Heroes-3 Battle

Учебное пособие: детальный разбор проекта в Unity

Скрипты · Техники визуализации · Устройство под капотом Unity

Материал для студентов · C# / Unity / Built-in Render Pipeline

Учебное пособие по проекту «Heroes-3 Battle»

Детальный разбор кодовой базы, техник визуализации и устройства проекта в Unity

> **Для кого это пособие.** Для студентов, которые уже знакомы с базовым C# и основами Unity (что такое `GameObject`, `MonoBehaviour`, `Transform`, как запускать сцену), но хотят понять, как из этих кирпичиков собирается настоящая пошаговая тактическая игра в духе «Heroes of Might & Magic III». Мы разберём **каждый** скрипт проекта, объясним **почему** он написан именно так, и покажем, **за счёт каких возможностей Unity** всё это работает.
>
> **Как читать.** Пособие состоит из трёх больших частей (по числу вопросов задания) плюс приложения. Части можно читать подряд как книгу или выборочно как справочник — каждый раздел самодостаточен и содержит ссылки на связанные темы.

Оглавление

Приложения - A. Полный словарь терминов - B. Упражнения и задания для самостоятельной работы - C. Частые ошибки и как их избежать - D. Что можно улучшить и куда развивать проект

Часть 0. Введение и общая картина

0.1. Что это за проект

Проект — это **боевая система пошаговой тактической игры**, вдохновлённая классической «Heroes of Might and Magic III» (далее — H3). Он воспроизводит ту сцену, которую все помнят: два войска стоят на **гексагональном поле боя**, между ними разбросаны препятствия (камни, деревья, кусты), и отряды по очереди ходят и бьют друг друга, пока одна из сторон не будет уничтожена.

Ключевые возможности, которые уже реализованы в коде:

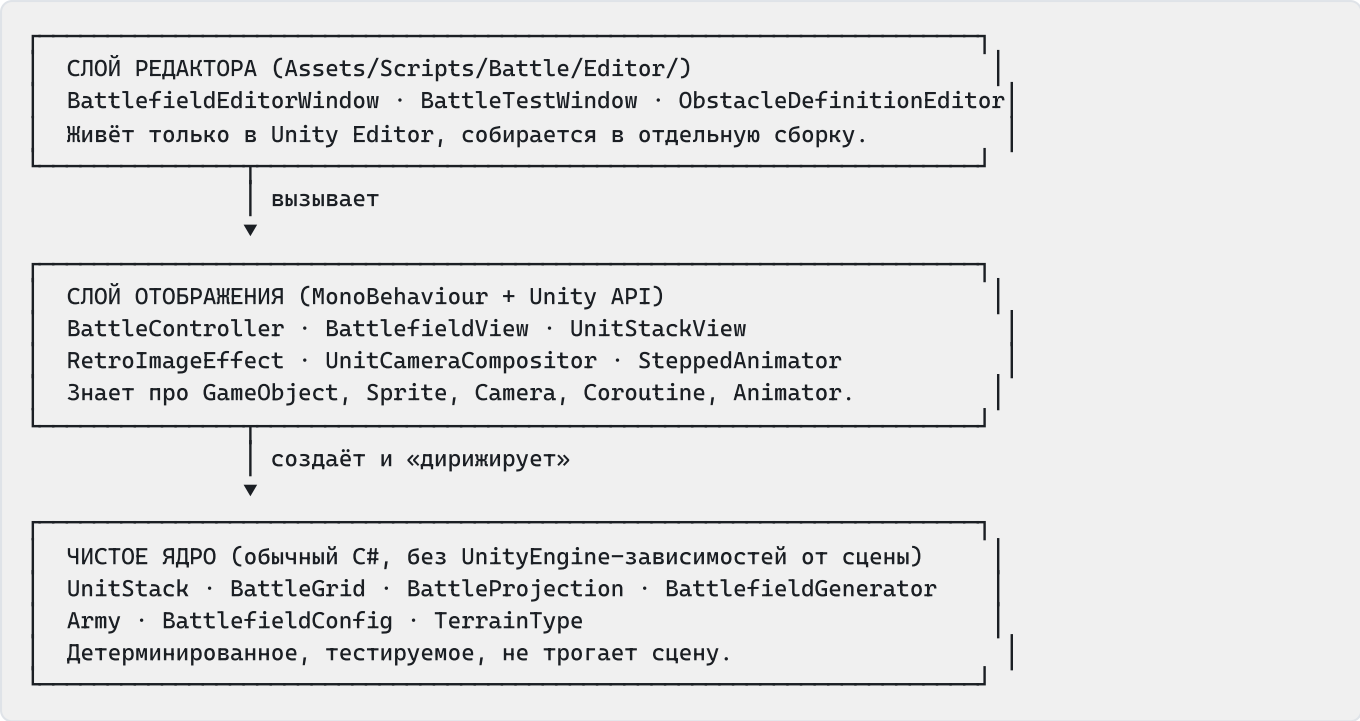
1. **Процедурная генерация поля боя.** По одному числу-«семени» (seed) и типу местности (`Grass`, `Snow`, `Lava`, …) детерминированно строится расстановка препятствий и выбирается фоновая картинка. Алгоритм повторяет логику open-source движка VCMI (свободная реализация H3).
2. **Две армии по 7 «слотов».** Как в оригинале, у каждой стороны 7 ячеек, в каждой — тип существа и его численность (например, «40 зомби»).
3. **Гексовая логика боя.** Поиск достижимых клеток в пределах скорости отряда (обход в ширину, BFS), вычисление врагов в зоне удара, построение пути, поддержка «широких» двухклеточных существ.
4. **Пошаговый цикл боя** с порядком инициативы по скорости, ручным управлением (клик мышью — переместиться / атаковать, пробел — пропустить ход) и формулой урона.
5. **Псевдо-3D визуализация.** Существа — это **3D-модели с анимацией**, но стоят они на **2D-поле**: позиция берётся из плоской гексовой сетки, а поворот вычисляется так, будто мы смотрим на них наклонной камерой. Это фирменный приём H3 — «двухмерное поле, трёхмерные бойцы».
6. **Ретро-стилизация картинки.** Пикселизация, ограниченная палитра (постеризация), плоское (cel) освещение моделей и **пошаговая анимация** в ~10 кадров/с, чтобы движения «щёлкали» позами, как спрайты 1999 года.
7. **Инструменты для дизайнера.** Три окна редактора Unity (генератор поля, тестовый бой, автоматический расчёт заблокированных гексов по спрайту препятствия) позволяют настраивать игру **без единой строчки кода**.

Помимо боевой системы, в проекте есть два самостоятельных учебных скрипта — классический **Pong** (`PongGame.cs`) и пустой шаблон `Example.cs`. Они не связаны с боем, но мы их тоже разберём: Pong — отличный пример «ручной физики» без Rigidbody.

> **Важно понимать с самого начала.** Проект намеренно разделён на **чистую игровую логику** (обычные C#-классы, которые ничего не знают про Unity) и **тонкий слой отображения** (`MonoBehaviour`, спрайты, шейдеры). Это центральная идея всей архитектуры, и мы будем возвращаться к ней постоянно. Такое разделение — не прихоть, а способ сделать код тестируемым, предсказуемым и понятным.

0.2. Общая архитектура: три слоя

Мысленно разложим весь код боевой системы на три горизонтальных слоя. Стрелки показывают, кто на кого «смотрит» (зависит):



- **Чистое ядро** — это «мозг» игры. Здесь живут правила: как ходит отряд, кого он может атаковать, сколько урона наносит, как расставляются препятствия. Эти классы **не создают `GameObject`-ов, не двигают `Transform`-ы и не рисуют спрайты**. Многие из них — обычные C#-классы (`sealed class UnitStack`), а не `MonoBehaviour`. Некоторые (`BattlefieldGenerator`, `BattleProjection`) вообще статические и не хранят состояния. Ими можно пользоваться из юнит-тестов, из консольного приложения — откуда угодно.

> ⚠️ Оговорка: «чистое» здесь означает «не зависит от **сцены**». Часть этих классов всё же использует лёгкие типы из `UnityEngine` — `Vector2Int`, `Mathf`, `Quaternion`. Это структуры и утилиты, не привязанные к игровому объекту; их можно вычислять хоть в фоновом потоке. А вот `System.Random` (а **не** `UnityEngine.Random`) в генераторе выбран сознательно — чтобы генерация была независимой от глобального состояния движка и полностью повторяемой.

- **Слой отображения** — это «тело» игры. Он берёт решения ядра и превращает их в то, что видит игрок: инстанцирует модели, двигает их по экрану, проигрывает анимации, накладывает ретро-эффект. Здесь и только здесь используются `MonoBehaviour`, корутины, `Camera`, `SpriteRenderer`, `Animator`.
- **Слой редактора** — это «мастерская». Он существует только в Unity Editor и нужен дизайнеру, чтобы генерировать поле, запускать тестовые бои и заполнять данные. В собранной игре (билде) этого слоя нет вообще — код в папке `Editor/` в билд не попадает.

Почему это важно для студента. Когда вы смотрите на незнакомый класс, первый вопрос — «к какому слою он относится?». Если это ядро — ищите чистую логику и математику, не ждите работы со сценой. Если это отображение — ищите `MonoBehaviour`, корутины и Unity-вызовы. Такое деление сразу задаёт правильные ожидания и резко упрощает чтение кода.

0.3. Карта файлов проекта

Ниже — все скрипты и шейдеры проекта с одной строкой назначения. Полный разбор каждого — в Части 1 и Части 2.

`Assets/Scripts/Battle/` — ядро + отображение боя

Файл	Слой	Назначение
`TerrainType.cs`	данные	Перечисление 10 типов местности.
`BattlefieldConfig.cs`	данные	Размеры поля (15×11), эталонные размеры гекса, результат генерации.
`ObstacleDefinition.cs`	данные (SO)	Один тип препятствия: спрайт, какие гексы блокирует, вес.
`ObstacleDatabase.cs`	данные (SO)	Каталог всех препятствий и фонов, параметры генерации.
`UnitDefinition.cs`	данные (SO)	Прототип существа: спрайт/префаб и боевые характеристики.
`Army.cs`	данные	Армия из 7 слотов (`ArmySlot`), плюс перечисление `BattleSide` .
`UnitStack.cs`	ядро	Конечный автомат одного развёрнутого отряда (позиция, здоровье, состояние).
`BattleGrid.cs`	ядро	Гексовая геометрия, занятость клеток, BFS-поиск пути и целей.
`BattlefieldGenerator.cs`	ядро	Детерминированная расстановка препятствий по seed.
`BattleProjection.cs`	ядро	Математика поворота модели под «виртуальную наклонную камеру».
`BattleController.cs`	отображение	Фасад боя и движок ходов (корутина цикла раундов).
`BattlefieldView.cs`	отображение	Материализует `BattlefieldConfig` на сцене (фон, гексы, препятствия).
`UnitStackView.cs`	отображение	Визуальное представление одного `UnitStack` (модель, анимации).
`UnitViewSettings.cs`	данные	Настройки отображения моделей (угол наклона, FPS анимации, имена параметров аниматора).
`SteppedAnimator.cs`	отображение	Заставляет `Animator` обновляться с низким FPS → stop-motion.
`RetroImageEffect.cs`	отображение	Пост-эффект камеры: пикселизация + постеризация.
`UnitCameraCompositor.cs`	отображение	Изолирует пост-эффект камеры юнитов через `RenderTexture` .

`Assets/Scripts/Battle/Editor/` — инструменты дизайнера

Файл	Назначение
`BattlefieldEditorWindow.cs`	Окно «Battlefield Generator»: генерирует случайное поле в открытой сцене.
`BattleTestWindow.cs`	Окно «Battle Test»: заполнить обе армии и запустить бой.
`ObstacleDefinitionEditor.cs`	Кастомный инспектор: авторасчёт заблокированных гексов по пикселям спрайта.

`Assets/Scripts/` — отдельные скрипты

Файл	Назначение
`PongGame.cs`	Самостоятельная мини-игра Pong (ручная физика мяча и ракеток).
`Example.cs`	Пустой шаблон `MonoBehaviour` (заготовка для урока).

`Assets/Shaders/` — шейдеры

Файл	Назначение
`HeroesFlat.shader`	Плоское/тунарное освещение моделей (`Heroes/Flat`).
`RetroPost.shader`	Постобработка цвета: постеризация + палитра (`Hidden/Heroes/RetroPost`).
`ColorKeySprite.shader`	Прозрачность по ключевому цвету для спрайтов (`Custom/ColorKeySprite`).

`Assets/Data/` — ассеты-данные (ScriptableObject-инстансы)

Ассет	Назначение
`ObstacleDatabase.asset`	Конкретный экземпляр каталога препятствий.
`Obstacles/Obstacle*.asset`	Конкретные препятствия (8 штук).
`Units/Zombie.asset`	Существо «Зомби» (единственный юнит на данный момент).

0.4. Технологический стек и глоссарий терминов

Прежде чем нырять в код, зафиксируем словарь. Полная версия — в Приложении А, здесь — минимум, без которого не понять первые главы.

- **Unity** — игровой движок. Наш проект использует **Built-in Render Pipeline (BiRP)** — «классический» конвейер рендеринга (не URP и не HDRP). Это важно: почти вся визуализация (пост-эффекты через `OnRenderImage`, шейдеры на CG/`ForwardBase`) написана под BiRP.
- `GameObject` — базовый объект сцены. Сам по себе пустой контейнер; всё поведение ему дают навешанные **компоненты**.
- `MonoBehaviour` — базовый класс для компонента-скрипта. Только его экземпляры Unity вызывает по событиям (`Awake`, `Start`, `Update`, `OnRenderImage` …). Если класс **не** наследует `MonoBehaviour`, Unity про него «не знает» — им управляет другой код.
- `ScriptableObject` **(SO)** — объект-данные, который сохраняется как ассет-файл в проекте (`.asset`), а не живёт на сцене. Идеален для «карточек» существ и препятствий: дизайнер правит цифры в инспекторе, программист читает их из кода.
- **Гекс (hex)** — шестиугольная клетка. Поле боя — сетка из гексов 15×11. Мы используем раскладку **«odd-r offset»**: строки нумеруются сверху вниз, **нечётные строки сдвинуты вправо** на полшага. Подробно — в разделах 1.6 и 2.1.
- **Отряд / стек (stack)** — группа одинаковых существ в одной клетке (например, «40 зомби»). В коде — класс `UnitStack`. У стека есть суммарное здоровье; численность (`Count`) вычисляется из него.
- **Конечный автомат (FSM, finite-state machine)** — модель, где объект в каждый момент находится в одном из нескольких **состояний** и переходит между ними по правилам. `UnitStack` — это FSM с состояниями `Idle → Active → Moving/Attacking → … → Dead`.
- **Корутина (coroutine)** — «функция, которую можно поставить на паузу и продолжить в следующем кадре». В Unity это способ растянуть действие во времени (анимация хода) без блокировки основного потока. Технически — `IEnumerator` с `yield return`.
- **Seed (семя)** — число, которым инициализируется генератор случайных чисел. Один и тот же seed → одинаковая последовательность «случайных» значений → **воспроизводимая** генерация поля.

- **Sorting order / render queue** — два механизма, определяющие **порядок отрисовки** (что рисуется поверх чего). Для 2D-спрайтов — `sortingOrder`, для материалов — `renderQueue`. Разберём в разделе 2.8.
- **Пост-эффект (post-processing)** — обработка **уже отрисованного кадра** целиком (пикселизация, изменение цвета). В BiRP делается через `OnRenderImage` + `Graphics.Blit`.

0.5. Сквозной сценарий: что происходит от «Start Battle» до победы

Чтобы отдельные классы не казались набором разрозненных деталей, проследим один полный проход. Держите эту картину в голове при чтении Части 1 — станет понятно, «кто кого зовёт».

1. **Дизайнер нажимает «Start Battle»** в окне `BattleTestWindow` (или другой код вызывает `BattleController.StartBattle(attacker, defender)`).
2. `BattleController` **регенерирует поле** (если включено): зовёт статический `BattlefieldGenerator.Generate(seed, terrain, database, …)`. Тот — **чистая логика** — возвращает `BattlefieldConfig`: список препятствий (id + координаты) и id фона. Ни одного `GameObject` пока не создано.
3. `BattleController` **просит** `BattlefieldView.Deploy(config)` материализовать конфиг: view ставит фоновый спрайт, инстанцирует 15×11 префабов-гексов (скрывая занятые препятствиями), расставляет спрайты препятствий. Теперь поле **видно** на сцене.
4. `BattleController` **строит** `_obstacleMap` — булеву матрицу `[15,11]` занятых клеток, разворачивая каждое препятствие в набор блокируемых гексов. Эта матрица — «мост» между визуальными препятствиями и логикой прохода.
5. `BattleController.DeployArmy` **разворачивает обе армии.** Для каждого непустого слота создаётся **логический** `UnitStack` (ядро) и **визуальный** `UnitStackView` (отображение), связанные подпиской на события. Attacker встаёт в колонку 0, defender — в колонку 14.
6. **Запускается корутина** `BattleRoutine` (только в Play Mode). Она крутит раунды, пока живы обе стороны. В каждом раунде `BuildTurnOrder` сортирует живые стеки по скорости (быстрые ходят первыми).
7. **Ход одного стека — корутина** `TakeTurn`**.** Она: - создаёт свежий `BattleGrid` из карты препятствий и текущих позиций стеков; - считает `ComputeReachable` (BFS до `Speed` шагов) и `ComputeAttackable` (враги в зоне удара); - подсвечивает синим достижимые гексы, красным — атакуемых врагов; - **ждёт ввода игрока** (клик/пробел), опрашивая мышь каждый кадр (`yield return null`); - по клику определяет: это перемещение или атака; - двигает стек по пути (`UnitStack.StartMove` → анимация во `UnitStackView` → `CompleteMove`), при атаке применяет формулу урона и проигрывает выпад.
8. `UnitStackView` **реагирует на смену состояния стека.** Он подписан на событие `StateChanged`. Когда стек входит в `Moving`, view запускает корутину скольжения модели по пути и включает булев параметр аниматора `IsMoving`. Когда в `Attacking` — выпад и триггер `Attack`. Когда `Dead` — триггер `Die` и отключение метки.
9. **Урон и смерть.** `ResolveAttack` считает `max(0, атака − защита) × численность` и зовёт `target.ApplyDamage`. Если суммарное здоровье падает до нуля, стек переходит в `Dead`, view проигрывает смерть.
10. **Победа.** Как только в живых остаётся только одна сторона, `BothSidesAlive()` возвращает `false`, цикл завершается, `DetermineWinnerText` формирует «Attacker wins!» и выводит его через `OnGUI`.

Параллельно всё это время работает **визуальный конвейер**: камера юнитов рендерит модели в `RenderTexture` (`UnitCameraCompositor`), `RetroImageEffect` пикселизует и постеризует кадр, `SteppedAnimator` заставляет анимации «щёлкать» на 10 FPS, шейдер `Heroes/Flat` даёт плоский свет. Эти системы независимы от логики боя и работают «поверх» неё.

Теперь, держа эту карту в голове, разберём каждый класс детально.

Часть 1. Какие скрипты у нас есть и как они работают

1.1. Принцип «чистое ядро + тонкий Unity-слой»

Прежде чем разбирать классы по одному, зафиксируем главный архитектурный приём — он объясняет почти все решения в коде.

Идея. Игровые правила отделены от их отображения. «Что происходит» (правила) — это обычные C#-классы. «Как это выглядит» (пиксели на экране) — это `MonoBehaviour`-ы. Связь между ними — **односторонняя**: отображение знает про ядро и подписывается на него, но ядро про отображение не знает **ничего**.

Как это выглядит на практике. Возьмём отряд. Есть два класса:

- `UnitStack` (ядро) — хранит: где стоит, сколько здоровья, в каком состоянии (`Idle`, `Moving`, `Dead`...). Умеет: переходить между состояниями, получать урон. **Не умеет:** рисовать себя, двигать модель, проигрывать анимацию. Не наследует `MonoBehaviour`.
- `UnitStackView` (отображение) — `MonoBehaviour` на сцене. Держит ссылку на свой `UnitStack`, **подписан** на его события. Когда стек меняет состояние, view это ловит и запускает нужную анимацию. Сам никаких правил не решает.

Механизм связи — события C#. В `UnitStack` объявлены два события:

```
public event Action<UnitStack> StateChanged;    // состояние изменилось
public event Action<UnitStack> HealthChanged;    // здоровье/численность изменились
```

`UnitStackView.Bind` на них подписывается:

```
Stack.StateChanged += OnStateChanged;
Stack.HealthChanged += OnHealthChanged;
```

Теперь стеку достаточно вызвать `StateChanged?.Invoke(this)` — и view **сам** отреагирует. Стек при этом понятия не имеет, кто его слушает: может быть один view, может быть десять, может быть логгер в тесте. Это называется **инверсией зависимостей через события** и подробно разобрано в разделе 3.9.

Зачем так сложно? Пять причин.

1. **Тестируемость.** `UnitStack`, `BattleGrid`, `BattlefieldGenerator` можно проверить обычными юнит-тестами без запуска сцены: создать стек, вызвать `ApplyDamage(100)`, проверить, что `IsAlive == false`. Никакой камеры, никакого рендера.
2. **Детерминизм.** Генератор поля использует `System.Random`, а не `UnityEngine.Random`, и не трогает сцену. Один seed → один и тот же результат, всегда. Это критично и для отладки, и для потенциального сетевого режима (оба игрока получат одинаковое поле).
3. **Читаемость.** Когда правила отделены от рендера, каждый файл делает одно дело. `BattleGrid` — это чистая гексовая математика, там нет ни одного `Instantiate`.
4. **Гибкость отображения.** Логика одна, а показать её можно как угодно: 3D-моделью, спрайтом, капсулой-заглушкой. В коде именно так: если у существа нет префаба, `UnitStackView` рисует капсулу-примитив — логика от этого не меняется.

5. **Работа в Edit Mode.** Многое (генерация и расстановка поля) работает **прямо в редакторе, без запуска игры**, потому что не завязано на игровой цикл. Интерактивный бой — только в Play Mode, и это единственное, что требует запуска.

Запомните этот принцип — дальше вы будете узнавать его в каждом классе.

1.2. Слой данных: `TerrainType`, `BattlefieldConfig`, `PlacedObstacle`, `HexOffset`

Начнём с самого фундамента — с типов, которые **только хранят данные** и не содержат поведения. Это «алфавит», которым разговаривают все остальные классы.

1.2.1. `TerrainType` — перечисление местности

```
namespace Heroes.Battle
{
    public enum TerrainType
    {
        Dirt, Sand, Grass, Snow, Swamp,
        Rough, Subterranean, Lava, Water, Rock,
    }
}
```

Это `enum` — перечисление из 10 именованных констант. Каждый тип местности соответствует своему набору фонов и разрешённых препятствий в оригинальной H3. В коде тип местности влияет на два решения генератора: **какой фон** выбрать и **какие препятствия** разрешены (у препятствия есть список `allowedTerrains`).

Почему `enum`, а не строки или числа? Перечисление даёт три выгоды: (1) автодополнение в IDE — не ошибёшься в написании; (2) компилятор ловит опечатки (`TerrainType.Grasss` не скомпилируется); (3) в инспекторе Unity `enum` показывается удобным выпадающим списком.

Namespace `Heroes.Battle`. Обратите внимание: почти весь код боя лежит в пространстве имён `Heroes.Battle`. Это группирует связанные типы и предотвращает конфликты имён (например, наш `Army` не столкнётся с каким-нибудь `Army` из плагина). Редакторные классы лежат в `Heroes.Battle.Editor`.

1.2.2. `BattlefieldConfig` — «чертёж» поля и его константы

`BattlefieldConfig` играет **двойную роль**: это и хранилище констант размеров поля, и контейнер результата генерации.

Константы — размеры и геометрия поля:

```
public const int Width = 15;     // столбцов (гексов в строке)
public const int Height = 11;    // строк
public const int ReferencePixelsPerUnit = 64;
public const int HexHorizontalStepPx = 44;  // шаг между гексами в строке (VCMI BattleHex::WIDTH)
public const int HexVerticalStepPx = 32;    // шаг между строками
public const int OddRowOffsetPx = 22;       // сдвиг нечётных строк = половина шага
```

Поле — **15×11 = 165 гексов**, ровно как в H3. Числа `44 / 32 / 22` — это шаги укладки гексов в **пикселях** при эталонном разрешении 64 пикселя на юнит. Они взяты из VCMI, чтобы раскладка совпадала с оригинальной. Разберём смысл:

- `HexHorizontalStepPx = 44` — по горизонтали центры соседних гексов в одной строке отстоят на 44 пикселя.
- `HexVerticalStepPx = 32` — строки идут с шагом 32 пикселя. Для «pointy-top» гексов (остриём вверх) вертикальный шаг — это **¾ высоты** гекса, поэтому строки **перекрываются**, и получается плотная сотовая укладка.
- `OddRowOffsetPx = 22` — нечётные строки сдвинуты вправо ровно на **полшага** (44/2 = 22). Это и есть «odd-r offset».

Дальше идут производные константы **в мировых единицах** (world units) — просто делим пиксели на PPU:

```
public const float HexHorizontalStep = (float)HexHorizontalStepPx / ReferencePixelsPerUnit; // 44/
public const float HexVerticalStep   = (float)HexVerticalStepPx   / ReferencePixelsPerUnit; // 32/
```

> **Понятие «мировая единица» (world unit).** В Unity расстояния измеряются в абстрактных «юнитах». Как пиксели спрайта превращаются в юниты, задаёт параметр импорта **Pixels Per Unit (PPU)**. При PPU = 64 спрайт шириной 64 пикселя займёт 1 юнит. Подробно — в разделе 3.4.

Контейнер результата генерации:

```
public int Seed;
public TerrainType Terrain;
public string BackgroundId;                          // id выбранного фона
public readonly List<PlacedObstacle> Obstacles;      // что и где расставлено
```

Ключевое: `BattlefieldConfig` **не содержит ни одной ссылки на Unity-объект** (нет `Sprite`, нет `GameObject`). Он хранит только **идентификаторы** (`BackgroundId`, `ObstacleId`) и координаты. Это делает его чистой сериализуемой структурой данных: такой конфиг можно сохранить в файл, передать по сети, сравнить в тесте. Превращение id в реальные спрайты — задача `BattlefieldView`, то есть слоя отображения.

Комментарий в самом файле формулирует это прямо: *«Pure-data description… Contains no Unity references so it can be produced, serialized or tested without the engine»* — «Чистое описание данных… не содержит ссылок на Unity, поэтому его можно создавать, сериализовать и тестировать без движка».

1.2.3. `PlacedObstacle` — одно размещённое препятствие

```
public struct PlacedObstacle
{
    public string ObstacleId;  // какое препятствие (id в базе)
    public int X;              // столбец якорного гекса
    public int Y;              // строка якорного гекса
}
```

Это `struct` (значимый тип), а не `class`. Каждое препятствие на поле описывается тройкой: **что** (id) и **где** (якорный гекс `X, Y`). Заметьте — здесь только **якорь**, одна клетка. А то, какие ещё клетки

препятствие блокирует и как выглядит, хранится в его `ObstacleDefinition` в базе. То есть `PlacedObstacle` — это «ссылка + позиция», максимально лёгкая запись.

> **`struct` vs `class` — короткое напоминание.** `struct` копируется по значению и обычно используется для маленьких неизменяемых «пучков данных» (координаты, цвет). `class` передаётся по ссылке и используется для объектов с идентичностью и поведением. Здесь `PlacedObstacle` и `HexOffset` — маленькие данные, поэтому `struct`; `BattlefieldConfig` — контейнер с изменяемым списком, поэтому `class`.

1.2.4. `HexOffset` — смещение гекса относительно якоря

```
[Serializable]
public struct HexOffset
{
    public int dx;
    public int dy;
}
```

`HexOffset` — это смещение `(dx, dy)` одного гекса относительно якорного. Из таких смещений складывается «отпечаток» (footprint) препятствия. Например, большой камень с якорем в `(7,5)` и списком `blockedHexes = [(0,0), (1,0), (0,1)]` займёт три клетки: `(7,5)`, `(8,5)`, `(7,6)`.

Атрибут `[Serializable]` говорит Unity: «этот тип можно сохранять и показывать в инспекторе». Без него список `HexOffset` не отобразился бы в редакторе `ObstacleDefinition`. Подробнее о сериализации — в разделе 3.3.

Итог по слою данных. Мы разобрали четыре «немых» типа. Они ничего не делают — только хранят. Но именно на них держится всё остальное: генератор наполняет `BattlefieldConfig` списком `PlacedObstacle`, view читает `HexOffset` из препятствий, а `TerrainType` определяет, что вообще можно поставить. Чистые данные без поведения — это скелет, на который наращивается логика.

1.3. ScriptableObject-ассеты: `UnitDefinition`, `ObstacleDefinition`, `ObstacleDatabase`

Следующий слой — **данные, которые редактирует дизайнер**. Это `ScriptableObject`-ы: объекты-ассеты, которые лежат в проекте отдельными файлами `.asset` и правятся в инспекторе. Разберём подробно, что это даёт.

1.3.1. Что такое `ScriptableObject` и зачем он здесь

Обычный `MonoBehaviour` живёт **на сцене**: чтобы задать характеристики зомби, пришлось бы класть на сцену объект-«зомби» и хранить цифры в нём. Это неудобно: данные размазаны по сценам, дублируются, теряются. `ScriptableObject` решает проблему — это **ассет-файл с данными**, не привязанный к сцене. Один файл `Zombie.asset` описывает зомби для всей игры; на него ссылаются откуда угодно.

Аналогия: `ScriptableObject` — это «карточка в картотеке». Дизайнер заводит карточку «Зомби» (атака 10, защита 8, здоровье 40…), а в бою мы просто ссылаемся на эту карточку и говорим «поставь 40 таких». Правки в карточке автоматически применяются везде.

Все три класса ниже помечены атрибутом `[CreateAssetMenu( ... )]` — он добавляет пункт в меню `Assets → Create`, чтобы дизайнер мог создавать новые карточки правым кликом.

1.3.2. `UnitDefinition` — прототип существа

```
[CreateAssetMenu(menuName = "Heroes/Battle/Unit", fileName = "Unit")]
public sealed class UnitDefinition : ScriptableObject
{
    public string id;              // стабильный идентификатор (для сейвов/конфигов)
    public string displayName;     // отображаемое имя
    public Sprite sprite;          // 2D-спрайт (запасной вариант)
    public GameObject prefab;      // 3D-модель с Animator (основной вариант)

    [Header("Stats")]
    [Min(0)] public int attack = 1;
    [Min(0)] public int defense = 1;
    [Min(1)] public int health = 10;    // здоровье ОДНОГО существа
    [Min(1)] public int speed = 5;      // дальность хода в гексах = инициатива
    [Range(1, 2)] public int hexWidth = 1;  // 1 = обычный, 2 = широкий
}
```

Это «карточка существа». Здесь собраны **все** боевые характеристики и оба варианта визуала. Разберём тонкости:

- `sprite` **и** `prefab` **— два способа показать существо.** Если задан `prefab` (3D-модель с аниматором), в бою инстанцируется он. Если нет — используется `sprite`. Если нет и его — `UnitStackView` рисует капсулу-заглушку. Это пример «деградации по возможностям»: система работает даже с неполными данными.
- **Атрибуты валидации** `[Min]` **и** `[Range]`**.** `[Min(0)]` не даёт дизайнеру ввести отрицательную атаку; `[Range(1,2)]` превращает поле `hexWidth` в слайдер от 1 до 2. Это «защита от дурака» прямо в инспекторе — данные не могут стать некорректными.
- `[Header("Stats")]` — рисует заголовок-разделитель в инспекторе. Чисто косметика для удобства дизайнера, но показывает заботу об инструменте.
- `speed` **— двойного назначения.** Скорость это и **дальность хода** (сколько гексов пройдёт за ход), и **инициатива** (кто ходит раньше). Одно число — два игровых смысла, ровно как в H3.
- `health` **— на одно существо.** Здоровье в карточке — это HP **одного** зомби. Суммарное здоровье стека считается как `health × count` уже в `UnitStack`. Это важный нюанс модели урона (см. 1.5).

Два метода-хелпера решают проблему «пустого имени»:

```
public string ResolveId() => string.IsNullOrEmpty(id) ? name : id;
public string ResolveDisplayName() => string.IsNullOrEmpty(displayName) ? name : displayName;
```

Если дизайнер не заполнил `id` / `displayName`, берётся имя ассета (`name`). Так карточка всегда имеет валидный идентификатор, даже если поля пустые. Приём «пустое поле → разумное значение по умолчанию» повторяется в проекте многократно.

`sealed class`**.** Почти все классы проекта помечены `sealed` — «от этого класса нельзя наследоваться». Это и лёгкая оптимизация (компилятор знает, что метод не переопределят), и сигнал намерения: «этот класс не задуман как база для иерархии».

1.3.3. `ObstacleDefinition` — прототип препятствия

```
[CreateAssetMenu(menuName = "Heroes/Battle/Obstacle", fileName = "Obstacle")]
public sealed class ObstacleDefinition : ScriptableObject
{
    public string id;
    public Sprite sprite;
    public ObstacleType type = ObstacleType.Usual;      // Usual или Absolute
    public Vector2Int absoluteAnchor = new Vector2Int(7, 5);
    public List<TerrainType> allowedTerrains;           // на каких землях разрешено
    public List<HexOffset> blockedHexes;                // какие клетки блокирует
    public Vector2 spriteOffset;                        // сдвиг спрайта для выравнивания
    [Min(0f)] public float weight = 1f;                 // вес для случайного выбора
}
```

Карточка препятствия. Ключевые поля:

- `blockedHexes` — список `HexOffset`, задающий «отпечаток» препятствия относительно якоря. По умолчанию — одна клетка `(0,0)`. Именно этот список читают и генератор (чтобы проверить, влезает ли препятствие), и view (чтобы скрыть занятые гексы), и `BattleController` (чтобы построить карту непроходимости).
- `type` (`Usual` / `Absolute`) — терминология VCMI/H3. **Usual** — обычное препятствие, ставится в случайное место, их много. **Absolute** — большое «сюжетное» украшение (затонувший корабль, гигантская скала), которое сидит в **фиксированной точке** (`absoluteAnchor`) и появляется максимум **одно** за бой. Разберём в 1.7.
- `allowedTerrains` — на каких типах местности препятствие может появиться. Куст растёт на траве, но не на лаве. Генератор фильтрует по этому списку.
- `spriteOffset` — визуальный сдвиг спрайта в мировых координатах. Нужен, когда арт «нависает» над клеткой (крона дерева выше его основания): логически дерево в клетке `(x,y)`, а спрайт приподнят.
- `weight` — относительный вес при случайном выборе. Препятствие с весом 3 выпадет втрое чаще, чем с весом 1. Механику взвешенного выбора разберём в 1.7.

Рядом объявлены вспомогательный `enum ObstacleType { Usual, Absolute }` и уже знакомый нам `struct HexOffset`.

1.3.4. `ObstacleDatabase` — каталог и параметры генерации

```
[CreateAssetMenu(menuName = "Heroes/Battle/Obstacle Database", fileName = "ObstacleDatabase")]
public sealed class ObstacleDatabase : ScriptableObject
{
    public List<ObstacleDefinition> obstacles;    // Все препятствия
    public List<TerrainBackground> backgrounds;   // фоны по типам местности

    [Header("Generation")]
    [Min(0)] public int minObstacles = 5;
    [Min(0)] public int maxObstacles = 15;
    [Min(1)] public int placementAttemptsPerObstacle = 20;
    [Range(0f, 1f)] public float absoluteObstacleChance = 0.5f;
    // ...
}
```

Это **центральный каталог**: генератор берёт из него всё, что нужно для постройки поля. Он объединяет две вещи: (1) списки контента (все препятствия и все фоны) и (2) настройки генерации (сколько препятствий ставить, сколько попыток делать, шанс «абсолютного» препятствия).

У базы три метода-запроса:

```
// Все препятствия для данной местности (и опционально данного типа).
public IEnumerable<ObstacleDefinition> GetObstaclesForTerrain(TerrainType terrain, ObstacleType? t

// Найти набор фонов для местности.
public TerrainBackground FindBackgrounds(TerrainType terrain)

// Найти препятствие по id (обратное преобразование id → определение).
public ObstacleDefinition FindById(string obstacleId)
```

Обратите внимание на `GetObstaclesForTerrain` — он использует `yield return`:

```
foreach (var o in obstacles)
{
    if (o == null) continue;
    if (typeFilter.HasValue && o.type ≠ typeFilter.Value) continue;
    if (o.allowedTerrains == null || o.allowedTerrains.Count == 0) continue;
    if (o.allowedTerrains.Contains(terrain)) yield return o;
}
```

`yield return` превращает метод в **ленивый итератор**: он не строит новый список в памяти, а «выдаёт» подходящие элементы по одному по мере обхода. Это идиоматичный C# и экономит аллокации. Обратите внимание на «защитное» программирование: проверки на `null` и на пустой список терраинов — данные от дизайнера могут быть неполными, и код к этому готов.

Вложенный класс `TerrainBackground` связывает тип местности с набором фоновых спрайтов:

```
[Serializable]
public sealed class TerrainBackground
{
    public TerrainType terrain;
    public List<Sprite> variants;    // несколько вариантов фона; генератор берёт случайный
    // ResolveVariantId(index) и GetVariantById(id) — прямое и обратное преобразование
}
```

Здесь снова видим **паттерн «id ↔ спрайт»**: генератор (чистая логика) оперирует **строковым id** фона (`ResolveVariantId` возвращает имя спрайта), а view потом по этому id достаёт реальный `Sprite` (`GetVariantById`). Так граница между «логикой без Unity» и «отображением с Unity» проходит ровно по идентификаторам.

Итог по SO-слою. `ScriptableObject`-ы — это «редактируемые данные». Они позволяют дизайнеру балансировать игру (менять цифры существ, вес препятствий, шанс абсолютных объектов) **не притрагиваясь к коду**. А код читает их как обычные объекты. Это классический паттерн Unity — «data-driven design», проектирование, управляемое данными.

1.4. Модель армии: `Army` и `ArmySlot`

Перед боем нужно описать, **кто** сражается. Этим занимаются два маленьких класса в `Army.cs`.

1.4.1. `BattleSide` — какая из сторон

```
public enum BattleSide { Attacker, Defender }
```

Всего две стороны: атакующий (слева, колонка 0) и защищающийся (справа, колонка 14). Это перечисление пронизывает весь код: по нему выбирают цвет-заглушку, стартовую колонку, направление «вперёд» и порядок при равной скорости.

1.4.2. `ArmySlot` — одна ячейка армии

```
[Serializable]
public struct ArmySlot
{
    public UnitDefinition Unit;   // ссылка на карточку существа
    [Min(0)] public int Count;    // сколько существ

    public bool IsEmpty => Unit == null || Count <= 0;
}
```

Слот — это пара «**какое** существо + **сколько** его». Слот считается **пустым**, если нет карточки существа **или** численность не положительна. Свойство `IsEmpty` инкапсулирует это правило в одном месте — везде дальше проверяют просто `slot.IsEmpty`, не повторяя условие.

1.4.3. `Army` — семь слотов

```
[Serializable]
public sealed class Army
{
    public const int SlotCount = 7;

    [SerializeField] private ArmySlot[] slots = new ArmySlot[SlotCount];

    public ArmySlot GetSlot(int index) { /* с проверкой границ */ }
    public void SetSlot(int index, UnitDefinition unit, int count) { /* ... */ }

    public bool HasUnits { get; }        // есть ли хоть один непустой слот
    public bool HasWideUnits { get; }    // есть ли широкие (2-гексовые) существа
}
```

Армия — это ровно **7 слотов**, как в H3. Несколько слотов могут ссылаться на одно и то же существо (два отряда зомби). Разберём детали:

- `[SerializeField] private ArmySlot[] slots`**.** Массив приватный (инкапсуляция — снаружи нельзя случайно сломать), но помечен `[SerializeField]`, чтобы Unity его **сериализовала** и показала в инспекторе. Это важный трюк: `private` для кода, но видимый для редактора. Подробнее — в 3.3.
- `HasUnits` — быстрый ответ на «есть ли вообще кого разворачивать». `BattleController` проверяет его, чтобы не начинать пустой бой.

- `HasWideUnits` — «есть ли двухклеточные существа». Это влияет на генерацию: если в армии есть широкий юнит, генератор резервирует под неё **две** колонки, а не одну, чтобы внутренний гекс широкого существа не налез на препятствие (см. 1.7 и 1.9).
- `EnsureSize()` — защитный метод. Если массив `slots` вдруг оказался неправильного размера (например, после ручной правки сериализованных данных или изменения `SlotCount`), он восстанавливает длину через `Array.Resize`. Вызывается перед каждым доступом. Ещё один пример «оборонительного» кода, который не доверяет входным данным.
- **Проверки границ.** `GetSlot` / `SetSlot` бросают `ArgumentOutOfRangeException` при выходе за `0..6`. Это «контракт» класса: некорректный индекс — ошибка программиста, и лучше упасть сразу с понятным сообщением, чем тихо испортить данные.

Итог по армии. `Army` — это чистая структура данных для описания состава войска до боя. Она сериализуема (её можно собрать в инспекторе или в коде), проста и защищена от неверных данных. Когда `BattleController.StartBattle` получает две `Army`, он разворачивает их непустые слоты в живые `UnitStack`-и — этим займёмся в 1.9, а сам `UnitStack` разберём прямо сейчас.

1.5. `UnitStack` — конечный автомат одного отряда

`UnitStack` — сердце боевой логики. Это **чистый C#-класс** (не `MonoBehaviour`), описывающий один развёрнутый отряд как **конечный автомат** (finite-state machine, FSM). Разберём его особенно тщательно — здесь сходятся сразу несколько важных идей.

1.5.1. Что такое конечный автомат и зачем он тут

Конечный автомат — это модель, где объект в каждый момент находится ровно в **одном** из нескольких состояний, и переходы между состояниями подчиняются явным правилам.

Для отряда состояния такие:

```
public enum UnitStackState
{
    Idle,       // ждёт своего хода
    Active,     // его ход: игрок выбирает действие
    Moving,     // скользит по пути к клетке
    Attacking,  // проигрывает атаку по цели
    Dead,       // уничтожен, больше не участвует
}
```

А разрешённые переходы (из комментария в коде):

```
Idle --Activate--> Active
Active --StartMove--> Moving --CompleteMove--> Active
Active --StartAttack--> Attacking --CompleteAttack--> Active
Active --Deactivate--> Idle
(любое) --Kill--> Dead
```

Зачем это нужно? Без FSM пришлось бы разбрасывать по коду флаги `bool isMoving`, `bool isDead`, `bool isMyTurn` и следить, чтобы они не противоречили друг другу (что, если `isMoving && isDead`?). FSM заменяет ворох флагов **одним** полем `State`, и любое состояние взаимоисключающее по определению. Код становится проще и надёжнее.

Почему FSM удобен именно для связки логика ↔ отображение? Потому что view может просто **реагировать на смену состояния**, а не получать императивные команды «а теперь подвигайся». Стек говорит «я теперь `Moving`», view слышит это и сам решает, как показать движение. Это прямое следствие принципа из 1.1.

1.5.2. Конструктор и производные характеристики

```
public UnitStack(UnitDefinition definition, int count, BattleSide side,
                 int slotIndex, int x, int y)
{
    Definition = definition;
    Side = side;
    SlotIndex = slotIndex;
    X = x;  Y = y;
    MaxUnitHealth = definition ≠ null ? Mathf.Max(1, definition.health) : 1;
    TotalHealth = Mathf.Max(0, count) * MaxUnitHealth;
    State = UnitStackState.Idle;
}
```

Стек **не хранит** боевые характеристики напрямую — он берёт их из своей карточки `Definition` через свойства-«обёртки»:

```
public int Speed   ⇒ Definition ≠ null ? Mathf.Max(1, Definition.speed)   : 1;
public int Attack  ⇒ Definition ≠ null ? Mathf.Max(0, Definition.attack)  : 0;
public int Defense ⇒ Definition ≠ null ? Mathf.Max(0, Definition.defense) : 0;
public int HexWidth ⇒ Definition ≠ null && Definition.hexWidth > 1 ? 2 : 1;
```

Это разумно: характеристики живут в одном месте (карточке), а стек — их «живой экземпляр». Обратите внимание на **защиту от** `null` в каждом свойстве: если карточки нет, возвращаются безопасные значения по умолчанию (скорость 1, атака 0). Код нигде не упадёт из-за пустой ссылки.

1.5.3. Модель здоровья — тонкий и важный момент

Здесь спрятана изящная механика. Стек хранит **суммарное** здоровье, а численность **вычисляет** из него:

```
public int MaxUnitHealth { get; }          // HP одного существа
public int TotalHealth { get; private set; } // суммарное HP всего стека

public int Count ⇒ TotalHealth ≤ 0 ? 0 : Mathf.CeilToInt((float)TotalHealth / MaxUnitHealth);
```

Разберём на примере: 40 зомби по 10 HP каждый → `MaxUnitHealth = 10`, `TotalHealth = 400`, `Count = 40`.

Наносим 25 урона → `TotalHealth = 375`. Теперь `Count = ceil(375 / 10) = ceil(37.5) = 38`.

Почему `CeilToInt` **(округление вверх)?** Потому что «недобитый» верхний зомби всё ещё считается живым. 375 HP — это 37 целых зомби плюс один раненый на 5 HP. Раненый жив → округляем вверх до 38. Это в точности механика H3: урон «съедает» существ по одному, а последнее, недобитое, продолжает драться с остатком HP. Одно свойство `Count` элегантно кодирует всю эту логику.

`ApplyDamage` замыкает модель:

```
public void ApplyDamage(int amount)
{
    if (amount ≤ 0 || State == UnitStackState.Dead) return;
    TotalHealth = Mathf.Max(0, TotalHealth - amount);
    HealthChanged?.Invoke(this);              // сообщаем подписчикам (view обновит цифру)
    if (TotalHealth ≤ 0) SetState(UnitStackState.Dead);
}
```

Урон вычитается из суммарного HP (не ниже нуля), поднимается событие `HealthChanged` (чтобы view обновил подпись с численностью), и если HP исчерпано — стек переходит в `Dead`.

`IsAlive` объединяет оба условия смерти:

```
public bool IsAlive ⇒ State ≠ UnitStackState.Dead && TotalHealth > 0;
```

1.5.4. Переходы состояний — методы-«команды»

Переходы оформлены отдельными методами. Все они проходят через приватный `SetState`:

```
private void SetState(UnitStackState next)
{
    if (State == next) return;          // нет смысла «переходить» в то же состояние
    State = next;
    StateChanged?.Invoke(this);         // единая точка оповещения
}
```

Это **единственное** место, где меняется `State`, и **единственное**, где поднимается `StateChanged`. Такая централизация — важный приём: никто не может изменить состояние в обход события, значит view никогда не «проспит» смену. Проверка `if (State == next) return` гасит лишние оповещения.

Публичные переходы (упрощённо):

```
public void Activate()   ⇒ SetState(Active);      // если не мёртв
public void Deactivate() ⇒ SetState(Idle);        // если не мёртв
public void StartMove(path)  { _movePath = path; SetState(Moving); }
public void StartAttack(target) { AttackTarget = target; SetState(Attacking); }
```

Отдельного внимания заслуживают «завершающие» методы `CompleteMove` и `CompleteAttack` — их вызывает **view**, когда анимация доиграна:

```
public void CompleteMove()
{
    if (_movePath.Count > 0)
    {
        var end = _movePath[_movePath.Count - 1];
        X = end.x;  Y = end.y;      // ТОЛЬКО ЗДЕСЬ фиксируется новая позиция
    }
    _movePath.Clear();
    if (State == UnitStackState.Dead) return;
    SetState(UnitStackState.Active);
}
```

Ключевой момент: логическая позиция `(X, Y)` обновляется **не** в начале движения, а в конце — когда view сообщил, что модель доехала. Пока идёт анимация, логика считает, что стек всё ещё на старом

месте. Это синхронизирует «правду данных» с «правдой экрана»: они меняются одновременно. Так избегается класс багов, где логика уже «телепортировала» отряд, а модель ещё ползёт.

> **Кто кого двигает — важное уточнение потока управления.** `BattleController` вызывает `StartMove` (логика входит в `Moving`) → `UnitStack` поднимает `StateChanged` → `UnitStackView` слышит и запускает корутину скольжения модели → когда модель доехала, view вызывает `stack.CompleteMove()` → логика фиксирует новую `(X,Y)` и возвращается в `Active`. Контроллер всё это время ждёт (`WaitUntil(() ⇒ stack.State ≠ Moving)`). Управление ходит по кругу «контроллер → стек → view → стек → контроллер», и на каждом участке отвечает свой слой. Это тот самый танец из 0.5, теперь в деталях.

1.5.5. Геометрия «широких» существ

Стек умеет занимать одну или две клетки (`HexWidth`). Метод, вычисляющий занятые столбцы:

```
public IEnumerable<int> OccupiedColumns() ⇒ ColumnsFor(Side, HexWidth, X);

public static IEnumerable<int> ColumnsFor(BattleSide side, int width, int anchorX)
{
    int w = Mathf.Max(1, width);
    for (int i = 0; i < w; i++)
        yield return side == BattleSide.Attacker ? anchorX + i : anchorX - i;
}
```

Тонкость в **направлении расширения**. Якорь `X` стоит у края поля (attacker — слева, defender — справа). Второй гекс широкого существа тянется **к центру** поля: у атакующего — вправо (`anchorX + i`), у защитника — влево (`anchorX - i`). Так широкое существо всегда «смотрит» второй клеткой внутрь поля, а якорь остаётся у своего края. Метод сделан `static`, потому что им пользуется и `BattleGrid` (проверить, влезет ли стек в клетку, ещё до того как он туда поставлен) — ему нужно посчитать столбцы для **гипотетического** якоря.

Итог по `UnitStack`. Мы разобрали FSM с пятью состояниями, изящную модель здоровья с округлением вверх, централизованную смену состояния через события и аккуратную фиксацию позиции в конце движения. Это образцовый пример «чистого ядра»: класс полон игровых правил и не содержит ни одного обращения к сцене. Всю визуальную работу за него делает `UnitStackView` (раздел 1.11), просто слушая его события.

1.6. `BattleGrid` — гексовая геометрия и поиск пути

`BattleGrid` — второй столп боевой логики. Это **чистый C#-класс**, отвечающий на три вопроса: (1) какие клетки заняты; (2) куда стек может дойти за свой ход; (3) кого он может атаковать. Никакой работы со сценой — только математика и алгоритмы на графах.

Важная деталь жизненного цикла: `BattleGrid` **создаётся заново на каждый ход**. Контроллер делает `new BattleGrid(_obstacleMap, _stacks)` в начале `TakeTurn`. Это снимок обстановки «здесь и сейчас»: где препятствия и кто где стоит в данный момент. Пересоздавать дёшево, а свежий снимок избавляет от необходимости синхронизировать сетку при каждом перемещении.

1.6.1. Гексовые координаты: «odd-r offset»

Главная сложность гексовой сетки — у гекса **шесть** соседей, и их смещения зависят от чётности строки. Проект использует раскладку **odd-r offset**: гексы «остриём вверх» (pointy-top), строки идут сверху вниз, **нечётные строки сдвинуты вправо** на полшага.

Из-за сдвига «диагональные» соседи у чётных и нечётных строк — разные. Это закодировано в таблице направлений:

```
private static readonly Vector2Int[][] Directions =
{
    // чётные строки (row & 1 == 0)
    new[] {
        new Vector2Int( 1, 0), new Vector2Int( 0,-1), new Vector2Int(-1,-1),
        new Vector2Int(-1, 0), new Vector2Int(-1, 1), new Vector2Int( 0, 1),
    },
    // нечётные строки (row & 1 == 1)
    new[] {
        new Vector2Int( 1, 0), new Vector2Int( 1,-1), new Vector2Int( 0,-1),
        new Vector2Int(-1, 0), new Vector2Int( 0, 1), new Vector2Int( 1, 1),
    },
};
```

Читается так: `Directions[y & 1]` выбирает нужную из двух таблиц по чётности строки (`y & 1` — это быстрый способ узнать «чётное или нечётное», побитовое И с единицей даёт младший бит). Каждая таблица — 6 смещений `(dCol, dRow)` к шести соседям. Метод-обёртка:

```
public static IEnumerable<Vector2Int> Neighbors(int x, int y)
{
    var dirs = Directions[y & 1];
    for (int i = 0; i < dirs.Length; i++)
        yield return new Vector2Int(x + dirs[i].x, y + dirs[i].y);
}
```

`Neighbors(x, y)` выдаёт координаты всех шести соседей клетки — фундамент для всех алгоритмов ниже. Таблица направлений здесь **строго согласована** с формулой `HexToWorld` в `BattlefieldView` (раздел 1.10): логика и геометрия отображения используют одну и ту же раскладку, иначе подсветка «уехала» бы относительно реальных гексов.

> **Почему offset-координаты, а не axial/cube?** В теории гексов есть более «математичные» системы (axial, cube), где соседи одинаковы для всех клеток и проще считать расстояния. Но offset-координаты **естественнее ложатся на прямоугольный массив** `[Width, Height]` и совпадают с тем, как VCMI хранит поле. Проект выбрал совместимость с эталоном и простое индексирование ценой двух таблиц направлений — разумный компромисс.

1.6.2. Модель занятости клеток

Конструктор строит две структуры: список стеков и матрицу «кто где стоит»:

```
public BattleGrid(bool[,] obstacle, IEnumerable<UnitStack> stacks)
{
    _obstacle = obstacle;                              // матрица препятствий [W,H]
    _stackAt = new UnitStack[Width, Height];           // матрица «в клетке стоит вот этот стек»
    foreach (var stack in stacks)
    {
        _stacks.Add(stack);
        if (!stack.IsAlive) continue;
        foreach (int col in stack.OccupiedColumns())   // широкий стек занимает 2 клетки
            if (InBounds(col, stack.Y)) _stackAt[col, stack.Y] = stack;
    }
}
```

Обратите внимание: широкий стек прописывается в матрицу `_stackAt` **во все** свои столбцы (`OccupiedColumns`), а мёртвые стеки пропускаются. Простые запросы:

```
public bool IsObstacle(int x, int y) => !InBounds(x, y) || (_obstacle ≠ null && _obstacle[x, y]);
public UnitStack StackAt(int x, int y) => InBounds(x, y) ? _stackAt[x, y] : null;
```

Заметьте: **всё, что за границей поля, считается препятствием** (`!InBounds → true`). Это элегантно избавляет алгоритмы от отдельных проверок краёв — «стена по периметру» получается автоматически.

1.6.3. `CanStand` — влезает ли стек в клетку

Прежде чем искать пути, нужен предикат «может ли стек встать якорем в `(anchorX, y)`»:

```
public bool CanStand(UnitStack stack, int anchorX, int y)
{
    foreach (int col in UnitStack.ColumnsFor(stack.Side, stack.HexWidth, anchorX))
    {
        if (!InBounds(col, y)) return false;       // за краем — нельзя
        if (IsObstacle(col, y)) return false;      // препятствие — нельзя
        var occupant = _stackAt[col, y];
        if (occupant ≠ null && occupant ≠ stack) return false;  // занято другим — нельзя
    }
    return true;
}
```

Проверяются **все** клетки, которые займёт стек (для широкого — обе). Клетка подходит, если она в границах, свободна от препятствий и либо пуста, либо занята **этим же** стеком (важно: стек не блокирует сам себя, иначе он не смог бы «стоять на месте»). Здесь снова используется статический `ColumnsFor` — для **гипотетического** якоря, куда стек ещё только собирается встать.

1.6.4. `ComputeReachable` — поиск в ширину (BFS)

Теперь главный алгоритм — какие клетки достижимы за один ход. Дальность ограничена скоростью стека. Используется **обход в ширину** (breadth-first search, BFS):

```
public Dictionary<Vector2Int, int> ComputeReachable(UnitStack stack, out Dictionary<Vector2Int, Ve
{
    var distance = new Dictionary<Vector2Int, int>();
    cameFrom = new Dictionary<Vector2Int, Vector2Int>();

    var start = new Vector2Int(stack.X, stack.Y);
    distance[start] = 0;

    var queue = new Queue<Vector2Int>();
    queue.Enqueue(start);

    while (queue.Count > 0)
    {
        var current = queue.Dequeue();
        int d = distance[current];
        if (d ≥ stack.Speed) continue;              // дальше скорости не идём

        foreach (var next in Neighbors(current.x, current.y))
        {
            if (distance.ContainsKey(next)) continue;    // уже посещали
            if (!CanStand(stack, next.x, next.y)) continue;  // не влезаем
            distance[next] = d + 1;
            cameFrom[next] = current;                // помним, откуда пришли
            queue.Enqueue(next);
        }
    }
    return distance;
}
```

Как работает BFS «на пальцах». Представьте волну, расходящуюся от стартовой клетки. Сначала обрабатываются все соседи на расстоянии 1, затем все на расстоянии 2, и так далее. Очередь (`Queue`) гарантирует именно такой порядок — «сначала ближние». Как только клетка попала в `distance`, мы знаем **кратчайшее** число шагов до неё.

Разберём ключевые части:

- `distance` — словарь «клетка → сколько шагов до неё». Одновременно служит и множеством посещённых (проверка `ContainsKey`). Стартовая клетка имеет расстояние 0.
- `if (d ≥ stack.Speed) continue` — обрезаем волну по скорости. Стек со скоростью 5 не уйдёт дальше 5 гексов. Это и есть «дальность хода».
- `cameFrom` — словарь «клетка → откуда мы в неё пришли». Он не нужен для самого поиска достижимости, но незаменим для **восстановления пути** (следующий метод). Возвращается через `out`-параметр.
- **Почему BFS, а не Дейкстра/A*?** Потому что в этой модели **все шаги стоят одинаково** (1 гекс = 1 очко движения, нет «дорогой» и «дешёвой» местности). Когда рёбра равновесны, BFS уже находит кратчайшие пути — более сложные алгоритмы не нужны. Это правильный выбор: простейший алгоритм, решающий задачу.

Сложность. Поле маленькое (165 клеток), у каждой 6 соседей — BFS отрабатывает мгновенно. Пересоздавать сетку и гонять BFS каждый ход абсолютно не накладно.

1.6.5. `BuildPath` — восстановление маршрута

BFS дал нам `cameFrom`. Чтобы получить конкретный путь от старта до цели, идём **назад** по цепочке «откуда пришёл» и разворачиваем:

```
public static List<Vector2Int> BuildPath(Dictionary<Vector2Int, Vector2Int> cameFrom,
                                         Vector2Int start, Vector2Int target)
{
    var path = new List<Vector2Int>();
    if (target == start) return path;                // некуда идти
    if (!cameFrom.ContainsKey(target)) return path; // цель недостижима

    var node = target;
    while (node ≠ start)
    {
        path.Add(node);
        node = cameFrom[node];      // шаг назад
    }
    path.Reverse();                 // разворачиваем: теперь от старта к цели
    return path;
}
```

Начинаем с цели, прыгаем по `cameFrom` к старту, накапливая клетки, затем `Reverse()` даёт путь в правильном порядке. Стартовая клетка в путь **не** входит (путь — это список клеток, куда нужно **прийти**). Именно этот список получает `UnitStack.StartMove`, и по нему `UnitStackView` двигает модель гекс за гексом. Метод `static`, потому что не нужен доступ к состоянию сетки — только к переданному `cameFrom`.

1.6.6. `ComputeAttackable` — кого можно ударить

Последний метод отвечает на вопрос «до каких врагов я дотянусь в ближнем бою за этот ход» и, для каждого, **с какой клетки** бить:

```
public Dictionary<UnitStack, Vector2Int> ComputeAttackable(UnitStack stack,
                                                           Dictionary<Vector2Int, int> reachable)
{
    var result = new Dictionary<UnitStack, Vector2Int>();

    foreach (var enemy in _stacks)
    {
        if (enemy == stack || enemy.Side == stack.Side || !enemy.IsAlive) continue;  // только жив

        var enemyCells = new HashSet<Vector2Int>();
        foreach (int col in enemy.OccupiedColumns())
            enemyCells.Add(new Vector2Int(col, enemy.Y));   // все клетки врага (широкий = 2)

        Vector2Int bestStand = default;
        int bestDist = int.MaxValue;

        foreach (var kvp in reachable)     // перебираем все достижимые клетки
        {
            Vector2Int stand = kvp.Key;
            int dist = kvp.Value;
            if (dist ≥ bestDist) continue;                       // ищем ближайшую точку удара
            if (!IsAdjacentToAny(stack, stand, enemyCells)) continue;  // отсюда достаём врага?
            bestStand = stand;
            bestDist = dist;
        }

        if (bestDist ≠ int.MaxValue) result[enemy] = bestStand;  // враг атакуем, вот откуда бить
    }
    return result;
}
```

Логика: для каждого вражеского стека перебираем **все** клетки, куда мы можем дойти (`reachable` уже посчитан BFS), и ищем среди них ту, что примыкает к врагу и при этом **ближайшую** к нашей текущей позиции (минимальный `dist`). Если такая нашлась — враг атакуем, и мы запоминаем «точку удара» (`bestStand`), с которой нужно нанести удар.

Вспомогательный `IsAdjacentToAny` проверяет, граничит ли хоть одна клетка нашего стека (если встать якорем в `stand`) с хоть одной клеткой врага:

```
private static bool IsAdjacentToAny(UnitStack stack, Vector2Int stand, HashSet<Vector2Int> enemyCe
{
    foreach (int col in UnitStack.ColumnsFor(stack.Side, stack.HexWidth, stand.x))
        foreach (var neighbor in Neighbors(col, stand.y))
            if (enemyCells.Contains(neighbor)) return true;
    return false;
}
```

Двойной цикл учитывает ширину **обоих** стеков: для каждой клетки, которую мы займём в `stand`, смотрим её шесть соседей и проверяем, нет ли среди них клетки врага. `HashSet` для клеток врага даёт быструю проверку `Contains` за O(1).

Как это использует контроллер. Результат `ComputeAttackable` — это словарь «враг → откуда его бить». Когда игрок кликает по подсвеченному красным врагу, контроллер берёт из словаря точку удара, ведёт стек туда (обычным перемещением по пути), а затем проигрывает атаку. Игроку не нужно вручную подходить вплотную — система сама находит клетку для удара.

Итог по `BattleGrid`. Это концентрат алгоритмов: таблицы гексовых направлений, BFS с ограничением по скорости, восстановление пути через `cameFrom`, поиск целей в зоне удара с учётом ширины существ. И всё это — чистый C# без единого обращения к Unity-сцене. Класс можно целиком покрыть юнит-тестами. Это лучший в проекте пример того, что «игровая логика — это просто структуры данных и алгоритмы».

1.7. `BattlefieldGenerator` — детерминированная генерация поля

`BattlefieldGenerator` — **статический** класс, превращающий `(seed, terrain, database)` в готовый `BattlefieldConfig`. Он полностью **детерминирован** и не трогает сцену: тот же seed всегда даёт то же поле. Алгоритм повторяет логику VCMI (`lib/battle/ObstaclePlacer.cpp`).

1.7.1. Почему `System.Random`, а не `UnityEngine.Random`

В самом начале генерации:

```
var rng = new Random(seed);    // это System.Random, НЕ UnityEngine.Random
```

Это принципиальный выбор, и вот почему:

- `UnityEngine.Random` — это **глобальный** генератор со **статическим** состоянием на весь движок. Любой чужой код (эффект частиц, чужой скрипт), дёрнувший `Random.value`, сдвинет его состояние — и ваша «детерминированная» генерация сломается. К тому же его нельзя создать как отдельный локальный экземпляр.
- `System.Random` — **локальный** объект. Вы создаёте свой `new Random(seed)`, и он никак не связан с остальным движком. Никто извне на него не влияет. Значит генерация **воспроизводима на 100%**.

Воспроизводимость критична: для отладки (баг всегда повторяется при том же seed), для тестов (проверяем, что seed=42 даёт ожидаемое поле) и для потенциального сетевого режима (оба клиента строят одинаковое поле из одного seed, не пересылая его целиком).

1.7.2. Общий поток генерации

```
public static BattlefieldConfig Generate(int seed, TerrainType terrain, ObstacleDatabase database,
                                         int reservedColumnsLeft, int reservedColumnsRight)
{
    var rng = new Random(seed);
    var config = new BattlefieldConfig {
        Seed = seed, Terrain = terrain,
        BackgroundId = PickBackgroundId(rng, database, terrain),
    };

    bool[,] occupied = new bool[Width, Height];
    ReserveUnitColumns(occupied, reservedColumnsLeft, reservedColumnsRight);  // шаг 1

    TryPlaceAbsoluteObstacle(rng, database, terrain, occupied, config);       // шаг 2
    PlaceUsualObstacles(rng, database, terrain, occupied, config);            // шаг 3
    return config;
}
```

Три шага, точно по VCMI:

1. **Резервируем колонки под войска.** Крайние колонки, где встанут отряды, помечаются как «занято», чтобы препятствия туда не попали.
2. **Пытаемся поставить одно «абсолютное» препятствие** (большое, в фиксированной точке).
3. **Ставим случайное число «обычных» препятствий.**

Матрица `occupied[,]` — рабочее поле генератора: она копит занятость по мере расстановки, чтобы препятствия не налезали друг на друга и на войска.

1.7.3. Шаг 1 — резервирование колонок под войска

```
private static void ReserveUnitColumns(bool[,] occupied, int reservedLeft, int reservedRight)
{
    reservedLeft = Math.Max(1, reservedLeft);
    reservedRight = Math.Max(1, reservedRight);
    for (int y = 0; y < Height; y++)
    {
        for (int i = 0; i < reservedLeft; i++)  occupied[UnitColumnLeft + i, y] = true;    // слева
        for (int i = 0; i < reservedRight; i++) occupied[UnitColumnRight - i, y] = true;   // справ
    }
}
```

Помечаем `reservedLeft` крайних левых и `reservedRight` крайних правых столбцов как занятые — во всех строках. Обычно резервируется по одной колонке с каждой стороны. **Но** если в армии есть широкие существа, `BattleController` передаёт `reservedLeft = 2` (см. 1.9), и тогда резервируются две колонки — чтобы внутренний гекс широкого юнита гарантированно не столкнулся с препятствием. Красивая деталь: генерация «знает» про широких существ ещё до их расстановки.

1.7.4. Шаг 2 — «абсолютное» препятствие

```
private static void TryPlaceAbsoluteObstacle( ... )
{
    var candidates = /* все Absolute-препятствия для этой местности */;
    if (candidates.Count == 0) return;

    if (rng.NextDouble() > database.absoluteObstacleChance) return;  // бросок «повезёт ли»

    var chosen = candidates[rng.Next(candidates.Count)];
    int ax = chosen.absoluteAnchor.x, ay = chosen.absoluteAnchor.y;  // фиксированный якорь

    if (!CanPlace(chosen, ax, ay, occupied)) return;    // если якорь занят — пропускаем
    Commit(chosen, ax, ay, occupied);
    config.Obstacles.Add(new PlacedObstacle { ObstacleId = chosen.ResolveId(), X = ax, Y = ay });
}
```

Абсолютное препятствие — это большой «сюжетный» объект (затонувший корабль, огромная скала). Особенности:

- Появляется с вероятностью `absoluteObstacleChance` (по умолчанию 0.5). Бросок: `rng.NextDouble() > chance` → пропускаем.
- Ставится в **фиксированную** точку `absoluteAnchor`, а не в случайную. Отсюда и название.
- Появляется максимум **одно** за бой (метод вызывается один раз и ставит одно).

- Если его фиксированный якорь пересекается с зарезервированными колонками — оно просто **не ставится** (проверка `CanPlace`). Войска важнее декорации.

1.7.5. Шаг 3 — «обычные» препятствия и взвешенный выбор

```
private static void PlaceUsualObstacles( ... )
{
    int count = RandomRange(rng, database.minObstacles, database.maxObstacles);  // сколько ставит
    var pool = /* все Usual-препятствия для местности с weight > 0 */;
    int attempts = database.placementAttemptsPerObstacle;   // попыток на каждое

    for (int i = 0; i < count; i++)
    {
        var obstacle = PickWeighted(rng, pool);             // выбор с учётом веса
        for (int a = 0; a < attempts; a++)
        {
            int ax = rng.Next(0, Width), ay = rng.Next(0, Height);  // случайная точка
            if (!CanPlace(obstacle, ax, ay, occupied)) continue;    // не влезло — новая попытка
            Commit(obstacle, ax, ay, occupied);
            config.Obstacles.Add(new PlacedObstacle { ObstacleId = obstacle.ResolveId(), X = ax, Y
            break;                                          // поставили — к следующему
        }
    }
}
```

Логика «попыток»: для каждого из `count` препятствий делаем до `attempts` (по умолчанию 20) попыток найти свободное место. Не нашли за 20 попыток — это препятствие просто пропускается (поле не резиновое, к концу оно заполняется, и это нормально). Так H3 и работает: «попробуй несколько раз, не вышло — забей».

Взвешенный случайный выбор (`PickWeighted`) — классический алгоритм «рулетки»:

```
private static ObstacleDefinition PickWeighted(Random rng, List<ObstacleDefinition> pool)
{
    float total = 0f;
    foreach (var o in pool) total += o.weight;              // сумма всех весов
    if (total ≤ 0f) return pool[rng.Next(pool.Count)];      // все веса нулевые — равновероятно

    float roll = (float)rng.NextDouble() * total;           // точка на отрезке [0, total)
    float acc = 0f;
    foreach (var o in pool)
    {
        acc += o.weight;
        if (roll ≤ acc) return o;                           // в чей «сектор» попали
    }
    return pool[pool.Count - 1];
}
```

Представьте отрезок длины `total`, разбитый на «сектора» по весам: препятствие с весом 3 занимает втрое больший сектор, чем с весом 1. Кидаем случайную точку `roll` на отрезок и идём накоплением `acc`, пока не превысим `roll` — в чей сектор попали, то и выбрали. Препятствия с бóльшим весом занимают бóльшую долю отрезка → выпадают чаще. Так дизайнер через поле `weight` управляет частотой каждого препятствия.

1.7.6. `CanPlace` и `Commit` — проверка и фиксация «отпечатка»

```
private static bool CanPlace(ObstacleDefinition obstacle, int ax, int ay, bool[,] occupied)
{
    var cells = obstacle.blockedHexes;
    if (cells == null || cells.Count == 0) return false;
    foreach (var cell in cells)
    {
        int x = ax + cell.dx, y = ay + cell.dy;
        if (x < 0 || x ≥ Width || y < 0 || y ≥ Height) return false;  // вышло за поле
        if (occupied[x, y]) return false;                              // клетка занята
    }
    return true;
}

private static void Commit(ObstacleDefinition obstacle, int ax, int ay, bool[,] occupied)
{
    foreach (var cell in obstacle.blockedHexes)
        occupied[ax + cell.dx, ay + cell.dy] = true;    // помечаем ВСЕ клетки отпечатка занятыми
}
```

Здесь работает «отпечаток» из `blockedHexes`: `CanPlace` проверяет, что **все** клетки отпечатка (с учётом якоря `ax, ay`) в границах поля и свободны; `Commit` помечает их занятыми. Так большое препятствие честно резервирует все свои клетки, и следующее не сможет на него налезть.

Детерминизм ещё раз. Обратите внимание: **порядок** всех бросков `rng` строго фиксирован (фон → абсолютное → обычные, слева направо по циклу). Поскольку `System.Random` с одним seed даёт одну последовательность, а порядок её потребления неизменен, весь результат воспроизводим до последнего препятствия. Это и есть «детерминированная генерация».

Итог по генератору. Чистый статический алгоритм: локальный ГПСЧ, три фазы расстановки, взвешенная рулетка, система попыток, честная работа с многоклеточными отпечатками. Ни одного `GameObject` — на выходе только данные (`BattlefieldConfig`). Превратить эти данные в картинку — работа `BattlefieldView`.

1.8. `BattleProjection` — математика «2.5D» поворота

`BattleProjection` — маленький (один метод!), но концептуально самый хитрый класс проекта. Он решает задачу, которая и создаёт фирменный вид H3: **как повернуть 3D-модель, стоящую на плоском 2D-поле, чтобы казалось, будто она стоит на наклонной земле, снятой наклонной камерой**.

1.8.1. Проблема, которую он решает

Вспомним ключевое дизайн-решение (раздел 2.2): существа — это 3D-модели, но их **позиция** берётся из плоской 2D-сетки (экранная точка гекса). Камера же — обычная **ортографическая**, смотрящая прямо на плоскость XY, без наклона. Если просто поставить модель и не поворачивать, она будет выглядеть как «наклейка», а движение вверх/вниз по полю не будет читаться как «вглубь/наружу».

Хочется, чтобы модель, идущая «вверх по полю» (в дальние ряды), **разворачивалась**, как если бы она уходила от нас вглубь наклонной сцены. Но менять позицию на настоящую 3D мы не хотим — вся раскладка поля 2D. Значит нужно сыграть на **повороте**: подобрать модели такую 3D-ориентацию,

чтобы обычная прямая камера показала ровно ту картинку, которую показала бы **наклонная** камера, глядя на **вертикально стоящую** модель на **наклонной** земле.

1.8.2. Математика проекции

Единственный метод:

```
public static Quaternion FacingRotation(Vector2 boardDir, float pitchDegrees,
                                        bool applyViewTilt, float yawOffsetDegrees)
{
    float sin = Mathf.Sin(pitchDegrees * Mathf.Deg2Rad);
    if (Mathf.Abs(sin) < 1e-3f) sin = sin < 0f ? -1e-3f : 1e-3f;   // защита от деления на ~0

    Vector3 ground = new Vector3(boardDir.x, 0f, boardDir.y / sin); // экран → земля
    if (ground.sqrMagnitude < 1e-6f) ground = Vector3.forward;

    float yaw = Mathf.Atan2(ground.x, ground.z) * Mathf.Rad2Deg + yawOffsetDegrees;
    Quaternion rotation = Quaternion.AngleAxis(yaw, Vector3.up);    // поворот вокруг вертикали

    if (applyViewTilt)
        rotation = Quaternion.AngleAxis(-pitchDegrees, Vector3.right) * rotation;  // наклон «стоя

    return rotation;
}
```

Разберём по шагам, что делает эта формула.

Шаг 0 — что на входе. `boardDir` — направление, куда должно смотреть существо, заданное **в плоскости поля** (2D: +x вправо, +y вверх по экрану). Например, атакующий смотрит вправо (`(1,0)`), а идущий в дальний ряд — вверх (`(0,1)`). `pitchDegrees` — угол наклона **виртуальной** камеры под горизонт (по умолчанию 25.06°).

Шаг 1 — обратная проекция «экран → земля». Наклонённая на угол φ камера проецирует направление по земле `(gx, gz)` на экран как `(gx, gz·sinφ)` — «глубинная» ось сжимается множителем `sinφ`. Мы решаем **обратную** задачу: по экранному направлению `(sx, sy)` найти земное `(gx, gz)`. Инвертируя, получаем `gx = sx`, `gz = sy / sinφ`. Именно это строка:

```
Vector3 ground = new Vector3(boardDir.x, 0f, boardDir.y / sin);
```

Деление `boardDir.y / sin` «растягивает» вертикальную экранную компоненту обратно в глубину. Защита `if (Mathf.Abs(sin) < 1e-3f)` не даёт делить на ноль при почти нулевом угле.

Шаг 2 — рыскание (yaw) к земному направлению. Теперь у нас есть направление по земле. Модель нужно повернуть **вокруг вертикали** (ось Y), чтобы её «перёд» смотрел туда:

```
float yaw = Mathf.Atan2(ground.x, ground.z) * Mathf.Rad2Deg + yawOffsetDegrees;
Quaternion rotation = Quaternion.AngleAxis(yaw, Vector3.up);
```

`Atan2(x, z)` даёт угол направления в горизонтальной плоскости. `yawOffsetDegrees` — поправка, если у префаба «перёд» смотрит не в +Z (например, 180°, если модель повёрнута задом). Это спасает от необходимости переделывать арт: поворот правится числом в настройках.

Шаг 3 — наклон «стояния» (view tilt). Наконец, домножаем на наклон вокруг оси X на `−φ`:

```
if (applyViewTilt)
    rotation = Quaternion.AngleAxis(-pitchDegrees, Vector3.right) * rotation;
```

Смысл: мы наклоняем вертикально стоящую модель назад на угол камеры, так что **прямая** (ненаклонённая) боевая камера воспроизводит изображение, которое **наклонная** камера увидела бы у вертикальной модели. Порядок умножения важен: сначала yaw (повернули лицом), потом tilt (наклонили) — кватернионы не коммутативны, `A * B ≠ B * A`.

> **Кватернионы — коротко.** `Quaternion` в Unity — способ задать поворот без «блокировки осей» (gimbal lock), присущей углам Эйлера. `Quaternion.AngleAxis(angle, axis)` — поворот на `angle` градусов вокруг оси `axis`. Умножение кватернионов = композиция поворотов, и оно **некоммутативно**: `q1 * q2` означает «сначала q2, потом q1». Отсюда и порядок в коде.

1.8.3. Почему это только поворот, а не позиция

Ещё раз подчеркнём главную мысль (её легко упустить): `BattleProjection` вычисляет **только** `Quaternion` **поворота**. Позицию он не трогает. Позиция модели остаётся честно 2D — её ставит `UnitStackView` в экранную точку гекса. Виртуальная наклонная плоскость существует **лишь для расчёта ориентации**. Именно этот трюк даёт «дёшево» ощущение объёма: рендер остаётся 2D-плоским (одна ортокамера, спрайтовая раскладка), а бойцы «живут» так, будто стоят на наклонной сцене. Отсюда и термин «2.5D».

Класс `static` и не хранит состояния — это чистая функция «вход → выход», её удобно тестировать и переиспользовать. Кто и когда её зовёт — разберём в 1.11 (`UnitStackView` вызывает `FacingRotation` каждый раз, когда модель меняет направление).

Итог по `BattleProjection`. Один метод, немного тригонометрии — и получается визуальная подпись игры. Это отличная иллюстрация того, что «красивый эффект» не обязательно означает «сложный код»: здесь всё держится на школьной формуле проекции и аккуратной работе с кватернионами.

1.9. `BattleController` — движок ходов и фасад боя

`BattleController` — самый большой класс проекта и «дирижёр» всего боя. Это `MonoBehaviour` (слой отображения), который: (1) служит **фасадом** — единой точкой входа `StartBattle`; (2) разворачивает армии в логику и визуал; (3) крутит **движок ходов** (цикл раундов) через корутины; (4) обрабатывает **ввод игрока**; (5) рисует служебную графику (подсветку, HUD).

Он связывает воедино всё ядро (`BattlefieldGenerator`, `BattleGrid`, `UnitStack`) и всё отображение (`BattlefieldView`, `UnitStackView`). Разберём по частям.

1.9.1. Роль фасада и точка входа `StartBattle`

Паттерн «Фасад» (Facade) — это простой интерфейс к сложной подсистеме. Снаружи запустить бой — это одна строка:

```
battleController.StartBattle(attackerArmy, defenderArmy);
```

А внутри `StartBattle` прячется вся оркестровка:

```
public void StartBattle(Army attacker, Army defender)
{
    // 1. Валидация входа
    if (battlefield == null) { Debug.LogError( ... ); return; }
    if (attacker == null || defender == null) { Debug.LogError( ... ); return; }
    if (!attacker.HasUnits && !defender.HasUnits) { Debug.LogWarning( ... ); return; }

    // 2. Остановить прошлый бой, если шёл
    StopBattleRoutine();

    // 3. (Опционально) сгенерировать поле
    if (generateBattlefield && !TryGenerateBattlefield(attacker, defender)) return;

    // 4. Развернуть обе армии
    ClearUnits();
    DeployArmy(attacker, BattleSide.Attacker);
    DeployArmy(defender, BattleSide.Defender);

    _round = 0;
    _resultText = null;

    // 5. В Play Mode — запустить интерактивный цикл
    if (Application.isPlaying)
        _battleRoutine = StartCoroutine(BattleRoutine());
}
```

Обратите внимание на две вещи. Во-первых, **обильная валидация** в начале с понятными `Debug.LogError` — метод не доверяет входу и объясняет, что не так. Во-вторых, **проверка** `Application.isPlaying`: разворачивание поля и войск работает и в Edit Mode (это использует редакторное окно для предпросмотра), а вот интерактивный цикл раундов запускается **только в игре**. Один метод обслуживает оба режима.

1.9.2. Разворачивание поля и «карта непроходимости»

```
private bool TryGenerateBattlefield(Army attacker, Army defender)
{
    if (battlefield.database == null) { Debug.LogError( ... ); return false; }
    if (randomizeSeed) seed = new System.Random().Next();   // свежий seed при желании

    // Ширина резервируемых колонок зависит от наличия широких существ
    int reservedLeft  = attacker.HasWideUnits ? 2 : 1;
    int reservedRight = defender.HasWideUnits ? 2 : 1;

    var config = BattlefieldGenerator.Generate(seed, terrain, battlefield.database, reservedLeft,
    battlefield.Deploy(config);                 // отображение: строим поле на сцене
    _obstacleMap = BuildObstacleMap(config);    // логика: строим матрицу непроходимости
    return true;
}
```

Здесь наглядно виден **мост между слоями**. Один `config` используется дважды: - `battlefield.Deploy(config)` — **отображение**: ставит спрайты, гексы, препятствия; - `BuildObstacleMap(config)` — **логика**: разворачивает препятствия в булеву матрицу `_obstacleMap[15,11]`, которую потом ест `BattleGrid`.

`BuildObstacleMap` проходит по всем `PlacedObstacle`, находит их `ObstacleDefinition` по id, и помечает каждую клетку отпечатка (`blockedHexes`) как занятую — с проверкой границ. Заметьте, что почти идентичный код есть и в `BattlefieldView.ComputeOccupancy` (для скрытия гексов). Небольшое дублирование, но оно разводит ответственность: view скрывает **визуальные** гексы, контроллер строит **логическую** карту прохода. Каждый слой владеет своей копией истины.

И вот та самая связь с шириной существ: если у атакующего есть широкие юниты, `reservedLeft = 2` — генератору передаётся приказ зарезервировать две колонки. Круг замкнулся: `Army.HasWideUnits` → параметр генератора → `ReserveUnitColumns`.

1.9.3. Разворачивание армий — создание пары «логика + view»

```
private void DeployArmy(Army army, BattleSide side)
{
    var root = EnsureChild(UnitRootName);
    int anchorX = side == BattleSide.Attacker ? 0 : BattlefieldConfig.Width - 1;  // 0 или 14
    Color placeholderTint = side == BattleSide.Attacker ? attackerColor : defenderColor;

    for (int i = 0; i < Army.SlotCount; i++)
    {
        var slot = army.GetSlot(i);
        if (slot.IsEmpty) continue;              // пустые слоты пропускаем

        // Индекс слота → номер строки (слоты заполняют поле сверху вниз)
        var stack = new UnitStack(slot.Unit, slot.Count, side, i, anchorX, i);
        _stacks.Add(stack);

        var view = UnitStackView.Create(root, stack, battlefield, unitSortingOrder,
                                        placeholderTint, unitMoveSpeed, unitView);
        _views.Add(view);
    }
}
```

На каждый непустой слот создаётся **пара**: логический `UnitStack` и визуальный `UnitStackView`. Они связываются внутри `UnitStackView.Create` (view подписывается на события стека). Attacker выстраивается в колонку 0, defender — в колонку 14. Индекс слота `i` служит одновременно номером строки `Y` — армия заполняет поле сверху вниз, слот за слотом.

Контроллер держит два параллельных списка — `_stacks` (логика) и `_views` (отображение). По сути это и есть материализация принципа из 1.1 на уровне контейнеров.

1.9.4. Движок ходов — корутина `BattleRoutine`

```
private IEnumerator BattleRoutine()
{
    IsBattleRunning = true;
    while (BothSidesAlive())                    // пока живы обе стороны
    {
        _round++;
        foreach (var stack in BuildTurnOrder())   // по порядку инициативы
        {
            if (!stack.IsAlive) continue;
            if (!BothSidesAlive()) break;
            yield return TakeTurn(stack);       // ждём завершения хода этого стека
        }
    }
    _activeStack = null;
    IsBattleRunning = false;
    _resultText = DetermineWinnerText();
}
```

Это **сердце пошагового боя**, и оно построено на корутине. Внешний `while` — раунды, внутренний `foreach` — ходы стеков в порядке инициативы. Ключевая строка — `yield return TakeTurn(stack)`: она **приостанавливает** цикл раундов, пока корутина одного хода не завершится. Одна корутина «ждёт» другую — так весь бой выражается линейным, читаемым кодом, хотя растянут на многие секунды и кадры.

`BothSidesAlive()` проверяет, остались ли живые стеки у **обеих** сторон (с ранним выходом, как только нашлись и там, и там). Как только одна сторона выбита, цикл заканчивается.

Порядок инициативы:

```
private List<UnitStack> BuildTurnOrder()
{
    var order = /* все живые стеки */;
    order.Sort((a, b) =>
    {
        int bySpeed = b.Speed.CompareTo(a.Speed);    // быстрые — первыми (по убыванию)
        if (bySpeed != 0) return bySpeed;
        int bySide = a.Side.CompareTo(b.Side);       // при равенстве: attacker раньше
        if (bySide != 0) return bySide;
        return a.SlotIndex.CompareTo(b.SlotIndex);   // затем по номеру слота
    });
    return order;
}
```

Трёхуровневая сортировка: сначала по **скорости** (по убыванию — быстрые ходят раньше), при равной скорости — **атакующий** раньше защитника, при равенстве и тут — по **номеру слота**. Это делает порядок ходов полностью **детерминированным** — никакой случайности, тай-брейки всегда разрешаются одинаково.

1.9.5. Один ход — корутина `TakeTurn`

Это самый насыщенный метод. Разберём его логику по частям.

Подготовка — расчёт вариантов и подсветка:

```
private IEnumerator TakeTurn(UnitStack stack)
{
    var grid = new BattleGrid(_obstacleMap, _stacks);          // свежий снимок поля
    var reachable = grid.ComputeReachable(stack, out var cameFrom);  // BFS
    var attackable = grid.ComputeAttackable(stack, reachable);       // цели
    var start = new Vector2Int(stack.X, stack.Y);

    _activeStack = stack;
    stack.Activate();                          // логика → Active (view покажет метку)
    ShowHighlights(reachable, attackable);     // синие/красные подсветки
    // ...
```

В начале хода контроллер создаёт **новый** `BattleGrid` (снимок текущей обстановки), прогоняет BFS и поиск целей, активирует стек и подсвечивает варианты. Подсветка — это служебные спрайты (см. 1.9.7).

Ожидание ввода игрока:

```
    UnitStack attackTarget = null;
    Vector2Int moveDestination = start;
    bool actionChosen = false, skip = false;

    while (!actionChosen && !skip)
    {
        if (Input.GetKeyDown(KeyCode.Space) || Input.GetMouseButtonDown(1))
        { skip = true; break; }                // пробел / ПКМ — пропустить ход

        if (Input.GetMouseButtonDown(0) && TryGetHexUnderMouse(out Vector2Int hex))
        {
            var clickedStack = grid.StackAt(hex.x, hex.y);
            if (clickedStack ≠ null && attackable.TryGetValue(clickedStack, out var stand))
            {                                  // кликнули по атакуемому врагу
                attackTarget = clickedStack;
                moveDestination = stand;       // точка удара из ComputeAttackable
                actionChosen = true;
            }
            else if (clickedStack == null && reachable.ContainsKey(hex) && hex ≠ start)
            {                                  // кликнули по достижимой пустой клетке
                moveDestination = hex;
                actionChosen = true;
            }
        }
        yield return null;                     // ждём следующего кадра
    }
```

Это **цикл опроса ввода**, растянутый во времени благодаря `yield return null` (=«продолжить в следующем кадре»). Каждый кадр контроллер проверяет мышь и клавиатуру, пока игрок не выберет действие. Логика клика:

- **ПКМ / пробел** → пропустить ход.
- **ЛКМ по врагу**, который есть в `attackable` → атака; точка удара берётся из словаря (`stand`), игроку не нужно подходить вручную.
- **ЛКМ по пустой достижимой клетке** (не текущей) → перемещение.
- Клик мимо (недостижимая клетка, свой стек) — игнорируется, цикл продолжается.

`TryGetHexUnderMouse` переводит позицию мыши в координаты гекса — разберём в 1.9.6.

Исполнение действия:

```
    ClearHighlights();
    if (!skip)
    {
        if (moveDestination ≠ start)          // если нужно двигаться
        {
            var path = BattleGrid.BuildPath(cameFrom, start, moveDestination);
            if (path.Count > 0)
            {
                stack.StartMove(path);         // логика → Moving; view анимирует
                yield return new WaitUntil(() ⇒ stack.State ≠ UnitStackState.Moving);
            }
        }
        if (attackTarget ≠ null && attackTarget.IsAlive)   // если нужно атаковать
        {
            ResolveAttack(stack, attackTarget);
            yield return new WaitUntil(() ⇒ stack.State ≠ UnitStackState.Attacking);
        }
    }
    if (stack.IsAlive) stack.Deactivate();     // ход окончен → Idle
    _activeStack = null;
}
```

Здесь виден **танец слоёв** из 1.5.4 во всей красе. Контроллер меняет **логическое** состояние (`StartMove`), а затем **ждёт**, пока view доиграет анимацию и вернёт стек в `Active` (`WaitUntil(() ⇒ stack.State ≠ Moving)`). Контроллер **не двигает модель сам** — он лишь переключает состояние и ждёт результата. `WaitUntil` — встроенный `YieldInstruction`, приостанавливающий корутину, пока лямбда не вернёт `true`.

Формула урона проста и повторяет H3:

```
private void ResolveAttack(UnitStack attacker, UnitStack target)
{
    int damage = Mathf.Max(0, attacker.Attack - target.Defense) * attacker.Count;
    attacker.StartAttack(target);      // логика → Attacking; view проиграет выпад
    target.ApplyDamage(damage);        // цель получает урон (возможно, умирает)
}
```

Урон = `max(0, атака − защита) × численность атакующего`. Чем больше существ в стеке, тем больше суммарный урон — классика H3. `max(0, …)` не даёт «отрицательного» урона, если защита превышает атаку.

1.9.6. Клик мышью → гекс: `TryGetHexUnderMouse`

```
private bool TryGetHexUnderMouse(out Vector2Int hex)
{
    var cam = battleCamera ≠ null ? battleCamera : Camera.main;
    if (cam == null) return false;

    Vector3 world = cam.ScreenToWorldPoint(Input.mousePosition);       // экран → мир
    Vector3 local = battlefield.transform.InverseTransformPoint(new Vector3(world.x, world.y, 0f));

    float bestDist = float.MaxValue;
    Vector2Int best = default;
    for (int y = 0; y < Height; y++)
        for (int x = 0; x < Width; x++)
        {
            Vector3 center = battlefield.HexToWorld(x, y);
            float d = /* квадрат расстояния от клика до центра гекса */;
            if (d < bestDist) { bestDist = d; best = new Vector2Int(x, y); }
        }

    // Принять клик, только если он достаточно близко к центру гекса
    float threshold = battlefield.EffectiveHorizontalStepPx / battlefield.EffectivePixelsPerUnit;
    if (bestDist > threshold * threshold) return false;

    hex = best;
    return true;
}
```

Задача — узнать, по какому гексу кликнул игрок. Алгоритм:

1. **Экран → мир.** `cam.ScreenToWorldPoint(Input.mousePosition)` переводит пиксели мыши в мировые координаты (работает благодаря ортокамере, см. 3.5).
2. **Мир → локальные координаты поля.** `InverseTransformPoint` учитывает, что само поле может быть сдвинуто/повёрнуто.
3. **Ближайший гекс перебором.** Пробегаем все 165 гексов, считаем `HexToWorld` каждого и ищем ближайший к точке клика (по квадрату расстояния — корень извлекать не нужно, сравнение сохраняется).
4. **Порог принятия.** Если ближайший гекс всё же слишком далеко (клик за краем поля), возвращаем `false`. Порог — примерно шаг между гексами. Сравнение ведётся по квадратам (`bestDist > threshold * threshold`), чтобы не считать корни.

Перебор всех 165 клеток на каждый клик — абсолютно допустимо (клики редки, поле мало). Это тот случай, когда простое решение лучше «умного» обратного преобразования координат гекса, которое легко написать с ошибкой.

1.9.7. Подсветка и служебные спрайты «из ничего»

Контроллер рисует подсветку достижимых/атакуемых клеток. Интересно, что он делает это **без единого ассета-спрайта** — генерирует белый квадрат в коде:

```
private static Sprite GetHighlightSprite()
{
    if (_highlightSprite == null)
    {
        var texture = Texture2D.whiteTexture;                // встроенная белая текстура
        _highlightSprite = Sprite.Create(texture,
            new Rect(0, 0, texture.width, texture.height),
            new Vector2(0.5f, 0.5f),                         // пивот по центру
            texture.width / 0.6f);                           // PPU задаёт мировой размер
        _highlightSprite.hideFlags = HideFlags.DontSave;
    }
    return _highlightSprite;
}
```

`Texture2D.whiteTexture` — встроенная в Unity белая текстура 1×1 (точнее, маленькая служебная). Из неё создаётся `Sprite`, который потом красится в нужный цвет (`reachableColor` — синеватый, `attackableColor` — красноватый) через `SpriteRenderer.color`. Спрайт кэшируется в статическом поле (создаётся один раз на всё приложение) и помечается `HideFlags.DontSave` — «не сохранять в сцену/ассеты, это временный объект».

`ShowHighlights` создаёт по спрайту на каждую достижимую и каждую атакуемую клетку, `CreateHighlight` ставит их в мировые точки гексов, `ClearHighlights` удаляет. Всё это — дети специального контейнера `Highlights` под полем, что упрощает массовую очистку.

1.9.8. HUD через `OnGUI`

```
private void OnGUI()
{
    if (!Application.isPlaying) return;
    // рамка + строка "Round N   Active: Attacker — Zombie (x40)"
    // + подсказка "Click blue hex = move, red enemy = attack, Space/RMB = skip"
    // при завершении — "Attacker wins!"
}
```

`OnGUI` — старый «немедленный» режим интерфейса Unity (immediate mode GUI, IMGUI). Он рисует поверх экрана номер раунда, кто сейчас ходит, подсказку по управлению и итоговый результат. IMGUI не годится для «настоящего» UI игры (он неэффективен и рисуется каждый кадр), но для отладочного/тестового HUD он идеален: пара строк кода — и информация на экране, без возни с Canvas. Это осознанный выбор «инструмента под задачу».

1.9.9. Управление жизненным циклом и очистка

`ClearUnits` останавливает корутину боя, чистит списки `_stacks` / `_views` и удаляет дочерние объекты контейнеров `Units` и `Highlights`. `StopBattleRoutine` аккуратно останавливает корутину, если она идёт. Метод `ClearChildren` умеет удалять объекты и в Play Mode (`Destroy`), и в Edit Mode (`DestroyImmediate`) — это повторяющийся в проекте приём для кода, работающего в обоих режимах.

Итог по `BattleController`. Это оркестратор: фасад (`StartBattle`), развёртывание пар «логика+view», движок раундов на вложенных корутинах, цикл опроса ввода, перевод клика в гекс, генерация подсветки и HUD. Он не содержит боевых **правил** (они в `UnitStack` / `BattleGrid`) — он их **соединяет и проигрывает во времени**. Именно здесь «чистое ядро» встречается с «живой сценой».

1.10. Слой отображения: `BattlefieldView`

`BattlefieldView` — `MonoBehaviour`, который **материализует** `BattlefieldConfig` на сцене: ставит фон, инстанцирует гексы, расставляет спрайты препятствий. Он же — авторитет по **геометрии**: метод `HexToWorld(col, row)` переводит логические координаты гекса в мировую точку, и все (контроллер, view юнитов, редактор) обращаются именно к нему.

Класс помечен `[ExecuteAlways]` — работает и в Edit Mode (это использует окно генератора для предпросмотра), и в игре.

1.10.1. `HexToWorld` — главная формула раскладки

```
public Vector3 HexToWorld(int col, int row)
{
    ResolveLayout();
    float px = _originPx.x + col * _stepXpx + ((row & 1) != 0 ? _offXpx : 0);  // odd-r сдвиг
    float py = _originPx.y - row * _stepYpx;                                    // строки вниз
    float ppu = Mathf.Max(1, pixelsPerUnit);
    return new Vector3(px / ppu, py / ppu, 0f);                                 // пиксели → юниты
}
```

Формула прямо кодирует odd-r раскладку из 1.6.1: - по X: базовый отступ + `col × шаг` + (для нечётных строк) сдвиг на полшага (`_offXpx`); - по Y: минус `row × шаг` (строки идут **вниз** от верха поля); - итог делится на PPU — переводим пиксели в мировые единицы.

Именно согласованность этой формулы с таблицей `Directions` в `BattleGrid` гарантирует, что логика и картинка «видят» один и тот же гекс. Если бы здесь odd-r делался иначе, подсветка разъехалась бы с реальными клетками.

1.10.2. Авторасчёт шага гекса из спрайта: `ResolveLayout`

```
private void ResolveLayout()
{
    if (autoComputeSpacing && TryGetHexSpritePixelSize(out float w, out float h))
    {
        _stepXpx = Mathf.RoundToInt(w);          // шаг = ширина спрайта
        _stepYpx = Mathf.RoundToInt(h * 0.75f);  // ¾ высоты — pointy-top тесселяция
        _offXpx  = Mathf.RoundToInt(w * 0.5f);   // сдвиг = полширины
    }
    else { /* ручные значения из инспектора */ }

    // Центрируем сетку вокруг (0,0) + ручной сдвиг gridOffsetPx
    float halfSpanX = ((Width - 1) * _stepXpx + _offXpx) * 0.5f;
    float halfSpanY = (Height - 1) * _stepYpx * 0.5f;
    _originPx = new Vector2(-halfSpanX + gridOffsetPx.x, halfSpanY + gridOffsetPx.y);
}
```

Здесь важная удобная фича: если включён `autoComputeSpacing`, шаги гексов **вычисляются из самого спрайта** гекса, а не задаются вручную. Для «остроконечных» гексов вертикальный шаг — ровно ¾ высоты (соседние ряды перекрываются на четверть, давая плотную сотовую укладку), а сдвиг нечётных строк — половина ширины. Благодаря этому проект работает с гекс-спрайтом **любого размера** без ручной подгонки чисел — геометрия подстраивается сама.

Далее сетка **автоцентрируется** вокруг начала координат: считается половина размаха по X и Y, и начало (`_originPx`) ставится так, чтобы центр поля был в `(0,0)` плюс ручной сдвиг `gridOffsetPx`.

1.10.3. `Deploy` — построение поля

```
public void Deploy(BattlefieldConfig config)
{
    // валидация config/database/hexPrefab ...
    EnsureRoots();                          // контейнеры Background/Hexes/Obstacles
    ClearChildren(_hexRoot);
    ClearChildren(_obstacleRoot);

    ApplyBackground(config);                // фон
    var occupied = ComputeOccupancy(config);
    if (showHexGrid) BuildHexGrid(occupied);    // 15×11 гексов, занятые скрыты
    PlaceObstacles(config);                 // спрайты препятствий
}
```

Метод раскладывает `config` на три группы дочерних объектов под тремя контейнерами: `Background`, `Hexes`, `Obstacles`. Такое разбиение упрощает очистку и сортировку.

`BuildHexGrid` инстанцирует префаб гекса в каждую клетку, ставит нужный `sortingOrder`, а **занятые препятствием** гексы **скрывает** (`go.SetActive(false)`) — под большим камнем сетка не рисуется. `ComputeOccupancy` строит матрицу занятости так же, как `BuildObstacleMap` в контроллере (см. замечание про дублирование в 1.9.2).

`PlaceObstacles` создаёт `SpriteRenderer` на каждый `PlacedObstacle`, ставит его в `HexToWorld(X, Y) + spriteOffset` (учитывая визуальный сдвиг «нависающего» арта) и, если задан, применяет общий `obstacleMaterial`.

1.10.4. Умное позиционирование фона: `ApplyBackground`

Самый хитрый метод view. Проблема: фон H3 — это цельная картинка, где верхние ~17% занимает **небо/горизонт**, а гексы должны лечь только на «землю». Решение — концепция **безопасной зоны** (`backgroundSafeZone`), прямоугольника в нормализованных координатах (0..1) внутри фона, куда должна попасть сетка.

```
public Rect backgroundSafeZone = new Rect(0.03f, 0.05f, 0.94f, 0.78f);
// x=3% слева, y=5% снизу, ширина 94%, высота 78% → верхние 17% (небо) вне зоны
```

`ApplyBackground` умеет: - **нормализовать масштаб** фона (`normalizeBackgroundScale`), чтобы он рендерился в заданном PPU независимо от того, с каким PPU импортирован; - **автоподогнать фон под сетку** (`autoFitBackgroundToGrid`): вычислить масштаб `fitScale`, при котором сетка целиком помещается в безопасную зону, и **сдвинуть** фон так, чтобы центр безопасной зоны совпал с центром сетки.

Математика подгонки:

```
Vector2 gridPx = GetGridSizePx();
float needX = gridPx.x / (bgPx.x * safeZone.width);   // во сколько раз растянуть по X
float needY = gridPx.y / (bgPx.y * safeZone.height);  // во сколько по Y
fitScale = Mathf.Max(1f, Mathf.Max(needX, needY));    // берём больший, но не меньше 1
```

Затем позиция фона считается так, чтобы **центр безопасной зоны** сел в центр сетки (с учётом пивота спрайта). Результат: художник может подложить любой фон H3, задать, где у него «земля» (безопасная зона), и сетка сама аккуратно ляжет на неё, а небо останется сверху пустым.

1.10.5. Отладочные гизмо: `OnDrawGizmos`

```
private void OnDrawGizmos()
{
    if (!drawGridGizmo) return;
    Gizmos.matrix = transform.localToWorldMatrix;
    Gizmos.DrawWireCube(GetGridBounds().center, GetGridBounds().size);  // рамка сетки
    Gizmos.DrawSphere(HexToWorld(0, 0), 0.05f);                          // угол (0,0)
    Gizmos.DrawSphere(HexToWorld(Width-1, Height-1), 0.05f);             // угол (14,10)
    // + оранжевая рамка безопасной зоны фона
}
```

Гизмо — это отладочная графика, видимая только в редакторе (Scene View), не попадающая в игру. Здесь она рисует: зелёную рамку границ сетки, жёлтые шарики в угловых гексах, оранжевую рамку безопасной зоны фона. Дизайнеру это бесценно: он **видит**, куда ляжет сетка и совпадает ли она с землёй фона, ещё до запуска. Подробнее о гизмо — в 3.11.

Итог по `BattlefieldView`. Это «маляр и геодезист» поля: он и рисует (фон/гексы/ препятствия), и служит единым источником геометрии (`HexToWorld`). Умная авто-раскладка (шаг из спрайта, автоцентрирование, подгонка фона под безопасную зону) делает его гибким к любому арту. И всё это работает в Edit Mode, что превращает его в полноценный инструмент дизайнера.

1.11. Слой отображения: `UnitStackView` и `UnitViewSettings`

`UnitStackView` — `MonoBehaviour`, визуальное «тело» одного `UnitStack`. Его девиз (прямо из комментария): *«Purely reactive»* — «чисто реактивный». Он **не принимает решений**, только **реагирует** на смену состояния своего стека, проигрывая нужную анимацию. Это финальное воплощение принципа 1.1.

1.11.1. Сборка представления: `Create`

Статическая фабрика собирает объект по частям:

```
public static UnitStackView Create(Transform parent, UnitStack stack, BattlefieldView battlefield,
                                   int sortingOrder, Color placeholderTint, float moveSpeed,
                                   UnitViewSettings settings)
{
    var go = new GameObject($"Unit_{stack.Side}_{stack.SlotIndex}_{unitName}");
    go.transform.SetParent(parent, false);
    var view = go.AddComponent<UnitStackView>();
    view._battlefield = battlefield;
    view._settings = settings ?? new UnitViewSettings();
    view.moveSpeed = moveSpeed;

    view.BuildModel(stack, placeholderTint, sortingOrder);     // модель или капсула
    view.BuildActiveMarker(sortingOrder);                      // жёлтый маркер «мой ход»
    view.BuildCountLabel(sortingOrder);                        // подпись численности

    view.Bind(stack);                                          // подписка на события
    view.transform.localPosition = view.CenterFor(stack.X, stack.Y);  // 2D-позиция
    view.FaceEnemySide();                                      // развернуть к врагу
    return view;
}
```

`Create` собирает три части: модель, маркер активности и подпись численности; связывает view со стеком (`Bind`) и ставит его в 2D-точку гекса. Использование фабричного метода вместо конструктора здесь оправдано: `MonoBehaviour` нельзя создать через `new`, его добавляют через `AddComponent`, и `Create` инкапсулирует всю эту многошаговую сборку.

1.11.2. Модель или капсула-заглушка: `BuildModel`

```
private void BuildModel(UnitStack stack, Color placeholderTint, int sortingOrder)
{
    bool isPlaceholder = stack.Definition == null || stack.Definition.prefab == null;
    GameObject model;
    if (isPlaceholder)
    {
        model = GameObject.CreatePrimitive(PrimitiveType.Capsule);   // заглушка
        // убрать коллайдер, уменьшить, покрасить в placeholderTint
    }
    else
        model = Instantiate(stack.Definition.prefab);                // настоящая модель

    model.transform.localPosition = new Vector3(0, 0, _settings.modelDepth);  // толкаем к камере
    _modelRoot = model.transform;
    _animator = model.GetComponentInChildren<Animator>();

    if (_settings.steppedAnimation && _animator != null)             // навесить SteppedAnimator
    {
        var stepped = _animator.gameObject.GetComponent<SteppedAnimator>()
                   ?? _animator.gameObject.AddComponent<SteppedAnimator>();
        stepped.SetTargetFps(_settings.animationFps);
    }
    ApplyRenderOrder(model, sortingOrder);
}
```

Здесь — та самая **деградация по возможностям**: есть префаб — ставим 3D-модель, нет — рисуем капсулу-примитив нужного цвета. Логика работает одинаково в обоих случаях. У капсулы удаляется

коллайдер (он не нужен, клики обрабатывает контроллер геометрически).

Если у модели есть `Animator` и включён режим stepped-анимации, view **на лету навешивает** компонент `SteppedAnimator` (см. 1.12.1) с заданным FPS — так достигается «щёлкающая» анимация без правки самих ассетов.

`ApplyRenderOrder` пробегает все рендереры модели, ставит им `sortingOrder` и (опционально) переопределяет `renderQueue`, чтобы **непрозрачные 3D-модели рисовались поверх 2D-спрайтов поля** (см. 1.11.6 и 2.8).

1.11.3. Подпись численности и маркер активности

`BuildCountLabel` создаёт `TextMesh` (legacy 3D-текст) с числом существ. `BuildActiveMarker` — полупрозрачный жёлтый квадрат под ногами активного стека. Оба — служебная графика: `TextMesh` использует встроенный шрифт `LegacyRuntime.ttf` (`Resources.GetBuiltinResource`), маркер — сгенерированный «из ничего» белый спрайт (как подсветки в 1.9.7).

`sortingOrder` расставлен послойно: маркер под моделью (`sortingOrder - 1`), модель на `sortingOrder`, подпись над всем (`sortingOrder + 2`). Так число всегда читается, а маркер не перекрывает бойца.

1.11.4. Реактивность: подписка и `OnStateChanged`

`Bind` подписывается на события стека (и аккуратно отписывается от старого, если был):

```
public void Bind(UnitStack stack)
{
    if (Stack ≠ null) { Stack.StateChanged -= OnStateChanged; Stack.HealthChanged -= OnHealthChan
    Stack = stack;
    if (Stack ≠ null) { Stack.StateChanged += OnStateChanged; Stack.HealthChanged += OnHealthChan
    RefreshCount();
}
```

Симметричная отписка есть и в `OnDestroy` — **важнейшая деталь**: если не отписаться, стек продолжит держать ссылку на уничтоженный view, и это (а) утечка памяти, (б) исключения при попытке анимировать «мёртвый» объект. Правило: подписался — обязательно отпишись (см. 3.9).

Ядро реактивности — `OnStateChanged`, чистый диспетчер по состоянию:

```
private void OnStateChanged(UnitStack stack)
{
    switch (stack.State)
    {
        case UnitStackState.Active:
            activeMarker.SetActive(true);
            transform.localPosition = CenterFor(stack.X, stack.Y);
            FaceEnemySide();
            break;
        case UnitStackState.Idle:      activeMarker.SetActive(false); break;
        case UnitStackState.Moving:    activeMarker.SetActive(false); StartCoroutine(AnimateMove()
        case UnitStackState.Attacking: StartCoroutine(AnimateAttack()); break;
        case UnitStackState.Dead:
            activeMarker.SetActive(false);
            countText.gameObject.SetActive(false);
            SetMoving(false);
            SetTrigger(_settings.deathTrigger);   // триггер "Die" аниматору
            break;
    }
}
```

Каждому состоянию FSM — свой визуальный отклик. `Moving` и `Attacking` запускают корутины анимации. `Dead` гасит маркер/подпись и дёргает триггер смерти. `HealthChanged` отдельно обновляет подпись численности (`RefreshCount`). View **никогда** не меняет логику — он только отражает её.

1.11.5. Анимации хода и атаки

`AnimateMove` — корутина скольжения модели по пути гекс за гексом:

```
private IEnumerator AnimateMove()
{
    SetMoving(true);                          // аниматор: IsMoving = true (шаг)
    foreach (var step in Stack.MovePath)
    {
        Vector3 target = CenterFor(step.x, step.y);
        FaceTowards(target);                  // повернуть модель по ходу (BattleProjection!)
        while ((transform.localPosition - target).sqrMagnitude > 0.0001f)
        {
            transform.localPosition = Vector3.MoveTowards(transform.localPosition, target, moveSpe
            yield return null;                // двигаемся по чуть-чуть каждый кадр
        }
    }
    SetMoving(false);
    FaceEnemySide();
    Stack.CompleteMove();                     // сообщаем логике: доехали → фиксируй (X,Y)
}
```

Модель плавно скользит к каждой клетке пути через `Vector3.MoveTowards` (равномерное движение с ограничением скорости), каждый кадр (`yield return null`). Перед каждым сегментом `FaceTowards` разворачивает модель по направлению шага — здесь и вызывается `BattleProjection` (см. 1.11.7). В конце — `Stack.CompleteMove()`, который фиксирует новую позицию **в логике** (вот почему `(X,Y)` меняется только в конце — см. 1.5.4). Пока это `WaitUntil` в контроллере ждёт завершения.

`AnimateAttack` — короткий **выпад** к цели и назад:

```
private IEnumerator AnimateAttack()
{
    Vector3 origin = transform.localPosition;
    Vector3 targetCenter = CenterFor(Stack.AttackTarget);
    FaceTowards(targetCenter);
    SetTrigger(_settings.attackTrigger);      // аниматор: триггер "Attack"

    Vector3 lungePeak = Vector3.Lerp(origin, targetCenter, 0.4f);  // на 40% пути к цели
    yield return Lerp(origin, lungePeak, half);   // рывок вперёд
    yield return Lerp(lungePeak, origin, half);   // назад
    transform.localPosition = origin;
    FaceEnemySide();
    Stack.CompleteAttack();                   // сообщаем логике: атака доиграна
}
```

Стек дёргается на 40% к цели и возвращается — простой, но убедительный «удар». Одновременно дёргается триггер `Attack` аниматора (если у модели есть такая анимация).

1.11.6. 2D-позиционирование центра стека: `LocalCenter`

```
public static Vector3 LocalCenter(BattlefieldView battlefield, BattleSide side, int width, int anc
{
    Vector3 sum = Vector3.zero; int count = 0;
    foreach (int col in UnitStack.ColumnsFor(side, width, anchorX))
    {
        sum += battlefield.HexToWorld(col, y);   // мировая точка каждой занятой клетки
        count++;
    }
    return count > 0 ? sum / count : Vector3.zero;   // среднее = геометрический центр
}
```

Позиция стека — это **среднее** мировых точек всех занятых им гексов. Для обычного существа — центр одной клетки; для широкого — точка между двумя клетками. Так широкая модель встаёт ровно посередине своего двухклеточного «отпечатка». Позиция всегда берётся из `HexToWorld`, то есть остаётся честно 2D — ещё раз подчёркивая, что «объём» дают только повороты.

1.11.7. Поворот через проекцию: `SetFacing`

```
private void SetFacing(Vector2 boardDir)
{
    if (_modelRoot == null || boardDir.sqrMagnitude < 1e-6f) return;
    _modelRoot.localRotation = BattleProjection.FacingRotation(
        boardDir, _settings.viewPitchDegrees, _settings.applyViewTilt, _settings.modelYawOffset);
}
```

Вот и точка вызова `BattleProjection` из 1.8. Всякий раз, когда модель должна повернуться (`FaceTowards` при движении, `FaceEnemySide` в покое), `SetFacing` берёт направление в плоскости поля и через `FacingRotation` превращает его в 3D-кватернион «стояния на наклонной плоскости». Так движение по полю читается объёмно, хотя позиция плоская.

1.11.8. `UnitViewSettings` — настройки в одном месте

`UnitViewSettings` — сериализуемый класс-контейнер всех визуальных параметров моделей:

```
[Serializable]
public sealed class UnitViewSettings
{
    public float viewPitchDegrees = 25.06f;   // угол виртуальной камеры (для поворотов)
    public bool applyViewTilt = true;
    public float modelYawOffset = 0f;          // поправка «переда» префаба

    public float modelDepth = -0.5f;           // сдвиг по Z к камере (рисоваться поверх поля)
    public bool overrideRenderQueue = true;
    public int renderQueue = 3100;             // чуть выше очереди спрайтов (3000)

    public bool steppedAnimation = true;       // stop-motion
    [Range(2f, 30f)] public float animationFps = 10f;

    public string moveBoolParam = "IsMoving";  // имена параметров аниматора
    public string attackTrigger = "Attack";
    public string deathTrigger  = "Die";
}
```

Собрав все настройки визуализации в один класс, проект добивается двух вещей: (1) они редактируются в инспекторе `BattleController` единым блоком; (2) имена параметров аниматора (`IsMoving`, `Attack`, `Die`) **не захардкожены** в коде view, а вынесены в данные — под любой готовый аниматор их можно переназначить без правки кода. `viewPitchDegrees = 25.06°` — это тот самый угол виртуальной камеры для `BattleProjection`.

Обратите внимание, что view **проверяет наличие параметра** перед его использованием:

```
private bool HasParameter(string name, AnimatorControllerParameterType type)
{
    foreach (var p in _animator.parameters)
        if (p.type == type && p.name == name) return true;
    return false;
}
```

Если у аниматора нет параметра `IsMoving`, view просто не будет его трогать — не упадёт с ошибкой. Ещё один пример устойчивости к неполным данным.

Итог по `UnitStackView`. Это чисто реактивное «тело» отряда: собирает модель (или капсулу), подписывается на события стека, на каждое состояние проигрывает анимацию, держит позицию 2D и поворот через проекцию, аккуратно отписывается при уничтожении. Ни одного игрового **правила** — только их визуальное воплощение. Пара «`UnitStack` + `UnitStackView`» — эталонный пример разделения логики и отображения во всём проекте.

1.12. Помощники рендеринга: `SteppedAnimator`, `RetroImageEffect`, `UnitCameraCompositor`

Три `MonoBehaviour`-а отвечают за ретро-«шкурку» картинки. Логику боя они не трогают вовсе — работают «поверх» неё. Здесь дадим обзор их **кода и роли**; сами визуальные техники подробно разберём в Части 2, а Unity-механизмы под ними — в Части 3.

1.12.1. `SteppedAnimator` — анимация «щелчками»

Задача: заставить плавную анимацию `Animator` выглядеть как последовательность резких поз — эффект «stop-motion», как у спрайтов H3.

```
[RequireComponent(typeof(Animator))]
public sealed class SteppedAnimator : MonoBehaviour
{
    [Range(2f, 30f)] public float targetFps = 10f;
    private Animator _animator;
    private float _accumulator;

    private void Awake()
    {
        _animator = GetComponent<Animator>();
        _animator.enabled = false;    // ОТКЛЮЧАЕМ автообновление Unity
    }

    private void Update()
    {
        float step = 1f / targetFps;
        _accumulator += useUnscaledTime ? Time.unscaledDeltaTime : Time.deltaTime;

        int guard = 0;
        while (_accumulator ≥ step && guard++ < 8)
        {
            _animator.Update(step);   // РУЧНОЙ шаг анимации на фиксированный интервал
            _accumulator -= step;
        }
        if (_accumulator > step) _accumulator = step;   // не копим «долг» после лагов
    }
}
```

Идея. Обычно Unity сама обновляет `Animator` каждый кадр, интерполируя позу — движение гладкое. Здесь мы (1) **выключаем** автообновление (`_animator.enabled = false`) и (2) **сами** двигаем аниматор фиксированными шагами `1/targetFps` секунд. Между шагами поза **заморожена**, поэтому движение читается как серия «щёлкающих» кадров.

Тонкости кода: - **Аккумулятор времени.** Копим `deltaTime`, и как только накопилось на целый шаг — продвигаем анимацию. Если игра «лагнула» и накопилось на несколько шагов, применяем их в цикле (`while`), чтобы **тайминг** оставался верным, хотя **видимая** поза меняется только на границе шага. - `guard < 8` — предохранитель от «спирали смерти»: после долгой паузы не даём прокрутить сотни догоняющих шагов за один кадр. - `[RequireComponent(typeof(Animator))]` — Unity гарантирует, что на объекте есть `Animator`, иначе не даст добавить компонент.

Важно: параметры (`SetBool`, `SetTrigger`), которые ставит `UnitStackView`, **не теряются** — они применяются на следующем ручном `Update`. То есть stepped-режим совместим с обычным управлением аниматором. Работает с любым клипом без правки ассетов.

1.12.2. `RetroImageEffect` — пикселизация и постеризация

Пост-эффект камеры для BiRP. Навешивается на камеру, обрабатывает **весь отрисованный кадр**.

```
[ExecuteAlways]
[RequireComponent(typeof(Camera))]
public sealed class RetroImageEffect : MonoBehaviour
{
    public bool pixelate = true;
    [Range(64, 720)] public int targetHeight = 240;   // до какого разрешения сжать
    [Range(1, 64)] public int colorLevels = 12;       // сколько уровней цвета на канал
    public float saturation = 1.15f, gamma = 1f;
    public Texture2D palette; public int paletteSize; // опциональная фикс-палитра

    private void OnRenderImage(RenderTexture source, RenderTexture destination)
    {
        // 1. Толкаем параметры цвета в материал (шейдер RetroPost)
        _material.SetFloat("_Levels", colorLevels);
        _material.SetFloat("_Saturation", saturation);
        _material.SetFloat("_Gamma", gamma);
        // ... _UsePalette / _PaletteTex / _PaletteSize

        if (!pixelate || targetHeight ≥ source.height)
        { Graphics.Blit(source, destination, _material); return; }  // только цвет

        // 2. Пикселизация: вниз до низкого разрешения, потом вверх с point-фильтром
        int lowH = Mathf.Max(16, targetHeight);
        int lowW = Mathf.RoundToInt(lowH * ((float)source.width / source.height));
        var lowRes = RenderTexture.GetTemporary(lowW, lowH, 0, source.format);

        Graphics.Blit(source, lowRes);           // downscale
        lowRes.filterMode = FilterMode.Point;    // «чёткий» upscale (без размытия)
        Graphics.Blit(lowRes, destination, _material);  // upscale + цветовой шейдер
        RenderTexture.ReleaseTemporary(lowRes);
    }
}
```

Эффект — это **цепочка блитов** (копирований текстур), делающая две вещи: 1. **Пикселизация** — кадр сжимается до низкого разрешения (`targetHeight`, напр. 240 строк), затем растягивается обратно с **точечной** фильтрацией (`FilterMode.Point`), отчего пиксели получаются крупными и чёткими, как в старой спрайтовой игре. 2. **Цвет** — на финальном блите отрабатывает шейдер `Hidden/Heroes/RetroPost` (разбор в 2.4), который постеризует цвета (сводит к нескольким уровням) и опционально привязывает их к фиксированной палитре.

`OnRenderImage` — специальный метод BiRP, вызываемый после рендера камеры с исходным кадром в `source` и результатом в `destination` (см. 3.6). `RenderTexture.GetTemporary` / `ReleaseTemporary` берут/возвращают временную текстуру из пула — эффективно, без постоянных аллокаций. `[ExecuteAlways]` позволяет крутить эффект прямо в редакторе.

1.12.3. `UnitCameraCompositor` — изоляция эффекта на камере юнитов

Это решение хитрой проблемы BiRP. Разберём её, потому что она поучительна.

Проблема. В сцене две камеры: одна рисует 2D-поле (фон, гексы), другая — 3D-юнитов поверх. Мы хотим ретро-эффект **только на юнитах** (или наоборот — по-разному на слоях). Но в BiRP «стопка» камер, рисующих в экран, **делит один буфер кадра**. Поэтому `OnRenderImage` второй камеры получает **весь накопленный кадр** (включая фон от первой камеры) — и эффект накладывается на всё, а не только на юнитов.

Решение (из комментария в коде): рендерить камеру юнитов **не в экран, а в собственную** `RenderTexture`, прогнать эффект там, а результат **скомпоновать** обратно поверх экрана через полноэкранный `RawImage` на overlay-канвасе.

```
[ExecuteAlways]
[RequireComponent(typeof(Camera))]
public sealed class UnitCameraCompositor : MonoBehaviour
{
    private void EnsureComposite()
    {
        // 1. Создать/пересоздать RT под размер экрана
        _rt = new RenderTexture(w, h, 24, RenderTextureFormat.ARGB32);

        // 2. Камера рисует ТОЛЬКО в RT, очищаясь в ПРОЗРАЧНЫЙ цвет
        _camera.targetTexture = _rt;
        _camera.clearFlags = CameraClearFlags.SolidColor;
        _camera.backgroundColor = new Color(0, 0, 0, 0);   // альфа 0 → фон просвечивает

        // 3. Показать RT полноэкранным RawImage на overlay-канвасе
        EnsureImage();
        _image.texture = _rt;
    }
}
```

Что тут важно: - Камера юнитов рендерит в `_rt`, а **не** в экран (`targetTexture = _rt`). - Она очищается в **прозрачный** цвет (`alpha = 0`), поэтому пустые места RT прозрачны — сквозь них виден фон, нарисованный другой камерой. - `RetroImageEffect` на этой камере теперь трогает **только** содержимое RT (юнитов). - Результат кладётся на полноэкранный `RawImage` поверх экрана — композитинг завершён. - Компонент **сам создаёт** overlay-канвас и `RawImage`, если их нет, и **пересоздаёт** RT при смене разрешения экрана (`EnsureComposite` вызывается каждый `Update`).

Это отличный учебный пример того, как знание внутренностей конвейера рендеринга (общий буфер кадра в BiRP) диктует архитектуру решения. Подробнее про `RenderTexture` — в 3.7.

1.13. Редакторные инструменты: три окна и кастомный инспектор

Папка `Assets/Scripts/Battle/Editor/` — это **инструменты дизайнера**. Код здесь исполняется **только в Unity Editor** и **не попадает в билд** (Unity автоматически исключает всё в папках с именем `Editor`). Пространство имён — `Heroes.Battle.Editor`.

Эти классы наследуют `EditorWindow` / `UnityEditor.Editor` — их разбор с точки зрения Unity-API см. в 3.10. Здесь — что каждый делает.

1.13.1. `BattlefieldEditorWindow` — окно «Battlefield Generator»

Открывается через меню `Tools → Heroes 3 → Battlefield Generator` (пункт добавляет атрибут `[MenuItem]`). Позволяет: - задать **seed** и **тип местности** (кнопка «Randomize» кидает новый seed); - назначить ссылки на **ObstacleDatabase**, **префаб гекса** и **материал препятствий**; - нажать **Generate** — и в открытой сцене появится сгенерированное поле.

Под капотом кнопка «Generate» делает ровно то, что мы разбирали:

```
var config = BattlefieldGenerator.Generate(_seed, _terrain, _database);  // чистая логика
_target.Deploy(config);                                                   // отображение
```

Плюс важные редакторные детали: - `Undo.RegisterFullObjectHierarchyUndo` — регистрирует операцию в системе отмены, чтобы Ctrl+Z вернул сцену как было. Инструмент обязан «дружить» с Undo. - `EditorSceneManager.MarkSceneDirty` — помечает сцену изменённой, чтобы Unity предложила её сохранить. - `EditorPrefs` — ссылки на базу/префаб/материал **запоминаются между сессиями** (хранятся по путям ассетов), чтобы не назначать их каждый раз. Есть даже дефолтный материал по GUID.

1.13.2. `BattleTestWindow` — окно «Battle Test»

Главный игровой инструмент. Открывается через `Tools → Heroes 3 → Battle Test`. Позволяет: - заполнить **обе армии** (по 7 слотов): выбрать существо и численность; - настроить местность/seed/генерацию; - кнопкой **Start Battle** запустить бой через `BattleController.StartBattle`; - **Auto-Fill Random** — заполнить слоты случайными существами из проекта; - **Clear Slots / Clear Units** — очистка.

Ключевые приёмы: - **Сохранение состояния окна** через `EditorPrefs` + `JsonUtility`: класс `WindowState` сериализуется в JSON, ссылки на существа хранятся как **GUID ассетов** (стабильнее путей). При переоткрытии окна армии восстанавливаются.

`csharp private void SaveState() ⇒ EditorPrefs.SetString(PrefKeyState, JsonUtility.ToJson(_state));` - **Работа и в Edit, и в Play Mode.** В Edit Mode перед стартом регистрируется Undo и сцена метится «грязной»; в Play Mode запускается интерактивный цикл. Один инструмент — оба режима. - **Автосоздание контроллера.** Если на сцене нет `BattleController`, окно создаёт его само (`EnsureController`), лишь бы был `BattlefieldView`. - **Понятные диалоги.** Если не хватает базы или префаба гекса, окно показывает `EditorUtility.DisplayDialog` с объяснением, а не молча падает.

1.13.3. `ObstacleDefinitionEditor` — авторасчёт заблокированных гексов

Это **кастомный инспектор** для `ObstacleDefinition` (атрибут `[CustomEditor(typeof(ObstacleDefinition))]`). Он добавляет к обычному инспектору кнопку **«Auto-compute Blocked Hexes (cyan = empty)»**, которая **сама** заполняет список `blockedHexes`, анализируя пиксели спрайта препятствия.

Идея алгоритма `AutoCompute`: 1. Прочитать пиксели спрайта (через блит в `RenderTexture` — чтобы **не** требовать включённого «Read/Write» у текстуры; трюк в `ReadTexturePixels`). 2. Пройти по всем пикселям области спрайта. Пиксель считается **пустым**, если он прозрачный (alpha = 0) **или** окрашен в **голубой ключ** `00FFFF` (cyan, с допуском) — метод `IsEmpty`. 3. Каждый **непустой** пиксель перевести из пиксельных координат спрайта в мировые (с учётом пивота, PPU и `spriteOffset`), затем найти **ближайший гекс** (`NearestHex`) в текущей раскладке поля. 4. Собрать множество занятых гексов, превратить в список `HexOffset`, отсортировать и записать в ассет (с регистрацией `Undo.RecordObject`).

Тонкости: - `ResolveHexStepsWorld` берёт шаги гекса из **реальной** `BattlefieldView` на сцене (если она есть), чтобы маска совпала с фактической раскладкой; иначе — из констант `BattlefieldConfig`. - `ReadTexturePixels` через `Graphics.Blit` в временную RT + `ReadPixels` получает читаемый снимок текстуры **без** изменения настроек импорта ассета. Красивый обход ограничения «текстура не Readable». - `Pack` / `Unpack` упаковывают пару `(x, y)` в один `long` для хранения в `HashSet` (быстрая проверка уникальности гексов).

Практическая ценность огромна: художник рисует спрайт препятствия, помечает «пустоту» голубым ключевым цветом, жмёт кнопку — и «отпечаток» блокируемых гексов считается автоматически, без ручного ввода координат. Это экономит часы и исключает ошибки.

1.14. Отдельные учебные скрипты: `PongGame`, `Example`

Два скрипта в `Assets/Scripts/` не связаны с боевой системой — это отдельные учебные работы. Разберём их для полноты (и потому что `PongGame` — прекрасный пример «ручной физики»).

1.14.1. `PongGame` — классический Pong на «ручной» физике

`PongGame` — самостоятельная мини-игра: две ракетки и мяч. Что в ней поучительно — **вся физика написана вручную**, без `Rigidbody` и коллайдеров. Это отличный контрпример к «физике движка».

Структура: - **Игрок 1** управляется клавишами (`W` / `S`), **игрок 2** — мышью (ракетка следует за курсором по Y). - **Мяч** двигается по вектору направления `_ballDirection` с постоянной скоростью.

Ключевые механики:

```
private void MoveBall()
{
    Vector3 newPosition = _ball.position + (Vector3)(_ballDirection * _ballSpeed * Time.deltaTime)

    // Отскок от верхней/нижней стенки — инверсия Y-компоненты
    if (newPosition.y + _ballHalfSize > _topY && _ballDirection.y > 0f)
    { newPosition.y = _topY - _ballHalfSize; _ballDirection.y = -_ballDirection.y; }
    //  ...  нижняя стенка симметрично

    // Столкновение с ракетками — проверка пересечения AABB
    if (_ballDirection.x < 0f && Intersects(newPosition, _player1,  ... ))
        BounceOffPaddle(_player1,  ... , 1f, ref newPosition);
    //  ...  правая ракетка

    // Гол — мяч ушёл за край
    if (newPosition.x < _leftX) { _player2Score++; ResetBall(); return; }
    if (newPosition.x > _rightX) { _player1Score++; ResetBall(); return; }

    _ball.position = newPosition;
}
```

Три техники, которые стоит усвоить:

1. **Отскок = инверсия компоненты скорости.** От горизонтальной стенки инвертируется `y` (`_ballDirection.y = -_ballDirection.y`), от ракетки — меняется `x`. Плюс мяч «выталкивается» из стенки (`newPosition.y = _topY - _ballHalfSize`), чтобы не «залипнуть».
2. **AABB-пересечение** (`Intersects`) — проверка столкновения двух прямоугольников по осям: объекты пересекаются, если расстояние между центрами по каждой оси меньше суммы полуразмеров. Простейший и самый быстрый тест столкновений.
3. **Управляемый угол отскока** (`BounceOffPaddle`): куда пришёл мяч по высоте ракетки, туда и отскочит под углом — попал в край ракетки, полетит круче. Это делает игру управляемой, а не случайной:

`csharp float relative = Mathf.Clamp((ballPos.y - paddle.position.y) / paddleHalfHeight, -1f, 1f); float angle = relative * _maxBounceAngleDeg * Mathf.Deg2Rad; _ballDirection = new Vector2(directionX * Mathf.Cos(angle), Mathf.Sin(angle)).normalized;`

`GetHalfExtents` аккуратно достаёт полуразмеры объектов из `Renderer`, затем `Collider2D`, а если ничего нет — из `localScale`. `Time.deltaTime` в движении делает скорость независимой от частоты кадров (см. 3.1). Счёт выводится в `TMP_Text` (TextMeshPro, см. 3.12).

Почему ручная физика? Для Pong она даёт полный контроль (точные углы отскока, никаких «дрожаний» физдвижка) и учит основам: векторы скорости, интегрирование позиции по времени, детект столкновений. В большой игре так делать не стоит, но как урок — идеально.

1.14.2. `Example` — пустой шаблон

```
public class Example : MonoBehaviour
{
    void Start() { }
    void Update() { }
}
```

Это стандартная **заготовка** нового скрипта Unity: пустые `Start` (вызывается один раз перед первым кадром) и `Update` (каждый кадр). Файл ничего не делает — это «чистый лист» для урока или эксперимента. Мы упоминаем его только для полноты картины проекта.

На этом Часть 1 завершена. Мы прошли по каждому скрипту: от «немых» данных (`TerrainType`, `BattlefieldConfig`) через чистое ядро логики (`UnitStack`, `BattleGrid`, `BattlefieldGenerator`, `BattleProjection`), слой отображения (`BattleController`, `BattlefieldView`, `UnitStackView`), помощники рендеринга и редакторные инструменты — до отдельных учебных скриптов. Красной нитью через всё прошёл один принцип: **отделяй правила от их отображения**. Теперь, зная «что есть», перейдём к «как это выглядит» — к техникам визуализации.

Часть 2. Какие техники визуализации мы используем

В этой части мы смотрим на проект **глазами художника и графического программиста**. Если Часть 1 отвечала на вопрос «что делает код», то здесь — «почему картинка выглядит именно так, как в Heroes 3». Мы разберём восемь-девять взаимодополняющих техник, каждая из которых вносит свой вклад в фирменный ретро-вид.

Список техник и их вклад в общий образ:

Техника	Что даёт	Где в коде
Гексовая сетка odd-r	Тактическое поле как в H3	`BattleGrid`, `BattlefieldView.HexToWorld`
Псевдо-3D (2D-позиция + 3D-поворот)	Объёмные бойцы на плоском поле	`BattleProjection`, `UnitStackView`
Плоское (cel) освещение	Модели «как нарисованные»	`HeroesFlat.shader`
Пикселизация + постеризация	Низкое разрешение и палитра 90-х	`RetroImageEffect`, `RetroPost.shader`
Stepped-анимация	«Щёлкающее» движение спрайтов	`SteppedAnimator`
Композитинг камеры	Эффект только на юнитах	`UnitCameraCompositor`
Прозрачность по ключу	Обрезка фона спрайтов	`ColorKeySprite.shader`
Порядок отрисовки	Правильное перекрытие слоёв	sorting order / render queue
Служебная графика без ассетов	Подсветки, маркеры	`Sprite.Create` из белой текстуры

2.1. Гексовая сетка «odd-r offset»

Первое, что делает игру похожей на H3, — **шестиугольное поле боя 15×11**. Разберём эту технику визуализации целостно (алгоритмическую сторону мы уже видели в 1.6, здесь — про геометрию и отображение).

2.1.1. Почему гексы, а не квадраты

У шестиугольной клетки **шесть равноудалённых соседей**, тогда как у квадрата — либо четыре (без диагоналей), либо восемь, но диагональные дальше по расстоянию. Гекс решает «проблему диагоналей»: движение во всех шести направлениях одинаково «стоит». Для тактической игры это честнее и красивее — отсюда выбор H3 (и нашего проекта).

2.1.2. Pointy-top и раскладка odd-r

Гексы у нас **остриём вверх** (pointy-top). Такие гексы удобно укладывать рядами: каждый следующий ряд вставляется в «зубцы» предыдущего, поэтому **вертикальный шаг между рядами — ¾ высоты гекса**, а ряды перекрываются на четверть. Это прямо закодировано в `ResolveLayout`:

```
_stepYpx = Mathf.RoundToInt(spriteHpx * 0.75f);   // ¾ высоты
_offXpx  = Mathf.RoundToInt(spriteWpx * 0.5f);    // полширины — сдвиг нечётных рядов
```

«odd-r offset» означает: нумеруем ряды сверху вниз (row), и **нечётные ряды сдвигаем вправо** на полшага. Это самый интуитивный способ хранить гексы в обычном прямоугольном массиве `[Width, Height]`: координаты гекса — это просто `(столбец, строка)`, как у таблицы.

Плата за удобство хранения — то, что **соседи зависят от чётности ряда** (см. таблицу `Directions` в 1.6.1). Но это спрятано в двух таблицах направлений и больше нигде не мешает.

2.1.3. Единый источник геометрии

Критично, что и **логика** (`BattleGrid.Neighbors`), и **отображение** (`BattlefieldView.HexToWorld`) используют **одну и ту же** odd-r раскладку. `HexToWorld` кладёт нечётные ряды со сдвигом `_offXpx`, а `Directions` описывает соседей ровно для такой укладки. Если бы они разошлись, подсветка достижимых клеток «уехала» бы относительно нарисованных гексов. Единый источник геометрии — залог того, что «где логика думает, там глаз и видит».

Числа раскладки (`44/32/22` px при PPU 64) взяты из VCMI, чтобы поле совпадало с оригинальной H3. Но благодаря `autoComputeSpacing` проект не привязан к ним намертво — шаг вычисляется из размера спрайта, и можно подложить гекс-текстуру любого разрешения.

2.2. Псевдо-3D: 2D-позиция + 3D-поворот через виртуальную наклонную плоскость

Это **центральная визуальная идея** проекта и, пожалуй, самая интересная. Математику мы уже разобрали в 1.8 (`BattleProjection`); здесь — про то, **какой визуальный результат** она даёт и почему такой подход выбран.

2.2.1. Дилемма: 2D-поле, но объёмные бойцы

H3 выглядела так: плоское нарисованное поле (2D-спрайт фона и клеток) — и на нём «живые» существа, которые поворачиваются, идут, бьют. В оригинале бойцы были **спрайтами** с покадровой анимацией под каждый угол. Мы же хотим использовать **настоящие 3D-модели** (их проще анимировать и поворачивать), но сохранить **плоское 2D-поле** и его простую раскладку.

Конфликт: если поставить 3D-модель на 2D-поле и снимать её обычной прямой камерой, движение «вглубь» поля (в дальние ряды) никак не читается — модель просто ползёт вверх по экрану, как наклейка.

2.2.2. Решение: виртуальная наклонная камера — только для поворота

Проект делает элегантный трюк: - **Позиция** модели остаётся строго 2D — экранная точка гекса из `HexToWorld` (`LocalCenter` в `UnitStackView`). - **Поворот** вычисляется так, будто модель стоит на **наклонной** земле, которую снимает **наклонная** камера под углом ~25°. Это делает `BattleProjection.FacingRotation`.

Результат: реальная камера — обычная ортографическая, смотрит прямо; поле — плоское; но **повёрнутые** модели создают убедительную иллюзию, что они стоят на уходящей вглубь сцене. Идущий в дальние ряды боец **разворачивается**, как будто удаляется от нас по наклонному полу. Отсюда «2.5D»: рендер плоский, но ощущение объёмное.

2.2.3. Почему это дёшево и надёжно

- **Одна ортокамера, плоская раскладка.** Не нужно строить настоящую 3D-сцену, расставлять модели в 3D-пространстве, настраивать перспективу и глубину. Вся раскладка поля — 2D- арифметика в пикселях.
- **Только поворот.** Меняется единственное свойство — `localRotation` корня модели. Позиция, сортировка, физика — всё остаётся простым 2D.
- **Настраиваемость.** Угол `viewPitchDegrees` (25.06°) и поправка `modelYawOffset` вынесены в `UnitViewSettings` — вид подстраивается под любой арт без правки кода.

Это образцовый пример геймдев-мышления: **дешёвый визуальный трюк вместо дорогого честного решения**. Игрок видит объём, а под капотом — плоская сетка и одна формула поворота.

2.3. Плоское (cel / toon) освещение — шейдер `Heroes/Flat`

Чтобы 3D-модели не выглядели как глянцевые PBR-объекты из современной игры, а читались как **нарисованные** спрайты H3, на них вешается кастомный шейдер `HeroesFlat.shader` (`Heroes/Flat`). Разберём его — это отличное введение в шейдеры для BiRP.

2.3.1. Что такое cel-shading

Cel-shading (тунарное затенение) — техника, имитирующая рисованную графику: вместо плавного градиента света по поверхности она даёт **несколько плоских зон** («светло» / «полутень» / «тень») с резкими границами. Так рисуют в мультфильмах (cel = целлулоидный лист анимации) и так выглядели спрайты 90-х.

2.3.2. Ключевая строка — квантование света в «полосы»

Сердце шейдера — фрагментный шейдер, где диффузный свет разбивается на дискретные ступени:

```
float ndotl = saturate(dot(n, l));        // сколько света падает (0..1), классический ламберт
float bands = max(1.0, floor(_Bands));    // число полос (по умолчанию 3)
float stepped = floor(ndotl * bands) / bands;   // КВАНТОВАНИЕ: непрерывное → ступени
```

Обычное ламбертово освещение даёт `ndotl` — плавную величину от 0 (тень) до 1 (полный свет). Строка `floor(ndotl * bands) / bands` **округляет** её вниз до ближайшей из `bands` ступеней. При `bands = 3` весь плавный градиент превращается в **три** плоские зоны. Именно этот `floor` создаёт резкие «мультяшные» границы света вместо гладкого перехода.

2.3.3. Тёплый свет / холодная тень и плоский ambient

Дальше банды смешивают два цвета — холодный оттенок тени и цвет источника света:

```
float3 lit = lerp(_ShadowTint.rgb, _LightColor0.rgb, stepped);   // тень→свет по ступеням
float3 ambient = unity_AmbientSky.rgb + _AmbientBoost;           // плоская подсветка
float3 color = albedo.rgb * (lit + ambient);
```

- `_ShadowTint` (по умолчанию холодный сине-серый) окрашивает теневые зоны — тени не «чёрные», а живописно-синеватые, как на картинах. `lerp` по ступенчатому `stepped` сохраняет резкие границы.

- `_AmbientBoost` гарантирует, что теневая сторона не проваливается в полную черноту — ещё один приём «рисованности».

2.3.4. Только главный свет и опциональный rim

```
Tags { "LightMode"="ForwardBase" }   // только основной направленный свет
```

Тег `ForwardBase` означает, что шейдер учитывает **только главный направленный свет** сцены, игнорируя остальные. Это делает вид **предсказуемым и плоским** независимо от того, сколько ламп на сцене — как и должно быть у «нарисованной» модели. Опциональный **rim light** (подсветка силуэта по краю) добавляет контур, но по умолчанию выключен (`_RimColor` с альфа 0). Есть и `_AlphaClip` для вырезания прозрачных частей.

Итог: шейдер `Heroes/Flat` заменяет реалистичный PBR на резко-банданное освещение с тёплым светом и холодной тенью — модели начинают читаться как крашеные фигурки/спрайты, а не как объекты из современного 3D-движка. Как шейдер устроен на уровне Unity (проходы, `CGPROGRAM`, семантики) — в 3.6.

2.4. Ретро-постобработка: пикселизация + постеризация + палитра

Даже с плоскими моделями и гексовым полем картинка всё ещё «слишком чистая» для 1999 года. Финальный штрих — **постобработка всего кадра**: пикселизация (снизить разрешение) и постеризация (ограничить палитру). Код-оркестратор — `RetroImageEffect` (1.12.2), цветовую работу делает шейдер `RetroPost.shader`. Разберём обе половины как единую технику.

2.4.1. Пикселизация — «down-up» блит

Пикселизация делается **без шейдера**, чистой игрой с разрешением (см. код в 1.12.2): 1. Кадр **сжимается** (`Graphics.Blit`) до низкого разрешения — например, 240 строк по высоте. 2. Затем **растягивается** обратно на экран, но с **точечной** фильтрацией (`FilterMode.Point`). Point-фильтр не сглаживает пиксели при увеличении — каждый маленький пиксель превращается в чёткий крупный «блок».

Результат — крупные, резкие пиксели, как на низком разрешении старого монитора. Чем меньше `targetHeight`, тем «чанковее» картинка. Обратите внимание: ширина низкого буфера считается из **соотношения сторон** экрана, чтобы картинка не искажалась:

```
int lowW = Mathf.RoundToInt(lowH * ((float)source.width / source.height));
```

2.4.2. Постеризация — квантование цвета

Цветовую часть делает шейдер `Hidden/Heroes/RetroPost` на финальном блите. Ключевая функция:

```
fixed3 ApplyPosterize(fixed3 c, float levels)
{
    if (levels ≤ 1.0) return c;
    return floor(c * levels + 0.5) / levels;    // округляем каждый канал к ближайшему уровню
}
```

Постеризация **сокращает число возможных значений** каждого цветового канала. При `levels = 12` каждый из R, G, B может принимать лишь 12 значений вместо 256. Плавные градиенты превращаются в

дискретные полосы цвета — характерный вид ограниченной палитры старых игр. Формула `floor(c * levels + 0.5) / levels` — это округление к ближайшему из `levels` уровней (`+0.5` даёт округление к ближайшему, а не вниз).

2.4.3. Фиксированная палитра — привязка к LUT

Опционально можно пойти дальше и привязать **каждый пиксель к ближайшему цвету из заданной палитры** (например, точной палитры H3):

```
fixed3 ApplyPalette(fixed3 c)
{
    float best = 1e9; fixed3 bestColor = c;
    [loop]
    for (int idx = 0; idx < 256; idx++)
    {
        if (idx ≥ count) break;
        float u = (idx + 0.5) / _PaletteSize;
        fixed3 p = tex2D(_PaletteTex, float2(u, 0.5)).rgb;   // цвет из палитры-текстуры
        float d = dot(c − p, c − p);                          // квадрат расстояния в RGB
        if (d < best) { best = d; bestColor = p; }            // ближайший
    }
    return bestColor;
}
```

Палитра хранится как **1-пиксельная по высоте текстура** (LUT, look-up table), цвета лежат слева направо. Для каждого пикселя шейдер перебирает палитру и находит **ближайший** цвет по евклидову расстоянию в RGB. Это даёт **истинно фиксированную** палитру — картинка использует ровно те цвета, что заданы, и никакие другие. `[loop]` и ограничение `< 256` — требования безопасности шейдера (динамические циклы в шейдерах ограничивают сверху).

2.4.4. Порядок операций и разделение труда

В шейдере операции идут в осмысленном порядке:

```
col.rgb = pow(saturate(col.rgb), _Gamma);              // 1. гамма
float luma = dot(col.rgb, float3(0.299,0.587,0.114));  // 2. яркость (веса восприятия)
col.rgb = lerp(luma.xxx, col.rgb, _Saturation);        //    насыщенность
col.rgb = ApplyPosterize(col.rgb, _Levels);            // 3. постеризация
if (_UsePalette > 0.5) col.rgb = ApplyPalette( … );    // 4. палитра
```

Сначала гамма и насыщенность **формируют** цветовой тон (насыщенность 1.15 делает картинку чуть сочнее, как в старых играх), потом квантование. Коэффициенты `0.299/0.587/0.114` — это стандартные веса восприятия яркости (глаз чувствительнее к зелёному).

Важное архитектурное решение: пикселизация делается в C# (игра с разрешением), а цвет — в шейдере. Они **композируются** чисто: шейдер трогает только цвет и потому «не мешает» пикселизации. Разделение «геометрия эффекта — в коде, цвет — в шейдере» делает обе части простыми и переиспользуемыми.

2.5. Пошаговая («stop-motion») анимация

Ещё один слой ретро-достоверности — **низкий FPS анимации**. Код (`SteppedAnimator`) мы разобрали в 1.12.1; здесь — про **визуальный смысл**.

Спрайты H3 анимировались небольшим числом кадров (порядка 8–12 в секунду), поэтому движения «щёлкали» позами, а не текли плавно. Современный `Animator` по умолчанию интерполирует позу **каждый** отрисованный кадр (60+ FPS) — движение слишком гладкое, «слишком 3D».

`SteppedAnimator` возвращает ретро-ощущение: он **замораживает** позу между шагами `1/targetFps` секунд и продвигает аниматор только на границах шага. При `targetFps = 10` модель меняет позу 10 раз в секунду — ровно тот «покадровый» вид старых спрайтов. Настройка `animationFps` в `UnitViewSettings` (2–30) позволяет крутить «степень ретро»: ниже — более дёрганно и олдскульно, выше — плавнее.

Красота приёма — в **неинвазивности**: он работает с любым готовым 3D-аниматором и клипом, не требуя ни перерисовки анимации, ни специальных ассетов. Просто навешивается на объект и «портит» плавность ровно так, как нужно для стиля.

2.6. Композитинг камеры юнитов через `RenderTexture`

Техника (`UnitCameraCompositor`, код в 1.12.3) отвечает на вопрос **«как применить ретро- эффект избирательно»** — например, только к 3D-юнитам, не трогая 2D-фон (или наоборот, применяя к ним разные настройки).

Визуальная задача: юниты и поле — это разные «миры» (3D-модели и 2D-спрайты), и им может требоваться разная обработка. Но в BiRP камеры, рисующие в экран, делят общий буфер, и пост-эффект второй камеры цепляет всё.

Решение — **закадровый рендер**: камера юнитов рисует в свою `RenderTexture` (не в экран), очищаясь в **прозрачный** фон. Ретро-эффект отрабатывает **на этой RT** — то есть только на юнитах. Затем RT кладётся полноэкранной картинкой поверх сцены, и сквозь её прозрачные области виден фон, нарисованный другой камерой. Итог: два слоя обрабатываются независимо и чисто компонуются. Это классический приём «render-to-texture + composite», и он подробно разобран с точки зрения Unity в 3.7.

2.7. Прозрачность по ключевому цвету — шейдер `ColorKeySprite`

`ColorKeySprite.shader` (`Custom/ColorKeySprite`) — спрайтовый шейдер, делающий прозрачными пиксели заданного **ключевого цвета**. Это цифровой аналог «зелёного экрана» (chroma key).

```
fixed4 frag(v2f IN) : SV_Target
{
    fixed4 tex = SampleSpriteTexture(IN.texcoord) * IN.color;
    float dist = distance(tex.rgb, _KeyColor.rgb);                // близость к ключу
    float mask = smoothstep(_Threshold, _Threshold + _Softness, dist);
    tex.a *= mask;                    // пиксели близко к ключу → прозрачные
    tex.rgb *= tex.a;                 // premultiplied alpha
    return tex;
}
```

Как работает: для каждого пикселя считается **расстояние** его цвета до ключевого (`_KeyColor`, по умолчанию пурпурный `1,0,1`). Если пиксель близок к ключу (`dist` мал), `smoothstep` даёт маску около 0 → пиксель становится прозрачным. `_Threshold` задаёт порог срабатывания, `_Softness` — мягкость края (плавный переход вместо резкого).

Зачем это нужно: старые спрайты часто хранят «пустоту» не прозрачностью, а сплошным ключевым цветом (cyan/magenta). Такой шейдер позволяет использовать подобные ассеты напрямую — «фон» спрайта вырезается на лету, без ручного редактирования альфа-канала. Обратите внимание на связь с `ObstacleDefinitionEditor` (1.13.3): там пиксели цвета `00FFFF` тоже трактуются как «пусто» — тот же принцип ключевого цвета, но для расчёта заблокированных гексов.

Строка `tex.rgb *= tex.a` — это **premultiplied alpha** (предумноженная прозрачность), согласованная с режимом смешивания `Blend One OneMinusSrcAlpha` в шейдере. Это корректный способ смешивания полупрозрачных краёв без тёмной каймы.

2.8. Порядок отрисовки: sorting order и render queue

Когда на экране слои (фон, гексы, препятствия, модели, подписи, подсветки), критично, **что рисуется поверх чего**. Проект аккуратно управляет этим двумя механизмами.

2.8.1. `sortingOrder` — для 2D и внутри слоёв

Для спрайтов и рендереров используется `sortingOrder` — целое число, где больше = рисуется позже (поверх). В коде расставлена чёткая иерархия:

```
backgroundSortingOrder = 0   // фон — в самом низу
hexSortingOrder        = 1   // гексы поверх фона
obstacleSortingOrder   = 2   // препятствия поверх гексов
highlightSortingOrder  = 2   // подсветки на уровне препятствий
unitSortingOrder       = 3   // юниты поверх всего
  └ маркер активности:  unitSortingOrder - 1  (под юнитом)
  └ подпись числа:      unitSortingOrder + 2  (над юнитом)
```

Так каждый визуальный слой знает своё место: фон внизу, бойцы наверху, подпись численности — поверх бойца, маркер «мой ход» — под ним. Всё это задаётся числами в инспекторе и легко перенастраивается.

2.8.2. `renderQueue` — чтобы 3D-модели рисовались поверх 2D-спрайтов

Есть тонкость: 2D-спрайты и непрозрачная 3D-геометрия сортируются **по-разному** (спрайты — по `sortingOrder`, непрозрачные модели — по глубине/очереди материала). Чтобы гарантировать, что непрозрачные модели лягут **поверх** спрайтов поля, `UnitStackView.ApplyRenderOrder` переопределяет `renderQueue` материалов модели:

```
if (_settings.overrideRenderQueue)
    foreach (var material in renderer.materials)
        material.renderQueue = _settings.renderQueue;    // 3100 — чуть выше очереди спрайтов (3000
```

В Unity очередь `3000` — это «Transparent» (где рисуются спрайты). Ставя моделям `3100`, мы говорим движку рисовать их **после** спрайтов, то есть поверх. Плюс модели физически сдвинуты к камере по Z (`modelDepth = -0.5`). Комбинация «render queue + Z-сдвиг + sorting order» надёжно кладёт бойцов поверх плоского поля.

Это тонкий, но важный момент: смешение 2D и 3D в одной сцене требует ручного согласования порядка отрисовки, иначе модели «тонут» в спрайтах или мерцают. Проект решает это явно.

2.9. Служебная графика без ассетов

Небольшая, но приятная техника: **подсветки, маркеры и подписи создаются в коде, без художественных ассетов**. Мы уже видели это в 1.9.7 и 1.11.3.

- **Подсветки клеток** (синие/красные) и **маркер активности** (жёлтый квадрат) — это спрайты, сгенерированные из встроенной `Texture2D.whiteTexture` через `Sprite.Create`, а затем покрашенные нужным цветом через `SpriteRenderer.color`.
- **Подпись численности** — legacy `TextMesh` со встроенным шрифтом `Resources.GetBuiltinResource<Font>("LegacyRuntime.ttf")`.

Зачем так? Служебная графика (индикаторы, подсветки) не требует уникального арта — квадрат нужного цвета решает задачу. Генерируя её в коде, проект: (1) не плодит мелкие ассеты; (2) делает цвет полностью настраиваемым из инспектора; (3) остаётся самодостаточным. Спрайты кэшируются статически (создаются один раз) и помечаются `HideFlags.DontSave`, чтобы не засорять сцену и не сохраняться в ассеты.

Итог Части 2. Ретро-вид H3 в проекте — это не один эффект, а **стек согласованных техник**: гексовое поле задаёт композицию, псевдо-3D даёт объёмных бойцов на плоской сетке, cel-шейдер делает модели «нарисованными», пикселизация и постеризация огрубляют кадр до вида 90-х, stepped-анимация возвращает «щёлкающее» движение спрайтов, а композитинг камеры позволяет применять эффекты избирательно. Каждая техника проста по отдельности; вместе они складываются в цельный узнаваемый образ. Теперь разберём **инженерный фундамент** — за счёт каких возможностей Unity всё это вообще работает.

Часть 3. За счёт чего это работает в Unity

Части 1 и 2 объясняли **что** и **как выглядит**. Эта часть — про **фундамент**: конкретные возможности движка Unity, на которых всё держится. Мы берём каждый механизм, объясняем его «с нуля» и показываем, **где именно** в нашем проекте он используется. Это связывает абстрактное знание Unity с живым кодом.

3.1. Жизненный цикл `MonoBehaviour`

3.1.1. Что это и зачем

`MonoBehaviour` — базовый класс любого скрипта-компонента. Его особенность: Unity **сама** вызывает у него специальные методы в определённые моменты («события жизненного цикла»). Вы не вызываете `Update` вручную — движок делает это каждый кадр. Это модель **inversion of control** (инверсия управления): не ваш код зовёт движок, а движок зовёт ваш код.

3.1.2. Ключевые события и где они в проекте

Метод	Когда вызывается	Пример в проекте
`Awake`	При создании объекта, до `Start`	`SteppedAnimator.Awake` — отключает `Animator`
`OnEnable`	При включении компонента	`RetroImageEffect.OnEnable` — создаёт материал
`Start`	Перед первым кадром	`PongGame.Start` — инициализация мяча/счёта
`Update`	**Каждый кадр**	`PongGame.Update`, `SteppedAnimator.Update`
`OnRenderImage`	После рендера камеры	`RetroImageEffect.OnRenderImage`
`OnDestroy`	При уничтожении	`UnitStackView.OnDestroy` — отписка от событий
`OnDisable`	При выключении	`UnitCameraCompositor.OnDisable` — освобождение RT
`OnGUI`	Каждый кадр GUI	`BattleController.OnGUI` — HUD
`OnDrawGizmos`	В Scene View (редактор)	`BattlefieldView.OnDrawGizmos` — гизмо сетки

Порядок важен. Например, `SteppedAnimator.Awake` отключает `Animator` **до** первого `Update`, чтобы Unity не успела обновить его автоматически. А `OnEnable` / `OnDisable` используются парами для захвата/освобождения ресурсов (материала, `RenderTexture`).

3.1.3. `Time.deltaTime` — движение, независимое от FPS

В `Update` время между кадрами непостоянно. Чтобы движение шло с одинаковой **скоростью** независимо от частоты кадров, всё умножается на `Time.deltaTime` (секунды с прошлого кадра):

```
// PongGame: мяч движется на "скорость × время", а не "скорость × кадр"
newPosition = _ball.position + (Vector3)(_ballDirection * _ballSpeed * Time.deltaTime);
// UnitStackView: модель скользит с учётом deltaTime
transform.localPosition = Vector3.MoveTowards(transform.localPosition, target, moveSpeed * Time.de
```

Без `deltaTime` игра на 120 FPS шла бы вдвое быстрее, чем на 60. Это одна из первых вещей, которую должен усвоить каждый Unity-разработчик, и проект применяет её везде, где что-то движется.

3.1.4. `[ExecuteAlways]` — жизнь в редакторе

Обычно `MonoBehaviour` «живёт» только в Play Mode. Атрибут `[ExecuteAlways]` заставляет его события работать **и в Edit Mode**. В проекте им помечены `BattlefieldView`, `RetroImageEffect`, `UnitCameraCompositor` — чтобы дизайнер видел поле и эффекты **без запуска игры**. Это ключ к инструментам «предпросмотра в редакторе».

Прочие полезные атрибуты в проекте: `[RequireComponent(typeof(Camera))]` (гарантирует наличие зависимого компонента), `[DisallowMultipleComponent]` (нельзя навесить дважды), `[AddComponentMenu( ... )]` (пункт в меню Add Component).

3.2. Корутины как машина времени игрового цикла

3.2.1. Проблема, которую решают корутины

Пошаговый бой растянут во времени: «активируй стек → жди клика игрока → двигай модель 1 секунду → жди конца анимации → нанеси урон». Написать это обычной функцией нельзя — функция выполняется целиком за один кадр и не умеет «ждать». А ставить `Thread.Sleep` смертельно — заморозится вся игра.

Корутина решает это: она умеет **приостановиться** и **продолжить в следующем кадре** (или после условия), не блокируя движок.

3.2.2. Как это устроено технически

Корутина — это метод, возвращающий `IEnumerator`, с инструкциями `yield return`:

```
private IEnumerator TakeTurn(UnitStack stack)  // возвращает IEnumerator
{
    // ... подготовка ...
    while (!actionChosen && !skip)
    {
        // проверить ввод
        yield return null;          // ← ПАУЗА до следующего кадра
    }
    stack.StartMove(path);
    yield return new WaitUntil(() ⇒ stack.State ≠ UnitStackState.Moving);  // ← ждать условие
}
```

Запускается через `StartCoroutine(TakeTurn(stack))`. Unity каждый кадр «дёргает» корутину до следующего `yield return`. Разные `yield` означают разное ожидание:

`yield return ...`	**Значение**
`null`	продолжить в следующем кадре
`new WaitUntil(() ⇒ cond)`	ждать, пока `cond` не станет `true`
`new WaitForSeconds(t)`	ждать `t` секунд
другая корутина	ждать её завершения

3.2.3. Вложенные корутины — как выражается весь бой

Гениальность подхода — во **вложенности**. `BattleRoutine` делает `yield return TakeTurn(stack)` — и ждёт завершения хода. `TakeTurn` делает `yield return AnimateMove()` через `WaitUntil`. Так многосекундный, многокадровый бой пишется **линейным** кодом, который читается сверху вниз, как обычная процедура:

```
// BattleRoutine — читается как простой сценарий, хотя тянется десятки секунд
while (BothSidesAlive())
{
    _round++;
    foreach (var stack in BuildTurnOrder())
        yield return TakeTurn(stack);    // каждый ход — полноценная подпрограмма
}
```

Без корутин пришлось бы вручную городить конечный автомат «в каком мы сейчас шаге боя», разбросанный по `Update`. Корутины сворачивают это в естественный последовательный код. Это — **главный** механизм, на котором держится весь движок ходов `BattleController`.

`UnitStackView` тоже весь на корутинах: `AnimateMove`, `AnimateAttack`, `Lerp` — каждая плавно меняет позицию по кадрам через `yield return null`.

3.3. `ScriptableObject`: данные, которые живут вне сцены

3.3.1. Суть

`ScriptableObject` (SO) — это объект-данные, сохраняемый как **ассет-файл** (`.asset`) в проекте, а не как часть сцены. Он сериализуется Unity, показывается и правится в инспекторе, но не привязан ни к какому `GameObject`.

В проекте на SO построен весь «контент»: `UnitDefinition` (существа), `ObstacleDefinition` (препятствия), `ObstacleDatabase` (каталог). Конкретные ассеты лежат в `Assets/Data/` (`Zombie.asset`, `Obstacle*.asset`, `ObstacleDatabase.asset`).

3.3.2. Зачем это лучше, чем `MonoBehaviour` на сцене

- **Единый источник истины.** Один `Zombie.asset` описывает зомби для всей игры. Правка в нём видна везде, где на него ссылаются. Не нужно дублировать цифры по сценам.
- **Разделение труда.** Дизайнер правит `.asset` в инспекторе, программист читает поля из кода — не мешая друг другу.
- **Не грузит сцену.** Данные не «висят» на объектах сцены, сцена остаётся лёгкой.
- **Переиспользование.** На один ассет ссылаются много `ArmySlot`, много боёв.

3.3.3. `[CreateAssetMenu]` и создание ассетов

Атрибут `[CreateAssetMenu(menuName = "Heroes/Battle/Unit", fileName = "Unit")]` добавляет пункт в меню `Assets → Create → Heroes → Battle → Unit`. Дизайнер правым кликом создаёт новую «карточку». Без кода. Это и есть **data-driven design** — игра управляется данными, которые редактируются визуально.

3.3.4. Сериализация: `[SerializeField]`, `[Serializable]`, атрибуты инспектора

Механизм сериализации Unity пронизывает проект:

- `[SerializeField] private ArmySlot[] slots` (в `Army`) — поле приватное (инкапсуляция), но Unity его сохраняет и показывает в инспекторе. Классический приём: «private для кода, видимый для редактора».
- `[Serializable] struct ArmySlot` / `class UnitViewSettings` — помечает **собственные** типы как сериализуемые, чтобы они отображались в инспекторе как вложенные блоки полей.
- **Атрибуты валидации/оформления:** `[Min(0)]`, `[Range(1,2)]`, `[Header("Stats")]`, `[Tooltip("...")]` — делают инспектор удобным и защищают данные от неверного ввода прямо в редакторе.

Понимание того, **что и как** Unity сериализует, критично: именно оно определяет, что дизайнер увидит и сможет настроить. Проект использует эти атрибуты вдумчиво и повсеместно.

3.4. Спрайты, `SpriteRenderer` и pixels-per-unit

3.4.1. Спрайт и его рендерер

`Sprite` — 2D-изображение (кусок текстуры) с метаданными (прямоугольник в атласе, пивот, PPU). `SpriteRenderer` — компонент, рисующий спрайт на сцене. В проекте `SpriteRenderer` используется для фона, гексов, препятствий, подсветок, маркеров.

3.4.2. Pixels Per Unit (PPU) — мост «пиксели ↔ мир»

Самое важное понятие 2D в Unity — **PPU**: сколько пикселей спрайта приходится на одну мировую единицу. При PPU = 64 спрайт шириной 64 px займёт ровно 1 юнит. Весь расчёт раскладки поля построен вокруг PPU:

```
// BattlefieldConfig: эталон
public const int ReferencePixelsPerUnit = 64;
// BattlefieldView.HexToWorld: пиксели → мир делением на PPU
return new Vector3(px / ppu, py / ppu, 0f);
```

Проект работает **в пиксельном пространстве** (шаги гексов 44/32/22 px), а в самом конце делит на PPU, получая мировые координаты. Это удобно: художник мыслит пикселями, а движок — юнитами, и PPU их связывает. `BattlefieldView` даже умеет **нормализовать** масштаб фона под целевой PPU независимо от того, с каким PPU импортирован спрайт (`normalizeBackgroundScale`).

3.4.3. `FilterMode` — как масштабируется текстура

Параметр фильтрации текстуры решает, как она выглядит при увеличении: - `FilterMode.Point` — без сглаживания, чёткие пиксели (нужен для пиксель-арта и пикселизации). - `FilterMode.Bilinear` — сглаживание (мягкая картинка).

`RetroImageEffect` явно ставит `lowRes.filterMode = FilterMode.Point` перед upscale-блитом — именно это делает пиксели крупными и резкими (см. 2.4.1). Без Point-фильтра пикселизация «размазалась» бы.

3.4.4. Создание спрайтов в коде: `Sprite.Create`

Спрайт не обязан быть импортированным ассетом. `Sprite.Create(texture, rect, pivot, ppu)` создаёт его прямо в коде. Проект генерирует так подсветки и маркеры из `Texture2D.whiteTexture` (см. 2.9). Здесь PPU в `Sprite.Create` задаёт **мировой размер** спрайта: `texture.width / 0.6f` подобрано так, чтобы квадрат подсветки был нужного размера относительно гекса.

3.5. Ортографическая камера и мировые координаты

3.5.1. Орто vs перспектива

Ортографическая камера проецирует сцену **без перспективы**: далёкие объекты не уменьшаются, параллельные линии остаются параллельными. Это стандарт для 2D-игр и идеально подходит нашему плоскому полю. (Перспективная камера, наоборот, даёт схождение линий и уменьшение с расстоянием — для 3D-миров.)

Именно ортокамера делает возможным трюк 2.5D (2.2): поле снимается «в лоб» без перспективных искажений, а объём создаётся только поворотами моделей.

3.5.2. Преобразования координат

Проект активно переводит координаты между тремя системами: **экран** (пиксели), **мир** (юниты), **локаль объекта**. Ключевые вызовы:

```
// BattleController: клик мыши (экран) → мир
Vector3 world = cam.ScreenToWorldPoint(Input.mousePosition);
// → локальные координаты поля (учитывает его Transform)
Vector3 local = battlefield.transform.InverseTransformPoint(new Vector3(world.x, world.y, 0f));
// PongGame: позиция мыши → мир для управления ракеткой
Vector3 mouseWorld = _camera.ScreenToWorldPoint(mouseScreen);
```

- `ScreenToWorldPoint` — из экранных пикселей в мировые координаты (работает благодаря ортокамере; для перспективной нужна ещё глубина `z`).
- `InverseTransformPoint` — из мировых координат в **локальные** координаты объекта. Нужен потому, что само поле может быть сдвинуто/повёрнуто через свой `Transform` — все гексы считаются относительно него.

Понимание этих трёх систем координат и переходов между ними — базовый навык. Проект показывает их «в бою»: перевод клика в гекс (1.9.6) и управление ракеткой в Pong.

3.6. Built-in Render Pipeline, `OnRenderImage` и `Graphics.Blit`

3.6.1. Какой конвейер и почему это важно

Проект использует **Built-in Render Pipeline (BiRP)** — «классический» конвейер Unity (не URP, не HDRP). Это видно по коду: пост-эффекты через `OnRenderImage`, шейдеры с тегом `LightMode = ForwardBase`, `#pragma multi_compile_fwdbase`, включения `UnityCG.cginc`. Всё это — идиомы **именно BiRP**. В URP пост-обработка и шейдеры устроены иначе (Volume, Render Features, Shader Graph/HLSL). Поэтому важно знать: **этот проект — под BiRP**.

3.6.2. `OnRenderImage` — перехват кадра

В BiRP пост-эффект — это метод `MonoBehaviour` на камере:

```
private void OnRenderImage(RenderTexture source, RenderTexture destination)
```

Unity вызывает его **после** того, как камера отрисовала сцену: `source` — готовый кадр, `destination` — куда положить результат (обычно экран). Внутри вы обрабатываете `source` и пишете в `destination`. Так работает `RetroImageEffect` (см. 1.12.2, 2.4).

3.6.3. `Graphics.Blit` — копирование с обработкой

`Graphics.Blit(src, dst, material)` копирует текстуру `src` в `dst`, прогоняя каждый пиксель через шейдер `material`. Это рабочая лошадка пост-обработки:

```
Graphics.Blit(source, lowRes);                  // просто уменьшить (без материала)
Graphics.Blit(lowRes, destination, _material);  // увеличить + прогнать через RetroPost
```

Проект строит **цепочку блитов**: downscale (пикселизация) → upscale через цветовой шейдер (постеризация). Чистая иллюстрация того, как из простого примитива «копируй с шейдером» собирается сложный эффект.

3.6.4. Анатомия шейдера в BiRP (на примере `HeroesFlat`)

Кастомный шейдер `Heroes/Flat` — хороший учебный образец шейдера BiRP:

```
Shader "Heroes/Flat"
{
    Properties { _Color( … ) _MainTex( … ) _Bands( … )  …  }   // параметры для инспектора/кода
    SubShader
    {
        Tags { "RenderType"="Opaque" "Queue"="Geometry" }       // как и когда рисовать
        Pass
        {
            Tags { "LightMode"="ForwardBase" }                  // только главный свет
            CGPROGRAM
            #pragma vertex vert                                  // вершинная функция
            #pragma fragment frag                                // фрагментная функция
            #pragma multi_compile_fwdbase
            #include "UnityCG.cginc"
            #include "Lighting.cginc"
            // struct appdata (вход) / v2f (вершина→фрагмент)
            // vert(): позиция в клип-пространство, нормаль в мир
            // frag(): освещение с квантованием в банды (см. 2.3)
            ENDCG
        }
    }
    Fallback "Diffuse"
}
```

Что здесь важно понять концептуально: - `Properties` — то, что видно в материале/инспекторе и что толкает код через `material.SetFloat( … )` (как `RetroImageEffect` толкает `_Levels`, `_Saturation`). - **Вершинный шейдер (`vert`)** — преобразует каждую вершину (позиция в клип-пространство, нормаль в мир, передача UV). - **Фрагментный шейдер (`frag`)** — считает цвет каждого пикселя. Именно здесь

живёт cel- логика (`floor(ndotl * bands)`). - `Fallback` — запасной шейдер, если основной не поддерживается железом.

Шейдеры `RetroPost` (пост-эффект, `vert_img` + `frag`, `ZTest Always`) и `ColorKeySprite` (спрайтовый, `SpriteVert` + маска по ключу) устроены по тем же принципам, но для своих задач. Понимать структуру `Shader { Properties / SubShader / Pass / CGPROGRAM }` — необходимо, чтобы читать любую графику проекта.

3.7. `RenderTexture` и закадровый рендеринг

3.7.1. Что это

`RenderTexture` **(RT)** — текстура, в которую можно **рендерить** (в отличие от обычной текстуры-картинки). Это «закадровый холст»: камера или блит пишут в RT, а потом её содержимое используется как обычную текстуру.

3.7.2. Два применения в проекте

1. **Изоляция пост-эффекта** (`UnitCameraCompositor`, см. 1.12.3, 2.6): камера юнитов рендерит **в свою RT** (`_camera.targetTexture = _rt`), а не в экран. Эффект отрабатывает на RT (только юниты), затем RT показывается полноэкранным `RawImage`.

```
csharp _rt = new RenderTexture(w, h, 24, RenderTextureFormat.ARGB32); _camera.targetTexture = _rt; // рисуем в RT, не в экран _camera.backgroundColor = new Color(0,0,0,0); // прозрачный фон _image.texture = _rt; // показываем RT поверх сцены
```

2. **Чтение пикселей без Readable-текстуры** (`ObstacleDefinitionEditor.ReadTexturePixels`, см. 1.13.3): чтобы прочитать пиксели спрайта, не включая «Read/Write» в импорте, код блитит текстуру в **временную RT**, делает её активной и читает через `ReadPixels`:

```
csharp var rt = RenderTexture.GetTemporary(w, h, 0, RenderTextureFormat.ARGB32, ...); Graphics.Blit(source, rt); RenderTexture.active = rt; readable.ReadPixels(new Rect(0,0,w,h), 0, 0); // копируем пиксели с GPU в CPU-текстуру
```

3.7.3. Управление ресурсами RT

RT — это память на GPU, её нужно **освобождать**. Проект делает это дисциплинированно: - `RenderTexture.GetTemporary` / `ReleaseTemporary` — берут/возвращают RT из пула (эффективно для кадровых эффектов). - `_rt.Release()` + `Destroy(_rt)` в `OnDisable` / `ReleaseTexture` — ручное освобождение постоянной RT компонента. - Пересоздание RT при смене разрешения экрана (`EnsureComposite` сравнивает размеры).

Правильная работа с RT — важный навык: забытая RT течёт памятью GPU. Проект показывает образцовое управление жизненным циклом ресурса (`OnEnable` создать → `OnDisable` освободить).

3.8. `Animator` и ручное управление им

3.8.1. Что делает `Animator`

`Animator` — компонент, проигрывающий анимации по графу состояний (Animator Controller). Обычно Unity сама обновляет его каждый кадр, интерполируя позы. Управляют им через **параметры**: `SetBool`,

`SetTrigger`, `SetFloat`.

3.8.2. Как проект управляет аниматором из логики состояний

`UnitStackView` переводит состояния FSM стека в параметры аниматора:

```
SetMoving(true);                        // _animator.SetBool("IsMoving", true) — начал идти
SetTrigger(_settings.attackTrigger);    // _animator.SetTrigger("Attack") — удар
SetTrigger(_settings.deathTrigger);     // _animator.SetTrigger("Die") — смерть
```

Имена параметров вынесены в `UnitViewSettings` (не захардкожены), и перед установкой view **проверяет их наличие** (`HasParameter`), чтобы не упасть на аниматоре без нужного параметра (см. 1.11.8). Это делает систему совместимой с разными готовыми аниматорами.

3.8.3. Ручное продвижение аниматора: `SteppedAnimator`

Самое нетривиальное — `SteppedAnimator` **перехватывает** управление аниматором:

```
_animator.enabled = false;      // Выключаем авто-обновление Unity
// ... в Update, по таймеру:
_animator.Update(step);         // сами продвигаем на фиксированный шаг
```

`_animator.enabled = false` останавливает автоматическое покадровое обновление, а `_animator.Update(deltaTime)` двигает анимацию **вручную** на заданный интервал. Это и даёт stop-motion (см. 1.12.1, 2.5). Важно: возможность вручную звать `Animator.Update` — не самая известная фича, но именно она позволяет тонко контролировать частоту анимации без правки клипов. Параметры (`SetBool` / `SetTrigger`) при этом не теряются — применяются на следующем ручном шаге.

3.9. События и делегаты C# как «нервная система» игры

3.9.1. Механизм

Делегат — это «указатель на метод»; **событие** (`event`) — безопасная обёртка над списком делегатов, на которую можно подписываться (`+=`) и отписываться (`-=`). Когда событие «поднимают» (`Invoke`), вызываются все подписчики.

Это стандартный **C#-механизм** (не Unity), но именно он держит связь «логика ↔ отображение» в проекте.

3.9.2. Как это связывает слои (без обратной зависимости)

`UnitStack` (ядро) объявляет события и поднимает их, **не зная** о подписчиках:

```
public event Action<UnitStack> StateChanged;
public event Action<UnitStack> HealthChanged;
// ...
private void SetState(UnitStackState next) { State = next; StateChanged?.Invoke(this); }
```

`UnitStackView` (отображение) подписывается и реагирует:

```
Stack.StateChanged += OnStateChanged;     // подписка
Stack.HealthChanged += OnHealthChanged;
```

Так достигается **инверсия зависимостей**: стек ничего не знает про view (можно подключить один view, десять, логгер в тесте — стеку всё равно), но view всегда узнаёт об изменениях. Это и есть техническая реализация принципа 1.1 «логика не зависит от отображения».

`?.Invoke` (null-conditional) защищает от вызова события без подписчиков. `Action<UnitStack>` — встроенный делегат «метод, принимающий `UnitStack`, без возврата».

3.9.3. Обязательная отписка

Критическое правило: **на что подписался — от того отпишись**. `UnitStackView` делает это в `OnDestroy` и при перепривязке в `Bind`:

```
private void OnDestroy()
{
    if (Stack ≠ null)
    {
        Stack.StateChanged -= OnStateChanged;
        Stack.HealthChanged -= OnHealthChanged;
    }
}
```

Без отписки: (а) стек держит ссылку на уничтоженный `MonoBehaviour` → утечка памяти; (б) попытка вызвать метод на уничтоженном объекте → исключение. Симметрия «подписался в `Bind` — отписался в `OnDestroy`» — признак аккуратного кода и частый источник багов у новичков, когда она нарушена.

3.10. Скриптинг редактора: `EditorWindow`, кастомные инспекторы, `Undo`, `EditorPrefs`

3.10.1. Отдельный мир — папка `Editor/`

Код в папках с именем `Editor` компилируется в **отдельную редакторную сборку** и **не попадает в билд игры**. Он использует пространство имён `UnityEditor` (недоступное в рантайме). Так устроены три инструмента проекта (1.13). Это разделение — фундаментальное: рантайм-код и редактор-код живут в разных сборках.

3.10.2. `EditorWindow` — своё окно инструмента

`BattlefieldEditorWindow` и `BattleTestWindow` наследуют `EditorWindow`:

```
[MenuItem("Tools/Heroes 3/Battle Test")]     // пункт меню
public static void ShowWindow() ⇒ GetWindow<BattleTestWindow>("Battle Test");

private void OnGUI() { /* рисуем поля, кнопки через EditorGUILayout */ }
```

- `[MenuItem( ... )]` добавляет пункт в главное меню Unity.
- `OnGUI` окна рисует интерфейс через `EditorGUILayout` (`ObjectField`, `IntField`, `Button`, `EnumPopup`, `HelpBox`).
- `GetWindow<T>` открывает/фокусирует окно.

Так создаются **кастомные инструменты пайплайна** — то, что превращает Unity из «движка» в «среду разработки под свою игру».

3.10.3. Кастомный инспектор — `[CustomEditor]`

`ObstacleDefinitionEditor` наследует `UnityEditor.Editor` с атрибутом `[CustomEditor(typeof(ObstacleDefinition))]` — он **заменяет** стандартный инспектор ассета:

```
public override void OnInspectorGUI()
{
    DrawDefaultInspector();                          // обычные поля
    if (GUILayout.Button("Auto-compute Blocked Hexes"))    // + своя кнопка
        AutoCompute(obstacle);
}
```

Так к обычному инспектору добавляется кнопка авторасчёта гексов (1.13.3). Кастомные инспекторы — способ дать дизайнеру специальные действия прямо в инспекторе ассета.

3.10.4. `Undo` — дружба с отменой

Инструмент, меняющий сцену/ассет, **обязан** поддерживать Ctrl+Z. Проект это делает:

```
Undo.RegisterFullObjectHierarchyUndo(_target.gameObject, "Generate Battlefield");  // до изменения
Undo.RecordObject(obstacle, "Auto-compute Blocked Hexes");                          // до правки а
Undo.RegisterCreatedObjectUndo(go, "Create Battle Controller");                     // созданный о
```

Регистрация состояния **до** изменения позволяет Unity откатить его. Игнорировать `Undo` — признак плохого инструмента: пользователь ждёт, что отмена работает везде.

3.10.5. `EditorPrefs` + `JsonUtility` — память между сессиями

Чтобы окна помнили настройки после перезапуска, проект хранит их в `EditorPrefs` (реестр редактора «ключ→строка»):

```
EditorPrefs.SetString(PrefKeyState, JsonUtility.ToJson(_state));   // сохранить состояние окна как
_state = JsonUtility.FromJson<WindowState>(EditorPrefs.GetString(PrefKeyState));  // восстановить
```

`JsonUtility` сериализует состояние окна в JSON. Ссылки на ассеты хранятся как **GUID** (стабильнее путей). Так `BattleTestWindow` восстанавливает заполненные армии, а `BattlefieldEditorWindow` — назначенные базу/префаб/материал. Плюс `MarkSceneDirty` помечает сцену изменённой (чтобы Unity предложила сохранить), а `DisplayDialog` показывает понятные предупреждения. Это всё — «хорошие манеры» редакторного инструмента.

3.11. `Gizmos` и отладочная визуализация

Gizmos — графика, видимая только в **Scene View** редактора (не в игре), для отладки и настройки. Рисуется в `OnDrawGizmos`:

```
// BattlefieldView.OnDrawGizmos
Gizmos.matrix = transform.localToWorldMatrix;                  // рисуем в локальных координатах п
Gizmos.DrawWireCube(GetGridBounds().center, GetGridBounds().size);  // рамка сетки
Gizmos.DrawSphere(HexToWorld(0, 0), 0.05f);                    // маркер угла
// + рамка безопасной зоны фона
```

Ценность: дизайнер **видит** границы сетки, угловые гексы и безопасную зону фона прямо в редакторе, ещё до запуска — и может точно подогнать раскладку (см. 1.10.5). `Gizmos.matrix` переключает систему координат, чтобы рисовать относительно объекта. Гизмо — недооценённый, но мощный инструмент: пара строк превращает невидимую математику раскладки в наглядные линии.

3.12. Текст: legacy `TextMesh` и TextMeshPro

Проект использует **два** способа рисовать текст, и это поучительно:

- **Legacy `TextMesh`** (`UnitStackView.BuildCountLabel`) — старый 3D-текст, живущий в мировом пространстве. Подпись численности стека висит над моделью как объект сцены:

`csharp countText = label.AddComponent<TextMesh>(); countText.font = Resources.GetBuiltinResource<Font>("LegacyRuntime.ttf"); // встроенный шрифт countText.characterSize = 0.02f; // мировой размер`

`TextMesh` прост, не требует Canvas и хорошо ложится на «мировую» подпись у модели. Встроенный шрифт (`Resources.GetBuiltinResource`) избавляет от ассета шрифта.

- **TextMeshPro (TMP)** (`PongGame`) — современная система качественного текста с чётким рендерингом на любом масштабе (SDF-шрифты):

`csharp [SerializeField] private TMP_Text _scoreText; _scoreText.text = $"{_player1Score} : {_player2Score}";`

Папка `Assets/TextMesh Pro/` — это ресурсы TMP (шрифты, спрайты эмодзи, настройки). TMP — рекомендуемый способ для «настоящего» UI: он резкий, гибкий, поддерживает богатое форматирование.

Выбор между ними — по задаче: `TextMesh` для простой мировой подписи у объекта, TMP — для качественного экранного UI. Проект осознанно применяет оба.

Итог Части 3. Мы прошли по инженерному фундаменту: жизненный цикл `MonoBehaviour` (когда движок зовёт наш код), корутины (как растянуть бой во времени), `ScriptableObject` (данные вне сцены), спрайты и PPU (мост пиксели ↔ мир), ортокамера и преобразования координат, BiRP с `OnRenderImage` / `Graphics.Blit` (пост-обработка), `RenderTexture` (закадровый рендер), `Animator` и его ручное управление, события C# (связь слоёв без обратной зависимости), скриптинг редактора (инструменты дизайнера) и гизмо (отладочная визуализация). Каждый механизм мы привязали к конкретному месту в коде — теперь понятно не только «что делает Unity», но и «как проект этим пользуется».

Приложения

Приложение A. Полный словарь терминов

AABB (Axis-Aligned Bounding Box) — прямоугольник/параллелепипед, выровненный по осям. Простейший тест столкновений: два AABB пересекаются, если перекрываются по каждой оси. Используется в `PongGame.Intersects`.

Ambient (ambient light) — рассеянный «фоновый» свет, равномерно освещающий всё. В `HeroesFlat` `_AmbientBoost` не даёт теням стать чёрными.

BFS (Breadth-First Search, поиск в ширину) — обход графа «волнами» от старта; находит кратчайшие пути при равных стоимостях рёбер. `BattleGrid.ComputeReachable`.

BiRP (Built-in Render Pipeline) — классический конвейер рендеринга Unity. Проект написан под него (`OnRenderImage`, `ForwardBase`, `UnityCG.cginc`).

Cel-shading (тунарное затенение) — освещение с резкими плоскими зонами света вместо градиента; «рисованный» вид. `HeroesFlat.shader`.

Chroma key (ключевой цвет) — приём вырезания фона по цвету («зелёный экран»). `ColorKeySprite.shader`, а также cyan-ключ в `ObstacleDefinitionEditor`.

Coroutine (корутина) — метод, умеющий приостанавливаться (`yield return`) и продолжаться в следующем кадре. Основа движка ходов и анимаций.

Delegate / event (делегат / событие) — «указатель на метод» / список делегатов с подпиской. Связь `UnitStack ⇔ UnitStackView`.

Determinism (детерминизм) — свойство «одинаковый вход → одинаковый выход». Обеспечено через `System.Random(seed)` в генераторе.

FSM (Finite-State Machine, конечный автомат) — модель с набором взаимоисключающих состояний и переходами. `UnitStack`.

Gizmos — отладочная графика в Scene View. `BattlefieldView.OnDrawGizmos`.

Hex / odd-r offset — шестиугольная клетка / раскладка, где нечётные ряды сдвинуты вправо. `BattleGrid`, `HexToWorld`.

IMGUI (`OnGUI`) — «немедленный» режим GUI, рисуется каждый кадр. HUD в `BattleController`.

Lerp (линейная интерполяция) — плавный переход между двумя значениями по параметру 0..1. `Vector3.Lerp` / `MoveTowards` в анимациях.

MonoBehaviour — базовый класс компонента-скрипта; движок вызывает его события жизненного цикла.

PPU (Pixels Per Unit) — сколько пикселей спрайта на 1 мировую единицу. Мост «пиксели↔мир».

Posterize (постеризация) — сокращение числа уровней цвета на канал → ограниченная палитра. `RetroPost.shader`.

Quaternion (кватернион) — способ задать 3D-поворот без gimbal lock. `BattleProjection`.

RenderTexture (RT) — текстура, в которую можно рендерить (закадровый холст). `UnitCameraCompositor`.

ScriptableObject (SO) — объект-данные как ассет-файл вне сцены. `UnitDefinition`, `ObstacleDefinition`, `ObstacleDatabase`.

Seed (семя) — число инициализации ГПСЧ; определяет всю «случайную» последовательность.

Sorting order / render queue — механизмы порядка отрисовки (для спрайтов / для материалов).

Stepped animation (stop-motion) — анимация с низким фиксированным FPS; «щёлкающие» позы. `SteppedAnimator`.

`Time.deltaTime` — секунды с прошлого кадра; делает движение независимым от FPS.

Weighted random (взвешенная случайность) — выбор с вероятностью пропорционально весу. `BattlefieldGenerator.PickWeighted`.

Приложение B. Упражнения и задания для самостоятельной работы

Задания идут от простого к сложному. Для большинства достаточно правок в существующих классах.

Уровень 1 — разминка (данные и настройки): 1. Создайте нового юнита (`Assets → Create → Heroes → Battle → Unit`) — «Скелет» с иными характеристиками. Через `Battle Test` устройте бой зомби против скелетов. 2. Измените в `UnitViewSettings` угол `viewPitchDegrees` и `animationFps`. Понаблюдайте, как меняются поворот моделей и «щёлканье» анимации. 3. В `RetroImageEffect` покрутите `targetHeight` и `colorLevels`. Найдите настройки, наиболее похожие на скриншот Н3.

Уровень 2 — чтение и понимание кода: 4. Проследите по коду полный путь одного клика по врагу: от `TryGetHexUnderMouse` до `ApplyDamage`. Выпишите цепочку вызовов и участвующие классы. 5. Объясните своими словами, почему `UnitStack.CompleteMove` меняет `(X,Y)` в конце движения, а не в начале. Что сломалось бы при обратном порядке? 6. Найдите все места, где используется `Application.isPlaying`, и объясните, зачем каждая проверка.

Уровень 3 — небольшие доработки: 7. Добавьте в `UnitDefinition` поле «дальнобойность» (`bool isRanged`) и разрешите таким существам атаковать врага **без** перемещения вплотную (правьте `ComputeAttackable`). 8. Реализуйте простейший ИИ для защитника: в `TakeTurn`, если активен defender, выбирать ближайшего атакуемого врага автоматически (вместо ожидания клика). 9. Добавьте счётчик потерь: выводите в HUD, сколько существ потеряла каждая сторона.

Уровень 4 — серьёзные расширения: 10. Добавьте «мораль/удачу»: с некоторой вероятностью урон удваивается (удача) или стек пропускает ход (плохая мораль). Не забудьте про детерминизм — используйте локальный `System.Random`, засеянный от seed боя. 11. Реализуйте контратаку: после атаки в ближнем бою цель, если жива, бьёт в ответ. 12. Напишите **юнит-тесты** для `BattleGrid` (проверьте `ComputeReachable` на поле с препятствиями) и `BattlefieldGenerator` (проверьте детерминизм: два вызова с одним seed дают идентичные конфиги). Это покажет ценность «чистого ядра».

Уровень 5 — визуализация: 13. Напишите свой пост-эффект (например, виньетку или скан-линии CRT) как второй шейдер в цепочке `RetroImageEffect`. 14. Сделайте полосу здоровья над стеком (мировой спрайт, ширина пропорциональна `TotalHealth / (MaxUnitHealth × стартовый Count)`), обновляемую по событию `HealthChanged`.

Приложение C. Частые ошибки и как их избежать

Собрано на основе приёмов, которые проект применяет **правильно** — учитесь на них.

1. **Забыли отписаться от события.** Подписались `+=`, но не сделали `-=` в `OnDestroy` → утечка памяти и исключения на уничтоженных объектах. Проект всегда отписывается (`UnitStackView.OnDestroy`). *Правило: подписка и отписка — симметричны.*
2. **Движение без `Time.deltaTime`.** Скорость станет зависеть от FPS. Всегда умножайте на `Time.deltaTime` (см. `PongGame`, `UnitStackView`).
3. **`UnityEngine.Random` там, где нужен детерминизм.** Глобальное состояние делает генерацию невоспроизводимой. Для детерминированных систем — локальный `System.Random(seed)` (см. `BattlefieldGenerator`).
4. **Логика знает про отображение.** Если «мозг» игры начинает дёргать `Instantiate` и двигать `Transform`, он становится нетестируемым и хрупким. Держите правила в чистых классах, а отображение — отдельно, связывая событиями (принцип 1.1).
5. **Не освободили `RenderTexture`.** RT течёт памятью GPU. Освобождайте в `OnDisable` (`Release` + `Destroy`) и используйте `GetTemporary` / `ReleaseTemporary` для кадровых (см. `UnitCameraCompositor`, `RetroImageEffect`).
6. **`Destroy` в Edit Mode.** В редакторе нужно `DestroyImmediate`, а в игре — `Destroy`. Проект инкапсулирует это в хелперах `ClearChildren` / `DestroyComponent`, проверяя `Application.isPlaying`.
7. **Захардкоженные имена параметров аниматора.** Если аниматор не имеет параметра `Attack`, `SetTrigger` молча промахнётся или бросит ошибку. Проект проверяет наличие (`HasParameter`) и выносит имена в настройки.
8. **Инструмент редактора без `Undo`.** Пользователь ждёт, что Ctrl+Z работает. Регистрируйте изменения в `Undo` до их внесения (см. редакторные окна).
9. **Неполные данные роняют игру.** Пустой префаб, пустой список терраинов, `null`-ссылка — всё это встречается в реальных проектах. Проект защищается: капсула-заглушка при отсутствии префаба, проверки на `null`/пустоту в `ObstacleDatabase`, безопасные дефолты в свойствах `UnitStack`.
10. **Рассинхрон логики и геометрии.** Если раскладка гексов в `HexToWorld` и таблица соседей в `BattleGrid` разойдутся, подсветка «уедет». Держите геометрию в одном согласованном месте (odd-r в обоих).

Приложение D. Что можно улучшить и куда развивать проект

Проект — крепкая учебная основа. Возможные направления роста (полезно обсудить на семинаре):

- **Больше контента.** Сейчас есть один юнит (Зомби) и 8 препятствий. Добавить расы, стрелков, летающих (игнорируют препятствия при движении), магию.
- **ИИ противника.** Полноценный бот: оценка целей, приоритет добивания, отступление слабых стеков. Хорошая задача на алгоритмы (минимакс/эвристики).
- **Юнит-тесты ядра.** «Чистое ядро» напрашивается на тесты (`BattleGrid`, `UnitStack`, `BattlefieldGenerator`). Это и защита от регрессий, и демонстрация ценности архитектуры.

- **Сетевой режим.** Детерминированная генерация уже готова к этому: клиенты могут строить одинаковое поле из общего seed. Дальше — синхронизация ходов (lockstep).
- **Дублирование карты занятости.** `BuildObstacleMap` (контроллер) и `ComputeOccupancy` (view) почти совпадают. Можно вынести общий метод в чистый класс — упражнение на рефакторинг.
- **Переход на URP (осторожно).** Если понадобится современный конвейер, пост-эффекты и шейдеры придётся переписать (Volume/Render Features, Shader Graph). Хорошее упражнение на понимание различий BiRP/URP — но большой объём работы.
- **Данные боя наружу.** `BattlefieldConfig` уже чистые данные; можно добавить сериализацию всего боя (сейв/реплей), раз логика отделена от сцены.

Заключение

Мы разобрали проект целиком: **каждый** скрипт (Часть 1), **все** техники визуализации (Часть 2) и **инженерный фундамент** Unity под ними (Часть 3). Если из этого пособия нужно вынести одну мысль, то вот она:

> **Хорошая архитектура — это разделение ответственности.** Правила игры (чистое ядро) не знают про экран; отображение (Unity-слой) реагирует на правила через события; инструменты (редактор) существуют отдельно и не попадают в билд. Это разделение делает код тестируемым, детерминированным, читаемым и гибким. А красивый ретро-вид достигается не одной «магией», а стеком простых, хорошо согласованных техник.

Именно сочетание **чистой инженерии** и **продуманной визуальной стилизации** делает этот небольшой проект отличным учебным материалом. Изучайте его код не как набор рецептов, а как пример **мышления**: почему разделили слои, почему выбрали `System.Random`, почему позиция 2D, а поворот 3D. Ответы на эти «почему» — и есть настоящее инженерное образование.

Успехов в изучении и разработке!

Пособие подготовлено на основе анализа исходного кода проекта Heroes-3 Battle. Все примеры кода взяты из реальных файлов проекта и при необходимости сокращены для наглядности — полные версии смотрите в `Assets/Scripts/`.