Тестирование, дебаг, оптимизация

Инструменты, методологии, узкие места производительности

Дебаг в Unity

Процесс поиска и устранения логических, функциональных и производственных ошибок (багов).

1. Найти причину (а не только симптом)
2. Определить точку в коде, где всё ломается
3. Проверить эффективность и корректность исправления

Дебаг в Unity

Воспроизведение → локализация → исправление → регресс-тест

Дебаг в Unity

Прежде чем исправлять — нужно гарантированно воспроизвести баг.

- Шаги: устройство, версия Unity, сцена, действия игрока, скриншот/видео.
- Пример: *“Если в меню нажать кнопку дважды — игра зависает”*

Дебаг в Unity

Сужаем контекст: убираем все лишнее, чтобы осталась только проблемная часть.

- Отключаем системы и скрипты по очереди (AI, UI, Audio).
- Проверяем зависимость: “падает ли без этого компонента?”
- Создаём минимальный воспроизводимый пример (Minimal Repro Scene)

Дебаг в Unity

После того как нашли место проблемы:

1. Выдвигаем гипотезу (“скорее всего, NullReference происходит из-за Destroy() в Update”).
2. Добавляем Debug.Log() или breakpoint.
3. Проверяем результат.

Дебаг, инструменты

1. Debug.Log / LogWarning / LogError — логирование (контекст, теги, stacktrace)
2. Unity Profiler — CPU, GPU, Rendering, Memory, GC, Audio, VSync.
3. Editor Debugger — через Visual Studio / Rider, breakpoints, watch, step over/into.
4. Memory Profiler package — снимки памяти, анализ утечек, retained objects.
5. Frame Debugger — пошаговый обзор рендеринга (draw calls, state changes).
6. Device logs / adb logcat — для Android
7. Xcode device logs — для iOS.
8. Crash reporting: Unity Cloud Diagnostics / Sentry / Crashlytics, Backtrace.

Дебаг, инструменты

Debug.Log, Debug.LogWarning, Debug.LogError, самый базовый инструмент.

- Debug.Log("Score: " + score);
- Debug.LogWarning("Player health is low!")
- Debug.LogError("Null reference in EnemyController")
- Debug.Break()
- Debug.Assert()

Можно добавлять контекст и теги:

```
Debug.Log($"[AI] Enemy {enemyId} took {damage} dmg", this);
```

Сделать свой враппер с необходимыми базовыми опциями

Дебаг, инструменты

```
/// <summary>
/// Log info
/// </summary>
/// <param name="data">Data to log</param>
/// <param name="label">Log label</param>
/// <param name="color">Log color</param>
Ссылок: 99+
public static void Info(object data, string label, LogColor color = LogColor.Default) {
    Debug.LogFormat(format: GetMask(label), args: DateTime.Now.ToString(format: DATE_FORMAT), GetColor(color), label, data);
}
```

Дебаг, инструменты

IDE Debugger — интерактивная отладка.

Подключается к Unity Editor, ставим breakpoint → игра останавливается в момент выполнения.

Можно:

- смотреть значения переменных;
- менять их на лету;
- выполнять код построчно (Step Over / Into / Out).

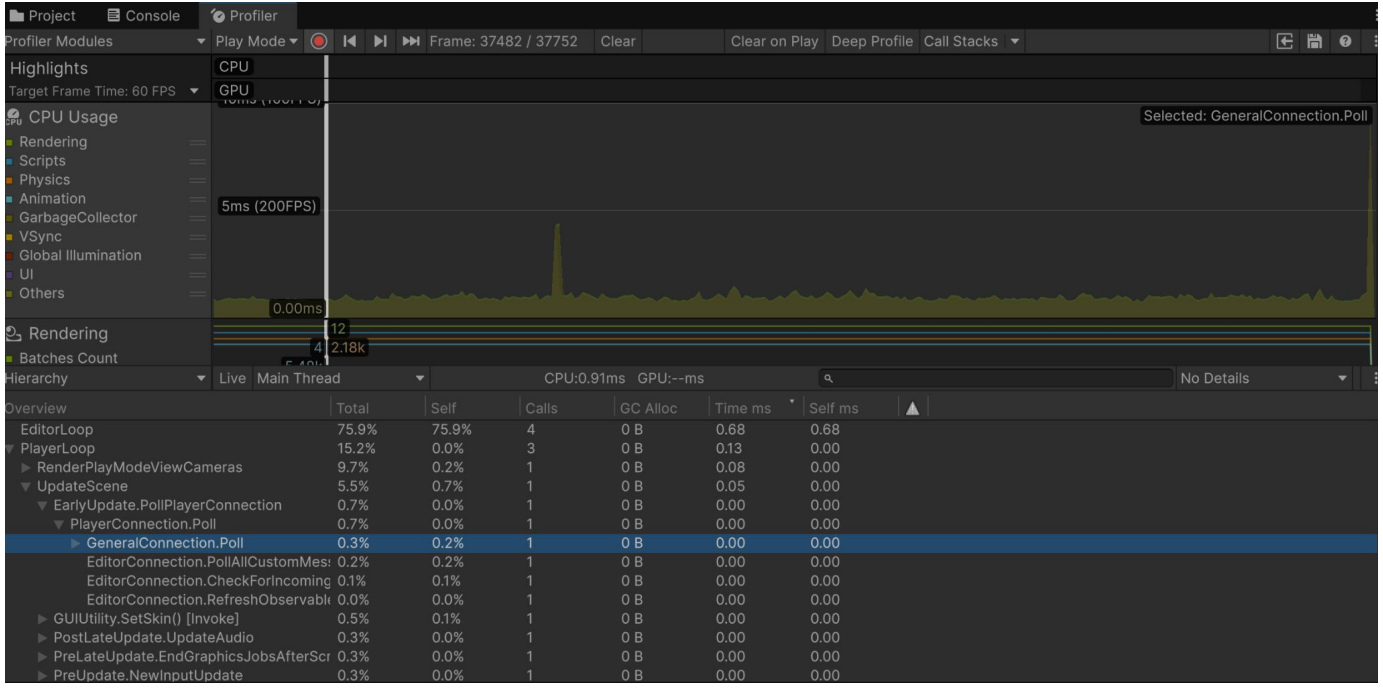
Демо

Дебаг, инструменты

Unity Profiler — инструмент анализа производительности

- Показывает: CPU, GPU, Rendering, Memory, Audio, Physics, GC Alloc, Timeline.
- Поддерживает Remote Profiling (профилировать сборку на телефоне по Wi-Fi или USB).
- Можно включить Deep Profiling (замедляет игру, но показывает все вызовы).

Дебаг, инструменты



Демо

Дебаг, инструменты

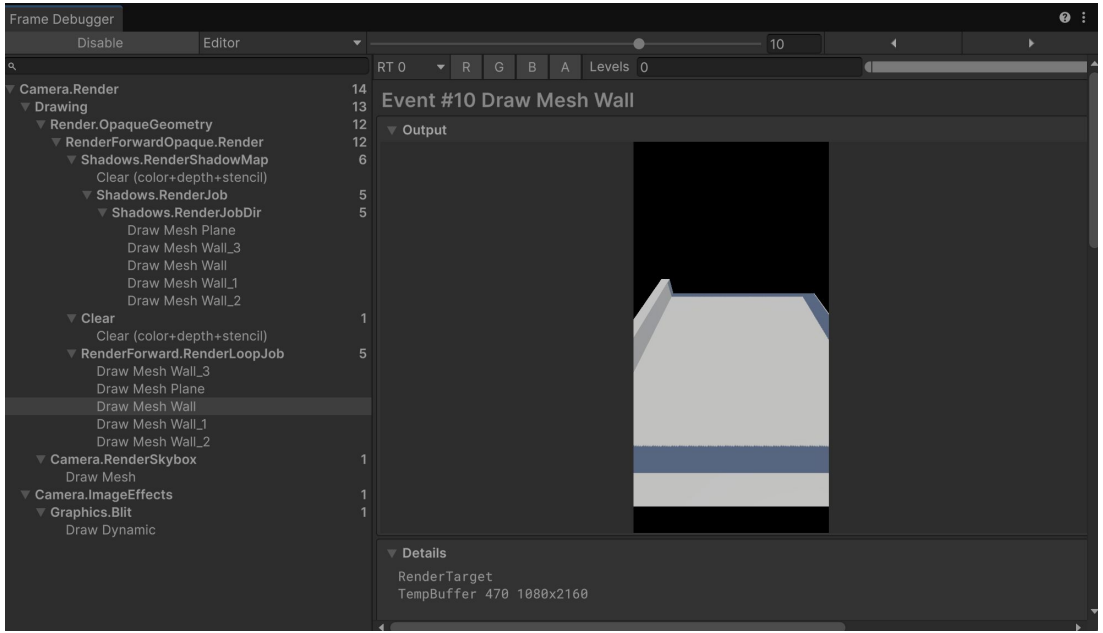
Frame Debugger — пошаговый разбор рендеринга кадра

Позволяет увидеть, какие объекты рисуются, в каком порядке, сколько draw calls.

Используется при:

- оптимизации рендера (снижение draw calls);
- поиске багов с прозрачностью, наложением UI;
- анализе overdraw и неправильных материалов.

Дебаг, инструменты



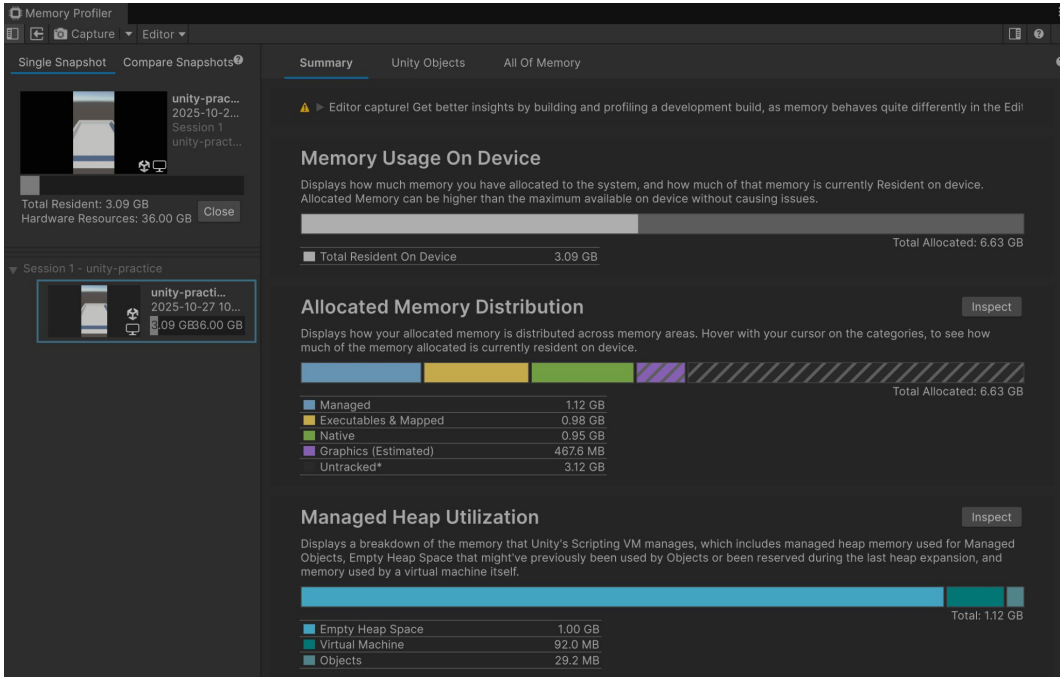
Демо

Дебаг, инструменты

Memory Profiler — инструмент для анализа утечек памяти

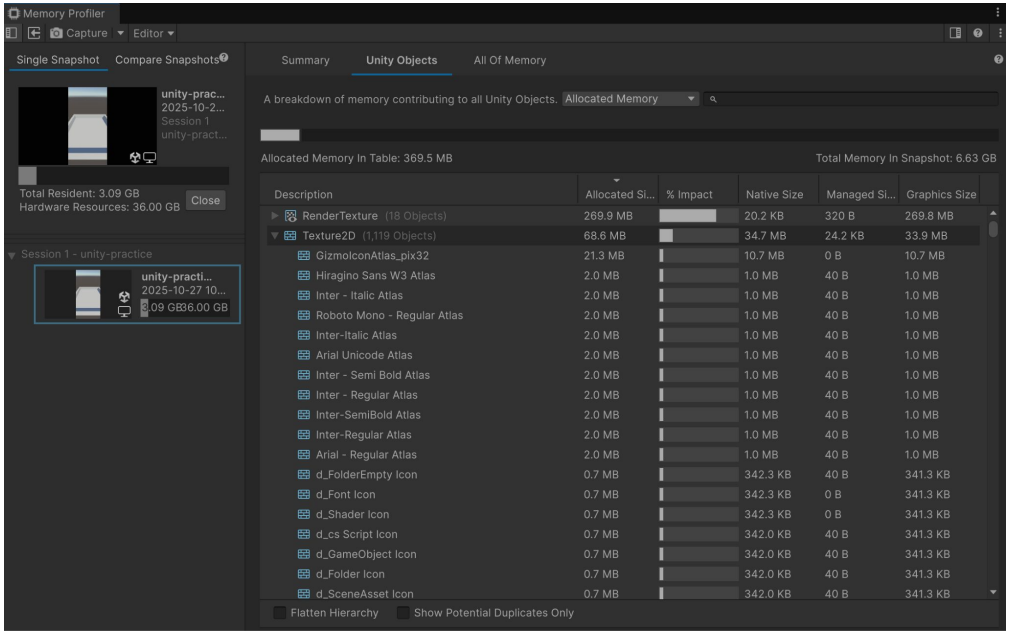
- Снимает “снимки” памяти (snapshots) и сравнивает.
- Показывает: кто удерживает объект в памяти, граф зависимостей (retained objects).
- Можно сравнить: “до загрузки уровня” и “после выгрузки”.

Дебаг, инструменты



Демо

Дебаг, инструменты



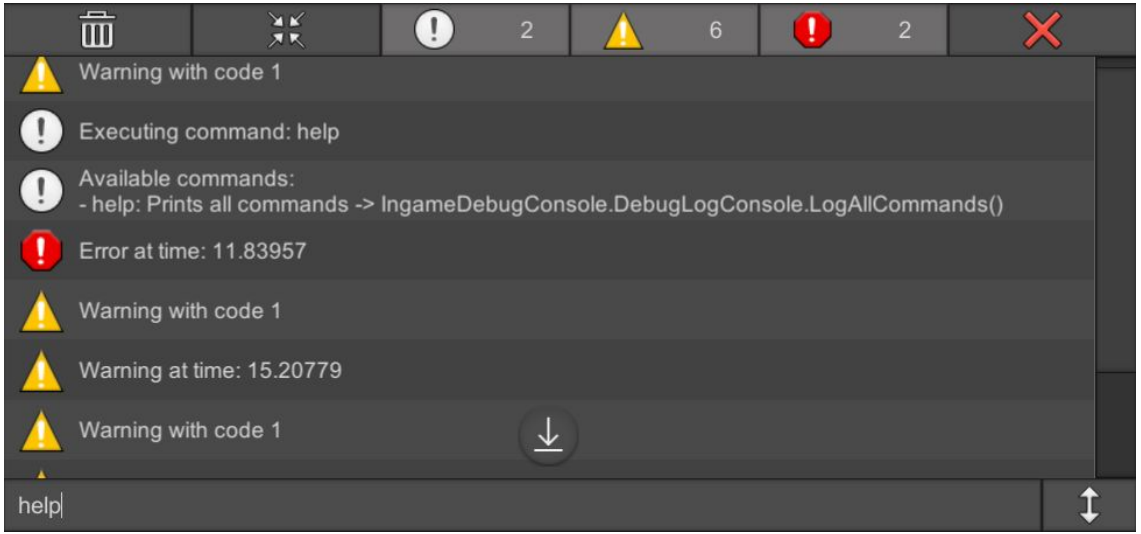
Дебаг, инструменты

Удобно отлаживать сразу на устройстве:

- Android: adb logcat -s Unity ActivityManager
- iOS: Xcode → Devices → View Device Logs

Дебаг, инструменты

https://github.com/yasirkula/UnityIngameDebugConsole



Дебаг, инструменты

Атрибут [Conditional] для логов, чтобы они не попадали в билд

```
[Conditional("DEBUG")]
public static void Log(object msg) => Debug.Log(msg);
```

Отрисовка дебаг-информации прямо в сцене:

```
void OnDrawGizmos() {
    Gizmos.color = Color.red;
    Gizmos.DrawLine(transform.position, target.position);
}
```

Дебаг, инструменты

При использовании атрибут [Conditional] компилятор удаляет сам вызов метода из кода, если символ не определён. Но метод (код) остается в сборке. Если надо чтобы в билд не попал и сам метод, то нужно обернуть весь метод в директиву препроцессора #if / #endif.

```
#if DEBUG
void LogPlayerStats(Player player)
{
    Debug.Log($"Player HP: {player.HP}");
}
#endif
```

Тестирование

Процесс проверки того, что программа работает корректно, устойчиво и соответствует требованиям.

Тестирование

- Unit-тесты
- Integration-тесты
- Functional / Gameplay-тесты
- Regression-тесты
- Performance-тесты
- UI-тесты

Unit-тесты (модульные)

Проверка корректности работы отдельных функций, классов или методов в изоляции от остальной игры.

```
[Test]
public void DamageCalculation_WorksCorrectly()
{
    var player = new Player(100);
    player.TakeDamage(30);
    Assert.AreEqual(70, player.Health);
}
```

- Проверка математической логики (урон, очки, прокачка)
- Проверка утилитарных классов (системы сохранений, сериализация, конвертация данных)
- Проверка поведения без запуска сцены.

Integration-тесты (интеграционные)

Проверка как несколько модулей взаимодействуют между собой.

```
[UnityTest]
public IEnumerator Inventory_SavesAndLoadsItems()
{
    var inventory = new Inventory();
    inventory.Add("Sword");
    SaveSystem.Save(inventory);
    yield return null;

    var loaded = SaveSystem.Load<Inventory>();
    Assert.AreEqual("Sword", loaded.Items[0]);
}
```

- Проверка связки систем (инвентарь + сохранение + UI)
- Проверка взаимодействия менеджеров (AudioManager + GameManager)
- Проверка сетевого обмена данными.

Functional / Gameplay-тесты (функциональные)

Проверка что игровой функционал работает так, как задумано.

```
[UnityTest]
public IEnumerator Enemy_AttacksPlayerWhenClose()
{
    var enemy = Object.Instantiate(Resources.Load<Enemy>("Enemy"));
    var player = Object.Instantiate(Resources.Load<Player>("Player"));

    player.transform.position = enemy.transform.position + Vector3.forward * 1f;
    yield return new WaitForSeconds(1f);

    Assert.IsTrue(enemy.HasAttacked);
}
```

- Проверка игровых механик.
- Проверка логики событий (trigger, collision, spawn, UI-реакции)

Regression-тесты (регрессионные)

Убедиться, что новые изменения не сломали уже работающий функционал.

- Автоматически прогоняют старые тесты после каждого обновления кода
- Используются CI/CD (GitHub Actions, GitLab, Jenkins)
- Позволяют быстро проверить, не сломал ли новый код старую механику

Пример: после добавления новой системы урона убедиться, что старая система сохранений всё ещё работает.

Unit / Integration тесты + CI-пайплайн, Husky, Unity Cloud Build, GitHub Actions, TeamCity.

Performance-тесты (производительности)

Проверка скорости, производительности, памяти, FPS и других метрик.

```
[Test, Performance]
public void EnemySpawner_Performance()
{
    Measure.Method(() => {
        EnemySpawner.Spawn(1000);
    })
    .WarmupCount(5)
    .MeasurementCount(10)
    .Run();
}
```

- Unity Performance Testing Extension
- Unity Profiler
- Frame Debugger
- Deep Profiling Mod
- Custom FPS logger

https://docs.unity3d.com/Packages/com.unity.test-framework.performance@1.0/manual/index.html

UI-тесты (интерфейса)

Проверка что UI работает и реагирует на действия пользователя корректно.

```
[UnityTest]
public IEnumerator StartButton_OnlyEnabledAfterDataLoaded()
{
    var ui = Object.Instantiate(Resources.Load<MainMenu>("MainMenu"));
    yield return new WaitUntil(() => ui.IsDataLoaded);
    Assert.IsTrue(ui.StartButton.interactable);
}
```

- Unity Test Framework + UI Toolkit Testing Tools
- UnityEngine.EventSystems + имитация нажатий
- Автоматизация через InputTestFixture.

Smoke-тесты / Build-тесты

Проверить, что игра вообще запускается и не падает после сборки.

Примеры:

- Проверка успешной сборки для Android / iOS.
- Проверка запуска первой сцены.

Инструменты:

- CI/CD (Unity Cloud Build).
- Автоматический запуск после сборки.

User / Playtesting (игровое)

Понять, как игроки реально воспринимают игру.

Виды:

- Ручное тестирование (QA-команда).
- Игровое наблюдение (play sessions).
- Метрики (Unity Analytics, GameAnalytics, Firebase, AppMetrica, custom events).

Пример:

- Анализировать, где игрок чаще умирает.
- Проверить, интуитивен ли интерфейс.

Тестирование

Вид теста	Автоматизируется	Где выполняется	Проверяет	Пример
Unit	✅ Да	В редакторе (Edit Mode)	Отдельный класс/метод	Damage, MathUtils
Integration	✅ Да	В игре (Play Mode)	Взаимодействие систем	Inventory + SaveSystem
Functional	✅ Да	В игре	Поведение механик	Сбор монет
Regression	✅ Да	CI/CD	Стабильность старого кода	Все старые тесты
Performance	⚙️ Частично	В игре / Profiler	FPS, память	1000 врагов
UI	✅ Да	В игре	Интерфейс, кнопки	Главное меню
Smoke	✅ Да	CI	Запуск билда	Загрузка сцены
User / Playtesting	❌ Нет	Игроки	UX, фидбек	Телеметрия, опросы

Test vs UnityTest

[Test] — обычный синхронный тест (NUnit)

Атрибут идёт из NUnit и работает точно так же, как в классическом C#.

- Не может использовать yield return
- Работает в одном кадре
- Подходит для чистой логики, без зависимостей на Unity Engine (Transform, GameObject и т.д.)

Test vs UnityTest

```
using NUnit.Framework;

public class MathUtilsTests
{
    [Test]
    public void Add_TwoNumbers_ReturnsSum()
    {
        int result = 2 + 3;
        Assert.AreEqual(5, result);
    }
}
```

Test vs UnityTest

[UnityTest] — тест, который работает по кадрам, расширение NUnit, добавленное Unity

- Возвращает IEnumerator
- Можно использовать yield return null, WaitForSeconds, WaitUntil, WaitForFixedUpdate, и т.п.
- Позволяет тестировать объекты сцены, поведение во времени, анимации, загрузку и т. д.

Test vs UnityTest

```
using UnityEngine;
using UnityEngine.TestTools;
using NUnit.Framework;
using System.Collections;

public class PlayerMovementTests
{
    [UnityTest]
    public IEnumerator Player_MovesForward_WhenInputIsPressed()
    {
        var player = new GameObject().AddComponent<PlayerMovement>();
        player.Move(Vector3.forward);


        yield return null;

        Assert.Greater(player.transform.position.z, 0);
    }
}
```

- Тест “ждёт” один кадр (yield return null), чтобы Unity успела обработать Update()
- Проверяется состояние объекта после обновления сцены
- Такой тест нельзя сделать через [Test], потому что обычный тест выполняется мгновенно и не имеет доступа к игровому циклу Unity

Test vs UnityTest

- [Test] запускается в одном кадре, вне контекста игрового цикла Unity.
  Это “чистый” NUnit-тест

- [UnityTest] запускается в игровом цикле, Unity создаёт скрытую сцену и обрабатывает yield return шаги, как корутину

- [UnityTest] можно использовать только с UnityEngine API, потому что он выполняется в Play Mode

Именование тестов

В тестах главное — понятность и выразительность, а не строгое следование правилам именования производственного кода.

Поэтому многие команды используют “описательные” имена тестов, которые читаются как фразы.

Именование тестов

```
[Test]
public void AddScore_WhenPlayerScoresPoints_ShouldIncreaseTotalScore()
```

```
[Test]
public void AddScore_When_Player_Scores_Points_Should_Increase_TotalScore()
```

Читается почти как предложение:

“AddScore — When player scores points — Should increase total score”

Именование тестов

Зачем?

1) Подчёркивания визуально отделяют логические части имени, превращая его в читаемую фразу.
2) Распространённая структура теста (Behavior-Driven Development, BDD): Given (условие) — When (действие) — Then (ожидание).
3) В Test Runner или CI-логах тест с длинным описанием проще понять.
4) При экспорте результатов тестов имена без пробелов, но остаются понятными для человека.

Именование тестов

Given_When_Then

Самый распространённый, особенно при TDD/BDD.

```
[Test]
public void PlayerDies_When_Health_ReachesZero()
```

Именование тестов

MethodName_State_ExpectedResult

Формат, часто используемый в NUnit / Unity Test Framework.

```
[Test]
public void AddScore_WithPositiveValue_IncreasesTotalScore()
```

Именование тестов

Should_ExpectedBehavior_When_State

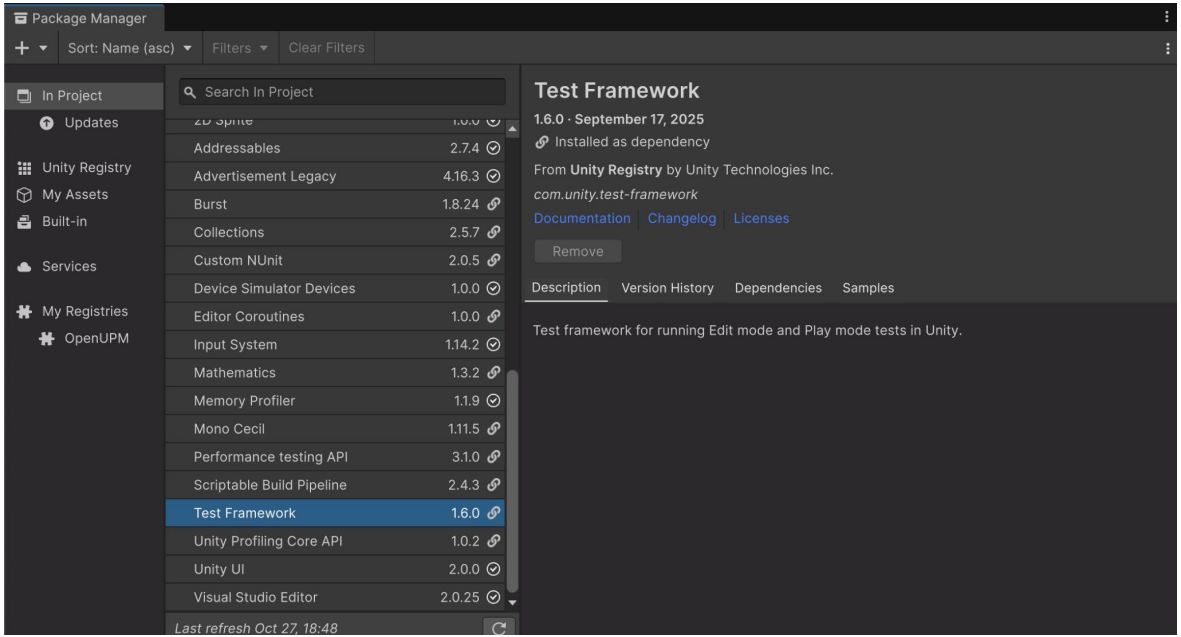
Более “говорящий” стиль, ближе к человеческому языку.

```
[Test]
public void Should_Increase_Score_When_Player_Scores()
```

Именование тестов

Главное — **консистентность**: придерживайся одного стиля во всём проекте.

Как запускать тесты



Test Framework

После установки появится окно Test Runner:

Window → General → Test Runner

Как запускать тесты

EditMode — тесты, выполняются в редакторе без запуска сцены. Используются для проверки чистой логики, не зависящей от Unity API.

PlayMode — тесты, выполняются в режиме игры (play). Используются для проверки объектов, поведения компонентов, UI и т.д.

```
Assets/
├── Scripts/
└── Tests/
    ├── EditMode/
    │   └── MyEditModeTests.cs
    └── PlayMode/
        └── MyPlayModeTests.cs
```

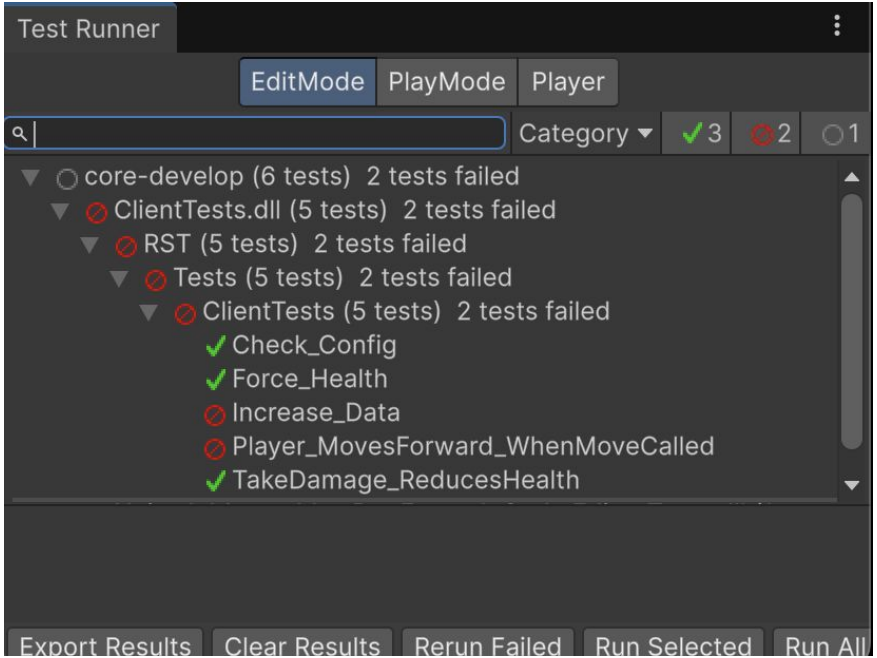
Важно: имена папок должны содержать “EditMode” и “PlayMode”, чтобы Unity автоматически определила тип тестов.

Как запускать тесты

Если Test Runner не видит тесты, или тесты находятся во вложенных папках — необходимо сделать сборку (asmdef) и указать что она для тестов, например:

```
{
   "name": "PlayModeTests",
   "rootNamespace": "",
   "references": [
       "UnityEditor.TestRunner",
       "UnityEngine.TestRunner"
   ],
   "includePlatforms": [],
   "excludePlatforms": [],
   "allowUnsafeCode": false,
   "overrideReferences": true,
   "precompiledReferences": [
       "nunit.framework.dll"
   ],
   "autoReferenced": false,
   "defineConstraints": [],
   "versionDefines": [],
   "noEngineReferences": false,
   "testAssemblies": true
}
```

Как запускать тесты



Window → General → Test Runner

- Запускать все тесты или выбранные
- Смотреть статус (зелёный — пройден, красный — провален)
- Смотреть лог ошибок, время выполнения
- Повторно запускать только неудачные тесты.

Демо

Как запускать тесты

```
[Test, Category("Gameplay")]
public void Score_Increases_OnCollectingCoin()
{
    Assert.AreEqual(10, GameManager.Instance.Score);
}
```

Можно группировать тесты по категориям, чтобы удобно фильтровать и запускать конкретную категорию — актуально при большом количестве тестов.

Как запускать тесты

Полезные атрибуты:

[SetUp] — вызывается перед каждым тестом

[TearDown] — вызывается после каждого теста

Как запускать тесты

Для командных проектов тесты можно запускать автоматически при каждом коммите.

GitHub Actions, GitLab CI, Husky, Jenkins, Unity Cloud Build.

Команда для запуска:

```
unity -runTests -projectPath . -testResults results.xml -testPlatform editmode
```

Что тестировать в игре

- Игровая логика — урон, опыт, апгрейды, коллизии, очки
- Баланс — проверка формул урона и роста сложности
- UI — нажатие кнопок, отображение текста
- Загрузка — адреса сцены, ресурсы Addressables
- Производительность — FPS, Instantiate/Destroy, загрузка ассетов
- Сохранения — корректная сериализация и загрузка данных
- Сеть — отправка/приём пакетов, reconnect

Полезные советы

Регулярно запускай тесты!

Не пытайся писать тесты “на всё”!

- Разделяй логику и поведение — всё, что можно протестировать без Unity API, выноси в обычные классы (тут про архитектуру, ага!)
- Изолируй тесты — каждый тест должен сам создавать и уничтожать объекты. Не полагайся на состояние сцены
- Пиши короткие тесты — один тест = одна проверка (Arrange → Act → Assert)
- Используй Mock-объекты — подменяй зависимости, если они обращаются к сцене или сети. (через интерфейсы, NSubstitute, Moq и т. д.)
- Категоризируй тесты — помогает фильтровать тесты в Test Runner
- Не забывай про PlayMode Tests — они медленнее, но незаменимы для проверки поведения во времени

Методики тестирования (TDD, BDD, CI/CD)

- TDD (Test-Driven Development)
- BDD (Behavior-Driven Development)
- CI/CD (Continuous Integration / Continuous Delivery)
- ATDD (Acceptance Test Driven Development)
- Exploratory Testing

TDD (Test-Driven Development)

Разработка через тестирование — сначала пишешь тест, потом код, который заставит этот тест пройти.

TDD — это процесс разработки, где тесты определяют, что нужно реализовать, прежде чем писать реализацию.

Классический цикл TDD:

1. Пишем тест, который заведомо упадёт (так как функционал ещё не реализован) — тест не проходит
2. Пишем минимально возможный код, чтобы тест прошёл — Тест проходит
3. Улучшаем код, не ломая тесты — Всё остаётся зелёным

TDD (Test-Driven Development)

```
[Test]
public void Player_StartsWithFullHealth()
{
    var player = new Player();
    Assert.AreEqual(100, player.Health);
}
```

Шаг 1. Пишем тест

Тест не проходит — потому что класса Player ещё нет.

TDD (Test-Driven Development)

```
public class Player
{
    public int Health = 100;
}
```

Шаг 2. Пишем минимальный код

Теперь тест проходит

TDD (Test-Driven Development)

```
public class Player
{
    public int Health { get; private set; } = 100;
}
```

Шаг 3. Улучшаем код

Тест проходит, ничего не сломалось

TDD (Test-Driven Development)

Плюсы

- Код изначально тестируемый и чистый
- Ускоряет отладку — меньше багов на поздних стадиях
- Уверенность при рефакторинге
- Заставляет думать о поведении, а не об имплементации.

Минусы

- Трудно применять к игровым механикам, зависящим от Update(), Time, Physics и UI
- Нужна хорошая изоляция логики от UnityEngine API
- Сложно соблюдать дисциплину “тест → код → рефактор”
- Увеличивает срок разработки

BDD (Behavior-Driven Development)

Разработка через поведение — пишем тесты на языке поведения, понятном человеку.

BDD — это развитие TDD, где тесты описывают ожидаемое поведение системы, а не её внутренние детали. Используется стиль “Given / When / Then” (дано / когда / тогда).

```
[Test]
public void GivenPlayerHas100Health_WhenTakes30Damage_ThenHealthBecomes70()
{
    var player = new Player(100);
    player.TakeDamage(30);

    Assert.AreEqual(70, player.Health);
}
```

BDD (Behavior-Driven Development)

Плюсы

- Понятно даже не программистам (геймдизайнерам, QA)
- Подходит для спецификаций и документации
- Помогает проектировать поведение игровых систем (например, “если здоровье < 0, то игрок умирает”)

Минусы

- Требует дисциплины и времени
- Иногда превращается в “много слов — мало пользы”, если не используется системно
- Увеличивает срок разработки

CI/CD (Continuous Integration / Continuous Delivery)

Тесты должны запускаться автоматически при каждом изменении кода.

- **CI** — постоянная интеграция: при каждом git push проект автоматически собирается и прогоняет все тесты
- **CD** — постоянная доставка: если тесты прошли, билд автоматически публикуется (например, на тестовом устройстве или сервере)

CI/CD (Continuous Integration / Continuous Delivery)

Плюсы

- Экономия времени
- Мгновенная проверка ошибок
- Меньше “сломанных” сборок

Минусы

- Настройка требует времени и скиллов
- Долгие тесты замедляют pipeline

ATDD (Acceptance Test Driven Development)

Пишем тесты, отражающие требования заказчика или дизайнера

ATDD — тесты пишутся на уровне требований — они проверяют, что реализовано именно то, что нужно пользователю.

```
[Test]
public void Player_ReachesGoal_WhenCollects100Coins()
{
    var player = new Player();
    for (int i = 0; i < 100; i++)
        player.AddCoin();

    Assert.IsTrue(player.HasReachedGoal);
}
```

Пример: если игрок набрал 100 монет — должен появиться экран победы

Exploratory Testing (исследовательское тестирование)

Когда человек вручную ищет баги, экспериментируя с системой.

В геймдеве это обязательно, потому что многие ошибки — в логике геймплея, восприятии, взаимодействии UI.

- Не заменяет автоматические тесты, а дополняет их
- Хорошо совмещается с автоматическими smoke-тестами

Оптимизация

Перед тем как что-то оптимизировать, важно понять:

- Что именно тормозит игру?
- Это CPU, GPU или память?
- Как часто это происходит?

Оптимизация

CPU

Симптомы: низкий FPS, лаги при логике, физике, AI
Причина: много объектов, тяжёлые скрипты, GC

GPU

Симптомы: FPS падает при графике, эффектах, постпроцессинге
Причина: слишком много draw calls, полигонов, шейдеров

Memory

Симптомы: подвисания, загрузки, вылеты
Причина: утечки, неосвобожденные ресурсы, большие текстуры

Оптимизация, инструменты

- Profiler — главный инструмент анализа производительности: CPU, GPU, Rendering, Memory, etc
- Frame Debugger — пошаговый разбор отрисовки кадра (draw calls, batching)
- Profile Analyzer — используется для сравнения разных профайлов (до/после оптимизаций)
- Memory Profiler — показывает использование памяти
- Deep Profile — детальный анализ вызовов методов (включать только при необходимости)
- Stats Window — быстрый просмотр FPS, batches, tris/verts
- Build Report Inspector — показывает, что занимает место в сборке

Оптимизация, инструменты

- Graphy — runtime-инструмент для мониторинга производительности, FPS и время кадра, нагрузку на CPU и GPU, использование памяти (RAM и VRAM), GC allocations

- Project Auditor — анализирует проект, код, ассеты, настройки и выявляет потенциальные проблемы производительности, памяти и размера сборки

- PVS-Studio — статический анализатор кода, скрытые ошибки и логические баги, потенциальные null reference, неправильное использование API, неоптимальные конструкции, антипаттерны

Build Report Inspector

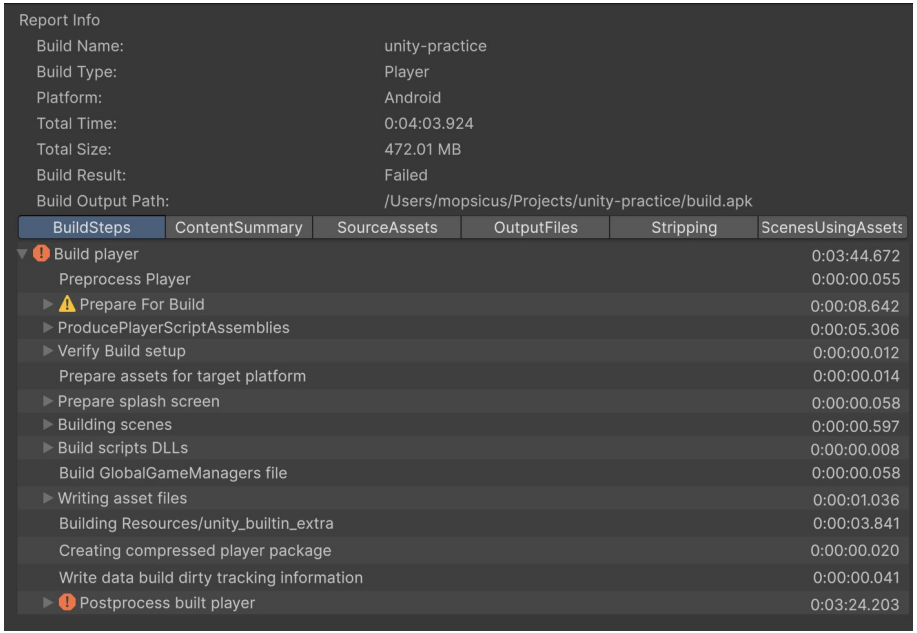
Установка через UPM:

```
"com.unity.build-report-inspector":
"https://github.com/Unity-Technologies/BuildReportInspector.git?path=com.unity.build-report-inspector"
```

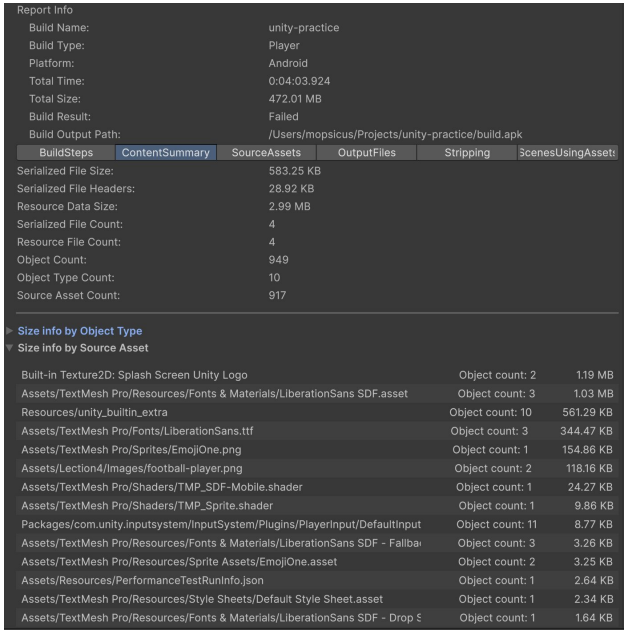
Показывает после билда:

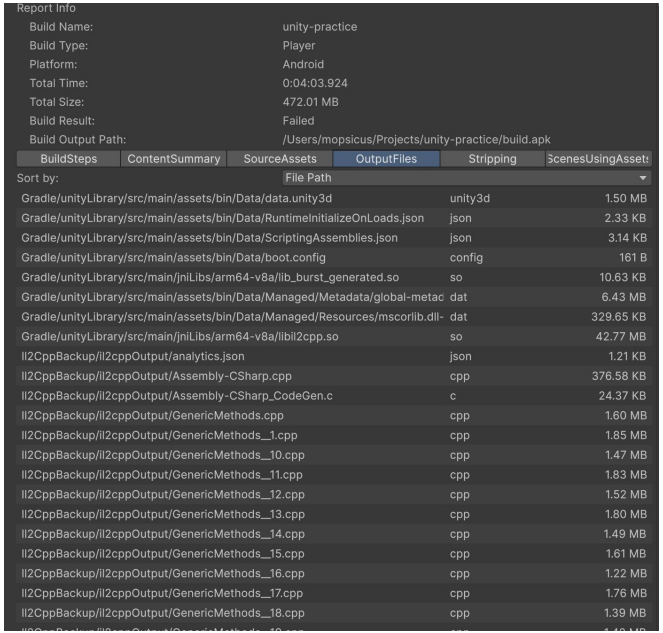
- Summary — общие сведения о сборке: время, размер, платформа, сцены
- Assets — полный список ассетов, вошедших в билд, с размером
- Build Steps — поэтапное время сборки (сколько занял компилятор, упаковка и т.д.)
- Stripping Info — что Unity вырезал из кода (Managed stripping level)
- Raw Sizes — подробные данные по каждому типу файлов

Build Report Inspector



Build Report Inspector





Build Report Inspector

Пример использования:

Допустим, билд стал с 90 МБ → 180 МБ. Открываешь Build Report Inspector и видишь:

```
Assets/Textures/Background.png = 25 MB
Assets/Models/Character.fbx = 12 MB
Assets/Sounds/music.mp3 = 38 MB
```

Значит, большая часть веса — текстуры и звук. Можно попробовать уменьшить размер:

- сжать текстуры в ASTC (мобильные) или DXT (ПК)
- перевести музыку в Vorbis и понизить bitrate
- вынести ассеты в Addressables

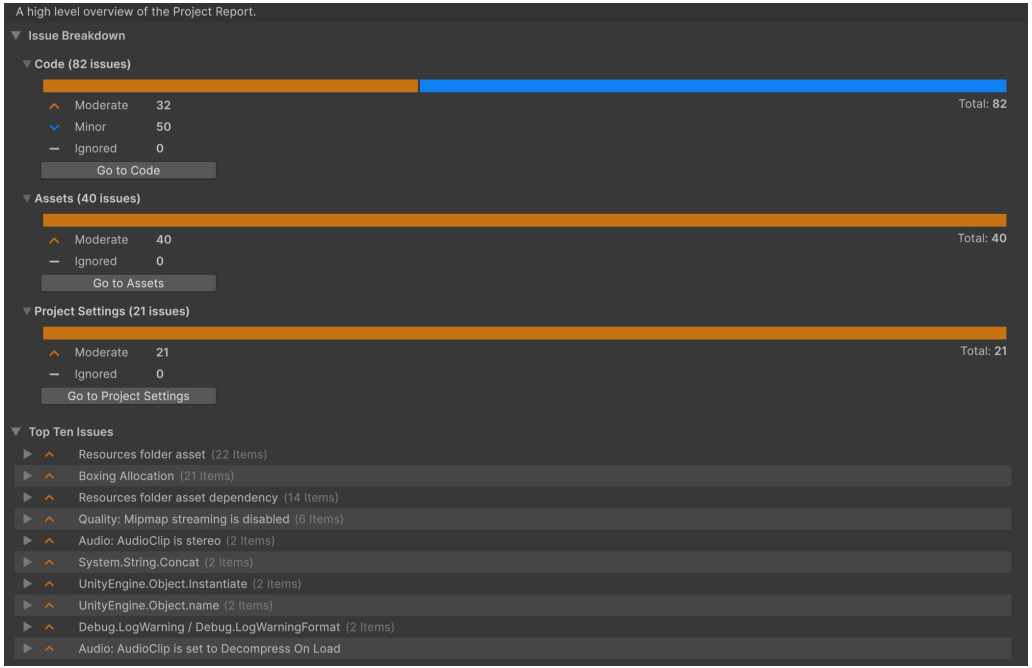
Project Auditor

Официальный инструмент от Unity Technologies для статического анализа проекта.

Что делает:

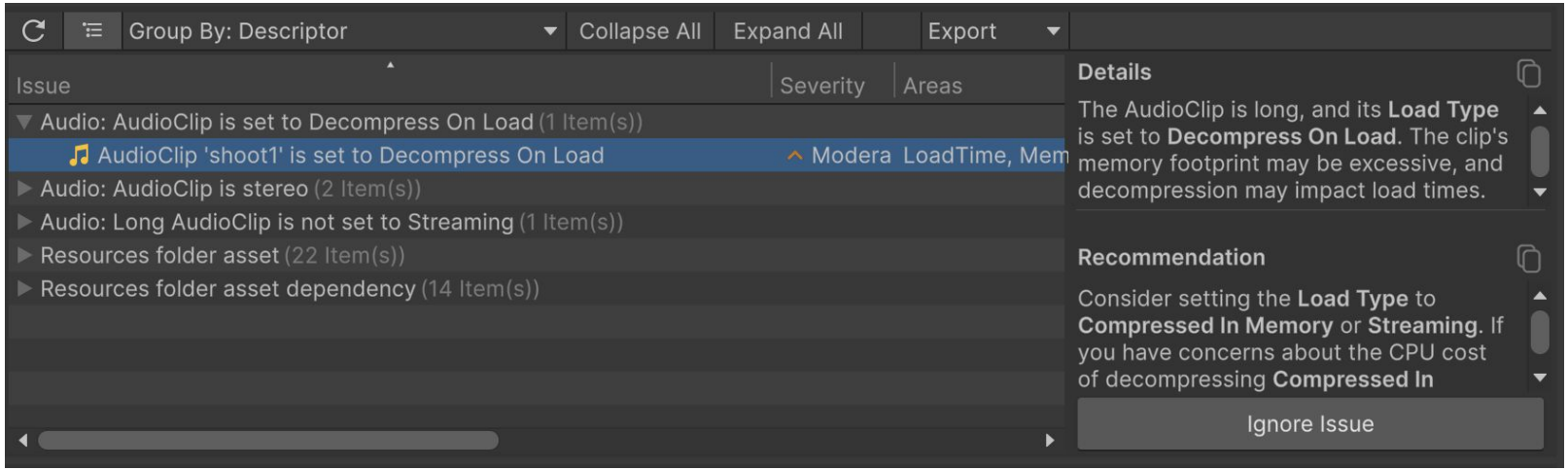
- Находит избыточные или неиспользуемые ассеты
- Показывает дорогие операции в коде (FindObjectOfType, GetComponent в Update и т.д.)
- Проверяет настройки Player Settings и Quality Settings
- Оценивает размер сборки, CPU / GPU hot spots и GC Allocations
- Формирует детальный отчёт с рекомендациями

Project Auditor

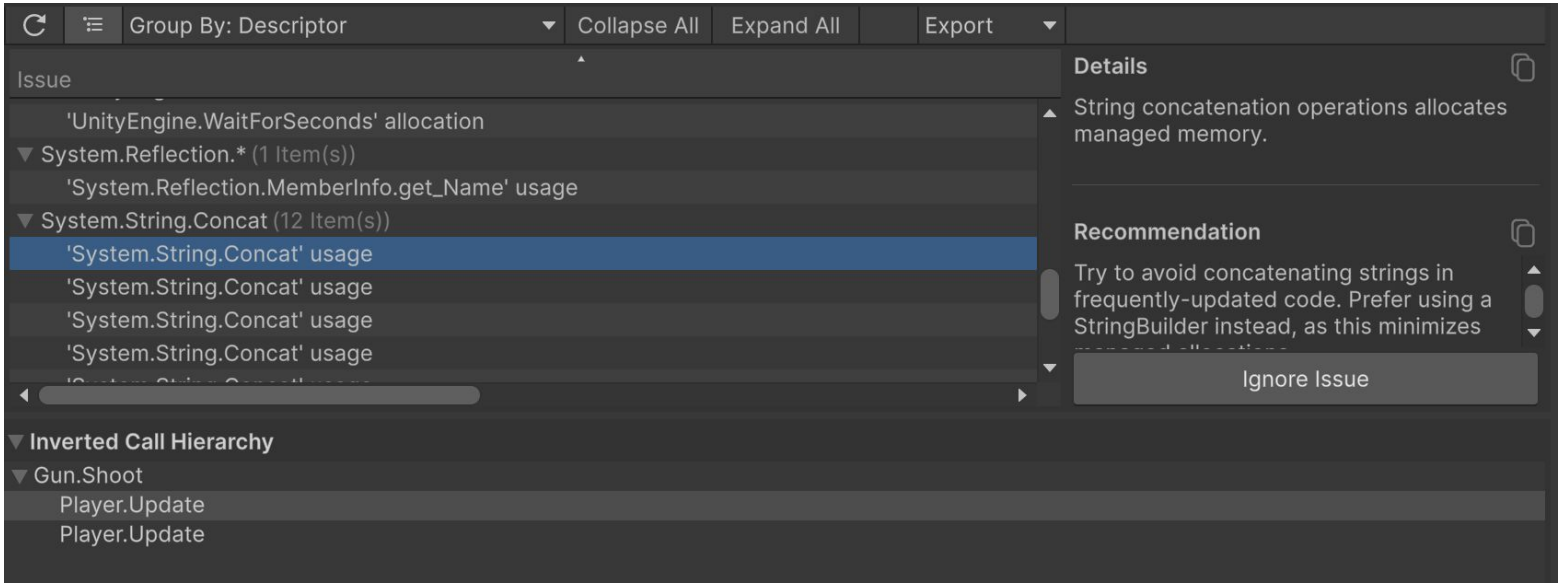


- CodeAnalyzer — сканирует C# код и вызывает анализ API
- AssetAnalyzer — Проверяет текстуры, аудио, модели
- Settings Analyzer — Проверяет настройки Player, Quality, Editor
- Build Report Analyzer — Анализирует результаты последнего билда
- Shader Analyzer — Проверяет количество ключевых слов и вариантов шейдеров
- GC Analyzer — Находит места, где создаются аллокации памяти

Project Auditor



Project Auditor



Демо

Project Auditor

```
using Unity.ProjectAuditor.Editor;

public class AuditRunner
{
    [MenuItem("Tools/Run Project Audit")]
    static void RunAudit()
    {
        var auditor = new Unity.ProjectAuditor.Editor.ProjectAuditor();
        var issues = auditor.Audit();

        foreach (var issue in issues)
        {
            UnityEngine.Debug.Log(issue.Description);
        }
    }
}
```

Можно подключить в GitHub Actions / Jenkins, чтобы проверять качество проекта при каждом коммите (например, не допускать FindObjectOfType в Update).

PVS-Studio

Один из самых сильных внешних инструментов анализа кода для Unity-проектов (и вообще C#).

Он не запускает игру, а анализирует исходники, чтобы найти:

- скрытые ошибки и логические баги,
- потенциальные null reference,
- неправильное использование API,
- неоптимальные конструкции,
- антипаттерны (например, дублирование кода, забытый Dispose и т.п.).

PVS-Studio

Для Unity он тоэе полезен, потому что:

- проверяет C#-код всех скриптов,
- умеет распознавать Unity-специфичные паттерны (MonoBehaviour, Unity API),
- находит ошибки в логике, которые тесты и дебаг не замечают.

https://habr.com/ru/companies/pvs-studio/articles/890962/
https://habr.com/ru/companies/pvs-studio/articles/932356/
https://habr.com/ru/companies/pvs-studio/articles/767944/
https://habr.com/ru/companies/pvs-studio/articles/886662/

Graphy

Graphy — это популярный runtime-инструмент для мониторинга производительности.

Работает прямо в билде:

- показывает статистику в реальном времени на экране,
- не требует подключения Profiler,
- подходит для мобильных тестов и консольных устройств.

Это особенно удобно, если:

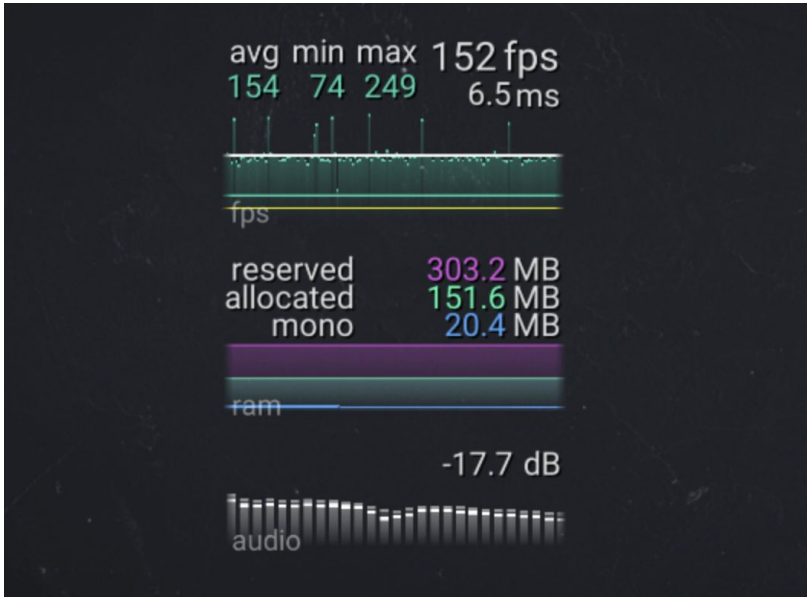
- вы тестируете оптимизацию на устройстве (Android/iOS);
- вы делаете playtests и хотите отслеживать FPS;

Graphy

- Работает на всех платформах: PC, Mac, Android, iOS, WebGL
- Практически не влияет на производительность (<0.5 ms overhead)
- Можно встроить в билд (для QA и внутренних тестов)
- Есть API для собственных данных — можно добавлять свои метрики (например, скорость персонажа, пинг, FPS физики)

Graphy

```
{
  "dependencies": {
    ...
    "com.tayx.graphy": "https://github.com/Tayx94/graphy.git",
    ...
  }
}
```



Оптимизация, рекомендации

Кешируйте всё.

Всё что используется больше 1-2 раз — лучше закешировать. Операции типа GameObject.Find(), GetComponent(), FindObjectOfType() достаточно ресурсозатратны, а если они вызываются где-нибудь в цикле или в Update(), то производительность наверняка упадёт.

Оптимизация, рекомендации

Все используемые картинки нужно упаковывать в атласы.

Можно использовать как встроенный инструмент, так и подготавливать атласы в какой-нибудь другой программе, например TexturePacker. Таким образом уменьшается количество вызовов отрисовок ваших спрайтов. Проверить как идёт отрисовка можно с помощью встроенного инструмента Frame Debugger.

Если упаковываете в Unity, убедитесь что атлас влезает на одну страницу! Это видно в инспекторе. Иначе каждая страница по сути новая текстура!

Оптимизация, рекомендации

Старайтесь по возможности делать меньше вызовов xxx.ToList() и xxx.ToArray()

Они создают новые коллекции при каждом вызове. Лучше построить свой код так, чтобы был прямой доступ к массивам, спискам и использовать ранее созданные коллекции повторно.

Оптимизация, рекомендации

Уделите внимание работе со строками, особенно если это частая операция.

Используйте StringBuilder и специальный форматный метод TMP — tmp.SetText("Data: {0}", data), он не создаёт промежуточные строки.

Оптимизация, рекомендации

Используйте пул объектов.

GameObject.Instantiate() — очень дорогая операция! Если есть возможность не использовать её в процессе игры — не используйте. Для большого количества однотипных объектов надо использовать пул объектов (object pool).

Оптимизация, рекомендации

Уменьшайте размер текстур.

Если не знаете как оптимизировать картинки при сохранении в редакторе, прогоните их через какой-нибудь онлайн оптимизатор. В некоторых случаях можно уменьшить размер на 20-40% от оригинала.

https://compresspng.com/

Оптимизация, рекомендации

Если используйте спрайты вне атласов — старайтесь делать картинки (да и сами атласы тоже) размером кратным степени 2, т.е. 1024×1024, 2048×2048. Особенно актуально для WebGL билдов.

Оптимизация, рекомендации

Используйте разные канвасы (Canvas).

При изменении почти любого параметра у любого объекта, весь канвас перерисовывается полностью! Поэтому при построении сложного UI имеет смысл статичные элементы располагать на отдельном канвасе.

И конечно не изменять UI в Update, а только по событиям, когда действительно что-то поменялось.

Оптимизация, рекомендации

Оптимизируйте прокручивающиеся списки. Родной ScrollView плохо справляется с большим количеством элементов.

Используйте data-driven списки. Также есть смысл заменить Mask на RectMask2D в списках, потому что обычная маска работает через шейдер с обрезкой по альфе с помощью дополнительного материала (больше DC), а прямоугольная маска работает только с прямоугольниками, отсеивает пиксели и вершины за пределами области, не трогая шейдеры и не создавая лишние материалы, соответственно меньше DC и нагрузки на GPU.

Оптимизация, рекомендации

Регулируйте частоту кадров.

Понижая фреймрейт на сценах или игровых меню, где ничего не двигается, можно значительно снизить CPU, а следовательно продлить жизнь батарее устройства.

```
QualitySettings.vSyncCount = 0;

Application.targetFrameRate = XX;
```

Если не отключить вертикальную синхронизацию, то изменение фреймрейта игнорируется!

Оптимизация, рекомендации

Обновляемый список рекомендаций по оптимизации:

https://mopsicus.ru/notes/unity-game-optimization.html

Оптимизация, ещё

- Batching/Static Batching/Dynamic Batching: правильные материалы и шейдеры
- Jobs + Burst, потоки: для тяжёлых параллельных задач (AI, pathfinding, массовая логика)
- Пулы объектов: пулить пули/врагов/буферы, ресайзить буферы заранее
- Минимизировать Reflection и LINQ: особенно при частых вызовах
- Lightmapping / Baked lighting: запекание света, вынос статических объектов в lightmap
- Draw Calls: комбинировать меши, использовать atlases, GPU instancing
- Streaming / Addressables: загружать контент по требованию, выгружать неиспользуемое
- Ресурсоёмкие форматы: использовать ETC2/ASTC для текстур на мобилке, компрессия аудио
- Сеть: эффективная сериализация (protobuf, MessagePack), лимитировать частоту отправки

Оптимизация

Не оптимизируй наугад. Сначала измерь — потом исправляй!

Преждевременная оптимизация — зло.

Оптимизация, алгоритм

1

Диагностика — определить, что именно тормозит, задача собрать метрики: FPS и frame time, CPU/GPU load, GC allocations, memory usage, draw calls, batches

Оптимизация, алгоритм

2

Анализ — интерпретировать данные, определить, что именно вызывает просадки. Основные типы узких мест: утечки, частые GC, много draw calls, overdraw, высокое время в Scripts, Physics, Animation

Оптимизация, алгоритм

3

Локализация — найти точное место проблемы, где именно в коде/ассетах происходит просадка: использовать Profiler, включать/отключать системы поочерёдно, профилировать на реальном устройстве

Оптимизация, алгоритм

4

Оптимизация — внести целевые изменения, исправить только то, что реально влияет на производительность

Оптимизация, алгоритм

5

Проверка — убедиться, что стало лучше, снова Profiler, Automated Tests / Performance Tests, Build Report Inspector. Задача: зафиксировать результат — например, FPS вырос с 45 до 60.

Оптимизация, алгоритм

5

Документирование и автоматизация — сделать оптимизацию повторяемой частью процесса

Бонус

Addressables

Управление ресурсами

В Unity все игровые объекты (спрайты, модели, сцены, звуки) — **Assets**.

По мере роста проекта:

- билд становится большим
- время загрузки увеличивается
- трудно обновлять игру без пересборки

Resources

Папка Assets/Resources позволяет загружать ресурсы по имени:

- все ресурсы в папке Resources включаются в билд
- нет возможности подгружать внешние данные
- увеличивает память и размер APK/IPA

```
var prefab = Resources.Load<GameObject>("Enemy");
```

Resources

Плюсы:

- Просто
- Работает “из коробки”

Минусы:

- Все ресурсы из Resources *всегда* попадают в билд
- Невозможно выгрузить из памяти частично
- Нет контроля зависимостей
- Невозможно обновить без новой сборки

Asset Bundles

Более гибкий способ: можно “упаковать” ассеты в отдельные пакеты. Можно хранить на сервере, обновлять частично.

Загружаются во время выполнения:

```
var bundle = AssetBundle.LoadFromFile(path);
var prefab = bundle.LoadAsset<GameObject>("Enemy");
```

Resources

Плюсы:

- Позволяют разделить игру на модули
- Можно реализовать обновления без пересборки

Минусы:

- Нужно вручную собирать и версионировать бандлы
- Самостоятельно отслеживать зависимости
- Требуется писать собственные системы загрузки, кэширования, контент-менеджеров
- Сложно в обслуживании

Самописные системы на основе бандлов

Многие студии писали “менеджеры ресурсов”, которые:

- Скачивают и хранят версии бандлов (через JSON-манифест)
- Проверяют наличие обновлений
- Кэшируют локально
- Следят за зависимостями

Каждый писал свою реализацию — поддерживать и тестировать сложно, особенно при обновлениях Unity.

Addressables

Addressables — это “официальная надстройка” Unity над Asset Bundles, которая автоматизирует всё то, что раньше приходилось писать вручную:

- Управление зависимостями
- Версионирование
- Кэширование и обновления контента
- Асинхронную загрузку
- Отладку и анализ

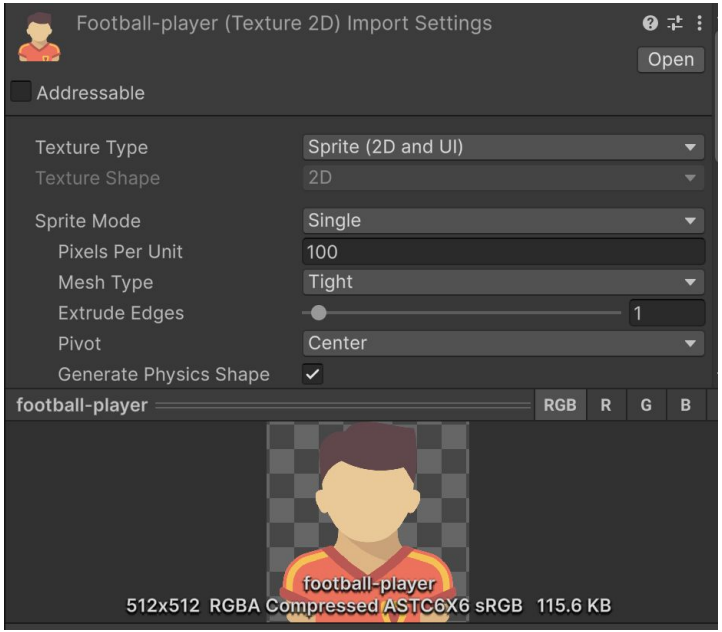
Addressables

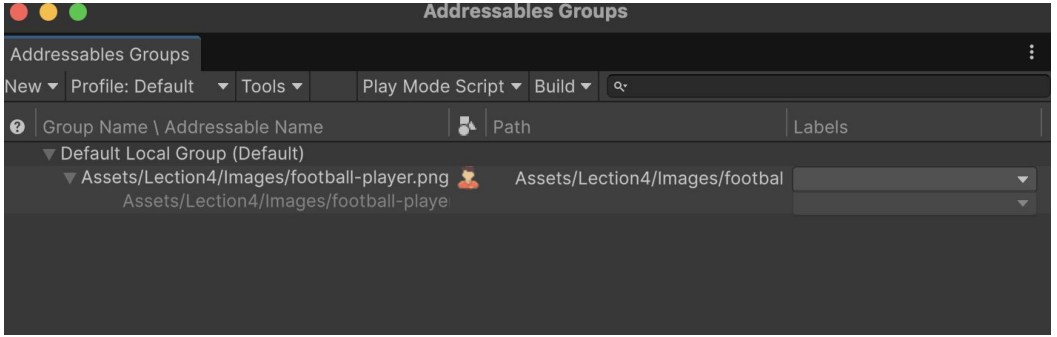
Любой ресурс можно сделать “адресуемым” и загружать его по имени или метке, не задумываясь о том, где он находится — в локальном билде, на сервере или в обновлении.

Addressables

- **Addressable Asset** — любой объект, отмеченный флажком Addressable
- **Address** — уникальное имя ресурса, по которому он загружается
- **Groups** — контейнеры для сборки и загрузки (например: “UI”, “Enemies”)
- **Labels** — теги для логической фильтрации (например: “Level1”, “Boss”)
- **Profiles** — набор путей сборки/загрузки для разных окружений
- **Catalog** — JSON-файл, который хранит метаданные всех ресурсов
- **Build / Load Paths** — пути, где ресурсы собираются и откуда загружаются

Addressables





Демо

Addressables

```
using UnityEngine;
using UnityEngine.AddressableAssets;
using Cysharp.Threading.Tasks;

public class Example : MonoBehaviour
{
    async void Start()
    {
        var handle = Addressables.LoadAssetAsync<GameObject>("EnemyPrefab");
        var prefab = await handle.Task;
        Instantiate(prefab);
        Addressables.Release(handle);
    }
}
```

- AsyncOperationHandle — структура, управляющая асинхронной операцией
- await handle.Task — дожидаемся загрузки
- После использования нужно вызывать Addressables.Release(handle)

Addressables

```
var obj = await Addressables.InstantiateAsync("EnemyPrefab", transform);
```

Addressables создаёт объект в сцене и сам следит за зависимостями

```
Addressables.ReleaseInstance(obj);
```

Важно освобождать, иначе память будет утекать

Addressables

```
if (handle.Status == AsyncOperationStatus.Failed)
    Debug.LogError(handle.OperationException);
```

Обработка ошибок и исключений

Addressables

Если проект использует стандартную схему (без Addressables):

- Все ассеты включаются в билд
- Unity загружает часть их в память при старте игры

- Результат:
  - Размер APK/IPA растёт
  - Время запуска увеличивается
  - Память занята даже тем, что сейчас не нужно

Addressables

Когда ты отмечаешь ассеты как Addressable:

- Они исключаются из основного билда (из data.unity3d)
- Вместо этого собираются в отдельные AssetBundles, по группам (group_0.bundle, group_1.bundle и т.д.)
- Эти бандлы могут храниться локально, в StreamingAssets, или загружаться по сети

Addressables

Когда мы делаем ассеты Addressable и разбиваем их по группам, Unity перестаёт класть их в общий файл data.unity3d.

Это:

- снижает размер базового билда
- ускоряет запуск
- уменьшает загрузку памяти
- позволяет подгружать ресурсы только тогда, когда они реально нужны

Addressables

- Используйте Addressables вместо Resources.Load
- Разделяйте группы по типу контента (UI, звуки, эффекты, сцены)
- Ставьте Label для удобной загрузки по категориям
- Освобождайте ресурсы (Release / ReleaseInstance)
- Используйте async/await или UniTask для асинхронной логики
- Тестируйте обновление контента через Remote группы
- Следите за зависимостями, чтобы не грузить лишнее

Addressables, ошибки

- Загрузка ресурсов без await → недогрузка/глюки
- Неосвобождение handle → утечки памяти
- Пересечение ссылок в сцене → зависимые объекты не выгружаются
- Случайное дублирование адресов → конфликт в каталоге

Addressables

https://youtu.be/hO_yNuJ5QGw?si=wUb3TWodn2dHi_G8

https://www.youtube.com/live/tZ9fyjW1ICM?si=EBjyYvhCZ9NbmDqT

Что дальше

Обзор сетевых решений
Клиент-серверная архитектура
Интеграция сервисов и SDK