УДК 004.5:004.4+004.42

# РАЗРАБОТКА МОДУЛЬНОЙ АРХИТЕКТУРЫ ПОЛЬЗОВАТЕЛЬСКИХ ИНТЕРФЕЙСОВ ДЛЯ ИГР НА ДВИЖКЕ UNREAL ENGINE С ИСПОЛЬЗОВАНИЕМ ПЛАГИНА-ФРЕЙМВОРКА COMMON UI

**М. В. Шматко[1] [0000-0002-7255-8885], А. Д. Мякишев[2] [0009-0001-3985-7672]**

[1, 2]*Омский государственный технический университет, г. Омск, Россия*

[1]marin298@gmail.com, [2]makisevaleksej537@gmail.com

***Аннотация***

Статья посвящена разработке и экспериментальной верификации модульной архитектуры пользовательского интерфейса для игровых проектов на движке Unreal Engine. В ходе исследования выявлены типовые антипаттерны проектирования UI (монолитные виджет-структуры, жесткие ссылки, пренебрежение абстракцией, платформенная зависимость) и проведен анализ существующих архитектурных подходов (Data-Driven UI, MVC, MVVM), показавший их ограничения. Предложена архитектура, интегрирующая фреймворк Common UI, механизм асинхронной загрузки с мягкими ссылками (Soft References), систему тегов Gameplay Tags и реестр данных для динамической генерации меню настроек. Архитектура реализована в виде плагина AdvancedGameUserInterface. Его экспериментальная апробация с участием 12 разработчиков и 5 дизайнеров интерфейсов подтвердила эффективность предложенного решения по трем ключевым метрикам: сопровождаемость, производительность и масштабируемость. Сформулированы практические рекомендации по применению разработанной архитектуры в зависимости от масштаба игрового проекта. Полученные результаты могут быть использованы разработчиками для повышения качества и снижения затрат на сопровождение пользовательских интерфейсов в игровых проектах на Unreal Engine.

***Ключевые слова:*** *Unreal Engine, пользовательский интерфейс, Common UI, модульная архитектура, Gameplay Tags, Data-Driven UI, антипаттерны, сопровождаемость, масштабируемость, производительность, игровая разработка.*

## ВВЕДЕНИЕ

В современной игровой индустрии стремительно растет сложность реализуемых проектов, что повышает требования к качеству программной архитектуры видеоигр. Пользовательский интерфейс (UI), являясь связующим звеном между игроком и игровой механикой, часто становится источником архитектурных проблем [1]. Анализ практик разработки игровых проектов в среде Unreal Engine показывает, что стихийное проектирование UI приводит к возникновению монолитных структур – так называемых «мастер-виджетов». Такие структуры обладают высокой связанностью (high coupling), что критически затрудняет внесение изменений: модификация одного элемента, например смена шрифта, требует ручного редактирования сотен виджетов, а адаптация интерфейса под различные типы ввода (клавиатура и мышь или геймпад) зачастую – создания отдельных сборок проекта [1, 2].

Несмотря на наличие в экосистеме Unreal Engine инструментов, призванных решить эти проблемы, в частности плагина-фреймворка Common UI, предоставляющего поддержку кроссплатформенности и навигации «из коробки», вопросы построения на их основе целостной, масштабируемой архитектуры остаются малоизученными. Существующая документация описывает отдельные компоненты, но не предлагает системного подхода к их интеграции [3].

Таким образом, актуальность настоящего исследования обусловлена необходимостью разработки архитектурного решения для создания UI игровых проектов в Unreal Engine, которое минимизировало бы связанность компонентов, автоматизировало бы управление стилями и обеспечивало бы эффективное использование ресурсов.

В связи с этим целями исследования являются разработка и экспериментальная верификация модульной архитектуры пользовательских интерфейсов для игровых проектов на движке Unreal Engine, обеспечивающей повышение сопровождаемости кода, производительности и масштабируемости. Для их достижения поставлены следующие задачи.

1. Обзорно-аналитическая задача: выявить типовые архитектурные антипаттерны при разработке UI в Unreal Engine, а также проанализировать возможности для интеграции и ограничения существующих архитектурных подходов (Data-Driven UI, MVC, MVVM).

2. Конструкторская задача: создать модульную архитектуру UI для игровых проектов на движке Unreal Engine с использованием Common UI.

3. Экспериментальная задача: реализовать прототип игрового проекта на основе предложенной архитектуры, провести его апробацию и выполнить сравнительный анализ с монолитным подходом по метрикам: время внесения изменений (сопровождаемость), потребление оперативной памяти (производительность), трудоемкость добавления нового функционала (масштабируемость).

Научная новизна настоящего исследования состоит в следующем:

1. Предложена модульная архитектура пользовательского интерфейса игровых проектов, интегрирующая Common UI, Gameplay Tags и механизм асинхронной загрузки (Blueprint Async Action, Soft References). Архитектура реализует слабосвязанную многослойную организацию UI, обеспечивающую динамическое управление виджетами без жестких ссылок на экземпляры, что уменьшает расход памяти, отводимый на UI.

2. Экспериментально верифицирован подход к централизованному управлению стилями на основе иерархии классов стилей, доказывающий свою эффективность в сокращении временных затрат на глобальные изменения визуального оформления.

3. Предложена схема использования реестра данных в качестве бэкенда для динамической генерации меню настроек, позволяющая отделить логику представления от данных конфигурации и упрощающая масштабирование системы (добавление новых параметров без изменения кода).

## АНАЛИЗ ПРОБЛЕМ И ОГРАНИЧЕНИЙ РАЗРАБОТКИ UI В UNREAL ENGINE

Пользовательский интерфейс в Unreal Engine имеет иерархическую структуру и включает несколько уровней абстракции. Его базовым слоем является Slate – низкоуровневый фреймворк для разработки интерфейсов на C++, который используется, как правило, для создания редакторских инструментов и спе-

циализированных систем, требующих прямого контроля над логикой рендеринга. Из-за высокого порога вхождения Slate редко напрямую применяют в игровом коде [4].

Поверх Slate надстроена система Unreal Motion Graphics (UMG) – основной инструмент разработки пользовательского интерфейса в игровых проектах. UMG представляет собой визуальную оболочку над Slate и предоставляет удобный редактор, позволяющий создавать виджеты непосредственно в среде игрового движка. Виджеты (Widgets) – это все элементы, из которых визуально строится интерфейс. Таким образом, кнопки, текстовые поля, изображения, контейнеры, а также такие более сложные структуры, как меню, окна и HUD, являются виджетами [4].

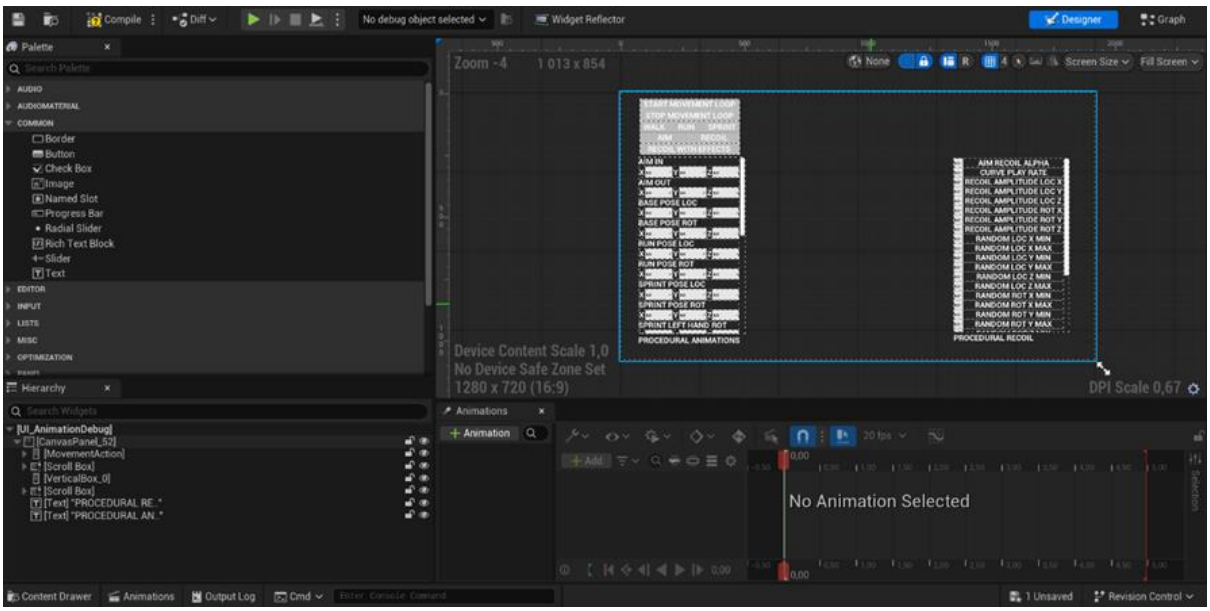


Рис. 1. Пример рабочей области в среде Unreal Engine с созданным интерфейсным окном.

Логика поведения компонентов пользовательского интерфейса обычно реализуется через систему Blueprints, что позволяет дизайнерам и разработчикам создавать интерактивные окна без необходимости писать большое количество C++ кода. Кроме того, UMG поддерживает анимацию интерфейса, что позволяет создавать плавные переходы между экранами и различные визуальные эффекты (см. рис. 1).

Следующим уровнем иерархической структуры пользовательского интерфейса в Unreal Engine является Common UI – специализированный плагин-фреймворк, расширяющий возможности UMG и предназначенный для разработки масштабируемых и кроссплатформенных интерфейсов [5].

Common UI не заменяет UMG, а работает поверх него, предоставляя передовые решения в сфере разработки интерфейса на Unreal Engine: например, с его помощью был организован интерфейс масштабного демонстрационного игрового проекта Lyra от Epic Games. Кодовая база Common UI активно развивается и поддерживается. Одной из ключевых возможностей плагина-фреймворка является система управления фокусом и навигацией между элементами пользовательского интерфейса. Это значительно упрощает его разработку для мультиплатформенных игровых проектов (решения с автоматической поддержкой различных устройств ввода, включая клавиатуру, мышь и геймпад). Кроме того, Common UI вводит концепцию активируемых виджетов (Activatable Widgets), позволяющую строить пользовательский интерфейс в виде иерархии экранов и слоев меню. Это облегчает реализацию типичных сценариев интерфейса, таких как переходы между меню, обработка кнопки «Назад», активация и отключение Pop-Up окон и др. Для централизованного управления визуальным стилем интерфейса Common UI использует отдельные классы, содержащие стилеобразующие данные, что позволяет задавать стили кнопок и других элементов в одном месте, а затем применять их ко всему интерфейсу игрового проекта.

Проанализируем типовые проблемы и архитектурные антипаттерны, которые возникают при разработке пользовательского интерфейса в Unreal Engine.

В каждом игровом проекте существует ряд распространенных проблем, связанных с организацией и управлением пользовательским интерфейсом. Например это управление активными и неактивными состояниями разных элементов интерфейса: в ситуации, когда игроку одновременно на экране показывается несколько меню, разработчику необходимо обеспечить взаимодействие игрока только с одним из них.

Другой частой проблемой становятся обработка всплывающих окон и обеспечение перехода между разными экранами интерфейса. Например, если

на экране появляется модальное окно, которое закрывается по нажатию на любую кнопку, как гарантировать, что после его закрытия пользователь переместится именно туда, куда нужно? Как эффективно отслеживать все открытые интерфейсные элементы и управлять порядком их отображения?

Кроме того, в любом игровом проекте разработчики пользовательского интерфейса отвечают за механизмы сбора и отслеживания нужных для отображения данных, собираемых из других игровых систем. Впоследствии эти данные могут также сохраняться в системах учетных записей и облачных хранилищах [6]. Возникает закономерный вопрос: какие системы сбора данных существуют и какие из них наиболее эффективно справляются со всем спектром подобных задач.

Описанные примеры иллюстрируют лишь часть проблем, с которыми сталкиваются разработчики при создании пользовательских интерфейсов игровых проектов. Для их устранения они нередко прибегают к локальным решениям, которые хорошо работают на ранних этапах, но плохо масштабируются [7]. Как правило, подобные решения связаны с применением антипаттернов.

Антипаттерны – это нетипичные способы решения задач, допустимые на этапах MVP и прототипирования ради скорости, но в долгосрочной перспективе затрудняющие расширение задач и сопровождение игрового проекта [8].

Антипаттерны разработки пользовательских интерфейсов в среде Unreal Engine можно разделить на несколько категорий: они имеют свои причины внедрения и недостатки [9]. В табл. 1 дано их описание и показаны характеристики.

Табл. 1. Антипаттерны разработки пользовательских интерфейсов в среде Unreal Engine.

| Антипаттерн | Описание | Причины внедрения | Недостатки внедрения |
|---|---|---|---|
| Monolithic (God) Widget (монолитные виджет-структуры) | Крупные, сложно поддерживаемые виджеты без деления на компоненты. Весь интерфейс или его значительная часть собраны в одном виджете. | Нехватка архитектурного планирования, быстрая разработка, отсутствие декомпозиции. | Трудность масштабирования и повторного использования, сложность сопровождения. |

| | | | |
|---|---|---|---|
| Hard References (жесткие ссылки) | Прямые ссылки на классы виджетов, что принуждает движок держать их в памяти постоянно, а также ссылки внутри виджета на внешние объекты для получения данных. | Нехватка планирования, быстрая разработка, отсутствие понимания механизмов загрузки. | Увеличение потребления оперативной памяти, высокая связанность компонентов. |
| Lack of Abstraction (пренебрежение абстракцией) | Отсутствие разделения ответственности между UI и геймплейной логикой: например, UI постоянно опрашивает данные на каждом кадре через Tick или Property Bindings. | Стремление к быстрой реализации, неиспользование архитектурных паттернов (MVC/MVVM). | Сложность сопровождения, высокая связанность, сложный рефакторинг, потеря производительности. |
| Platform Dependency (платформенная зависимость) | Отсутствие поддержки различных платформ и систем ввода. Интерфейс завязан на конкретный тип устройств: например, только клавиатура и мышь. | Отсутствие архитектуры, учитывающей кроссплатформенность, недостаток планирования. | Невозможность выпуска игры на нескольких платформах без переработки UI, потеря части аудитории. |

Представленные антипаттерны разработки интерфейсов в среде Unreal Engine влекут за собой не только технические проблемы сопровождения, но и прямые нарушения качества интерфейса с точки зрения пользователя.

Исследования в области создания пользовательских интерфейсов выделяют ряд фундаментальных критериев для оценки их качества: согласованность (consistency), обратная связь (feedback), предсказуемость (predictability) и др. [10]. В табл. 2 показано соответствие между выявленными антипаттернами и нарушаемыми ими критериями.

Табл. 2. Влияние архитектурных антипаттернов на качество UI.

| Нарушаемый критерий UI | Антипаттерн | Проявление для пользователей |
|---|---|---|
| Согласованность (Consistency) | Monolithic (God) Widget | Глобальное изменение стиля (шрифт, цвет) применяется непоследовательно: одни элементы интерфейса обновляются, другие – сохраняют старый стиль. |
| Обратная связь (Feedback) | Hard References | Длительная загрузка интерфейса при старте игры, заметные фризы при открытии меню из-за конкуренции за ресурсы. |
| Обратная связь (Feedback) | Lack of Abstraction (Tick/Property Bindings) | Неотзывчивость интерфейса при взаимодействии, задержки между действием игрока и откликом системы, снижение частоты кадров. |
| Предсказуемость (Predictability) | Platform Dependency | Навигация по меню с геймпада работает иначе, чем с клавиатуры, что дезориентирует пользователей и требует повторного освоения управления. |

Таким образом, основа для разработки устойчивого и расширяемого пользовательского интерфейса в игровых проектах – это грамотно спроектированная архитектура UI.

## АРХИТЕКТУРНЫЕ ПОДХОДЫ К РАЗРАБОТКЕ UI В UNREAL ENGINE

Для преодоления описанных выше проблем и антипаттернов в практике разработки пользовательских интерфейсов на Unreal Engine применяют несколько архитектурных подходов, ориентированных на разделение данных и представления. Наиболее распространенными из них являются Data-Driven-подход, а также паттерны MVC и MVVM.

Data-Driven-подход позволяет отделить данные и логику от визуального представления. Он хорошо сочетается с принципами SOLID, что способствует снижению связанности компонентов и повышению расширяемости системы. Схематично Data-Driven-подход представлен на рис. 2.

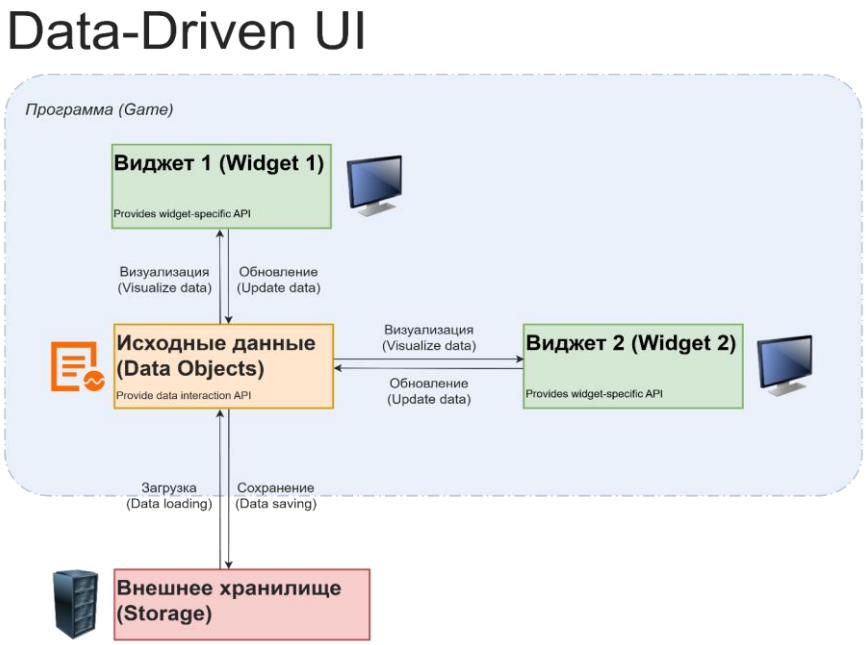


Рис. 2. Общая схема Data-Driven-подхода.

Идея подхода заключается в создании классов для инкапсуляции всех данных (информации о том, что UI должен показывать игроку) и предоставлении набора методов, раскрывающих важные аспекты системы и ее состояние [6]. В проекте на Unreal Engine это помогает избежать использования событий Tick в Blueprint и Property Bindings, которые приводят к частым запросам данных и создают дополнительную нагрузку на систему. Отделение жизненного цикла данных от жизненного цикла UI особенно важно, поскольку предметы инвентаря, предложения внутриигрового магазина или статистика игрока в определенный момент времени могут не совпадать с объектами пользовательского интерфейса. В случае с Data-Driven-подходом программист может использовать несколько виджетов, получающих данные от одних и тех же объектов, что упрощает разработку.

Data-Driven-подход к проектированию архитектуры интерфейса в проекте на Unreal Engine чаще всего реализуют с использованием паттернов MVC и MVVM.

Паттерн MVC (Model-View-Controller) разделяет систему на три компонента:

1) модель (Model) – хранит данные и бизнес-логику;

2) представление (View) – отвечает за отображение данных (виджеты);

3) контроллер (Controller) – управляет взаимодействием между моделью и представлением.

На рис. 3 показана схема паттерна MVC.

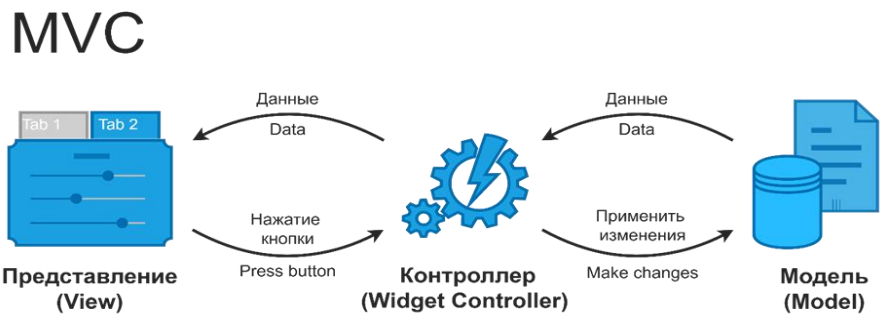


Рис. 3. Схема паттерна MVC.

Такое разделение делает код модульным: замена виджетов не влияет на контроллер, а замена контроллера – на хранилище данных [11].

Паттерн MVVM (Model-View-ViewModel) расширяет MVC, добавляя ViewModel – посредника с двунаправленной привязкой данных для автоматического обновления UI [12]. В Unreal Engine MVVM реализован через официальный плагин UMG Viewmodel, входящий в состав движка (см. рис. 4).

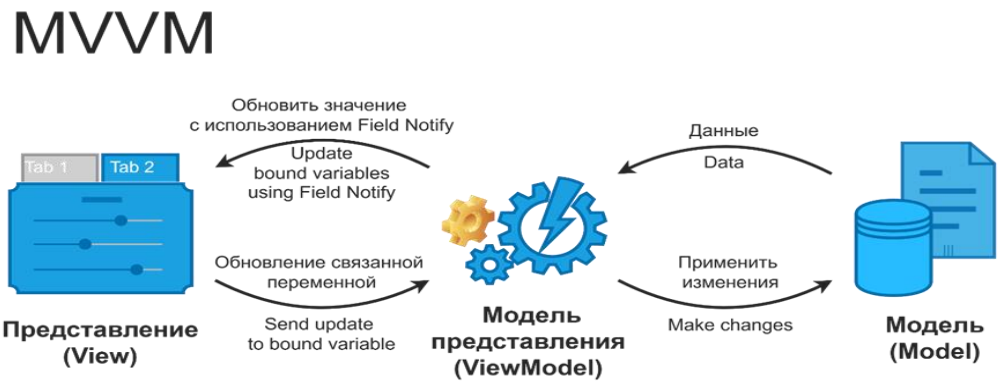


Рис. 4. Схема паттерна MVVM.

Рассмотренные подходы Data-Driven UI, MVC и MVVM предоставляют разработчикам инструменты для снижения связанности компонентов, отделения логики от представления и повышения масштабируемости интерфейсных решений. Однако их применение в чистом виде не решает всех проблем, характерных для разработки пользовательского интерфейса в Unreal Engine. В частности, они не предлагают встроенных механизмов для:

– управления многослойной организацией интерфейса (стеки виджетов);
– асинхронной загрузки экранов с использованием мягких ссылок;
– гибкой адресации элементов интерфейса через теги;
– централизованного конфигурирования меню настроек на основе реестра данных.

Указанные ограничения определяют необходимость разработки специализированного архитектурного решения, интегрирующего существующие паттерны с дополнительными механизмами, адаптированными к специфике Unreal Engine.

## АРХИТЕКТУРА МОДУЛЬНОЙ СИСТЕМЫ UI НА ОСНОВЕ COMMON UI И C++

Анализ типичных антипаттернов, проведенный в рамках настоящего исследования, позволил оптимизировать подход и на основе Common UI создать архитектуру модульной системы пользовательского интерфейса. Эта архитектура реализована в виде отдельного плагина AdvancedGameUserInterface, что обеспечивает возможность повторного использования и расширения системы UI в различных игровых проектах. На рис. 5 представлена карта связей внутриигровых меню в разработанной системе.


Рис. 5. Карта связей внутриигровых меню в разработанной системе UI.

Предлагаемая система UI включает следующие компоненты:

– система шаблонных базовых классов для унификации визуальных элементов;

– многослойная организация интерфейса на основе стеков (Widget Stacks);

– механизм адресации и управления интерфейсом с использованием Gameplay Tags;

– динамическая система конфигурирования меню настроек на основе Data Registry;

– подсистема управления загрузочными экранами.

Логика работы системы UI строится на использовании ряда базовых механизмов Unreal Engine, обеспечивающих модульность и масштабируемость архитектуры.

1. Subsystems – классы, автоматически создаваемые в Runtime в рамках определенного контекста (GameInstance, World, LocalPlayer). В разработанной

системе используется ULocalPlayerSubsystem, что обеспечивает привязку UI к конкретному игроку.

2. Developer Settings – классы, наследуемые от UDeveloperSettings, которые отображаются в Project Settings и сохраняются в ini-файлы проекта: задействованы, чтобы хранить ссылки на классы виджетов и другие параметры системы UI. Это позволило избежать констант в коде.

3. Blueprint Async Action – асинхронные функции Blueprint: в разработанной системе применены для загрузки и отображения интерфейсных экранов без блокировки игрового потока.

4. Gameplay Tags – иерархические строковые метки (ярлыки) для описания состояний и данных: использованы для адресации и идентификации элементов пользовательского интерфейса.

5. Input Processor – механизм низкоуровневого перехвата пользовательского ввода на уровне Slate, позволивший реализовать корректную систему переназначения клавиш.

6. Data Assets – объекты хранения данных: применены в разработанной системе для конфигурирования интерфейсных элементов.

7. CommonUI, UMG widgets – базовые классы виджетов, от которых наследуются пользовательские элементы интерфейса.

8. Blueprint Function Library – классы для хранения публичных статических функций: задействованы для обеспечения удобного доступа к данным конфигурации.

Представим основные элементы архитектуры модульной системы пользовательского интерфейса, созданной на основе Common UI**.**

1. Шаблонизация и централизованное управление стилями в Common UI.

Этот механизм реализован в базовом шаблоне интерфейсного окна WBP_Template_Layout. Шаблон использует именные слоты – Named Slots, которые позволяют динамически вставлять различные элементы интерфейса в заранее определенные области (см. рис. 6).

Такой подход позволяет разделить структуру интерфейса и его содержимое: разработчик создает новые окна, используя единый шаблон, что обеспечивает единообразие расположения элементов управления, областей описания и панелей меню.

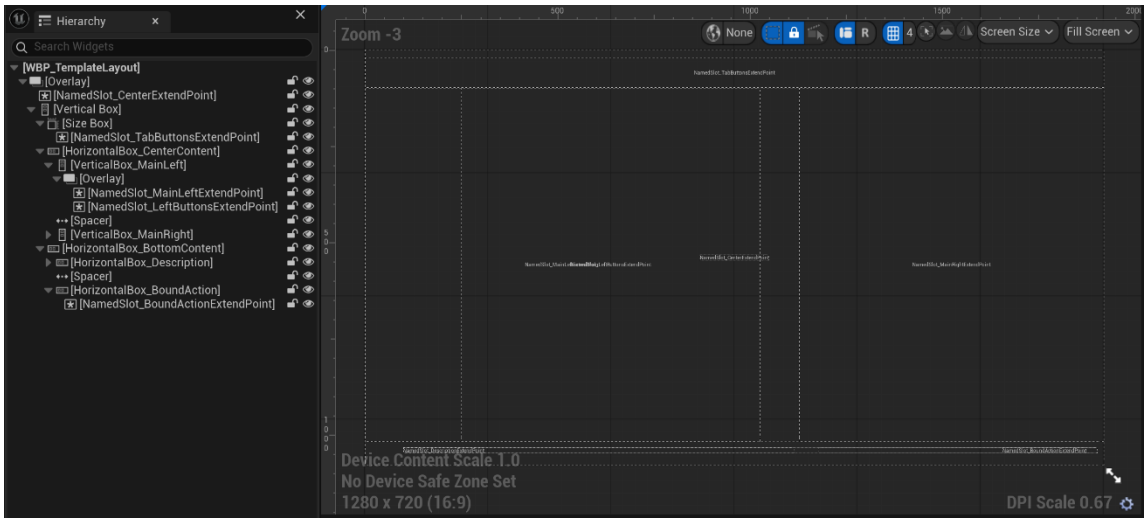


Рис. 6. Базовый шаблон интерфейсного окна WBP_Template_Layout.

Наряду с этим реализовано централизованное хранение стилевых данных для визуальных компонентов интерфейса, таких как кнопки, текстовые блоки и пр. В системе UI выделены отдельные классы стилей, содержащие параметры визуального оформления – цвета, шрифты, размеры и пр. (см. рис. 7). Такое решение позволяет вносить глобальные изменения в оформление интерфейса, не модифицируя каждый виджет по отдельности.

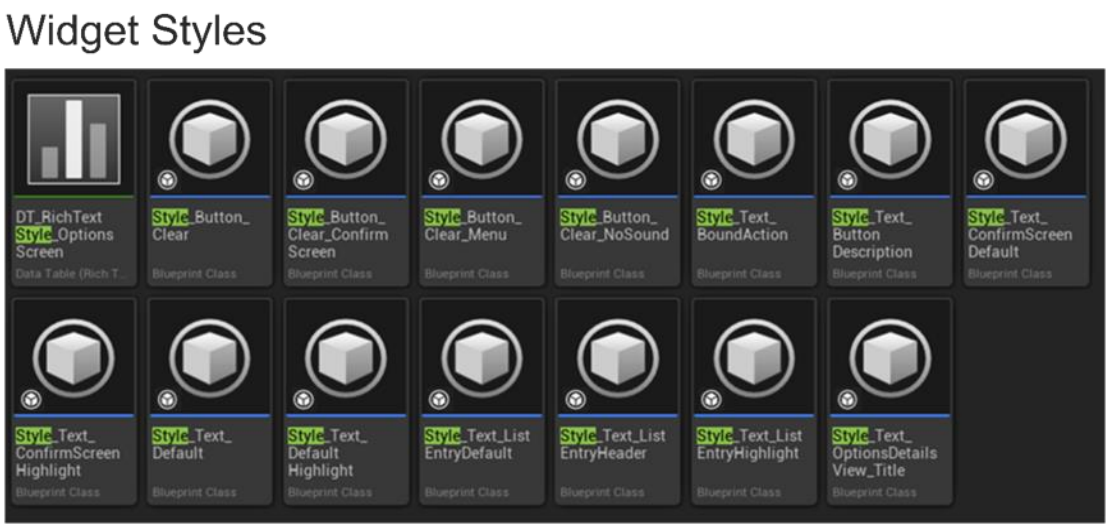


Рис. 7. Классы стилей модульной системы UI.

2. Многослойная организация пользовательского интерфейса на основе стеков виджетов (Widget Stacks).

Преимуществом созданной архитектуры является многослойная система UI, центральным элементом которой является корневой виджет Primary Layout: он добавляется в Viewport при инициализации пользовательского интерфейса. Primary Layout содержит набор контейнеров-стеков, каждый из которых отвечает за отображение определенного слоя интерфейса. В разработанной системе использованы четыре основных слоя:

– GameHUD – слой для окон, отображаемых во время игрового процесса. В этот слой входят элементы HUD, такие как индикаторы состояния игрока, мини-карта, подсказки и другие элементы игрового интерфейса;

– GameMenu – слой для отображения внутриигровых меню, доступных во время игры, например меню инвентаря;

– Frontend – слой главного меню игры, отображающий экраны запуска игры, главное меню, экран настроек и другие элементы интерфейса, которые не имеют непосредственного отношения к игровому процессу;

– Modal – слой модальных окон, который располагается поверх остальных слоев и используется для отображения всплывающих диалогов подтверждения.

Каждый слой в системе UI реализован как контейнер CommonActivatableWidgetContainerBase, который управляет отображением виджетов по принципу стека. При открытии нового интерфейсного окна оно помещается в соответствующий стек, а при закрытии – удаляется из него (см. рис. 8).

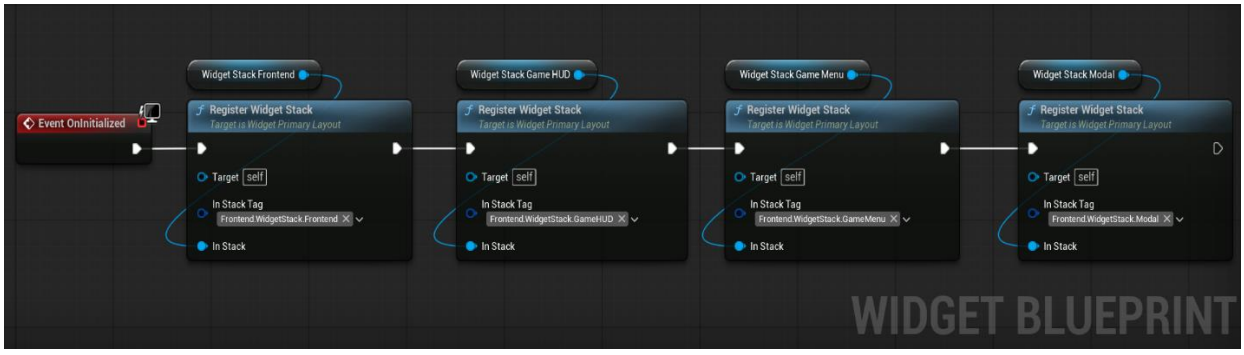


Рис. 8. Регистрация стеков при инициализации в корневом виджете Primary Layout.

Многослойную архитектуру UI реализует класс FrontendUISubsystem (листинг header-файла приведен на рис. 9). Этот ключевой класс, наследуемый от ULocalPlayerSubsystem, обеспечивает привязку системы интерфейса к конкретному игроку и отвечает за асинхронную инициализацию корневого виджета Primary Layout, его хранение, а также за добавление виджетов в соответствующие стеки по тегам с использованием мягких ссылок и асинхронной загрузки.

3. Адресация и управление с помощью Gameplay Tags.

В созданной архитектуре Gameplay Tags обеспечивают идентификацию слоев интерфейса, получение классов интерфейсных экранов, управление изображениями и др. ресурсами.

Использование Gameplay Tags позволяет отказаться от жестких ссылок на конкретные классы виджетов: система оперирует мягкими ссылками (Soft References), которые загружаются в память только в нужный момент. Такой подход обеспечивает два преимущества:

– снижение потребления оперативной памяти, поскольку неиспользуемые интерфейсные элементы не загружаются заранее;

– повышение гибкости архитектуры, так как разработчик может изменять используемые виджеты через настройки игрового проекта без изменения исходного кода.

```C++
UCLASS()
class ADVANCEDGAMEUSERINTERFACE_API UFrontendUISubsystem : public ULocalPlayerSubsystem
{
	GENERATED_BODY()

public:

	//static UFrontendUISubsystem* Get(const UObject* WorldContextObject);
	static UFrontendUISubsystem* GetFromPlayerController(const APlayerController*
PlayerController);
	static UFrontendUISubsystem* GetFromLocalPlayer(const ULocalPlayer* LocalPlayer);


	//~Begin USubsystem Interface
	//virtual bool ShouldCreateSubsystem(UObject* Outer) const override;
	//~End USubsystem Interface


	UFUNCTION(BlueprintCallable)
	bool IsPrimaryLayoutWidgetRegistered() const;

	void RegisterAndAddPrimaryLayoutAsync(TWeakObjectPtr<APlayerController> PlayerController,
		TSoftClassPtr<UWidget_PrimaryLayout> InSoftPrimaryLayoutWidgetClass,
		TFunction<void(EAsyncPushWidgetState, UWidget_PrimaryLayout*)>
AsyncCreationStateCallback);

	void PushSoftWidgetToStackAsync(const FGameplayTag& InWidgetStackTag,
		TSoftClassPtr<UWidget_ActivatableBase> InSoftWidgetClass,
		TFunction<void(EAsyncPushWidgetState, UWidget_ActivatableBase* )>
AsyncPushStateCallback);


	void PushConfirmScreenToModalStackAsync(EConfirmScreenType InScreenType,
		const FText& InScreenTitle,
		const FText& InScreenMsg,
		TFunction<void(EConfirmScreenButtonType)> ButtonClickedCallback);

	UPROPERTY(BlueprintAssignable)
	FOnButtonDescriptionTextUpdatedDelegate OnButtonDescriptionTextUpdated;

private:
	UPROPERTY(BlueprintReadOnly, Transient, meta = (AllowPrivateAccess = true))
	TObjectPtr<UWidget_PrimaryLayout> CreatedPrimaryLayout;
};
```

Рис. 9. Листинг header-файла FrontendUISubsystem.

Асинхронная загрузка интерфейсных экранов реализуется с помощью специальных Blueprint-узлов, созданных с использованием класса UBlueprintAsyncActionBase (см. рис. 10). Узлы создают виджет, загружают его в память и помещают в соответствующий стек.

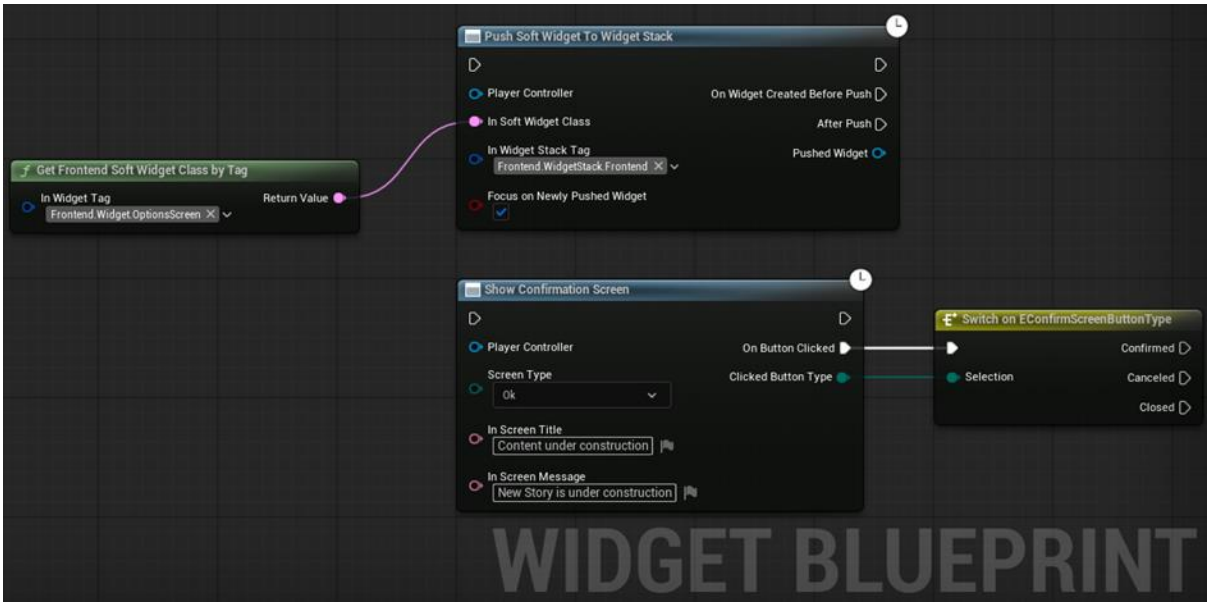


Рис. 10. Асинхронная загрузка UI-экранов с помощью Blueprint-узлов.

Виджеты задаются в Custom Developer Settings. На рис. 11 продемонстрирована разработанная секция Developer Settings, где каждому тегу соответствует свой виджет.

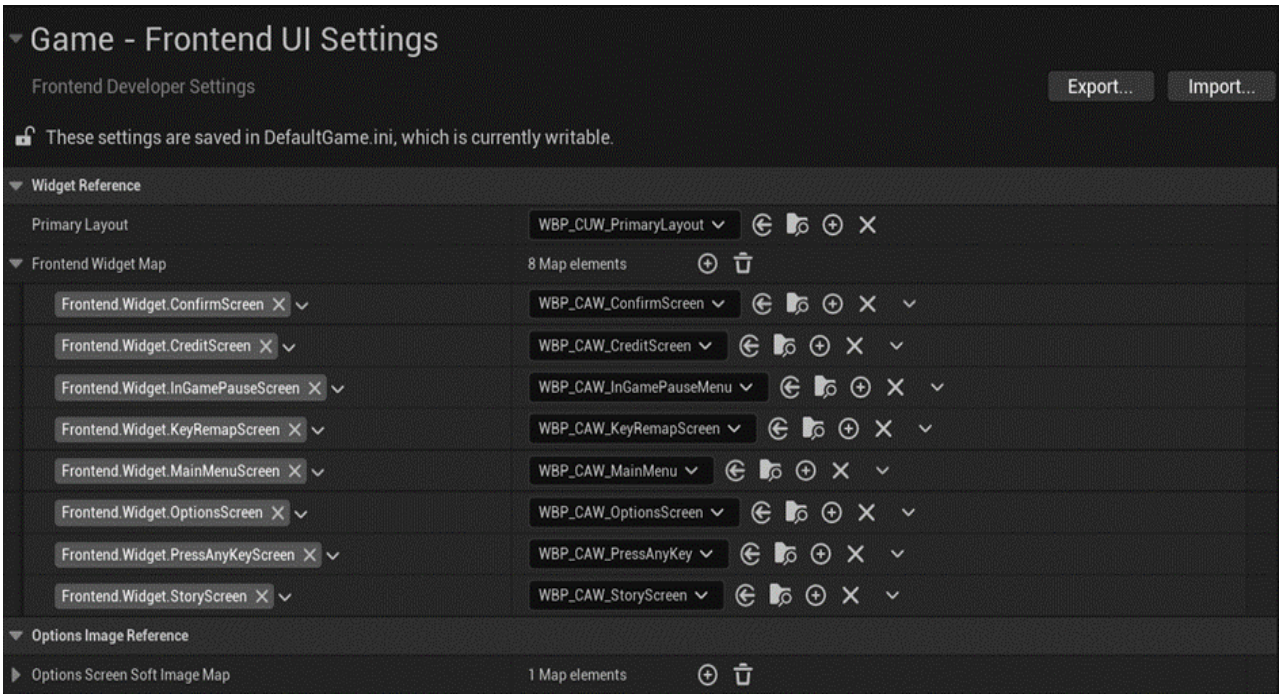


Рис. 11. Секция Custom Developer Settings с соответствием Gameplay Tags и виджетов интерфейса.

4. Динамическая система настроек на основе Data-Driven подхода.

В традиционных подходах структура меню настроек жестко задана в виджетах, что усложняет изменение и добавление новых параметров. В созданной архитектуре эта проблема решена за счет динамического подхода, основанного на данных (Data-Driven): структура меню формируется процедурно на основе объектов данных.

Основу такой системы составляет иерархия классов данных, показанная на рис. 12.

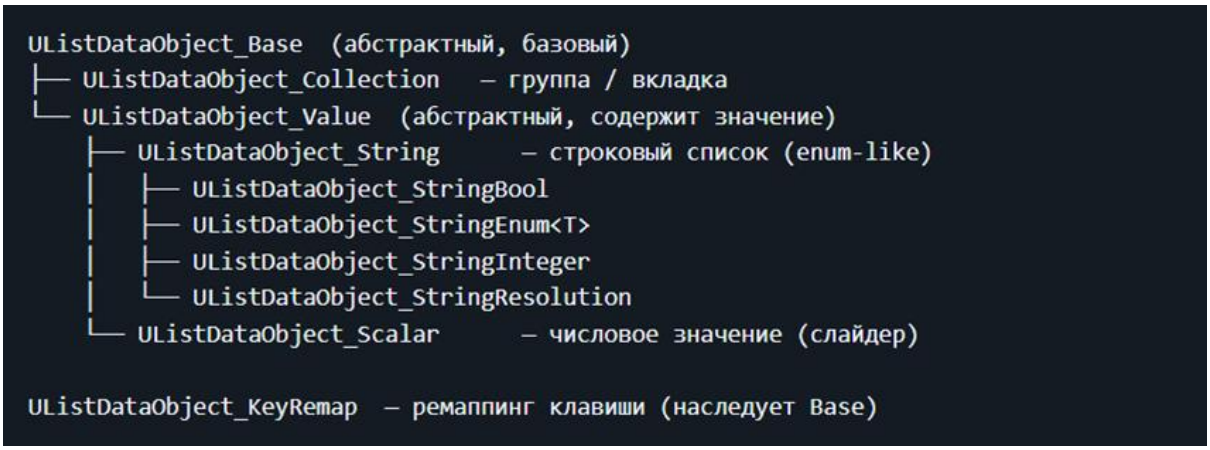


Рис. 12. Структура классов данных, обеспечивающих процедурное формирование меню настроек.

Каждый объект данных содержит информацию о параметре настройки: текущее значение, доступные варианты выбора, условия редактирования, а также зависимости от других параметров. Кроме того, система поддерживает возможность сброса параметров к значениям по умолчанию.

Все данные настроек регистрируются в специальном реестре OptionsDataRegistry, который формирует структуру вкладок меню настроек. При открытии экрана настроек система автоматически запрашивает данные из реестра и динамически формирует список элементов интерфейса. Благодаря этому добавление новых настроек выполняется без изменения структуры Blueprints.

Использование реестра данных позволяет отделить логику меню настроек от визуального представления, что соответствует Data-Driven-подходу и значительно упрощает масштабирование системы.

Рассмотрим, как разработанная модульная система UI интегрируется в игровой проект.

Для тестирования логики и примера интеграции был создан тестовый игровой проект, включающий уровни MainMenuLevel, Level_1, а также контроллеры BP_MainMenuPlayerController и BP_FirstPersonPlayerController. Добавление Blueprint-кода в эти контроллеры обеспечивает их взаимодействие с созданной системой UI (см. рис. 13 и 14).

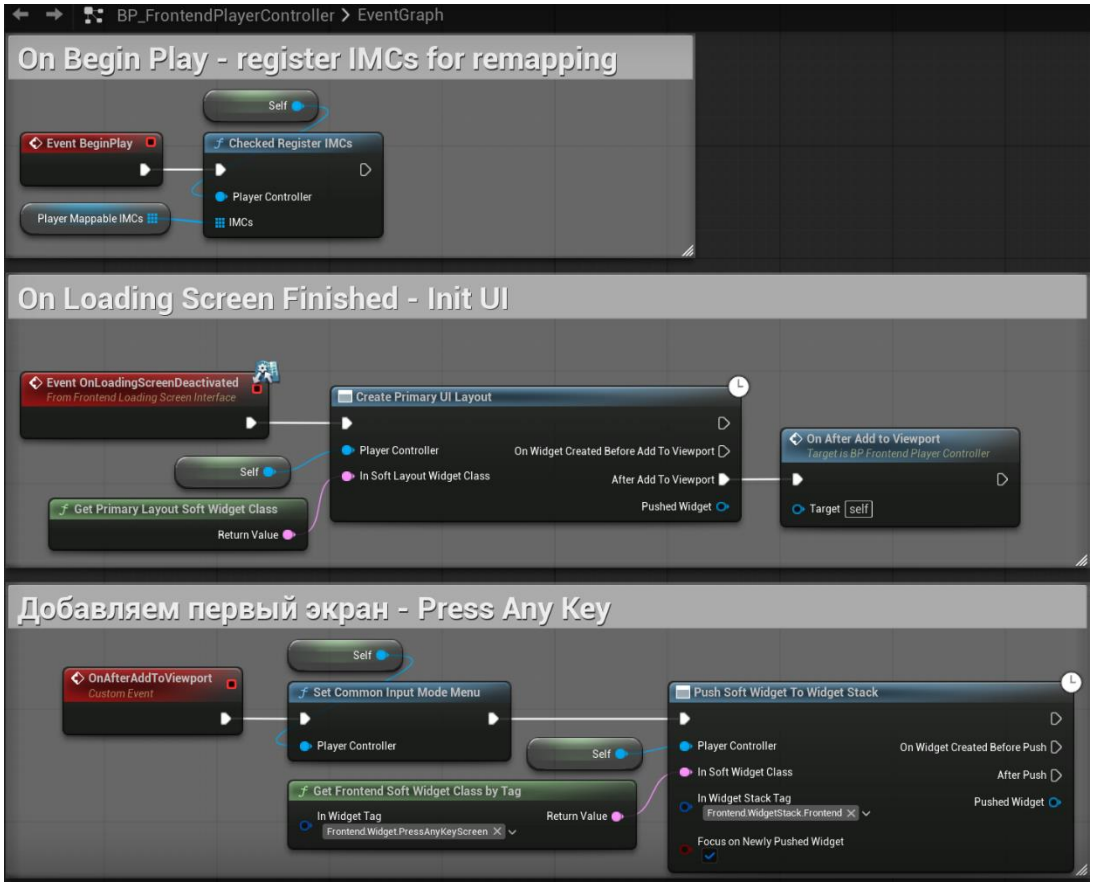


Рис. 13. Настройка контроллера главного меню.

Предложенная модульная система пользовательского интерфейса, реализованная в виде плагина AdvancedGameUserInterface, базируется на архитектурных принципах, обеспечивающих гибкость, масштабируемость и эффективное использование ресурсов: шаблонизации интерфейсных окон, многослойной организации на основе стеков, адресации через Gameplay Tags с асинхронной за-

грузкой, а также динамической конфигурации настроек на основе данных. Проведенная интеграция системы в тестовый игровой проект подтвердила ее работоспособность. Оценка эффективности предложенного решения представлена в следующем разделе.


Рис. 14. Настройка контроллера тестового уровня.

## ЭКСПЕРИМЕНТАЛЬНАЯ АПРОБАЦИЯ И ОЦЕНКА ЭФФЕКТИВНОСТИ ПРЕДЛОЖЕННОГО РЕШЕНИЯ

Для оценки эффективности модульной системы пользовательского интерфейса, созданной на основе Common UI, в лабораторных условиях на базе Омского государственного технического университета была проведена ее экспериментальная апробация.

В ходе апробации на основе плагина AdvancedGameUserInterface в среде Unreal Engine 5.6 был реализован прототип пользовательского интерфейса, включающий все стандартные компоненты, как правило, задействованные в реальном игровом проекте:

- главное меню (WBP_MainMenuScreen);
- внутриигровой HUD;
- многостраничное меню настроек (графика, звук, управление);
- экран переназначения клавиш (WBP_KeyRemapScreen);
- диалоги подтверждения (WBP_ConfirmScreen).

Для сравнения с предложенным решением была реализована альтернативная версия на базе стандартного UMG (без использования Common UI), намеренно выполненная с применением типичных антипаттернов: весь пользовательский интерфейс был собран в едином «мастер-виджете» с жесткими ссылками на классы виджетов, без иерархии базовых классов стилей и без централизованного реестра настроек.

Целью экспериментального исследования было количественное сравнение предложенной архитектуры модульной системы пользовательского интерфейса с монолитной архитектурой по трем ключевым метрикам:

1) сопровождаемость (время внесения изменений);
2) производительность (потребление оперативной памяти);
3) масштабируемость (трудоемкость добавления нового функционала).

В экспериментальной апробации была задействована группа общей численностью 17 человек. Это бакалавры 4 курса, магистранты и аспиранты, имеющие опыт 2–5 лет в реализации реальных игровых проектов и работы с UMG: 12 разработчиков на Unreal Engine и 5 дизайнеров интерфейсов, работающих не только с Figma, но и с игровым движком.

Перед его началом все участники были ознакомлены с обоими вариантами реализации пользовательского интерфейса, а порядок предъявления условий в ходе эксперимента балансировался для исключения эффекта переноса навыков.

Представим полученные результаты.

Метрика № 1. Сопровождаемость – время внесения типовых изменений в пользовательский интерфейс.

Каждому из 17 участников предлагалось выполнить три типовые задачи в двух вариантах – монолитном и модульном:

1) смена гарнитуры шрифта для всех текстовых элементов;

2) изменение стиля оформления кнопок на нескольких экранах;

3) изменение цветовой схемы всех элементов управления.

Время выполнения каждого задания участниками эксперимента фиксировалось с момента его получения до момента визуального подтверждения корректности результата в Unreal Engine.

Среднее время выполнения задач в монолитном варианте составило: смена шрифта – 26.4 мин (SD = 4.7), смена стиля оформления кнопок – 34.2 мин (SD = 3.1), смена цветовой схемы – 31.8 мин (SD = 5.2). В модульном варианте соответствующие показатели таковы: 0.9 мин (SD = 0.2), 2.1 мин (SD = 0.4) и 1.3 мин (SD = 0.3).

Разброс результатов в монолитном варианте объясняется различным опытом участников в навигации по Blueprint-графам Unreal Engine. В модульном варианте высокая воспроизводимость результата обусловлена централизованным хранением стилей, шаблонной структурой интерфейса и использованием мягких ссылок.

Для проверки статистической значимости использован t-критерий Стьюдента для зависимых выборок. По всем трем задачам достигнуты статистически значимые различия ($p < 0.001$). Среднее сокращение времени выполнения типовой задачи составило 92.6%, что подтверждает высокую эффективность предложенной архитектуры с точки зрения сопровождаемости.

Метрика № 2. Производительность – потребление оперативной памяти.

Измерения проводились с помощью встроенного профайлера Unreal Engine. Для каждого из двух вариантов выполнялось 10 независимых запусков прототипа, в ходе которых фиксировалось пиковое потребление оперативной памяти в момент полной отрисовки главного меню. Результаты 10 запусков приведены в табл. 3.

Табл. 3. Пиковое потребление ОЗУ при открытии главного меню.

| Запуск | Монолитный вариант, МБ | Модульный вариант, МБ | Экономия, МБ |
|---|---|---|---|
| 1 | 438 | 372 | −66 |
| 2 | 432 | 369 | −63 |

| | | | |
|---|---|---|---|
| 3 | 435 | 370 | −65 |
| 4 | 441 | 373 | −68 |
| 5 | 434 | 368 | −66 |
| 6 | 436 | 371 | −65 |
| 7 | 433 | 369 | −64 |
| 8 | 437 | 372 | −65 |
| 9 | 439 | 370 | −69 |
| 10 | 434 | 368 | −66 |
| Среднее (SD) | 436 (±3) | 370 (±2) | −66 (−15.1%) |

Детализации структуры потребления памяти представлена в табл. 4: указаны объемы памяти, занятые после открытия главного меню (в модульном варианте UI-ресурсы загружаются по требованию).

Табл. 4. Детализация структуры потребления памяти.

| Компонент | Монолитный вариант, МБ | Модульный вариант, МБ |
|---|---|---|
| Базовые ресурсы движка и игры | 350 | 350 |
| UI-ресурсы (виджеты, текстуры, шрифты) | 86 | 20 |
| Итого (пиковое ОЗУ) | 436 | 370 |

В монолитном варианте жесткие ссылки на все виджеты принуждают движок загрузить полный граф зависимостей при инициализации. Суммарный объем UI-ресурсов в памяти при старте составил 86 МБ – все виджеты, текстуры и шрифты загружаются единовременно.

В модульном же варианте соответствие тегов и классов виджетов хранится в Developer Settings с использованием мягких ссылок (TSoftClassPtr). При открытии главного меню загружаются только непосредственно используемые экраны (WBP_MainMenuScreen и WBP_PressAnyKeyScreen), занимая около 12 МБ. Ресурсы экранов настроек и переназначения клавиш (еще ~8 МБ) остаются выгруженными до первого обращения, что обеспечивает отложенную загрузку.

Среднее пиковое потребление ОЗУ в монолитном варианте составило 436 МБ (SD = 3 МБ), в модульном – 370 МБ (SD = 2 МБ). Экономия составила 66 МБ (15.1%). Различия статистически значимы (t-критерий для независимых выборок: $t(18) = 54.2$; $p < 0.001$).

Важно подчеркнуть, что эффект накапливается по мере роста игрового проекта: добавление каждого нового экрана пользовательского интерфейса с жесткими ссылками увеличивает начальный объем занятой памяти, тогда как в модульной архитектуре новые экраны (если они не требуются на момент инициализации) не влияют на начальный объем потребляемой памяти.

Метрика № 3. Масштабируемость – трудоемкость добавления нового функционала.

Двенадцать разработчиков из группы эксперимента выполняли задачу по добавлению нового раздела в меню настроек. Требовалось создать раздел «Специальные возможности» (Accessibility), включив в него три новые настройки и перенеся две существующие настройки из других разделов. Каждый разработчик выполнял задачу дважды с перерывом в двое суток; в табл. 4 приведены средние значения по двум попыткам для каждого участника.

В монолитном варианте участникам эксперимента требовалось:

1) добавить функции для изменения и применения настроек в класс UTestGameUserSettings.h;

2) скопировать или вручную создать Blueprint-виджеты для каждого параметра новой вкладки;

3) переписать логику отображения и изменения параметров.

Соответственно, в модульной архитектуре задача решалась без внесения изменений в существующие Blueprint. В этом варианте участникам требовалось:

1) добавить функции для изменения и применения настроек в класс UFrontendGameUserSettings.h;

2) добавить enum-варианты для применения настроек;

3) обновить OptionsDataRegistry, добавив новую запись для представления данных на вкладке.

После этого система автоматически генерировала соответствующие виджеты при следующем открытии нужной вкладки экрана настроек.

Результаты измерений времени выполнения задачи участниками эксперимента приведены в табл. 5.

Среднее время выполнения в монолитном варианте составило 77 мин. (SD = 7 мин), в модульном – 9.8 мин (SD = 1.5 мин). Ускорение составляет 7.9 раза. Это сокращение времени на 87.3%. Различия статистически значимы (t-критерий для зависимых выборок: $t(11) = 28.6$, $p < 0.001$).

Важно подчеркнуть, что ни один из разработчиков в модульном варианте не вносил изменений в существующие Blueprint-виджеты, что позволило минимизировать их временные затраты и исключить риск внесения ошибок в уже работающий функционал прототипа интерфейса.

Обобщим результаты эксперимента (см.в табл. 6): данные, полученные по трем метрикам, демонстрируют статистически значимое превосходство созданной архитектуры модульной системы пользовательского интерфейса над монолитной.

Проведенная экспериментальная апробация с участием 17 начинающих специалистов по разработке видеоигр (12 разработчиков и 5 дизайнеров игровых интерфейсов) и техническими замерами позволила получить количественные подтверждения эффективности предложенной модульной архитектуры по всем трем заявленным метрикам.

Табл. 5. Время выполнения задачи по добавлению нового раздела меню настроек (среднее по двум попыткам).

| Разработчики | Монолитный вариант, мин | Модульный вариант, мин | Ускорение (раз) |
|---|---|---|---|

| | | | |
|---|---|---|---|
| 1 | 72 | 8 | 9.0× |
| 2 | 78 | 10 | 7.8× |
| 3 | 90 | 13 | 6.9× |
| 4 | 66 | 9 | 7.3× |
| 5 | 84 | 10 | 8.4× |
| 6 | 75 | 9 | 8.3× |
| 7 | 81 | 11 | 7.4× |
| 8 | 69 | 8 | 8.6× |
| 9 | 87 | 12 | 7.3× |
| 10 | 73 | 9 | 8.1× |
| 11 | 79 | 10 | 7.9× |
| 12 | 70 | 8 | 8.8× |
| Среднее (SD) | 77 (±7) | 9.8 (±1,5) | ~7.9× |

Табл. 6. Сводные результаты экспериментального исследования.

| Метрика | Задача | Монолитный вариант | Модульный вариант | Улучшение | Стат. значимость |
|---|---|---|---|---|---|
| Сопро-вождае-мость (*n* = 17) | Смена шрифта | 26.4 мин | 0.9 мин | −96.6% (29.3×) | t(16) = 24.8; *p*<0.001 |
| | Смена стиля оформления кнопок | 34.2 мин | 2.1 мин | −93.9% (16.3×) | t(16) = 47.8; *p* < 0.001 |
| | Смена цвето-вой схемы | 31.8 мин | 1.3 мин | −95.9% (24.5×) | t(16) = 22.1; *p* < 0.001 |

| | | | | | |
|---|---|---|---|---|---|
| Производительность (10 запусков) | Пиковое ОЗУ | 436 МБ | 370 МБ | 15.1% (−66 МБ | t(18) = 54.2; p < 0.001 |
| Масштабируемость (*n* = 12) | Добавление нового раздела меню | 77 мин | 9.8 мин | −87.3% (7.9×) | t(11) = 28.6; p < 0.001 |

Относительно метрики сопровождаемости, централизованное управление стилями через иерархию базовых классов сокращает время глобального изменения, например для смены шрифта – в 29.3 раза. Эффект достигается именно за счет введения иерархии стилевых классов: даже при использовании Common UI, но без централизованного хранения параметров оформления каждое изменение потребовало бы ручного редактирования каждого виджета.

Что касается производительности, применение мягких ссылок и асинхронной загрузки через механизм Blueprint Async Action снижает пиковое потребление оперативной памяти при открытии главного меню на 66 МБ (15.1%). Данный эффект накапливается пропорционально числу экранов в игровом проекте и особенно значим для платформ с ограниченными ресурсами (мобильные устройства, консоли прошлого поколения).

Относительно метрики масштабируемости, Data-Driven-подход сокращает для разработчиков игровых проектов трудоемкость добавления нового функционала, в частности при добавлении раздела настроек – в 7.9 раза. Ключевым результатом по данной метрике является полное отсутствие необходимости в изменении существующих Blueprint: расширение системы настроек сводится к конфигурированию реестра данных и добавлению нескольких строк кода в класс настроек.

Таким образом, совокупность полученных результатов подтверждает исходную гипотезу исследования: предложенная модульная архитектура позво-

ляет существенно снизить связанность компонентов пользовательского интерфейса и повысить эффективность разработки и сопровождения игровых интерфейсов на движке Unreal Engine.

**ЗАКЛЮЧЕНИЕ**

Настоящее исследование осуществлено для устранения ряда проблем, возникающих при стихийной разработке пользовательских интерфейсов в Unreal Engine, таких как высокая связанность компонентов, трудоемкость внесения изменений, неэффективное использование памяти и сложность масштабирования. Несмотря на наличие в игровом движке таких мощных инструментов, как Common UI, вопросы их использования для построения целостной и масштабируемой архитектуры остаются малоизученными.

Поставленная цель достигнута – создана и экспериментально апробирована архитектура модульной системы UI, предназначенная для игровых проектов, которые разрабатываются на движке Unreal Engine. Она реализована в виде отдельного плагина AdvancedGameUserInterface, что обеспечивает возможность повторного использования и расширения системы UI в различных игровых проектах.

Результаты экспериментального исследования подтвердили эффективность предложенного решения по трем ключевым метрикам – сопровождаемость, производительность и масштабируемость. В отличие от готовых плагинов, которые ускоряют внедрение, но при этом сложно кастомизируются под специфические требования проекта, созданная архитектура использует проверенные механизмы Common UI и UMG в качестве фундамента, надстраивая собственные уровни абстракции (иерархию стилей, многослойные стеки, адресацию через Gameplay Tags, динамическую систему настроек на основе данных), что позволяет избежать ограничений готовых решений без избыточных трудозатрат.

Предложенное решение имеет несколько ограничений. Прежде всего, архитектура имеет дополнительную сложность на начальном этапе проекта: создание иерархии классов, настройка тегов и реестров данных требуют времени и архитектурного мышления. Это не всегда целесообразно для небольших игровых проектов. Второе ограничение – это порог вхождения для новых разработ-

чиков: переход к многослойной архитектуре требует их обучения. Третье ограничение – это влияние модульной архитектуры на скорость компиляции и сборки игрового проекта в настоящем исследовании не оценивалось и требует дальнейшего изучения.

На основе полученных результатов сформулируем практические рекомендации. Для крупных проектов (AA/AAA) с длительным циклом разработки видеоигры модульная архитектура – это обоснованный выбор, который окупается при масштабировании и снижает затраты на сопровождение. Для проектов среднего размера ее стоит применять выборочно: централизованное управление стилями – в обязательном порядке, многослойные стеки – при сложном интерфейсе с перекрывающимися экранами, систему адресации через Gameplay Tags – при необходимости гибкой конфигурации виджетов. Для небольших проектов и прототипов достаточно выделять стили в отдельные ресурсы, избегать жестких ссылок и использовать мягкие ссылки для асинхронной загрузки.

Среди перспективных направлений для будущих исследований выделим два. Во-первых, это разработка методов автоматического тестирования UI в рамках предложенной архитектуры: модульность и слабая связанность компонентов создают необходимые условия для написания unit- и интеграционных тестов. Во-вторых, исследование применимости предложенных принципов многослойной организации, адресации через теги, Data-Driven-подхода к смежным игровым подсистемам, таким как инвентарь, диалоговые системы, внутриигровые магазины и др. Это позволит унифицировать архитектурные подходы не только для разработки пользовательского интерфейса, но и для всего игрового проекта в целом.

**СПИСОК ЛИТЕРАТУРЫ**

1. *Ullmann G.C., Guéhéneuc Y.-G., Petrillo F., Anquetil N., Politowski C.* SyDRA: An approach to understand game engine architecture // Entertainment Computing. 2025. Vol. 52, No. 100742. https://doi.org/10.1016/j.entcom.2024.100832

2. *Wojtaszczyk D.* Unlocking Cross-Platform UI with Unreal's Common UI. 2025. URL: https://pixelantgames.com/blog/unlocking-cross-platform-ui-with-unreals-commonui/

3. *Chen L.* Architecting UI Systems for Design Fidelity and Platform Consistency // TechRxiv. 2025. URL: https://www.techrxiv.org/users/949700/articles/1322954-architecting-ui-systems-for-design-fidelity-and-platform-consistency

4. YawLighthouse. UMG Slate Compendium: A Compendium Information Guide for UMG and Slate in Unreal Engine. URL: https://github.com/YawLighthouse/UMG-Slate-Compendium/tree/main/languages/ru

5. Epic Games. Common UI Overview.
URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/overview-of-advanced-multiplatform-user-interfaces-with-common-ui-for-unreal-engine

6. Unreal Engine. UMG Best Practices. 2019.
URL: https://www.unrealengine.com/en-US/tech-blog/umg-best-practices

7. *Ramač R., Mandić V., Taušan N., Rios N., Freire S., Pérez B., Castellanos C., Correal D., Pacheco A., Lopez G., Izurieta C., Seaman C., Spinola R.* Prevalence, Common Causes and Effects of Technical Debt: Results from a Family of Surveys with the IT Industry // Journal of Systems and Software. 2022. Vol. 184, No. 111114. https://doi.org/10.1016/j.jss.2021.111114

8. *Nardone V., Muse B.A., Abidi M., Khomh F., Di Penta M.* Video Game Bad Smells: What They Are and How Developers Perceive Them // ACM Transactions on Software Engineering and Methodology. 2023. Vol. 32, No. 4 (88). P. 1–35.
https://doi.org/10.1145/3563214

9. *Agrahari V., Chimalakonda S.* A Catalogue of Game-Specific Anti-Patterns // Proceedings of the 15th Innovations in Software Engineering Conference (ISEC 2022). 2022. P. 1–10. URL: https://dl.acm.org/doi/epdf/10.1145/3511430.3511436

10. *Bittner D., Hauser F., Engl F., Mottok J.* Eye Movement Modelling Examples on Usability Heuristics // Proceedings of the 6th European Conference on Software Engineering Education (ECSEE'25). ACM. 2025. P. 106–114.
https://doi.org/10.1145/3723010.3723035

11. *Olsson T., Toll D., Wingkvist A., Ericsson M.* Evolution and Evaluation of the Model-View-Controller Architecture in Games // Proceedings of the 4th International Workshop on Games and Software Engineering (GAS 2015). IEEE. 2015. URL: https://ieeexplore.ieee.org/document/7169463

12. Microsoft. The Model-View-ViewModel Pattern // .NET MAUI Documentation. 2024.
URL: https://learn.microsoft.com/en-us/dotnet/architecture/maui/mvvm

## DEVELOPMENT OF MODULAR USER INTERFACE ARCHITECTURE FOR UNREAL ENGINE GAMES USING THE COMMON UI PLUGIN-FRAMEWORK

**M. V. Shmatko[1] [0000-0002-7255-8885], A. D. Myakishev[2] [0009-0001-3985-7672]**

[1, 2]*Omsk State Technical University, Omsk, Russia*

[1]marin298@gmail.com, [2]makisevaleksej537@gmail.com

***Abstract***

The paper is devoted to the development and experimental verification of a modular user interface architecture for game projects in Unreal Engine. The study identifies typical UI design antipatterns (monolithic widget structures, hard references, lack of abstraction, platform dependency) and provides an analysis of existing architectural approaches (Data-Driven UI, MVC, MVVM), demonstrating their limitations. The authors propose an architecture that integrates the Common UI framework, an asynchronous loading mechanism with soft references, the Gameplay Tags system, and a data registry for dynamic settings menu generation. The architecture is implemented as the AdvancedGameUserInterface plugin. Its experimental evaluation involving 12 developers and 5 UI designers confirmed the effectiveness of the proposed solution across three key metrics: maintainability, performance, and scalability. Based on the research results, practical recommendations are formulated for applying the developed architecture depending on the scale of the game project. The research findings can be used by developers to improve quality and reduce maintenance costs for user interfaces in game projects on Unreal Engine.

***Keywords:*** *Unreal Engine, user interface, Common UI, modular architecture, Gameplay Tags, Data-Driven UI, antipatterns, maintainability, scalability, performance, game development.*

**REFERENCES**

1. *Ullmann G.C., Guéhéneuc Y.-G., Petrillo F., Anquetil N., Politowski C.* SyDRA: An approach to understand game engine architecture // Entertainment Computing. 2025. Vol. 52, No. 100742. https://doi.org/10.1016/j.entcom.2024.100832

2. *Wojtaszczyk D.* Unlocking Cross-Platform UI with Unreal's Common UI. 2025. URL: https://pixelantgames.com/blog/unlocking-cross-platform-ui-with-unreals-commonui/

3. *Chen L.* Architecting UI Systems for Design Fidelity and Platform Consistency // TechRxiv. 2025. URL: https://www.techrxiv.org/users/949700/articles/1322954-architecting-ui-systems-for-design-fidelity-and-platform-consistency

4. YawLighthouse. UMG Slate Compendium: A Compendium Information Guide for UMG and Slate in Unreal Engine. URL: https://github.com/YawLighthouse/UMG-Slate-Compendium/tree/main/languages/ru

5. Epic Games. Common UI Overview.
URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/overview-of-advanced-multiplatform-user-interfaces-with-common-ui-for-unreal-engine

6. Unreal Engine. UMG Best Practices. 2019.
URL: https://www.unrealengine.com/en-US/tech-blog/umg-best-practices

7. *Ramač R., Mandić V., Taušan N., Rios N., Freire S., Pérez B., Castellanos C., Correal D., Pacheco A., Lopez G., Izurieta C., Seaman C., Spinola R.* Prevalence, Common Causes and Effects of Technical Debt: Results from a Family of Surveys with the IT Industry // Journal of Systems and Software. 2022. Vol. 184, No. 111114. https://doi.org/10.1016/j.jss.2021.111114

8. *Nardone V., Muse B.A., Abidi M., Khomh F., Di Penta M.* Video Game Bad Smells: What They Are and How Developers Perceive Them // ACM Transactions on Software Engineering and Methodology. 2023. Vol. 32, No. 4 (88). P. 1–35.
https://doi.org/10.1145/3563214

9. *Agrahari V., Chimalakonda S.* A Catalogue of Game-Specific Anti-Patterns // Proceedings of the 15th Innovations in Software Engineering Conference (ISEC 2022). 2022. P. 1–10. URL: https://dl.acm.org/doi/epdf/10.1145/3511430.3511436

10. *Bittner D., Hauser F., Engl F., Mottok J.* Eye Movement Modelling Examples on Usability Heuristics // Proceedings of the 6th European Conference on Software Engineering Education (ECSEE'25). ACM. 2025. P. 106–114. https://doi.org/10.1145/3723010.3723035

11. *Olsson T., Toll D., Wingkvist A., Ericsson M.* Evolution and Evaluation of the Model-View-Controller Architecture in Games // Proceedings of the 4th International Workshop on Games and Software Engineering (GAS 2015). IEEE. 2015. URL: https://ieeexplore.ieee.org/document/7169463

12. Microsoft. The Model-View-ViewModel Pattern // .NET MAUI Documentation. 2024. URL: https://learn.microsoft.com/en-us/dotnet/architecture/maui/mvvm

## СВЕДЕНИЯ ОБ АВТОРАХ

***ШМАТКО Марианна Владимировна*** – к. филос. н, доцент кафедры «Математические методы и информационные технологии в экономике» ОмГТУ. Сфера научных интересов – гейм-дизайн, UX-аналитика и юзабилити информационных систем. Сфера практической деятельности – цифровой маркетинг.

***Marianna Vladimirovna SHMATKO*** – Candidate of Philosophical Sciences, associate professor at the Department Mathematical Methods and Information Technologies in Economics, Omsk State Technical University. Research interests: game design, UX analytics and usability of information systems. Practical activities: digital marketing.

email: marin298@gmail.com

ORCID: 0000-0002-7255-8885

***МЯКИШЕВ Алексей Денисович*** – студент ОмГТУ, направление 09.03.02 «Информационные системы и технологии» (профиль – «Цифровые медиатехнологии, виртуальная и дополненная реальность»). Сфера научных интересов – программирование, разработка видеоигр, разработка игровых механик и интерфейсных решений.

***Aleksei Denisovich MIAKISHEV*** – student at the Omsk State Technical University, majoring in 09.03.02 Information Systems and Technologies (specialization: “Digital Media Technologies, Virtual and Augmented Reality”). Research interests: programming, video game development, game mechanics and UI development.

email: makisevaleksej537@gmail.com

ORCID: 0009-0001-3985-7672

*Материал поступил в редакцию 25 июня 2026 года*