СОДЕРЖАНИЕ

ВВЕДЕНИЕ 5
1. ОСНОВНАЯ ЧАСТЬ 7
1.1. Исследование предметной области 7
1.2. ПЛАНИРОВАНИЕ ПРОЕКТА 8
1.2.1. Анализ требований 8
1.2.2. Анализ данных 8
1.2.3. Анализ методов и алгоритмов решений 8
1.2.4. Планирование 20
2. ПРОЕКТИРОВАНИЕ 23
3. ПРОГРАММИРОВАНИЕ 25
4. ТЕСТИРОВАНИЕ И ОТЛАДКА 27
5. ДОКУМЕНТИРОВАНИЕ 34
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 37
ПРИЛОЖЕНИЕ 1 37
ПРИЛОЖЕНИЕ 2 48
ПРИЛОЖЕНИЕ 3 52

ВВЕДЕНИЕ

Одним из важных потребностей в жизни человека является отдых. От качества отдыха зависит эффективность человека в его жизни. Человеческому организму требуется не только отдых в физическом плане, но и в моральном тоже. Что иное, если не игры, помогают современному человеку отдохнуть от повседневной рутины?

Развитие игровой индустрии в современном мире находится на высоком уровне. Многообразие игр с их захватывающим сюжетом способствуют привлечению всё большему числа игроков. Сами игры, выйдя в сеть, становятся не просто сферой отдыха и развлечения, но вырастает в отдельную ветку киберспорта.

Казалось бы, что сложного в этих играх? Но разработка больших игр занимает отнюдь не пару дней, как кто-нибудь мог бы предположить. И поэтому крупные проекты делаются далеко не пятью людьми. В команды разработчиков входят от 3D дизайнеров до программистов, и каждый из них выполняет свою задачу. Ну, а когда игру делает один человек, все эти обязанности складываются на его плечи.

Целью курсового проекта создание мобильной игры «I don’t know».

Ставятся следующие задачи:

- Проанализировать предметную область.
- Моделирование всех игровых объектов.
- Разработка и реализация игровых механик.
- Настройка UI элементов.
- Разработать программный продукт.
- Всяческая работа с анимациями.
- Работа с эффектами.
- Написание шейдеров.
- Написание Unit тестов
- Оптимизация кода.
- Рефакторинг кода.
- Исправление багов.

- Проведение тестирование и отладки.
- Разработать модульную структуру.
- Написание руководство пользователя.

1. ОСНОВНАЯ ЧАСТЬ

1.1. Исследование предметной области

Игровая индустрия разделятся на два лагеря. Это ААА игры и инди игры. Это как две стороны медали, которые вместе образуют индустрию. Но хоть ААА и инди игры образуют одну область. Они живут обособленно друг от друга, и имеют массу отличий. ААА проекты, разрабатываются огромными командами с отдельными штатами сотрудников, в одном таком штате могут числится от 100 человек разных специальностей. Если команда не справляется со сроками, к ним могут подключить дополнительные силы для реализации проекта в срок. Как правило, такие продукты выпускаются издателями, а не самими разработчиками.

Если говорить про инди игры, тут ситуация диаметрально противоположна. Такие продукты разрабатываются инди компаниями, как понятно из названия "Инди игры". Команда разработчиков в таких компаниях, как правило, на порядок меньше, бюджеты ниже, а может быть, что бюджета вовсе нет, и всё создание продукта держится на энтузиазме. Конечно, инди игры разрабатывают не только командами разработчиков, но также и отдельными личностями. Выпускаются на рынок такие игры под своим издательством. После выхода в Store, команду разработчиков ждёт три пути. Первый - это полный провал в продажах и как следствие, раскол команды. Так как работа на энтузиазме долго держаться не может. Разработчики уходят в другие более крупные проекты. Проекты, которые застали второй путь, о котором написано далее. Второй путь - игра привлекает пользователей, тем самым расширяя возможности команды разработчиков. После, разработчики приступают к новому проекту, с расширенными бюджетами и некой известностью среди геймерского сообщества. Эта известность, безусловно, увеличит шансы на успех будущего проекта. Третий - игру также постиг успех. Но командой заинтересовался крупный игрок на рынке. После чего разработчиков покупают либо образуют новый штат сотрудников или распределяют в уже существующие.

Но всё, что было сказано в предыдущем абзаце, актуально лишь для Desktop,

Pastgen и Nextgen. Ведь на рынке мобильных игр ситуация обстоит иначе. Множество игр на мобильных устройствах разрабатываются одним проектировщикам, так-как ресурсы создания таких игр на порядок меньше, чем на других устройствах. Не требуют таких строгих сроков, усилий, бюджета. Потому что игроки на мобильных устройствах не требуют глубину сюжета, разнообразие игровых механик, проработанности мира, передовой графики и т.д. Для игры не телефоне достаточно одной двух интересных механик и не графику уровня первого Doom.

1.2. Планирование проекта

1.2.1. Анализ требований

Требования к интерфейсу в проекте будут состоять из следующих пунктов:

- Компактность интерфейса.
- Понятность интерфейса.
- Сочетаемость интерфейса.
- Интегрируемость интерфейса в геймплей.
- Расширяемость интерфейса.
- Стилизованность интерфейса.

Минимальные системные требования:

- Количество ядер у процессора – 6;
- Частота работы процессора – 2.4;
- Объем оперативной памяти – 6.

1.2.2. Анализ данных

Входными данными, в игре, будет служить только имя пользователя, которое он будет вводить при первом входе. Выходными данными будет служить общая статистика игрока.

1.2.3. Анализ методов и алгоритмов решений

Во время разработки будет использоваться принцип проектирования SOLID. Он расшифровывается как:

– **S** (SRP) - Принцип единой ответственности. Обозначает, что каждый объект должен иметь одну ответственность и эта ответственность должна быть полностью инкапсулирована в класс. Все его поведения должны быть направлены исключительно на обеспечение этой ответственности.

– **O** (OCP) - Принцип открытости/закрытости. Устанавливает следующее положение: «программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для изменения»

– **L** (LSP) - Принцип подстановки Лисков. Роберт С. Мартин определил этот принцип так: Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом.

– **I** (ISP) - Принцип разделения интерфейса. Говорит о том, что слишком «толстые» интерфейсы необходимо разделять на более маленькие и специфические, чтобы программные сущности маленьких интерфейсов знали только о методах, которые необходимы им в работе. В итоге, при изменении метода интерфейса не должны меняться программные сущности, которые этот метод не используют.

– **D** (DIP) – Принцип инверсии зависимости. Модули верхних уровней не должны импортировать сущности из модулей нижнего уровня. Оба типа модулей должны зависеть от абстракций.

Также будет использоваться принцип KISS («Keep it simple, stupid»)

Для разработчика формулировка звучит следующем образом:

– Разбивайте задачи на подзадачи, которые не должны, по вашему мнению, длиться более 4-12 часов написания кода

– Разбивайте задачу на множество более маленьких задач, каждая задача должна решаться одним или парой классов

– Сохраняйте ваши методы маленькими. Каждый метод должен состоять не более чем из 30-40 строк. Каждый метод должен решать одну маленькую задачу, а не множество случаев. Если в вашем методе множество условий, разбейте его на несколько. Это повысит читаемость, позволит легче поддерживать код и быстрее находить ошибки в нём. Вы полюбите улучшать код.

– Сохраняйте ваши классы маленькими. Здесь применяется та же техника, что и с методами.

– Сначала придумайте решение задачи, потом напишите код. Никогда не поступайте иначе. Многие разработчики придумывают решение задачи во время написания кода, и в этом нет ничего плохого. Вы можете делать так и при этом придерживаться выше обозначенного правила. Если вы можете в уме разбивать задачу на более мелкие части, когда вы пишете код, делайте это любыми способами. И не бойтесь переписывать код ещё, ещё и ещё… В счёт не идёт число строк, до тех пор, пока вы считаете, что можно ещё меньше/ещё лучше.

– Не бойтесь избавляться от кода. Изменение старого кода и написание нового решения — два важных момента. Если вы столкнулись с новыми требованиями, или не были оповещены о них ранее, тогда порой лучше придумать новое, более изящное решение, решающее и старые, и новые задачи.

И также будет эксплуатироваться принцип Yagni.

Yagni - процесс и принцип проектирования ПО, при котором в качестве основной цели и/или ценности декларируется отказ от избыточной функциональности, — то есть отказ добавления функциональности, в которой нет непосредственной надобности.

В основе принципа YAGNI лежит несколько наблюдений:

1. Программисты, как и люди в целом, плохо предсказывают будущее.

2. Затраты, сделанные сейчас могут быть оправданными или неоправданными в будущем.

3. Ни одно гибкое решение не будет достаточно гибким.

Использованные паттерны

Были использованы паттерны программирования такие как:

– **Одиночка** (Singleton, Синглтон) - порождающий паттерн, который гарантирует, что для определенного класса будет создан только один объект, а также предоставит к этому объекту точку доступа. Но на деле этот паттерн приносит больше вреда, чем пользы. И хотя авторы предупреждают, что использовать пат-

терн надо осторожно, их послание часто теряют, когда послание касается индустрии игр. Любой паттерн, если его применять не там, где нужно, принесёт столько же пользы, сколько наложение гипса при огнестрельном ранении.

В игре может быть класс, который сохраняет игру. Очень важно, чтобы у вас был только один экземпляр этого класса, иначе можно сохранить разные версии игры, если каждый экземпляр содержит разные данные. Также должно быть легко получить доступ к этому классу сохранения игры оттуда, где он нужен. Для этого можно использовать паттерн Singleton.

И согласно книге Game Programming Patterns (Паттерны Программирования Игр), нужно избегать этого шаблона, потому что глобальные объекты могут вызвать проблемы. Если нужно использовать этот шаблон, то он должен быть для классов менеджеров, таких как GameController, SaveGame и т. Д.

Пример использования в коде:

SceneTransition.cs

```
private static SceneTransition instance;
private static bool shouldPlayOpeningAnimation = false;

private Animator componentAnimator;
private AsyncOperation loadingSceneOperation;

public static void SwitchToScene(string sceneName)
{
    instance.componentAnimator.SetTrigger("sceneClosing");

    instance.loadingSceneOperation = SceneManager.LoadSceneAsync(sceneName);

    instance.loadingSceneOperation.allowSceneActivation = false;

    instance.LoadingProgressBar.fillAmount = 0;
}

private void Start()
{
    instance = this;

    componentAnimator = GetComponent<Animator>();
```

```
if (shouldPlayOpeningAnimation)
{
    componentAnimator.SetTrigger("sceneOpening");
    instance.LoadingProgressBar.fillAmount = 1;

    shouldPlayOpeningAnimation = false;
}
```

Класс SceneTransition.cs представляет собой переход между сценами Unity, и вызывается из разных скриптов. Паттерн Синглтон обеспечивает надёжность этих переходов.

– **Состояние** (State) – шаблон проектирования, который позволяет объекту изменять своё поведение в зависимости от внутреннего состояния.

Игра может находиться в нескольких состояниях. Например, у главного героя могут быть следующие состояния: прыжок, ходьба, бег и т. д. Теперь вам нужен простой способ переключения между состояниями. Этот шаблон также известен как конечный автомат, если у вас есть конечное количество состояний, вы получаете конечный автомат (FSM).

Как реализовать?

- Можно использовать перечисление, которое отслеживает каждое состояние, а затем оператор переключения.
- Проблема с оператором switch заключается в том, что чем больше состояний добавляется, тем сложнее он становится. Лучший способ — определить объект для каждого состояния, а затем переключаться между объектами при переключении состояний.

Когда это полезно?

- Когда слишком много вложенных операторов if, например, в системе меню. В коде можно увидеть пример системы меню, использующей этот шаблон.
- Unity использует этот шаблон в движке анимации.
- Если нужно сделать игру в стиле GTA. Есть одно состояние для вождения, одно для того, когда персонаж не находится в транспортном средстве, дру-

гое состояние для полета и т. д. Затем также можно добавить состояния состояний. Например, в классе состояний, где персонаж не находится в транспортном средстве, может быть несколько подсостояний, таких как ничего не держать, держать гранату, держать пистолет и т. д.

Пример использования в коде:

MenuController.cs:

```
using System.Collections.Generic;
using UnityEngine;

namespace State.Menu
{
   public class MenuController : MonoBehaviour
   {
      public _MenuState[] allMenus;

      public enum MenuState
      {
         Game, Main, Settings
      }


      private Dictionary<MenuState, _MenuState> menuDictionary = new Diction-
ary<MenuState, _MenuState>();

      private _MenuState _activeState;

      private Stack<MenuState> stateHistory = new Stack<MenuState>();

      void Start()
      {
         foreach (_MenuState menu in allMenus)
         {
            if (menu == null)
            {
               continue;
            }

            menu.InitState(menuController: this);

            if (menuDictionary.ContainsKey(menu.state))
```

```
            {
                Debug.LogWarning($"The key <b>{menu.state}</b> already exists in the menu dictionary!");

                continue;
            }

            menuDictionary.Add(menu.state, menu);
        }

        foreach (MenuState state in menuDictionary.Keys)
        {
            menuDictionary[state].gameObject.SetActive(false);
        }

        SetActiveState(MenuState.Game);
    }

    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Escape))
        {
            _activeState.JumpBack();
        }
    }

    public void JumpBack()
    {
        if (stateHistory.Count <= 1)
        {
            SetActiveState(MenuState.Main);
        }
        else
        {
            stateHistory.Pop();

            SetActiveState(stateHistory.Peek(), isJumpingBack: true);
        }
    }

    public void SetActiveState(MenuState newState, bool isJumpingBack = false)
    {
        if (!menuDictionary.ContainsKey(newState))
```

```
        {
            Debug.LogWarning($"The key <b>{newState}</b> doesn't exist so you can't
activate the menu!");

            return;
        }

        if (_activeState != null)
        {
            _activeState.gameObject.SetActive(false);
        }

        _activeState = menuDictionary[newState];

        _activeState.gameObject.SetActive(true);

        if (!isJumpingBack)
        {
            stateHistory.Push(newState);
        }
    }

    public void QuitGame()
    {
        Debug.Log("You quit game!");

        Application.Quit();
    }
  }
}
```

_MenuState.cs

```
using UnityEngine;

namespace State.Menu
{
  public class _MenuState : MonoBehaviour
  {
    public MenuController.MenuState state { get; protected set; }

    protected MenuController menuController;

    public virtual void InitState(MenuController menuController)
```

```
        {
            this.menuController = menuController;
        }

        public void JumpBack()
        {
            menuController.JumpBack();
        }
    }
}
```

MainMenu.cs

```
namespace State.Menu
{
    public class MainMenu : _MenuState
    {
        public override void InitState(MenuController menuController)
        {
            base.InitState(menuController);

            state = MenuController.MenuState.Main;
        }

        public void JumpToSettings()
        {
            menuController.SetActiveState(MenuController.MenuState.Settings);
        }

        public void QuitGame()
        {
            menuController.QuitGame();
        }
    }
}
```

SettingsMenu.cs

```
namespace State.Menu
{
    public class SettingsMenu : _MenuState
    {
        public override void InitState(MenuController menuController)
        {
            base.InitState(menuController);
```

```
            state = MenuController.MenuState.Settings;
        }
    }
}
```

Класс MenuController манипулирует состояниями, а именно в методе SetActiveState (MenuState newState, bool isJumpingBack = false) переключает разные разделы меню. Метод QuitGame () реализует выход из игры. Метод JumpBack () активирует предыдущее состояние.

Базовый класс состояния _MenuState имеет ссылку на MenuController. Виртуальный метод InitState (MenuController menuController) инициализирует передаваемый ему в параметре MenuController.

От класса _MenuState наследуются классы разных состояний меню (рис. 1).

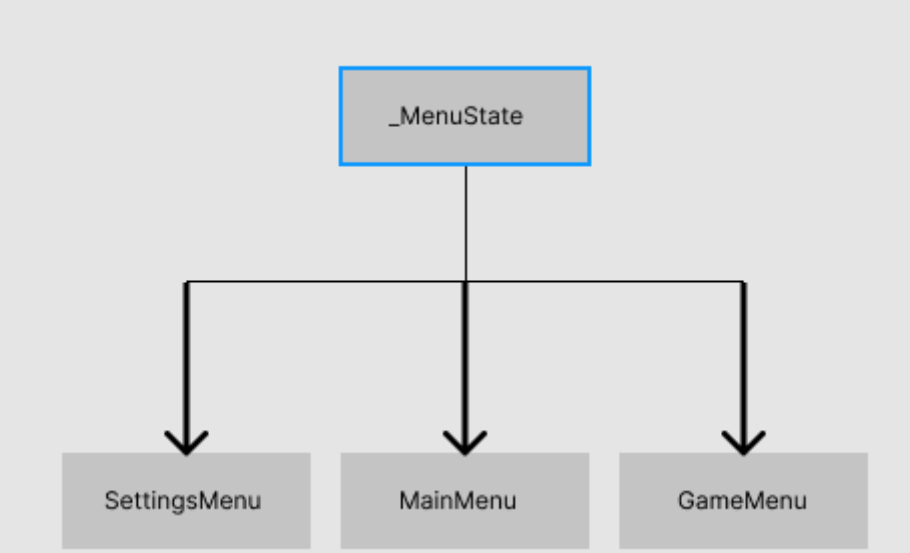

Рисунок 1 – Иерархия классов состояний.

Вот почему для проектирования меню был выбран паттерн State. Функционал меню можно в любом промежутке времен расширить, не нарушая общей семантики кода, и так же легко добавить новое состояние.

– **Наблюдатель** (Observer) - определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом и автоматически обновляются. Другими словами: наблюдатель уведомляет все заинтересованные стороны о произошедшем событии или об изменении своего состояния.

В игре постоянно происходит множество вещей. Эти вещи называются событиями (а иногда и сообщениями). Разница между событием и сообщением заключается в том, что событие произошло, а сообщение — это то, что произойдет. Таким образом, этот паттерн полностью описывает то, что произойдет после того, как событие произошло. Какие методы следует вызывать после того, как например был убит враг, чтобы обновить счет, показать анимацию смерти и т. д.? Эти методы должны подписаться на событие.

Этот шаблон настолько популярен, что разработчики C# реализовали его в языке. У Unity тоже есть своя реализация. Итак, есть множество альтернатив при использовании этого паттерна:

EventHandler

Action

UnityEvent

Это так же можно реализовать с использованием делегата.

Этот паттерн действительно полезен, если нужно избежать спагетти-кода, делая классы независимыми друг от друга, что также известно, как развязка. Лучшая часть событий заключается в том, что часть, запускающая событие, не заботится о том, какие методы присоединены к событию. Там может быть ноль методов. Поэтому, если событие срабатывает, но ничего не происходит, будет проще найти, где может быть ошибка.

Другой способ отделить код — сделать событие статическим.

В игре “I don’t know” есть множество реализаций этого паттерна, вот одна из них:

InventoryUIChannel:

```
using UnityEngine;

[CreateAssetMenu(menuName = "ScriptableObject/Inventory/InventoryUIChannel")]
public class InventoryUIChannel : ScriptableObject
{
    public delegate void InventoryToggleCallback(InventoryHolder inventoryHolder);
    public InventoryToggleCallback OnInventoryToggle;
```

```
    public void RaiseToggle(InventoryHolder inventoryHolder)
    {
        OnInventoryToggle?.Invoke(inventoryHolder);
    }
}
```

StatUIController:

```
SerializeField] private InventoryUIChannel InventoryUIChannel;
[SerializeField] private StatUIData[] m_Stats;

private Inventory m_DisplayedEquippedItemsInventory;

private void Awake()
{
    InventoryUIChannel.OnInventoryToggle += OnInventoryToggle;
    gameObject.SetActive(false);
}

private void OnDestroy()
{
    InventoryUIChannel.OnInventoryToggle -= OnInventoryToggle;
}

private void OnInventoryToggle(InventoryHolder inventoryHolder)
{
    Inventory equippedItemInventory = inventoryHolder.GetCompo-
nent<EquippedItemsHolder>().EquippedItemsInventory;

    if (m_DisplayedEquippedItemsInventory == null)
    {
        gameObject.SetActive(true);
        m_DisplayedEquippedItemsInventory = equippedItemInventory;
    }
    else if (m_DisplayedEquippedItemsInventory == equippedItemInventory)
    {
        gameObject.SetActive(false);
        m_DisplayedEquippedItemsInventory = null;
    }
}
```

Класс InventoryUIChannel наследуется от ScriptableObject.

ScriptableObject – В Unity работа происходит с объектами, такими как персонаж, стена, UI элементы, и вообще всё что угодно, всё это объекты, на которых

висят компоненты. Компоненты в свою очередь это либо Unity компоненты, либо собственные скрипты, хоть и компоненты Unity тоже являются скриптами. Но скрипт, который наследуется от ScriptableObject не является компонентом. Это обособленный объект, который может хранить в себе данные и выполнять некоторую логику.

То есть InventoryUIChannel это ScriptableObject, в котором объявлен колбэк, колбэк который вызывается из метода RaiseToggle (InventoryHolder inventory-Holder), который в свою очередь вызывается через обработчика событий (рис. 2).

Рисунок 2 – Обработчик событий Unity

После вызывания OnInventoryToggle, все слушатели такие как StatUIControl-ler начинают отрисóвывать свои части ответственности такие как инвентарь и т.д. Все слушатели (рис. 3):

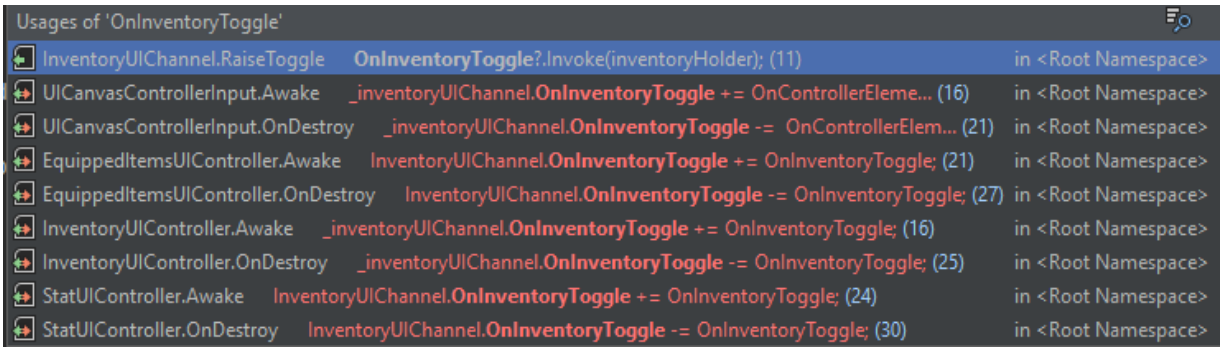

Рисунок 3 – Все наблюдатели

Это лишь одна из реализаций, но она является самой наглядной.

1.2.4. Планирование

Сюжет игры будет разворачиваться в Канаде, район в 100 километрах от поселения Бейкер-Лейк, Нунавут. Во время суровой зимы. Вы как геолог, попадаете в эти непростые условия. Изначальной задачей вашей экспедиции был Голубой гранат, цена которого составляет более чем 2 млн. долларов. Но внезапная смена погоды прерывает ваши планы. Вертолёт, на котором вы летели потерпел крушение, и вся группа гибнет, кроме вас. Единственной вашей целью после

этого является выживание. Это будет не самой лёгкой задачей, ведь район где было совершенно крушение заброшен. Высокие холода заставили жителей этих мест покинуть свои дома. В игре так же будет режим простого выживания, без сюжета.

Модульная структура представляет собой представление программы в модулях (рис. 4).

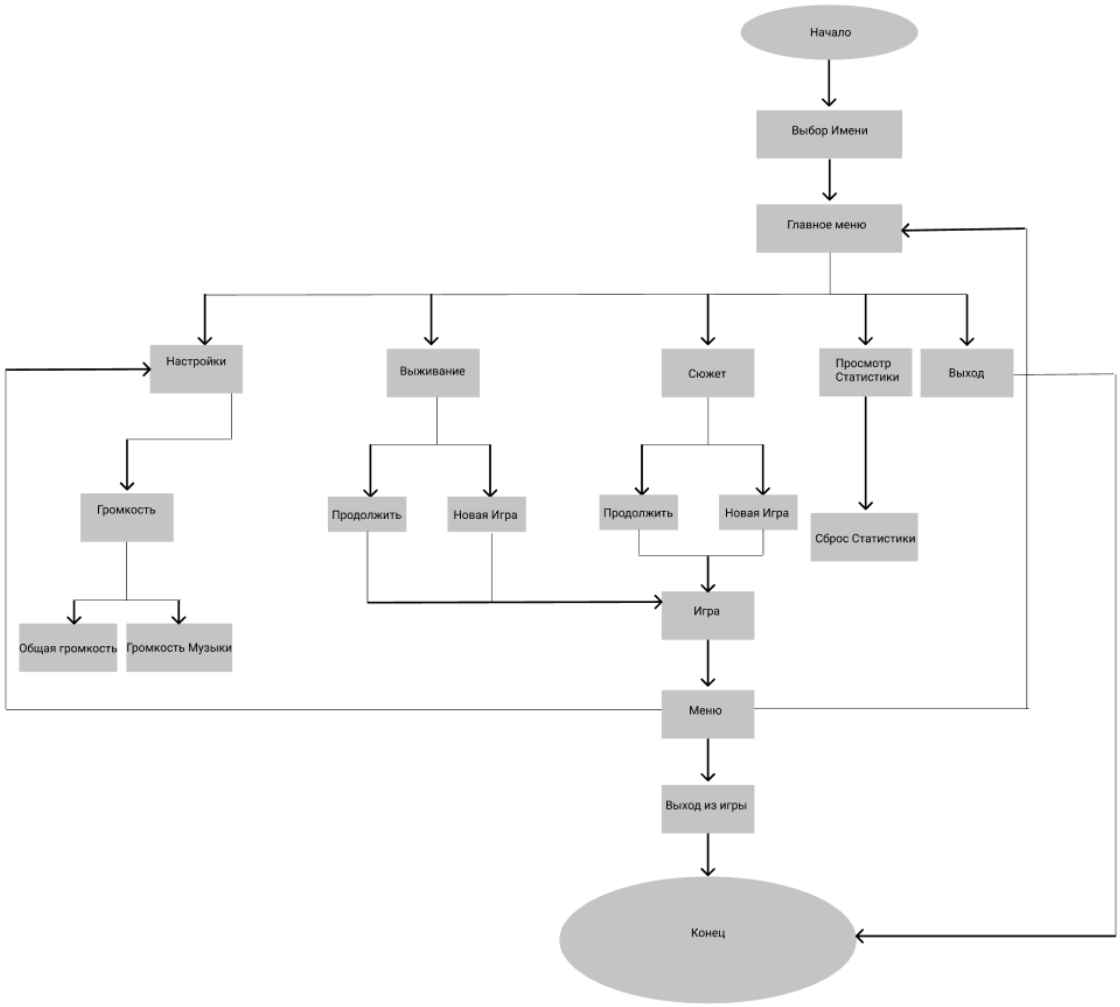

Рисунок 4 - Модульная структура Игры

Макет интерфейса — это документ, содержащий в себе информацию о том, как в перспективе будет выглядеть интерфейс игры (рис. 5-8).

Для чего можно использовать макеты

1) Обсуждение функциональности с заказчиком. Заказчики, как правило, люди очень занятые и зачастую физически не способные выделить время на чтение многостраничных документов (даже если это в их интересах). В этом смысле картинка стоит тысячи слов: она гарантирует, что у заказчика появится

четкое представление о том, что именно будет сделано.

2) Оценка юзабилити. Макеты – это фактически первый артефакт в проекте, юзабилити которого уже можно и нужно оценивать. На этом этапе проще и дешевле всего устранить проблемы, если они обнаружатся.

3) Постановка задачи разработчикам. Очень маленький процент разработчиков вчитывается в спецификации. Наличие наглядной иллюстрации того, что должно быть сделано, гораздо полезнее, чем длинное текстовое описание (хотя, разумеется, оно также должно присутствовать, ибо не все можно отобразить на макете).

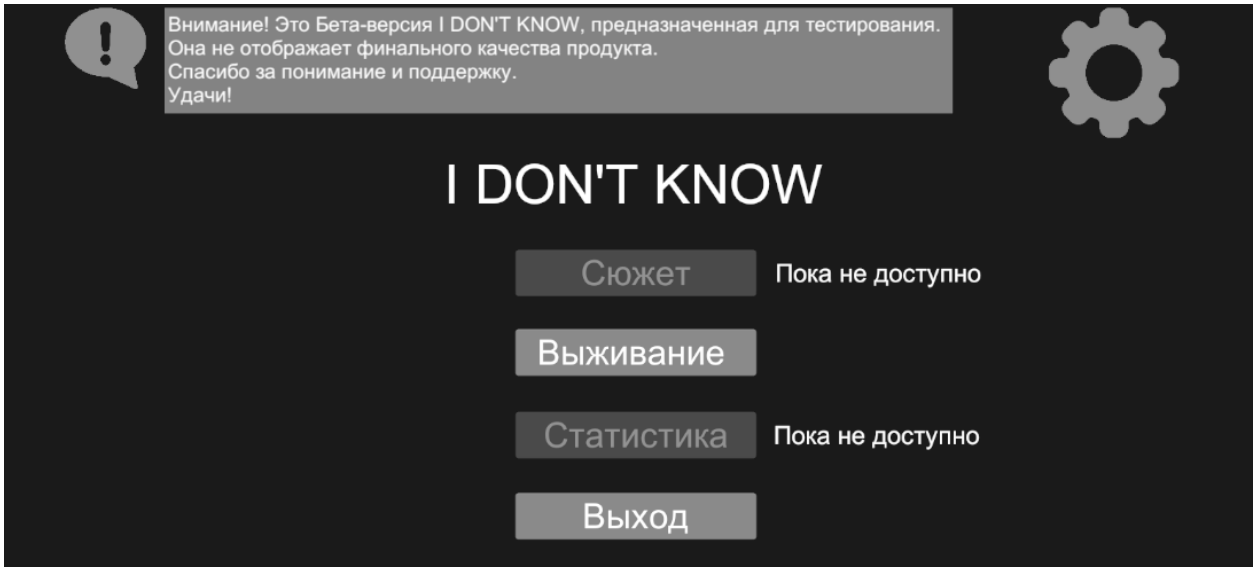

Рисунок 5 – Главное меню

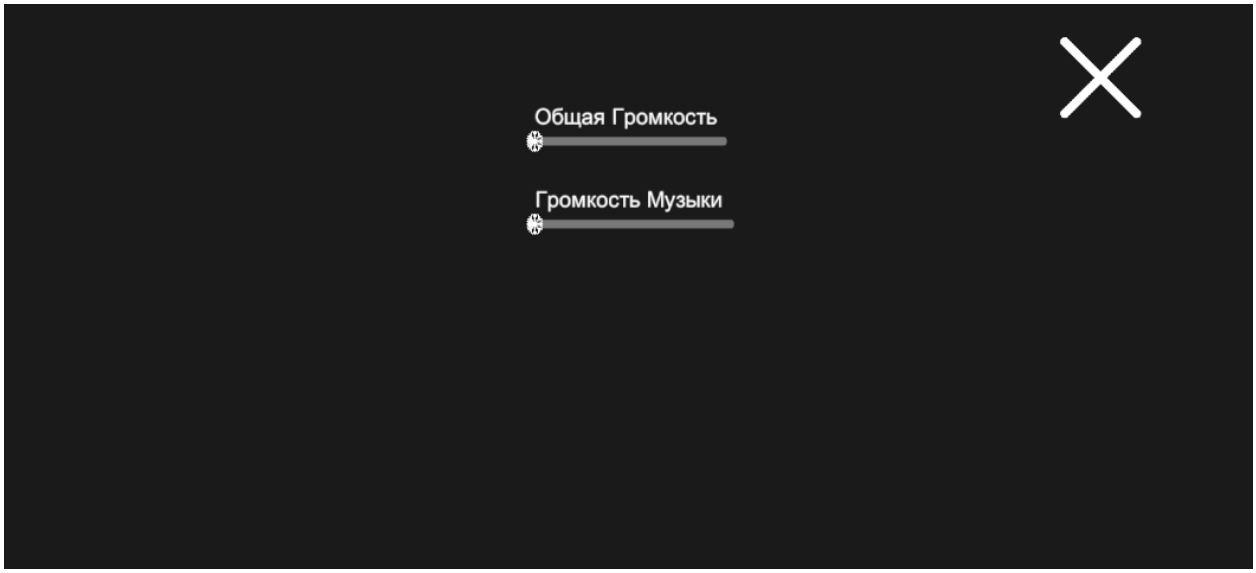

Рисунок 6 – Настройки

Рисунок 7 – Игровой интерфейс

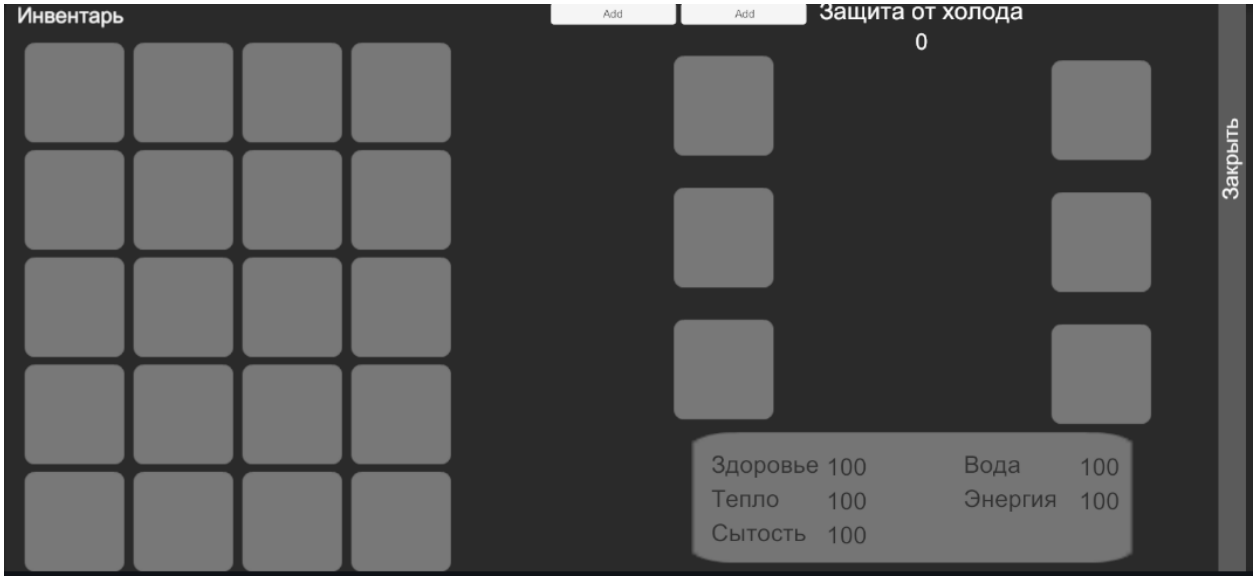

Рисунок 8 –Инвентарь

Выводом этого раздела может являться то, что суть планирования заключается в продумывании и организации проект. Также оно помогает максимально тщательно и быстро выполнять поставленные задачи.

2. ПРОЕКТИРОВАНИЕ

Такие понятия, как архитектура, проектирование и пр., ассоциируются в первую очередь с отраслью строительства, т.е. созданием сооружений. Эти термины используются для того, чтобы, с одной стороны, кратко, а с другой – полно

выразить ощущения от восприятия конкретных сооружений: внешний вид, внутренняя обстановка, сочетание с окружающим ландшафтом и пр.

С программным обеспечением дело обстоит примерно так же. Под архитектурой и предваряющей ее стадией проектирования понимаются все те необходимые параметры, которые предопределяют создаваемые программные продукты.

"Для того чтобы уберечь бизнес от дезинтеграции, концепция создания архитектуры информационных систем перестает быть просто одной из возможных опций, а становится настоятельной необходимостью".

Дж. Захман

Для специалиста в области разработки программного обеспечения стадия проектирования и используемый *инструментарий* означают намного больше. Профессионал сразу сможет оценить все идеи и наработки, используемые для достижения необходимого результата. Также немаловажным фактором является и *личность* самого проектировщика.

В архитектуре информационных систем, так же, как и в классической архитектуре, состав, структура и назначение компонент информационной системы, ее реальный эффект в том числе определяется субъективными аспектами восприятия со стороны конечных пользователей. Проработанный интерфейс, поддержка и развитие реально используемых функциональных возможностей создадут предпосылки для успеха и обеспечат его организации, эксплуатирующей информационную систему. Устаревшие приложения и архаичные технические средства станут тормозом продуктивной работы. Они должны будут разделить судьбу сооружений, канувших в лету обломков и руин.

Игра "i don't know" будет реализована на игровом движке Unity5 3D. Все скрипты будут написаны на языке C#. Для анимирования и моделирования будут использованы такие программы, как Blender 3D и Magic Voxel. Для текстурирования, Adobe Substance Painter 3D. В качестве IDE будет использован Rider. Контроль версий будет происходить с помощью Git и TortoiseGit.

Одной из самых важных задач в проектировании - это создание правильной архитектуры программного кода.

3. ПРОГРАММИРОВАНИЕ

Программирование игр — часть процесса разработки компьютерных игр (видеоигр). Программирование игр требует специализации в одной или нескольких из следующих областей, которые в значительной степени присутствуют в создании игр: симуляция, компьютерная графика, искусственный интеллект, физика, звук и ввод данных. Для многопользовательских онлайн-игр часто необходимы дополнительные знания, такие как сетевое программирование и программирование баз данных

Прототипирование игр — процесс реализации базового функционала для чернового варианта. Обязательность прототипирования обусловлена необходимостью анализа системы в целом. Для игровой индустрии прототипом называется демоверсия с базовым функционалом (набор ключевых особенностей игры). Сколько базовых особенностей будет включено в демоверсию решается возможностями бюджета и важностью этих вещей в игровом процессе.

Есть множество игровых движков такие как: Unity, Unreal Engine, CryEngine и ещё множество других. Лидирующее место в программировании игр занимают два языка это C++ и C#. Но C++ подходит для создания игр чуть больше, чем его конкурент т.к. является языком более низкого уровня и умеет работать с железом на прямую. У этого есть и свое плюсы и свои минусы. Например, C++ лучше работает с памятью, но там ей нужно управлять самостоятельно, конечно, у этого есть свои преимущества ведь память будет задействоваться более органично нежели на C# но минусам этого является не редкое явление как утечка памяти. В C# же всем этим занимается Garbage Collector (Сборщик Мусора) – он самостоятельно отлавливает не используемые объекты в памяти, и за ними не нужно следить самостоятельно. Но у этого есть свой минус, GC работает самостоятельно и это не лучшем образом играет на использовании оперативной памяти ПК, а также сама работа GC требует некоторых ресурсов. Можно сделать вывод что каждый язык хорош в чём-то своём.

Unity так же поддерживает скрипты на JavaScript, но он сильно уступает C#

по всём параметрам. Хотя стоит признать, что Unity на самом деле не поддерживает "реальный' JavaScript, а вместо этого поддерживает JavaScript-подобный язык UnityScript. UnityScript похож на JavaScript во многих отношениях, но также имеет некоторые существенные изменения. Самое главное, UnityScript поддерживает статические типизированные объекты. Но всё же C# намного легче в поддержке кода и документации для него на порядок больше, также, как и обсуждений на официальном форуме Unity. Ну и конечно C# лучше взаимодействует с платформой MONO.

Что на счёт игровых движков? Сейчас движков с бесплатным типом продвижения достаточно много. Самые выделяющееся это Unity, Unreal Engine, CryEngine. Примерно до 2018 года все эти движки использовались для разных целей. Тогда Unity был движком для мелких по сравнению с его конкурентами игр. Но в 2018 году разработчики Unity начали развивать и дополнять свой движок. И вот в 2022 году этот движок не уступает уровню игр от своих конкурентов, и даже наоборот обходит, например, работа с анимациями в Unity намного удобнее и технологичней чем у его конкурентов. И именно из всех преимуществ, которые даёт Unity, был выбран этот движок.

Все шейдеры будут написаны на языке HLSL (High-level shader language). HLSL — это C-подобный язык шейдеров высокого уровня, который используется с программируемыми шейдерами в DirectX. Например, можно использовать HLSL для написания вершинного шейдера или пиксельного шейдера и использовать эти шейдеры в реализации средства визуализации в приложении Direct3D. С помощью шейдеров в графическом плане можно создать всё, что угодно.

Для написания скриптов была выбрана IDE Rider. **JetBrains Rider** (рис. 9) — быстрый и мощный редактор C# для Unity, который работает на Windows, Mac и Linux. Непревзойденное количество интеллектуальных механизмов инспекции кода и рефакторинга (более 2500), включенных в Rider, позволяют писать безошибочный код на C# гораздо быстрее. Из преимуществ Rider перед Visual Studio можно отметить. В Rider уже встроена поддержка Unity, лучшая поддержка Git. В общем Rider намного лучше работает с Unity чем Visual Studio. У

Rider есть огромное количество функций, унаследованных от платформы IntelliJ:

Поддержка систем контроля версий: помимо Git и Mercurial, Rider работает с CVS и Subversion. Интеграция VSTS доступна через специальный плагин, поддерживаемый компанией Microsoft.

Rider (с помошью DataGrip) поддерживает подключение к базам данных и SQL. Пользователям Visual Studio в большинстве случаев нужно будет использовать ODBC.

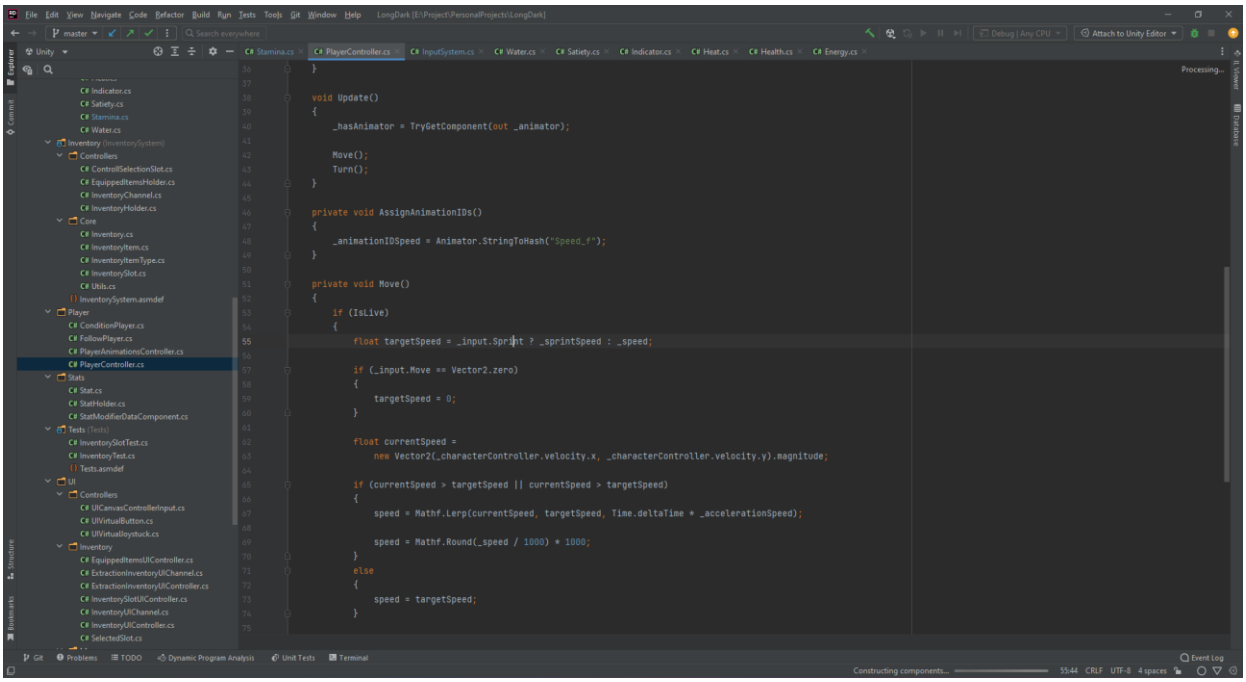

Рисунок 9 – Интерфейс Rider

Вывод: языка программирования - это лишь инструмент, у которого в сравнении с другими есть свои плюсы и минусы.

4. ТЕСТИРОВАНИЕ И ОТЛАДКА

Тестирование программного обеспечения — это метод проверки соответствия фактического программного продукта ожидаемым требованиям и обеспечения отсутствия дефектов программного продукта. Оно включает в себя выполнение компонентов программного обеспечения/системы с использованием ручных или автоматизированных инструментов для оценки одного или нескольких свойств, представляющих интерес. Целью тестирования программного обеспечения является выявление ошибок, пробелов или отсутствующих требований в отличие от фактических требований.

Большинство мероприятий, связанных с тестированием игр или QA (обеспечением качества), делятся на 2 категории – тестирование черного ящика и тестирование белого ящика.

Тестирование черного ящика - наиболее доступная и объективно нейтральная форма тестирования. По сути, тестер практически не имеет предварительных знаний об игре и тестирует различные ее аспекты с точки зрения пользователя. Тестировщику игр с черным ящиком не нужно знать программирование или понимать архитектуру игры, они просто фокусируются на том, что пользователи могут видеть и испытывать.

Напротив, **тестирование белого ящика** очень точное и предназначено для того, чтобы способствовать функциональным улучшениям игры, которые могут быть даже не заметны обычному игроку. Поскольку это требует глубоких знаний игровой архитектуры и кодирования, этот подход в основном используется самими разработчиками до того, как программное обеспечение будет представлено для дальнейшего рассмотрения командой QA.

Есть так же методы тестирования, такие как:

Тестирование функциональности игры.

Функциональное тестирование — это деятельность, направленная на определение того, работает ли игра в соответствии со спецификациями, и выявление любых ошибок или проблем, которые могут негативно повлиять на опыт игрока. Это относится как к тестированию мобильных игр, так и к другим платформам, включая ПК, консоль, веб и VR / AR. Можно сортировать тесты на несколько подтипов:

- **Тестирование игр на совместимость** — Используется для проверки того, как игра взаимодействует с другими приложениями. Тестеры ищут любые проблемы совместимости, включая недоступность функций, низкую производительность и задержки в общении. В основном используется для многопользовательских игр и тех, которые используют расширенные функции устройства (например, восходящий интернет, Bluetooth, камера и т.д.).
- **Регрессионное игровое тестирование** — Этот цикл применяется после

основных обновлений кода, чтобы убедиться, что обновление не повлияло отрицательно на существующую функциональность (например, нарушение функции, добавление многочисленных новых ошибок и т. Д.). Цель состоит в том, чтобы убедиться, что код все еще работает. Это очень распространенная практика как во время разработки (когда создаются новые сборки), так и в период после запуска, когда осуществляется техническое обслуживание и обновления. Многие из этих тестов можно автоматизировать.

– **Локализация тестирования игр** — Этот метод предназначен для того, чтобы игра была полностью полезной и приятной для игроков в разных странах и регионах. Во-первых, все одни и те же функции и функциональные возможности должны быть доступны в разных местах (если не запланировано иное). Во-вторых, содержание должно быть адаптировано к тем языкам и культурам, на которых оно представлено.

– **Тестирование игры контроля доступа безопасности** — Очень важная практика, которая проверяет, существуют ли какие-либо лазейки или несанкционированные шлюзы, которые могут позволить кому-либо получить доступ к игровому бэкэнду или элементам / функциям, которые в противном случае ограничены. Например, некоторые игроки пытаются взломать игры, чтобы получить бесплатные внутриигровые награды или валюту, сделать себя непобедимыми и т.д., а контроль доступа предотвращает это или, по крайней мере, снижает вероятность.

– **Тестирование игр для пользователей** — Это один из последних этапов тестирования перед выпуском игры в производство. Иногда он проводится как бета-тест, привлекая игроков вне команды разработчиков. Большие проблемы встречаются редко, поэтому основное внимание уделяется исправлению любых оставшихся незначительных ошибок и небольшим улучшениям пользовательского опыта.

Нефункциональное игровое тестирование.

Эта категория обеспечения качества охватывает аспекты игры, к которым

функциональное тестирование не относится, такие как оценка производительности, масштабируемости и удобства использования. Можно использовать автоматическое тестирование игр в большинстве ситуаций, но лишь немногие из них требуют творческого подхода и большого количества ручных манипуляций. Есть следующие разновидности:

– **Тестирование производительности игры** — Как следует из названия, этот подход оценивает, насколько хорошо работает игра, в таких аспектах, как потребление батареи, использование памяти и частота кадров. Он также может оценить время отклика клиента / сервера и то, как игра функционирует с различными уровнями сетевого подключения.

– **Нагрузочное тестирование игры** — Эта практика в чем-то похожа на тестирование производительности, но цель несколько иная. Тестировщики пытаются довести игру до предела и записывают, когда что-то начинает идти не так (сбои, низкая производительность и т.д.). Часто тестировщики имитируют большое количество игроков, использующих игру одновременно, чтобы увидеть, со сколькими может справиться сервер.

– **Объемное тестирование игр** — Тестирование объема очень похоже на нагрузочное тестирование, но оно оценивает не реакцию сервера на соединения, а передачу данных и производительность базы данных. Он проверяет события потери данных и определяет обстоятельства, которые их вызвали.

Тестирование, это очень важный этап разработки игр. Ведь на этапе тестирования решается на сколько конечный продукт на сколько игра будет стабильна и играбельна.

Игра на всей стадии разработки подвергалась нагрузочному тестированию, тестированию для пользователей, регрессивному тестированию, тестированию чёрного и белого ящика, так же были написаны Unit тесты.

Unit тесты:

Таблица 1 - Тестирование отдельного слота в инвентаре.

Название проекта	I don’t know
Номер версии	0.0.0.14
Имя тестера	Иван
Даты тестирования	23.04.2022

продолжение Таблицы 1

Test Case #	TC_IST_1
Приоритет теста	Высокий
Название тестирования/Имя	Тестирование отдельного слота в инвентаре_ StoreAndClearSlot()
Резюме испытания	Был написан Unit тест для слота в инвентаре, который проверяет сложение в пустой слот и его отчистки.
Шаги тестирования	1. Написание автономного теста 2. Запуск тестов при внедрении нового функционала, связанного с инвентарём
Данные тестирования	В слот переместили testItem в количестве 10, затем удаление слота.
Ожидаемый результат	После создания в слоте должен находиться testItem в размере 10 единиц, после удаления в слоте итем ровняется null, количество 0.
Фактический результат	Фактический результата является ожидаемым результатом.
Предпосылки	
Постусловия	Улучшение общего вида системы
Статус (Pass/Fail)	Pass
Комментарии	

Таблица 2 - Тестирование отдельного слота в инвентаре.

Название проекта	I don’t know
Номер версии	0.0.0.14
Имя тестера	Иван
Даты тестирования	23.04.2022
Test Case #	TC_IST_2
Приоритет теста	Высокий
Название тестирования/Имя	Тестирование отдельного слота в инвентаре_ MoveToEmptySlot ()
Резюме испытания	Был написан Unit тест для слота в инвентаре, который проверяет сложение перемежение в пустой слот.
Шаги тестирования	1. Написание автономного теста 2. Запуск тестов при внедрении нового функционала, связанного с инвентарём
Данные тестирования	Добавление к slotSource testItem в количестве 10, перемещение из slotSource в slotDestination в размере 4 единиц.
Ожидаемый результат	После перемежения в slotSource должен остаться итем в размере 6 единиц. В slotDestinaton должен переместиться итем из slotSource в количестве 4.
Фактический результат	Фактический результата является ожидаемым результатом.
Предпосылки	
Постусловия	Улучшение общего вида системы
Статус (Pass/Fail)	Pass
Комментарии	

Таблица 3 - Тестирование отдельного слота в инвентаре на перемещение.

Название проекта	I don’t know
Номер версии	0.0.0.14
Имя тестера	Иван

Даты тестирования	23.04.2022
Test Case #	TC_IST_3
Приоритет теста	Высокий
Название тестирования/Имя	Тестирование отдельного слота в инвентаре_ MoveAdd()
Резюме испытания	Был написан Unit тест для слота в инвентаре, а именно проверки перемещения из одного слота в другой всех итемов.
Шаги тестирования	1. Написание автономного теста 2. Запуск тестов при внедрении нового функционала, связанного с инвентарём
Данные тестирования	В slotSource был добавлен testItem в размере 2. В slotDestination так же был добавлен итем типа testItem в размере 4.
Ожидаемый результат	После перемещения всего в slotDestination, в slotSource не должно остаться итема, в slotDestination должен быть testItem в размере 6.
Фактический результат	Фактический результата является ожидаемым результатом.
Предпосылки	
Постусловия	Улучшение общего вида системы
Статус (Pass/Fail)	Pass
Комментарии	

После написания тестов прослеживалось улучшение в скорости добавления нового функционала.

Ручное тестирование:

Таблица 1 – Тест производительности.

Название проекта	I don’t know
Номер версии	0.0.0.1-0.0.0.14
Имя тестера	Иван
Даты тестирования	Тестирование производилась на протяжении всего этапа проектирования
Test Case #	TC_FR_1
Приоритет теста	Средний
Название тестирования/Имя	Тестирование производительности
Резюме испытания	Производилась разная нагрузка на систему.
Шаги тестирования	3. Запуск игры с минимальной нагрузкой. 4. Запуск игры со средней загрузкой 5. Запуск игры с максимальной загрузкой 6. Сбор информации с профайлера Unity 7. Анализ данных тестирования. 8. Оптимизация кода, партиклов, графических элементов.
Данные тестирования	Систему очень сильно нагружали тени, после многочисленных тестов производительности, с разными настройками теней, было принято решение, убрать их вовсе.
Ожидаемый результат	Сильная нагрузка системы
Фактический результат	Сильная нагрузка системы

Предпосылки	Снижение FPS
Постусловия	Улучшилась работа игры
Статус (Pass/Fail)	Pass
Комментарии	

Тестирование на всех этапах проектирования является очень важным элементом.

5. ДОКУМЕНТИРОВАНИЕ

Есть 3 уровня документации в проекте:

Верхний уровень: что такое компоненты и как они связаны друг с другом? В системе клиент-сервер, каков протокол между ними? Как передаются данные?

Средний уровень: что такое организация клиента, как компоненты клиента собраны вместе? (Вам это тоже нужно от сервера.) Есть ли серверная база данных? Каков организационный принцип вашего программного обеспечения? Вы используете шаблон проектирования, такой как MVC?

Низкий уровень: каждый класс должен иметь краткое описание своей цели, а имя должно отражать цель. Аналогично с методами, если они не очевидны. Лучшее - это совершенно очевидное именование, но это не всегда возможно.

5.1. Руководство пользователя

После входа в игру открывается главное меню игры (рис. 10), в нём можно выбрать раздел настроек (рис. 11), где можно настроить громкость музыки и общую громкость. Вход в игру (рис. 12) осуществляется через кнопку Выживание или Сюжет в зависимости от режима, в который хочет поиграть пользователь. После входа в игру есть инвентарь, в котором есть текущие вещи игрока и его одежда.

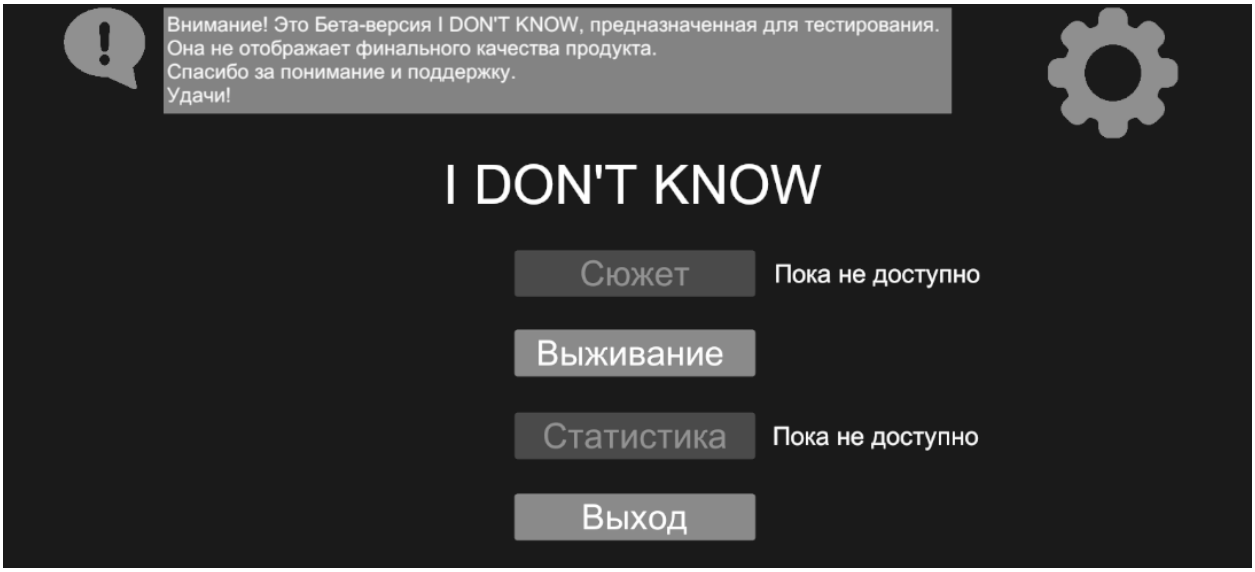

Рисунок 10 – Главное меню игры

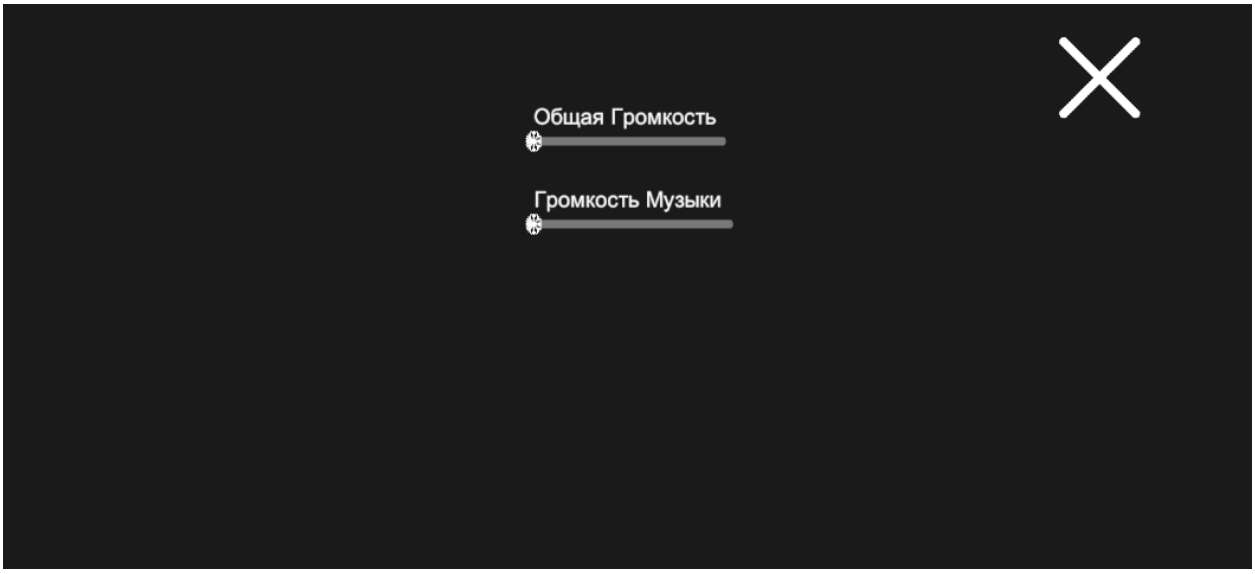

Рисунок 11 – Настройки

Рисунок 12 – Игра

ЗАКЛЮЧЕНИЕ

Игра “I don’t know” отвечает потребностям человека таким как отдых. От качества отдыха зависит эффективность человека в его жизни. Человеческому организму требуется не только отдых в физическом плане, но и в моральном тоже. Что иное, если не игры, помогают современному человеку отдохнуть от повседневной рутины?

На протяжении всего процесса разработки были выполнены следующие задачи:

- проанализирован предметная область,
- моделирование всех игровых объектов,
- настройка UI,
- были разработаны анимации,
- сделаны эффекты,
- написаны шейдеры,
- написаны Unit тесты,
- был оптимизирован код,
- были исправлены баги,
- было написано руководство пользователя.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1) ГОСТ 2.104-2006. Единая система конструкторской документации. Основные надписи. – Москва: Изд-во стандартов, 2007. - 14 с.

2) ГОСТ 2.105-95. Единая система программной документации. Общие требования к текстовым документам. – Минск, 1995. – 28 с.

3) ГОСТ 7.12-93. Единая система конструкторской документации. Сокращение слов в тексте и подписях под иллюстрациями. – Минск, 1995. – 28 с.

4) ГОСТ 7.32-2017. Единая система программной документации. Оформление иллюстраций и рисунков. – Москва: Изд-во стандартов, 2017. - 28 с.

5) ГОСТ 7.82-2001. Единая система программной документации. Библиографическая запись. Библиографическое описание электронных ресурсов. – Минск, 2001. – 24 с.

6) ГОСТ 19.001-77. Единая система программной документации. Общие положения. – Москва: Изд-во стандартов, 2010. - 5 с.

7) ГОСТ ISO/IEC 23270:2018. Определение формы и интерпретация программ, написанных на языке программирования C#. – Технический комитет: ИСО/МЭК СТК 1/ПК 22, 2018-04. -471с.

8) Роберт Нистрем. Game Programming Patterns. Genever Benning, 1-е издание. — 2014, — С. 432.

9) «Банда четырёх»: Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес. Design Patterns. Эддисон-Уэсли Профессионал; 1-е издание. — 1994, — С. 395.

10) Герберт Шилдт. C# 4.0 Полное Руководство. Макгроу Хилл; 1-е издание. — 2010, — С. 1056.

11) Clean Code, 2008.

12) Роберт Мартин. Clean Code. Pearson Education, 1-е издание. — 2008, — С. 464.

13) Джо Хокинг, Unity in Action, Мэннинг; 2-е издание, —2018, — С. 344.

14) Джесси Шелл, THE ART OF GAME DESIGN, CRC Press; 1-е издание. — 2008, — С. 637.

15) Сергей Тепляков, Паттерны проектирования на платформе .NET. Издательский Дом ПИТЕР; 3-е издание. — 2016, —С. 320.

16) Документация Microsoft — https://docs.microsoft.com/ru-ru/dotnet/csharp/.

17) Форум для программистов на Unity — https://forum.unity.com/.

18) Документация Unity — https://docs.unity3d.com/Manual/.