ДМИТРИЙ ДЕНИСОВ

ГОНКА КАК КОНСТРУКТОР

Собери и опубликуй свою игру на Unity

Практикум: разбираем живой проект до винтика

Играй в браузере

Дмитрий Денисов

ГОНКА КАК КОНСТРУКТОР. Собери и опубликуй свою игру на Unity

«Автор»
2026

Денисов Д. В.

ГОНКА КАК КОНСТРУКТОР. Собери и опубликуй свою игру на Unity / Д. В. Денисов — «Автор», 2026

Хотите сделать свою первую игру - и не потратить на это три года? Книга-практикум ведёт коротким путём: берём готовую казуальную гонку Pursuit (машина, бесконечная дорога, тающее топливо, встречные грузовики и гараж) и разбираем её до последнего скрипта. Пятнадцать коротких скриптов, шесть сцен - достаточно, чтобы игра была настоящей, и достаточно мало, чтобы понять каждую строчку. Вы освоите физику и контроллер машины, процедурную дорогу, спавн препятствий, интерфейс и переходы сцен; почините реальные недочёты проекта, спроектируете улучшения как гейм-дизайнер и научитесь дорабатывать игру вместе с ИИ-ассистентом. Финал - публикация в браузере и на Яндекс.Играх. Стартовые файлы проекта скачиваются там же бесплатно, а в конце книги - решебник с разборами всех заданий.

Содержание

Дмитрий Денисов
ГОНКА КАК КОНСТРУКТОР. Собери и опубликуй свою игру на Unity

Table of Contents

Введение

Эта книга — о том, как собрать свою первую гоночную игру и довести её до публикации в браузере. Не «когда-нибудь, после трёх лет обучения», а за несколько вечеров — с работающим результатом, которым можно поделиться ссылкой.

Есть распространённый миф: чтобы сделать игру, нужно сначала выучить программирование «от и до», написать всё с нуля и только потом, может быть, увидеть результат. Индустрия так не работает. Веб-разработчики опираются на фреймворки, мобильные — на готовые SDK, а в геймдеве мы используем движки, плагины и шаблоны. Игра по своей сути — это **конструктор**: вы собираете проект из деталей — сцен, префабов, материалов, скриптов и интерфейса.

Поэтому мы пойдём не «с нуля», а от готового проекта. Перед вами **Pursuit** (на заставке она представляется полным именем Petrol Pursuit, но мы для краткости будем звать её просто Pursuit) — казуальная гонка: машина мчится по бесконечной дороге, топливо тает, дорогу приходится делить с грузовиками и автобусами, а на обочинах ждут спасительные канистры. В гараже — три машины на выбор, для смелых — сложный режим со встречным движением. Игра уже работает: откройте https://sciencepub.ru/carcasual/ и проедьте пару заездов прямо в браузере. Это цель, к которой мы пойдём, — только теперь вы будете не игроком, а автором.

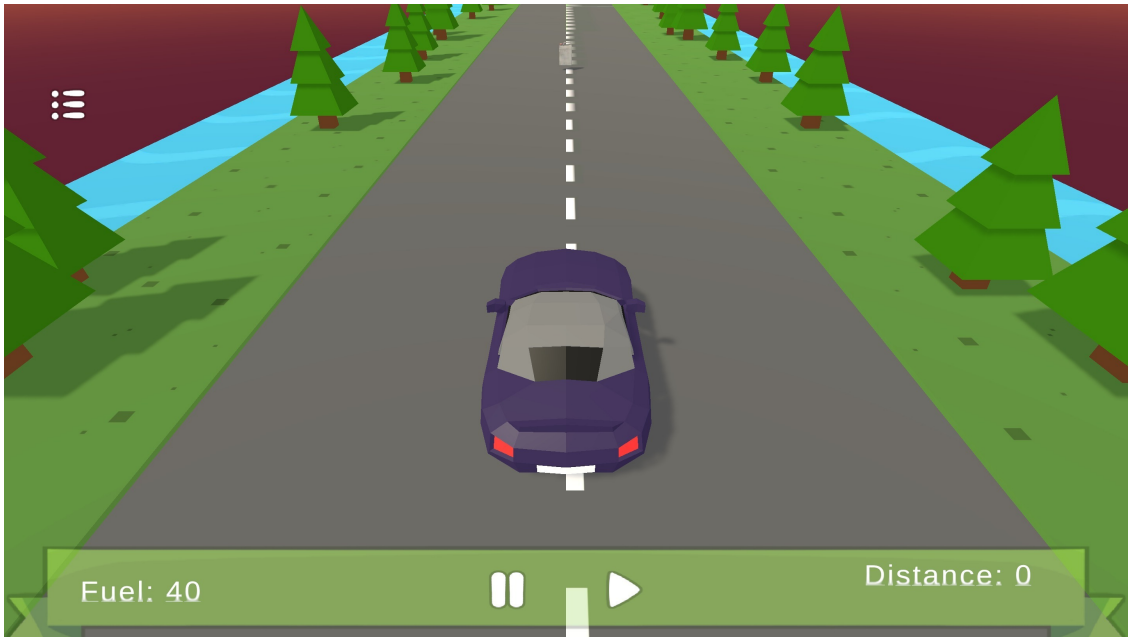

Рис. В.1. Pursuit в браузере: машина, дорога, счётчики топлива и дистанции.

Чем эта книга отличается от видеокурсов и справочников? Тем, что мы разбираем **настоящий, живой проект** — целиком, до последнего скрипта. В Pursuit всего пятнадцать коротких скриптов и шесть сцен: этого достаточно, чтобы игра ощущалась настоящей, и достаточно мало, чтобы вы поняли каждую строчку. К концу книги в проекте не останется ни одного «чёрного ящика».

Эта книга продолжает линейку практикумов «…как конструктор»: в «Шутере как конструкторе» мы собирали космический шутер Deep Space, в «Ловце Драконов» — аркаду с нуля. Читать их перед этой книгой не обязательно: каждый практикум самодостаточен. Но если вы уже прошли один из них — здесь вы увидите знакомый путь на новом жанре, и это лучший способ закрепить навык.

Вот наш маршрут:

В **первой главе** — быстрый старт: установим Unity, откроем проект и запустим игру в редакторе. Уже через час у вас на экране будет собственная работающая гонка.

Во **второй** разберём «скелет» игры: шесть сцен и переходы между ними — меню, заезд, гараж, экран поражения. Познакомимся с менеджером игры и паузой.

В **третьей** сядем за руль: разберём контроллер машины — физику, управление, расход топлива и звук двигателя. Это сердце игры.

В **четвёртой** построим бесконечную дорогу: узнаем, как игра создаёт полотно трассы впереди машины и убирает его за спиной, не съедая память.

В **пятой** выпустим на трассу препятствия: случайные машины, взрывы при столкновении и «замирание» мира в момент аварии.

В **шестой** займёмся ресурсами и интерфейсом: канистры с топливом, счётчик дистанции и передача рекорда между сценами.

В **седьмой** откроем гараж: выбор из трёх машин — и первая серьёзная самостоятельная доработка проекта.

В **восьмой** посмотрим на игру глазами гейм-дизайнера: цель, ядро геймплея, уникальность, мини-диздок и баланс — то, что превращает «работает» в «интересно».

В **девятой** займёмся развитием: сложный режим со встречным движением, рекорды, новые механики — от нитро до полиции на хвосте — и разговор о монетизации.

В **десятой** впряжём в работу нейросети: как ставить задачи ИИ-ассистенту, проверять его код и отлаживать игру вдвоём с машиной — так, чтобы ускоряться, а не терять контроль.

В **одиннадцатой** — публикация: соберём браузерную WebGL-версию, разберём плагин публикации на Яндекс.Играх и посмотрим, как игра живёт на собственном сайте с рейтингом игроков.

Важно: все материалы книги — стартовые файлы проекта Pursuit — скачиваются **бесплатно** на сайте автора. Где именно и как — подробно написано в Приложении «Материалы к книге» в конце; короткая версия: откройте https://sciencepub.ru/carcasual/ и нажмите кнопку «Стартовые файлы проекта · бесплатно».

Совет: читайте эту книгу за компьютером с открытым Unity. Практикум работает только тогда, когда вы повторяете шаги руками: глазами код не выучивается.

Игра, которую мы разбираем, намеренно написана просто — так, как пишут первые проекты. Кое-где вы встретите решения, которые опытный разработчик сделал бы иначе, — и мы будем честно на них останавливаться: почему так написано, чем это грозит и как улучшить. Умение читать и дорабатывать чужой несовершенный код — это, пожалуй, самый востребованный навык программиста вообще.

Поехали!

Глава 1. Быстрый старт: Unity и первый запуск

Дорога начинается не с первого километра, а с поворота ключа.

Введение

Задача этой главы — максимально короткая: к её концу игра должна работать у вас на компьютере. Мы установим Unity, откроем проект Pursuit, дождёмся импорта и нажмём Play. Никакой теории сверх необходимого — всё интересное начнётся, когда перед глазами будет живая игра, которую можно «пощупать».

1.1 Устанавливаем Unity Hub и редактор

Unity ставится в два шага: сначала **Unity Hub** — небольшая программа-менеджер, которая управляет версиями редактора и списком проектов, затем через неё — сам редактор.

Скачайте Unity Hub с официального сайта https://unity.com/download и установите его.

Запустите Hub и войдите в аккаунт Unity (или создайте новый — это бесплатно). Для некоммерческого обучения и небольших проектов действует бесплатная лицензия Unity Personal.

Во вкладке **Installs** нажмите **Install Editor** и выберите версию ветки **Unity 6 LTS** — проект из этой книги собран на версии 6000.3.18f1. Подойдёт и более свежая версия 6000.3.x.

В списке модулей отметьте **WebGL Build Support** — он понадобится в одиннадцатой главе для сборки браузерной версии. Остальные модули можно не ставить.

Совет: установка редактора занимает несколько гигабайт и заметное время. Запустите её и налейте чаю — торопить этот процесс бесполезно.

1.2 Открываем проект

Проект Pursuit распространяется архивом (как вы его получили — по ссылке из книги или купив готовый проект — не важно, содержимое одно и то же).

Распакуйте архив в удобную папку. Надёжнее всего — путь без кириллицы и пробелов, например C:\Games\Pursuit или ~/Games/Pursuit.

В Unity Hub во вкладке **Projects** нажмите **Add → Add project from disk** и укажите распакованную папку. Правильная папка — та, внутри которой лежат Assets, Packages и ProjectSettings.

Кликните по проекту в списке. Первый запуск идёт несколько минут: Unity строит служебную папку Library и импортирует все ассеты. Это разовая операция — дальше проект будет открываться быстро.

Важно: папки Library, Temp и Logs — это кэш. Их не бывает в архиве, их не нужно копировать при переносе проекта и не нужно жалеть при удалении: Unity пересоздаст их автоматически.

1.3 Экскурсия по редактору

Рис. 1.1. Редактор Unity с проектом Pursuit: слева Hierarchy, в центре Scene с выделенной машиной, справа Inspector с её компонентами, внизу Project.

Первое впечатление от Unity у всех одинаковое: слишком много окон, и все чем-то заняты. Хорошая новость — рабочих окон всего пять, и каждое отвечает ровно на один вопрос. Пять минут ориентирования сейчас сэкономят вам полчаса растерянности потом.

Hierarchy (слева) — *что есть в сцене*. Обычный список всех объектов открытой сцены: камера, свет, дорога, машина, интерфейс. Треугольники слева от имён разворачивают вложенность: объекты в Unity живут семьями — родитель и дети.

Scene (в центре) — *взгляд режиссёра*. Свободная камера, которой можно облететь мир с любой стороны, выделить объект и подвинуть его мышкой. Здесь вы собираете сцену; игрок этого вида никогда не увидит.

Game (вкладка рядом со Scene) — *взгляд игрока*: ровно то, что видит игровая камера. Именно эта картинка появится в браузере или на телефоне.

Inspector (справа) — *настройки выделенного*. Выделили объект в Hierarchy — справа выпал список его компонентов и всех их полей. Здесь живут те самые значения, которые побеждают код: если в скрипте написано одно, а в инспекторе выставлено другое, работать будет инспектор. Мы будем возвращаться к этому правилу всю книгу.

Project (внизу) — *все файлы игры*: сцены, скрипты, модели, звуки, префабы. Это проводник по папке Assets, только внутри редактора.

Есть и шестое окно, которое стоит открыть сразу же: **Console** (**Window → General → Console**). Это место, куда игра жалуется — ошибки компиляции, предупреждения и ваши собственные сообщения. Пока в консоли красная строка, кнопка Play работать не будет, поэтому привычка заглядывать туда первым делом окупается с первой же ошибки.

Как летать в окне Scene. Колесо мыши — зум. Зажатая правая кнопка — поворот взгляда, и, не отпуская её, можно лететь клавишами WASD, как в шутере. Выделили объект и потеряли его из виду — нажмите **F**, камера подлетит вплотную. **Alt** с зажатой левой кнопкой — облёт выделенного по орбите. Всё, этого набора хватит на всю книгу.

Теперь — живой пример, чтобы почувствовать, как окна связаны между собой. Откройте в окне Project сцену Assets/Scenes/Game.unity.

В **Hierarchy** разверните объект GameWorld и кликните по newRoads. В **Inspector** появится компонент **Road Controller** — генератор дороги, а в нём числа: Road Length 14, Despawn Offset 30, Spawn Offset 42, Despawn Distance 56. Дорога в Pursuit бесконечная, и вот эти четыре значения задают, какими кусками она склеивается. В четвёртой главе мы разберём их по косточкам.

Разверните объект carPrefab и кликните по player_purple — это машина игрока. В инспекторе выстроился её «паспорт»: **Rigidbody** (физическое тело — масса, торможение, ограничения), **Box Collider** (невидимая коробка, которой машина касается мира), **Car Controller** (управление и топливо) и **Obstacle (Script)** (обработка аварии; да, имя странное — это отдельная история из пятой главы).

Нажмите **F** — окно Scene подлетит к машине. Переключитесь на вкладку **Game** и обратно на **Scene**: одна и та же сцена, два разных взгляда.

Вот и весь приём осмотра: *выделил в Hierarchy — прочитал в Inspector*. Им мы будем пользоваться в каждой главе, разбирая чужой проект по частям.

Совет: окна редактора можно таскать за заголовки, как вкладки браузера, — собирайте раскладку под себя, это нормально. Если в процессе всё разъехалось и половина окон потерялась, есть кнопка «как было»: **Window → Layouts → Default**. Ничего в проекте при этом не ломается — раскладка окон и содержимое игры никак не связаны.

Задание: откройте сцену Menu и найдите в Hierarchy объект ButtonPlay (внутри него — дочерний play, это просто надпись-картинка). Выделите ButtonPlay и найдите в инспекторе компонент **Button**, а в нём — блок **On Click ()**. Пока просто посмотрите, что там лежит: во второй главе мы выясним, почему кнопка Play дёргает вовсе не тот скрипт, который можно было бы ожидать.

1.4 Что лежит в проекте

Теперь заглянем в окно **Project** поближе — это файловая система игры. Пройдёмся по главным папкам внутри Assets:

Scenes — шесть сцен игры, от меню до экрана поражения; их карту мы разберём во второй главе.

Scripts — вся логика игры: пятнадцать коротких скриптов на C#. К концу книги вы будете знать каждый из них в лицо.

Player, Road, Obstacle — префабы и модели машин, дороги и препятствий.

Meshtint Free City Lite Pack Mega Toon, 48 Particle Effect Pack, 2D Casual UI, Vehicle_Essentials — готовые наборы: модели города, эффекты взрывов, элементы интерфейса и большая библиотека автомобильных звуков (лежит про запас — в сценах пока звучит один carSound.wav из папки Audio). Типичный пример «конструкторного» подхода: готовые паки снимают вопрос арта, чтобы сосредоточиться на механике.

Audio — звуки: двигатель, меню, столкновение.

YandexGame и WebGLTemplates — плагин PluginYG для публикации на Яндекс.Играх; о нём поговорим в одиннадцатой главе.

Editor — служебный скрипт сборки WebGL (тоже дождётся одиннадцатой главы).

1.5 Первый заезд

Пора за руль.

В окне **Project** откройте Assets/Scenes/Menu.unity двойным кликом.

Нажмите кнопку **Play** над сценой. Появится главное меню игры (рис. 1.2).

Кнопка **Play** в меню запускает заезд. Управление — стрелки или WASD: вверх — газ, влево и вправо — перестроение между полосами. Следите за счётчиком топлива: он тает на ходу, а пополняется канистрами, которые появляются на дороге.

Врезались в машину или сожгли всё топливо — увидите экран Game Over с пройденной дистанцией. Это и есть ваш счёт.

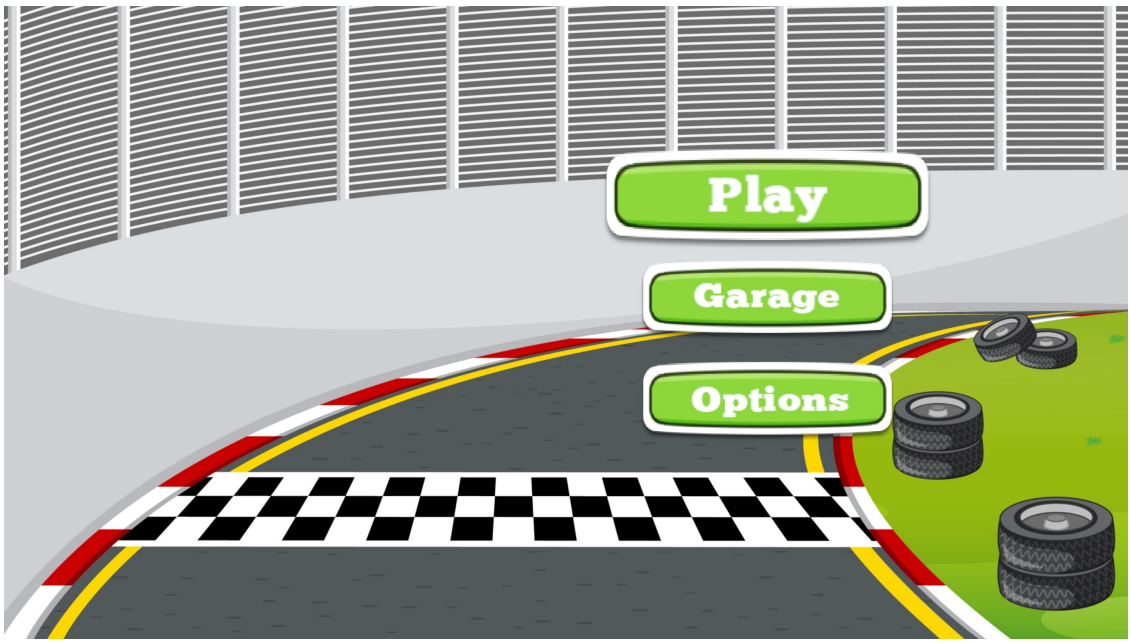

Рис. 1.2. Главное меню Pursuit: Play, Garage, Options.

Задание: сыграйте три заезда и запишите лучшую дистанцию. А теперь загляните в гараж (кнопка **Garage** в меню) и попробуйте другую машину. Заодно найдите в игре сложный режим — и почувствуйте разницу.

Важно: остановить игру в редакторе — повторное нажатие **Play**. Пока игра запущена, кнопка подсвечена, а любые правки в сцене живут только до остановки — Unity откатит их. Это классическая ловушка новичка: полчаса настраивать сцену в режиме Play и потерять всё при остановке.

Выводы

Unity ставится через Unity Hub; для этой книги нужна ветка Unity 6 LTS и модуль WebGL Build Support.

Проект открывается через Add project from disk; правильная папка содержит Assets, Packages, ProjectSettings.

Library — пересоздаваемый кэш, его не копируют и не хранят в архивах.

Рабочих окон редактора всего пять: Hierarchy (что есть в сцене), Scene (взгляд режиссёра), Game (взгляд игрока), Inspector (настройки выделенного), Project (файлы игры); шестое — Console — открывается через Window → General → Console.

Вся игра — шесть сцен и пятнадцать скриптов в Assets/Scenes и Assets/Scripts; арт и эффекты — из готовых наборов.

Запуск и остановка игры в редакторе — кнопка Play; правки в режиме Play не сохраняются.

Итог: игра запускается у вас на компьютере — значит, инструмент настроен и можно разбираться, как всё устроено. В следующей главе мы посмотрим на «скелет» Pursuit: как шесть сцен связаны в единое целое и кто управляет переходами между ними.

Глава 2. Скелет игры: сцены и переходы

Игра — это не одна комната, а квартира. Главное — знать, где двери.

Введение

Прежде чем лезть в физику машины и генерацию дороги, разберёмся, из каких «комнат» состоит Pursuit и как игрок между ними перемещается. В Unity такие комнаты называются **сценами**: каждая — отдельный мир со своими объектами, камерой и интерфейсом. Понимание карты сцен — это половина понимания архитектуры любой небольшой игры.

2.1 Шесть сцен Pursuit

Откройте папку Assets/Scenes:

Menu — главное меню: кнопки Play, Garage, Options.

Game — обычный заезд: весь трафик едет попутно.

HardGame — сложный режим: к попутному трафику добавляется встречный.

Garage — выбор одной из трёх машин.

GameOver — экран поражения с пройденной дистанцией.

SampleScene — пустая сцена-заготовка из шаблона Unity; в игре не используется, её можно смело игнорировать (или удалить — это ни на что не повлияет).

Схема переходов проста: из меню — в заезд или гараж; из гаража — в заезд; из заезда — на экран поражения (или через паузу назад в меню); с экрана поражения — снова в меню. Круг замкнулся.

2.2 Кто переключает сцены: MenuController

За навигацию из меню отвечает крошечный скрипт MenuController.cs — он висит на объекте SceneManager сцены Menu:

```
using UnityEngine;

using UnityEngine.SceneManagement;


public class MenuController : MonoBehaviour

{

public void PlayGame()

{

SceneManager.LoadScene("Game"); // загружаем сцену "Game"

}


public void OpenGarage()

{

SceneManager.LoadScene("Garage"); // загружаем сцену "Garage"

}

public void OpenMenu()

{

SceneManager.LoadScene("Menu"); // загружаем сцену "Menu"

}

}
```

Вся магия — в одной строке: SceneManager.LoadScene("имя") выгружает текущую сцену и загружает названную. Обратите внимание: сцена вызывается **по имени**. Отсюда два практических следствия. Первое — имена сцен нельзя менять бездумно: строка "Game" в коде не узнает, что файл переименован. Второе — каждая сцена должна быть добавлена в список сборки (**File → Build Profiles → Scene List**), иначе при загрузке игра «не найдёт дверь».

А как кнопки узнают об этих методах? Выделите кнопку в сцене и посмотрите в инспекторе на блок **On Click ()** компонента Button: туда перетаскивается объект с нужным скриптом, а в выпадающем списке выбирается метод. Это стандартный приём UI в Unity: кнопка — отдельно, логика — отдельно, связываются они в инспекторе.

И вот тут проект преподносит первый сюрприз — посмотрите внимательно, *что именно* подключено к кнопкам меню. Кнопка ButtonGarage честно вызывает MenuController.OpenGarage. А вот кнопка Play идёт **в обход** MenuController: в её On Click перетащен объект GarageManager (да, он есть и в сцене меню) с методом StartGame — тем самым, который мы разберём в главе про гараж: он проверяет, какая машина выбрана, и только потом загружает заезд. Метод MenuController.PlayGame(), который мы только что прочитали, **не вызывается ниоткуда** — это мёртвый код, оставшийся от ранней версии игры. А кнопка ButtonOptions и вовсе без обработчика: нажимайте сколько угодно — ничего не произойдёт. Такие «окаменелости» есть почти в любом реальном проекте; уметь их замечать — часть профессии.

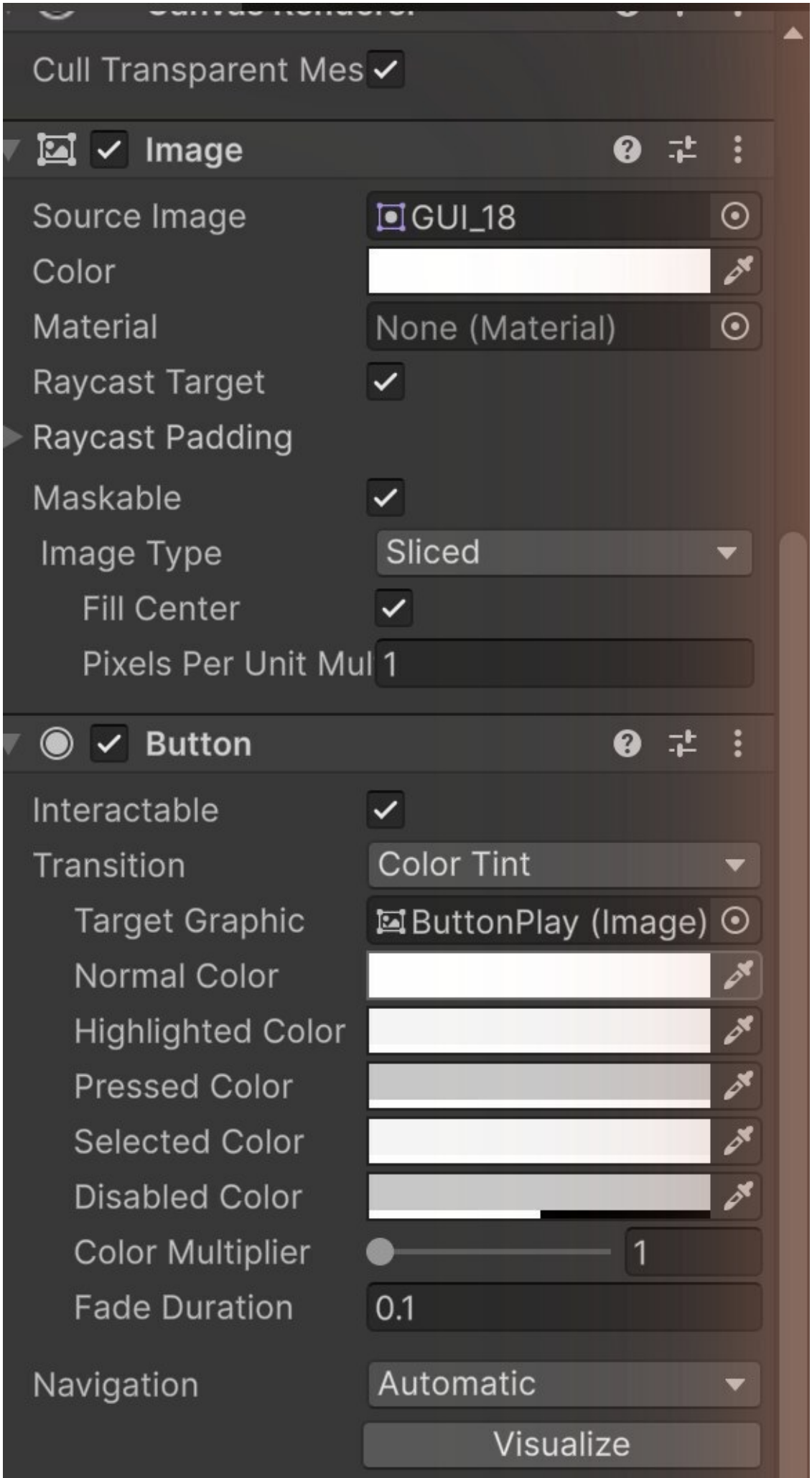
Cull Transparent Mes
Image
Source Image
GUI_18
Color
Material
None (Material)
Raycast Target
Raycast Padding
Maskable
Image Type
Sliced
Fill Center
Pixels Per Unit Mul
1
Button
Interactable
Transition
Color Tint
Target Graphic
ButtonPlay (Image)
Normal Color
Highlighted Color
Pressed Color
Selected Color
Disabled Color
Color Multiplier
1
Fade Duration
0.1
Navigation
Automatic
Visualize

Рис. 2.1. Инспектор кнопки ButtonPlay: в блоке On Click () вместо MenuController — GarageManager.StartGame.

Задание: добавьте в меню четвёртую кнопку — «Hard», — которая запускает сложный режим. Вам понадобится новый метод в MenuController с LoadScene("HardGame") и копия кнопки Play с новым обработчиком On Click — на этот раз подключите её к MenuController сами, как положено. А если хотите размяться — оживите заодно кнопку Options: пусть она хотя бы открывает гараж.

2.3 Менеджер игры и пауза

В сцене заезда живёт второй управляющий скрипт — GameManager.cs. Посмотрим на его ключевые части:

```
public class GameManager : MonoBehaviour

{

public static GameManager instance;


public Button PauseButton;

public Button ResumeButton;

public Button ExitButton;

private bool isPaused = false;


private void Awake()

{

instance = this;

}


public void PauseGame()

{

Time.timeScale = 0;

isPaused = true;

PauseButton.gameObject.SetActive(false);

ResumeButton.gameObject.SetActive(true);

}


public void ResumeGame()

{
```

```
Time.timeScale = 1;

isPaused = false;

PauseButton.gameObject.SetActive(true);

ResumeButton.gameObject.SetActive(false);

}

}
```

Здесь три важных приёма.

Синглтон. Строка public static GameManager instance и присваивание instance = this в Awake() — это упрощённый паттерн «одиночка»: любой скрипт игры может обратиться к менеджеру через GameManager.instance, не разыскивая его по сцене. Для маленькой игры — то, что нужно.

Пауза через время. Time.timeScale = 0 останавливает игровое время целиком: физика замирает, Time.deltaTime становится нулём, всё, что двигалось, — останавливается. timeScale = 1 — обычная скорость. Одна строка — и честная пауза готова.

Важно: Time.timeScale — глобальная настройка, она переживает загрузку сцен. Поэтому в методах перехода (GameOver(), ExitMenu()) менеджер обязательно возвращает timeScale = 1: забудете — и следующая сцена встретит вас «замороженной». Это реальная ошибка, на которую натыкается почти каждый новичок.

2.4 Кто связывает кнопки: GameUI

В проекте есть и второй способ связать кнопки с логикой — кодом. Для этого написан скрипт GameUI.cs:

```
public class GameUI : MonoBehaviour

{

private void Awake()

{

GameManager gameManager = GameManager.instance;

Transform gameUI = transform;

gameManager.PauseButton = gameUI.Find("Button_Pause").GetComponent<Button>();

gameManager.ResumeButton = gameUI.Find("Button_Resume").GetComponent<Button>();

gameManager.ExitButton = gameUI.Find("Button_Exit").GetComponent<Button>();


// Настроить обработчики событий для кнопок

gameManager.PauseButton.onClick.AddListener(gameManager.PauseGame);

gameManager.ResumeButton.onClick.AddListener(gameManager.ResumeGame);

gameManager.ExitButton.onClick.AddListener(gameManager.ExitMenu);

}

}
```

transform.Find("имя") ищет дочерний объект по имени, а onClick.AddListener(метод) подписывает метод на нажатие — то же самое, что блок On Click в инспекторе, только из кода. Кодовый способ удобнее, когда кнопок много или интерфейс создаётся на лету.

А теперь — вторая окаменелость этой главы, и она выразительнее первой. Проверьте, на каком объекте висит GameUI.cs. Ответ: **ни на каком** — во всём проекте нет ни одного объекта с этим компонентом. Кнопки паузы в сцене заезда называются Pause, Resume и ExitMenu и давно подключены к GameManager *через инспектор*, как и кнопки меню. А скрипт ищет объекты Button_Pause, Button_Resume, Button_Exit — таких имён в сцене больше нет: повесь его сейчас — он упадёт с NullReferenceException прямо в Awake. Судя по всему, автор начал с кодовой привязки, потом переподключил кнопки мышкой и переименовал их, а скрипт остался лежать. Урок тот же, что с PlayGame(): *наличие кода в проекте ещё не значит, что он работает* — проверяйте, кто и откуда его вызывает.

Совет: поиск по строковому имени (Find("Button_Pause")) хрупок так же, как загрузка сцены по имени: переименуете кнопку — получите ошибку уже во время игры, а не при компиляции. Привыкайте читать такие строки как «места повышенного риска» проекта.

Выводы

Игра состоит из шести сцен; переходы между ними — SceneManager.LoadScene("имя").

Все используемые сцены должны быть в списке сборки (File → Build Profiles → Scene List).

GameManager — синглтон: доступен из любого скрипта через GameManager.instance.

Пауза — это Time.timeScale = 0; не забывайте возвращать 1 перед сменой сцены.

Кнопки в этом проекте связаны с логикой через On Click в инспекторе; кодовый способ (AddListener) показан в GameUI.cs — скрипте-окаменелости, который никуда не повешен.

Итог: у игры есть скелет — сцены и переходы, менеджер и пауза. Пора нарастить на него мышцы: в следующей главе мы разберём главный скрипт игры — контроллер машины — и узнаем, как «оживить» кусок пластика с колёсами.

Глава 3. Сердце игры: контроллер машины

Машина в игре не едет — это мир договорился делать вид, что она едет.

Введение

CarController.cs — самый большой и самый важный скрипт проекта: он читает ввод игрока, двигает машину, тратит и пополняет топливо, считает дистанцию и управляет звуком двигателя. Сто тридцать строк — и в них вся суть жанра. В этой главе мы разберём его целиком, по кусочкам. Заодно познакомимся с тремя китами Unity-скриптинга: жизненным циклом Start/Update, физикой Rigidbody и системой ввода.

3.1 Поля: состояние машины

```
public class CarController : MonoBehaviour
{
// Компонент Rigidbody для моделирования физики машины
private Rigidbody rb;

// Скорость движения вперед
public float forwardSpeed = 10.0f;

// Ограничение движения машины по оси Z
private float zLimitMin = 1.25f;
private float zLimitMax = 4.75f;

// Количество бензина в автомобиле
private float fuelAmount = 40f;

// Объекты для вывода топлива и дистанции на экран
public TMPro.TextMeshProUGUI fuelText;
public TMPro.TextMeshProUGUI distanceText;

private Vector3 startPosition;
private float distanceTraveled = 0f;
private bool isMoving = false;
private bool isGameOver = false;
private AudioSource carSound;
```

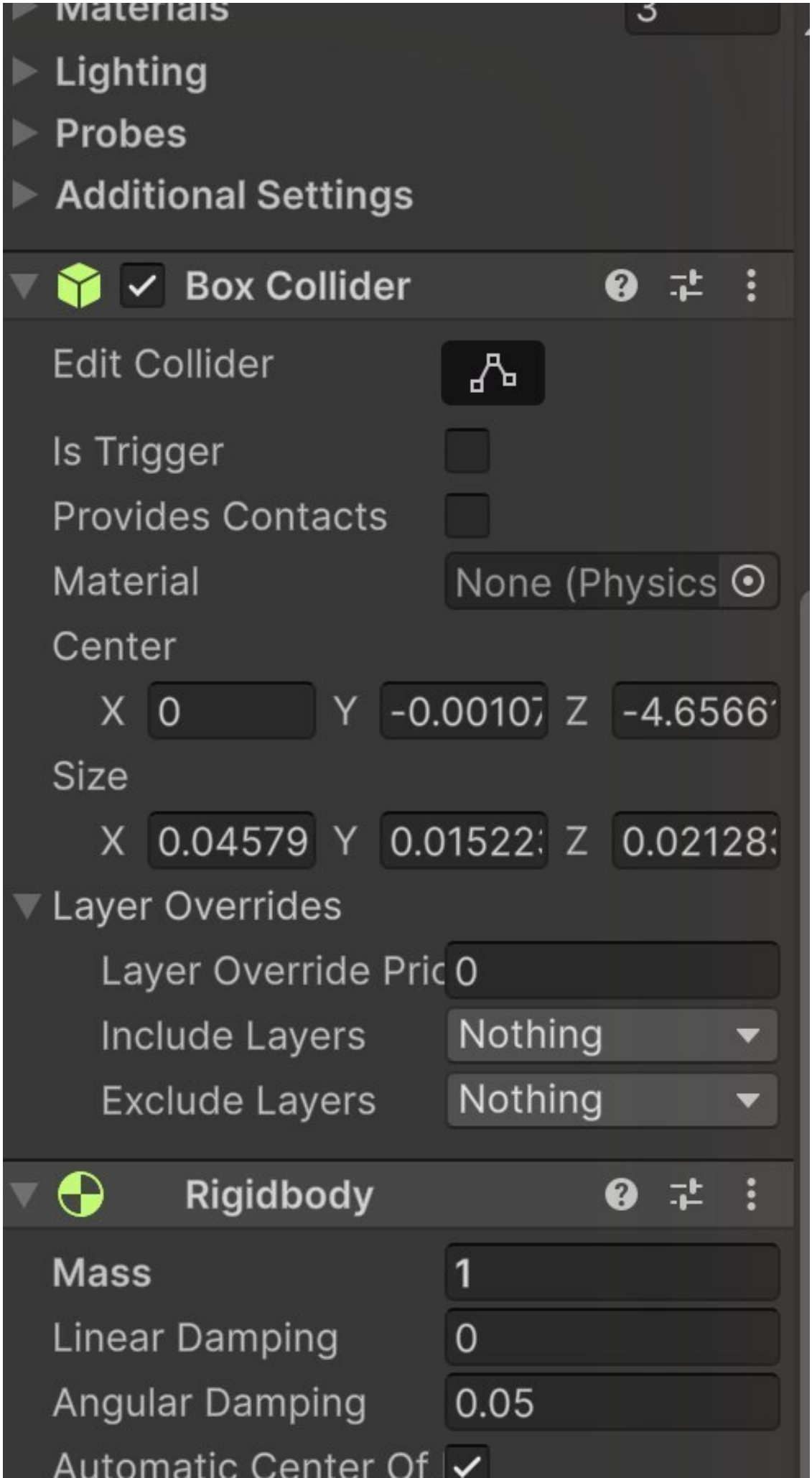
Materials
3
Lighting
Probes
Additional Settings
Box Collider
Edit Collider
Is Trigger
Provides Contacts
Material
None (Physics
Center
X 0
Y -0.00107
Z -4.6566
Size
X 0.04579
Y 0.01522
Z 0.02128
Layer Overrides
Layer Override Pric 0
Include Layers
Nothing
Exclude Layers
Nothing
Rigidbody
Mass
1
Linear Damping
0
Angular Damping
0.05
Automatic Center Of

Рис. 3.1. «Паспорт» машины игрока в инспекторе: Box Collider, Rigidbody (Collision Detection — Continuous Dynamic) и Car Controller с полями скорости и ссылками на HUD.

Обратите внимание на разницу между public и private. Публичные поля Unity показывает в инспекторе — их можно крутить, не трогая код: скорость forwardSpeed и ссылки на текстовые поля HUD назначаются прямо в сцене. Приватные — внутреннее состояние: запас топлива, пройденный путь, флаги.

Здесь же — первая особенность проекта, о которой стоит сказать честно. Машина в Pursuit едет **вдоль оси X** мира, а **ось Z** используется для перестроений поперёк дороги — «полосы» лежат между zLimitMin и zLimitMax. Привычнее было бы ехать вдоль Z («вперёд» в терминах Unity), но авторы выбрали X — и весь остальной код (дорога, препятствия, канистры) следует этому соглашению. И ещё одна оговорка: размеченных «полос» в игре на самом деле нет — есть непрерывный коридор между zLimitMin и zLimitMax, просто игрок читает его как полосы. Мы тоже будем говорить «полоса» для краткости. Для нас важно одно: **во всём проекте «вперёд» — это +X, «поперёк» — Z**. Запомните, дальше это встретится не раз.

3.2 Start: подготовка

```
void Start()

{

rb = GetComponent<Rigidbody>();

fuelText.text = "Fuel: " + fuelAmount;

distanceText.text = "Distance: " + distanceTraveled;

startPosition = transform.position;


// Получение ссылки на компонент AudioSource

carSound = GameObject.Find("CarSound").GetComponent<AudioSource>();

}
```

Start() вызывается один раз, когда объект появляется в сцене, — идеальное место для подготовки. Здесь скрипт берёт ссылку на собственный Rigidbody (компонент физики — о нём ниже), выводит стартовые значения на экран и находит в сцене объект CarSound со звуком двигателя.

Важно: GameObject.Find("CarSound") ищет объект по имени во всей сцене. Если объекта нет или он переименован — вернётся null, и первое же обращение к carSound уронит скрипт с ошибкой NullReferenceException. Такие ошибки — главный источник загадочных поломок в Unity-проектах; научиться читать их в консоли важнее, чем выучить любой API.

3.3 FixedUpdate: движение

Основная логика живёт не в Update(), а в FixedUpdate():

```
void FixedUpdate()

{

// Получение значений осей ввода

float horizontalAxis = Input.GetAxis("Horizontal");

float verticalAxis = Input.GetAxis("Vertical");


// Расчёт движения машины

Vector3 movement = new Vector3(verticalAxis, 0.0f, -horizontalAxis)

* forwardSpeed * Time.deltaTime;


// Ограничение движения по оси Z (полосы дороги)

float newPosZ = Mathf.Clamp(transform.position.z + movement.z,

zLimitMin, zLimitMax);

movement.z = newPosZ - transform.position.z;


// Определение наличия движения

isMoving = (movement != Vector3.zero);


// Применение движения к Rigidbody машины

rb.MovePosition(rb.position + transform.TransformDirection(movement));
```

Разберём по строкам.

Почему FixedUpdate? Update() вызывается каждый кадр — то 30, то 144 раза в секунду, как повезёт. Физика же в Unity считается с фиксированным шагом (по умолчанию 50 раз в секунду), и всё, что двигает Rigidbody, положено делать в FixedUpdate() — иначе движение станет дёрганым и непредсказуемым.

Оси ввода. Input.GetAxis("Vertical") возвращает число от до : стрелка вверх или W — плюс, вниз или S — минус. То же с "Horizontal" для стрелок влево/вправо. Ввод плавный: при нажатии значение нарастает постепенно, поэтому машина трогается и останавливается мягко, без рывка.

Сборка вектора. Вертикальная ось идёт в компоненту X (вперёд/назад — помните наше соглашение?), горизонтальная — со знаком минус в Z (перестроение). Умножение на forwardSpeed задаёт темп, а на Time.deltaTime — делает движение независимым от частоты кадров: «десять метров в секунду», а не «десять метров за кадр».

Клетка для машины. Mathf.Clamp(значение, мин, макс) зажимает координату Z в пределах дороги: как бы игрок ни давил на стрелку, машина не вылетит на обочину. Приём «посчитай желаемую позицию — зажми — вычти обратно» встречается в играх постоянно.

Движение через физику. rb.MovePosition(...) — это просьба к физическому движку: «передвинь тело сюда, учитывая столкновения». Если на пути стоит препятствие с коллайдером, физика это заметит — в отличие от прямой записи в transform.position, которая просто телепортирует объект сквозь стены.

3.4 Топливо: расход и канистры

```
// Проверка на столкновение с канистрой бензина

Collider[] fuelCans = Physics.OverlapSphere(transform.position, 1.0f);

foreach (Collider fuelCan in fuelCans)

{

if (fuelCan.CompareTag("FuelCan"))

{

fuelAmount += 10f;

fuelText.text = "Fuel: " + fuelAmount;

Destroy(fuelCan.gameObject);

}

}


// Уменьшение количества бензина только при наличии движения

if (isMoving)

{

fuelAmount -= Time.deltaTime;

fuelText.text = "Fuel: " + Mathf.Round(fuelAmount).ToString();

}
```

Сбор канистр устроен через Physics.OverlapSphere: вокруг машины мысленно надувается сфера радиусом один метр, и метод возвращает все коллайдеры, попавшие внутрь. Дальше — проверка **тега**: у префаба канистры в инспекторе выставлен тег FuelCan, и только такие объекты дают +10 топлива и исчезают (Destroy).

Расход устроен ещё проще: пока машина движется, топливо убывает на Time.deltaTime — то есть ровно на единицу в секунду. Стоишь — не тратишь. Отсюда простая математика баланса: стартовых 40 единиц хватает на 40 секунд езды, каждая канистра дарит ещё 10.

Совет: теги назначаются в инспекторе (выпадающий список Tag над именем объекта) и сравниваются методом CompareTag — он быстрее и безопаснее, чем сравнение строк fuelCan.tag == "FuelCan".

3.5 Конец заезда и счёт

```
// Проверка на окончание бензина
if ((fuelAmount <= 0 || isGameOver) && !isGameOver)
{
isGameOver = true;
carSound.Stop();

if (fuelAmount <= 0)
{
SceneManager.LoadScene("GameOver");
}
}

// Увеличение пройденного расстояния
if (isMoving)
{
Vector3 currentPos = transform.position;
float dx = currentPos.x - startPosition.x;
if (dx >= 0)
{
distanceTraveled += dx;
}
startPosition = currentPos;
distanceText.text = "Distance: " + Mathf.Round(distanceTraveled).ToString();
}
```

Присмотритесь к условию (fuelAmount <= 0 || isGameOver) && !isGameOver: оно избыточно — вторая половина всегда отсекает случай isGameOver, так что добавка || isGameOver в первой не делает ничего. Работает? Да. Нужно? Нет. Это ровно тот тип мест, о которых мы говорили во введении: «так написано — чем грозит — как улучшить». Здесь не грозит ничем, просто читается тяжелее, чем могло бы, — смело упрощайте до if (fuelAmount <= 0 && ! isGameOver).

Дистанция считается «шагами»: каждый кадр скрипт смотрит, насколько машина продвинулась по X с прошлого кадра (dx), и прибавляет к счётчику — но только если шаг положительный. Ехать назад, чтобы «намотать» счёт, не выйдет.

А как результат попадает на экран поражения, если это другая сцена? Через PlayerPrefs — встроенное хранилище «ключ — значение», которое переживает и смену сцен, и перезапуск игры:

PlayerPrefs.SetFloat("lastPosX", distanceTraveled);

Эта строка стоит в начале FixedUpdate() и сохраняет дистанцию **пятьдесят раз в секунду**. Работает? Работает. Красиво? Не очень: правильнее сохранять один раз — в момент окончания заезда. PlayerPrefs на некоторых платформах пишет на диск, и дёргать его каждый кадр — расточительно. Возьмите на заметку как первую кандидатуру на рефакторинг.

Звук двигателя устроен незатейливо: если машина движется и звук не играет — включить, если остановилась — выключить. Плюс carSound.Stop() при окончании топлива, чтобы двигатель не «пел» на экране поражения.

Выводы

Во всём проекте «вперёд» — это ось X, «поперёк дороги» — ось Z; полосы ограничены Mathf.Clamp.

Всё, что двигает Rigidbody, живёт в FixedUpdate(); движение умножается на Time.deltaTime.

rb.MovePosition — движение с учётом физики; прямая запись в transform.position игнорирует столкновения.

Сбор предметов: Physics.OverlapSphere + проверка тега CompareTag + Destroy.

Данные между сценами передаются через PlayerPrefs; сохранять их каждый кадр — плохая привычка.

GameObject.Find по имени — хрупкое место: переименование объекта ломает игру в рантайме.

Итог: машина слушается игрока, тратит топливо и считает метры. Но ехать ей пока некуда: дорога под колёсами должна возникать из ниоткуда и исчезать за спиной. Как устроен этот фокус — в следующей главе.

Глава 4. Бесконечная дорога

Дорога бесконечна ровно настолько, насколько хватает взгляда. Дальше она не нужна.

Введение

Если бы дизайнер уровня выкладывал трассу вручную, игра закончилась бы там, где у него кончилось терпение. В раннерах поступают иначе: мир **генерируется на ходу**. Впереди машины дорога достраивается, позади — разбирается, а в памяти в каждый момент живёт лишь небольшое «окно» трассы. В этой главе мы разберём скрипт RoadController.cs, который и показывает этот фокус.

4.1 Стартовый участок и генератор

Загляните в сцену Game: под объектом GameWorld/roads лежит **статичный стартовый участок** — несколько плиток дороги, травы, деревья и «река» по краям, расставленные вручную. Это декорация первых метров заезда. А рядом висит пустой объект newRoads со скриптом RoadController — он отвечает за всё, что дальше.

```
public class RoadController : MonoBehaviour

{

private Transform playerTransform;

public GameObject roadPrefab;

public float roadLength = 50f;

public float despawnOffset = -30f;

public float spawnOffset = 80f;

public float despawnDistance = 100f;

private List<GameObject> roads = new List<GameObject>();


private void Start()

{

playerTransform = GameObject.FindGameObjectWithTag("Player").transform;

Spawn();

}
```

Знакомая картина: публичные поля-настройки, ссылка на игрока через поиск по тегу, список созданных секций. Обратите внимание на важную деталь Unity: **значения в инспекторе побеждают значения в коде**. В скрипте написано roadLength = 50, но в сцене у компонента выставлено 14 — работать будет именно 14. Дефолты в коде — лишь начальная подсказка; правда всегда в инспекторе. Забудете об этом — будете полчаса гадать, почему правка числа в коде «не работает».

4.2 Spawn: дорога из ниоткуда

```
private void Spawn()

{

GameObject gameObject = Instantiate(roadPrefab);

gameObject.transform.position = roads.Count == 0

? Vector3.zero

: roads[roads.Count - 1].transform.position + Vector3.right * roadLength;

roads.Add(gameObject);

}
```

Instantiate(префаб) — главный глагол процедурной генерации: создать копию префаба в сцене. Первая секция встаёт в ноль, каждая следующая — вплотную к предыдущей, со сдвигом на roadLength по X (Vector3.right — это (1, 0, 0); помните соглашение «вперёд = X»?). Секция — это целый префаб с полотном, обочинами и деревьями, так что одна строчка Instantiate выкладывает сразу десяток метров мира.

4.3 Update: конвейер секций

```
private void Update()

{

// Создаём новые дороги впереди

if (playerTransform.position.x >

roads[roads.Count - 1].transform.position.x - spawnOffset)

{

Spawn();

}


// Удаляем дороги, оставшиеся позади

for (int i = 0; i < roads.Count; i++)

{

if (playerTransform.position.x - roads[i].transform.position.x

> despawnDistance)

{

Destroy(roads[i]);

roads.RemoveAt(i);

i--;

}

}

}
```

Логика двухтактная, как конвейер.

Такт первый — достроить. Как только игрок приблизился к последней секции ближе, чем на spawnOffset, впереди появляется новая. Игрок никогда не видит «край мира»: дорога всегда успевает построиться за горизонтом.

Такт второй — разобрать. Секции, отставшие от игрока больше, чем на despawnDistance, уничтожаются. Здесь есть тонкость, ради которой стоит остановиться: **уда-**

ление из списка во время обхода. Когда мы делаем RemoveAt(i), все следующие элементы сдвигаются на одну позицию влево — и если просто продолжить цикл, один элемент окажется пропущенным. Строчка i-- компенсирует сдвиг. Это классическая ловушка работы со списками в любом языке, не только в C#.

Важно: без «разборки» игра рано или поздно захлебнётся: каждая брошенная секция — это десятки объектов, которые Unity продолжает обсчитывать и рисовать. Правило раннера: всё, что игрок больше не увидит, должно быть уничтожено.

Честная оговорка: листинг выше слегка причёсан. В настоящем RoadController.cs циклов уборки *два* — второй сравнивает отставание с полем despawnOffset (в коде даже отрицательным: -30). Это наследие ранней версии: в сцене у despawnOffset выставлено 30, и фактически секции убирает именно второй цикл, а первый (с порогом despawnDistance = 56) почти никогда не успевает сработать. Игре это не вредит — дорога убирается, память не течёт, — но два цикла, делающих одну работу, это классический кандидат на рефакторинг; мы вспомним о нём в девятой главе.

4.4 Настраиваем конвейер под себя

Поиграйте с параметрами в инспекторе объекта newRoads — это лучший способ почувствовать механику:

roadLength — длина секции. Должна совпадать с реальной длиной префаба, иначе в дороге появятся щели или нахлёсты.

spawnOffset — насколько заранее достраивать. Слишком мало — игрок на большой скорости увидит, как мир «рождается»; слишком много — лишние объекты в памяти.

despawnDistance — как далеко позади разбирать. Уменьшите до 20 — и, оглянувшись (сместив камеру), увидите, как мир исчезает за спиной.

Задание: временно поставьте spawnOffset = 5 и проедьте вперёд на максимальной скорости. Вы увидите «стройку мира» своими глазами. Верните значение на место — и запомните этот кадр: так выглядит внутренность любого раннера, от Subway Surfers до Temple Run.

4.5 Камера: хвост за машиной

Раз уж мы говорим о том, как мир движется вокруг игрока, — два слова о камере. За ней отвечает CameraController.cs:

```
public class CameraController : MonoBehaviour

{

public Vector3 offset;

public float smoothSpeed = 0.125f;


void FixedUpdate()

{

GameObject player = GameObject.FindGameObjectWithTag("Player");

if (player != null)

{

Transform target = player.transform;

Vector3 desiredPosition = target.position + offset;

Vector3 smoothedPosition = Vector3.Lerp(transform.position,

desiredPosition, smoothSpeed);

transform.position = smoothedPosition;


transform.LookAt(target);

}

}

}
```

Идея проста: желаемая позиция — это позиция машины плюс смещение offset (в сцене: чуть позади и выше), а Vector3.Lerp каждый такт передвигает камеру на долю smoothSpeed пути к цели: чем меньше значение, тем ленивее «хвост». LookAt доворачивает камеру на машину. И вновь правило инспектора: в коде smoothSpeed = 0.125, а в сцене выставлено **0.9** — камера держится за машиной почти жёстко. Хотите мягкую «резинку» — опустите значение до 0.1–0.2 и почувствуйте разницу.

Важно: а вот GameObject.FindGameObjectWithTag *каждый физический такт* — расточительность: поиск по всей сцене пятьдесят раз в секунду ради объекта, который никуда не девается. Правильный ход — найти игрока один раз в Start() и запомнить ссылку, как делает RoadController. Работает и так — сцена маленькая, — но привычку стоит перенять правильную.

Выводы

Бесконечный мир — это иллюзия: живёт только «окно» вокруг игрока, дальше — генерация и уборка.

Instantiate(префаб) создаёт секции на ходу; позиция каждой считается от предыдущей.

Значения в инспекторе всегда побеждают дефолты в коде.

При удалении из списка в цикле не забывайте компенсировать сдвиг индексов (i--).

Всё, что осталось позади игрока, обязано быть уничтожено — иначе память и производительность утекут.

Камера — это Lerp к позиции машины; ссылку на игрока ищите один раз в Start(), а не каждый такт.

Итог: дорога строится и разбирается сама. Но пустая трасса — это не гонка, а медитация. В следующей главе мы выпустим на дорогу трафик: случайные машины, честные столкновения и эффектные взрывы.

Глава 5. Трафик: препятствия и столкновения

Гонка без риска — это просто дорога на дачу.

Введение

Пустая трасса не создаёт игры. Игру создаёт **конфликт**: грузовики в вашем ряду, узкие просветы, цена ошибки. В этой главе мы разберём три скрипта, которые вместе отвечают за трафик: ObstacleManager расставляет машины, ObstacleMovement их двигает, а скрипт из файла Collision.cs превращает касание в аварию — со взрывом и экраном поражения.

5.1 ObstacleManager: спавн по прогрессу

```
public class ObstacleManager : MonoBehaviour

{

public GameObject[] obstaclePrefabs;

public float obstacleSpeed = 5f;

public float spawnDistance = 40f;

public float spawnOffset = 30f;

public float despawnDistance = 80f;


private GameObject player;

private float lastPlayerPosition;

private float currentDistance;


void Update()

{

currentDistance = player.transform.position.x - lastPlayerPosition;


if (currentDistance >= spawnDistance)

{

lastPlayerPosition = player.transform.position.x;


for (float spawnPosX = lastPlayerPosition + spawnOffset;

spawnPosX <= lastPlayerPosition + currentDistance;

spawnPosX += spawnDistance)

{

SpawnObstacle(new Vector3(spawnPosX, 1.4f, 1.7f));
```

```
    }


    DestroyObstacles(player.transform.position.x - despawnDistance);

    }

    }
```

Прежде чем разбирать логику — знакомая по четвёртой главе оговорка: значения в инспекторе побеждают значения в коде. В листинге spawnDistance = 40, а в сцене Game у объекта Obstacle выставлено **35** (а ещё spawnOffset 25 вместо 30 и despawnDistance 35 вместо 80) — работают именно сценовые числа, их мы и возьмём в расчёты баланса. И вторая деталь: рядом в сцене лежит *выключенный*

Конец ознакомительного фрагмента.