1 ОБЩАЯ ЧАСТЬ

* 1. Актуальность проекта

Главной задачей диплома является разработка веб приложения для проведения вебинаров**, и развертывание приложения на сервере. Приложение нацелено на привлечение клиентов для основного продукта. Основные элементы вебинара, это общение с аудиторией и сбор данных для связи с потенциальными клиентами в будущем.**

* 1. Сетевые технологии
     1. Краткие сведения о веб приложениях

Основное отличие веб приложения от сайта, это генерация контента в зависимости от пользовательского ввода (реактивность). Веб приложение часто связывается с сервером для получения и отправки данных. Это могут быть чаты, социальные сети и т.п.

Типичное веб-приложение состоит из HTML-верстки, CSS-стилей и JavaScript-кода, который позволяет достичь максимального уровня интерактивности и отзывчивости, кода выполняющего внутреннюю логику и базы данных.

Есть несколько подходов к архитектуре веб приложения. Выбор может варьироваться в зависимости от масштаба и сложности приложения.

Монолитная архитектура – это архитектурный подход, в котором вся основная логика приложения собрана в одном месте. Монолитное приложение состоит из однослойного объединения разных компонент в одно целое. Клиент получает только готовые страницы.

Сервис-ориентированная архитектура (SOA) – это полная противоположность монолитной архитектуры. Это модульный подход к разработке программного обеспечения, базирующийся на обеспечении удаленного использования по стандартизированным протоколам распределённых, слабо связанных легко заменяемых компонентов (сервисов) со стандартизированными интерфейсами.

REST (Representational state transfer) – вляется очень простым интерфейсом управления информацией без использования каких-то дополнительных внутренних прослоек. Каждая единица информации однозначно определяется глобальным идентификатором, таким как URL. Каждая URL в свою очередь имеет строго заданный формат.

Бессерверная архитектура – Новая разновидность SOA, отличается тем, что обеспечивает работу облачных сервисов по требованию. Это означает, что бессерверные функции запускаются только в момент фиксации определенного события.

1.2.2 Этапы разработки

Разработка технического задания – составление документации. на этом этапе нужно описать как будет выглядеть и работать приложение.

Создание дизайна и навигации – прорисовка дизайн макетов, разработка UI и UX дизайна, логотипа, структуры навигации и т.д.

Верстка – описание страницы на языке разметки в соответствии с дизайн макетом.

Программирование – оживление верстки с помощью JavaScript, написание внутренней логики.

Тестирование – то процесс исследования ПО с целью выявления ошибок и определения соответствия между реальным и ожидаемым поведением ПО.

Развертывание – запуск приложения на рабочем сервере, настройка конфигурации сервера, базы данных, сторонних сервисов.

1.2.3 Основные сервисы веб прложения

Frontend (фронтенд) – это сервис предоставляющий пользователю интерфейс через который он может общаться с базой другими сервисами веб приложения. Основной функционал фронтенда состоит в отправке запросов по заранее определенным стандартам (интерфейсам, протоколам) к сервисам, отвечающим за обработку данных.

Backend (бэкенд) – это может быть один монолитный сервис или кластер сервисов отвечающих за внутреннюю логику. Внутренняя логика – обработка данных поступающих с внешних ресурсов, работа с базой(-ами) данных, управление другими сервисами.

База данных – группированный набор однородной информации. В контексте применяется для хранения наборов данных авторизации, токенов.

SMTP server (почтовый сервер) – необходим для отправки писем по электронной почте.

Proxy server (сервер переадресации) – может служить для перенаправления трафика основываясь на заголовках пакета данных, либо для распределения нагрузки на сервера.

* 1. Описание стека технологий

1.3.1 Общие сведения

Стек технологий - это набор технологий на основе которых разрабатывается веб приложение. Самый распространенный стек – это HTLM, CSS, JavaScript (далее js), PHP, MySQL. При чем первые 3 присутствуют в каждом стеке, либо получаются при компиляции проекта. Так, например в React (библиотеке для js), для разметки страницы используются файлы-шаблоны с расширением «.jsx», которые в последствии компилируются в .html и .js файлы. Или препроцессоры для CSS, такие как SASS, SCSS, LESS, которые компилируются в CSS файлы.

Для своего проекта я взял архитектурный подход REST, но веб приложение не соответствует всем критериям REST. Приложение я разбил на 3 части; это пользовательский интерфейс (Frontend), внутренняя логика (Backend) и База данных. Так же я обращаюсь к сторонним сервисам, в частности это почтовый сервер и доменный сервер.

Мой стек состоит из Angular 11 для фронтенда (HTML, CSS, js), NestJs для бэкенда (внутренняя логика), MySQL – база данных. Для развертывания я использовал Docker. Почтовый сервер и доменное имя я купил на dreamhost.com, Сервер для развертывания и базу данных приобрел на digitalocean.com

1.3.2 REST архитектура

REST (Representational state transfer) – это стиль архитектуры программного обеспечения для распределенных систем, таких как World Wide Web, который, как правило, используется для построения веб-служб. Термин REST был введен в 2000 году Роем Филдингом, одним из авторов HTTP-протокола. Системы, поддерживающие REST, называются RESTful-системами.

Как было упомянуто выше – REST является очень простым интерфейсом управления информацией без использования каких-то дополнительных внутренних прослоек. Каждая единица информации однозначно определяется глобальным идентификатором, таким как URL. Каждая URL в свою очередь имеет строго заданный формат.

Как это работает.

Отсутствие дополнительных внутренних прослоек означает передачу данных в том же виде, что и сами данные. Т.е. мы не заворачиваем данные в XML, как это делает SOAP и XML-RPC, не используем AMF, как это делает Flash и т.д.

Каждая единица информации однозначно определяется URL – это значит, что URL по сути является первичным ключом для единицы данных. К примеру путь до 10 квартиры в 3-ем доме будет выглядеть как /house/3/apartment/10.

Управление информацией сервиса – целиком и полностью основывается на протоколе передачи данных. Наиболее распространенный протокол конечно же HTTP. Так вот, для HTTP действие над данными задается с помощью методов: GET (получить), PUT (добавить, заменить), POST (добавить, изменить, удалить), DELETE (удалить). Таким образом, действия CRUD (Create-Read-Updtae-Delete) могут выполняться как со всеми 4-мя методами, так и только с помощью GET и POST.

Чтобы распределенная система считалась сконструированной по REST архитектуре, необходимо, чтобы она удовлетворяла следующим критериям:

* Client-Server. Система должна быть разделена на клиентов и на серверов.
* Stateless. Сервер не должен хранить какой-либо информации о клиентах. В запросе должна храниться вся необходимая информация для обработки запроса и если необходимо, идентификации клиента.
* Cache․ Каждый ответ должен быть отмечен является ли он кэшируемым или нет.
* Uniform Interface. Универсальный интерфейс между компонентами системы
* Layered System. В REST допускается разделить систему на иерархию слоев но с условием, что каждый компонент может видеть компоненты только непосредственно следующего слоя. Например, если вы вызывайте службу PayPal а он в свою очередь вызывает службу Visa, вы о вызове службы Visa ничего не должны знать.
* Code-On-Demand. В REST позволяется загрузка и выполнение кода или программы на стороне клиента.

1.3.3 Angular 11

Angular это платформа и фреймворк для разработки одностраничных приложений с использованием TypeScript и HTML. Он включает в себя ядро и опциональные TypeScript библиотеки которые можно импортировать в приложение. Angular приложение состоит из компонентов Angular которые организованы в модули NgModules. NgModules собирают связанный код в функциональные наборы; Приложение Angular определяется набором модулей NgModules. В приложении всегда есть как минимум корневой модуль, обеспечивающий начальную загрузку, и, как правило, гораздо больше функциональных модулей.

* Компоненты определяют шаблоны (виды, представления), которые представляют собой наборы элементов экрана, которые Angular может выбирать и изменять в соответствии с логикой и данными программы.
* Компоненты используют службы, которые предоставляют определенные функции, не связанные напрямую с представлениями. Поставщики услуг могут быть введены в компоненты как зависимости, что сделает код модульным, многоразовым и эффективным.

1.3.3.1 Модули

1.3.3.2 Компоненты

1.3.3.3 Службы

1.3.3.4 Шаблоны

1.3.3.5 Способы структурирования программ

1.3.4Максимально допустимый временной шаг

Фиксированный временной интервал сохраняет точность физического моделирования в реальном времени, но он может вызвать проблемы в тех случаях, когда игра интенсивно использует физику, а частота кадров в игровом процессе также становится низкой (например, из-за большого количества объектов в игре). Обработка обновлений основного кадра должна быть «зажата» между регулярными обновлениями физики, и, если требуется выполнить большую обработку, в течение одного кадра может происходить несколько обновлений физики. Поскольку время кадра, положения объектов и другие свойства заморожены в начале кадра, графика может не синхронизироваться с более часто обновляемой физикой.

Естественно, есть только так много доступной мощности процессора, но Unity имеет возможность, позволяющую вам эффективно замедлять физическое время, чтобы ускорить обработку кадров. Параметр «Максимально допустимый временной шаг» (в «Диспетчере времени») ограничивает количество времени, которое Unity будет тратить на обработку физических вызовов и вызовов FixedUpdate во время обновления данного кадра. Если обновление кадра занимает больше времени, чем Максимально допустимый временной шагчтобы обработать, физический движок «остановит время» и позволит обработке кадров наверстать упущенное. Как только обновление кадра закончится, физика возобновится, как будто времени не прошло с момента его остановки. Результатом этого является то, что твердые тела не будут двигаться идеально в режиме реального времени, как это обычно происходит, но будут слегка замедлены. Однако физические «часы» все равно будут отслеживать их, как если бы они двигались нормально. Замедление времени физики обычно не заметно и является приемлемым компромиссом с производительностью геймплея.

Шкала времени.

Для специальных эффектов, таких как «время маркера», иногда полезно замедлить прохождение игрового времени, чтобы анимация и отклики сценария происходили с пониженной скоростью. Кроме того, иногда вы можете захотеть полностью заморозить игровое время, например, когда игра приостановлена. Unity имеет свойство Time Scale, которое контролирует, как быстро игровое время протекает относительно реального времени. Если масштаб установлен на 1,0, то игровое время соответствует реальному времени. При значении 2.0 время в Unity увеличивается вдвое быстрее (т.е. действие будет ускорено), а при значении 0.5 замедляется игровой процесс до половины скорости. Нулевое значение заставит время полностью остановиться. Обратите внимание, что шкала времени на самом деле не замедляет выполнение, а просто изменяет шаг по времени, сообщаемый функциям Update и FixedUpdate через Time.deltaTime и Time.fixedDeltaTime. Функция обновления, вероятно, будет вызываться чаще, чем обычно, когда игровое время замедляется, но шаг deltaTime, о котором сообщается, каждый кадр будет просто уменьшен. Шкала времени не влияет на другие функции скрипта, поэтому вы можете, например, отображать графический интерфейс с нормальным взаимодействием, когда игра находится в режиме паузы.

Time Manager имеет свойство, чтобы установить масштаб времени во всем мире, но это, как правило, более полезным, чтобы установить значение из сценария, используя Time.timeScale свойства:

//C# script example

using UnityEngine;

using System.Collections;

public class ExampleScript : MonoBehaviour {

void Pause() {

Time.timeScale = 0; }

void Resume() {

Time.timeScale = 1;

}

}

//JS script example

function Pause() {

Time.timeScale = 0;

}

function Resume() {

Time.timeScale = 1;

}

Захват кадров.

Особый случай управления временем - это когда вы хотите записать игровой процесс в виде видео. Поскольку задача сохранения изображений на экране занимает значительное время, обычная частота кадров в игре будет значительно снижена, если вы попытаетесь сделать это во время обычного игрового процесса. Это приведет к видео, которое не отражает истинную производительность игры.

К счастью, Unity предоставляет свойство Capture Framerate, которое позволяет обойти эту проблему. Если для свойства задано значение, отличное от нуля, игровое время будет замедлено, а обновления кадров будут выпускаться с точными регулярными интервалами. Интервал между кадрами равен 1 / Time.captureFramerate, поэтому, если установлено значение 5.0, обновления происходят каждую пятую секунды. С уменьшением требований к частоте кадров у вас есть время в функции обновления, чтобы сохранить скриншоты или предпринять другие действия:

//C# script example

using UnityEngine;

using System.Collections;

public class ExampleScript : MonoBehaviour {

// Capture frames as a screenshot sequence. Images are

// stored as PNG files in a folder - these can be combined into

// a movie using image utility software (eg, QuickTime Pro).

// The folder to contain our screenshots.

// If the folder exists we will append numbers to create an empty folder.

string folder = "ScreenshotFolder";

int frameRate = 25;

void Start () {

// Set the playback framerate (real time will not relate to game time after this).

Time.captureFramerate = frameRate;

// Create the folder

System.IO.Directory.CreateDirectory(folder);

}

void Update () {

// Append filename to folder name (format is '0005 shot.png"')

string name = string.Format("{0}/{1:D04} shot.png", folder, Time.frameCount );

// Capture the screenshot to the specified file.

Application.CaptureScreenshot(name);

}

}

//JS script example

// Capture frames as a screenshot sequence. Images are

// stored as PNG files in a folder - these can be combined into

// a movie using image utility software (eg, QuickTime Pro).

// The folder to contain our screenshots.

// If the folder exists we will append numbers to create an empty folder.

var folder = "ScreenshotFolder";

var frameRate = 25;

function Start () {

// Set the playback framerate (real time will not relate to game time after this).

Time.captureFramerate = frameRate;

 // Create the folder

System.IO.Directory.CreateDirectory(folder);

}

function Update () {

// Append filename to folder name (format is '0005 shot.png"')

var name = String.Format("{0}/{1:D04} shot.png", folder, Time.frameCount );

// Capture the screenshot to the specified file.

Application.CaptureScreenshot(name);

}

Хотя видео, записанное с использованием этой техники, обычно выглядит очень хорошо, в игру может быть трудно играть, когда она сильно замедлена. Возможно, вам придется поэкспериментировать со значением Time.captureFramerate, чтобы обеспечить достаточное время записи без чрезмерного усложнения задачи тестового проигрывателя.

1.3.5 Пространства имён

Когда проект становится больше, а количество классов увеличивается, вероятность совпадения имён классов также возрастает. Такое может произойти, если несколько программистов работают отдельно над различными аспектами игры. Например, один программист пишет код для управления персонажем игрока, в то время как другой пишет эквивалентный код для управления противниками. Оба программиста предпочли назвать главный класс Controller, однако при объединении их проектов это приведет к совпадению имен классов, в результате чего использование класса с таким именем оказалось бы двусмысленным.

В некоторой степени, эта проблема может быть решена путем переименования классов, когда обнаружено совпадение имен (например, классы выше могут иметь имена PlayerController и EnemyController). Однако, это весьма проблематично, когда есть несколько таких классов, или когда объявляются переменные с их использованием - каждое упоминание старого названия класса должно быть заменено, чтобы код мог скомпилироваться.

Язык C# предлагает пространства имен в качестве решения данной проблемы. Пространство имен - это просто набор классов, в котором для всех имён этих классов используется определенный префикс. В приведённом ниже примере классы Controller1 и Controller2 относятся к пространству имен Enemy:-

namespace Enemy {

public class Controller1 : MonoBehaviour {

... }

public class Controller2 : MonoBehaviour {

... }

}

В коде эти классы вызываются соответственно Enemy.Controller1 и Enemy.Controller2. Это лучше, чем переименование классов, поскольку объявление пространства имен может быть выполнено для существующих объявлений класса (то есть, нет необходимости менять имена всех классов индивидуально). Кроме того, вы можете использовать одинаковые пространства имен для классов, которые хранятся в разных исходных файлах.

Вы можете избежать многократного указания префикса пространства имен, добавив директиву using в верхней части файла.

using Enemy;

Эта строка говорит о том, что использование названия классов Controller1 и Controller2 на самом деле будет обозначать классы Enemy.Controller1 и Enemy.Controller2 соответственно. Если в скрипте предполагается использование классов с таким же именем из другого пространства имен (предположим из пространства Player), то для их вызова нужно будет указать префикс. Если имена используемых классов из двух разных пространств имен указанных в директиве совпадут - компилятор сообщит об ошибке.

Атрибуты.

Атрибуты (Attributes) это маркеры, которые могут быть помещены перед классом, свойствами или функциями в скрипте, чтобы указать особое поведение. Например атрибут HideInInspector может быть добавлен перед объявлением свойства для предотвращения отображения этого свойства в инспекторе, даже если оно публичное. В JavaScript имя атрибута начинается со знака “@”, а в C# и Boo, он помещается между квадратными скобками: -

// JS

@HideInInspector

var strength: float;

// C#

[HideInInspector]

public float strength;

Unity предоставляет ряд атрибутов, которые перечислены в справочнике по скриптам (выберите секцию атрибутов Редактора или рантайм атрибутов из выпадающего меню слева). Есть также атрибуты, определенные в .NET библиотеке, которые иногда могут быть полезны в Юнити коде.

Примечание: атрибут ThreadStatic определенный в .NET нельзя использовать, т.к. он вызовет крах Unity.

1.3.6 Порядок выполнения функций событий

В скриптинге Unity есть некоторое количество функций события, которые исполняются в заранее заданном порядке по мере исполнения скрипта. Этот порядок исполнения описан ниже:

Редактор.

Reset: Reset (сброс) вызывается для инициализации свойств скрипта, когда он только присоединяется к объекту и тогда, когда используется команда Reset.

Первая загрузка сцены.

Эти функции вызываются при запуске сцены (один раз для каждого объекта на сцене).

Awake: Эта функция всегда вызывается до любых функций Start и также после того, как префаб был вызван в сцену (если GameObject неактивен на момент старта, Awake не будет вызван, пока GameObject не будет активирован, или функция в каком-нибудь прикреплённом скрипте не вызовет Awake).   
OnEnable: (вызывается только если объект активен): Эта функция вызывается сразу после включения объекта. Это происходит при создании образца MonoBehaviour, например, при загрузке уровня или был вызван GameObject с компонентом скрипта.

OnLevelWasLoaded: This function is executed to inform the game that a new level

has been loaded.

Учтите, что для объектов, добавленных в сцену сразу, функции Awake и OnEnable для всех скриптов будут вызваны до вызова Start, Update и т.д. Естественно, для объектов вызванных во время игрового процесса такого не будет.

Перед первым обновлением кадра.

Start: Функция Start вызывается до обновления первого кадра(first frame) только если скрипт включен.

Для объектов добавленных на сцену, функция Start будет вызываться во всех скриптах до функции Update. Естественно, это не может быть обеспечено при создании объекта непосредственно во время игры.

Между кадрами.

OnApplicationPause: Эта функция вызывается в конце кадра, вовремя во время которого вызывается пауза, что эффективно между обычными обновлениями кадров. Один дополнительный кадр будет выдан после вызова OnApplicationPause, чтобы позволить игре отобразить графику, которая указывает на состояние паузы.

Порядок обновления.

Когда вы отслеживаете игровую логику и взаимодействия, анимации, позиции камеры и т.д. есть несколько разных событий, которые вы можете использовать. По общему шаблону, большая часть задач выполняется внутри функции Update, но есть также ещё другие функции, которые вы можете использовать.

FixedUpdate: Зачастую случается, что FixedUpdate вызывается чаще чем Update. FU может быть вызван несколько раз за кадр, если FPS низок и функция может быть и вовсе не вызвана между кадрами, если FPS высок. Все физические вычисления и обновления происходят сразу после FixedUpdate. При применении расчётов передвижения внутри FixedUpdate, вам не нужно умножать ваши значения на Time.deltaTime. Потому что FixedUpdate вызывается в соответствии с надёжным таймером, независящим от частоты кадров.

Update: Update вызывается раз за кадр. Это главная функция для обновлений кадров.

LateUpdate: LateUpdate вызывается раз в кадр, после завершения Update. Любые вычисления произведённые в Update будут уже выполнены на момент начала LateUpdate. Часто LateUpdate используют для преследующей камеры от третьего лица. Если вы перемещаете и поворачиваете персонажа в Update, вы можете выполнить все вычисления перемещения и вращения камеры в LateUpdate. Это обеспечит то, что персонаж будет двигаться до того, как камера отследит его позицию.

1.3.6.1 Рендеринг

OnPreCull: Вызывается до того, как камера отсечёт сцену. Отсечение определяет, какие объекты будут видны в камере.

OnPreCull вызывается прямо перед тем, как начинается отсечение.

OnBecameVisible/OnBecameInvisible: Вызывается тогда, когда объект становится видимым/невидимым любой камере.

OnWillRenderObject: Вызывается один раз для каждой камеры, если объект в поле зрения.

OnPreRender: Вызывается перед тем, как камера начнёт рендерить сцену.

OnRenderObject: Вызывается, после того, как все обычные рендеры сцены завершатся. Вы можете использовать класс GL или Graphics.DrawMeshNow, чтобы рисовать пользовательскую геометрию в данной точке.

OnPostRender: Вызывается после того, как камера завершит рендер сцены.

OnRenderImage (только в Pro версии): Вызывается после завершения рендера сцены, для возможности пост-обработки изображения экрана.

OnGUI: Вызывается несколько раз за кадр и отвечает за элементы интерфейса (GUI). Сначала обрабатываются события макета и раскраски, после чего идут события клавиатуры/мышки для каждого события.

OnDrawGizmos Используется для отрисовки гизмо в окне Scene View в целях визуализации.

Сопрограммы.

Нормальные обновления сопрограмм запускаются после завершения из функции Update. Сопрограмма это функция, которая приостанавливает своё исполнение (yield), пока данные YieldInstruction не завершатся. Разные способы использования сопрограмм:

yield Сопрограмма продолжит выполнение, после того, как все Update функции были вызваны в следующем кадре.

yield WaitForSeconds Продолжает выполнение после заданной временной задержки, и после все Update функций, вызванных в итоговом кадре.

yield WaitForFixedUpdate Продолжает выполнение после того, как все функции FixedUpdate были вызваны во всех скриптах

yield WWW продолжает выполнение после завершения WWW-загрузки.

yield StartCoroutine сцепляет сопрограмму, и будет ждать, пока не завершится сопрограмма MyFunc.

Когда объект разрушается.

OnDestroy: Эта функция вызывается после всех обновлений кадра в последнем кадре объекта, пока он ещё существует (объект может быть уничтожен при помощи Object.Destroy или при закрытии сцены).

При выходе.

Эти функции вызываются во всех активных объектах в вашей сцене:

OnApplicationQuit: Эта функция вызывается для всех игровых объектов перед тем, как приложение закрывается. В редакторе вызывается тогда, когда игрок останавливает игровой режим. В веб-плеере вызывается по закрытия веб окна.

OnDisable: Эта функция вызывается, когда объект отключается или становится неактивным.

Блок-схема жизненного цикла скрипта.

Следующая диаграмма совмещает порядок и повтор функций событий в течение жизни скрипта.

Блок схема жизненного цикла скрипта представлена на рисунке 1.2.

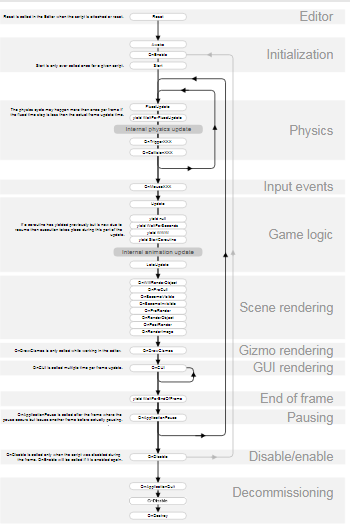


Рисунок 1.2– Блок-схема жизненного цикла скрипта

1.3.7 Общие функции

Некоторые функции в справке по скриптам (например, различные функции GetComponent) перечислены в варианте, который содержит букву T или имя типа в угловых скобках после имени функции: -

//C#

void FuncName<T>();

//JS

function FuncName.<T>(): T;

Они известны как общие функции. Их значимость для программирования заключается в том, чтобы указать типы параметров и/или возвращаемого типа при вызове функции. В JavaScript, это может быть использовано, чтобы обойти ограничения динамической типизации: -

// The type is correctly inferred since it is defined in the function call.

//In C#

var obj = GetComponent<Rigidbody>();

//In JS

var obj = GetComponent.<Rigidbody>();

В C# это может сократить количество кода: -

Rigidbody rb = go.GetComponent<Rigidbody> ();

// ...as compared with:

Rigidbody rb = (Rigidbody) go.GetComponent(typeof(Rigidbody));

Любая функция, имеющая общий вариант, который указан на своей странице справки, позволяет использовать специальный синтаксис вызова.

1.3.8 Важные классы

Это некоторые из наиболее важных классов, которые вы будете использовать при написании сценариев в Unity. Они охватывают некоторые основные области скриптовых систем Unity и являются хорошей отправной точкой для поиска доступных функций и событий.

Описание классов представлено на рисунке 1.3.

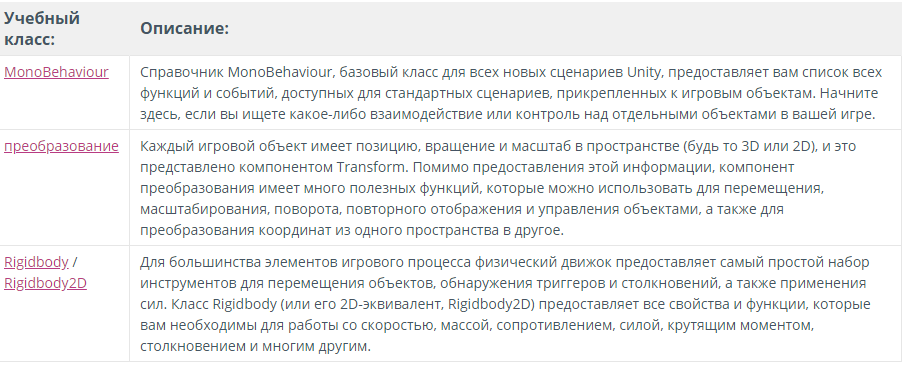


Рисунок 1.3– Описания классов

Unity события.

UnityEvents это способ позволяющий поддерживать управляемую пользователем функцию обратного вызова от момента редактирования до момента запуска без необходимости дополнительного программирования и конфигурации скриптов.

UnityEvents полезны для целого ряда вещей:

Содержит управляемую пользователем функцию обратного вызова

Системы связей

Постоянная функция обратного вызова

Предварительно настроенные события вызова

UnityEvents могут быть добавлены к любому MonoBehavior и вызываться в коде как стандартные .net делегаты. После добавления UnityEvent в MonoBehaviour, оно отобразится в инспекторе и можно будет добавлять постоянные функции обратного вызова.

UnityEvents have similar limitations to standard delegates. That is, they hold references to the element that is the target and this stops the target being garbage collected. If you have a UnityEngine.Object as the target and the native representation disappears the callback will not be invoked.

Использование UnityEvents

Вот несколько шагов по настройке обратного вызова в редакторе:

Make sure your script imports/uses UnityEngine.Events.

Выберите иконку +, чтобы добавить слот для функции обратного вызова

Выберите объект UnityEngine.Object к которому вы хотите обратиться посредством функции обратного вызова (Для этого вы можете использовать селектор объектов)

Выберите функцию, которую вы хотите вызвать

Вы можете добавить больше одной функции обратного вызова к событию

Во время настройки UnityEvent в инспекторе, вам доступно два типа поддерживаемых функций вызова:

Статические вызовы, это - пред настроенные вызовы с предустановленными значениями, которые настраиваются в UI (пользовательском интерфейсе). Это означает, что при срабатывании функции обратного вызова, целевая функция вызывается с аргументами предварительно введенными в UI. Динамические вызовы при срабатывании используют аргументы получаемые из кода, и ограничены типом вызванного UnityEvent. UI фильтрует функции обратного вызова и показывает только те динамические вызовы, которые действительны для данного UnityEvent.

Общие UnityEvents

По умолчанию UnityEvent в Monobehaviour связывает динамически функции без аргументов. Это не обязательно должно быть так, потому что UnityEvents поддерживает связь функций содержащих до 4-х аргументов. Чтобы сделать это, вам необходимо переопределить стандартный класс UnityEvents как поддерживающий множественные аргументы. Сделать это довольно просто:

[Serializable]

public class StringEvent : UnityEvent <string> {}

Потом это может быть вызвано с помощью вызова функции Invoke со ‘строкой(string)’ в качестве аргумента.

UnityEvents могут быть переопределены с наличием до 4-х аргументов в их общем определении. Добавление экземпляра этого класса к вашему, вместо базового UnityEvent, даст возможность функции обратного вызова связывать динамические строковые функции.

1.3.9 Понимание векторной арифметики

Векторная арифметика - основа 3D графики, физики и анимации, и, для получения максимальной отдачи от Unity, весьма полезно досконально разбираться в этой теме. Ниже приведены описания основных операций и несколько советов о том, для чего они могут быть использованы.

Сложение.

При сложении 2 векторов результат эквивалентен тому, что получится если исходные векторы принять за следующие друг за другом “шаги”. Заметьте, что порядок двух слагаемых не важен, т.к. в любом случае результат будет одинаковый.

График сложение представлен на рисунке 1.4.

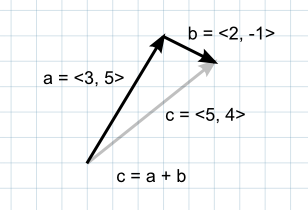


Рисунок 1.4– График сложения

Если первый вектор принять за точку в пространстве, то второй вектор можно интерпретировать как сдвиг или “прыжок” из этой точки. Например, чтобы для поиска точки 5-тью единицами выше точки на земле, вы могли бы использовать следующий расчёт: -

var pointInAir = pointOnGround + new Vector3(0, 5, 0);

Вычитание.

Вычитание векторов чаще всего используется, чтобы узнать расстояние и направление одного объекта относительно другого. Заметьте, что при вычитании порядок параметров имеет значение: -

 // The vector d has the same magnitude as c but points in the opposite direction.

var c = b - a;

var d = a - b;

Как и с обычными числами, прибавление отрицательного вектора - это то же

самое, что и вычитание положительного.

// These both give the same result.

var c = a - b;

var c = a + -b;

Отрицательный вектор имеет ту же величину, что и исходный вектор, и лежит на той же прямой, только в обратном направлении.