1 ОБЩАЯ ЧАСТЬ

* 1. Актуальность проекта

Главной задачей диплома является разработка игры «Survival» в среде Unity**, адаптированной для операционной системы windows 7. Жанр: приключение, симулятор. С элементами хоррора и выживание.** Разработка проекта заключается в оптимизации компьютерной игры Survival с ограничением возраста до четырнадцати лет.

* 1. Сетевые технологии
     1. Краткие сведения об игровых движках

Большинство игровых движков разработаны и настроены для того, чтобы запустить определённую игру на определённой платформе. И даже наиболее обобщённые многоплатформенные движки подходят для построения игр определённого жанра, например шутеров первого лица или гонок.

Основными популярными игровыми движками являются:

* Decima Engine.
* EGO Engine.
* Game Maker Studio.
* 4A Engine.
* IW Engine.
* RAGE.
* CryEngine.
* Id Tech.
* Frostbite и Unity Engine.
* Source и Unreal Engine.

Исторически шутеры от первого лица относятся к играм, которые наиболее технологически сложны, так как им необходимо представлять игроку иллюзию трёхмерного мира, и делать это для активных действий в реальном времени.

Движки шутеров от первого лица больше обращают внимание на такие технологии, как эффективный рендеринг трёхмерных миров, отзывчивая игровая механика контроля и прицеливания, высокая точность анимации оружия и рук управляемого игроком персонажа, широкий спектр ручного вооружения, «прощающая» модель движения игрока и его столкновения с препятствиями, высокое качество анимации и искусственного интеллекта неигровых персонажей.

Чаще всего 3D-движки или системы рендеринга в игровых движках построены на графическом API, таком как Direct3D или OpenGL, который обеспечивает программную абстракцию GPU или видеокарты. Низкоуровневые библиотеки, например, DirectX, SDL и OpenAL, также используются в играх, так как обеспечивают аппаратно-независимый доступ к другому аппаратному обеспечению компьютера, такому как устройства ввода (мышь, клавиатура и джойстик), сетевые и звуковые карты. До появления аппаратно-ускоряемой 3D-графики использовались программные визуализаторы. Программный рендеринг всё ещё используется в некоторых инструментах моделирования для рендеринга изображений, для которых визуальная достоверность важнее производительности (количество кадров в секунду) или когда аппаратное обеспечение компьютера не удовлетворяет требованиям, например, не поддерживает шейдеры.

1.2.2 Этапы разработки игры

Концептирование (Concept). На этом первом шаге команда придумывает концепцию игры, и проводит начальную проработку игрового дизайна. Главная цель данного этапа – это геймдизайнерская документация, включающая в себя Vision (развернутый документ, описывающий игру, как конечный бизнес-продукт) и Concept Document (начальную проработку всех аспектов игры).

Прототипирование (Prototyping). Важный этап проектирования любой игры – это создание прототипа. То, что хорошо выглядит «на бумаге», совершенно не обязательно будет интересно в реальности. Прототип реализуется для оценки основного игрового процесса, проверки различных гипотез, проведения тестов игровых механик, для проверки ключевых технических моментов.

Очень важно на этапе создания прототипа реализовывать только то, что нужно проверить и в сжатые сроки. Прототип должен быть простым в реализации, т.к. после достижения поставленных перед ним целей, он должен быть «выкинут». Серьёзная ошибка начинающих разработчиков – нести временную инфраструктуру и «костыли» реализации кода в основной проект.

Вертикальный срез (Vertical Slice). Цель Вертикального среза – получить минимально возможную полноценную версию игры, включающую в себя полностью реализованный основной игровой процесс. При этом высокое качество проработки обязательно нужно воплотить только для тех игровых элементов, которые существенно влияют на восприятие продукта. При этом все базовые фичи игры присутствуют как минимум в черновом качестве. Реализован минимальный, но достаточный для воплощения полноценного игрового процесса набор контента (один уровень или одна локация).

Производство контента (Content production). На этом этапе производится достаточное количество контента для первого запуска на внешнюю аудиторию. Реализуются все фичи, запланированные к закрытому бета-тестированию. Это наиболее продолжительный этап, который может занимать, для крупных клиентских проектов год и более.

На этом этапе задействуется наибольшее количество специалистов, которые занимаются производством всего основного наполнения игры. Художники создают все графические ресурсы, геймдизайнеры настраивают баланс и заполняют конфиги, программисты реализуют и полируют все фичи.

Friends & Family / CBT (закрытое бета-тестирование). На этапе CBT продукт впервые демонстрируется достаточно широкой публике, хотя и лояльной продукту или компании. Среди наиболее важных задач на этом этапе выступают: поиск и исправление гейм-дизайнерских ошибок, проблем игровой логики и устранение критических багов. На этом этапе в игре присутствуют уже все ключевые фичи, создано достаточно контента для полноценной игры продолжительное время, настроены сбор и анализ статистики. Тестирование идет по тест-плану, проводятся стресс-тесты уже с привлечением реальных игроков.

Soft Launch / OBT (открытый бета-тест). На этом этапе продолжается тестирование игры, но уже на широкой аудитории. Идет оптимизация под большие нагрузки. Игра должна быть готова для приема большого трафика. В игре реализован биллинг и принимаются платежи.

На этом этапе полностью завершается разработка новых фичей. Происходит feature freeze, программисты перестают реализовывать что-то новое, а полностью переключаются на отладку и тюнинг имеющихся фичей. Геймдизайнеры, продюсер и аналитики делают выводы из собранной на CBT статистики и проверяют эффективность монетизации.

Release. Ключевая цель – это получение прибыли. Базовый применяемый для оценки прибыльности критерий: количество денег, принесенных в среднем одним игроком за все время (LTV aka lifetime value), должно превосходить расходы на привлечение этого игрока (CPI aka cost per install).

На этом этапе должно быть полностью отлажено оперирование продукта (техническая поддержка, работа с комьюнити), соблюдаются маркетинговые и финансовые планы, ведутся работы по улучшению финансовых показателей, активно отрабатываются каналы по привлечению трафика.

1.2.3 Основные аспекты игр

Игровой программист — это инженер-программист, который в основном разрабатывает компьютерные игры или относящееся к ним программное обеспечение (такое как инструменты разработки игр). Всю работу с исходным кодом выполняют программисты. Как правило, есть один или несколько ведущих программистов, которые реализуют начальную кодовую базу, планируют развитие проекта в будущем, а также координируют других программистов.

Игровой программист может быть занят разработкой таких аспектов, как:

Физика — программирование игрового движка, в том числе симуляция «физики» (физическая модель) — движения объектов, столкновений и т.п.

Искусственный интеллект (ИИ) — создание компьютерных агентов с использованием методов ИИ; сюда входит написание скриптов, планирование, логическое программирование и др.

Графика — управление использованием графического содержимого и памяти компьютера; разработка графического движка, интеграция трёхмерных моделей, текстур для работы по движку физики.

Звуковое сопровождение — интеграция музыки, речи, звуковых эффектов в нужное место и время.

Геймплей — реализация различных игровых механик и особенностей.

Сценарии — разработка и поддержание высокоуровневой системы команд для различных внутри-игровых задач, таких как ИИ, триггеры редактора уровней и др.

Пользовательского интерфейс — программирование пунктов меню, визуального интерфейса, системы помощи и обратной связи и др.

Устройства ввода — обработка и настройка работы с различными устройствами ввода, таких как клавиатура, мышь, геймпад и т. д.

Сетевые коммуникации — управление вводом и выводом данных для локальных сетей или для Интернета.

Инструменты разработки игр — производство вспомогательного специализированного программного обеспечения, сопровождающих разработку игры; используется чаще в помощь дизайнерам и сценаристам.

* 1. Описание сценарного языка

1.3.1 Общие сведения

Скриптинг - необходимая составляющая всех игр. Даже самые простые игры нуждаются в скриптах для реакции на действия игрока и организации событий геймплея. Кроме того, скрипты могут быть использованы для создания графических эффектов, управления физическим поведением объектов или реализации пользовательской ИИ системы для персонажей игры.

Несмотря на то, что Unity для скриптинга использует стандартную среду исполнения Mono, есть свои практики и техники для доступа к движку. В этом разделе объясняется, как созданные в редакторе Unity объекты управляются из скриптов и детали отношений между геймплеем Unity и средой исполнения Mono.

Поведение игровых объектов контролируется с помощью компонентов (Components), которые присоединяются к ним. Несмотря на то, что встроенные компоненты Unity могут быть очень разносторонними, вскоре вы обнаружите, что вам нужно выйти за пределы их возможностей, чтобы реализовать ваши собственные особенности геймплея. Unity позволяет вам создавать свои компоненты, используя скрипты. Они позволяют активировать игровые события, изменять параметры компонентов, и отвечать на ввод пользователя каким вам угодно способом.

В отличии от других ассетов, скрипты обычно создаются непосредственно в Unity. Вы можете создать скрипт используя меню Create в левом верхнем углу панели Project или выбрав Assets > Create > C# Script (или JavaScript/Boo скрипт) в главном меню.

Новый скрипт будет создан в папке, которую вы выбрали в панели Project. Имя нового скрипта будет выделено, предлагая вам ввести новое имя.

Лучше ввести новое имя скрипта сразу после создания чем изменять его потом. Имя, которое вы введете будет использовано, чтобы создать начальный текст в скрипте.

1.3.2 Элементы сценарного языка

Unity изначально поддерживает три языка программирования:

* C# (произносится как Си-шарп), стандартный в отрасли язык подобный Java или C++.
* UnityScript, язык, разработанный специально для использования в Unity по образцу JavaScript.

Console Window (меню: Window> Console) показывает ошибки, предупреждения и другие сообщения, созданные Unity. Чтобы помочь с отладкой, вы также можете показывать свои собственные сообщения в консоли, используя функции Debug.Log, Debug.LogWarning и Debug.LogError Панель инструментов в окне консоли имеет ряд параметров, которые влияют на способ отображения сообщений. Кнопка очистить удаляет все сообщения, сгенерированные из вашего кода, но сохраняет ошибки компилятора. Вы можете настроить автоматическую очистку консоли при каждом запуске игры, включив параметр «Очистить при игре».

MonoDevelop - это интегрированная среда разработки (IDE), поставляемая вместе с Unity. IDE сочетает в себе функции текстового редактора с дополнительными возможностями для отладки и выполнения других задач по управлению проектами. Здесь не будет рассматриваться текстовый редактор, так как он достаточно интуитивен, но интеграция редактора и отладчика с Unity описана ниже.

MonoDevelop устанавливается по умолчанию вместе с Unity. Во время установки Unity на Windows, вы можете отменить установку MonoDevelop. Вы должны убедиться, что MonoDevelop установлен в качестве внешнего редактора скриптов в Preferences (меню: Unity > Preferences, а затем выберите панель External Tools). Если эта опция включена, Unity запустит MonoDevelop и будет использовать его в качестве редактора по умолчанию для всех скриптовых файлов.

В модулях ввода находится основная логика желаемого поведения EventSystem, они используются для:

* обработки ввода;
* управления состояниями событий;
* отправки событий объектам на сцене.

Только один модуль ввода может быть активным в системе событий одновременно. Он должен находиться на том же игровом объекте, что и компонент EventSystem.

Если вы желаете написать собственный модуль ввода, рекомендуется отправлять события, поддерживаемые уже существующими UI-компонентами Unity. Однако, вы можете расширять и создавать собственные события, как описано в документации по системе сообщений (Messaging System).

1.3.2.1 Типы данных

Типы данных имеют особенное значение в C#, поскольку это строго типизированный язык. Это означает, что все операции подвергаются строгому контролю со стороны компилятора на соответствие типов, причем недопустимые операции не компилируются. Следовательно, строгий контроль типов позволяет исключить ошибки и повысить надежность программ. Для обеспечения контроля типов все переменные, выражения и значения должны принадлежать к определенному типу. Такого понятия, как "бестиповая" переменная, в данном языке программирования вообще не существует. Более того, тип значения определяет те операции, которые разрешается выполнять над ним. Операция, разрешенная для одного типа данных, может оказаться недопустимой для другого.

В C# имеются две общие категории встроенных типов данных: типы значений и ссылочные типы. Они отличаются по содержимому переменной. Концептуально разница между ними состоит в том, что тип значения (valuetype) хранит данные непосредственно, в то время как ссылочный тип (referencetype) хранит ссылку на значение.

Эти типы сохраняются в разных местах памяти: типы значений сохраняются в области, известной как стек, а ссылочные типы — в области, называемой управляемой кучей.

Классификация типов данных

Давайте разберем типы значений.

Целочисленные типы

В C# определены девять целочисленных типов: char, byte, sbyte, short, ushort, int, uint, long и ulong. Но тип char применяется, главным образом, для представления символов и поэтому рассматривается отдельно. Остальные восемь целочисленных типов предназначены для числовых расчетов.

Вероятно, самым распространенным в программировании целочисленным типом является тип int. Переменные типа int нередко используются для управления циклами, индексирования массивов и математических расчетов общего назначения. Когда же требуется целочисленное значение с большим диапазоном представления чисел, чем у типа int, то для этой цели имеется целый ряд других целочисленных типов.

Так, если значение нужно сохранить без знака, то для него можно выбрать тип uint, для больших значений со знаком — тип long, а для больших значений без знака — тип ulong. В качестве примера ниже приведена программа, вычисляющая расстояние от Земли до Солнца в сантиметрах. Для хранения столь большого значения в ней используется переменная типа long:

Наравне с представлением char как символьных литералов, их можно представлять как 4-разрядные шестнадцатеричные значения Unicode (например, '\u0041'), целочисленные значения с приведением (например, (char) 65) или же шестнадцатеричные значения (например, '\x0041'). Кроме того, они могут быть представлены в виде управляющих последовательностей.

Логический тип данных Тип bool представляет два логических значения: "истина" и "ложь". Эти логические значения обозначаются в C# зарезервированными словами true и false соответственно. Следовательно, переменная или выражение типа bool будет принимать одно из этих логических значений. Кроме того, в C# не определено взаимное преобразование логических и целых значений. Например, 1 не преобразуется в значение true, а 0 — в значение false.

1.3.2.2 Агрегаты данных

Массивы. Массив представляет собой совокупность переменных одного типа с общим для обращения к ним именем. В C# массивы могут быть как одномерными, так и многомерными. Массивы служат самым разным целям, поскольку они предоставляют удобные средства для объединения связанных вместе переменных.

Массив представляет собой совокупность переменных одного типа с общим для обращения к ним именем. В C# массивы могут быть как одномерными, так и многомерными. Массивы служат самым разным целям, поскольку они предоставляют удобные средства для объединения связанных вместе переменных.

Массивами в C# можно пользоваться практически так же, как и в других языках программирования. Тем не менее у них имеется одна особенность: они реализованы в виде объектов.

Для тoго чтобы воспользоваться массивом в программе, требуется двухэтапная процедура, поскольку в C# массивы реализованы в виде объектов. Во-первых, необходимо объявить переменную, которая может обращаться к массиву. И во-вторых, нужно создать экземпляр массива, используя оператор new.

Следует иметь в виду, что если массив только объявляется, но явно не инициализируется, каждый его элемент будет установлен в значение, принятое по умолчанию для соответствующего типа данных (например, элементы массива типа

bool будут устанавливаться в false, а элементы массива типа int — в 0).

Инициализация массива

Помимо заполнения массива элемент за элементом (как показано в предыдущем примере), можно также заполнять его с использованием специального синтаксиса инициализации массивов. Для этого необходимо перечислить включаемые в массив элементы в фигурных скобках { }. Такой синтаксис удобен при создании массива известного размера, когда нужно быстро задать его начальные значения:

Реализация в C# массивов в виде объектов дает целый ряд преимуществ. Одно из них заключается в том, что с каждым массивом связано свойство Length, содержащее число элементов, из которых может состоять массив. Следовательно, у каждого массива имеется специальное свойство, позволяющее определить его длину.

Когда запрашивается длина многомерного массива, то возвращается общее число элементов, из которых может состоять массив. Благодаря наличию у массивов свойства Length операции с массивами во многих алгоритмах становятся более простыми, а значит, и более надежными.

Свойство Comparer получает способ сравнения для вызывающего множества. Свойство Мах получает наибольшее значение во множестве, а свойство Min — наименьшее значение во множестве.

Схематическое представление двумерного массива

Если вам приходилось раньше программировать на С, С++ или Java, то будьте особенно внимательны, объявляя или организуя доступ к многомерным массивам в C#. В этих языках программирования размеры массива и индексы указываются в отдельных квадратных скобках, тогда как в C# они разделяются запятой.

C# --- Руководство по C# --- Многомерные массивы

Многомерным называется такой массив, который отличается двумя или более измерениями, причем доступ к каждому элементу такого массива осуществляется с помощью определенной комбинации двух или более индексов. Многомерный массив индексируется двумя и более целыми числами.

Двумерные массивы

Простейшей формой многомерного массива является двумерный массив. Местоположение любого элемента в двумерном массиве обозначается двумя индексами. Такой массив можно представить в виде таблицы, на строки которой указывает один индекс, а на столбцы — другой

1.3.2.3 Программирование разветвляющихся и циклических процессов

C# имеются четыре различных вида циклов (for, while, do...while и foreach), позволяющие выполнять блок кода повторно до тех пор, пока удовлетворяется определенное условие. В этой статье мы познакомимся с циклами for и while.

Цикл while

Подобно for, while также является циклом с предварительной проверкой. Синтаксис его аналогичен, но циклы while включают только одно выражение:

while(условие)

оператор (операторы);

где оператор — это единственный оператор или же блок операторов, а условие означает конкретное условие управления циклом и может быть любым логическим выражением. В этом цикле оператор выполняется до тех пор, пока условие истинно. Как только условие становится ложным, управление программой передается строке кода, следующей непосредственно после цикла.

Как и в цикле for, в цикле while проверяется условное выражение, указываемое в самом начале цикла. Это означает, что код в теле цикла может вообще не выполняться, а также избавляет от необходимости выполнять отдельную проверку перед самим циклом.

Цикл for

Цикл for в C# предоставляет механизм итерации, в котором определенное условие проверяется перед выполнением каждой итерации. Синтаксис этого оператора показан ниже:

for (инициализатор; условие; итератор)

оператор (операторы)

Здесь: инициализатор это выражение, вычисляемое перед первым выполнением тела цикла (обычно инициализация локальной переменной в качестве счетчика цикла). Инициализация, как правило, представлена оператором присваивания, задающим первоначальное значение переменной, которая выполняет роль счетчика и управляет циклом;

Условие - это выражение, проверяемое перед каждой новой итерацией цикла (должно возвращать true, чтобы была выполнена следующая итерация);

Итератор - выражение, вычисляемое после каждой итерации (обычно приращение значения счетчика цикла).

Обратите внимание на то, что эти три основные части оператора цикла for должны быть разделены точкой с запятой. Выполнение цикла for будет продолжаться до тех пор, пока проверка условия дает истинный результат. Как только эта проверка даст ложный результат, цикл завершится, а выполнение программы будет продолжено с оператора, следующего после цикла for.

Стоит отметить, что цикл for отлично подходит для повторного выполнения оператора или блока операторов заранее известное количество раз.

1.3.2.4 Использование вспомогательных алгоритмов

Процедуры. Процедура – это подпрограмма, которая решает некоторую частную задачу или объединяет группу часто встречающихся операторов. Процедура описывается в блоке описания. Описание процедуры состоит из заголовка и тела.

Отличия функции от процедуры:

В заголовке указывается имя типа значения описываемой функции.

В теле функции должен быть хотя бы один оператор присваивания, в левой части которого указывается имя функции, иначе значение функции будет не определено.

Вызов функции может использоваться только в качестве операнда какого-либо выражения.

Модули. Кроме подпрограмм, средства языка позволяют создавать модули. В отличие от программы, модуль не может быть автономно запушен на выполнение и содержит элементы, например, переменные и подпрограммы, которые допускается использовать в программе или в других модулях. Для того чтобы можно было использовать средства модуля, его необходимо подключить, указав имя этого модуля в разделе uses. Типичными примерами модулей являются system и sysutils, содержащие большое количество стандартных подпрограмм (некоторые из них уже были рассмотрены). Напомним, что для каждой формы приложения создается отдельный модуль.

Компилятор распознает модуль по его заголовку и создает в результате своей работы не исполняемый файл (ЕХЕ), как это было для приложения, а файл модуля с расширением DCU.

Модуль состоит из заголовка, в котором после ключевого слова unit указывается имя модуля, и четырех разделов: интерфейса (interface), реализации (implementation), инициализации (initialization) и деинициализации (Finalization).

Заголовок процедуры состоит из служебного слова Procedure, за которым следует имя процедуры и в скобках список формальных параметров.

В программе все переменные делятся на глобальные и локальные. Переменные, описанные внутри процедуры, могут быть использованы только внутри этой процедуры. Эти переменные называются локальными.

Переменные, описанные в основной или головной программе, называются глобальными по отношению к данной процедуре, область действия глобальных переменных – вся программа, включая и процедуру.

Параметры, перечисляемые в скобках, называются формальными. Фактические параметры – это параметры, которые передаются процедуре при обращении к ней.

Формальные параметры делятся на параметры-значения и параметры-переменные.

В разделе реализации располагается код подпрограмм, заголовки которых были приведены в разделе интерфейса. Порядок следования подпрограмм может не совпадать с порядком расположения их заголовков, приводимых в разделе интерфейса. Кроме того, допускается оставлять в заголовке только имя подпрограммы, т. к. список параметров и тип результата функции уже были предварительно указаны. В разделе реализации можно также описывать типы, объявлять константы и переменные и описывать подпрограммы, которые используются только в этом модуле и за его пределами не видны. Раздел интерфейса начинается, словом implementation.

В разделе интерфейса размещаются описания идентификаторов, которые должны быть доступны всем модулям и программам, использующим этот модуль и содержащим его имя в списке uses. В разделе интерфейса объявляются типы, константы, переменные и подпрограммы. При этом для подпрограмм указываются только их заголовки. Другие используемые модули указываются в списке uses. Раздел интерфейса начинается ключевым словом interface.

1.3.2.5 Способы структурирования программ

Инкапсуляция представляет собой объединение данных и обрабатывающих их методов (подпрограмм) внутри класса (объекта). Это означает, что и в классе инкапсулируются (объединяются и помешаются внутрь) поля, свойства и методы. При этом класс получает определенную функциональность, например, обеспечивая полный набор средств для создания программы поддержки некоторого элемента интерфейса (окна Windows, редактора и т. п.) или прикладной обработки.

Наследование — это процесс порождения новых объектов-потомков от существующих объектов-родителей, при этом потомок берет от родителя все его поля, свойства и методы. В дальнейшем наследуемые поля, свойства и методы можно использовать в неизменном виде или переопределять (модифицировать).

#### Классы. В языке ObjectPascal классы — это специальные типы данных, используемые для описания объектов. Соответственно объект, имеющий тип какого-либо класса, является экземпляром этого класса или переменной этого типа.

Класс представляет собой особый тип записи, имеющий в своем составе такие элементы (члены), как поля, свойства и методы. Поля класса аналогичны полям записи и служат для хранения информации об объекте. Методами называются процедуры и функции, предназначенные для обработки полей. Свойства занимают промежуточное положение между полями и методами. С одной стороны, свойства можно использовать как поля, например, присваивая им значения с помощью оператора присваивания; с другой стороны, внутри класса доступ к значениям свойств выполняется методами класса.

Базой ООП являются:

* инкапсуляция;
* наследование;
* полиморфизм.

В приведенной структуре описаниями являются объявления свойств, методов и событий.

Для различных элементов класса можно устанавливать разные права доступа (видимости), для чего в описании класса используются отдельные разделы, обозначенные специальными спецификаторами видимости.

Разделы private и protected содержат защищенные описания, которые дос­тупны внутри модуля, в котором они находятся. Описания из раздела protected, кроме того, доступны для порожденных классов за пределами названного модуля.

Раздел public содержит общедоступные описания, которые видимы в любом месте программы, где доступен сам класс. Раздел published содержит опубликованные описания, которые в дополнение к общедоступным описаниям порождают динамическую (т. е. во время выполнения программы) информацию о типе (Run-TimeTypeInformation, RTTJ). По этой информации при выполнении приложения производится проверка на принадлежность элементов объекта тому или иному классу. Одним из назначений раздела published является обеспечение доступа к свойствам объектов при конструировании приложений.

TNewdass создается на основе базового класса Tobject и получает в дополнение три новых поля Fcode, FSign и FMote, имеющих, соответственно, целочисленный, символьный и строковый типы.

При создании новых классов класс-потомок наследует все поля родителя, при этом удаление или переопределение этих полей невозможно, но допускается добавление новых. Таким образом, чем дальше по иерархии какой-либо класс находится от родительского класса, тем больше полей он имеет.

Напомним, что изменение значений полей обычно выполняется с помощью методов и свойств объекта.

#### Свойства. Свойства реализуют механизм доступа к полям. Каждому свойству соответствует поле, содержащее значение свойства, и два метода, обеспечивающих доступ к этому полю. Описание свойства начинается со слова property, при этом типы свойства и соответствующего поля должны совпадать. Ключевые слова read и write являются зарезервированными внутри объявления свойства и служат для указания методов класса, с помощью которых выполняется чтение значения поля, связанного со свойством, или запись нового значения в это поле.

Пример описания свойств:

Для доступа к полям Fcode, FSign и Fnote, которые описаны в защищенном разделе и недоступны другим классам, используются свойства code, Sign и Note, соответственно.

Методы. Метод представляет собой подпрограмму (процедуру или функцию), являющуюся элементом класса. Описание метода похоже на описание обычной подпрограммы модуля. Заголовок метода располагается в описании класса, а сам код метода находится в разделе реализации. Имя метола в разделе реализации pявляется составным и включает в себя тип класса.

Метод, объявленный в классе, может вызываться различными способами, что зависит от вида этого метода. Вид метода определяется модификатором, который указывается в описании класса после заголовка метода и отделяется от заголовка точкой с запятой. Приведем некоторые модификаторы:

* virtual - виртуальный метод;
* dynamic - динамический метод;
* override - переопределяемый метод;
* message - обработка сообщения;
* abstract - абстрактный метод.

При выполнении конструктора часто также осуществляется инициализация элементов объекта с помощью значений, передаваемых в качестве параметров конструктора.

В конструкторе класса-потомка сначала вызывается конструктор родителя, а затем выполняются остальные действия. В классе-потомке директива override (переопределить) обеспечивает возможность родительскому классу использовать новый метод. Ключевое слово inherited служит для вызова методов родительского класса.

1.3.3 Управление игровыми объектами

В редакторе Unity вы изменяете свойства Компонента используя окно Inspector. Так, например, изменения позиции компонента Transform приведет к изменению позиции игрового объекта. Аналогично, вы можете изменить цвет материала компонента Renderer или массу твёрдого тела (RigidBody) с соответствующим влиянием на отображение или поведение игрового объекта. По большей части скрипты также изменяют свойства компонентов для управления игровыми объектами. Разница, однако, в том, что скрипт может изменять значение свойства постепенно со временем или по получению ввода от пользователя. За счет изменения, создания и уничтожения объектов в заданное время может быть реализован любой игровой процесс.

Наиболее простым и распространенным является случай, когда скрипту необходимо обратиться к другим компонентам, присоединенных к тому же GameObject. Как упоминалось во разделе Введение, компонент на самом деле является экземпляром класса, так что первым шагом будет получение ссылки на экземпляр компонента, с которым вы хотите работать. Это делается с помощью функции GetComponent. Типично, объект компонента сохраняют в переменную, это делается в C# посредством следующего синтаксиса:

void Start () {

Rigidbody rb = GetComponent<Rigidbody>();

}

Как только у вас есть ссылка на экземпляр компонента, вы можете устанавливать значения его свойств, тех же, которые вы можете изменить в окне Inspector:

void Start () {

Rigidbody rb = GetComponent<Rigidbody>();

  / Change the mass of the object's Rigidbody.  rb.mass = 10f;

Дополнительная возможность, недоступная в окне Inspector - вызов функций экземпляра компонента:

void Start () {

Rigidbody rb = GetComponent<Rigidbody>();

// Add a force to the Rigidbody.  rb.AddForce(Vector3.up \* 10f);

}

Имейте ввиду, что нет причины, по которой вы не можете иметь больше одного пользовательского скрипта, присоединенного к одному и тому же объекту. Если вам нужно обратиться к одному скрипту из другого, вы можете использовать, как обычно, GetComponent, используя при этом имя класса скрипта (или имя файла), чтобы указать какой тип Компонента вам нужен. Если вы попытаетесь извлечь Компонент, который не был добавлен к Игровому Объекту, тогда GetComponent вернет null; возникнет ошибка пустой ссылки при выполнении (null reference error at runtime), если вы попытаетесь изменить какие-либо значения у пустого объекта Обращение к другим объектам. Пусть иногда они и существуют изолированно, все же, обычно, скрипты отслеживают другие объекты. Например, преследующий враг должен знать позицию игрока. Unity предоставляет несколько путей получения других объектов, каждый подходит для конкретной ситуации. Связывание объектов через переменные. Самый простой способ найти нужный игровой объект - добавить в скрипт переменную типа GameObject с уровнем доступа public:

public class Enemy : MonoBehaviour {

  public GameObject player;  // Other variables and functions...

}

переменная в окне Inspector представлена на рисунке 1.1

https://docs.unity3d.com/ru/current/uploads/Main/GameObjectPublicVar.png

Рисунок1.1– Переменная в окне

Теперь вы можете перетащить объект со сцены или из панели Hierarchy в эту переменную, чтобы назначить его. Функция GetComponent и доступ к переменным компонента доступны как для этого объекта, так и для других, то есть вы можете использовать следующий код:

public class Enemy : MonoBehaviour {

public GameObject player;

void Start() {

// Start the enemy ten units behind the player

character.  transform.position = player.transform.position - Vector3.forward \* 10f;

}  }

Кроме того, если объявить переменную с доступом public и заданным типом компонента в вашем скрипте, вы сможете перетащить любой объект, который содержит присоединенный компонент такого типа. Это позволит обращаться к компоненту напрямую, а не через игровой объект, public Transform playerTransform. Соединение объектов через переменные наиболее полезно, когда вы имеете дело с отдельными объектами, имеющими постоянную связь. Вы можете использовать массив для хранения связи с несколькими объектами одного типа, но связи все равно должны быть заданы в редакторе Unity, а не во время выполнения. Часто удобно находить объекты во время выполнения, и Unity предоставляет два основных способа сделать это, описанных ниже.

1.3.3.1 Нахождение дочерних объектов

Иногда игровая сцена может использовать несколько объектов одного типа, таких как враги, путевые точки и препятствия. Может возникнуть необходимость отслеживания их в определенном скрипте, который управляет или реагирует на них (например, все путевые точки могут потребоваться для скрипта поиска пути).

Можно использовать переменные для связывания этих объектов, но это сделает процесс проектирования утомительным, если каждую новую путевую точку нужно будет перетащить в переменную в скрипте. Аналогично, при удалении путевой точки придется удалять ссылку на отсутствующий объект. В случаях, наподобие этого, чаще всего удобно управлять набором объектов, сделав их дочерними одного родительского объекта. Дочерние объекты могут быть получены, используя компонент Transform родителя (так как все игровые объекты неявно содержат Transform):

using UnityEngine;

public class WaypointManager : MonoBehaviour {

public Transform[] waypoints;

void Start() {

waypoints = new Transform[transform.childCount];

int i = 0;

foreach (Transform t in transform)

{  waypoints[i++] = t;

}

}

}

Вы можете также найти заданный дочерний объект по имени, используя функцию Transform.Find:

transform.Find("Gun");

Нахождение объектов по имени или тегу.

Нахождение игровых объектов в любом месте иерархии доступно всегда, когда у вас есть некоторая информация, по которой их можно идентифицировать. Отдельные объекты могут быть получены по имени, используя функцию GameObject.Find:

GameObject player;

void Start() {

player = GameObject.Find("MainHeroCharacter");

}

Объект или коллекция объектов могут быть также найдены по их тегу, используя функции GameObject.FindWithTag и GameObject.FindGameObjectsWithTag:

GameObject player;  GameObject[] enemies;

void Start() {

player=GameObject.FindWithTag("Player");

enemies= GameObject.FindGameObjectsWithTag("Enemy");

}

1.3.3.2 Функции событий

Скрипт в Unity не похож на традиционную идею программы, где код работает постоянно в цикле, пока не завершит свою задачу. Вместо этого, Unity периодически передаёт управление скрипту при вызове определённых объявленных в нём функций. Как только функция завершает исполнение, управление возвращается обратно к Unity. Эти функции известны как функции событий, т.к. их активирует Unity в ответ на события, которые происходят в процессе игры. Unity использует схему именования, чтобы определить, какую функцию вызвать для определённого события. Например, вы уже видели функцию Update (вызывается перед сменой кадра) и функцию Start (вызывается прямо перед первым кадром объекта). В Unity доступно гораздо большее количество функций событий; полный список с дополнительной информацией по их применению можно найти на странице документации класса MonoBehaviour. Далее указаны одни из самых важных и часто встречающихся событий.

Обычные Update события

Игра - это что-то вроде анимации, в которой кадры генерируются на ходу. Ключевой концепт в программировании игр заключается в изменении позиции, состояния и поведения объектов в игре прямо перед от рисовкой кадра. Такой код в Unity обычно размещают в функции Update. Update вызывается перед от рисовкой кадра и перед расчётом анимации.

void Update() {

float distance = speed \* Time.deltaTime \* Input.GetAxis("Horizontal");

transform.Translate(Vector3.right \* distance);

}

Физический движок также обновляется фиксированными по времени шагами, аналогично тому как работает от рисовка кадра. Отдельная функция события FixedUpdate вызывается прямо перед каждым обновлением физических данных. Т.к. обновление физики и кадра происходит не с одинаковой частотой, то вы получите более точные результаты от кода физики, если поместите его в функцию FixedUpdate, а не в Update.

void FixedUpdate() {

Vector3 force = transform.forward \* driveForce \* Input.GetAxis("Vertical");

rigidbody.AddForce(force); }

Также иногда полезно иметь возможность внести дополнительные изменения в момент, когда у всех объектов в сцене отработали функции Update и FixedUpdate и рассчитались все анимации. В качестве примера, камера должна оставаться привязанной к целевому объекту; подстройка ориентации камеры должна производиться после того, как целевой объект сместился. Другим примером является ситуация, когда код скрипта должен переопределить эффект анимации (допустим, заставить голову персонажа повернуться к целевому объекту в сцене). В ситуациях такого рода можно использовать функцию LateUpdate.

void LateUpdate(){

Camera.main.transform.LookAt(target.transform);

}

События инициализации.

Зачастую полезно иметь возможность вызвать код инициализации перед любыми обновлениями, происходящими во время игры. Функция Start вызывается до обновления первого кадра или физики объекта. Функция Awake вызывается для каждого объекта в сцене в момент загрузки сцены. Учтите, что хоть для разных объектов функции Start и Awake и вызываются в разном порядке, все Awake будут выполнены до вызова первого Start. Это значит, что код в функции Start может использовать всё, что было сделано в фазе Awake.

События GUI.

В Unity есть система для отрисовки элементов управления GUI поверх всего происходящего в сцене и реагирования на клики по этим элементам. Этот код обрабатывается несколько иначе, нежели обычное обновление кадра, так что он должен быть помещён в функцию OnGUI, которая будет периодически вызываться.

void OnGUI() {

GUI.Label(labelRect, "Game Over");

}

Вы также можете определять события мыши, которые срабатывают у GameObject’а, находящегося в сцене. Это можно использовать для наведения орудий или для отображения информации о персонаже под указателем курсора мыши. Существует набор функций событий OnMouseXXX (например, OnMouseOver, OnMouseDown), который позволяет скрипту реагировать на действия с мышью пользователя. Например, если кнопка мыши нажата в то время, когда курсор мыши находится над определённым объектом, то, если в скрипте этого объекта присутствует функция OnMouseDown, она будет вызвана.

События физики.

Физический движок сообщит о столкновениях с объектом с помощью вызова функций событий в скрипте этого объекта. Функции OnCollisionEnter, OnCollisionStay и OnCollisionExit будут вызваны по началу, продолжению и завершению контакта. Соответствующие функции OnTriggerEnter, OnTriggerStay и OnTriggerExit будут вызваны когда коллайдер объекта настроен как Trigger (т.е. этот коллайдер просто определяет, что его что-то пересекает и не реагирует физически). Эти функции могут быть вызваны несколько раз подряд, если обнаружен более чем один контакт во время обновления физики, поэтому в функцию передаётся параметр, предоставляющий дополнительную информацию о столкновении (координаты, “личность” входящего объекта и т.д.).

void OnCollisionEnter(otherObj: Collision) {

if (otherObj.tag == "Arrow") {

ApplyDamage(10);

}

}

1.3.3.3 Управление временем и частотой кадров

Функция обновления позволяет регулярно отслеживать входные данные и другие события из сценария и предпринимать соответствующие действия. Например, вы можете перемещать символ при нажатии клавиши «вперед». При обработке таких действий, основанных на времени, важно помнить, что частота кадров в игре не постоянна и не является промежутком времени между вызовами функции Update.

В качестве примера рассмотрим задачу постепенного перемещения объекта вперед по одному кадру за раз. Сначала может показаться, что вы можете просто сдвинуть объект на фиксированное расстояние в каждом кадре:

//C# script example

using UnityEngine;

using System.Collections;

public class ExampleScript : MonoBehaviour {

public float distancePerFrame;

void Update() {

transform.Translate(0, 0, distancePerFrame);

}

}

//JS script example

var distancePerFrame: float;

function Update() {

transform.Translate(0, 0, distancePerFrame);

}

Однако, учитывая, что время кадра не является постоянным, объект будет двигаться с нерегулярной скоростью. Если время кадра составляет 10 миллисекунд, то объект будет двигаться вперед с помощью distancePerFrame сто раз в секунду. Но если время кадра увеличивается до 25 миллисекунд (скажем, из-за нагрузки на процессор), то оно будет только шагать вперед сорок раз в секунду и, следовательно, охватывать меньшее расстояние. Решение состоит в том, чтобы масштабировать размер движения по времени кадра, которое вы можете прочитать из свойства Time.deltaTime :

//C# script example

using UnityEngine;

using System.Collections;

public class ExampleScript : MonoBehaviour {

public float distancePerSecond;

void Update() {

transform.Translate(0, 0, distancePerSecond \* Time.deltaTime);

}

}

//JS script example

var distancePerSecond: float;

function Update() {

transform.Translate(0, 0, distancePerSecond \* Time.deltaTime);

}

Обратите внимание, что движение теперь задается как distancePerSecond, а не distancePerFrame. По мере изменения частоты кадров размер шага перемещения будет меняться соответственно, и поэтому скорость объекта будет постоянной.

Фиксированный временной шаг.

В отличии от основного обновления кадров, система физики Юнити делает работу с фиксированным временным шагом, который имеет важное значение для точности и последовательности моделирования. В начале обновления физики Unity устанавливает «сигнал тревоги», добавляя фиксированное значение временного шага ко времени окончания последнего обновления физики. Затем физическая система будет выполнять вычисления, пока не сработает сигнализация.

Вы можете изменить размер фиксированного временного шага из Time Manager и прочитать его из скрипта, используя свойство Time.fixedDeltaTime. Обратите внимание, что более низкое значение временного шага приведет к более частым обновлениям физики и более точному моделированию, но за счет большей загрузки процессора. Вероятно, вам не нужно будет менять фиксированный временной интервал по умолчанию, если вы не предъявляете высокие требования к физическому движку.

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.

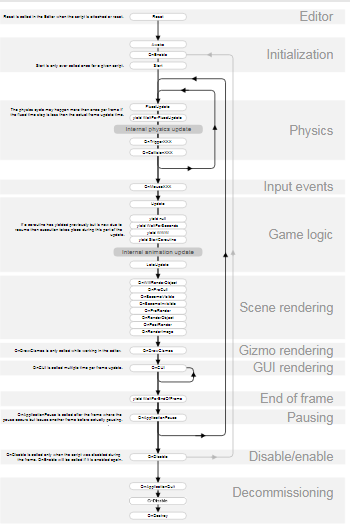


Рисунок 1.2– Блок-схема жизненного цикла скрипта

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.

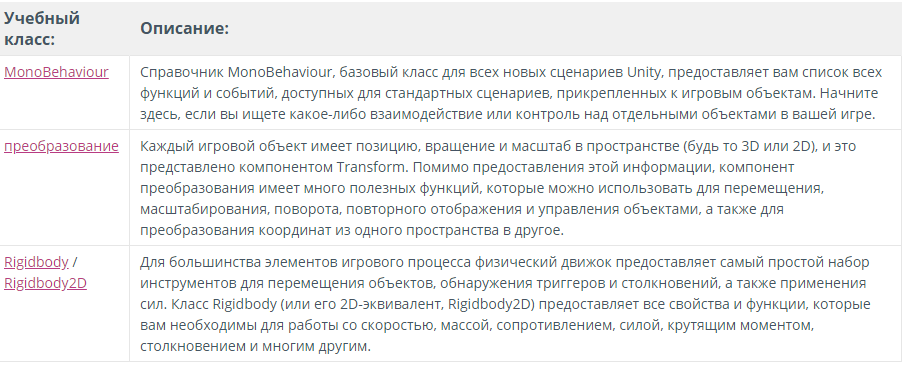


Рисунок 1.3– Описания классов

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.

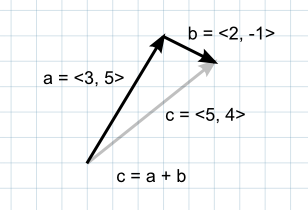


Рисунок 1.4– График сложения

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.