### Лабораторная работа №1. Знакомство с игровым движком Unity. Создание первого 2D проекта.

**Цель работы**: изучить теоретические сведения, ознакомится с программным приложением Unity. Создать 2D проект.

**1.1. Общие теоретические сведения**

Unity — профессиональный игровой движок, который используется при создании видеоигр для различных платформ.

Любой игровой движок предоставляет множество функциональных возможностей, которые задействуются в различных играх. Реализованная на этом движке игра получает все эти функциональные возможности, кроме того, добавляются ее собственные игровые ресурсы и код игрового сценария. Unity предлагает моделирование физических сред, карты нормалей, преграждение окружающего света в экранном пространстве (Screen Space Ambient Occlusion, SSAO), динамические тени и др. Список можно продолжать долго. Подобным набором функциональных возможностей могут похвастаться многие игровые движки, но у Unity есть два основных преимущества перед другими передовыми инструментами разработки игр: чрезвычайно производительный визуальный рабочий процесс и мощная межплатформенная поддержка.

**Достоинства и недостатки Unity**

Инструмент Unity имеет множество достоинств, которые превращают его в замечательное средство разработки игр, но с моей стороны было бы упущением не упомянуть о его недостатках. В частности, сочетание визуального редактора со сложным кодом при всей его эффективности в рамках компонентной системы Unity является нетипичным и может вызвать затруднения. В сложных сценах можно потерять из виду некоторые из присоединенных компонентов. В Unity существует функция поиска, позволяющая обнаруживать присоединенные сценарии, но она могла бы быть и более надежной — порой возникают ситуации, когда для поиска связанных сценариев приходится вручную просматривать все элементы сцены. Такое случается не часто, тем не менее этой кропотливой и трудоемкой работы хотелось бы совсем избежать.

Другим неожиданным и обескураживающим для опытных программистов недостатком является тот факт, что Unity не поддерживает ссылки на внешние библиотеки кода. Все доступные библиотеки, которые вы планируете задействовать, следует вручную скопировать в проект, вместо того чтобы просто дать ссылку на одну папку общего доступа. Отсутствие единой папки с библиотеками затрудняет коллективное использование функционала разными проектами. Это неудобство можно обойти, рационально применяя системы контроля версий, но готовое решение данной проблемы в Unity отсутствует.

**1.2. Начало работы с Unity**

Для начала вам нужно скачать Unity. Сделать это можно с официального сайта (https://unity.com/download). Учебная версия программы бесплатная. После, запустите Unity HUB. Эта подпрограмма позволяет скачивать непосредственно версии движка, документацию, работать с лицензиями и редакторами кода.

Взглянем на интерфейс юнити. Как видите, интерфейс напоминает продукты Adobe и Autodesk.

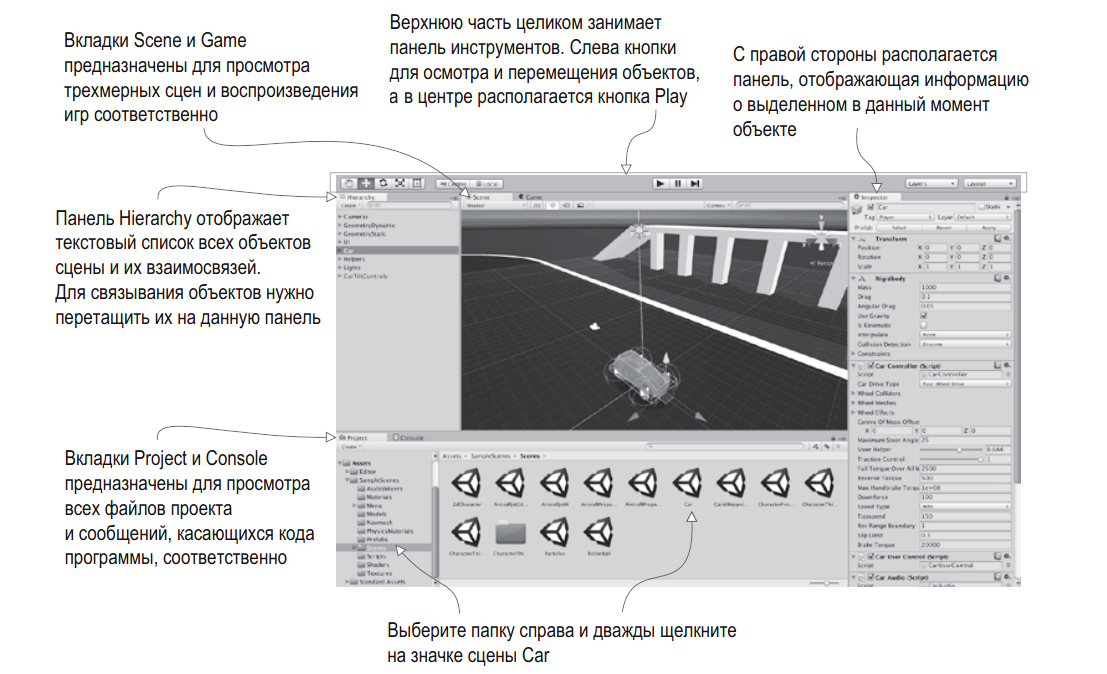
****

Рисунок 1.1 - Фрагменты интерфейса Unity

Интерфейс Unity разбит на несколько частей: вкладка Scene, вкладка Game, панель инструментов, вкладка Hierarchy, панель Inspector, вкладки Project и Console. У каждой части есть собственное предназначение, при этом все они играют важную роль в цикле создания игры:

1. Просмотр файлов выполняется на вкладке Project.
2. Помещенные в трехмерную сцену объекты просматриваются на вкладке Scene.
3. Панель инструментов предоставляет вам элементы управления сценой.
4. Менять взаимосвязи между объектами можно методом перетаскивания на вкладке Hierarchy.
5. Панель Inspector отображает информацию о выделенных объектах, в том числе и о связанном с ними коде.
6. Тестировать полученные результаты можно на вкладке Game, одновременно просматривая сообщения об ошибках на вкладке Console.

Эта компоновка предлагается по умолчанию; все доступные представления помещены на вкладки, которые можно перемещать, можно менять их размер и фиксировать в разных частях экрана. Чуть позже вы поэкспериментируете с выбором компоновки, пока же нам нужно понять назначение каждого элемента интерфейса, поэтому вариант, предлагаемый по умолчанию, является оптимальным.

**1.3. Пример выполнения задания. Создание 2D игры “Пинг-понг”**

Создаем пустой (без ассетов) 2D проект на Unity. Назовем его pong. Сохраним пустую сцену (Верхняя вкладка меню File->Save Scene). Когда спросит имя, можно ввести что-нибудь вроде scene\_main.

Начнем с настройки камеры, чтобы она отображала игру в правильном размере и цвете. Мы можем настроить камеру выбрав её в иерархии справа.

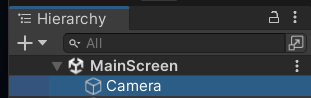


Рисунок 1.2 - Отображение камеры в разделе Hierarchy

Далее можно увидеть настройки данного объекта в инспекторе. Мы поменяем цвет фона (Background) на красный и установим нужный нам размер (Size):

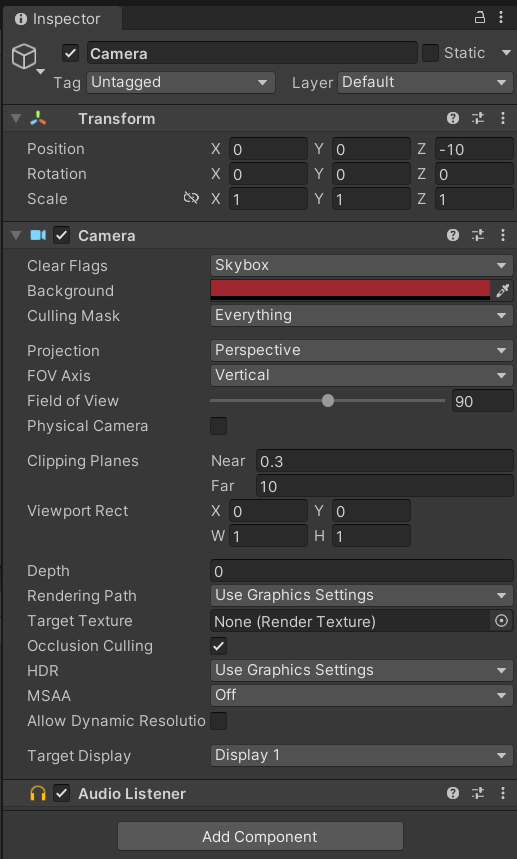


Рисунок 1.3 – Вкладка Inspector камеры

Создадим стены. На вкладке иерархия нажмите правой кнопкой мыши и пройдите по пути на скриншоте.

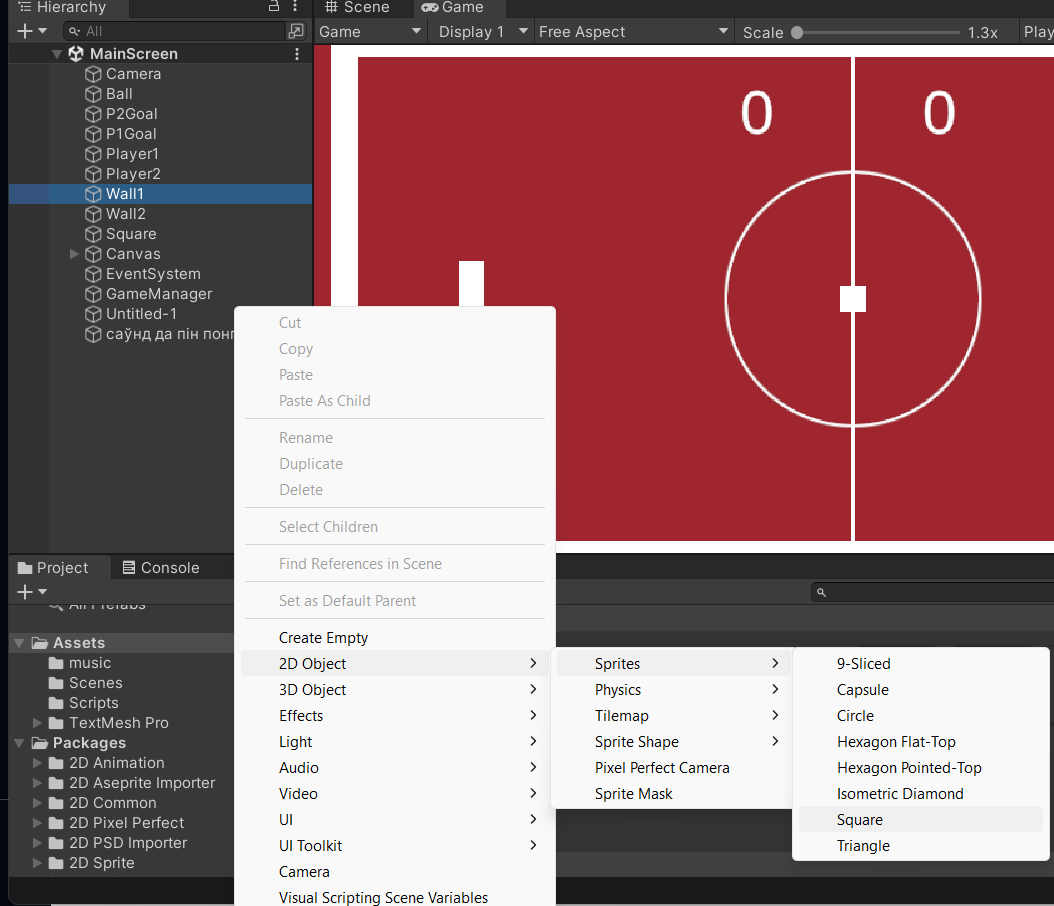


Рисунок 1.4 - Создание спрайтов для стен

Для визуализации поля необходимо создать два типа стен. Верхняя и нижняя стены от которых отскакивает мячик и боковые стены которые являются воротами игроков. Объекты на вкладке иерархии на самом деле являются контейнерами. Внутри каждого такого контейнера есть компоненты: обязательные и необязательные, встроенные и кастомные. Разберём какие встроенные компоненты могут встречаться в лабораторной работе (рис. 1.5 и 1.6). Transform – обязательный компонент любого объекта. Тут можно увидеть местоположение, вращение и масштабирование. Sprite renderer – компонент отвечающий за отображение объекта на сцене. Box Collider – компонент отвечающий за восприятие объекта движком (его физические границы). RigidBody – компонент для движущихся объектов, обычно для сил тяжести, но в данном случае для управления ракеткой.

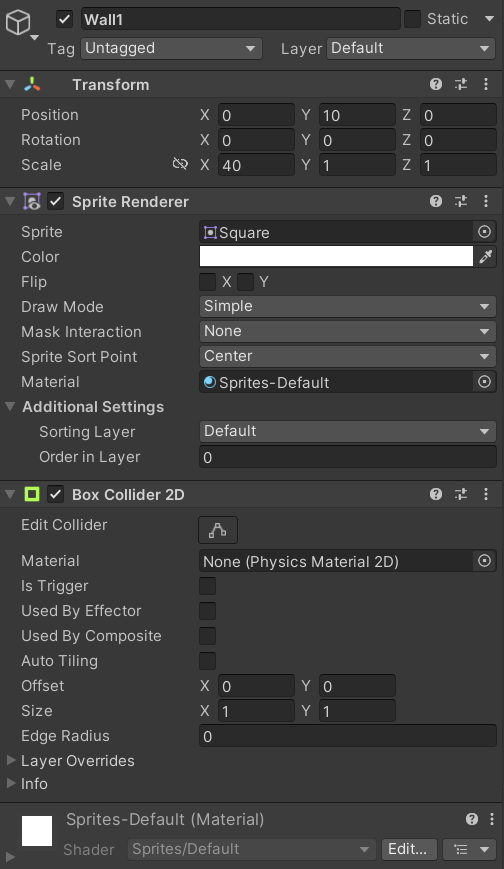


Рисунок 1.5 – Компоненты нижней стены

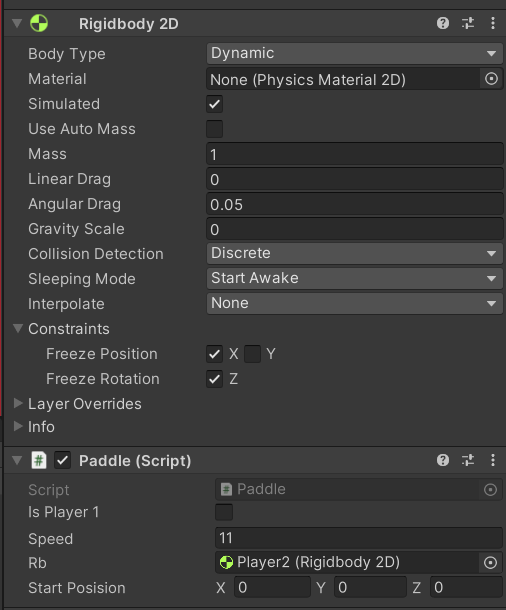


Рисунок 1.6 – Компонент Rigidbody у “ракетки"

По аналогии со стенами сделайте ракетки, но теперь нам нужно добавить к ним кастомный компонент–скрипт. Скрипт–это файл с кодом. В Unity используется язык программирования C#. Этот код позволяет управлять ракеткой с помощью соответствующих осей.

Листинг:

**using** System.Collections;

**using** System.Collections.Generic;

**using** UnityEngine;

**public** **class** Paddle : MonoBehaviour

{

**public** **bool** isPlayer1;

**public** **float** speed;

**public** Rigidbody2D rb;

**public** Vector3 startPosision;

**private** **float** movement;

**void** Start()

{

startPosision = transform.position;

}

**void** Update()

{

**if** (isPlayer1)

{

movement = Input.GetAxisRaw("Vertical");

}

**else**

{

movement = Input.GetAxisRaw("Vertical2");

}

rb.velocity = **new** Vector2(rb.velocity.x, movement \* speed);

}

**public** **void** Reset()

{

rb.velocity = Vector2.zero;

transform.position = startPosision;

}

}

Далее код который отвечает за голы и счёт.

Листинг:

**using** System.Collections;

**using** System.Collections.Generic;

**using** UnityEngine;

**public** **class** GoalScript : MonoBehaviour

{

**public** **bool** isPlayer1Goal;

**private** **void** OnTriggerEnter2D(Collider2D collision)

{

**if**(collision.gameObject.CompareTag("Ball"))

{

**if**(!isPlayer1Goal)

{

Debug.Log("Player 1 Scored...");

GameObject.Find("GameManager").GetComponent<GameManager>().Player1Scored();

}

**else**

{

Debug.Log("Player 2 Scored...");

GameObject.Find("GameManager").GetComponent<GameManager>().Player2Scored();

}

}

}

}

И наконец напишем код который будет управлять поведением мяча.

Листинг:

**using** System.Collections;

**using** System.Collections.Generic;

**using** UnityEngine;

**public** **class** Ball : MonoBehaviour

{

**public** **float** speed;

**public** Rigidbody2D rb;

**public** Vector3 startPosition;

**void** Start()

{

Invoke("Launch", 2f);

Debug.Log("Start");

}

**public** **void** Reset ()

{

rb.velocity = Vector2.zero;

transform.position = startPosition;

Invoke("Launch", 2f);

}

**private** **void** Launch()

{

**float** x = Random.Range(0, 2) == 0 ? -1 : 1;

**float** y = Random.Range(0, 2) == 0 ? -1 : 1;

rb.velocity = **new** Vector2(speed\*x, speed \* y);

}

}

Полный список объектов в иерархии может выглядеть следующим образом (рис. 1.7).

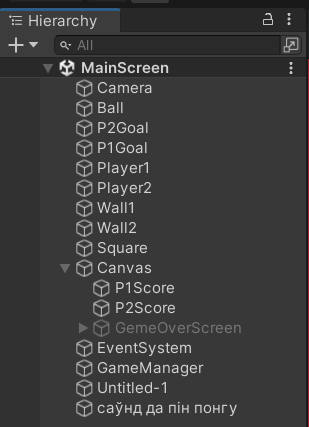


Рисунок 1.7 – Объекты сцены

**1.3. Индивидуальное задание**

На основе теоретических сведений создайте свой индивидуальный 2D проект в Unity - игра “Пинг-понг”. В данной игре должен отображаться счёт и по достижении определенного количества очков, игра должна заканчиваться.

### Лабораторная работа №2. Игра Memory на основе 2D-функциональности.

**Цель работы**: изучить теоретические сведения. Создать в Unity 2D игру “Memory”.

**2.1. Общие теоретические сведения**

Мы собираемся воспроизвести классическую игру Memory. Опишем ее для тех, кто не знаком с правилами. Набор парных карт раскладывается рубашкой вверх. Расположение карт игроку неизвестно. Он может перевернуть любые две карты, пытаясь найти совпадающие. Если открытые карты не совпали, они переворачиваются назад, а игрок делает следующую попытку. На рис. 2.1 показан макет этой игры.

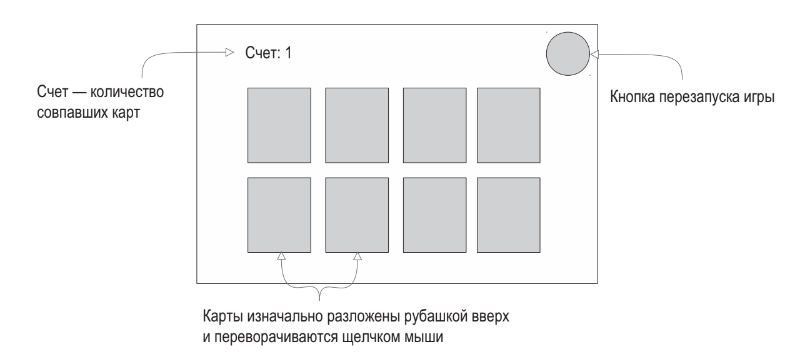


Рисунок 2.1 - Макет игры Memory

**2.2. Пример выполнения задания**

Начнём с создания необходимых спрайтов. Как минимум, нам понадобятся рубашка карты и несколько картинок для передней стороны карт.

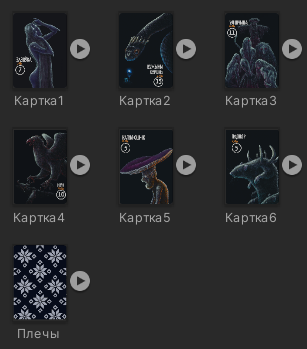


Рисунок 2.1 - Спрайты лицевых и задней сторон карт

Так как все изображения уже импортированы и готовы к использованию, давайте создадим объекты, представляющие карты, которые сформируют ядро нашей игры. В игре Memory карты выкладываются лицевой стороной вниз и переворачиваются только на время после выбора пары карт. Для реализации такой функциональности мы создадим объекты, состоящие из положенных друг на друга спрайтов. Затем мы напишем код, который заставит карты переворачиваться по щелчку мыши.

Чтобы карта могла реагировать на щелчки мыши, ей требуется такой компонент, как коллайдер. У новых спрайтов он отсутствует, поэтому щелкать на них бесполезно. Мы присоединим коллайдер к корневому объекту, а не к рубашке карты, поэтому именно лицевая сторона будет восприимчива к щелчкам. Для этого выделите корневой объект на вкладке Hierarchy (не имеет смысла щелкать на карте в сцене, так как сверху находится рубашка, и именно она будет выделена), а затем щелкните на кнопке Add Component в нижней части панели Inspector. Выделите строку Physics 2D (строка Physics относится к физике трехмерных игр) и выберите вариант Box collider. Кроме коллайдера карте нужен сценарий, который заставит ее реагировать на действия игрока, поэтому давайте напишем его код. Создайте новый сценарий с именем MemoryCard.cs и присоедините его к корневой карте (а не к рубашке).

Итак, теперь мы можем щелкнуть на карте. Подобно методу Update(), функция OnMouseDown() также происходит от класса MonoBehaviour, именно она вызывает реакцию на щелчок мыши.

Листинг:

**using** System.Collections;

**using** System.Collections.Generic;

**using** UnityEngine;

**public** **class** MemoryCard : MonoBehaviour

{

[Tooltip("Плечи карты")]

[SerializeField]

**private** GameObject \_cardBack;

[Tooltip("лицевая часть карты")]

[SerializeField]

**private** Sprite \_ImageCard;

[Tooltip("Сценарий для управления всеми картами")]

[SerializeField]

**private** CardControllr \_cardsController;

[SerializeField]

**private** **int** \_id = 0;

**private** Animation \_animation;

**private** **bool** \_front = **false**;

**private** **void** Awake()

{

\_animation = GetComponent<Animation>();

}

**public** **int** id {

**get** { **return** \_id; }

}

**private** **void** OnMouseDown()

{

**if**(\_front == **false** && \_cardsController.canOpen)

{

\_animation.Play("toFront");

\_front = **true**;

\_cardsController.CardOpened(**this**);

}

}

**public** **void** SetCard(**int** ID, Sprite im)

{

\_id = ID;

GetComponent<SpriteRenderer>().sprite = im;

\_ImageCard = im;

}

**public** **void** Close()

{

\_animation.Play("toBack");

\_front = **false**;

}

**public** **void** Open()

{

\_animation.Play("toFront");

\_front = **true**;

}

}

Мы написали программу, которая сначала отображает рубашку карты, а после щелчка на ней показывает лицевую сторону. Но это всего лишь одна карта, тогда как для игры требуется целый набор, по большей части с разными изображениями. Мы разложим карты, воспользовавшись как парой концепций из предыдущих глав, так и совершенно новыми пока для вас понятиями.

В создаваемой нами игре есть четыре варианта изображений для карт. Все восемь лежащих на столе карт (по две для каждого символа) получаются клонированием одного и того же оригинала, поэтому изначально на всех картах будет один рисунок.

Хотя этот сценарий намного длиннее предыдущих, объяснять тут особо нечего, потому что большинство дополнений представляют собой обычные объявления переменных и математические вычисления. Самым странным фрагментом кода является, вероятно, выражение, начинающее условную инструкцию if (i == 0 && j == 0). Это выражение заставляет либо выбрать исходный объект-карту для первой ячейки нашей сетки, либо клонировать этот объект для остальных ячеек. Так как исходная карта в сцене уже есть, ее копирование на каждом шаге цикла приведет к появлению слишком большого количества карт. Карты раскладываются путем смещения в соответствии с количеством итераций цикла. Для создания полнофункциональной игры Memory нам не хватает проверки совпадений. Хотя у нас есть выложенные на стол карты, открывающиеся при щелчке на них, они никак не связаны друг с другом. Нам же нужно, чтобы каждая открытая пара проверялась на совпадение.

Эта абстрактная логическая схема — проверка совпадений и соответствующая реакция — требует, чтобы карты в ответ на щелчок уведомляли сценарий SceneController.

Листинг:

**using** System.Collections;

**using** System.Collections.Generic;

**using** UnityEngine;

**using** UnityEngine.SceneManagement;

**using** UnityEngine.UI;

**public** **class** CardControllr : MonoBehaviour

{

[Header("Настройки спавна")]

**public** **const** **int** gridRows = 2;

**public** **const** **int** gridColumns = 6;

**public** **const** **float** offSetX = 2.5f;

**public** **const** **float** offSetY = 3f;

[Header("Параметры карты")]

[Tooltip("ССылка на карты в сцене")]

[SerializeField]

**private** MemoryCard \_originalCard;

[Tooltip("Ссылки на лицевую часть карт")]

[SerializeField]

**private** Sprite[] \_imCards;

[Tooltip("Счёт")]

[SerializeField]

**private** Text scoreText;

[Tooltip("Сколько времени всё видно в начале")]

[Range(0f, 5f)]

[SerializeField]

**private** **float** visibleTime=0.3f;

**private** **int** \_score = 0;

**private** MemoryCard \_firstOpenCard;

**private** MemoryCard \_secondOpenCard;

**private** **void** Start()

{

Vector3 startPos = **new** Vector3(-6f, -3.5f, 0);

**int**[] numID = { 0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5 };

numID = ShuffleArray(numID);

**for** (**int** i = 0;i<gridColumns;i++)

{

**for** (**int** j = 0; j < gridRows; j++)

{

MemoryCard card;

**if** (i == 0 && j == 0)

{

card = \_originalCard;

}

**else**

{

card = Instantiate(\_originalCard) **as** MemoryCard;

}

**int** index = j \* gridColumns + i;

**int** id = numID[index];

card.SetCard(id, \_imCards[id]);

**float** posX = (offSetX \* i) + startPos.x;

**float** posY = (offSetY \* j) + startPos.y;

card.transform.position = **new** Vector3(posX, posY, startPos.z);

StartCoroutine(ShawCardsCoroutine(visibleTime, card));

}

}

}

**private** **int**[] ShuffleArray(**int**[] array)

{

**int**[] newArray = array.Clone() **as** **int**[];

**for**(**int** i=0; i<newArray.Length; i++)

{

**int** num = newArray[i];

**int** rand = Random.Range(i, newArray.Length);

newArray[i] = newArray[rand];

newArray[rand] = num;

}

**return** newArray;

}

**public** **bool** canOpen

{

**get** { **return** \_secondOpenCard == **null**; }

}

**public** **void** CardOpened(MemoryCard card)

{

**if**(\_firstOpenCard == **null**)

\_firstOpenCard = card;

**else**

{

\_secondOpenCard = card;

StartCoroutine(CheckMathCoroutine());

}

}

**private** IEnumerator CheckMathCoroutine()

{

**if** (\_firstOpenCard.id == \_secondOpenCard.id)

{

\_score++;

scoreText.text = \_score.ToString();

**if**(\_score==6)

{

SceneManager.LoadScene("VictoryMenu");

}

\_secondOpenCard = **null**;

\_firstOpenCard = **null**;

}

**else**

{

**yield** **return** **new** WaitForSeconds(0.5f);

\_secondOpenCard.Close();

\_firstOpenCard.Close();

\_secondOpenCard = **null**;

\_firstOpenCard = **null**;

}

}

**private** IEnumerator ShawCardsCoroutine (**float** waitSecond, MemoryCard card)

{

card.Open();

**yield** **return** **new** WaitForSeconds(waitSecond);

card.Close();

}

}

****

Рисунок 2.2 - Пример игры Memory

Мы успешно создали еще одну игру! Разумеется, о «готовности» игры можно говорить только в относительном смысле — вы всегда можете добавлять новые функциональные возможности. Но все, что было намечено в начальном плане, мы реализовали. Многие концепции двухмерной графики применимы и к трехмерным играм, особенно связанные с проверкой состояния игры и с загрузкой игровых уровней. Пришло время изучить новую тему и выполнить новый проект.

**2.3. Индивидуальное задание**

На основе теоретических сведений создайте свой индивидуальный 2D проект в Unity - игра “Memory”. В данной игре должен отображаться счёт, должна быть кнопка рестарта, анимация карт и экран меню после победы.

### Лабораторная работа №3. Создание 2D шутера “Space ship”.

**Цель работы**: изучить теоретические сведения. Создать игру на Unity - 2D шутер “Space ship”.

**3.1. Общие теоретические сведения**

В данной лабораторной работе для фона мы будем использовать такое понятие, как Параллакс-скроллинг. Параллакс-скроллинг (также Параллаксная прокрутка) — техника в [компьютерной графике](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%B3%D1%80%D0%B0%D1%84%D0%B8%D0%BA%D0%B0), когда фоновые изображения перемещаются относительно виртуальной камеры медленнее, чем изображения переднего плана, что создает иллюзию глубины двумерной сцены и ощущение погружения в виртуальный мир.

Техника довольно старая, но в ней тоже есть свои нюансы. Для начала создадим на сцене пустой объект и закинем в него фон. У вложенных объектов нужно выставить по координате X оффсет на размер спрайта. У одного с плюсом, а другого с минусом. В итоге корневой элемент будет по центру, а два вложенных по бокам.

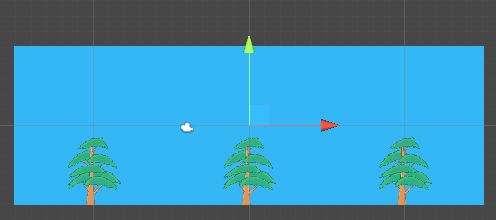


Рисунок 3.1 - Расположение фона на сцене

Теперь нужно написать простенький скрипт для смены положения фона и прикрепить его к корневому объекту и закинуть на него ссылку на камеру.

Листинг:

**public** **class** ParallaxScript : MonoBehaviour {

**private** **float** startPos, length;

**public** GameObject camera;

**public** **float** paralaxEffect;

**void** Start() {

startPos = transform.position.x;

length = GetComponent<SpriteRenderer>().bounds.size.x;

}

**void** Update() {

**float** temp = camera.transform.position.x \* (1 - paralaxEffect);

**float** dist = camera.transform.position.x \* paralaxEffect;

// двигаем фон с поправкой на paralaxEffect

transform.position = **new** Vector3(startPos + dist, transform.position.y, transform.position.z);

// если камера перескочила спрайт, то меняем startPos

**if** (temp > startPos + length)

startPos += length;

**else** **if** (temp < startPos - length)

startPos -= length;

}

}

За счёт 3 одинаковых спрайтов в момент ухода камеры за центральный спрайт мы можем поменять оффсет на ширину спрайта. В Scene view в этом момент мы увидим как весь фон сместится в сторону на ширину спрайта.

Важно учитывать размер камеры. Если охват камеры будет больше, чем размер спрайта, то при перескоке между спрайтами будет вот такая проблема, когда оффсет фона ещё не будет пересчитан.

**3.2. Пример выполнения задания**

Из assets переносим заранее заготовленный корабль на сцену. Переносим корабль перед фонов и настраиваем его положение и положение камеры.

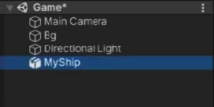


Рисунок 3.2 - Добавление корабля на сцену

Добавим в сцену коллайдер, который будет отвечать за столкновение.

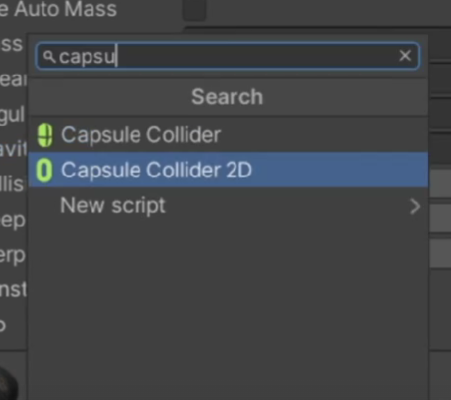


Рисунок 3.3 - Добавление коллайдера к кораблю

Далее напишем скрипт с необходимыми настройками нашего корабля.

Листинг:

**public** **class** PlayerShip : MonoBehaviour

{

[SerializeField] **private** **float** \_speed = 15;

[Serializefield] **private** **float** \_COOlDown = 0.1f;

**public** **int** \_maxHealth = 100;

[SerializeField] **private** **float** shipRollEuler = 45;

[Serializefield] **private** **float** shipRo11Speed = 80;

[SerializeField] **private** **float** \_smothness - 1.2f;

**private** Rigidbody2D rigidbody;

**private** **float** coolDownCurrent = 10;

**private** MeshRenderer MR;

**private** Vector3 \_sizeworldShip;

**private** Controller \_controller;

[HideInInspector] **public** ReactiveProperty<**int**> health = **new** ReactiveProperty<**int**>();

**private** **void** Awake() {

\_rigidbody = GetComponent<Rigidbody2D> ();

mR = GetComponent <MeshRenderer> ();

controller = Controller. Instance;

}

}

**private** vold Start(){

\_health.Value = \_maxHealth

}

Добавьте крипт к кораблю.

Есть два основных способа реализовать стрельбу:

1. Префабы. Добавляет на карту снаряд, которому можно прописать поведение — направление полета, действия при попадании и так далее.
2. Лучи (Raycast). Движок рисует невидимую линию от какой-нибудь точки в заданном направлении и возвращает данные о том, есть ли что-нибудь на пути.

В первую очередь нужно подготовить всё, чтобы персонаж мог стрелять. Начать стоит с создания точки, откуда будет лететь снаряд или направляться луч. Для этого добавьте пустой объект с именем FirePoint и поместите его внутрь персонажа, расположив возле дула его оружия.

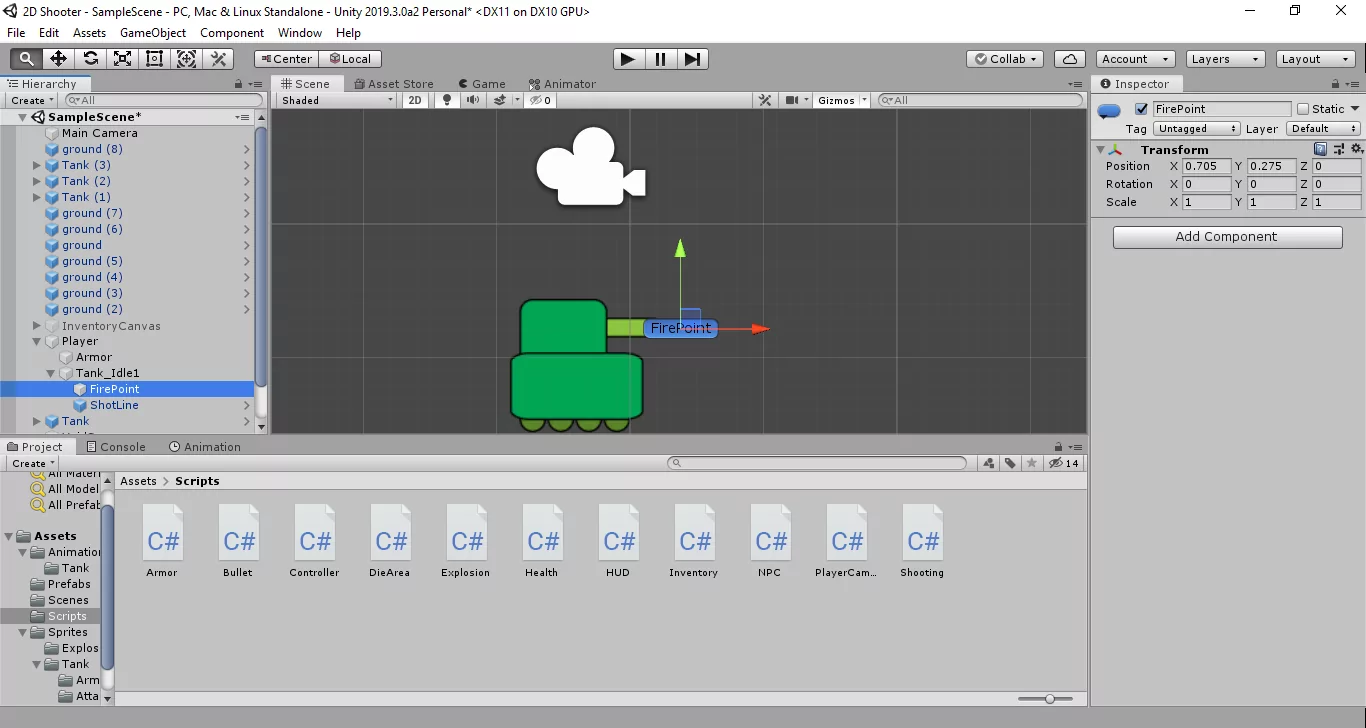


Рисунок 3.4 - Добавление FirePoint

Затем нужно написать код, который позволит персонажу вращаться вместе с этой точкой. Создайте скрипт Controller.cs и прикрепите его к персонажу.

После этого можно приступить к скрипту, который позволит стрелять. Назовите его Shooting.cs, добавьте к персонажу и используйте следующий код.

Листинг:

**public** **class** Shooting : MonoBehaviour

{

**public** GameObject bullet; //Снаряд

**public** Transform firePoint; //Точка, с которой будут отправляться снаряды и лучи

**public** LineRenderer lineRenderer; //Луч

**void** Update()

{

**if** (Input.GetKeyDown(KeyCode.RightControl)) //Если игрок нажал на правый Ctrl

{

//Вызов метода стрельбы снарядами

ShootBullet();

}

}

}

Для начала нужно создать снаряд. Для этого перетащите на карту спрайт и назовите его Bullet.

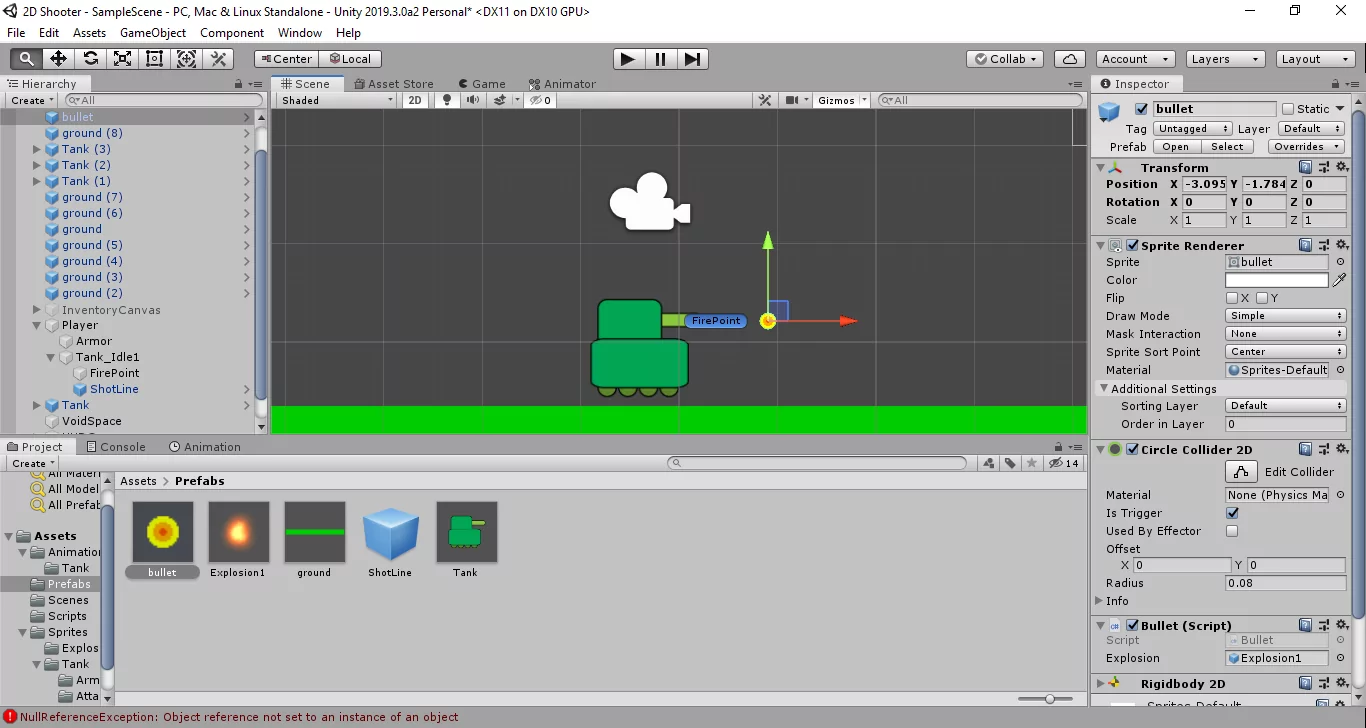


Рисунок 3.5 - Создание снаряда

Добавьте коллайдер с триггером и создайте скрипт Bullet.cs, в котором будут обрабатываться попадания. Сохраните объект в качестве префаба, а потом перетащите его в компонент Shooting.cs. Туда же перетащите FirePoint.

Теперь нужно написать метод, который будет создавать (инстанцировать) новые снаряды на карте:

Листинг:

**void** ShootBullet()

{

Instantiate(bullet, firePoint.position, firePoint.rotation);

}

Пока снаряд остается на месте. Чтобы это исправить, нужно прописать в Bullet.CS этот код.

Листинг:

**public** **class** Bullet : MonoBehaviour

{

**private** Rigidbody2D rb;

**private** **float** speed = 15f;

**private** **int** damage = 20;

**private** **int** life = 0;

**private** **int** lifeMax = 500;

**void** Start()

{

rb = GetComponent<Rigidbody2D>();

rb.velocity = transform.right \* speed; //Изменение скорости

}

**void** Update()

{

life++;

**if** (life >= lifeMax)

{

Explode(); //Если снаряд пролетел определенное расстояние и ни с чем не столкнулся, его нужно удалить, чтобы он не расходовал ресурсы

}

}

**void** OnTriggerEnter2D(Collider2D hitInfo) //Метод, который срабатывает при попадании

{

Explode();

}

**void** Explode()

{

Destroy(gameObject); //Уничтожение объекта

}

}

Теперь снаряд будет лететь и уничтожаться при попадании во что-то.

**3.3. Индивидуальное задание**

На основе теоретических сведений создайте свой индивидуальный 2D проект в Unity - шутер “Space ship”. Игра должна заканчиваться по истечению показателя “жизни” у корабля. По желанию сделать в проекте стрельбу с разной скоростью.

### Лабораторная работа №4. Создание 2D проекта “Rocket Mouse”.

**Цель работы**: изучить теоретические сведения лабораторной работы. Создать 2D игру на Unity “Rocket Mouse”.

**4.1. Общие теоретические сведения**

В Unity имеется компонент Particle System, который имитирует жидкие субстанции наподобие разных жидкостей, облаков и чего-нибудь связанного с огнём путём генерации и анимации в сцене большого количества небольших 2D изображений.

**Свойства**

Компонент Particle System имеет множество настроек и для удобства, инспектор упорядочивает их в сворачиваемые секции или “модули” описываемые ниже. Каждая секция может быть развёрнута и свёрнута при нажатии на заголовок. С левой стороны расположены чекбоксы, которые можно отмечать для включения или отключения тех или иных настроек. Например, если вы не хотите менять размер частиц во время их излучения, вы можете просто снять галочку с параметра Size over lifetime.

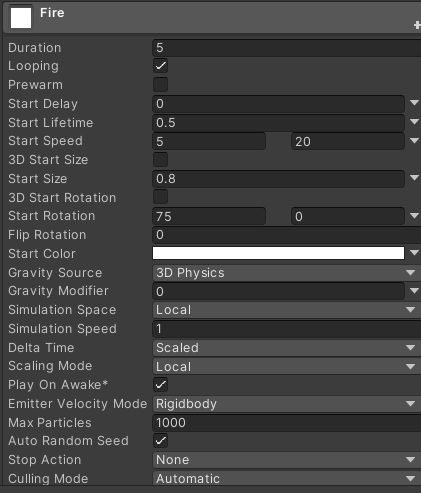


Рисунок 3.1 – Свиток параметров Particle System

Помимо модулей, в инспекторе есть ещё несколько других элементов управления. Кнопка Open Editor отображает опции в отдельном окне редактора, которое также позволяет вам редактировать одновременно несколько систем. А чекбокс Resimulate отвечает за то, будут или не будут тут же применяться изменения свойств к только что созданным системой частицам (в качестве альтернативы существующие частицы будут исчезать, так как они будут единственными новыми частицами, свойства которых были изменены). Кнопка Wireframe отображает контуры меш объектов, чтобы показать находящиеся за ними частицы.

**4.2. Пример выполнения задания**

Начнем с импорта игровых ассетов. Все необходимые для проекта ассеты вы можете скачать в следующем публичном репозитории GitHub: <https://github.com/albertmiro/RocketMouse>

Чтобы добавить все ассеты за один раз выберите обе папки Sprites и Audio из архива и перетащите их в Браузер проекта, расположив рядом с папкой Scenes. Вам придется подождать, пока файлы не обработаются Unity, но это не займет много времени.

**Создание персонажа**

Откройте папку Sprites в Браузере проектов и найдите в этой папке спрайт по имени mouse\_fly. Перетащите его на сцену.

Выберите mouse\_fly в иерархии и сделайте следующие изменения в Инспекторе:

1. Измените название на mouse, поскольку это будет лучше описывать игрового персонажа.
2. Установите значение Position на (0, 0, 0).
3. Добавьте компонент Circle Collider 2D
4. Установите радиус Circle Collider 2D равным 0.5.
5. Добавьте компонент Rigidbody 2D, он поможет нам добавить персонажу физику.
6. Поставьте галочку в чекбоксе Freeze Rotation внутри компонента Rigidbody 2D во вкладке Constraints.

Зелёный круг внутри спрайта персонажа показывает коллайдер, если его не видно - нажмите на объект (персонажа) или переключите режим отображения Gizmos в правом верхнем углу окна вида сцены. Коллайдеры определяют форму, которая используется для определения столкновений с другими объектами.

Вы можете создать сложный коллайдер с большим числом пикселей с помощью компонента Polygon Collider 2D. Несмотря на более точную обработку формы, использование сложных коллайдеров делает трудным для физического движка обнаружение столкновений, что, в свою очередь, приводит к потере производительности. Легче проверить, сталкивается ли круг с прямоугольником, чем, когда сталкиваются два комплексных полигона. Самыми простыми формами коллайдера являются Box, Sphere и Capsule, который чаще всего используется для персонажей.

Rigidbody ставит объект вашей игры под контроль встроенного физического движка, что значительно упрощает разработку небольшого проекта. Без Rigidbody игровой объект не зависит от силы тяжести, поэтому вы не сможете применить на него силу и крутящий момент, если не напишите для этого свои собственные скрипты.

Также вы не сможете обнаружить столкновение между двумя игровыми объектами, хотя оба могут иметь в качестве компонента коллайдер. Один из объектов должен иметь компонент Rigidbody.

**Создание скрипта для управления ранцем**

Чтобы не позволить мыши упасть, вы должны добавить скрипт, который позволит применять силу к объекту mouse, чтобы переместить его вверх и удержать от падения. Чтобы добавить скрипт к объекту mouse выполните следующие действия:

1. В Браузере проекта создайте новую папку под названием Scripts. Убедитесь, что эта папка выбрана, так как Unity добавляет новый скрипт в выбранную папку.
2. Выберите Assets\Create\C# script в верхнем меню и назовите скрипт MouseController.
3. Перетащите скрипт на mouse в Иерархии, чтобы добавить его к объекту игры mouse (см. рисунок. 4.2).

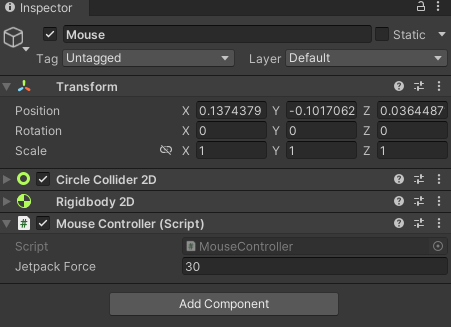


Рисунок 4.2 – Скрипт, добавленный как компонент к объекту Mouse.

Приступим к написанию скрипта. Добавьте переменную jetpackForce внутри объявления класса. Это будет сила, возникающая при включении реактивного ранца и не дающая мышке упасть.

Движение мышки на ракетном ранце реализуем с помощью кода ниже.

Листинг:

**public** **class** MouseController : MonoBehaviour

{

[SerializeField] **private** **float** jetpackForce = 30.0f;

**private** **bool** jetpackActive = **false**;

**private** **void** FixedUpdate()

{

**if** (jetpackActive)

{

GetComponent<Rigidbody2D>().AddForce(**new** Vector2(0, jetpackForce));

}

}

**void** Update()

{

jetpackActive = Input.GetButton("Fire1");

}

}

Обратите внимание, что публичная переменная хоть и позволяет нам взаимодействовать с ним непосредственно внутри Инспектора Unity, однако нарушает инкапсуляцию. Чтобы пользоваться возможностями Инспектора мы добавляем атрибут [SerializeField] к нашей приватной переменной.

Ракетный ранец работает, но есть несколько проблем. Во-первых, в зависимости от вашей точки зрения, то ли ракетный ранец слишком мощный, то ли гравитация слишком слаба. В результате вы легко можете отправить мышь за пределы верхней части экрана.

Вместо изменения силы, создаваемой ракетным ранцем, вы можете изменить настройки гравитации всего проекта. Для глобального изменения силы тяжести выберите Edit\Project Settings\Physics 2D. Это откроет настройки Physics 2D проекта в Инспекторе (см. рисунок 4.3). Найдите поле Gravity (Гравитация) и установите значение Y равным -15. Мышь стала более управляемой, но тем не менее по-прежнему может вылететь или провалится за пределы экрана. Чтобы это не произошло, добавим ограничители пространства: пол и потолок.

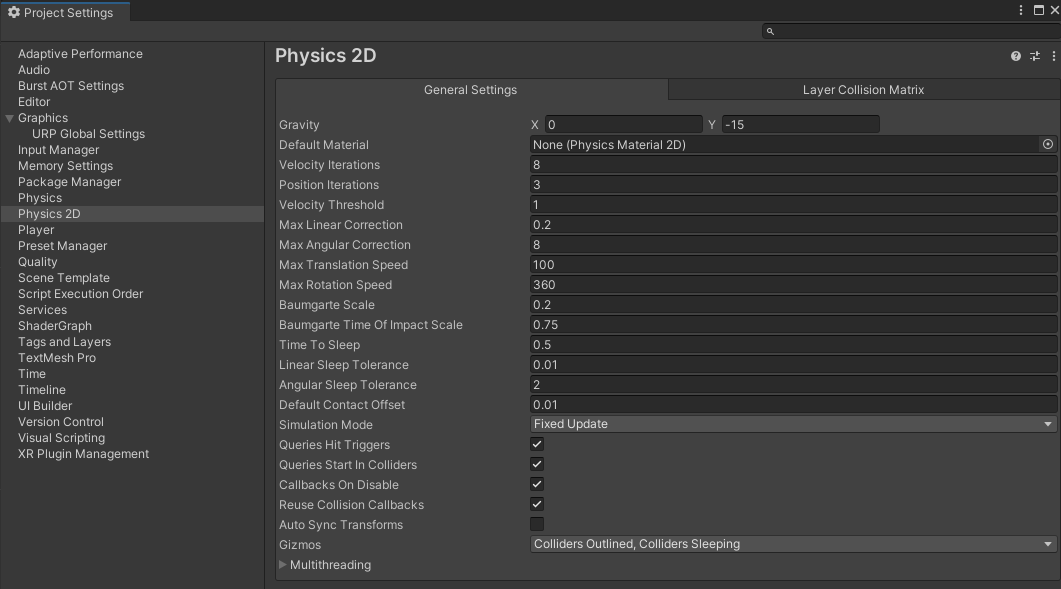


Рисунок 4.3 - Рисунок 3 – Параметры 2D физики проекта.

**Создание пола и потолка**

Пол и потолок могут быть представлен пустыми игровыми объектами, поскольку они никогда не двигаются и их расположение очевидно для пользователя. Выберите GameObject\Create Empty чтобы создать пустой объект. Вы не увидите его на экране прямо сейчас. Выберите GameObject в Иерархии и сделайте следующие изменения в Инспекторе:

1. Переименуйте его в Floor.
2. Установите Position в (0, -3.5, 0).
3. Установите Scale в (14.4, 1, 1).
4. Нажмите Add Component (Добавить Компонент) и добавьте компонент Box Collider 2D выбрав Physics 2D\Box Collider 2D.

Добавляем потолок самостоятельно. Задаем позицию потолка Position координатами (0, 3.7, 0), переименовываем его в Ceiling. Теперь, если вы запустите игру, вы увидите, что мышь никогда не улетает за пределы игрового экрана.

**4.3. Индивидуальное задание**

Создайте игровой проект Rocket Mouse на основе теоретических сведений. Используйте системы частиц для создания эффекта горения у ранца, добавьте препятствия для мыши, а также “монетки”, которые она будет подбирать. Игра должна заканчиваться, когда мышь сталкивается с препятствием. На экране должен отображаться текущий счет и количество собранных монет.

### Лабораторная работа №5. Создание 3D FPS шутера.

**Цель работы**: изучить теоретические сведения. Создать игру на Unity - 3D FPS шутер с движением, прыжками, врагами, меню, экраном смерти и двумя типами стрельбы.

**5.1. Общие теоретические сведения**

Создание сцены из шутера от первого лица — мы будем обозначать этот тип игр аббревиатурой FPS (First-Person Shooter). Вы построите комнату, по которой можно перемещаться, при этом игроки будут наблюдать окружающий мир с точки зрения игрового персонажа. Управлять этим персонажем игрок сможет посредством мыши и клавиатуры. Разработаем приблизительный сценарий:

1. Разработка комнаты: создание пола, внешних и внутренних стен.

2. Размещение источников света и камеры.

3. Создание объекта-игрока (в том числе и присоединение камеры к его верхней части).

4. Написание сценариев перемещения: повороты при помощи мыши и перемещения при помощи клавиатуры.

**5.2. Пример выполнения задания**

В расположенном в верхней части экрана меню GameObject наведите указатель мыши на строчку 3D Object, чтобы открыть дополнительное меню. Выберите в нем вариант Cube, так как для нашей сцены требуется куб (позднее вы поработаете и с другими фигурами, такими как Sphere и Capsule). Отредактируйте положение и масштаб появившегося в сцене куба, а также его имя таким образом, чтобы получить пол; значения, которые следует присвоить параметрам этого объекта на панели Inspector, показаны на рисунке 5.1 (для превращения куба в пол его нужно растянуть).

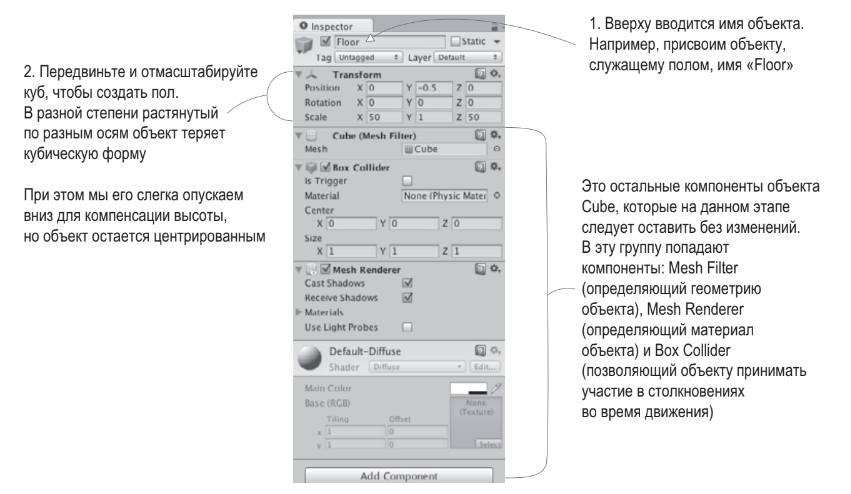


Рисунок 5.1 – Свиток параметров стены

Повторите описанную последовательность для создания внешних стен комнаты. Можно каждый раз задействовать новый куб, а можно копировать и вставлять существующий объект, указывая стандартные сокращения. Двигайте, поворачивайте и масштабируйте стены. Экспериментируйте с различными значениями (например, 1, 4, 50 для полей Scale).

Не забудьте про средства навигации, позволяющие рассматривать сцену под разными углами и менять ее масштаб, например имитируя взгляд с высоты птичьего полета. При этом нажатие клавиши F вернет вас к просмотру выделенного в данный момент объекта.

Точные значения преобразований для стен будут зависеть от того, каким образом вы повернете и отмасштабируете исходные объекты Cube, подогнав их размеры и положение, а также от способа их связывания на вкладке Hierarchy. Если вы предпочитаете просто скопировать рабочие значения, скачайте пример проекта и возьмите все данные оттуда.

Связи между объектами устанавливаются простым перетаскиванием объектов друг на друга на вкладке Hierarchy. Объект, к которому присоединены другие объекты, называется предком (parent); объекты, присоединенные к другим объектам, называются потомками (children). Перемещение (поворот или масштабирование) родительского объекта сопровождается аналогичным преобразованием всех его потомков.

Для систематизации объектов сцены подобным образом применяются пустые игровые объекты. Связывание видимых объектов с корневым позволяет сворачивать списки объектов на вкладке Hierarchy. Но помните, что перед этой операцией следует расположить пустой корневой объект в точке с координатами 0, 0, 0, чтобы в дальнейшем избежать проблем с позиционированием.

В этом проекте игрока будет представлять обычный примитив. В меню GameObject (напоминаю, что для открытия этого дополнительного меню нужно навести указатель мыши на строку 3D Object) выберите вариант Capsule. Появится цилиндрическая фигура со скругленными концами — это и есть наш игрок. Сместите объект вверх, сделав его координату Y равной 1.1 (половина высоты объекта, плюс еще немного, чтобы избежать перекрывания с полом). Теперь наш игрок может произвольным образом перемещаться вдоль осей X и Z при условии, что он остается внутри комнаты и не касается стен. Присвойте объекту имя Player. На панели Inspector вы увидите, что этому объекту назначен капсульный коллайдер. Это очевидный вариант для объекта Capsule, точно так же, как объект Cube по умолчанию обладает коллайдером Box. Но так как наша капсула будет представлять игрока, ее компоненты должны слегка отличаться от компонентов большинства объектов. Капсульный коллайдер мы удалим. Для этого нужно щелкнуть на значке с изображением шестерни справа от имени компонента. Откроется меню, в котором в числе прочих вы найдете и команду Remove Component. Коллайдер выглядит как окружающая объект зеленая сетка, поэтому после удаления компонента вы обнаружите, что она исчезла.

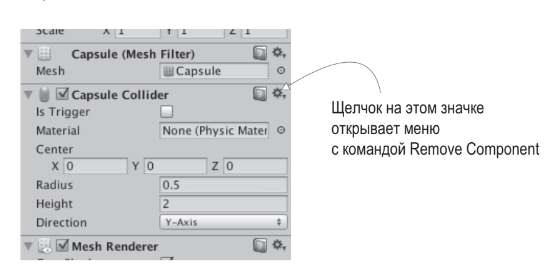


Рисунок 5.2.2 – Настройки объекта – тела игрока

Чтобы заставить игрока перемещаться по сцене, нам потребуются сценарии движения, которые будут присоединены к игроку. Напоминаю, что компонентами называются модульные фрагменты функциональности, добавляемые к объектам, поэтому сценарии тоже можно считать своего рода компонентами. Именно они будут реагировать на клавиатурный ввод и манипуляции мышью, но для начала мы заставим игрока поворачиваться на месте. Это продемонстрирует вам, как применять преобразования в коде. Надеюсь, вы помните, что преобразования бывают трех видов — перемещение, поворот и масштабирование; вращение объекта на месте соответствует преобразованию поворота. Но вы должны знать об этой задаче еще кое-что, кроме того, что она «сводится к изменению ориентации объекта».

Напишем пример движения персонажа с прыжками и бегом на shift.

Листинг:

**using** System.Collections;

**using** System.Collections.Generic;

**using** UnityEngine;

**public** **class** Muvement2 : MonoBehaviour

{

[SerializeField] **private** **float** \_speedWalk;

[SerializeField] **private** **float** \_gravity;

[SerializeField] **private** **float** \_jumpPower;

[SerializeField] **private** **float** \_speedRun;

**private** CharacterController \_characterController;

**private** Vector3 \_walkDirection;

**private** Vector3 \_velocity;

**private** **float** \_speed;

**private** **void** Start()

{

\_speed = \_speedWalk;

\_characterController = GetComponent<CharacterController>();

}

**private** **void** Update()

{

Jump(Input.GetKey(KeyCode.Space) && \_characterController.isGrounded);

Run(Input.GetKey(KeyCode.LeftShift));

Sit(Input.GetKey(KeyCode.LeftControl));

**float** x = Input.GetAxis("Horizontal");

**float** z = Input.GetAxis("Vertical");

\_walkDirection = transform.right \* x + transform.forward \* z;

}

**private** **void** FixedUpdate()

{

Walk(\_walkDirection);

DoGravity(\_characterController.isGrounded);

}

**private** **void** Walk(Vector3 direction)

{

\_characterController.Move(direction \* \_speedWalk \* Time.fixedDeltaTime);

}

**private** **void** DoGravity(**bool** isGrounded)

{

**if** (isGrounded && \_velocity.y < 0)

\_velocity.y = -1f;

\_velocity.y -= \_gravity \* Time.fixedDeltaTime;

\_characterController.Move(\_velocity \* Time.fixedDeltaTime);

}

**private** **void** Jump(**bool** canJump)

{

**if** (canJump)

\_velocity.y = \_jumpPower;

}

**private** **void** Run(**bool** canRun)

{

\_speedWalk = canRun ? \_speedRun : \_speed;

}

**private** **void** Sit(**bool** canSit)

{

\_characterController.height = canSit ? 1f : 2f;

}

}

Теперь нужно заставить преобразование поворота реагировать на ввод с помощью мыши. В данном случае подразумевается вращение объекта, к которому присоединен сценарий, то есть игрока. Задача будет решаться в несколько этапов путем постепенного добавления персонажу двигательных возможностей. Сначала мы заставим игрока поворачиваться только из стороны в сторону, а затем — только вверх и вниз. В конечном счете игрок научится смотреть во всех направлениях (поворачиваясь одновременно в горизонтальной и вертикальной плоскостях). Такое поведение называют слежением за мышью (mouse-look).

Листинг:

**using** System.Collections;

**using** System.Collections.Generic;

**using** UnityEngine;

**public** **class** Mouselook2 : MonoBehaviour

{

[SerializeField] **private** **float** \_sensitivity;

[SerializeField] **private** Transform \_character;

**private** **float** \_xRotation;

**void** Start()

{

Cursor.lockState = CursorLockMode.Locked;

}

**void** Update()

{

Tracking();

}

**private** **void** Tracking()

{

**float** mouseX = Input.GetAxis("Mouse X") \* \_sensitivity \* Time.deltaTime;

**float** mouseY = Input.GetAxis("Mouse Y") \* \_sensitivity \* Time.deltaTime;

\_xRotation -= mouseY;

\_xRotation = Mathf.Clamp(\_xRotation, -90, 90);

transform.localRotation = Quaternion.Euler(\_xRotation, 0, 0);

\_character.Rotate(Vector3.up \* mouseX);

}

}

Первой дополнительной функциональной возможностью, которую мы добавим в наш демонстрационный ролик, станет стрельба. Умение оглядываться по сторонам и перемещаться в шутере от первого лица является, без сомнения, решающим, но игра не начнется, пока игроки не смогут влиять на окружающее пространство и показывать свое мастерство. Стрельба в трехмерных играх реализуется несколькими способами, наиболее важным из которых является бросание лучей.

Вы формируете луч и затем определяете, с чем он пересекается. Подумайте, что происходит, когда вы стреляете из пистолета: пуля вылетает из точки, в которой находится пистолет, и летит по прямой вперед, пока не столкнется с каким-нибудь препятствием. Луч можно сравнить с путем пули, а бросание луча аналогично выстрелу из пистолета и наблюдению за тем, куда попадет пуля.

Эта операция в Unity осуществляется методом ScreenPointToRay(). Метод создает луч, начинающийся с камеры, и проецирует его по линии, проходящей через указанные экранные координаты. Обычно при выборе с помощью мыши используются координаты указателя, но в шутерах от первого лица эту роль играет центр экрана. Появившийся луч передается методу Physics. Raycast(), который и выполняет его «бросание». Таким образом у нас будет работать не столько срельба, сколько ориентирование ИИ на локации.

Листинг:

**using** System.Collections;

**using** System.Collections.Generic;

**using** UnityEngine;

**public** **class** EnemyAI : MonoBehaviour

{

**public** **float** speed = 5;

**public** **float** obstacleRange = 5;

**public** **bool** \_alive = **true**;

[SerializeField]

**private** GameObject[] \_fireballsPrefab;

**private** GameObject \_fireball;

**void** Start()

{

\_alive = **true**;

}

**void** Update()

{

**if** (\_alive)

{

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

Ray ray = **new** Ray(transform.position,transform.forward);

RaycastHit hit;

**if**(Physics.Raycast(ray,**out** hit))

{

GameObject hitObject = hit.transform.gameObject;

//если игрок

**if** (hitObject.GetComponent<CharacterController>())

{

**if**(\_fireball==**null**)

{

//стреляем в направлении игрока

**int** randFireball = Random.Range(1, \_fireballsPrefab.Length);

\_fireball = Instantiate(\_fireballsPrefab[randFireball]) **as** GameObject;

\_fireball.transform.position = transform.TransformPoint(Vector3.forward \* 1.5f);

\_fireball.transform.rotation = transform.rotation;

}

}

//если не игрок а стенка

**else** **if** (hit.distance<obstacleRange)

{

**float** angleRotation = Random.Range(-100, 100);

transform.Rotate(0, angleRotation, 0);

}

}

}

}

**public** **void** SetAlive(**bool** alive)

{

\_alive = alive;

}

}

Саму же стрельбу лучше реализовать путём создания экземпляров. Так игрок сможет уклоняться от снарядов и мы в будующем сможем добавить им физичности (скорость полёта, силу гравитации).

Напишем код полуавтоматичной стрельбы путём создания экземпляров для игрока.

Листинг:

**using** System.Collections;

**using** System.Collections.Generic;

**using** System.Runtime.CompilerServices;

**using** UnityEngine;

**public** **class** Shooting : MonoBehaviour

{

**private** Camera \_camera;

[SerializeField]

**private** GameObject[] \_fireballsPrefab;

**private** GameObject \_fireball;

**private** **float** \_time;

**private** **void** Start()

{

\_camera = GetComponent<Camera>();

Cursor.lockState = CursorLockMode.Locked;

Cursor.visible = **false**;

}

**private** **void** Update()

{

**int** rateOfspeed = 0;

**if** (\_time<= rateOfspeed)

{

\_time += Time.deltaTime;

}

**if** (Input.GetMouseButtonDown(0))

{

\_time = 0;

\_fireball = Instantiate(\_fireballsPrefab[1]) **as** GameObject;

rateOfspeed = \_fireball.GetComponent<fireball>().rateOfSpeed;

Debug.Log("Player heath: " + rateOfspeed);

\_fireball.transform.position = transform.TransformPoint(Vector3.forward \* 2f);

\_fireball.transform.rotation = transform.rotation;

}

}

}

**5.3. Индивидуальное задание**

На основе теоретических сведений создайте свой индивидуальный 3D FPS шутер с движением, прыжками, врагами, меню, экраном смерти и двумя типами стрельбы в Unity.