БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Механико-математический факультет
Кафедра веб-технологий и компьютерного моделирования

Иванов Тимофей Владимирович
Данилевич Матвей Кириллович
Крайний Эдвард Самвелович

Разработка 2D-Платформера на Unity

Курсовая работа
Студентов III курса
Руководитель:
Нагорный Юрий Евгеньевич
Старший преподаватель кафедры веб-технологий и компьютерного моделирования

Минск 2021

Содержание

Введение

Научно-технический прогресс, набравший к концу XX века головокружительную скорость, послужил причиной появления такого гений современной мысли как компьютер. С совершенствованием компьютеров совершенствовались и игры, привлекая все больше и больше людей. На сегодняшний день компьютерная техника достигла такого уровня развития, что позволяет программистам разрабатывать очень реалистичные игры с хорошим графическим и звуковым оформлением.

Появление компьютерных игр можно отнести к моменту, когда компьютеры из сферы экспериментальной и почти секретной (ведь на них должны были рассчитываться траектории снарядов и ракет во время военных действий) начали переходить в мир научный и практический. Это произошло в конце 60-х гг. XX в.

Компьютер стал обладать неким более или менее дружественным пользователю интерфейсом - вместо лампочек и загадочных индикаторов появились алфавитно-цифровые дисплеи. Конечно, в тот момент ни о какой графике не могло идти и речи. Но вот в один прекрасный вечер после тяжёлого трудового дня молодой программист решил написать небольшую программу, которая играла бы с ним в какую-нибудь не очень сложную игру, например, «Быки и коровы». И такая идея пришла в голову не только одному человека – вскоре программы для развлечения начали появляться всё чаще и чаще и даже стали входить в состав пакетов программ, поставляемых вместе с компьютерами.

С созданием компьютерной графики и появлением настоящих домашних компьютеров игровая индустрия стремительно выросла. Игры выпускались тысячными тиражами. Примерно за десять лет для домашнего компьютера ZX-Spectrum фирмы Sinclair Research было выпущено более 6 тысяч игр.

Сейчас игровая индустрия является одной из точек опоры, на которых стоит индустрия персональных компьютеров, да и для чего нужен обычному человеку компьютер, как не для развлечений?

Одним из популярных способов разработки игр является использование игрового движка Unity.

Первая причина популярности - Unity бесплатен для коммерческого использования. Сейчас любой желающий программист или небольшая студия может свободно скачать последнюю версию движка, сделать в нём игру, опубликовать её и начать зарабатывать с продаж или внутриигровых покупок совершенно бесплатно.

Unity современный игровой движок, постоянно совершенствующийся и способный конкурировать по качеству получаемого игрового продукта с ведущими игровыми движками, такими как: Unreal Engine 4, CryEngine 3, Source , RAGE, Frostbite Engine и другие.

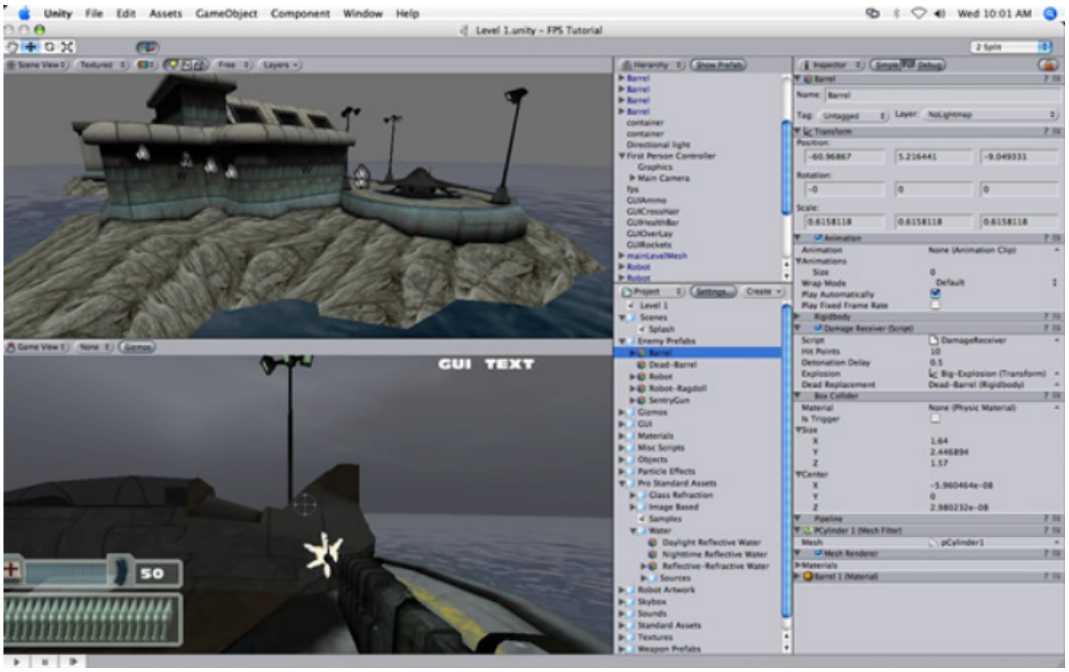
Огромная кроссплатформенность также является большим плюсом: игры, разработанные на этом движке доступны на многих платформах от мобильных до ПК. На мобильных устройствах Unity занимает одну из лидирующих позиций. Все вышеописанные плюсы Unity выделяют его на фоне остальных игровых движков, и поэтому он широко используется среди начинающих, и не только, игровых разработчиков. Именно по данным критериям и мы решили остановить свой выбор на данном игровом движке.

Глава 1. Unity как платформа для разработки

1. История развития Unity

Первая версия, **Unity 3D**, была создана в июне 2005 года тремя людьми: Джошимом Анте, Дэвидом Хелгасоном и Николасом Френсисом. Их целью было создание доступного любительского движка с удобным графическим интерфейсом. Первая версия Unity имела схожий с финальной версией пользовательский интерфейс, работающий по принципу drag-and-drop (перетаскивание элементов между подменю программы).

Изначально Unity работал только под macOS, но разработчики отлично понимали, что пользователей Windows явно больше пользователей macOS. Поэтому первое глобальное добавление, появившееся в августе 2005 в версии 1.1 — это возможность сборки игр под Windows и поддержка плагинов C/C++. Однако на тот момент полноценным игровым движком Unity назвать было трудно — он не поддерживал множество графических эффектов и подходил разве что для создания качественных анимаций. Разработчики исправили это недоразумение в версии 1.2, выпущенной в декабре 2005 года. Также в ней добавили эффекты постобработки, тени, встроенный скрипт управления персонажем, расширили возможности редактора скриптов. После этого движком стали интересоваться инди-разработчики. На версии 1.6.2 Графический интерфейс и игры на Unity 1 выглядели так:



В октябре 2007 команда разработчиков выпустила вторую версию движка, **Unity 3D 2**. Основные изменения — добавление полноценной среды разработки под Windows а также улучшенный веб-плеер. Так как macOS использует API OpenGL, а Windows — в основном DirectX, разработчики добавили поддержку DirectX в версию для Windows. Также была добавлена поддержка веб-стриминга и мягких теней реального времени, Terrain Engine, а также полностью переработали GUI движка. В Unity 2.6 инди-версия движка стала бесплатной и появилась версия для Wii.

Третья версия Unity, **Unity 3D 3**, вышла в сентябре 2010. Было внесено много изменений — появилась возможность менять местами все элементы редактора для удобства использования, улучшили карты освещения. Добавили возможность отложенного рендеринга, возможность отрисовки лишь тех элементов, которые отображаются на экране, низкоуровневую отладку.

Новая версия движка Unity, **Unity 3D 4**, вышла в ноябре 2012. Основное изменение — движок научился работать под Linux. Также была добавлена поддержка API DirectX 11, улучшена система анимации и освещения.

Самая новая версия движка, **Unity 3D 5**, развивается с марта 2014 и по сей день. Количество зарегистрированных разработчиков превышает 3 миллиона — в основном потому, что Indie-версия движка бесплатна. Для тех, кому нужны дополнительные возможности, есть версия Plus и Pro, а так же есть возможность собрать редактор самому и договориться с разработчиками о цене.

В Unity 5 добавили достаточно много функций:

- Новые инструменты графического интерфейса отдельно для 3D и 2D игр.
- Полноценный звуковой редактор (можно в реальном времени объединять различные звуки, добавлять эффекты, связывать их с событиями в игре).
- Поддержка WebGL — игры работают напрямую в браузере без установки веб-плеера.

- Глобальное освещение в реальном времени для консолей нового поколения, ПК и мобильных платформ.
- Отражение света в реальном времени на основе Reflection Probes.
- Физически корректные материалы (к примеру кусок дерева теперь плавает в воде сам, без дополнительных скриптов).
- Новые возможности 2D физики: точечные силы притягивания и отталкивания; тангенциальные силы (силы, направленные по касательной к поверхности объекта); силы, направленные вдоль любых осей; одностороннее столкновение.
- Отслеживание загрузки процессора, видеокарты и памяти на временной шкале в режиме реального времени.
- Добавление полноценного 64-битного редактора.
- Интеграция Terrain Speedtree.
- Добавление новых API для 2D физики и редактора анимации в Box2D, обновление 3D физики до nVidia PhysX3.
- Просмотр сцен в HDR-режиме.
- Настройки для рендера сцены с помощью заполняющего (Ambient) света.
- Улучшена работа LOD (теперь нет падения производительности для непропорционально скейлированной геометрии).
- Новые формы для препятствий Navmesh и сжатые текстуры для Cubemaps.
- Поддержка джойстика для Windows Store.
- Внутриигровая реклама без сторонних плагинов (к сожалению — трудно вырезаемая, так как зашита в саму игру в виде текстур и показывается даже без наличия интернета).
- Повторяющаяся анимация может передвигать персонажа.

На данный момент это один из самых быстроразвивающихся движков, разработчики которого постоянно улучшают его и внедряют новые функции.

2. Разработка 2D приложений

Создание приложений в Unity можно разделить на составные части, работающие и взаимодействующие друг с другом.

Physics2D

В Unity имеется отдельный физический движок для обработки физики объектов в 2D пространстве. Большинство 2D физических компонент - это просто “сплющенные” версии своих 3D компонент.

Настройки параметров в 2D Physics определяют пределы точности моделирования физики. Более точное моделирование требует больших затрат в производительности, и настройка данных параметров позволяет выбрать тот самый баланс между точностью и производительностью.

Scripting

Скрипты - это основной элемент во всех приложениях, написанных с помощью Unity. Большинству приложений скрипты требуются для обработки взаимодействия персонажа с окружением, для обработки событий, происходящих при взаимодействии. Кроме этого, скрипты могут быть использованы для создания графических эффектов, контролирования физического окружения, также для создания ИИ внутриигровых персонажей.

Скриптинг обеспечивает почти все происходящие на экране: геймплейные механики, активация визуальных эффектов и т.д.

Audio

Audio в Unity обеспечивает полностью пространственный звук, возможность микширования и аудио-мастеринга в реальном времени.

Аудио-составляющая в любой игре – очень важная часть любой игры, без нее невозможно представить хорошую игру. Благодаря тонкой настройке аудио-дизайнер может очень точно выверять настроение игры, и добавлять атмосферные эффекты.

Animation

В Unity для создания анимаций можно использовать спрайтовые анимации. Спрайтовой анимацией являются специально собранные анимационные клипы, содержащие последовательность для смены спрайтов. Такой анимационный клип просто будет менять спрайт заданного объекта каждые несколько кадров.

Для 2D игр зачастую используются спрайтовые анимации, то есть заранее собранные в последовательность несколько изображений. Их быстрая смена и создает иллюзию движения.

Кроме такого способа анимирования, существует возможность анимировать объекты по заранее заданным ключевым кадрам. Если в ключевые кадры будет передано значение, которое можно экстраполировать, то встроенный аниматор Unity сможет создать дополнительные кадры между ключевыми кадрами. Такой тип анимации позволяет создавать очень плавное движение, плавность которого в итоге будет зависеть от частоты кадров (и будет масштабироваться).

UI

Unity предоставляет 3 UI системы, которые пользователь может использовать для создания интерфейсов:

- UI Toolkit
- The Unity UI package (uGUI)
- IMGUI

Unity рекомендует UI Toolkit систему для разработки интерфейсов, но некоторый функционал пока что доступен лишь в uGUI и IMGUI. Несмотря на то, что эти системы более старые, их использование в некоторых случаях будет более предпочтительным.

Благодаря данным инструментам в Unity есть возможность создания собственных интерфейсов.

Cross-platform

Unity является кроссплатформенным инструментом, и поддерживает следующие платформы:

- Windows, macOS и Linux Standalone
- tvOS

- iOS
- Lumin
- Android
- WebGL
- PS4
- Xbox One

Кросс-платформенность – очень важное свойство любого инструмента, ведь именно оно позволяет утилите быть более гибкой для любого пользователя.

Глава 2. Реализация проекта на Unity

3. 2D Графика

Изначально Unity создавался для 3D графики, и чтобы имитировать 2D графику разработчикам приходилось идти на ухищрения, используя плоские объекты с натянутыми текстурами на плоскости в пространстве и переписывая работу перспективной камеры. С популяризацией 2D направления разработчики Unity добавили большое количество инструментов для работы в этом направлении.

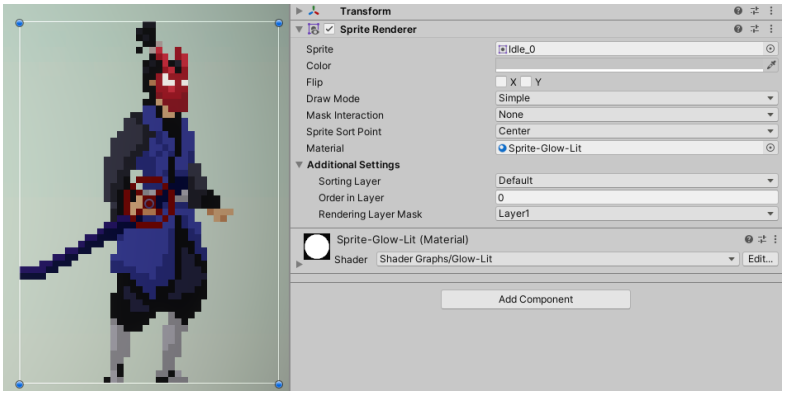
Теперь для использования доступны ортографические камеры и спрайты, стандартный аниматор поддерживает работу 2D компонентов, а в редактор интегрирован физический движок 2D объектов.

2D объекты, отображаемые на дисплее, называются Спрайтами. Unity предоставляет встроенный редактор спрайтов, позволяющий извлечь спрайт из большого изображения, например, для разделения рук, ног и тела персонажа внутри одного изображения.

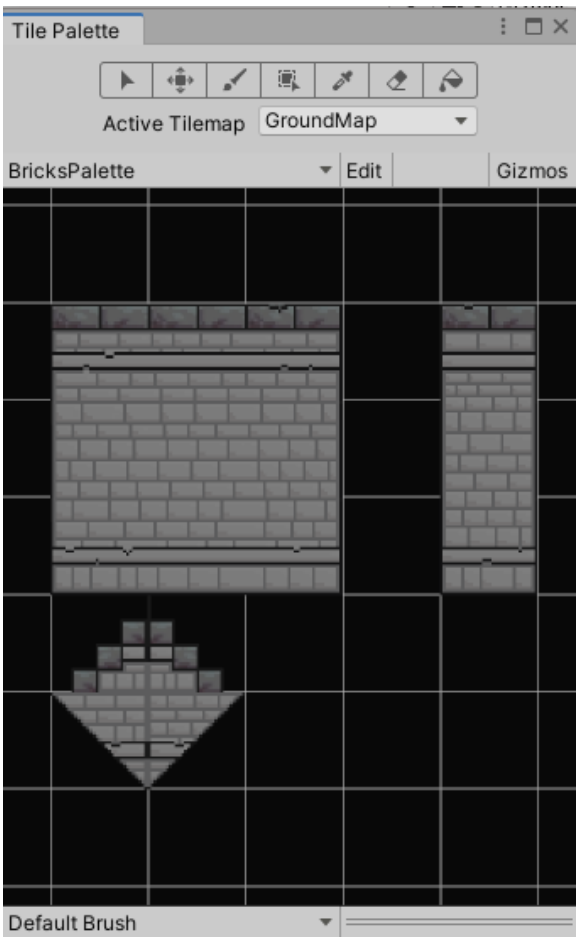
Так как обычно все объекты в 2D находятся на одной плоскости, для решения проблемы, какие объекты будут отображаться поверх остальных, в Unity существуют слои сортировки и порядок отображения. Если порядок отображения первого объекта выше порядка второго объекта внутри одного и того же слоя, то первый будет отображен поверх второго. Слои же определяют общий порядок отображения независимо от порядка внутри слоя. То есть, если объект порядка 2 находится на слой ниже объекта порядка 1, то все равно отображен будет второй объект.

Такое разделение на слои и порядки позволяет определить внутри приложения какие объекты будут находиться на заднем фоне, а какие на переднем, и, кроме того, тонко манипулировать наложением объектов.

Для создания 2D объектов используется компонент SpriteRenderer:



Для быстрого создания игровых уровней зачастую используются компонент Tilemap, хранящий в себе набор спрайтов, которые отображаются на заданной игровой сетке. Для быстрого редактирования этого компонента существует встроенный инструмент TilePalette, позволяющий изменять сетку с использованием заранее созданных элементов:



Кроме специальных возможностей для 2D графики, Unity предоставляет и много других визуальных улучшений:

- Post-processing – большое количество эффектов коррекции изображения, применяемых уже после отрисовки кадра: Ambient Occlusion, Anti-aliasing, Bloom, Auto Exposure, Depth of Field, Grain, Lens Distortion, Motion Blur, Lift, Gamma, Gain, Tone Mapping, Vignette Mask и т.д.
- Lighting – затемнение и освещение объектов на сцене в зависимости от источников освещения, а также отбрасывание теней.

- Particle system – система, которая обрабатывает и рендерит большое количество небольших изображений или объектов (частиц), чтобы создавать какие-либо визуальные эффекты.
- Materials and shaders – материалы определяют каким образом поверхность объекта должна рендериться, в свою очередь материалы используют шейдеры, небольшие программы, выполняющиеся на графическом процессоре и содержащие математические вычисления для определения цвета рендерящихся пикселей.
- HDR, или же широкий динамический диапазон – техника, производящая изображения с большим диапазоном яркости, чем обычный способ изображения.

Пример отрендеренного 2D изображения:

4. Интерфейс

Одним из основных способов взаимодействия пользователя с типичным игровым приложением является пользовательский интерфейс (UI - User Interface). Для быстрой реализации таких интерфейсов Unity предоставляет собственный набор из трех разных систем:

- UI Toolkit – новейшая система UI в Unity, разработанная для работы на разных платформах и основанная на стандартных веб-технологиях. UI Toolkit может применяться не только для создания интерфейсов внутри приложений, но и для реализации расширений редактора Unity.
- Unity UI (или же uGUI) – более старая, основанная на игровых объектах система интерфейсов. Эта система применяется исключительно для создания внутриигровых интерфейсов и не позволяет создавать расширения для редактора. Для размещения интерфейса применяются игровые объекты и компоненты, что позволяет несколько лучше сохранять однородную структуру проекта чем в случае использования интерфейса на основе веб-технологий.
- IMGUI – инструмент построения пользовательского интерфейса, полностью основанный на написании кода. При помощи функции OnGUI разработчик определяет поведение элементов интерфейса, отрисовывает графику и управляет отдельными компонентами. Эта система в первую очередь используется для создания расширений и собственных инструментов редактора Unity. Данный инструмент не рекомендуется использовать для создания внутриигровых пользовательских интерфейсов.

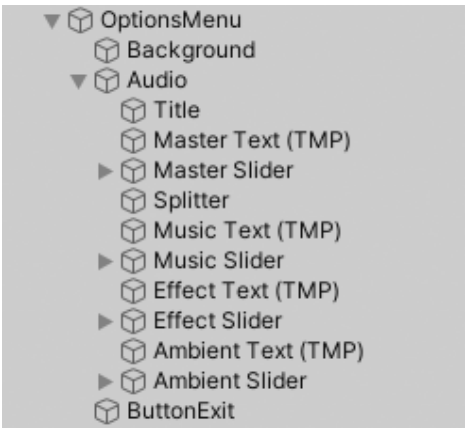
В нашем случае оптимальном выбором будет использование Unity UI (uGUI), так как наше приложение содержит очень небольшое количество интерфейсов. Такой подход позволит нам не заниматься отдельным направлением разработки интерфейса, а все так же использовать подход, основанный на игровых объектах, как и во всех остальных элементах приложения.

Рассмотрим пример создания интерфейса меню настроек. Для этого разберем по частям следующий пример:

На приложенном скриншоте видно меню состоящее из следующих компонентов:

- TextMesh – компонент, отображающий текст и использующий набор разных возможностей (Изменение цвета, изменение шрифта, использование подчеркивания и выделения, настройка отступов и межстрочных интервалов и т.д.)
- Slider – набор объектов и компонентов, позволяющий пользователю перемещать ползунок и задавать значение из установленного диапазона. Работа с такими объектами ведется через callback event OnValueChanged, который позволяет вызывать выбранные функции при изменении положения ползунка.
- Image – компонент, отображающий любое изображение, добавленное в ресурсы приложения.
- Button – компонент, обрабатывающий нажатия пользователя на объект. Работает так же, как и Slider, через callback event-ы.

Общая структура такого меню выглядит следующим образом:

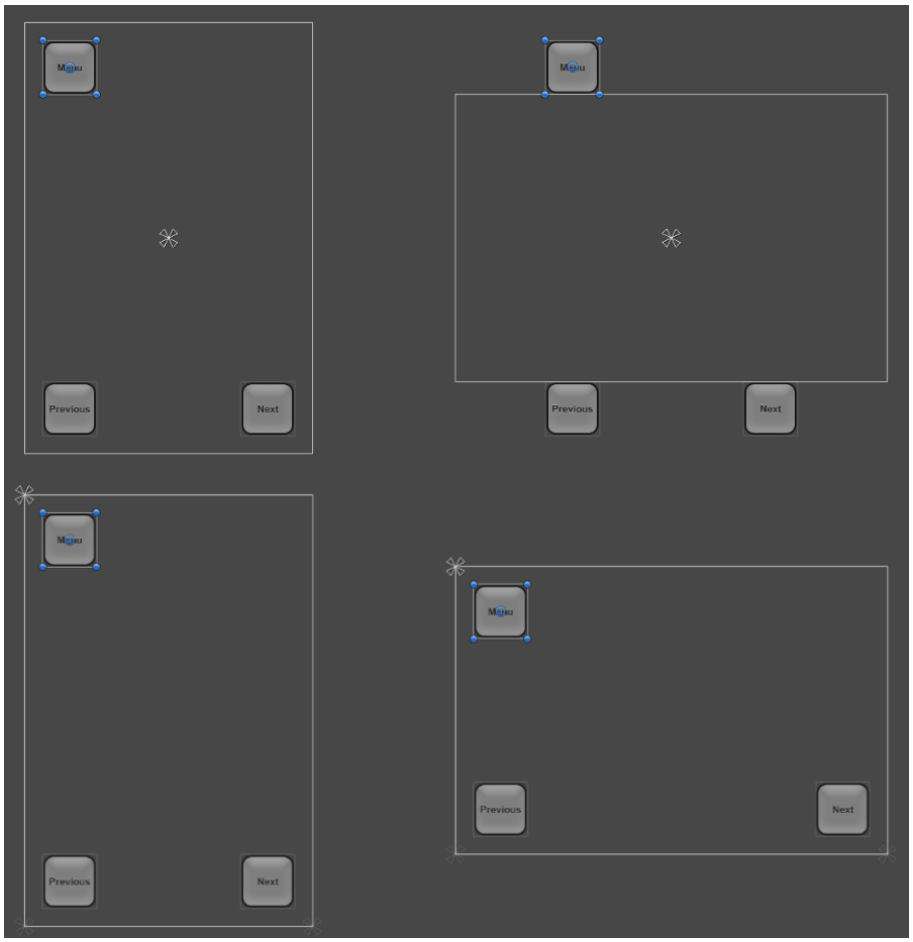


Unity позволяет создавать стандартные объекты через специальное меню. Таким образом при создании, например, объекта Slider не приходится описывать его структуру, состоящую из 6 разных объектов.

Изменение визуала компонентов происходит путем замены стандартных изображений своими собственными. Например, стандартный ползунок выглядит следующим образом:

Размещение элементов интерфейса должно происходить внутри родительского объекта, содержащего компонент Canvas, который будет заниматься отображением всех элементов интерфейса.

Для адаптивности интерфейса на разных соотношениях дисплея для объектов интерфейса используется такое понятие как Anchor (якорь), он определяет относительно чего будет перемещаться объект при изменении соотношения экрана. Приведем пример использования якоря (Слева кнопка в изначальном положении, а справа то, что стало с расположением кнопки после изменения разрешения экрана):



5. Звук

Звук – это неотъемлемая часть игровых приложений. Игра была бы неполной без какого-либо звука, будь то музыкальный фон или звуковые эффекты. Аудиосистема Unity гибкая и мощная. Она может импортировать большинство стандартных аудио форматов и имеет сложные функции для воспроизведения звуков в 3D пространстве, с опциональными эффектами, такими как применение эхо и фильтрации. Unity также может записывать аудио из любого доступного микрофона на компьютере пользователя, для использования во время игры или для хранения и передачи.

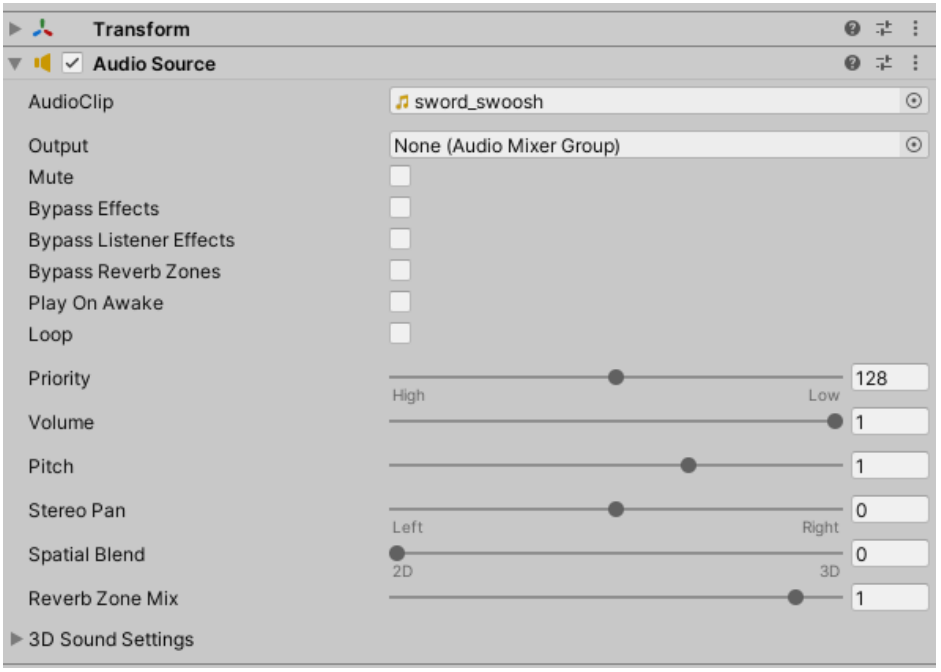
Unity может импортировать файлы в различных форматах, например, WAV, MP3 или Ogg так же, как и другие ресурсы. Импортирование аудио файла создает аудиоклип (Audio Clip), который затем применяется при воспроизведении звука.

Для имитации эффекта пространственного затухания, необходимо, чтобы звуки исходили из компонентов Audio Source, прикрепленных к объектам. Затем, эти звуки обрабатываются компонентом Audio Listener, прикрепленным к другому объекту, чаще всего, к камере. Затем звуковая система Unity может имитировать эффекты ревербации и пространственного положения источника от слушателя и проигрывать их для пользователя соответствующим образом. Относительная скорость объектов источника и слушателя также может быть использована для имитации эффекта Допплера для дополнительной реалистичности.

Unity не может рассчитать эхо только исходя из геометрии сцены, но может имитировать его, используя аудио фильтры (Audio Filters). Например, Echo фильтр для звука, который предназначен для звучания из пещеры, либо фильтр Reverb Zone для объектов, которые могут двигаться внутрь и наружу из области сильного эхо.

Кроме самой звуковой системы, Unity предоставляет специальные звуковые микшеры (Audio Mixers), они позволяют смешивать различные звуки, применять эффекты, управлять громкостью и т.д.

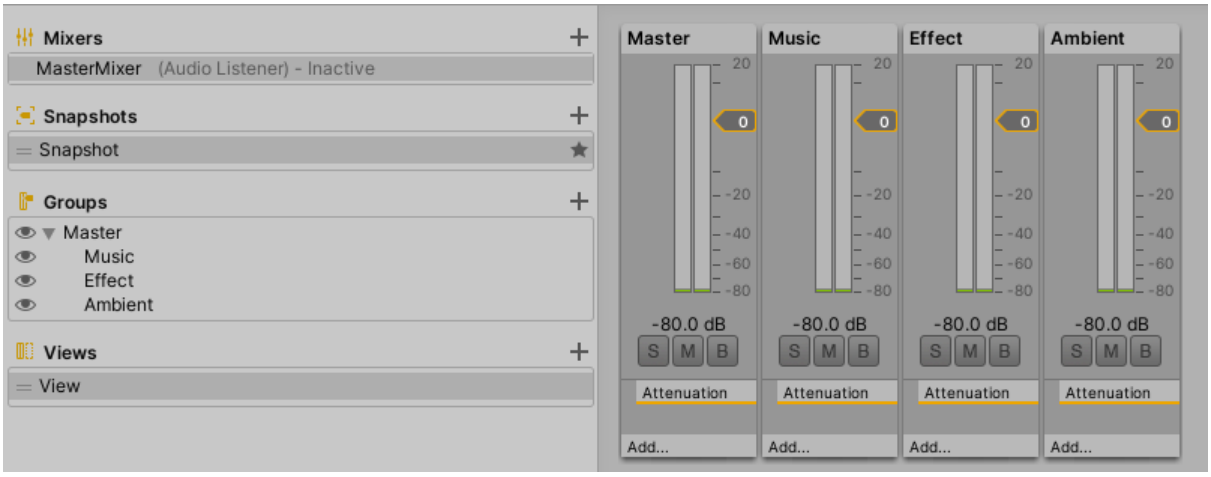
Приведем пример создания звукового эффекта. Для этого создадим объект, добавим в него компонент AudioSource и выберем для него любой звук:



После добавления компонента нам будут доступны для настройки большое количество опций: громкость, приоритет, стерео смещение, активация на старте сцены и т.д.

Важной настройкой является Spatial Blend – он определяет насколько сильно данный источник звука подвержен пространственному затуханию. Если мы установим этот ползунок в 1, то звук будет полностью зависеть от расположения слушателя и затухать с расстоянием.

Кроме того, для управления группами звуков существует Output, он определяет общую настройку для всех звуков, ссылающихся на него. Пример такого объекта Audio Mixer.



6. 2D Физика

Unity имеет отдельный движок для 2D физики, оптимизированный специально для взаимодействия плоских объектов. Физический движок симулирует поведение динамически движущихся объектов, корректно определяет ускорения, коллизии объектов, гравитацию, и применение различных сил.

Rigidbody2D – это основной компонент физического движка, определяющий физическое поведение для объекта, к которому он прикреплен. С использованием Rigidbody2D, объекты немедленно начинают реагировать на гравитацию, а при наличии компонентов Collider2D, будут корректно реагировать на столкновения.

Иногда необходимо, чтобы объекты корректно создавали препятствия для других, но при этом сами не подвергались физическому воздействию. Для таких случаев существует режим Static Rigidbody. Таким телом, может быть, например, пол, целые здания, камни и т.п.

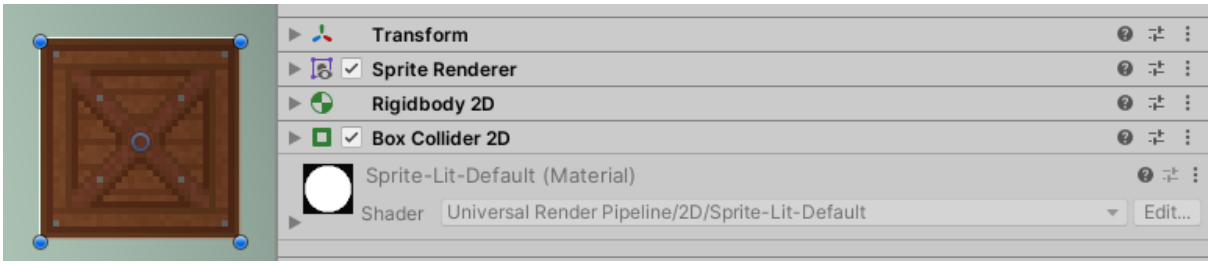
Также может оказаться, что тело не должно иметь физических коллизий, но все еще должно регистрировать события столкновения, нахождения в триггере и т.д. В таком случае используется режим Kinematic Rigidbody, означающий, что объект будет изменять свое положение исходя из какого-то скрипта, но все еще должен регистрировать все события.

Для того, чтобы определить свойства разных объектов, такие как трение и степень отскока, в Unity существуют физические материалы. Несколько таких ассетов создается внутри приложения, а затем они могут быть использованы, чтобы определять свойства объектов, например, дерева, камня, резины и т.д.

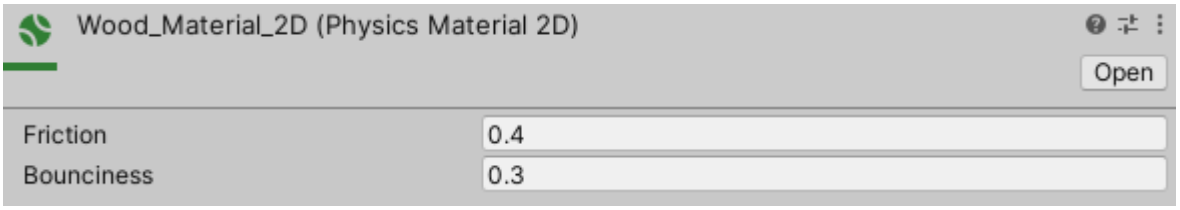
Приведем пример создания физического объекта, корректно взаимодействующего с окружением и имеющего особые физические свойства. Для этого создадим новый 2D объект со спрайтом ящика.

Показателем того, что объект обрабатывается физическим движком является компонент RigidBody2D, а для того, чтобы он мог обрабатывать

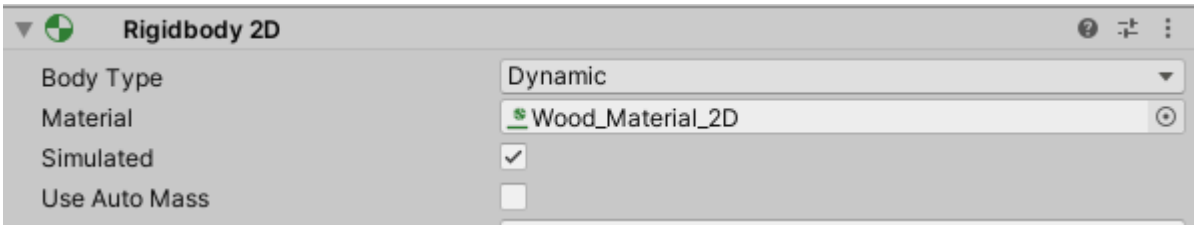
коллизии необходим любой компонент типа Collider2D. Пример итогового объекта:



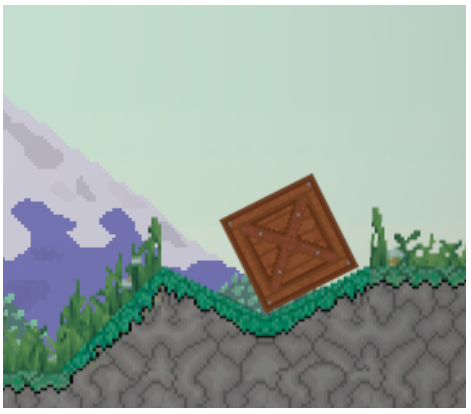
Хотя такой объект уже сейчас будет взаимодействовать с другими физическими объектами, он все еще не имеет настроенного физического материала и будет применен стандартный. Для того, чтобы добавить в проект материал, необходимо создать ресурс вида Physics Material 2D и задать значения трения (Friction) и степени отскока (Bounciness):



Материал прикрепляется к компоненту Rigidbody 2D:

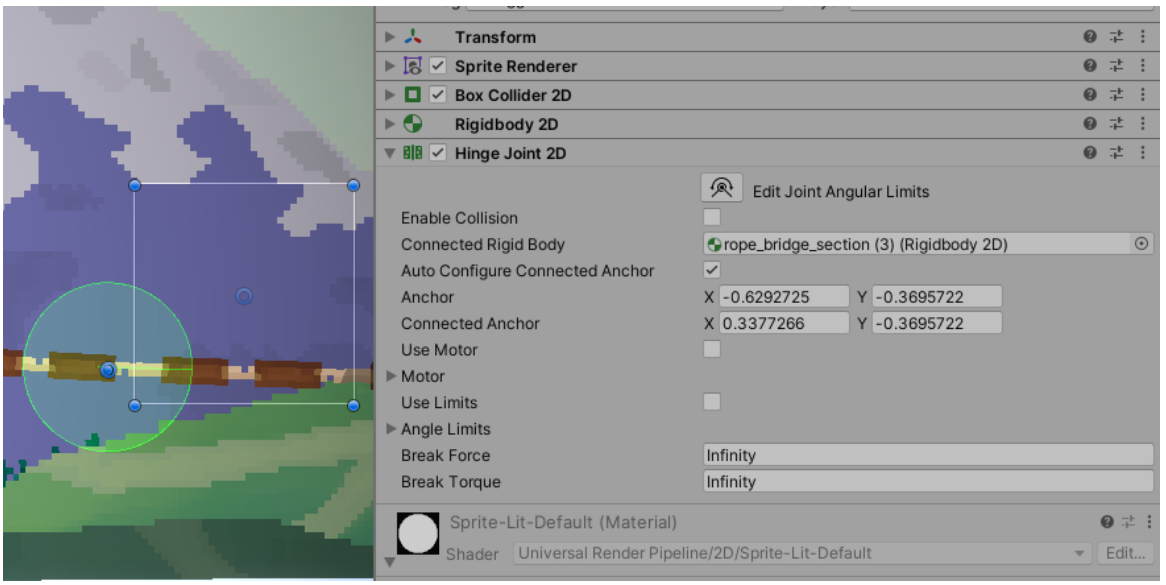


На выходе получается физический объект корректно взаимодействующий с остальными объектами и при этом обладающий собственными свойствами:



Еще одним важным компонентом физического движка являются соединения (Joints), они, как следует из названия, соединяют физические объекты между собой или с какой-либо точкой в пространстве. По сути, соединения каким-либо образом создают ограничения для движения в пространстве для физических объектов; например, некоторые соединения ограничивают движение по определенной линии, некоторые позволяют объектам находится только на определенном расстоянии от точки соединения и т.п.

Например, для создания подвесного моста можно использовать шарнирное соединение (Hinge Joint), в таком случае мост делится на сегменты, а каждый сегмент соединяется с предыдущим с помощью компонента Hinge Joint 2D. В итоге получается подвесной мост, корректно реагирующий на физическое воздействие (Например, он будет прогибаться под весом упавшего на него ящика). Пример такого моста и его сегмента:



7. Игровая логика

Самая важная часть любой видеоигры – игровая логика. Не так важно, как приложение будет выглядеть или как будет звучать, ведь если в приложение нельзя играть, то оно, очевидно, и не может называться игрой.

Приведем основные примеры кода, отвечающие за важные элементы приложения.

Передвижение

Самой важной частью любого 2D платформера является передвижение. Персонаж игрока должен уметь передвигаться, прыгать, проваливаться под платформы и, даже, сражаться.

Для регистрации нажатия клавиш используется встроенный в Unity класс Input:

```
Frequently called  2 usages  Timofey
public static Vector2 GetMove(){...}

Frequently called  1 usage  Timofey
public static bool GetJump() => Input.GetKeyDown(KeyCode.Space);

Frequently called  1 usage  Timofey
public static bool GetContinuousJump() => Input.GetKey(KeyCode.Space);

Frequently called  1 usage  Timofey
public static bool GetEvade() => Input.GetKeyDown(KeyCode.LeftShift);

Frequently called  1 usage  Timofey
public static bool GetCombatMode() => Input.GetKeyDown(KeyCode.Tab);

Frequently called  1 usage  Timofey
public static bool GetAttack() => Input.GetKeyDown(KeyCode.Mouse0);

Frequently called  2 usages  Timofey
public static bool GetInteract() => Input.GetKeyDown(KeyCode.E);
```

Получив нажатые клавиши, мы затем может использовать их, чтобы заставить персонажа двигаться. Для этого мы используем подключенный класс Movement Controller, который использует Raycast, чтобы определить возможно ли двигаться в ту сторону, в которую сейчас движется персонаж. Движение персонажа определяется с помощью вектора скорости. Пример задания такого вектора:

```
private void UseUsualMovement()
{
    Movement.ignoreOneWayPlatformsThisFrame = ToIgnorePlatform;

    _playJumpAnimation = ToJump && Movement.IsGrounded;
    if (_playJumpAnimation)
    {
        _velocity.y = Mathf.Sqrt(f: 2f * jumpHeight * gravity);
        _wasToContinueJump = true;
    }

    if (_wasToContinueJump && !ToContinueJump)
    {
        if (_velocity.y > 0)
        {
            _velocity.y -= jumpManualDumping;
            _velocity.y = _velocity.y < 0 ? 0 : _velocity.y;
        }

        _wasToContinueJump = false;
    }

    _velocity.x = moveSpeed * MoveX;
}
```

Регистрация урона

Еще одной важной системой является система нанесения/получения урона. Для реализации подобной системы нам необходимо создать два объекта. Каждый из них должен содержать Collider2D, а один из них Rigidbody2D. Хотя бы один из них должен быть триггером. После этого остается добавить тому, что будет наносить урон класс Damage Deliver, а принимающему урон Damage Receiver. Пример класса наносящего урон:

```
public class DamageDeliver : MonoBehaviour
{
    public DamageType type = DamageType.LightDamage;  HeavyDamage
    public DamageStats stats;  Changed in 3 assets

    private Collider2D _damageCollider;

     Event function  Timofey
    private void Start()
    {
        _damageCollider = GetComponent<Collider2D>();
        _damageCollider.enabled = false;
    }

     Event function  Timofey
    private void OnTriggerEnter2D(Collider2D otherCollider)
    {
        var receiver = otherCollider.GetComponent<DamageReceiver>();
        if (receiver != null) receiver.ReceiveDamage(stats, type);
    }
}
```

Класс принимающий урон в свою очередь должен только содержать метод Receive Damage, уменьшающий внутренний счетчик после каждой активации.

Активация объектов

Зачастую в играх необходимо что-либо активировать, например, какой-то предмет, чтобы переместить его в инвентарь, или дверь, чтобы зайти в нее. Для этого реализуется класс, который будет проверять нахождение рядом предмета и активировать, а также класс представляющий собой такие объекты. Пример класса активатора и интерфейса.

```
private void OnTriggerStay2D(Collider2D other)
{
    if (isLocked || _interacts.Count == 0)
    {
        interactButton.SetActive(false);
        return;
    }

    interactButton.SetActive(true);
    if (InputUtil.GetInteract())
        _interacts[0].Interact();
}

Event function  Timofey
private void OnTriggerEnter2D(Collider2D other)
{
    var interactive = other.GetComponent<IInteractive>();
    if (interactive != null)
        _interacts.Add(interactive);

}

Event function  Timofey
private void OnTriggerExit2D(Collider2D other)
{
    var interactive = other.GetComponent<IInteractive>();
    if (interactive != null)
    {
        _interacts.Remove(interactive);
        if (_interacts.Count == 0)
            interactButton.SetActive(false);
    }
}
```

```
namespace Environment.Interactive
{
    6 usages  3 inheritors  Timofey
    public interface IInteractive
    {
        1 usage  2 implementations  Timofey
        public void Interact();
    }
}
```

Подгрузка файлов

Еще одним важным аспектом является необходимость загружать какие-либо объекты с компьютера игрока. Это можно сделать разными способами: сериализация/десериализация, обычные текстовые объекты, json, xml и т.д.

Мы рассмотрим загрузку файлов с помощью xml. Для этого необходимо реализовать классы, отражающие структуру .xml файла, и контейнер, который будет подгружать сам файл, преобразовывая его данные в поля классов.

Классы отображающие структуру xml файла:

```
public class CollectibleItemData
{
    [XmlElement] public int ID;
    [XmlElement] public string Name;
    [XmlElement] public string Description;
    [XmlElement] public string PathSpriteFull;
    [XmlElement] public string PathSpriteMini;

    [XmlArray(elementName:"Types")]
    [XmlArrayItem(elementName:"Type")]
    public List<string> Types;
}
[XmlRoot(elementName:"Collectibles")]
3 usages  Timofey
public class CollectibleData
{
    [XmlArray(elementName:"Items")]
    [XmlArrayItem(elementName:"Item")]
    public List<CollectibleItemData> Items;
}
```

Класс являющийся контейнером этих классов:

```
public static class CollectiblesContainer
{
    private const string CollectiblesPath = "Data/items";
    private static Dictionary<int, CollectibleItemData> _collectibles;

    1 usage  Timofey
    public static CollectibleItemData GetItem(int id)
    {
        return GetCollectibles()[id];
    }

    1 usage  Timofey
    private static Dictionary<int, CollectibleItemData> GetCollectibles()
    {
        if (_collectibles != null)
        {
            return _collectibles;
        }

        var store :CollectibleData = GetCollectiblesFromXML();
        _collectibles = new Dictionary<int, CollectibleItemData>();

        foreach (var item in store.Items)
        {
            var regex = new Regex( pattern: @"[\s]{2,}");
            item.Description = regex // Regex
                .Replace( input: item.Description,  replacement: "")
                .Replace( oldValue: "\\n",  newValue: "\n")
                .Trim(); // string
        }

        foreach (var item in store.Items)
            _collectibles.Add(item.ID, item);

        return _collectibles;
    }

    1 usage  Timofey
    private static CollectibleData GetCollectiblesFromXML()
    {
        var serializer = new XmlSerializer(typeof(CollectibleData));
        var xmlText = Resources.Load(CollectiblesPath) as TextAsset;
        if (xmlText == null)
            throw new FileLoadException( message: $"File can't be found: {CollectiblesPath}");
        var s = serializer.Deserialize(new StringReader(xmlText.text)) as CollectibleData;
        return s;
    }
}
```

Заключение

В современном мире создание видеоигр является одним из наиболее крупных сегментов индустрии развлечений. Масштабы игровой индустрии сопоставимы, например, с киноиндустрией. А по скорости роста за последние пять лет индустрия видеоигр существенно ее опережала.

Геймдев или разработку игр невозможно рассматривать обособленно от индустрии компьютерных игр в целом. Непосредственно создание игр – это только часть комплексной «экосистемы», обеспечивающей полный жизненный цикл производства, распространения и потребления таких сложных продуктов, как компьютерные игры.

2D - платформеры в настоящие время вновь набирают популярность, так как они являются относительно нетребовательны к невероятно огромной команде из сотен человек, как многие 3D AAA игры. Это позволяет многим независимым разработчикам преподносят интересные и креативные механики в современное море видеоигр.

Данная курсовая работа была направлена на изучение технологий игрового движка Unity, а также его особенностей. В ходе данной работы был разработан 2D-Action платформер, демонстрирующий применение всех описанных ранее технологий.

Было приведено полное описание Unity: история его развития, сравнение с конкурентами, инструментов для разработки, будущих перспектив. А также реальное применение на практике.

Список литературы

1. https://docs.unity3d.com/2020.2/Documentation/Manual/
2. Joe Hocking ”Unity in Action Multiplatform Game Development in C# with Unity 5”, 2016
3. Jonathon Manning ”Unity Game Development Cookbook: From the Basics to Virtual Reality ”, 2017
4. Alan Thorn ”Animation Essentials”, 2015
5. Clifford Peters, Aung Sithu Kyaw, Dr. Davide Aversa ”Unity Artificial Intelligence Programming - 4th Edition”, 2021
6. Harrison Ferrone ”Learning C# by Developing Games with Unity 2019: Code in C# and build 3D games with Unity, 4th Edition”, 2020
7. Franz Lanzinger ”Classic Game Design Using Unity”, 2013
8. Robert Nystro ”Game Programming Patterns”, 2011
9. Kenneth Lammers ”Unity Shaders and Effects Cookbook”, 2014
10. https://www.iguides.ru/main/gadgets/other_vendors/istoriya_igrovykh_dvizhkov_3_unity_3d/

Приложение. Исходный код приведенных классов

PlayerController.cs

```
using UnityEngine;
using Util;

namespace Entity.Player
{
    public class PlayerController : BaseEntityController
    {
        #region Fields and properties

        #region Unity assigns

        [Header("Vertical Movement")]
        public float jumpHeight = 3;
        public float jumpManualDumping = 5;
        public int maxAttacksInFly = 1;
        public int maxEvadesInFly;

        #endregion

        #region Input

        private float MoveX { get; set; }
        private bool ToJump { get; set; }
        private bool ToContinueJump { get; set; }
        private bool ToEvade { get; set; }
        private bool ToIgnorePlatform { get; set; }
        private bool ToInteract { get; set; }
        private bool ToAttack { get; set; }
        private bool ToParry { get; set; }
        private bool IsInParryMode { get; set; }

        #endregion

        private Vector2 _velocity;

        private bool _wasToContinueJump;
        private bool _playJumpAnimation;
        private bool _notMoveThisFrame;

        private bool _wasEvading;
        private bool _isEvadingForbidden;
        private float _evadeForbidTime;
        private int _evadesInFly;

        private bool _toAttackLight;
        private bool _toAttackHeavy;
        private int _attacksInFly;

        public bool IsInteracting
        {
            get => !IsAttacking && ToInteract;
            set => ToInteract = value;
        }

        #endregion

        #region Unity behaviour

        protected override void Start()
        {
            base.Start();
            Cursor.lockState = CursorLockMode.Locked;
        }
```

```
private void Update()
{
    // when game on pause
    if (Time.deltaTime == 0) return;

    GetControls();
    UpdateCounters();
    CorrectControls();
    UpdateMovement();
    if (!IsLocked)
    {
        UpdateDirection();
        UpdateTimer();
        UpdateCombatState();
    }
    UpdateAnimation();
}

private void GetControls()
{
    MoveX = InputUtil.GetMove().x;
    ToJump = InputUtil.GetJump();
    ToContinueJump = InputUtil.GetContinuousJump();
    ToEvade = InputUtil.GetEvade();

    ToIgnorePlatform = InputUtil.GetIgnorePlatform() ;
    ToInteract ^= InputUtil.GetInteract();

    ToAttack = InputUtil.GetAttack();
    ToParry = InputUtil.GetParry();
    IsInParryMode ^= InputUtil.GetCombatMode();
}

private void UpdateCounters()
{
    switch (Movement.IsGrounded)
    {
        case false when (ToAttack || ToParry):
            _attacksInFly++;
            break;
        case false when ToEvade:
            _evadesInFly++;
            break;
        case true:
            _attacksInFly = 0;
            _evadesInFly = 0;
            break;
    }
}

private void CorrectControls()
{
    if (IsAttacking || _attacksInFly > maxAttacksInFly)
        ToAttack = ToParry = false;
    if (_isEvadingForbidden || IsEvading || _evadesInFly > maxEvadesInFly)
        ToEvade = false;
}

// handles character movement and jumping using movement controller
private void UpdateMovement()
{
    _velocity.y -= gravity * Time.deltaTime; // (m/s^2)

    if (IsLocked)
        UseLockedMovement();
```

```
        else if (IsAttacking)
            UseAttackMovement();
        else if (IsEvading)
            UseEvadeMovement();
        else
            UseUsualMovement();

        var move = (Vector3)_velocity * Time.deltaTime;
        Movement.Move(move);
        _velocity = Movement.Velocity;
    }

    // flips character is it's necessary
    private void UpdateDirection()
    {
        if (IsAttacking || IsEvading) return;
        var input = InputUtil.GetMove();
        if (input.x > 0 && !IsFacingRight || input.x < 0 && IsFacingRight)
            FlipDirection();
    }

    private void UpdateTimer()
    {
        if (_wasEvading && !IsEvading)
        {
            _isEvadingForbidden = true;
            if (_evadeForbidTime < waitForEvadeTime)
                _evadeForbidTime += Time.deltaTime;
            else
            {
                _evadeForbidTime = 0;
                _isEvadingForbidden = false;
                _wasEvading = false;
            }
        }
    }

    // starts attack
    private void UpdateCombatState()
    {
        _toAttackHeavy = ToParry;
        _toAttackLight = !ToParry && ToAttack;

        if (IsAttacking || _attacksInFly > maxAttacksInFly)
        {
            _toAttackHeavy = false;
            _toAttackLight = false;
        }
    }

    private void UpdateAnimation()
    {
        var velocityScaleX = GetMoveScale(_velocity.x, moveSpeed);
        var velocityScaleY = _velocity.y;
        var inFall = !IsGroundedAfterSlope;
        var inAttack = IsAttacking;
        var toEvade = ToEvade && !IsAttacking;
        var toJump = _playJumpAnimation;
        var toAttackLight = _toAttackLight;
        var toAttackHeavy = _toAttackHeavy;

        SetAnimationState(
            velocityScaleX, velocityScaleY, inFall,
            inAttack, toAttackLight, toAttackHeavy,
            toJump, toEvade
```

```
                );
        }

        #region Movement behaviour

        private void UseUsualMovement()
        {
            Movement.ignoreOneWayPlatformsThisFrame = ToIgnorePlatform;

            _playJumpAnimation = ToJump && Movement.IsGrounded;
            if (_playJumpAnimation)
            {
                _velocity.y = Mathf.Sqrt(2f * jumpHeight * gravity);
                _wasToContinueJump = true;
            }

            if (_wasToContinueJump && !ToContinueJump)
            {
                if (_velocity.y > 0)
                {
                    _velocity.y -= jumpManualDumping;
                    _velocity.y = _velocity.y < 0 ? 0 : _velocity.y;
                }

                _wasToContinueJump = false;
            }

            _velocity.x = moveSpeed * MoveX;
        }

        private void UseLockedMovement()
        {
            _velocity.x = 0;
        }

        private void UseAttackMovement()
        {
            // this time we use direct IsGrounded property cause we don't need to consider slop
            if (Movement.IsGrounded) _velocity.x = 0;
        }

        private void UseEvadeMovement()
        {
            var minus = (IsFacingRight) ? 1 : -1;
            _velocity = new Vector2(minus * evadeSpeed, 0);
            _wasEvading = true;
        }

        #endregion

        #endregion
    }
}
```

DamageDeliver.cs

```
using UnityEngine;

namespace Entity.Damage
{
    [RequireComponent(typeof(Collider2D))]
    public class DamageDeliver : MonoBehaviour
    {
        public DamageType type = DamageType.LightDamage;
        public DamageStats stats;

        private Collider2D _damageCollider;

        private void Start()
        {
            _damageCollider = GetComponent<Collider2D>();
            _damageCollider.enabled = false;
        }

        private void OnTriggerEnter2D(Collider2D otherCollider)
        {
            var receiver = otherCollider.GetComponent<DamageReceiver>();
            if (receiver != null) receiver.ReceiveDamage(stats, type);
        }
    }

    public enum DamageType
    {
        LightDamage,
        HeavyDamage,
        PierceDamage
    }
}
```

InteractionController.cs

```
using System.Collections.Generic;
using Environment.Interactive;
using UnityEngine;
using Util;

namespace Entity.Player
{
    public class InteractionController : MonoBehaviour
    {
        private List<IInteractive> _interacts;

        [Header("External")]
        public GameObject interactButton;
        public bool isLocked;

        private void Start()
        {
            _interacts = new List<IInteractive>();
            interactButton.SetActive(false);
        }

        private void OnTriggerStay2D(Collider2D other)
        {
            if (isLocked || _interacts.Count == 0)
            {
                interactButton.SetActive(false);
                return;
            }

            interactButton.SetActive(true);
            if (InputUtil.GetInteract())
                _interacts[0].Interact();
        }

        private void OnTriggerEnter2D(Collider2D other)
        {
            var interactive = other.GetComponent<IInteractive>();
            if (interactive != null)
                _interacts.Add(interactive);

        }

        private void OnTriggerExit2D(Collider2D other)
        {
            var interactive = other.GetComponent<IInteractive>();
            if (interactive != null)
            {
                _interacts.Remove(interactive);
                if (_interacts.Count == 0)
                    interactButton.SetActive(false);
            }
        }
    }
}
```

Исходный код: https://github.com/Tshmofen/Course-Platformer-Unity