs(player.position.Y - b.position.Y) < 20)

{

player.Health -= b.damage;

b.IsBumped = true;

}

}

}

private void ImplemOfShooting()

{

level.CountShootingTime++;

if (level.CountShootingTime > 90)

{

foreach (Enemy e in enemyList)

{

if (e is ShootEnemy shootEnemy)

{

level.bullets.Add(new Bullet(shootEnemy.position));

}

}

level.CountShootingTime = 0;

}

}

private void ImplemOfFalling()

{

if (player.climbing == false && player.grounded == false && player.onLadder == false)

{

level.CountOfJumpingTime++;

}

else

{

if (level.CountOfJumpingTime > 50)

{

player.Health -= level.CountOfJumpingTime - 40;

}

level.CountOfJumpingTime = 0;

}

}

private void ImplemOfKeys()

{

if (keys != player.Key)

{

if (levelNum < lvlNames.Count - 1)

{

levelNum++;

}

else

levelNum = 0;

LoadNewLvl();

}

}

private void ImplemOfMEnemies()

{

level.CountOfMEnemy++;

if (level.CountOfMEnemy > 360)

{

level.CountOfMEnemy = 0;

}

}

private void ForcedRespawn()

{

player.position =

new Vector2(level.playerStartPos.X + 0.5f, level.playerStartPos.Y + 0.5f) \* GRIDSIZE;

player.Speed = Vector2.Zero;

}

private void IfRespawnButPressed()

{

if (Input.KeyPress(OpenTK.Input.Key.R))

{

player.position =

new Vector2(level.playerStartPos.X + 0.5f, level.playerStartPos.Y + 0.5f) \* GRIDSIZE;

player.Speed = Vector2.Zero;

}

}

/// <summary>

/// Отрисовка кадра.

/// </summary>

/// <param name="e"></param>

protected override void OnRenderFrame(FrameEventArgs e)

{

base.OnRenderFrame(e);

GL.Clear(ClearBufferMask.ColorBufferBit);

GL.ClearColor(0.258f, 0.149f, 0.094f, 1.0f);

Spritebatch.Begin(this.Width, this.Height);

view.ApplyTransform();

for (int x = 0; x < level.Width; x++)

{

for (int y = 0; y < level.Height; y++)

{

RectangleF source = new RectangleF(0, 0, 0, 0);

switch (level[x, y].Type)

{

case BlockType.Ladder:

source = new RectangleF(2 \* TILESIZE, 0 \* TILESIZE, TILESIZE, TILESIZE);

break;

case BlockType.LadderPlatform:

source = new RectangleF(3 \* TILESIZE, 0 \* TILESIZE, TILESIZE, TILESIZE);

break;

case BlockType.Solid:

source = new RectangleF(1 \* TILESIZE, 0 \* TILESIZE, TILESIZE, TILESIZE);

break;

case BlockType.Platform:

source = new RectangleF(0 \* TILESIZE, 1 \* TILESIZE, TILESIZE, TILESIZE);

break;

case BlockType.SpikeUp:

source = new RectangleF(1 \* TILESIZE, 1 \* TILESIZE, TILESIZE, TILESIZE);

break;

case BlockType.SpikeRight:

source = new RectangleF(2 \* TILESIZE, 1 \* TILESIZE, TILESIZE, TILESIZE);

break;

case BlockType.SpikeDown:

source = new RectangleF(3 \* TILESIZE, 1 \* TILESIZE, TILESIZE, TILESIZE);

break;

case BlockType.SpikeLeft:

source = new RectangleF(0 \* TILESIZE, 2 \* TILESIZE, TILESIZE, TILESIZE);

break;

case BlockType.Key:

source = new RectangleF(1 \* TILESIZE, 2 \* TILESIZE, TILESIZE, TILESIZE);

break;

}

Spritebatch.Draw(tileset, new Vector2(x \* GRIDSIZE, y \* GRIDSIZE), new Vector2((float)

GRIDSIZE / TILESIZE), Color.White, Vector2.Zero, source);

}

}

foreach (Enemy en in enemyList)

en.Draw();

foreach (Bullet b in level.bullets)

b.Draw();

player.Draw();

SwapBuffers();

}

private void IsIntersectsWithEnemies()

{

if (enemyList.Count > 0)

{

foreach (Enemy e in enemyList)

{

if (e is HorizontalEnemy he && Math.Abs(player.position.Y - he.position.Y) < 20 &&

Math.Abs(player.position.X - he.position.X) < 25)

player.Health -= 1;

if (e is MotionlessEnemy me)

{

if (level.CountOfMEnemy > 180)

{

if (Math.Abs(player.position.Y - me.position.Y) < 50 &&

Math.Abs(player.position.X - me.position.X) < 50)

player.Health -= 1;

}

else

{

if (Math.Abs(player.position.Y - me.position.Y) < 20 &&

Math.Abs(player.position.X - me.position.X) < 20)

player.Health -= 1;

}

}

if (e is ShootEnemy se && Math.Abs(player.position.Y - se.position.Y) < 20 &&

Math.Abs(player.position.X - se.position.X) < 20)

player.Health -= 1;

}

}

}

private void IsHealthAboveZero()

{

if (player.Health <= 0)

{

player.Health = 100;

player.LivesCount -= 1;

ForcedRespawn();

}

}

/// <summary>

/// Удаление загруженных ресурсов.

/// </summary>

/// <param name="e"></param>

protected override void OnUnload(EventArgs e)

{

base.OnUnload(e);

}

}

}

Листинг класса GameObject.cs:

using OpenTK;

using System.Drawing;

namespace GameEngine

{

/// <summary>

/// Игровой объект.

/// </summary>

public abstract class GameObject

{

/// <summary>

/// Размер сетки.

/// </summary>

public static readonly int Gridsize = 32;

/// <summary>

/// Позиция.

/// </summary>

public Vector2 position;

/// <summary>

/// Размер.

/// </summary>

public Vector2 size;

/// <summary>

/// Прямоугольник столкновения.

/// </summary>

public RectangleF ColRec

{

get

{

return new RectangleF(position.X - size.X / 2.0f, position.Y - size.Y / 2.0f, size.X, size.Y);

}

}

/// <summary>

/// Отрисовка прямоугольника.

/// </summary>

public RectangleF DrawRec

{

get

{

return new RectangleF(ColRec.X - 5, ColRec.Y, ColRec.Width + 10, ColRec.Height);

}

}

/// <summary>

/// Инициализатор игрового объекта.

/// </summary>

public GameObject()

{

this.position = Vector2.Zero;

this.size = Vector2.Zero;

}

}

}

Листинг класса GameComponent.cs:

namespace GameEngine

{

/// <summary>

/// Тип компонентов игрового объекта.

/// </summary>

public abstract class GameComponent { }

}

Листинг перечисления TweenType.cs:

namespace GameEngine

{

/// <summary>

/// Анимация движения.

/// </summary>

public enum TweenType

{

/// <summary>

/// Мгновенный тип.

/// </summary>

Instant,

/// <summary>

/// Линейный тип.

/// </summary>

Linear,

/// <summary>

/// Квадратичный тип.

/// </summary>

QuadraticInOut,

/// <summary>

/// Кубический тип.

/// </summary>

CubicInOut,

/// <summary>

/// Суперкубический тип.

/// </summary>

QuarticOut

}

}

Листинг класса Input.cs:

using OpenTK;

using OpenTK.Input;

using System.Collections.Generic;

namespace GameEngine

{

/// <summary>

/// Работа с событиями клавиатуры.

/// </summary>

public class Input

{

private static List<Key> keysDown;

private static List<Key> keysDownLast;

/// <summary>

/// Инициализация.

/// </summary>

/// <param name="game">Игровое окно.</param>

public static void Initialize(GameWindow game)

{

keysDown = new List<Key>();

keysDownLast = new List<Key>();

game.KeyDown += game\_KeyDown;

game.KeyUp += game\_KeyUp;

}

private static void game\_KeyUp(object sender, KeyboardKeyEventArgs e)

{

while (keysDown.Contains(e.Key))

keysDown.Remove(e.Key);

}

private static void game\_KeyDown(object sender, KeyboardKeyEventArgs e)

{

keysDown.Add(e.Key);

}

/// <summary>

/// Метод обновления.

/// </summary>

public static void Update()

{

keysDownLast = new List<Key>(keysDown);

}

/// <summary>

/// Метод, возвращающий зажатую клавишу.

/// </summary>

/// <param name="key">Нажатая клавиша.</param>

/// <returns>Нажатая клавиша.</returns>

public static bool KeyPress(Key key)

{

return (keysDown.Contains(key) && !keysDownLast.Contains(key));

}

/// <summary>

/// Метод, возвращающий последнее нажатие клавиши.

/// </summary>

/// <param name="key">Нажатая клавиша.</param>

/// <returns>Нажатая клавиша.</returns>

public static bool KeyRelease(Key key)

{

return (!keysDown.Contains(key) && keysDownLast.Contains(key));

}

/// <summary>

/// Метод, возвращающий нажатую клавишу.

/// </summary>

/// <param name="key">Нажатая клавиша.</param>

/// <returns>Нажатая клавиша.</returns>

public static bool KeyDown(Key key)

{

return keysDown.Contains(key);

}

}

}

Листинг класса ContentPipe.cs:

using OpenTK.Graphics.OpenGL;

using System.Drawing;

using System.Drawing.Imaging;

using System.IO;

namespace GameEngine

{

/// <summary>

/// Загрузка текстуры.

/// </summary>

public class ContentPipe

{

/// <summary>

/// Создание текстуры.

/// </summary>

/// <param name="path"> Путь к изображению. </param>

/// <returns> Класс с параметрами текстуры. </returns>

public static Texture2D LoadTexture(string path)

{

if (!File.Exists("Content/" + path))

{

throw new FileNotFoundException($"Файл не найден, проверьте путь {path}");

}

int id = GL.GenTexture();

GL.BindTexture(TextureTarget.Texture2D, id);

Bitmap bmp = new Bitmap("Content/" + path);

BitmapData data = bmp.LockBits(new Rectangle(0, 0, bmp.Width, bmp.Height),

ImageLockMode.ReadOnly,

System.Drawing.Imaging.PixelFormat.Format32bppArgb);

GL.TexImage2D(TextureTarget.Texture2D, 0, PixelInternalFormat.Rgba, data.Width, data.Height,

0, OpenTK.Graphics.OpenGL.PixelFormat.Bgra, PixelType.UnsignedByte, data.Scan0);

bmp.UnlockBits(data);

GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureWrapS,

(int)TextureWrapMode.Clamp);

GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureWrapT,

(int)TextureWrapMode.Clamp);

GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureMinFilter,

(int)TextureMinFilter.Linear);

GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureMagFilter,

(int)TextureMagFilter.Linear);

return new Texture2D(id, bmp.Width, bmp.Height);

}

}

}

Листинг класса Spritebatch.cs:

using OpenTK;

using OpenTK.Graphics.OpenGL;

using System.Drawing;

namespace GameEngine

{

/// <summary>

/// Работа с текстурой, отрисовка.

/// </summary>

public class Spritebatch : GameComponent

{

/// <summary>

/// Отрисовка текстуры.

/// </summary>

/// <param name="texture">Текстура.</param>

/// <param name="rectangle">Прямоугольник.</param>

public static void Draw(Texture2D texture, RectangleF rectangle)

{

Draw(texture, new Vector2(rectangle.X, rectangle.Y),

new Vector2(rectangle.Width / texture.Width, rectangle.Height / texture.Height),

Color.White, Vector2.Zero);

}

/// <summary>

/// Отрисовка текстуры.

/// </summary>

/// <param name="texture"> Текстура. </param>

/// <param name="position"> Позиция. </param>

/// <param name="scale"> Масштаб. </param>

/// <param name="color"> Цвет. </param>

/// <param name="origin"> Начало. </param>

/// <param name="sourceRec">Источник.</param>

public static void Draw(Texture2D texture, Vector2 position, Vector2 scale, Color color,

Vector2 origin, RectangleF? sourceRec = null)

{

Vector2[] vertices = new Vector2[4]

{

new Vector2 (0, 0),

new Vector2 (1, 0),

new Vector2 (1, 1),

new Vector2 (0, 1)

};

GL.BindTexture(TextureTarget.Texture2D, texture.ID);

GL.Begin(PrimitiveType.Quads);

GL.Color3(color);

for (int i = 0; i < vertices.Length; i++)

{

if (sourceRec == null)

GL.TexCoord2(vertices[i]);

else

{

GL.TexCoord2((sourceRec.Value.Left + vertices[i].X \* sourceRec.Value.Width) / texture.Width,

(sourceRec.Value.Top + vertices[i].Y \* sourceRec.Value.Height) / texture.Height);

}

vertices[i].X \*= (sourceRec == null) ? texture.Width : sourceRec.Value.Width;

vertices[i].Y \*= (sourceRec == null) ? texture.Height : sourceRec.Value.Height;

vertices[i] -= origin;

vertices[i] \*= scale;

vertices[i] += position;

GL.Vertex2(vertices[i]);

}

GL.End();

}

/// <summary>

/// Begin метод.

/// </summary>

/// <param name="screenWidth"></param>

/// <param name="screenHeight"></param>

public static void Begin(int screenWidth, int screenHeight)

{

GL.MatrixMode(MatrixMode.Projection);

GL.LoadIdentity();

GL.Ortho(-screenWidth / 2.0f, screenWidth / 2.0f, screenHeight / 2.0f, -screenHeight / 2.0f, 0.0f, 1.0f);

}

}

}

Листинг класса Texture2D.cs:

namespace GameEngine

{

/// <summary>

/// Свойства текстуры.

/// </summary>

public class Texture2D : GameComponent

{

private int id;

private int width;

private int height;

/// <summary>

/// Номер текстуры.

/// </summary>

public int ID

{

get { return id; }

}

/// <summary>

/// Ширина.

/// </summary>

public int Width

{

get { return width; }

}

/// <summary>

/// Высота.

/// </summary>

public int Height

{

get { return height; }

}

/// <summary>

/// Инициализатор текстуры.

/// </summary>

/// <param name="id"> Номер текстуры. </param>

/// <param name="width"> Ширина. </param>

/// <param name="height"> Высота. </param>

public Texture2D(int id, int width, int height)

{

this.id = id;

this.width = width;

this.height = height;

}

/// <summary>

/// Метод, сравнивающий текущий объект с переданным в параметр (нужен для тестов).

/// </summary>

/// <param name="obj">Объект для сравнения.</param>

/// <returns>Да, если равны, в противном случае - нет.</returns>

public override bool Equals(object obj)

{

return obj is Texture2D d &&

ID == d.ID &&

Width == d.Width &&

Height == d.Height;

}

/// <summary>

/// Получение хэш-кода.

/// </summary>

/// <returns>Хэш-код.</returns>

public override int GetHashCode()

{

int hashCode = 1463928665;

hashCode = hashCode \* -1521134295 + ID.GetHashCode();

hashCode = hashCode \* -1521134295 + Width.GetHashCode();

hashCode = hashCode \* -1521134295 + Height.GetHashCode();

return hashCode;

}

}

}

Листинг класса View.cs:

using OpenTK;

using OpenTK.Graphics.OpenGL;

using System;

namespace GameEngine

{

/// <summary>

/// Работа с камерой.

/// </summary>

public class View : GameComponent

{

/// <summary>

/// Позиция.

/// </summary>

private Vector2 position;

/// <summary>

/// Вращение (в радианах, по часовой стрелке).

/// </summary>

public double rotation;

/// <summary>

/// Масштаб

/// 1 - no zoom

/// 2 - 2x zoom

/// </summary>

public double zoom;

private Vector2 positionGoto, positionFrom;

private TweenType tweenType;

private int currentStep, tweenSteps;

/// <summary>

/// Свойство позиции.

/// </summary>

public Vector2 Position

{

get { return this.position; }

}

/// <summary>

/// Свойство позиции перехода.

/// </summary>

public Vector2 PositionGoto

{

get { return this.positionGoto; }

}

/// <summary>

/// Метод движения по игровому миру.

/// </summary>

/// <param name="input">Входной параметр.</param>

/// <returns></returns>

public Vector2 ToWorld(Vector2 input)

{

input /= (float)zoom;

Vector2 dX = new Vector2((float)Math.Cos(rotation), (float)Math.Sin(rotation));

Vector2 dY = new Vector2((float)Math.Cos(rotation + MathHelper.PiOver2), (float)Math.Sin(rotation + MathHelper.PiOver2));

return (this.position + dX \* input.X + dY \* input.Y);

}

/// <summary>

/// Конструктор камеры.

/// </summary>

/// <param name="startPosition">Начальная позиция.</param>

/// <param name="startRotation">Начальное значение поворота.</param>

/// <param name="startZoom">Начальное увеличение.</param>

public View(Vector2 startPosition, double startRotation = 0.0, double startZoom = 1.0)

{

this.position = startPosition;

this.rotation = startRotation;

this.zoom = startZoom;

}

/// <summary>

/// Метод обновления.

/// </summary>

public void Update()

{

if (currentStep < tweenSteps)

{

currentStep++;

switch (tweenType)

{

case TweenType.Linear:

position = positionFrom + (positionGoto - positionFrom) \*

GetLinear((float)currentStep / tweenSteps);

break;

case TweenType.QuadraticInOut:

position = positionFrom + (positionGoto - positionFrom) \*

GetQuadraticInOut((float)currentStep / tweenSteps);

break;

case TweenType.CubicInOut:

position = positionFrom + (positionGoto - positionFrom) \*

GetCubicInOut((float)currentStep / tweenSteps);

break;

case TweenType.QuarticOut:

position = positionFrom + (positionGoto - positionFrom) \*

GetQuarticOut((float)currentStep / tweenSteps);

break;

}

}

else

{

position = positionGoto;

}

}

/// <summary>

/// Набор позиций.

/// </summary>

/// <param name="newPosition">Новая позиция.</param>

public void SetPosition(Vector2 newPosition)

{

this.position = newPosition;

this.positionFrom = newPosition;

this.positionGoto = newPosition;

tweenType = TweenType.Instant;

currentStep = 0;

tweenSteps = 0;

}

/// <summary>

/// Набор позиций.

/// </summary>

/// <param name="newPosition">Новая позиция.</param>

/// <param name="type">Тип.</param>

/// <param name="numSteps">Номер шага.</param>

public void SetPosition(Vector2 newPosition, TweenType type, int numSteps)

{

this.positionFrom = position;

this.position = newPosition;

this.positionGoto = newPosition;

tweenType = type;

currentStep = 0;

tweenSteps = numSteps;

}

/// <summary>

/// Получение линейной функции.

/// </summary>

/// <param name="t">Переменная.</param>

/// <returns>t</returns>

public float GetLinear(float t)

{

return t;

}

/// <summary>

/// Получение квадратичной функции.

/// </summary>

/// <param name="t">Переменная.</param>

/// <returns>Квадратичная функция.</returns>

public float GetQuadraticInOut(float t)

{

return (t \* t) / ((2 \* t \* t) - (2 \* t) + 1);

}

/// <summary>

/// Получение кубической функции.

/// </summary>

/// <param name="t">Переменная.</param>

/// <returns>Кубическая функция.</returns>

public float GetCubicInOut(float t)

{

return (t \* t \* t) / ((3 \* t \* t) - (3 \* t) + 1);

}

/// <summary>

/// Получение суперкубической функции.

/// </summary>

/// <param name="t">Переменная.</param>

/// <returns>Суперкубическая функция.</returns>

public float GetQuarticOut(float t)

{

return -((t - 1) \* (t - 1) \* (t - 1) \* (t - 1)) + 1;

}

/// <summary>

/// Применение изменений.

/// </summary>

public void ApplyTransform()

{

Matrix4 transform = Matrix4.Identity;

transform = Matrix4.Mult(transform, Matrix4.CreateTranslation(-position.X, -position.Y, 0));

transform = Matrix4.Mult(transform, Matrix4.CreateRotationZ(-(float)rotation));

transform = Matrix4.Mult(transform, Matrix4.CreateScale((float)zoom, (float)zoom, 1.0f));

GL.MultMatrix(ref transform);

}

}

}