Содержание

От издательства 9

Предисловие 10

Глава 1. Почему C в Godot 12

1.1. Роль C# в современной разработке игр 12
1.2. Обзор архитектуры Godot 17
1.3. Управляемая и неуправляемая среда 20
1.4. Когда выбирать C# вместо GDScript 29
1.5. Установка и настройка окружения 33
1.6. Первый минимальный C#-скрипт 36

Глава 2. Понимание движка 39

2.1. Знакомство с Nodes и Scenes 39
2.2. Дерево сцены 42
2.3. Жизненный цикл Node 47
2.3.1. Контракт на исполнение: фазы вместо хаоса 47
2.3.2. Ключевые callback'и и их роли 49
2.3.3. Типичные ошибки и как они убивают архитектуру 50
2.4. Сигналы и поток событий 52
2.4.1. От прямых вызовов к модели «факт – реакция» 52
2.4.2. Как проектировать сигнал: контракт, имя, данные и локальность 53
2.4.3. Типичные ошибки и дисциплина подписок 55
2.5. Память и владение объектами 57
2.5.1. Два времени жизни, которые нужно подружить 57
2.5.2. Дерево сцен как основа владения 57
2.5.3. Висячие ссылки, забытые подписки и утечки – как это лечить 58
2.5.4. Стратегия чистого владения: кто создаёт, тот и уничтожает 59
2.6. Основы отладки 60
2.6.1. Цикл гипотез: от наблюдения к первопричине 61
2.6.2. Инструменты наблюдаемости: как заставить систему заговорить 61

2.6.3. Карта ошибок: связываем симптомы с архитектурными нарушениями 62
2.6.4. Ловушки мышления и профилактическая архитектура 63

Глава 3. C# как инструмент системного дизайна 66

3.1. Классы и ответственность 66
3.1.1. Класс как архитектурный рубеж 66
3.1.2. Связность и цена размытых границ 67
3.1.3. Проектируем классы: от хаоса к ролям 69
3.2. Инкапсуляция 71
3.2.1. Инкапсуляция как архитектурная защита и язык дизайна 71
3.2.2. Как построить надёжный контракт: свойства, методы и инварианты 72
3.2.3. Инкапсуляция на практике: отладка, масштабирование и работа в команде 73
3.3. События и делегаты 75
3.3.1. Делегаты: ссылки на методы как значения 75
3.3.2. События: контролируемая рассылка уведомлений 76
3.3.3. Жизненный цикл подписки и утечки памяти 78
3.3.4. События, прямые вызовы и сигналы Godot: когда что использовать 79
3.4. Интерфейсы: как договариваться о возможностях, не привязываясь к деталям 80
3.5. Асинхронность: как не заморозить игру и не потерять контроль 84
3.5.1. Task и async/await: как работают обещания и паузы 84
3.5.2. Асинхронность в мире Godot: где она уместна, а где опасна 85
3.5.3. Состояние между мирами: главная ловушка await 87
3.5.4. Где асинхронность действительно нужна, а где лучше без неё 88
3.6. Как не превратить проект в тыкву: типичные архитектурные ошибки и их профилактика 88
Шаг 2: прямые зависимости и скрытое знание о структуре сцены 91

Глава 4. Структурирование игровой логики 96

4.1. Разделение ответственности (Separation of Concerns) 96
4.1.1. Concern как единица архитектурной устойчивости 96
4.1.2. Три слоя архитектуры: представление, правила, инфраструктура 97
4.1.3. Цена смешения: как невинные shortcuts превращаются в системный долг 98
4.1.4. От хаоса к границам: рефакторинг на примере инвентаря 99
4.1.5. Модульное мышление и отладка в разделённой системе 101
4.2. Композиция поведения: от монолитных сущностей к набору ролей 101

4.2.1. Роль как архитектурная единица: контракт, состояние и границы ... 102
4.2.2. Контейнер сущности: сборка, оркестрация и связывание ... 104
4.2.3. Гранулярность компонентов и предотвращение архитектурного шума ... 106
4.2.4. Компонентное мышление в контексте Godot: две плоскости композиции ... 106
4.3. Наследование и композиция: два способа выразить вариативность ... 107
4.3.1. Два отношения: «является» и «содержит» ... 108
4.3.2. Ловушки наследования: неявные контракты и комбинаторный рост ... 108
4.3.3. Композиция как инструмент вариативности и динамической замены ... 110
4.3.4. Реалистичный гибрид и практические ориентиры ... 111
4.4. Организация больших проектов: от стихийного кода к управляемой структуре ... 112
4.4.1. От прототипа к сотням файлов: три эффекта разрушения порядка ... 112
4.4.2. Слои, фичи и вертикальные срезы: поиск осей организации ... 113
4.4.3. Границы модулей, неймспейсы и конвенции: удержание порядка в долгой перспективе ... 114
4.4.4. Когда и как перестраивать структуру: триггеры роста и рефакторинг без катастрофы ... 115
4.5. Направление зависимостей: как защитить правила игры от деталей реализации ... 117
4.5.1. Почему ядро не должно знать о периферии: пример с растущей энтропией ... 118
4.5.2. Инверсия зависимостей через контракты и адаптеры ... 119
4.5.3. Профилактика циклов и сохранение дисциплины в долгосрочной перспективе ... 121

Глава 5. Игрок как система ... 123

5.1. Абстракция ввода: учим персонажа слушаться не кнопок, а намерений ... 123
5.1.1. Главная ошибка новичка: рождение «божественного метода» ... 124
5.1.2. Проектирование намерения: контракт, понятный и кнопке, и ИИ ... 124
5.1.3. Контроллер как потребитель намерений: логика без привязки к железу ... 126
5.1.4. Магия буферизации: как простить игроку ошибку на 100 миллисекунд ... 128
5.1.5. Реплеи, AI и мультиплеер: один интерфейс, чтобы править всеми ... 129

5.1.6. Подводные камни: о чём нельзя забывать 130
5.2. Архитектура движения: конвейер от намерения к позиции 130
5.2.1. Конвейер движения: три шага, которые нельзя смешивать 131
5.2.2. Владение скоростью: почему так важно, чтобы у каждой величины был хозяин 133
5.2.3. Детерминизм и отладка: превращаем ощущения в науку 133
5.2.4. Жёсткая граница: почему анимация не имеет права двигать персонажа 134
5.2.5. Собираем всё воедино: как это выглядит в Godot-сцене 135
5.3. PhysicsBody и CharacterBody: выбор физической основы для движения 136
5.3.1. Две модели движения: симуляция физики и контроллер 136
5.3.2. Архитектурные последствия: управляемость, отладка и предсказуемость 137
5.3.3. Практические сценарии и критерии выбора 138
5.4. Реализация машины состояний для движения персонажа 140
5.4.1. От неявных флагов к явным состояниям 141
5.4.2. Минимальный каркас: состояния, переходы и фазы жизни 143
5.4.3. Интеграция с моделью движения и обработка прерываний 145
5.4.4. Проектирование переходов и предотвращение хаоса 147
5.5. Разделение поведения и анимации: односторонний поток данных 148
5.5.1. Ловушка обратной связи: когда анимация начинает управлять поведением 148
5.5.2. Контракт визуального представления: факты вместо команд 149
5.5.3. Синхронизация, тайминги и проблема root motion 151
5.5.4. Влияние на командную работу, отладку и мультиплеер 152

Глава 6. Мир и взаимодействие 154

6.1. Коллизионные слои и маски: проектирование политики взаимодействия 154
6.1.1. Декларация ролей и политики обнаружения 154
6.1.2. Разделение каналов: блокировка и обнаружение 156
6.1.3. Проектирование политики через константы и владение 157
6.1.4. Архитектурная дисциплина: избавляемся от хаоса в коде 158
6.2. Системы взаимодействия: контрактная модель вместо прямых вызовов 159
6.2.1. Протокол взаимодействия: инициатор, цель и контекст 159
6.2.2. Разделение обнаружения и исполнения 161
6.2.3. Отказы как часть контракта и расширяемость системы 163
6.3. Архитектура камеры: наблюдатель, а не контроллер 164
6.3.1. Разделение обязанностей игрока и наблюдателя 164
6.3.2. Слойный пайплайн: от цели до финального кадра 165

6.3.3. Управление режимами и модификаторами 167
6.4. Событийно-ориентированный геймплей: факты вместо опроса 168
6.4.1. От постоянного опроса к подписке на изменения 168
6.4.2. Область видимости событий и предотвращение хаоса 170
6.4.3. Синхронный вызов против отложенной обработки 171
6.5. Базовая структура AI: конвейер восприятия, решения и действия 172

Глава 7. State Machine и проектирование поведения 179

7.1. Конечный автомат как фундамент игровой логики 179
7.2. Иерархические конечные автоматы: управление глубиной поведения 184
7.3. Стратегия как механизм вариативности алгоритмов 188
7.4. Связывание модулей через уведомления: Наблюдатель 193

Глава 8. Event-Driven архитектура 196

8.1. Сигналы Godot и события C# – два инструмента для уведомлений 196
8.2. Глобальная шина событий: централизованная коммуникация в масштабах всей игры 199
8.3. Техники ослабления связанности: контракты, события и границы модулей 202
8.4. Тестирование систем в изоляции как архитектурный инструмент 206

Глава 9. UI как отдельный слой 210

9.1. Архитектура UI как слоя представления 210
9.2. Привязка данных в UI: от ручного хаоса к реактивной синхронизации 212
9.3. Удержание UI в роли слоя представления 216

Глава 10. Системы сохранения и загрузки 221

10.1. Сериализация как контракт данных 221
10.2. Версионирование сохранений: эволюция данных без потери прогресса 224
10.3. Безопасность сохранений: защита данных в недоверенной среде 227
10.4. Асинхронные системы сохранения 230

Глава 11. Производительность и оптимизация 234

11.1. Профилирование как инженерная дисциплина 234
11.2. Снижение аллокаций: управление памятью в горячих путях 237
11.3. Объектные пулы: переиспользование вместо создания 240
11.4. Динамическое управление сценами: жизненный цикл мира 243

Глава 12. Подготовка к релизу и сопровождение проекта 247

12.1. Экспорт как снятие маски с неявных зависимостей 248
12.2. Конфигурация как инструмент управления окружениями 252
12.3. Debug и Release: два разных продукта 256
12.4. От прототипа к архитектуре: реконструкция и рефакторинг 260
12.5. Код-ревью, дрейф и долгосрочное сопровождение 264

Глава 13. Las Vegas Digital Hub: архитектура уровня продакшен на практике 268

13.1. Архитектурный манифест: уход от стандартных скриптов 269
13.2. Система состояний: State Machine для игрового автомата 274
13.3. Data-Driven подход: Resource-пресеты 285
13.4. Событийная модель: Event Bus для децентрализованной синхронизации 291
13.5. Многопоточность и RNG: безопасные вычисления вероятностей 297
13.6. Оптимизация в условиях Лас-Вегаса: проверенные практики 303

Заключение 312

Предметный указатель 313

От издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте *www.dmkpress.com*, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу *dmkpress@gmail.com*; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу *http://dmkpress.com/authors/publish_book/* или напишите в издательство по адресу *dmkpress@gmail.com*.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу *dmkpress@gmail.com*. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты *dmkpress@gmail.com*.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Предисловие

Люди начинают делать игры, чтобы увидеть движение на экране. Первый же персонаж, дёрнувшийся в ответ на нажатие клавиши, приносит ни с чем не сравнимое ощущение: мир оживает, и вы – его создатель. Но между этим детским восторгом и способностью построить игру, которая не развалится через месяц разработки, лежит пропасть. Большинство прыгает в неё, не оглядываясь, и разбивается о скалы хаотичного кода. Перед вами книга для тех, кто хочет не просто перепрыгнуть, а построить мост. Мы живём в эпоху быстрых решений. Короткие видео, пошаговые туториалы, готовые шаблоны убеждают нас, что главное – результат. Но результат без понимания – карточный домик. Вы можете за десять минут заставить персонажа прыгнуть, следуя инструкции. Вы даже можете собрать целый прототип, удивляющий друзей. Но когда проект начнёт расти, а требования – множиться, хаотичный код перестанет работать. Система станет хрупкой, изменения начнут ломать логику в самых неожиданных местах, и вы поймёте: проблема не в языке и не в движке, а в отсутствии архитектуры.

Книга адресована тем, кто хочет начать с нуля, но не желает оставаться на уровне случайных решений. Новичкам, только знакомящимся с C# и Godot. Разработчикам, переходящим с других движков, например Unity или Unreal. Инженерам, которым важны структура и предсказуемость системы.

Почему C#? За пределами игровой индустрии это язык, на котором строят распределённые системы, облачные сервисы и корпоративные приложения. Его статическая типизация не враг, а союзник: компилятор ловит ошибки до того, как они добираются до игрока. Чёткая структура классов, управляемая среда выполнения, десятилетия проверенных инженерных практик – всё это создаёт фундамент, на котором можно возводить здание любой сложности. Godot с его открытостью и гибкостью идеально ложится на этот фундамент, а связка двух инструментов становится мощнее каждого по отдельности.

Почему архитектура? Игра – не сцена и не набор спрайтов. Это совокупность взаимодействующих систем. Игрок, камера, ввод, физика, интерфейс, сохранения – все эти части должны работать согласованно, как оркестр, а не как толпа солистов. С правильной структурой даже небольшой проект остаётся устойчивым и готовым к расширению. Если же структурой пренебречь, даже простая игра быстро превратится в хаос, где каждая правка вызывает лавину новых ошибок.

Книга начинается с нуля, но ноль здесь – не отсутствие знаний, а чистая точка старта. Мы будем разбирать не только «как сделать», но и «почему

устроено именно так». Каждая строчка кода, каждый архитектурный принцип появляются здесь по определённой причине, и задача читателя – эту причину понять.

Для кого эта книга

Издание рассчитано на несколько категорий читателей. Начинающие разработчики, которые хотят стартовать с правильной основы. Если у вас нет опыта в Godot или C#, книга проведёт от базовых принципов к системному пониманию движка. Разработчики, переходящие с других движков, например Unity или Unreal. Если вы уже знакомы с геймдевом, но хотите освоить Godot и использовать C# как основной инструмент, книга поможет быстро адаптироваться к архитектуре движка. Инженеры и программисты, которым важна структура. Если вы предпочитаете продуманную организацию кода, разделение ответственности и масштабируемые решения, материал будет соответствовать вашему подходу. Книга может **не подойти**, если вы ищете быстрый результат без понимания системы, хотите лишь повторять шаги за автором или вам достаточно минимального прототипа без перспективы дальнейшего развития.

Как работать с книгой

Материал построен последовательно, каждая часть опирается на предыдущую. Рекомендуется читать главы по порядку, особенно при старте с нуля. Даже знакомые на первый взгляд темы закладывают принципы, которые будут использоваться дальше.

Пишите код вручную. Не копируйте примеры автоматически, набирайте их самостоятельно. Тактильная память и необходимость обдумывать каждую строку дают более глубокое понимание структуры. Вы заметите детали, которые легко пропустить при копировании, и сформируете привычку аккуратного проектирования. Код в книге не является «магией» – каждая строка появляется по определённой причине.

Не пропускайте архитектурные разделы. В книге есть главы с меньшим объёмом практики и более глубоким анализом структуры – именно они формируют системное мышление. Если пропустить эти разделы, проект может работать, но понимание останется поверхностным.

Работайте в своём темпе. Материал рассчитан на постепенное освоение. Некоторые главы можно перечитывать после завершения книги – по мере роста опыта многие принципы начнут восприниматься иначе. Цель издания – не только дать инструменты, но и научить мыслить категориями систем. При осознанном подходе к чтению результат окажется значительно глубже, чем просто готовый проект.

ГЛАВА

1

Почему C в Godot

1.1. Роль C# в современной разработке игр

Когда говорят о языках в геймдеве, спор обычно сводится к двум полюсам: скорость прототипирования и производительность исполнения. В реальных проектах язык гораздо чаще определяет не быстроту обсчёта кадров, а скорость и безопасность изменения системы. А это уже характеристика архитектурная – она диктует, насколько предсказуемо можно добавлять механики, переносить ответственность между подсистемами, заменять реализации и удерживать проект в состоянии развития, а не набора исключений. C# занял в индустрии нишу языка, позволяющего строить игровые системы на уровне продукта. Его положение сложилось на пересечении трёх факторов: управляемая среда выполнения, статическая типизация и зрелая инженерная экосистема. Как следствие, C# выбирают для задач, в которых важны не единичные трюки, а дисциплина изменений – контролируемые зависимости, минимизация регрессий, обозримая сложность и способность поддерживать кодовую базу годами.

Выбор языка как фундамент архитектуры

Выбор языка всегда означает выбор набора ограничений и инструментов, которые будут подталкивать архитектуру в одни стороны и делать болезненными другие. Два языка, способных выразить одинаковую логику, могут радикально различаться стоимостью ошибок и ценой улучшений. Архитектурное измерение проявляется здесь в нескольких конкретных вещах.

Первое – контракты и границы. Чем проще выражать интерфейсы, типы данных и ограничения на входы и выходы, тем легче фиксировать границы подсистем и удерживать направление зависимостей. Второе – обратная

связь: компиляция, анализаторы, подсказки IDE и тестовые фреймворки определяют, насколько быстро разработчик получает сигнал «ты сломал систему» и насколько точно этот сигнал локализован. Третье – рефакторинг как нормальная операция. В растущем проекте переписывание частей системы не исключение, а рутина, и язык либо поддерживает безопасные переименования, поиск использований и типовые преобразования, либо превращает изменения в серию ручных проверок. Далее идёт модель исполнения и цена абстракций: управляемая память, сборка мусора, JIT/AOT, исключения и асинхронность формируют своего рода «физику» кода, которую архитектура обязана учитывать. Наконец, интеграция с инструментами и процессами – код-ревью, CI, статический анализ, профилирование, форматирование и генерация документации – является не украшением, а частью производственного контура, и игнорировать её при выборе языка нельзя. За выбором языка по принципу «на нём удобнее писать» фактически стоит выбор архитектуры по умолчанию: неявные правила композиции кода, стиль выражения зависимостей и типовые паттерны, которые будут воспроизводиться десятки раз. В небольшом прототипе такой выбор почти незаметен. В системе из десятков тысяч строк он превращается в главный источник либо порядка, либо хаоса.

Статическая типизация как инструмент фиксации договорённостей

Статическая типизация в C# полезна не сама по себе, а как механизм фиксации договорённостей. Типы здесь играют три взаимосвязанные роли. Прежде всего они дают сжатое описание данных: тип задаёт форму, с которой должна работать подсистема, и тем самым снижает число плавающих соглашений, существующих только в голове разработчика или в комментариях. Кроме того, типы служат инструментом локализации ошибок – компилятор и анализаторы находят несоответствия там, где динамический код часто падает далеко от источника проблемы. Архитектурно это означает более короткую цепочку причинно-следственных связей. Наконец, они создают основание для безопасного рефакторинга: когда типы и сигнатуры отражают реальные границы, IDE может проводить изменения механически корректно, и чем больше проект, тем сильнее это влияние на стоимость развития. Из этого следует важный вывод: типы не служат ни украшением, ни формальной строгостью; они выступают частью архитектурного языка проекта. Если подсистемы общаются через размытые структуры вроде «объекта со случайными полями» или «словаря строк», любая попытка модульности будет постоянно протыкаться исключениями и скрытыми зависимостями. Если же границы выражены явными типами, меняется сама динамика работы: спор о том, что передаётся в подсистему, переходит из неформального общения в проверяемое условие. Допустим, вы вводите отдельный тип PlayerId вместо string. Компилятор немедленно подсветит все места, где идентификатор игрока пытались передать как обычную строку или, хуже того, как целое число. Десятки потенциальных

багов устраняются ещё до первого запуска. Статическая типизация не гарантирует хорошую архитектуру автоматически – она лишь делает видимыми противоречия. Можно выстроить иерархию типов, которая создаёт иллюзию структуры, но на деле усиливает связанность. Поэтому полезный подход состоит в том, чтобы начинать не с классов, а с ответственностей и потоков данных, и лишь затем выбирать минимально достаточные типы для фиксации границ. Современная эволюция C# – nullable-аннотации, record-типы, pattern matching, улучшения в generics – расширяет набор средств для выражения инвариантов, но суть остаётся прежней: сделать недопустимые состояния труднее выразимыми, а допустимые – проще проверяемыми.

В игровой разработке множество состояний, которые почти всегда должны быть валидными: здоровье не уходит ниже нуля, идентификатор сущности не пуст, ссылка на ресурс указывает на существующий контент, команда содержит полный набор параметров. В прототипе такие требования держат в голове, но в системе их необходимо фиксировать. C# позволяет закреплять инварианты на нескольких уровнях строгости. На уровне API сигнатура метода и типы параметров задают, что именно ожидается – отдельный тип для идентификатора, отдельный для времени, отдельный для расстояния. Всё это не избыточная формальность, а способ уменьшить число неявных соглашений. На уровне структуры данных необязательные поля, явные значения по умолчанию и различие между отсутствием значения и нулевым значением делают такие контракты проверяемыми. На уровне поведения к ним добавляются предусловия, постусловия, ограничение способов создания объекта через factory-методы и закрытие сеттеров там, где состояние не должно меняться произвольно.

Системный эффект возникает тогда, когда язык проекта начинает совпадать с языком предметной области. Если в коде есть понятия «команда взаимодействия», «результат симуляции», «снимок состояния», «правило начисления очков», обсуждение архитектуры становится конкретным: можно указать, какие подсистемы имеют право создавать эти сущности, кто их читает, кто трансформирует и где заканчивается ответственность. C# хорошо подходит для такого словаря за счёт выразительных типов и средств организации кода.

Среда исполнения, экосистема и масштабирование

Языковая среда выполнения – такая же часть архитектуры, как и система типов. Управляемая память и сборка мусора создают характерный профиль рисков, приводя не к классической утечке памяти, а к непредсказуемым паузам, росту аллокаций, фрагментации управляемой кучи, ошибкам из-за захвата ссылок и долгоживущих объектов. Архитектурный вывод прост: горячие пути должны проектироваться и измеряться. Если логика распределена случайно, невозможно понять, где и почему происходят аллокации и растёт давление на GC. При ясных границах подсистем, напротив, легче ограничить создание временных объектов в критических местах, заранее

выбрать между value types и reference types там, где это действительно важно, и локализовать оптимизации, не переписывая весь проект. Другая сторона модели исполнения – публикация под разные платформы. У проектов могут быть различные требования: быстрые итерации на ПК, жёсткие ограничения на платформах с AOT-компиляцией, лимиты на размер сборки, запрет динамической генерации кода и reflection. Эти требования не решаются «в конце», а влияют на выбор подходов к сериализации, конфигурации, инструментам и способам расширения системы. В современном производственном геймдеве язык представляет собой связку из синтаксиса, компилятора, IDE, экосистемы анализаторов, тестового контура, сборки и публикации. C# превосходит многие альтернативы именно зрелостью и хорошей интеграцией этой связки. Ключевые свойства, которые её отличают: навигация и понимание кода через быстрый переход к определению, поиск использований и граф вызовов; рефакторинги как операции первого класса – извлечение метода, изменение сигнатур, безопасное переименование; статический анализ, включающий предупреждения о null, неиспользуемых переменных и сомнительных преобразованиях; развитая инфраструктура тестирования – unit-тесты, моки, измерение покрытия. Практическим критерием зрелости инструментов служит способность поддерживать массовые изменения. Если инструментальный контур позволяет выполнять их последовательно, с компиляцией и тестами, команда получает возможность улучшать архитектуру по мере роста. Если контур слаб, появляется стратегия «не трогать то, что работает», и архитектура закрепляется в раннем, случайном состоянии.

Экосистема C# – это прежде всего NuGet и культурная привычка оформлять повторно используемый код как пакет с версионированием. Лёгкость подключения зависимостей может как ускорить разработку, так и увеличить связанность. На сторонние библиотеки полезно смотреть как на экономический выбор: каждая добавляет поверхность обновлений и потенциальных конфликтов, приносит свою модель данных и философию API. Для прототипа быстрое подключение пакета часто рационально. Для системы важнее, чтобы зависимость закрепляла границу, а не размывала её. Если библиотека заставляет протаскивать свои типы через весь проект, она становится частью архитектуры независимо от намерений. Если же она локализована в одном слое и скрыта за собственным интерфейсом, проект сохраняет управляемость. В одиночных проектах архитектура часто подменяется личной памятью автора: он знает, где что лежит, и держит в голове неявные зависимости. С ростом проекта и числа участников эта опора исчезает. Возникает потребность в формализованных правилах: что считается публичным API подсистемы, какие данные служат входом и выходом, где заканчивается ответственность одного слоя и начинается другого. C# помогает такому масштабированию. Он делает выразимыми границы через интерфейсы, абстрактные классы, generics и модульность через сборки, а также позволяет частично перевести архитектурные правила из устных соглашений в машинную валидацию через анализаторы, форматирование и CI.

Прототипный код против системного: разница в стоимости изменений

Прототипный код решает задачу «проверить идею». Он допускает временные зависимости с пометкой «потом разберёмся», жёстко прошитые параметры, дублирование логики и смешение слоёв – UI, геймплей, сохранения и отладка в одном месте. Такой код полезен ровно до момента, когда проект перестаёт быть экспериментом. Переход заметен по симптомам: добавление механики требует трогать всё сразу, исправление бага ломает другое, любое изменение вызывает цепочку неожиданных последствий. Системный код решает иную задачу: делать изменения предсказуемыми. Его ключевая единица – не функция и не класс, а граница: что служит входом для подсистемы (данные, события, команды), что служит выходом (состояние, события, результаты вычислений), что скрыто внутри и может меняться без каскада правок. Разница между прототипом и системой заключается в стоимости изменения. Прототип даёт выигрыш в скорости первого результата, а система – в скорости десятого изменения. Архитектурный смысл выбора C# состоит в том, что он поддерживает системный режим работы: позволяет фиксировать границы, проверять их и безопасно перерабатывать. Важно не путать структуру со сложностью. Плохая архитектура проявляется не в отсутствии слоёв, а в отсутствии ясных причин, по которым эти слои существуют. В C# легко создать инфраструктуру, которая выглядит солидно, но не помогает. Поэтому ориентиром должны быть не паттерны как таковые, а наблюдаемая цель: уменьшение связанности, локализация изменений, ясные контракты и управляемое состояние. Игровой проект – набор взаимодействующих процессов: ввод, симуляция, визуализация, аудио, интерфейс, сохранения, загрузка контента, сетевое взаимодействие. Даже в одиночной игре эти процессы неизбежно пересекаются. Если не определить, как именно они связаны, связи возникают случайно – там, где оказалось удобно прямо сейчас. *Структурное мышление* – это привычка отвечать на вопросы до написания кода. Какая часть системы владеет состоянием? Где проходит граница между доменной логикой и представлением? Какие зависимости должны быть направлены внутрь, к домену, а какие – наружу, к инфраструктуре? C# предоставляет для этого выразительные средства, а также инструментальную поддержку, чтобы ограничения не забывались со временем. Полезно различать два режима архитектурных решений. Решения уровня домена – что такое действие игрока, правило, событие, результат – должны быть стабильными и понятными, даже если реализация меняется. Решения уровня исполнения – как часто обновляется подсистема, где хранятся кеши, как формируются данные для отрисовки – должны быть измеряемыми и заменяемыми без изменения доменных контрактов. Если эти два уровня перепутаны, язык уже не спасает: оптимизации расползаются по коду, а логика превращается в набор условий «для производительности». Хорошая архитектура, напротив, позволяет в одном месте говорить о правилах игры, а в другом – о стоимости их вычисления. C# удобен тем, что поддерживает оба слоя: выразительные доменные типы и практич-

ные инструменты для измерения и оптимизации. Современная разработка игр часто включает несколько дисциплин: геймдизайн, программирование, инструменты, пайплайн контента, тестирование, аналитику. Язык, на котором описываются системы, становится общей точкой соприкосновения. C# играет роль инженерного языка общего назначения, который остаётся достаточно высокоуровневым для быстрой разработки и достаточно строгим, чтобы удерживать большие проекты. Он не отменяет необходимости думать о структуре, но снижает трение там, где структура должна быть закреплена технически. Переход от кода, который просто делает что-то на экране, к коду, описывающему систему, начинается с осознания того, что язык является частью архитектуры, а не просто сменным синтаксисом.

1.2. Обзор архитектуры Godot

Любой игровой движок удобнее всего рассматривать как архитектурную платформу, которая чётко разделяет обязанности между универсальной, высокопроизводительной инфраструктурой и уникальным кодом конкретной игры. Инфраструктура забирает себе повторяющиеся, сложные и требовательные к ресурсам задачи: взаимодействие с графическим API, расчёт физики, управление памятью, обработку ввода с разных устройств. Игровой код, в свою очередь, сосредоточен на доменных правилах, уникальном состоянии и поведении, которые, собственно, и отличают симулятор фермы от космического шутера. Это разделение является фундаментальной причиной существования движков. Они избавляют разработчика от необходимости переписывать технический фундамент для каждого нового проекта и позволяют строить систему на основе уже отлаженных и оптимизированных подсистем. Godot в этой рамке предстаёт как многослойная система с ясной иерархией. Нижние слои реализуют «физику исполнения»: управление потоками, памятью, ресурсами, математические библиотеки и низкоуровневые API. Над ними надстраиваются профильные подсистемы – рендеринг, физика, аудио. Верхние слои предоставляют модель сцены и среду для скриптинга, где и живёт пользовательская логика. У каждого слоя свои инварианты, своя зона ответственности и своя цена ошибок. Чем яснее разработчик представляет себе эти границы, тем проще ему принимать верные архитектурные решения: где хранить состояние, как организовать обмен событиями, какие части системы можно безболезненно заменить, а какие образуют её несущий каркас.

Ядро, подсистемы и их роль в формировании игрового мира

С точки зрения архитектуры движок выполняет четыре ключевые роли. Он предоставляет набор сервисов времени выполнения, задавая главный цикл обновления, точки входа, очереди событий, модель ресурсов и правила взаи-

модействия с операционной системой. Он содержит профильные подсистемы с чёткими контрактами: рендеринг ожидает данные определённой формы, физика обрабатывает коллайдеры и параметры среды, аудио работает с потоками и источниками. Движок также предлагает модель композиции мира – способ описывать существующие в игре объекты и их связи через иерархии, сцены и ресурсы. Наконец, он включает мощный инструментальный контур: редактор, импорт ассетов, сериализацию и средства отладки. Для архитектуры критически важно понимать, что инструменты задают формат данных и влияют на процессы изменений, поэтому считать их чем-то второстепенным было бы опасной ошибкой.

Ядро Godot написано на C++. Это не идеологическое, а сугубо практическое решение, гарантирующее максимальную близость к платформе и полный контроль над расходами процессора и памяти. На этом уровне решаются задачи, для которых цена абстракций и накладных расходов критически высока: прямое взаимодействие с графическими API, управление ресурсами, базовая математика, работа с файловой системой и сетевые примитивы. Принципиально, что C++ служит здесь не языком для описания геймплея, а средством создания надёжной и быстрой инфраструктуры. Ядро предоставляет верхним слоям стабильные сервисы, и пользовательский код на C# живёт над этими сервисами, используя их через публичные контракты, а не подменяя их. Внутри движка удобно мыслить категориями подсистем, каждая из которых отвечает за свою часть работы с собственными типами данных и временными ограничениями. Рендеринг преобразует состояние мира в изображение, выступая в роли потребителя данных, почти никогда не являясь источником истины. Физика, напротив, нередко сама становится частью этой истины, определяя итоговое положение объектов и характер их столкновений. Поэтому игровая логика обязана уважать временную модель физики и границы её ответственности. Аудио, как и рендеринг, в основном потребляет состояние, реагируя на события и параметры, но требует неразрывной буферизации, из-за чего внезапные задержки становятся сразу заметными игроку. Общая закономерность такова: все подсистемы движка – это специализированные вычислительные машины с собственными шкалами времени, и задача архитектуры игры – грамотно формировать для них данные и читать результаты, не позволяя подсистемам превращаться в «глобальный мозг» игры.

Сцены, ресурсы и время как архитектурные абстракции

Над ядром и подсистемами располагается слой скриптинга. Здесь, в окружении C#, размещается логика конкретной игры: её правила, состояния, реакции на события и координация подсистем. В Godot этот слой полностью отделён от ядра и общается с ним через публичный API. Архитектурно это крайне важная граница, означающая, что игра не должна опираться на внутренние детали движка. Иначе любое обновление Godot или смена платформы станет огромным риском. Скриптовый слой служит адаптером между уникальным

Конец ознакомительного фрагмента.
Приобрести книгу можно
в интернет-магазине
«Электронный универс»
e-Univers.ru