



# Input System, камеры, звуки

Проекции: перспективная, ортогографическая, менеджер звуков, бонус




# Input System

```
void Update()
{
    float horizontal = Input.GetAxis("Horizontal");
    float vertical = Input.GetAxis("Vertical");

    Vector3 move = new Vector3(horizontal, 0, vertical);
    transform.Translate(move * Time.deltaTime * 5f);

    if (Input.GetKeyDown(KeyCode.Space))
        Debug.Log("Jump!");
}
```



- Доступен по умолчанию через класс `UnityEngine.Input`
  - Работает на уровне проекта, без необходимости настраивать объекты
  - Много примеров
  - Плохо подходит для сложных схем управления
  - Не работает с UI напрямую
  - Не поддерживает горячую смену устройств
- 



# Новый Input




```
using UnityEngine;
using UnityEngine.InputSystem;

public class PlayerController : MonoBehaviour
{
    private Vector2 moveInput;
    public float speed = 5f;

    // Автоматически вызывается, если в PlayerInput стоит Behavior = Send Messages
    void OnMove(InputValue value)
    {
        moveInput = value.Get<Vector2>();
    }

    void OnJump()
    {
        Debug.Log("Jump pressed!");
    }

    void Update()
    {
        Vector3 move = new Vector3(moveInput.x, 0, moveInput.y);
        transform.Translate(move * speed * Time.deltaTime);
    }
}
```



- Поддерживает клавиатуру, мышь, геймпады, тач, VR
- Построена по событийному принципу (event-based input)
- Высокая гибкость (можно делать схемы, карты, профили)
- Не все плагины и ассеты поддерживают новую систему
- Сложнее отлаживать
- Выше порог входа и меньше примеров



# Новый Input

```
using UnityEngine;
using UnityEngine.InputSystem;

public class PlayerInputExample : MonoBehaviour
{
    private PlayerControls controls;
    private Vector2 move;

    void Awake()
    {
        controls = new PlayerControls();

        controls.Player.Move.performed += ctx => move = ctx.ReadValue<Vector2>();
        controls.Player.Move.canceled += ctx => move = Vector2.zero;

        controls.Player.Jump.performed += _ => Debug.Log("Jump!");
    }

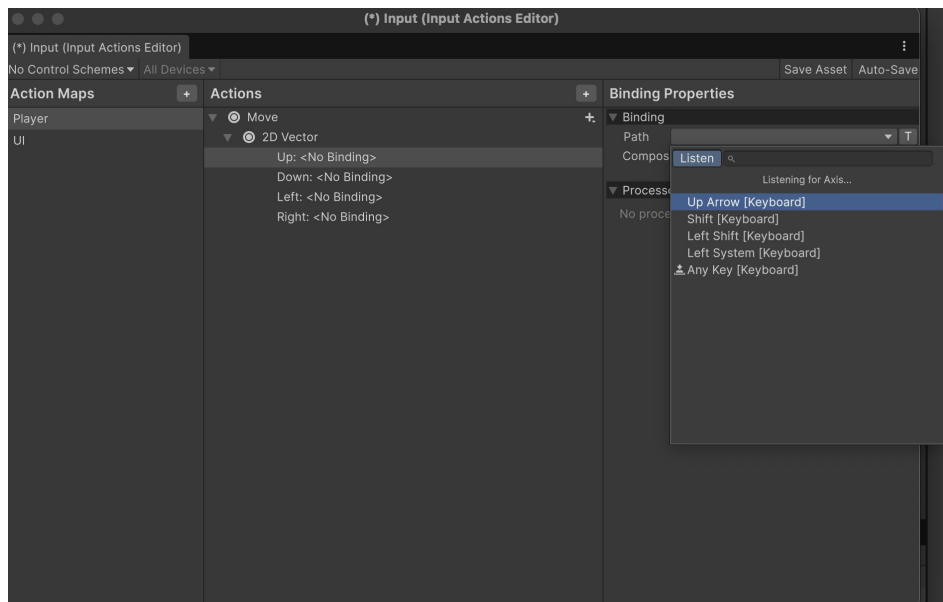
    void OnEnable() => controls.Enable();
    void OnDisable() => controls.Disable();

    void Update()
    {
        transform.Translate(new Vector3(move.x, 0, move.y) * 5f * Time.deltaTime);
    }
}
```

Можно работать напрямую с классом  
PlayerControls

<https://docs.unity3d.com/Packages/com.unity.inputsystem@1.15/manual/index.html>

# Новый Input



```
using UnityEngine;
using UnityEngine.InputSystem;

public class UIController : MonoBehaviour
{
    public PlayerInput playerInput;

    public void OpenMenu()
    {
        playerInput.SwitchCurrentActionMap("UI");
        Cursor.visible = true;
        Cursor.lockState = CursorLockMode.None;
    }

    public void CloseMenu()
    {
        playerInput.SwitchCurrentActionMap("Player");
        Cursor.visible = false;
        Cursor.lockState = CursorLockMode.Locked;
    }
}
```

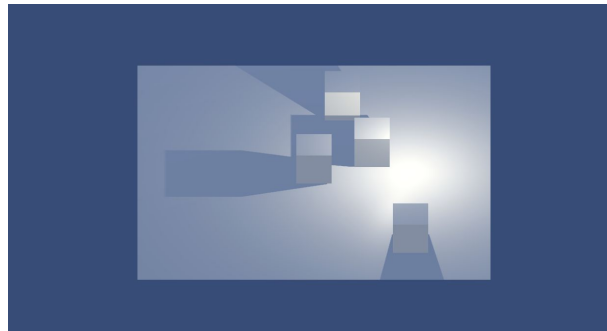
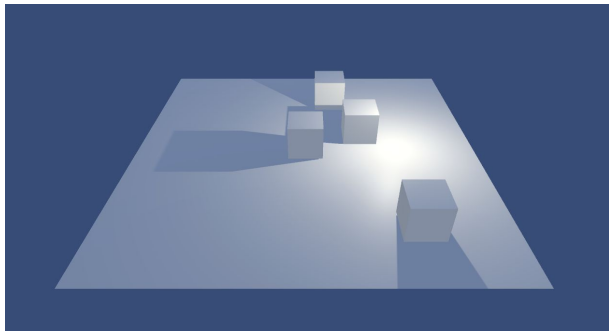


# Старый Input vs. новый Input

Особенность	Старый Input Manager	Новый Input System
Подход	Опрос (polling) через <code>Input.GetKey</code> в <code>Update</code>	События (event-based), <code>Input Actions</code>
Множество устройств	Ограниченно (в основном клавиатура и геймпад)	Поддержка любых устройств (геймпады, VR, тач, кастомные)
Переназначение клавиш	Только через <code>Project Settings</code> (глобально)	Можно делать прямо в игре, динамически
UI и геймпад	Раздельно	Единая система ввода для UI и геймплея
Скорость прототипирования	Быстро, просто	Требует настройки <code>Asset</code> , но потом удобнее
Поддержка мультиплеера (Split Screen)	Почти нет	Встроенная поддержка нескольких игроков
Гибкость	Низкая	Высокая (можно делать схемы, карты, профили)
Совместимость со старым кодом	По умолчанию	Можно включить "Both" режим в <code>Project Settings</code>
Рекомендация Unity	Устаревшая	Современный стандарт

# Камеры

Название	Используется где	Особенности
Перспективная	3D-игры, шутеры, платформеры	Объекты вдали кажутся меньше. Эффект глубины.
Ортографическая	2D, стратегии, изометрия	Без перспективы — все объекты одинакового размера независимо от расстояния.





# Камеры

Параметр	Описание	Рекомендации
Clear Flags	Что очищать перед рендером: Skybox, Solid Color, Depth Only, Nothing	Обычно Skybox или Solid Color
Background	Цвет фона, если Skybox не задан	Подходит для меню, минималистичных сцен
Culling Mask	Какие слои рендерить	Можно исключить UI, эффекты и т.п.
Projection	Perspective / Orthographic	Переключает тип камеры
Field of View (FOV)	Угол обзора (только Perspective)	Стандарт: 60°–70°
Orthographic Size	Размер обзора (только Orthographic)	Чем больше — тем шире область видимости
Clipping Planes (Near / Far)	Диапазон видимости	Слишком маленькие значения → артефакты z-fighting
Depth	Порядок отрисовки (при нескольких камерах)	Камеры с большим Depth рисуются поверх
Viewport Rect	Область экрана, где камера рисует (0–1)	Можно использовать для split-screen или миникарты
Rendering Path	Forward, Deferred, etc.	Зависят от рендера (URP / HDRP)





# Камеры, перспективная

Классическая “реалистичная” камера. Лучи сходятся в одной точке, и объекты, находящиеся дальше, кажутся меньше — как в реальной жизни.

Тип игры / сцены	Почему подходит
3D приключения, RPG, шутеры	Камера показывает глубину и масштаб мира.
Гонки, паркур, полёты	Позволяет ощущать движение и скорость.
Платформеры с 3D графикой	Добавляет “живости” и объёма.
Катсцены и кинематографичные сцены	Можно использовать красивую перспективу, FOV, наклон.
Камеры “от первого лица”	Необходима для реалистичного восприятия.



# Камеры, перспективная



**Реализм**

Создаёт ощущение глубины и объёма, привычное глазу.



**Подходит для 3D миров**

Любые 3D сцены (город, интерьер, горы) выглядят естественно.



**Хорошо передаёт масштаб**

Объекты вдали кажутся дальше — игрок чувствует расстояние.



**Красивые композиции**

Можно ставить FOV (угол обзора), добиваясь “киношных” кадров.



**Подходит для динамики**

Камера движения, прыжки, приближения, вращения выглядят эффектно.



# Камеры, ортографическая

Проекция без перспективы — параллельные линии не сходятся, объекты сохраняют одинаковый размер независимо от расстояния до камеры.

## Тип игры / сцены

## Почему подходит

2D платформеры

Камера стабильная, без искажений.

Изометрические RPG и стратегии

Геометрия читается, удобно кликать по объектам.

Пиксель-арт игры

Спрайты не деформируются, пиксели сохраняют форму.

Пазлы, головоломки, сеточные игры

Удобно проектировать и позиционировать элементы.

Меню, UI-сцены

Геометрия выглядит чётко и без перспективы.



# Камеры, ортографическая

Преимущество	Пояснение
 Без искажений	Размер объекта не зависит от расстояния — идеально для 2D.
 Удобно для точных расчётов	Прекрасно подходит для сеток, тайлов и изометрии.
 Просто проектировать уровни	Координаты объектов на экране стабильны.
 UI и 2D элементы выглядят предсказуемо	Нет перспективных “прыжков” или наклонов.
 Оптимальнее для 2D сцен	Меньше вычислений по глубине (Z).

Камеры можно комбинировать



## Несколько камер

- UI отдельно от 3D-сцены
- Миникарта
- Отражения (например, зеркало или вода)
- Разделённый экран (split screen)
- Эффекты постобработки на отдельных слоях
- Сцена в сцене (порталы, мониторы, камеры наблюдения)
- Разные перспективы или слои (2.5D, смешанные сцены)



# Управление камерой

```
using UnityEngine;

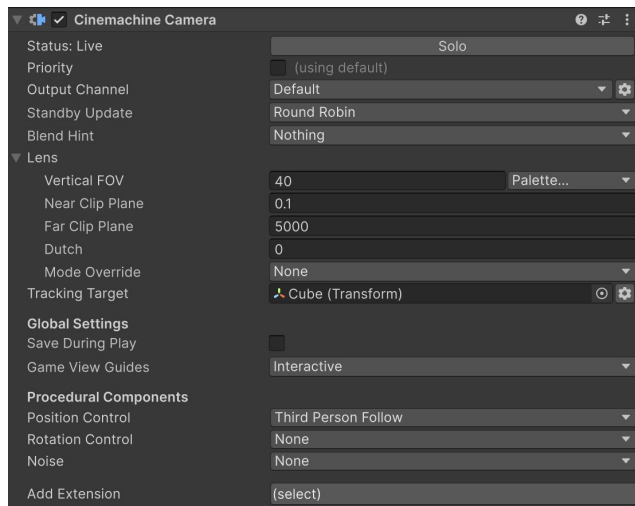
public class SimpleCameraController : MonoBehaviour
{
    public Transform target;
    public float distance = 5f;
    public float sensitivity = 2f;
    private float rotationX, rotationY;

    void LateUpdate()
    {
        rotationX += Input.GetAxis("Mouse X") * sensitivity;
        rotationY -= Input.GetAxis("Mouse Y") * sensitivity;
        rotationY = Mathf.Clamp(rotationY, -30f, 60f);

        Quaternion rotation = Quaternion.Euler(rotationY, rotationX, 0);
        transform.position = target.position - rotation * Vector3.forward * distance;
        transform.LookAt(target);
    }
}
```

# Cinemachine

Надстройка над системой камер Unity, позволяет создавать умные, динамичные камеры без написания кода.



- Не нужно писать код для движения камеры, всё управляется визуально через инспектор
- Разные режимы: следование, вращение, рельсы, катсцены. Можно комбинировать несколько виртуальных камер.
- Оптимизированный код, почти не грузит CPU
- Камера ведёт себя как “живая”: плавные переходы, тряска, коллизии.

<https://docs.unity3d.com/Packages/com.unity.cinemachine@3.1/manual/index.html>










# Cinemachine, зачем

Проблема	Без Cinemachine	С Cinemachine
Нужно, чтобы камера плавно следила за игроком	Пишешь код <code>transform.position = Vector3.Lerp(...)</code>	Просто ставишь "Follow"
Нужно переключать камеры (катсцена, смена ракурса)	Активируешь/деактивируешь вручную	Делается автоматически с плавным переходом
Нужно трясти камеру (взрыв, урон)	Пишешь шейк-код вручную	Есть готовый Noise-профиль
Нужно коллизию камеры со стенами	Пишешь Raycast	Есть компонент <code>CinemachineCollider</code>
Нужно сглаживание и "умное следование"	Много кода	Настраивается ползунками












# Cinemachine, когда использовать

Ситуация / Тип игры	Почему это удобно
 3D от третьего лица (RPG, action)	Простое слежение за игроком, плавное смещение и вращение мышью.
 Гонки, погони, динамичные сцены	Позволяет плавно следить за машиной, автоматически компенсировать скорость и коллизии.
 Катсцены и интро	Несколько камер можно плавно переключать через Priority или Timeline.
 Игры с “кинематографичной” подачей	Плавные переходы, zoom, дрожание — создают “фильмографичный” эффект.
 Игры с переключением режимов камеры	Можно быстро переключать между first-person и third-person камерами.
 2.5D платформеры и стратегии сверху	Легко настраивается Framing Transposer / Top-down режим.
 Мини-катсцены (взрыв, появление босса)	Можно временно поднять Priority другой камеры, сделать эффект “кинематографа”.



# Cinemachine, когда **НЕ** использовать

Ситуация / Тип игры	Почему лучше не использовать
 Простые 2D игры	Overkill: камера не движется, фиксирована.
 Шутеры от первого лица (FPS)	Требуется точный контроль мыши, а Cinemachine добавляет инерцию и сглаживание.
 VR или AR проекты	Камеру управляет устройство (гарнитура / трекинг), Cinemachine мешает.
 RTS / симуляторы / редакторы	Нужен точный zoom, панорамирование и ограниченные границы — проще реализовать вручную.
 Игры с фиксированными ракурсами (Resident Evil стиль)	Проще переключать обычные камеры по триггерам.
 Очень специфическое поведение камеры	Если камера привязана к физике, имеет собственную логику (например, Shake по Rigidbody).
 Прототипы на одну сцену без движущихся объектов	Настройка Cinemachine займёт больше времени, чем эффект.

## Cinemachine, пример

*Ori and the Blind Forest* камера почти сама “снимает фильм” — плавные переходы, фокус на персонаже, эффект глубины.

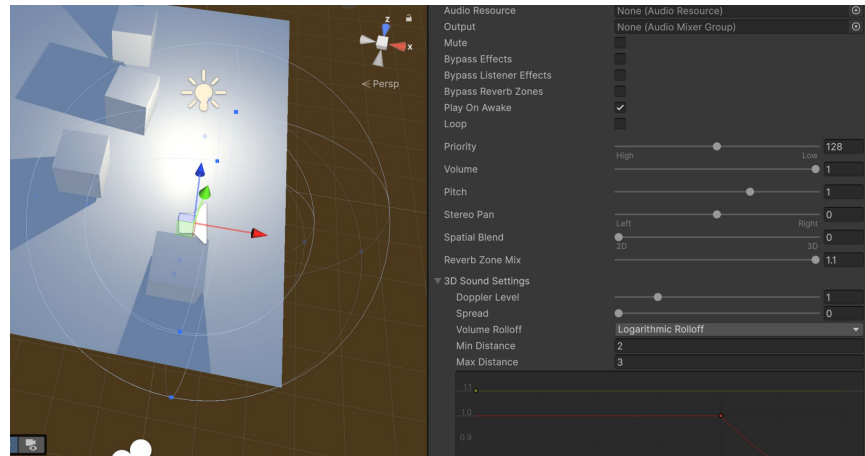


# Звуки

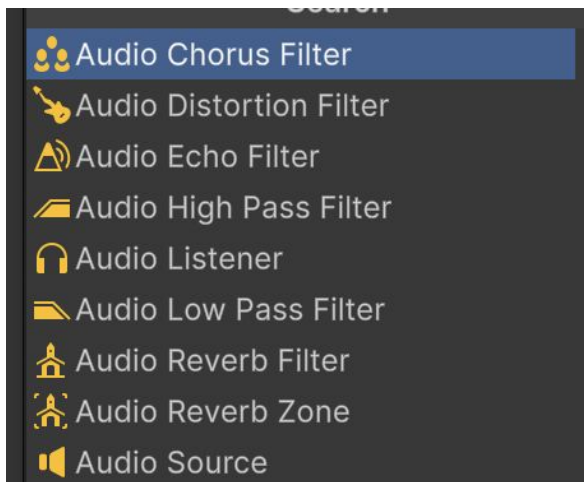
AudioClip — звуковой файл (mp3, wav, ogg)

AudioSource — компонент, который воспроизводит звук

AudioListener — точка, из которой “слышно”.

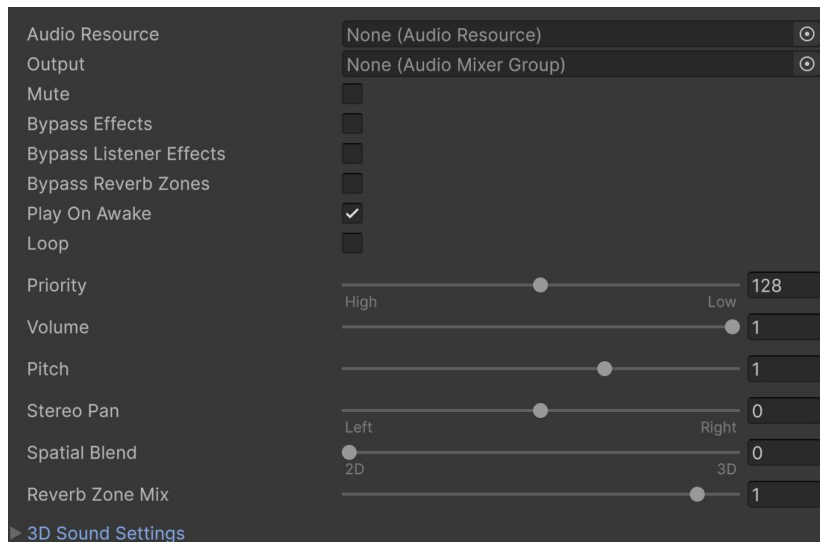


# Звуки



- Audio Chorus Filter — добавляет эффект “хора” (chorus effect) — дублирует сигнал с лёгким сдвигом по фазе и высоте тона. Делает звук “шире” и “толще”, часто используется для гитар, синтезаторов или фоновых мелодий.
- Audio Distortion Filter — добавляет эффект искажения (distortion, overdrive) превращает звук в “перегруженный” — как у электрогитары.
- Audio Echo Filter — добавляет эффект эха. Каждый повтор становится тише — создаётся ощущение отражений.
- Audio High Pass Filter — пропускает только высокие частоты. Используется, чтобы “обрезать” низкие звуки — например, убрать гул или бас.
- Audio Reverb Filter — добавляет эффект реверберации (эхо помещения). Симулирует звучание в разных типах помещений — от комнаты до собора.
- Audio Reverb Zone — пространственная зона, создающая эффект помещения. При входе игрока в зону — применяет реверберацию к звукам. Можно задать радиус и параметры “эха”.

# Звуки



Поле	Описание
Audio Clip	Какой звук проигрывать.
Play On Awake	Автоматически играть при запуске сцены.
Loop	Зацикливать звук.
Volume	Громкость (0–1).
Pitch	Скорость воспроизведения (0.5–2). Можно использовать для "разнообразия".
Spatial Blend	0 = 2D звук, 1 = 3D звук.
Min / Max Distance	Радиус, в котором слышен звук.
Doppler Level	Эффект Доплера (изменение тона при движении).



# Звуки

```
using UnityEngine;

public class ButtonSound : MonoBehaviour
{
    public AudioClip clickSound;
    private AudioSource audioSource;

    void Start()
    {
        audioSource = GetComponent();
    }

    public void PlayClick()
    {
        audioSource.PlayOneShot(clickSound);
    }
}
```

2D

```
public class Explosion : MonoBehaviour
{
    public AudioClip explosionClip;

    void Start()
    {
        AudioSource.PlayClipAtPoint(explosionClip, transform.position);
    }
}
```

3D



# Звуки, менеджер

```
using UnityEngine;

public class AudioManager : MonoBehaviour
{
    public static AudioManager Instance;
    public AudioSource musicSource;
    public AudioSource sfxSource;

    void Awake()
    {
        if (Instance == null)
        {
            Instance = this;
            DontDestroyOnLoad(gameObject);
        }
        else Destroy(gameObject);
    }

    public void PlaySFX(AudioClip clip)
    {
        sfxSource.PlayOneShot(clip);
    }

    public void PlayMusic(AudioClip clip, bool loop = true)
    {
        musicSource.clip = clip;
        musicSource.loop = loop;
        musicSource.Play();
    }

    public void StopMusic()
    {
        musicSource.Stop();
    }
}
```

AudioManager.Instance.PlaySFX(jumpClip);

AudioManager.Instance.PlayMusic(menuMusic);





## Звуки, менеджер

```
public enum FxChannel {  
    Ссылка: 3  
    FIRST = 0,  
    Ссылка: 0  
    SECOND = 1,  
    Ссылка: 0  
    THIRD = 2,  
    Ссылка: 0  
    FOURTH = 3,  
    Ссылка: 1  
    FIFTH = 4,  
    Ссылка: 2  
    MAX = FIFTH  
}
```

Каналы для звуков

```
_music = gameObject.AddComponent<AudioSource>();  
_sounds = new AudioSource[(int)FxChannel.MAX + 1];  
var GameObject go = gameObject;  
for (var int i = 0; i < _sounds.Length; i++) {  
    _sounds[i] = go.AddComponent<AudioSource>();  
}  
_music.loop = false;  
_music.playOnAwake = false;  
foreach (var AudioSource item in _sounds) {  
    item.loop = false;  
    item.playOnAwake = false;  
}
```

Инициализация



## Звуки, менеджер

```
public void PlaySound(string sound, FxChannel channel = FxChannel.FIRST, bool forceInterrupt = false,
    var AudioSource fx = _sounds[(int)channel];
    if (!forceInterrupt && fx.isPlaying) {
        return;
    }
    StopSound(channel);
    fx.clip = _resources.Load<AudioClip>(path: sound);
    fx.pitch = pitch;
#if CORE_DEBUG
    Logger.Info(string.Format("Play sound {0} in channel {1}", sound, channel));
#endif
    if (SoundVolume > 0f && fx.clip != null) {
        fx.Play();
    }
}
```

Воспроизведение

[illegible]



# Звуки, FMOD

Мощный аудиодвижок, используемый в играх и интерактивных приложениях для управления звуком.

- Динамическая аудиомикшировка: можно менять громкость, фильтры и эффекты в реальном времени
- Событийная система: звуки привязываются к событиям (Events) в FMOD Studio
- Интерактивный звук: звук может меняться в зависимости от состояния игры (например, скорость персонажа, здоровье, окружение)
- Поддержка 3D-звука: точная позиция источника звука в пространстве
- Параметры (Parameters): можно управлять громкостью, питчем или эффектами через параметры, которые меняются в игре



## Звуки, FMOD

- FMOD Studio – отдельное приложение для создания и редактирования аудио событий
- FMOD Studio API / Unity Integration – плагин для Unity, который позволяет использовать события FMOD внутри проекта



## Звуки, FMOD, зачем

- Можно создавать интерактивную музыку, которая меняется под действия игрока
- Легко управлять одним и тем же звуком через несколько объектов
- Можно добавлять реальные аудиозффекты (реверберацию, фильтры, side-chain) прямо в игре
- Поддержка мобильных платформ, консолей и ПК



## Звуки, FMOD

- Сложная звуковая дизайн: если проект требует сложной звуковой архитектуры с динамическими изменениями звука в зависимости от игрового процесса
- Интерактивная музыка: создание адаптивной музыки, которая меняется в зависимости от действий игрока
- Поддержка различных платформ: FMOD может предложить более оптимизированную производительность

В Unity 6 многое было улучшено в аудиосистеме, но для сложной звуковой архитектуры, адаптивной музыки или поддержки множества платформ, FMOD может быть полезен.



## Звуки, FMOD, минусы

- Дополнительная интеграция
- Сложность для новичков
- Дополнительный вес проекта
- Зависимость от стороннего инструмента
- Необходимость сборки событий





# Бонус

Сохранение данных



# Бонус

Сохранение данных



## Бонус

1. PlayerPrefs – для небольших данных, настроек
2. Файлы JSON / XML / Binary – для прогресса и сложных структур
3. Серверное хранение (Cloud / Backend) – для мультиплеера или кроссплатформенных игр



## Бонус, PlayerPrefs

```
using UnityEngine;

public class PlayerScore : MonoBehaviour
{
    private int score = 0;

    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            score += 10;
            Debug.Log("Score: " + score);
        }

        if (Input.GetKeyDown(KeyCode.S))
        {
            PlayerPrefs.SetInt("playerScore", score);
            PlayerPrefs.Save();
            Debug.Log("Score saved!");
        }

        if (Input.GetKeyDown(KeyCode.L))
        {
            score = PlayerPrefs.GetInt("playerScore", 0);
            Debug.Log("Score loaded: " + score);
        }
    }
}
```

- int — целые числа
- float — числа с плавающей точкой
- string — строки

### Минусы

- Лимиты по объему
- Безопасность
- Отсутствие сложных функций



# Бонус, JSON

```
[System.Serializable]
public class PlayerData
{
    public int level;
    public float health;
    public string playerName;
}
```

```
void SaveData()
{
    PlayerData data = new PlayerData();
    data.level = 5;
    data.health = 75.5f;
    data.playerName = "Student";

    string json = JsonUtility.ToJson(data, true);
    File.WriteAllText(filePath, json);
    Debug.Log("Data saved: " + filePath);
}
```

```
void LoadData()
{
    if (File.Exists(filePath))
    {
        string json = File.ReadAllText(filePath);
        PlayerData data = JsonUtility.FromJson<PlayerData>(json);
        Debug.Log($"Loaded: {data.playerName}, Level: {data.level}, Health: {data.health}");
    }
    else
    {
        Debug.Log("No save file found.");
    }
}
```



## Бонус, JSON

### Минусы

- Сериализация/десериализация больших объектов может занимать время
- Текстовый JSON занимает больше места, чем бинарные форматы
- JSON-файлы легко редактировать игроком
- Ограничения Unity JsonUtility



## Бонус, бинарное

- Данные сериализуются в бинарный формат и сохраняются в файл
- Файл не читается обычным текстовым редактором — это защищает от случайного изменения
- Подходит для больших объектов, карт уровней, инвентаря, прогресса, когда важна скорость и компактность



# Бонус, бинарное

```
void LoadData()
{
    if (File.Exists(filePath))
    {
        BinaryFormatter bf = new BinaryFormatter();
        using (FileStream file = File.Open(filePath, FileMode.Open))
        {
            PlayerData data = (PlayerData)bf.Deserialize(file);
            Debug.Log($"Loaded: {data.playerName}, Level: {data.level}, Health: {data.health}");
        }
    }
    else
    {
        Debug.Log("No binary save file found.");
    }
}
```

```
void SaveData()
{
    PlayerData data = new PlayerData();
    data.level = 5;
    data.health = 75.5f;
    data.playerName = "Student";

    BinaryFormatter bf = new BinaryFormatter();
    using (FileStream file = File.Create(filePath))
    {
        bf.Serialize(file, data);
    }

    Debug.Log("Data saved in binary format at: " + filePath);
}
```





## Бонус, бинарное

### Минусы

1. Нечитаемость – нельзя просто открыть файл и посмотреть содержимое
2. Бинарные данные могут зависеть от платформы или структуры классов
3. Сложнее отлаживать – сложно проверять или исправлять данные вручную
4. BinaryFormatter считается небезопасным, лучше использовать System.IO + сериализацию вручную или сторонние библиотеки вроде MessagePack

<https://github.com/MessagePack-CSharp/MessagePack-CSharp>



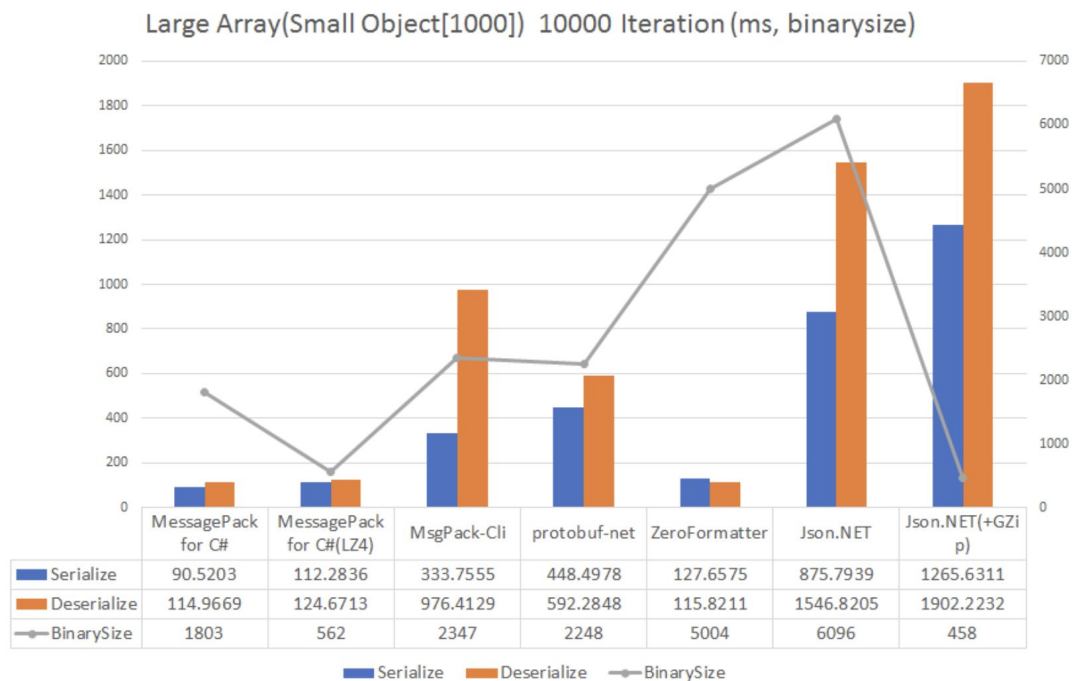
## Бонус, бинарное

### Минусы

1. Нечитаемость – нельзя просто открыть файл и посмотреть содержимое
2. Бинарные данные могут зависеть от платформы или структуры классов
3. Сложнее отлаживать – сложно проверять или исправлять данные вручную
4. BinaryFormatter считается небезопасным, лучше использовать System.IO + сериализацию вручную или сторонние библиотеки вроде MessagePack

<https://github.com/MessagePack-CSharp/MessagePack-CSharp>

## Бонус, бинарное





## Бонус, серверное

### Плюсы

1. Доступ с разных устройств – игрок может продолжить игру на другом устройстве.
2. Безопасность – сложнее обмануть или изменить данные (по сравнению с PlayerPrefs/JSON).
3. Масштабируемость – можно хранить прогресс миллионов игроков.
4. Аналитика – сервер может собирать статистику поведения игроков.

### Минусы

1. Необходим интернет – без сети сохранение не работает.
2. Сложнее реализовать – нужен сервер, API, база данных.
3. Задержки – данные нужно отправлять и получать через сеть, возможна небольшая задержка.



## Бонус, серверное

1. **Firebase Realtime Database**
2. **Supabase**
3. **Backendless**
4. **PlayFab**



## Что дальше

- Тестирование
- Дебаг
- Оптимизация