Міжнародний гуманітарний університет
Кафедра Комп'ютерної інженерії та інноваційних технологій

Лебедєва О.Ю.

ПРОГРАМУВАННЯ ТА ЗАХИСТ PLAY-ТЕХНОЛОГІЙ

методичні рекомендації для практичних занять
для здобувачів другого (магістерського) рівня,
які навчаються за спеціальністю 125 – Кібербезпека та захист інформації

Одеса 2024

Затверджено Вченою Радою Міжнародного гуманітарного університету (протокол № 13 від 16.08.2024 р.).

Лебедєва О.Ю.

Програмування та захист play-технологій: методичні рекомендації для практичних занять для здобувачів другого (магістерського) рівня, які навчаються за спеціальністю: 125 – Кібербезпека та захист інформації [Електронне видання] / О.Ю. Лебедєва, кафедра комп'ютерної інженерії та інноваційних технологій Міжнародного гуманітарного університету. Одеса, 2024. – 63 с.

Методичні рекомендації для практичних занять для курсу «Програмування та захист play-технологій» призначені для здобувачів, що навчаються за другим (магістерським) рівнем за спеціальністю: 125 – Кібербезпека та захист інформації. Дисципліна «Програмування та захист play-технологій» надає здобувачам знання в галузі комп'ютерних ігор та захисту інформації. На основі загальних понять в програмуванні та об'єктно-орієнтованого підходу дисципліна розглядає процес створення 2D-ігор в середовищі Unity, написання скриптів для реалізації функціональності гри та захисту даних.

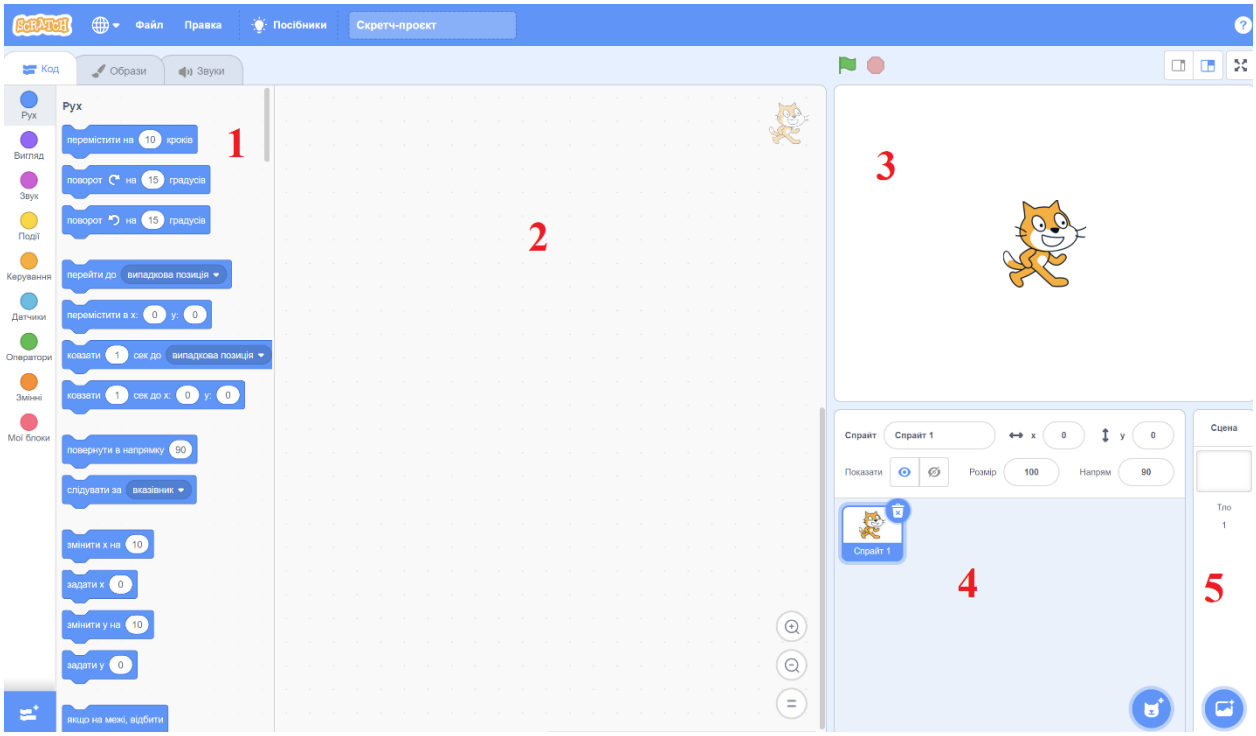
ЗМІСТ

Практична робота 1. Створення ігор за допомогою середовища Scratch

Мета роботи: познайомитися із програмою Scratch. Навчитися створювати гру за допомогою програми Scratch.

1. Теоретичний матеріал

Під час запуску програми Scratch відкривається вікно виду:



Вікно програми Scratch 3 містить декілька робочих зон та меню.

Верхній рядок дозволяє перемикати мову написів у всьому додатку, а також зберігати / відкривати файли проектів і скасовувати останні скоєні дії.

У верхній частині вікна, відразу після синьої смуги з кнопками, знаходяться вкладки з інструментами.

Закладка «Код» – основна робота виконується тут. Додавання персонажів та об'єктів на екран, програмування анімації та взаємодій.

Закладка «Образи» – сюди перемикаємося для налаштування вибраного спрайту. Зміна кольору, розміру, форми чи зовнішнього вигляду цілком. У Scratch персонажі та предмети, що додаються до проекту, називаються спрайтами.

Закладка «Звуки» – якщо об'єкт може видавати звуки, то на цій вкладці їх можна прослухати та змінити.

Основні зони вкладки «Код»:

1. Палітра блоків
2. Область коду
3. Сцена
4. Панель спрайтів
5. Вибір фону

Палітра блоків – тут можна знайти всі можливі блоки, з яких будуються проекти та завдяки яким відбуваються різні дії у грі чи анімації. Для зручності, блоки поділені на групи, що відрізняються за кольором та призначенням.

Область коду – тут із блоків збираються послідовності – скрипти, – а також налаштовуються дії, які виконуються спрайтами. Події, що викликаються, можуть залежати від часу від запуску гри, взаємодії об'єктів на екрані та натискання якихось кнопок гравцем.

Сцена – на цьому екрані відбувається анімація чи ігровий процес (залежить від мети проекту). Спрайти можна встановити в потрібні місця вручну або за допомогою спеціалізованих блоків коду.

Панель спрайтів – всі об'єкти, які розміщуються на сцені, з'являються на цій панелі. Тут проводиться їх додавання та налаштування. Можна керувати розміром та положенням спрайту в просторі, а також задати йому ім'я та режим відображення (приховати з екрана, повернути навколо осі).

Вибір фону – невелика панель поряд зі сценою дозволяє вибрати фон для анімації із вбудованої бібліотеки або завантажити свій. Можна вибрати кілька зображень та керувати ними зі свого алгоритму.

Базова послідовність дій для створення своєї гри чи мультфільму:

– Запускаємо Scratch та додаємо перший спрайт на сцену. Можна назвати проект, але це не обов'язково.
– Налаштовуємо зовнішній вигляд об'єкта чи персонажа (дивлячись з кого почали). Малюємо костюм або вибираємо із готових.
– Збираємо скрипт із блоків. Зліва в палітрі шукаємо потрібну групу і по черзі тягнемо потрібні умови, оператори, дії в область коду.
– Додаємо нові спрайти та скрипти, перемикаємося між ними та доопрацьовуємо, доки не реалізуємо свій задум.
– Періодично запускаємо проект та тестуємо його роботу. Вносимо редагування, знову пробуємо грати або дивитися свій мультик.
– Коли все готово, зберігаємо файл з проектом (по можливості краще зберігати і проміжні варіанти).

Створимо просту гру за допомогою Scratch. При створенні нового проекту створюється спрайт за замовчуванням у вигляді кота:

Рисунок 1 – Спрайт за замовчуванням

У панелі спрайтів можна зробити налаштування спрайтів у вигляді його розташування на сцені, його імені, його розміру та кута повороту картинки тощо.

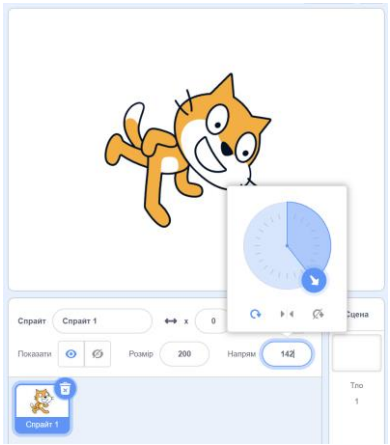


Рисунок 2 – Настройка спрайту на панелі спрайтів

На закладці Образи спрайт можна розфарбувати чи додати йому деталей:

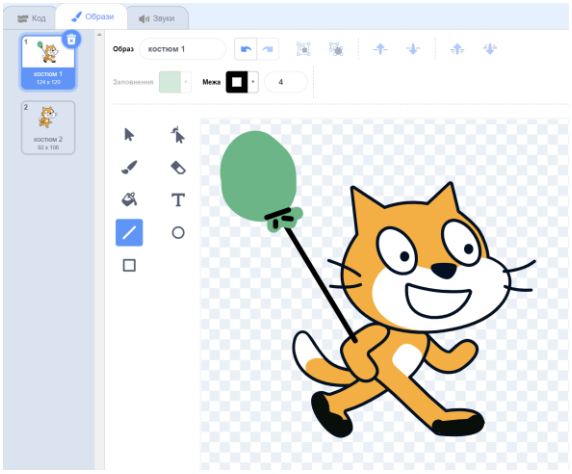
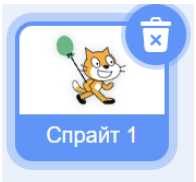


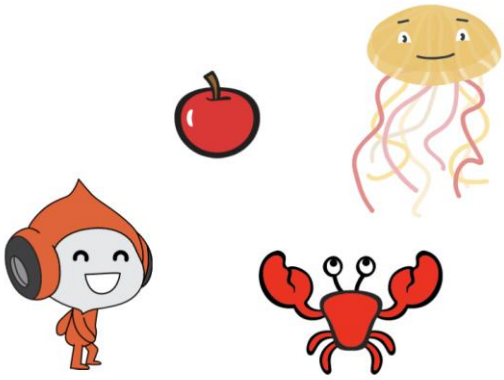
Рисунок 3 – Настройка спрайту на закладці Образи

Видалимо та додаймо нові спрайти в проект. Для видалення знадобиться кнопка кошика на спрайте на панелі спрайтів:

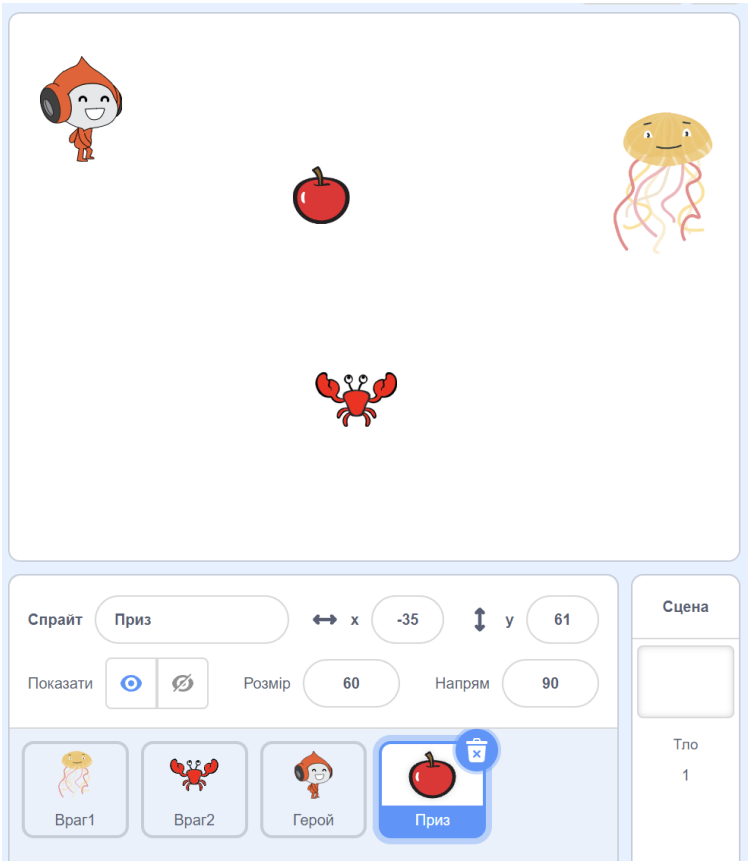


Для додавання можна скористатися кнопкою на панелі спрайтів:

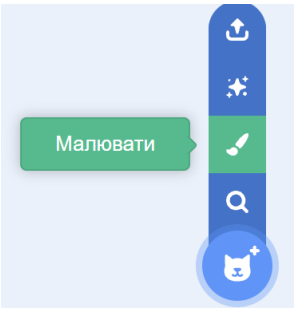
Наприклад, додаймо такі спрайти:



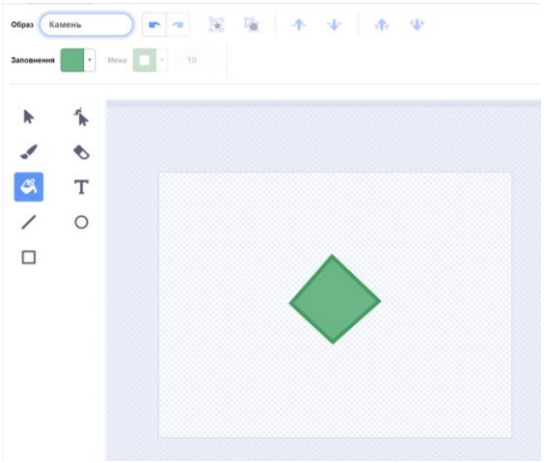
Змінимо розміри доданих спрайтів і дамо кожному своє ім'я:



Додаймо новий спрайт, лише тепер намалюємо його самостійно. Для цього підводимо покажчик миші до кнопки додавання спрайтів і в меню виберемо пункт «Малювати»:



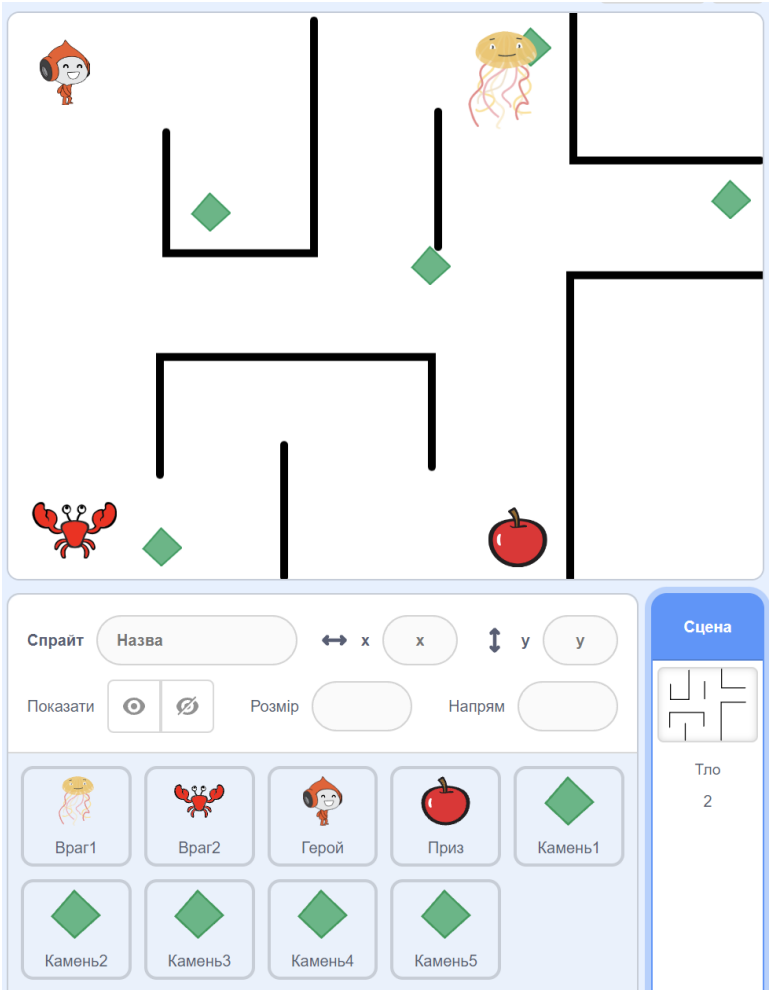
Створимо якусь фігуру, зафарбуємо її та дамо ім'я:



Основних учасників гри підготували, займмося фоном. У нашій грі фоном буде якийсь лабіринт, яким треба пройти нашому Герою.

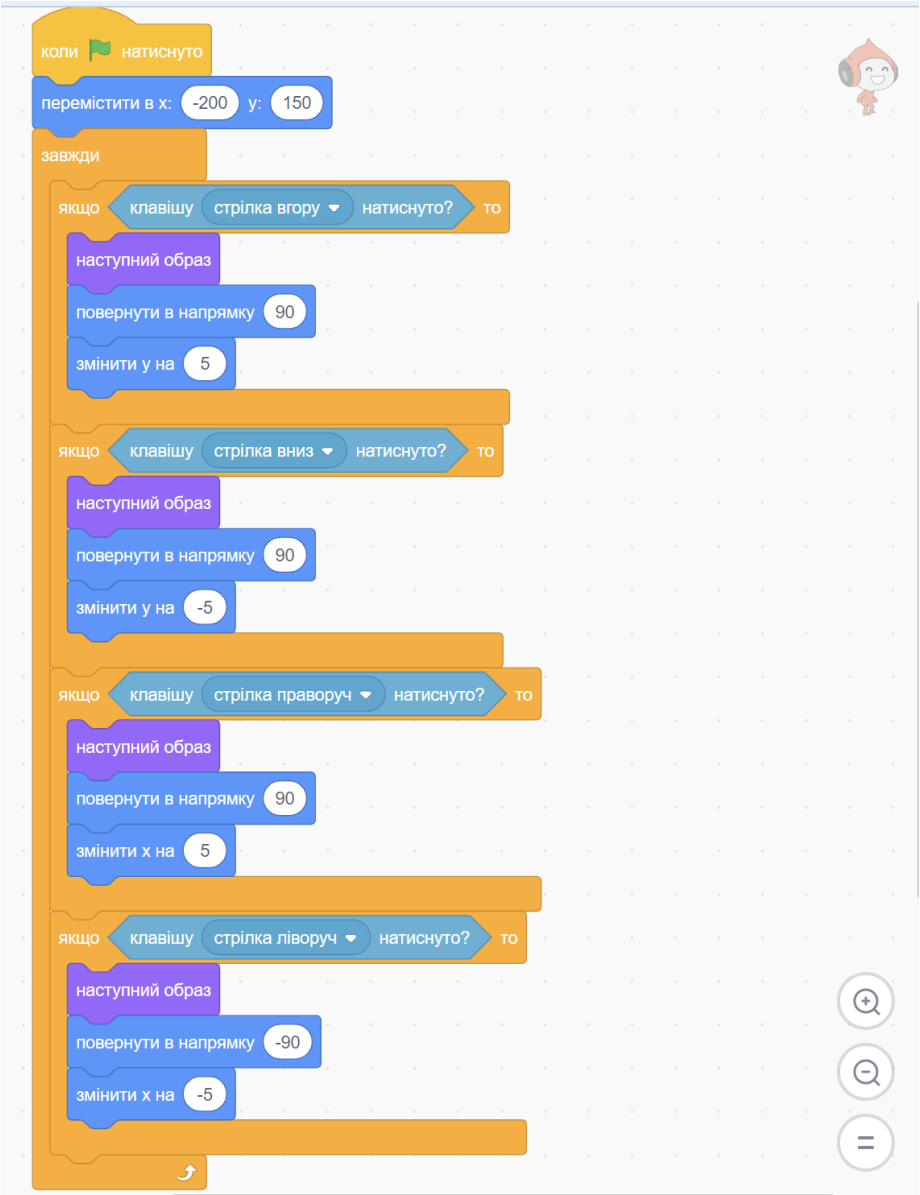
Для створення фону підводимо покажчик миші до кнопки Вибрати фон на панелі Вибору фону та вибираємо Намалювати. Намалюємо якийсь лабіринт і новому фону дамо назву «Рівень1».

Розподілимо по лабіринту обрані та створені спрайти:

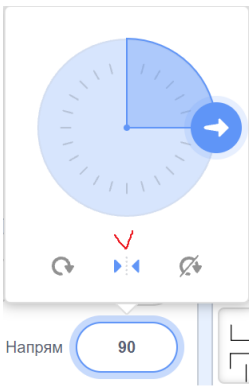


Налаштуємо рух головного Героя гри. Нехай він ходитиме сценою при натисканні кнопок управління курсорів: ←, →, ↑, ↓.

Вибираємо спрайт головного героя та в панелі «Область коду» запишемо наступний код:



Додатково налаштуємо, що головний герой може повертатися не навколо своєї осі, а лише праворуч і ліворуч. Для цього вибираємо спрайт Герой і в полі Напрям натискаємо на стрілочки «Зліва-направо».



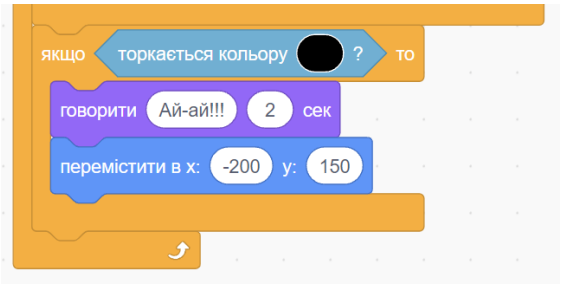
Якщо запустити гру ( ), то головний герой буде пересуватися сценою при натисканні кнопок управління курсору.

Налаштуємо рух ворогів нашого героя. Вороги ходитимуть деяким заданим маршрутом. Виділяємо спрайт ворога, наприклад, як краба і додаємо йому наступний код:

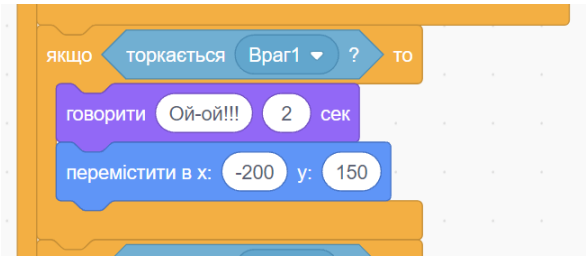
Аналогічно можна задати маршрут для другого ворога у вигляді медузи.

Тепер додаємо реакцію головного героя на зіткнення зі стінами лабіринту, з ворогами, з камінням та призів у вигляді яблука.

Так як стіни лабіринту ми задали чорного кольору, то відповідно будемо відловлювати зіткнення головного героя з чорним кольором і повертати героя на початкову позицію. Додаємо до наявного коду Героя наступний код:



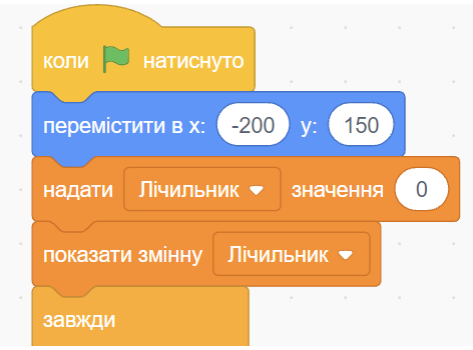
Аналогічно можна задати реакцію на зіткнення з ворогами.



Додаймо реакцію на зіткнення з камінням. Коли герой зіткнеться з камінням, вони будуть зникати з поля. Коли все каміння зникне відобразимо у випадковому місці яблуко.

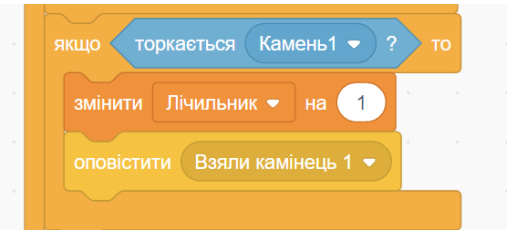
Створимо змінну, в якій зберігатимемо кількість підібраних каменів. Для цього на панелі «Палітра блоків» натискаємо на розділ Змінні та кнопку «Створити змінну». Даємо ім'я цієї змінної «Лічильник».

Додаємо код до нашого герою:

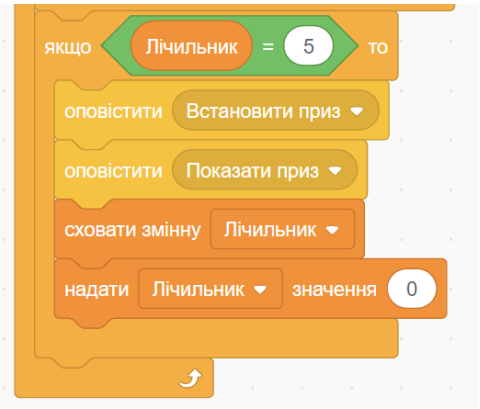


…..

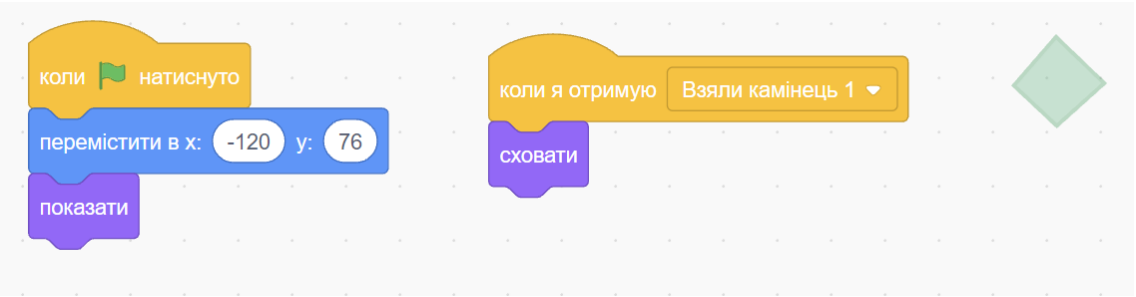
наступний кусок коду для кожного каменю, створюючи для нього його особисте повідомлення



Потім додаємо перевірку чи зібрали всі камінці

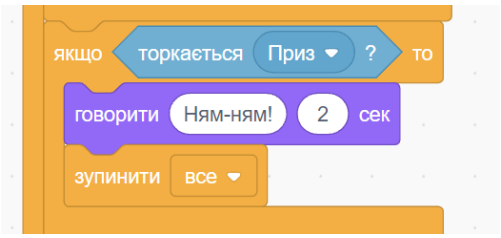


Переходимо до кожного каменю та додаємо наступний код (змінюємо для кожного камінця його місцеположення та назву повідомлення):



Переходимо до спрайту з яблуком та додаємо наступний код:

Тепер залишається налаштувати поведінку героя, коли він зіткнеться з яблуком. Переходимо до спрайту головного герою та додаємо код перед реакцією на чорний колір:

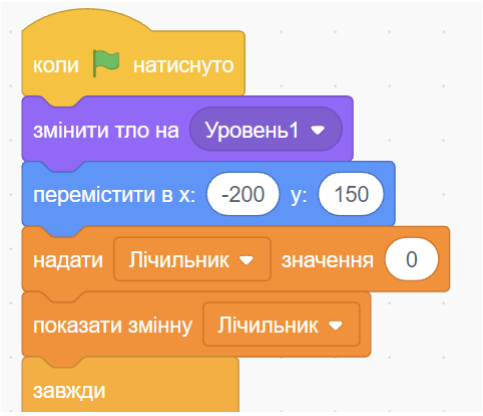


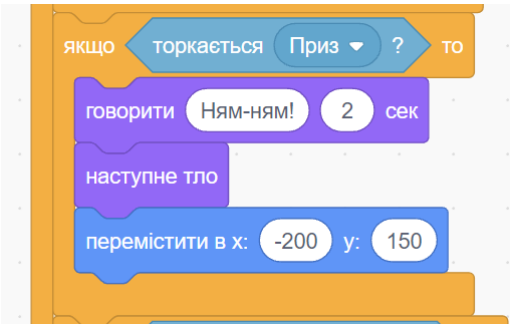
Запустимо гру.

Зробимо ускладнення гри. Додаймо ще один рівень у грі. Створимо новий фон, на якому намалюємо інший лабірінт.

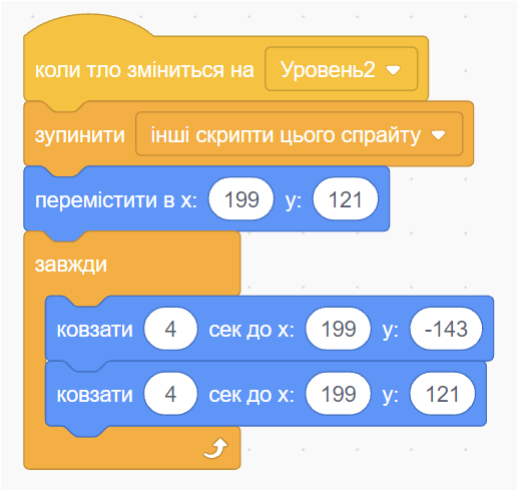
Змінемо код коли головний герой торкається ябкука. Раніше ми зупиняли гру. Тепердодамо перехід на наступний рівень. Для цього з блока коду, який відлавлює торкання яблука заберемо дію «Зупинити все». Додамо нові дії «Змінити тло» та «Наступне тло», а також необхідно прописвти для кожного спрайту що він повинен робити при зміні рівню.

Розглянемо доработки для головного герою:

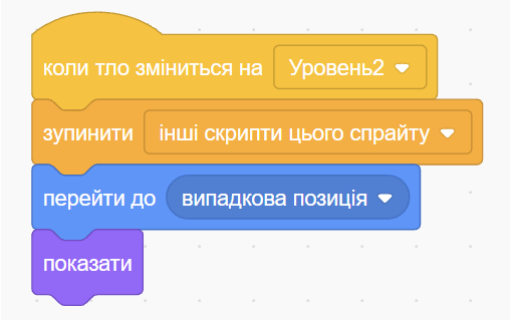




Розглянемо доробки для ворога медузи. Для цього знадобитья подія «Коли тло зміниться на»:



Аналогічним чином зробити доробки для ворога у вигляді крабу.
Для всіх камінців додамо можливість з’являтися у випадковому місці:



До яблука додаємо код, що при переході на новий рівень, яблуко ховається.
Запустимо гру.

2. Завдання до лабораторної роботи №1
Створити власну гру за допомогою програми Scratch.

Практична робота 2. Створення гри Space shooter на Unity

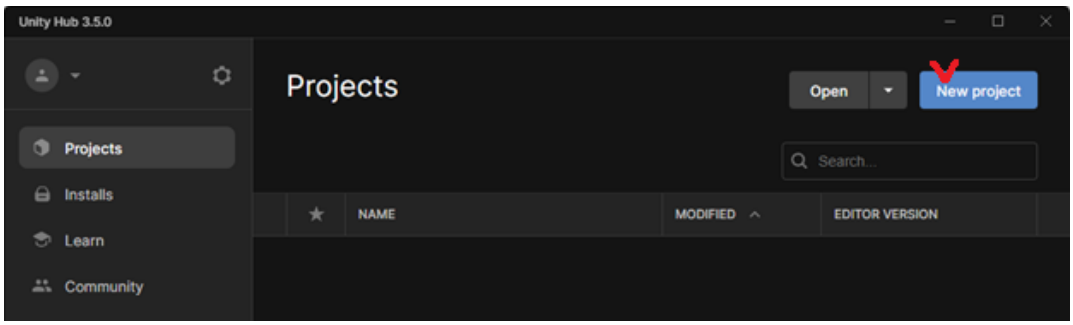
Мета роботи: познайомитися із програмою Unity. Інтерфейс двигуна Unity. Створення нової гри. Робота з Asset Store. Додавання асетів з Asset Store. Створення скриптів. Створення гри з асетів з Asset Store Навчитися створювати 2D-гру за допомогою вже існуючих асетів в програмі Unity.

1. Теоретичний матеріал

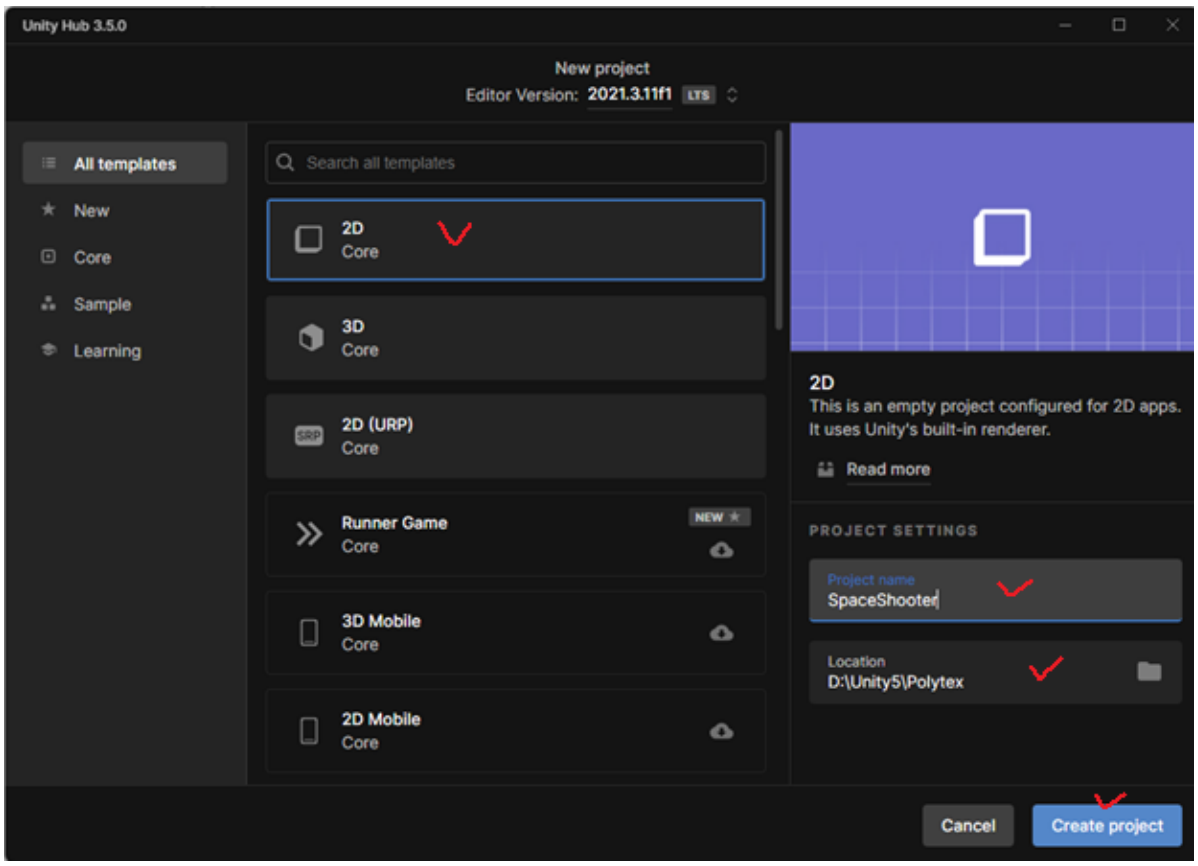
Створимо гру, використовуючи такі кроки:
1. Створення порожнього проекту для гри 2D;
2. Створення нескінченного тла;
3. Додавання ігрового об'єкта;
4. Додавання ворогів

Крок 1. Створення порожнього проекту для гри 2D.

Запускаємо Unity Hub, в лівій панелі вікна переходимо до Projects та натискаємо кнопку New project



Вибираємо тип проекту, вводимо назву та місцеположення для гри, що створюємо:



Ассет – це представлення будь-якого предмета, який можна використовувати у грі чи проекті. Ресурс може бути отриманий з файлу, створеного поза Unity, наприклад 3D-моделі, аудіофайлу, зображення або іншого типу файлу, який підтримує Unity. У Unity можна також створити деякі типи ресурсів, такі як Animator Controller, Audio Mixer або Render Texture.

Assets (активи), створені поза Unity, мають бути перенесені до Unity шляхом збереження файлу безпосередньо у папці «Assets» вашого проекту чи копіювання до цієї папки. Для багатьох поширених форматів ви можете зберегти вихідний файл прямо в папці Assets вашого проекту, і Unity зможе його прочитати. Unity помітить, коли ви збережете нові зміни у файлі, та повторно імпортує їх за потреби.

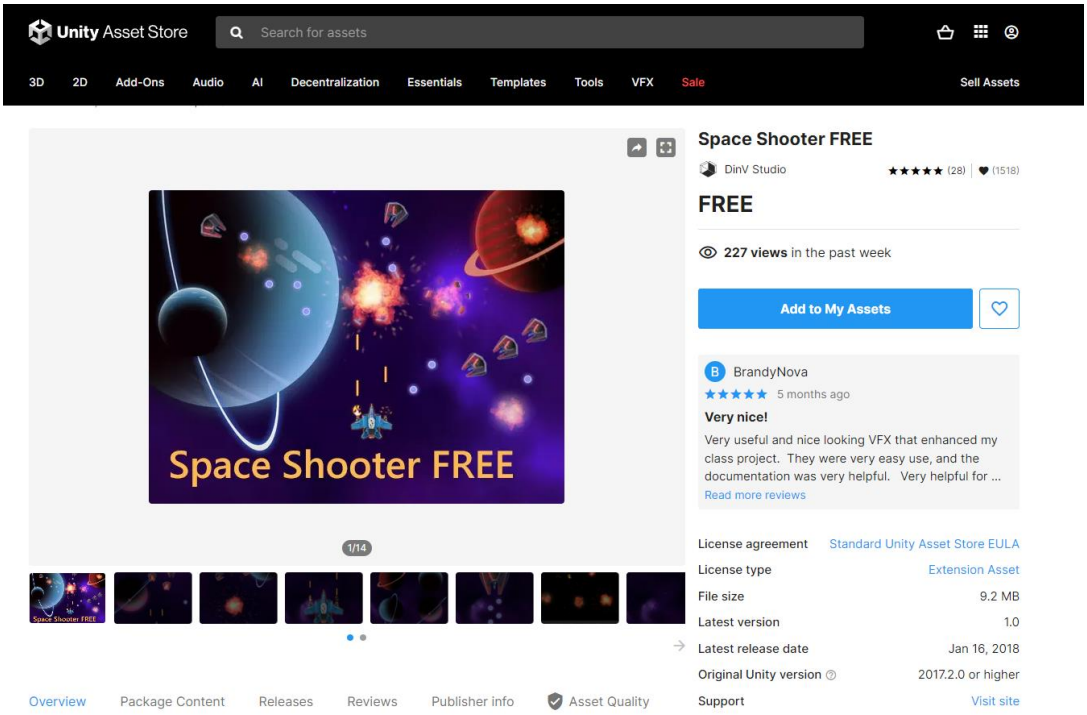
Якщо ви перетягнете файл у вікно проекту Unity зі свого комп'ютера (наприклад, із провідника у Windows), він буде скопійований до папки Assets і з'явиться у вікні проекту.

Найпростіший спосіб безпечно перемістити або перейменувати ваші активи завжди робити це з папки проекту Unity. Таким чином, Unity автоматично перемістить або перейменує відповідний метафайл.

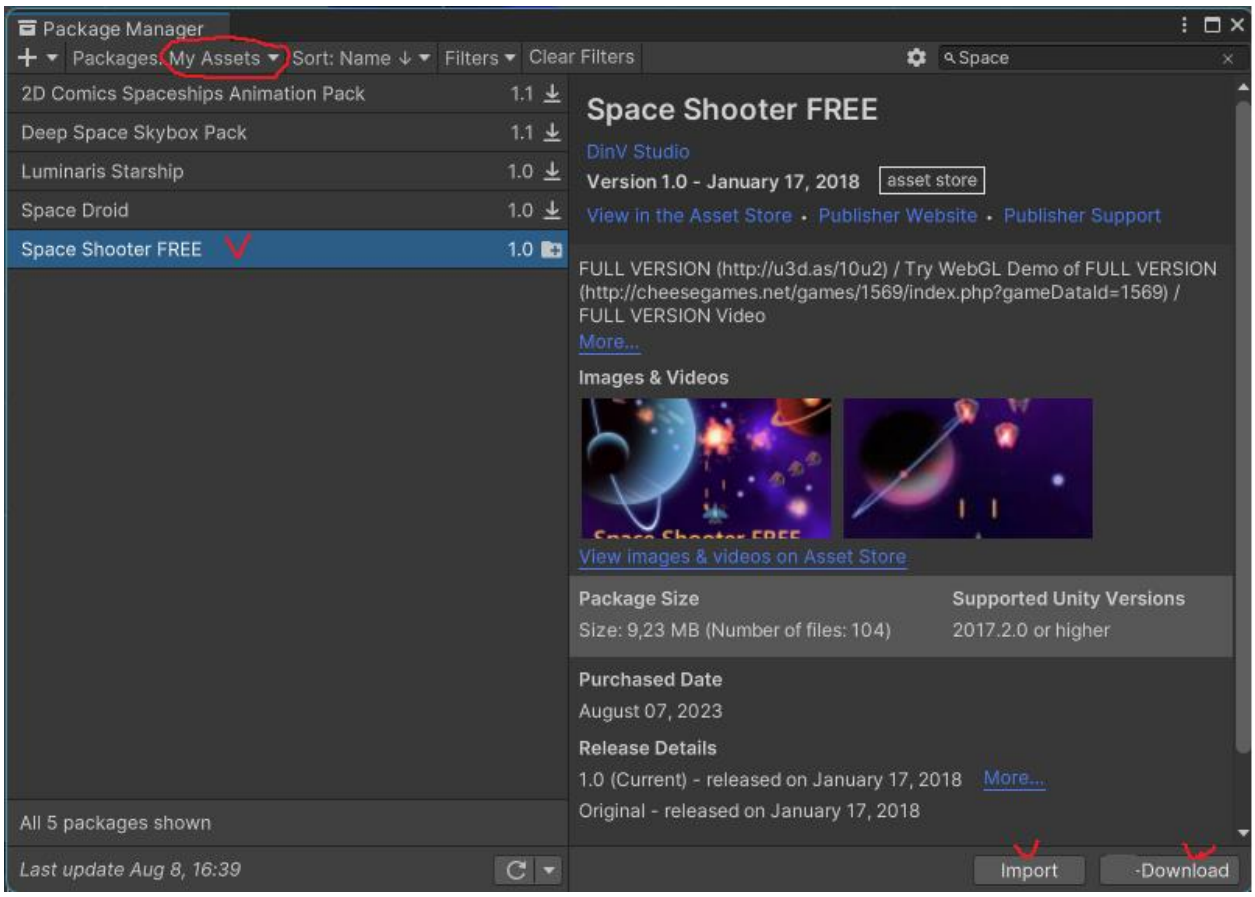
Unity Магазин (Asset Store) – це бібліотека, в якій зібрані безкоштовні та комерційні асети, створені як Unity Technologies, так і членами спільноти. Доступний великий вибір ассетів – текстури, моделі та анімації, приклади проектів, підручники та розширення для редактора. Ассети доступні через простий інтерфейс, вбудований у редактор Unity, через нього можна завантажити та імпортувати безпосередньо у ваш проект.

Відкриємо вікно Asset Store, вибравши в головному меню Window → AssetStore. При першому відвідуванні вам буде запропоновано зареєструвати безкоштовний обліковий запис, який буде використовуватись для доступу до магазину.

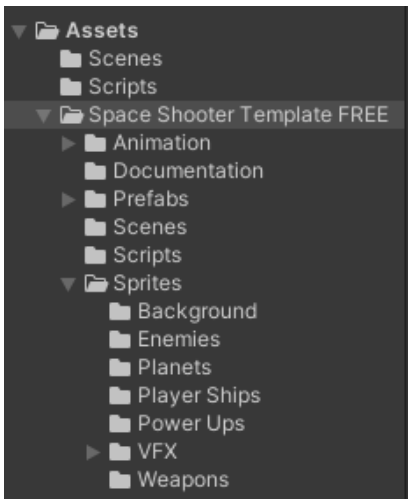
Додамо до проекту активи для створюваної нами гри. Для цього вибираємо Window → Asset Store та натиснути кнопку Search Online. У вікні Unity Asset Store, що відкрилося в браузері, необхідно знайти Space Shooter FREE. Зайти в нього та натиснути на Add to My Assets. Вибраний ассет закачається у добірку ассетів на вашому обліковому записі.



Переглянути та встановити потрібний ассет можна за допомогою Window → Package Manager.

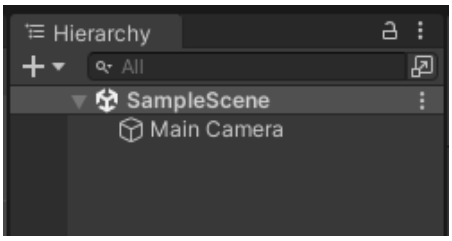


В результаті імпорту вибраного ассета в папці Assets з'явиться нова папка Space Shooter Template FREE із набором ассетів:



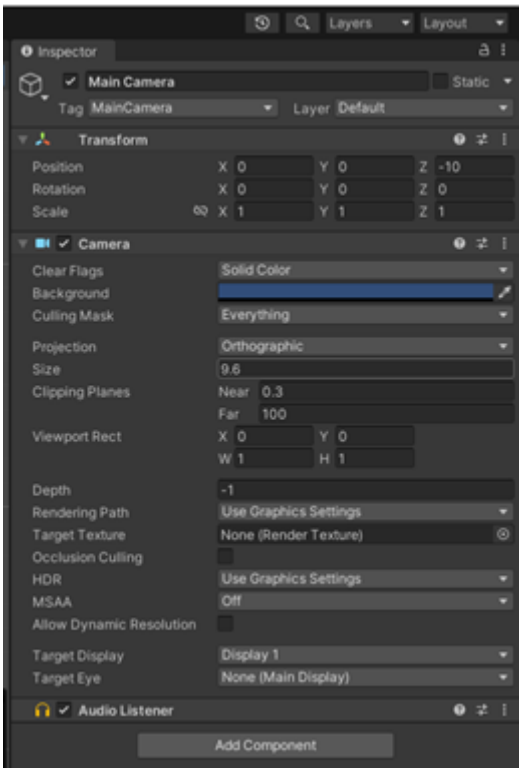
Ієрархія (Hierarchy) містить всі об'єкти (GameObject) у поточній сцені. Unity використовує концепцію під назвою Parenting. Щоб зробити будь-який GameObject дочірнім для іншого, перетягніть бажаний дочірній об'єкт на бажаний батьківський в ієрархії. Дочірній об'єкт успадковуватиме переміщення та повороти свого батька. Ви можете використовувати стрілку батьківського об'єкта, що розкриває, для того, щоб, при необхідності, показати або приховати його дітей.

При створенні проекту 2D гри в ієрархії створився об'єкт сцени і дочірнім до нього створена камера.



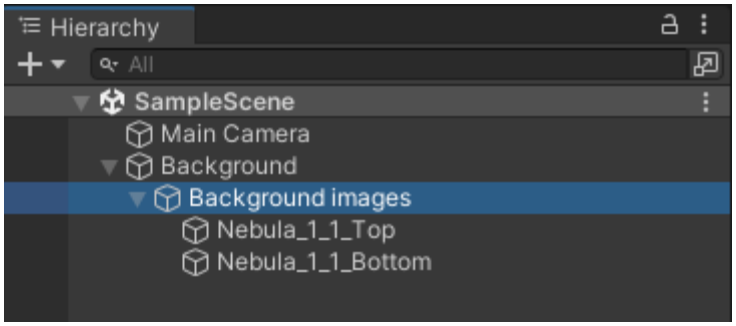
Камери є пристроями, які захоплюють та відображають світ гравцю. Можна мати необмежену кількість камер у сцені. Ви можете налаштувати рендеринг камерами в будь-якому порядку, на будь-якому місці екрана або лише в певних частинах екрану.

Встановимо камеру вертикально. Переходимо в режим Game вибираємо або створюємо 9:16 Aspect. Виділяємо Main Camera та у властивостях у полі Size встановлюємо значення 9,6.

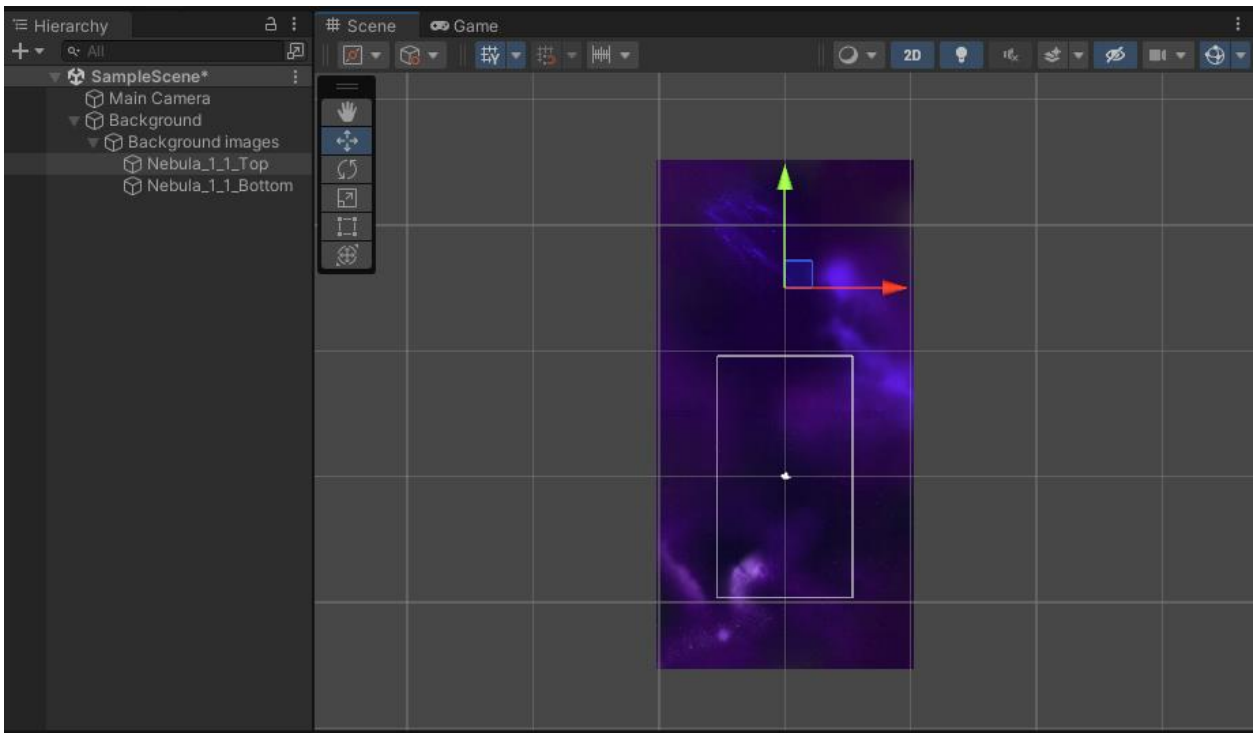


Крок 2. Створення нескінченного фону

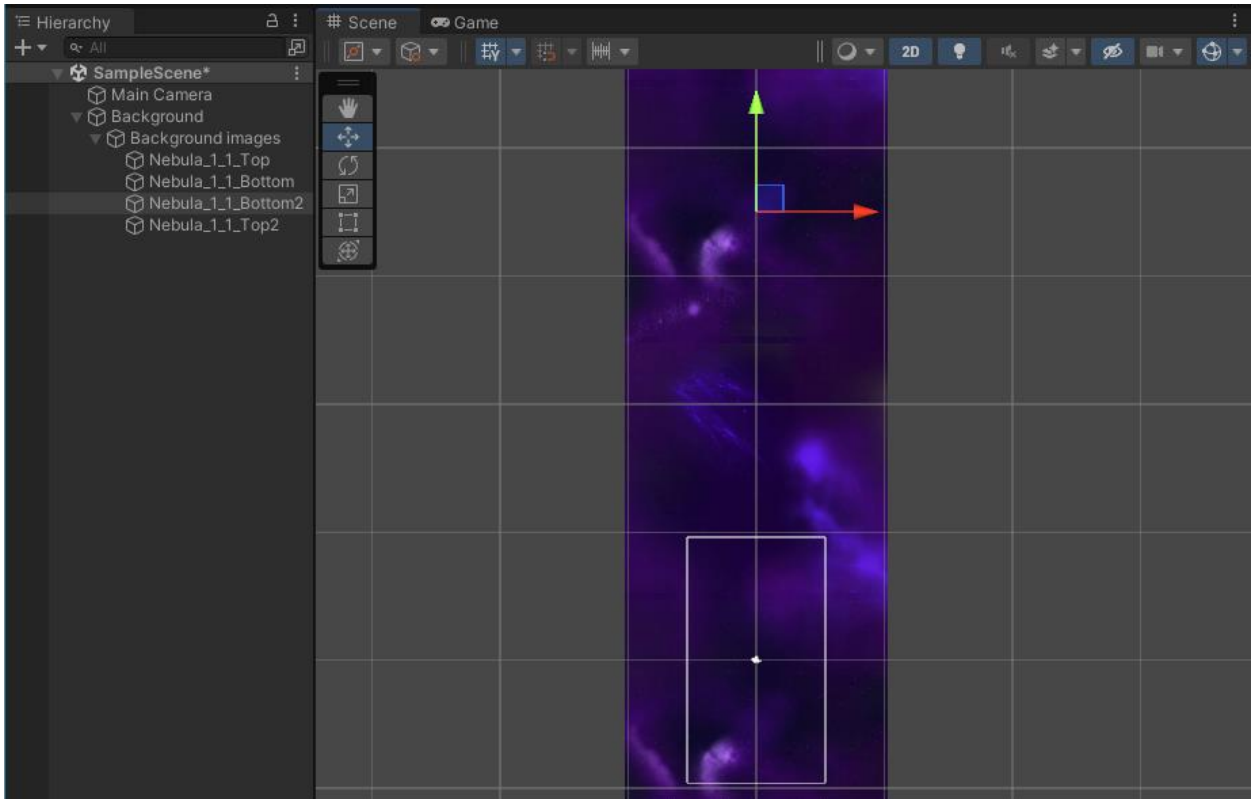
Створимо новий порожній об'єкт (Create Empty), назвемо його Background. У ньому створимо ще один порожній об'єкт Background images і помістимо спрайти з підвантаженого ассета.



Розташуємо їх на сцені в такий спосіб:



Продублюємо наявні спрайти і розташуємо вище початкових:



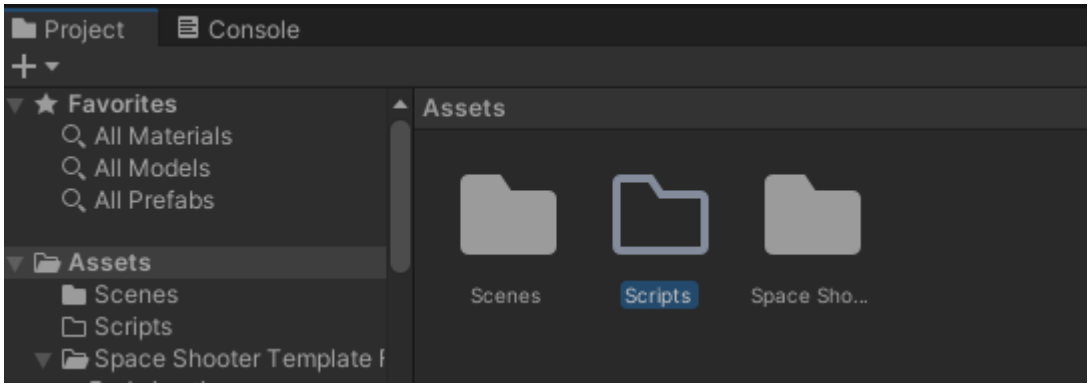
Скриптинг – необхідна складова для всіх ігор. Навіть найпростіші ігри потребують скриптів для реакції на дії гравця та організації подій геймплею. Крім того, скрипти можуть бути використані для створення графічних ефектів, управління фізичною поведінкою об'єктів або реалізації користувальницької AI системи для персонажів гри.

Поведінка ігрових об'єктів контролюється за допомогою компонентів (Components), які приєднуються до них. Незважаючи на те, що вбудовані компоненти Unity можуть бути дуже різнобічними, але часто необхідно реалізувати власні особливості геймплею. Unity

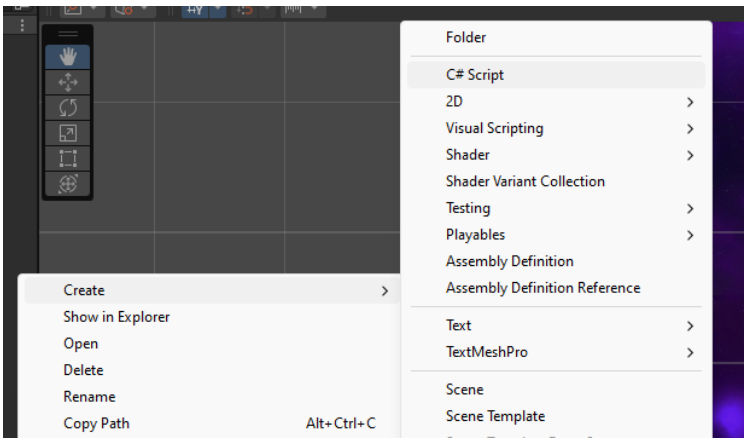
дозволяє створювати свої компоненти, використовуючи скрипти. Вони дозволяють активувати ігрові події, змінювати параметри компонентів, і відповідати на введення користувача в будь-який спосіб.

На відміну від інших ассетів, скрипти зазвичай створюються безпосередньо в Unity. Ви можете створити скрипт, використовуючи меню Create у лівому верхньому куті панелі Project або вибравши Assets → Create → C# Script у головному меню.

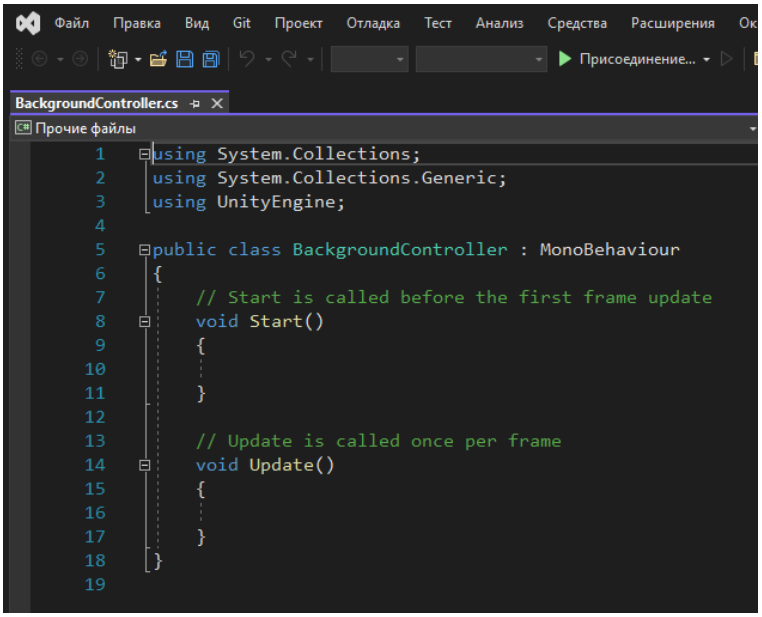
Створимо скрипт. Для цього в дереві проекту в папці Assets створимо нову папку Scripts



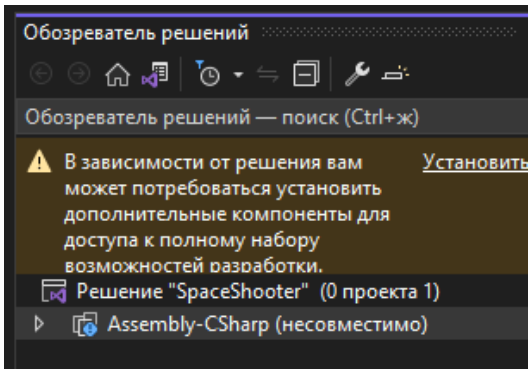
У папці Scripts створимо скрипт. Правою кнопкою миші всередині цієї папки вибираємо Create → C# Script. Вводим назву BackgroundController.



Подвійне клацання миші по іконці створеного скрипту відкриває Visual Studio і в ньому шаблон створеного скрипту:



Перевірте, щоб у налаштуваннях Unity стояла Visual Studio: Edit→Preferences... →External tools→External Script Editor – вибрати Visual Studio Зберегти проект. Закрити Visual Studio та перезапустити Unity. Подвійне клацання миші по іконці скрипта і він відкриється у Visual Studio. При відкритті Visual Studio може з'явитися повідомлення що потрібно встановити додаткові компоненти. Встановлюємо їх.



Скрипт взаємодіє із внутрішніми механізмами Unity за рахунок створення класу, успадкованого від вбудованого класу, званого MonoBehaviour. Ви можете думати про клас як про свого роду план створення нового типу компонента, який може бути прикріплений до ігрового об'єкта. Щоразу, коли ви приєднуєте скриптовий компонент до ігрового об'єкта, створюється новий екземпляр об'єкта, визначений планом. Ім'я класу береться з імені, яке ви вказали під час створення файлу. Ім'я класу та ім'я файлу повинні бути однаковими для того, щоб скриптовий компонент міг бути приєднаний до ігрового об'єкта.

Функція Update – це місце для розміщення коду, який оброблятиме оновлення кадру для ігрового об'єкта. Це може бути рух, спрацювання дій і реакція на введення користувача, в основному все, що повинно бути оброблено з часом в ігровому процесі. Щоб дозволити функції Update виконувати свою роботу, часто буває корисно ініціалізувати змінні, рахувати властивості та здійснити зв'язок з іншими ігровими об'єктами до того, як будуть здійснені будь-які дії.

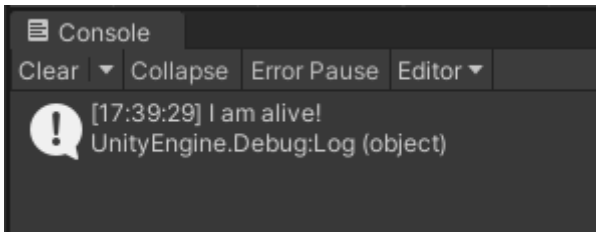
Функція *Start* буде викликана Unity до початку ігрового процесу (тобто до першого виклику функції Update) і це ідеальне місце для виконання ініціалізації.

Debug.Log це команда, яка просто виводить повідомлення на консольне виведення Unity.

Наприклад, код:

```
void Start()
{
   Debug.Log("I am alive!");
}
```

Якщо ви натиснете зараз Play, ви побачите повідомлення внизу основного вікна редактора Unity у вікні Console (меню: Window > Console).



Скрипт Unity не схожий на традиційну ідею програми, де код працює постійно в циклі, поки не завершить своє завдання. Натомість Unity періодично передає керування скрипту при викликі певних оголошених у ньому функцій. Як тільки функція завершує виконання, керування повертається назад до Unity. Ці функції відомі як функції подій, тобто їх активує Unity у відповідь на події, що відбуваються у процесі гри. Unity використовує схему іменування, щоб визначити, яку функцію викликати певної події. Наприклад, ми вже розглянули функцію Update (викликається перед зміною кадру) та функцію Start (викликається перед першим кадром об'єкта). У Unity є набагато більше функцій подій.

Додаємо наступний код:

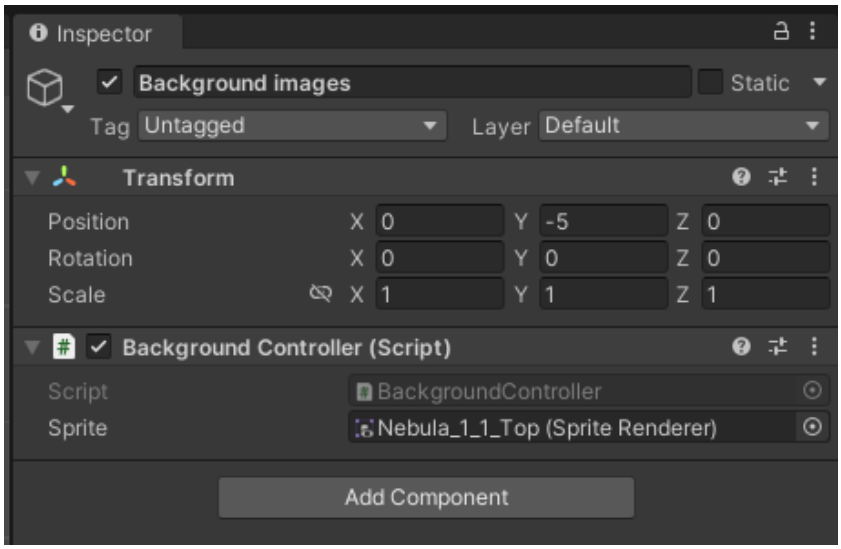
```
public class BackgroundController : MonoBehaviour
{
   [SerializeField]
   private SpriteRenderer sprite;

   private float speed = 2.5f;
   private float positionMin;
   private Vector2 positionRestart;

   // Start is called before the first frame update
   void Start()
   {
      positionRestart = transform.position;
      positionMin = sprite.bounds.size.y * 2 - positionRestart.y;
      //Debug.Log("I am alive!");
   }

   // Update is called once per frame
   void Update()
   {
      transform.Translate(Vector3.down * speed * Time.deltaTime);
      if (transform.position.y <= -positionMin)
         transform.position = positionRestart;
   }
}
```

Unity дозволяє змінювати значення змінних скрипта у запущеній грі. Це дуже корисно, щоб побачити ефекти від змін відразу ж, без зупинки та перезапуску. Коли програвання закінчується, значення змінних скидаються в стан, який вони мали до натискання кнопки Play.

Атрибути (Attributes) – це маркери, які можуть бути розміщені перед класом, властивостями або функціями в скрипті, щоб вказати особливу поведінку. Наприклад, атрибут HideInInspector може бути доданий перед оголошенням властивості для запобігання відображенню цієї властивості в інспекторі, навіть якщо воно публічне. У C# він міститься між квадратними дужками.

У C# ви повинні оголосити змінну загальнодоступною, щоб побачити її в інспекторі. Якщо ви також хочете, щоб Unity серіалізувала private поля (побачити їх в Інспекторі), ви можете додати до цих полів атрибут SerializeField.

Застосуємо створений скрипт на об'єкт Background images шляхом перетягування іконки скрипта до властивостей зазначеного об'єкта. Заповнимо значенням поле Sprite.



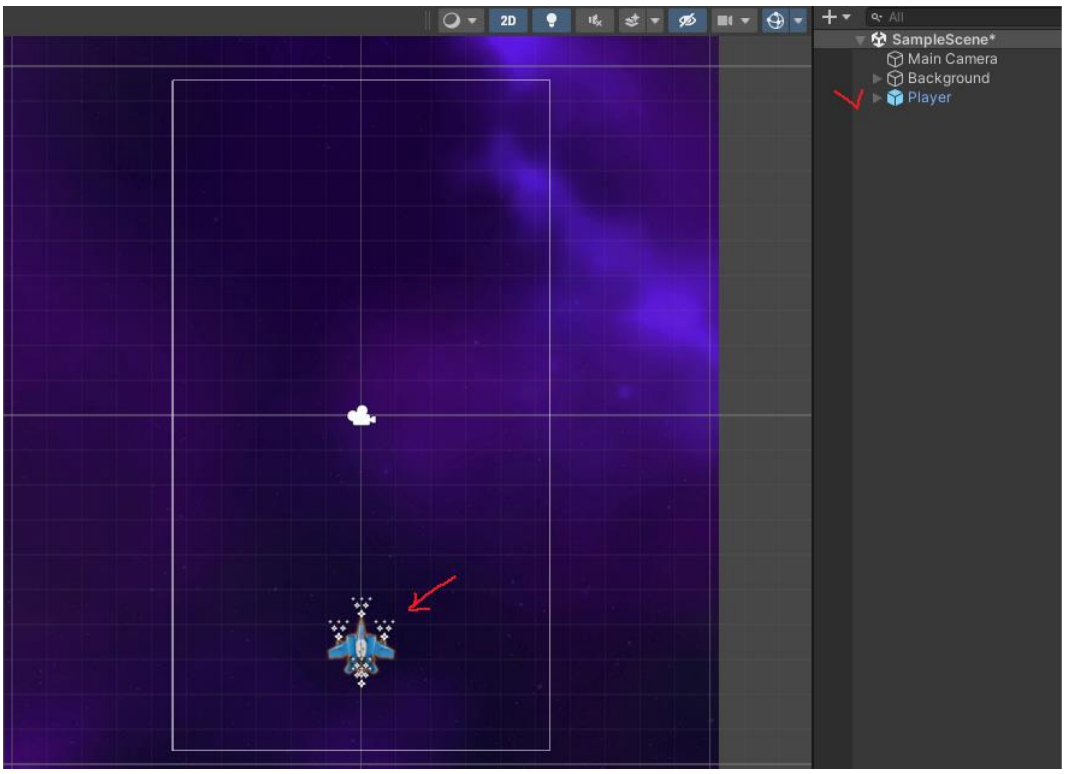
Запустимо проект гри. Тепер маємо нескінченне тло.

Крок 3. Додавання ігрового об'єкту

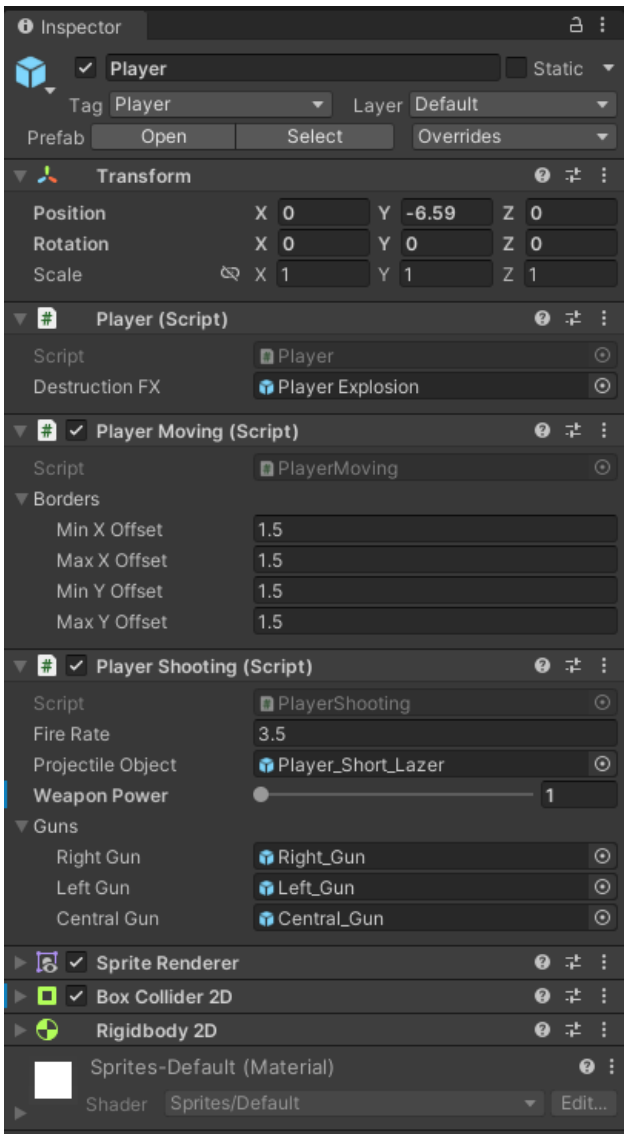
Додамо на сцену гравця як космічного корабля.

Система Prefab Unity дозволяє створювати, налаштовувати та зберігати GameObject у комплекті з усіма його компонентами, значеннями властивостей та дочірніми ігровими об'єктами як повторно використовуваний актив.

У нашому проекті такий Prefab ми візьмемо із встановленого асету. Для цього в панелі Project заходимо до папки Space Shooter Template FREE → Prefabs та перетягуємо на сцену Prefab під назвою Player.



Prefab під назвою Player вже має ряд компонентів, які додав розробник асета, що використовується. Зокрема, на об'єкт Player навішено три скрипти.: Player, PlayerMoving та PlayerShooting.



У скрипті Player є параметр Destruction FX - префаб відеоефекту, що генерується після смерті гравця.

У скрипті PlayerMoving є параметр Borders, що налаштовується - відступ від кожної межі екрана, який гравець не може проходити.

У скрипті PlayerShooting є параметри Weapon Power - тип зброї гравця, що впливає на стрілянину (шляхом за замовчуванням від 1 до 4) і Guns - гармати і променеві об'єкти в ієрархії гравців.

Компонент Sprite Renderer візуалізує двовимірні графічні об'єкти Sprite та керуйте його візуальним відображенням на сцені.

При створенні спрайту (GameObject → 2D Object → Sprite) Unity автоматично створює GameObject із прикріпленим компонентом Sprite Renderer. Ви також можете додати компонент до існуючого ігрового об'єкту через Компоненти (меню Component → Rendering → Sprite Renderer).

Компонент Box Collider 2D – це колайдер, який взаємодіє з системою 2D-фізики. Це прямокутник у формі з певним положенням, шириною та висотою в локальному просторі координат Sprite. Прямокутник вирівняний по осі, яке краї паралельні осям X чи Y локального простору.

Колайдер – невидима форма, що використовується для обробки фізичних зіткнень. колайдер не обов'язково має бути саме тієї ж форми, що й сітка об'єкта.

Виявлення зіткнень – автоматичний процес, що виконується Unity, який визначає, чи є GameObject з Rigidbody, що рухається, і колайдер увійшов в контакт з будь-яким іншим колайдером.

Система сценаріїв може визначати виникнення колізій та ініціювати дії за допомогою функції OnCollisionEnter. Однак також можна просто виявити, коли один колайдер входить у простір іншого, не створюючи зіткнення. Колайдер, налаштований як Тригер (використовуючи властивість Is Trigger), не веде себе як твердий об'єкт і просто пропускає інші колайдери. Коли колайдер входить у своє місце, тригер викликає функцію OnTriggerEnter для об'єкта тригера scripts.

Composite Collider 2D компонент – це Колайдер, який взаємодіє із системою 2D-фізики. На відміну більшості колайдерів, він визначає внутрішню форму. Натомість він поєднує форми будь-якого Box Collider 2D або Polygon Collider 2D, які ви налаштували для використання. Composite Collider 2D використовує вершини (геометрію) будь-якого з цих колайдерів та об'єднує їх разом у нову геометрію, керовану самим Composite Collider 2D.

Жорсткі тіла дозволяють вашим GameObjects діяти під контролем фізики. Rigidbody може сприймати сили та крутний момент, щоб ваші об'єкти рухалися реалістично. Будь-який GameObject повинен містити Rigidbody, щоб на нього вплинула гравітація, щоб він діяв під дією додаткових сил за допомогою сценаріїв або взаємодіяв з іншими об'єктами.

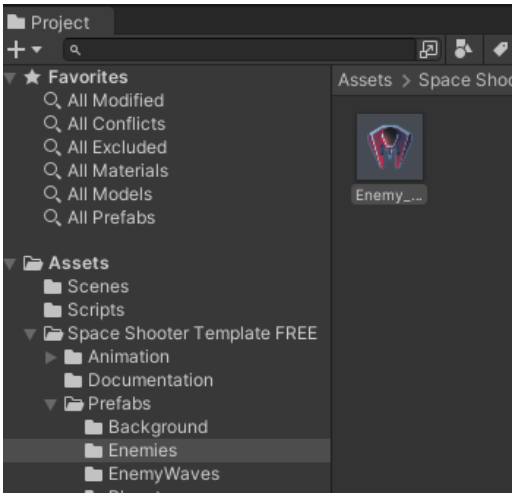
Компонент Rigidbody 2D поміщає об'єкт під управління фізичного двигуна. 2D об'єкти можуть рухатися тільки в площині XY і можуть обертатися навколо осі, перпендикулярної цій площині.

Зазвичай компонент Transform редактора Unity визначає, як GameObject (і його дочірні об'єкти GameObject) позиціонується, обертається та масштабується усередині Сцени. Коли він змінюється, він оновлює інші компоненти, які можуть оновлювати такі речі, як де вони відображаються або де колайдери.

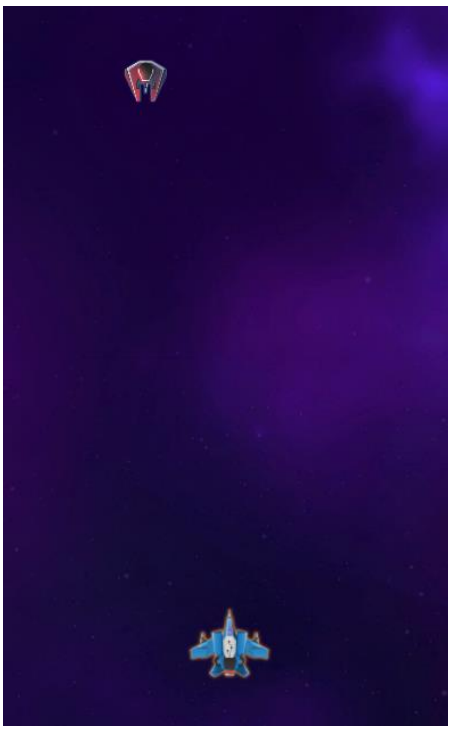
Крок 4. Додавання ворогів

Наступним кроком буде додавання на сцену ворогів та інших об'єктів.

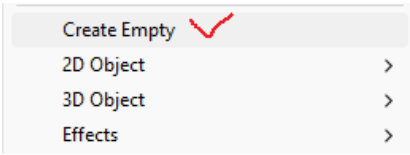
У встановленому асеті в папці Prefab можна знайти створений префаб ворога:



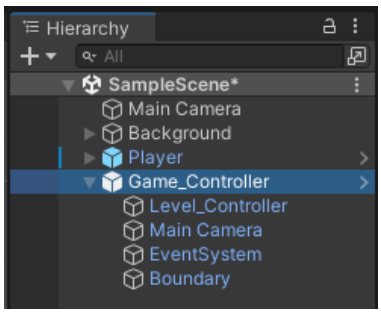
Якщо цей префаб просто перетягнути на сцену та запустити гру, то ворог перебуватиме там, де його поставили. Навіть якщо написати для нього скрипт, який його пересуватиме, то один ворог це мало.



В ідеалі нам потрібен потік таких ворогів, які з'являтимуться через деякий час у різних координатах. Для цього створюється порожній об'єкт GameObject і на нього навішується скрипт, який генеруватиме поява ворогів. Або створюється порожній об'єкт GameObject із дочірніми об'єктами і на ці об'єкти додаються відповідні скрипти.



Так як у цій роботі ми використовуємо встановлений ассет, а він вже містить такий об'єкт, то створювати порожній об'єкт не будемо. Перетягнемо на сцену з папки Prefab об'єкт під назвою Game_Controller. У нашому випадку він міститиме кілька дочірніх об'єктів:

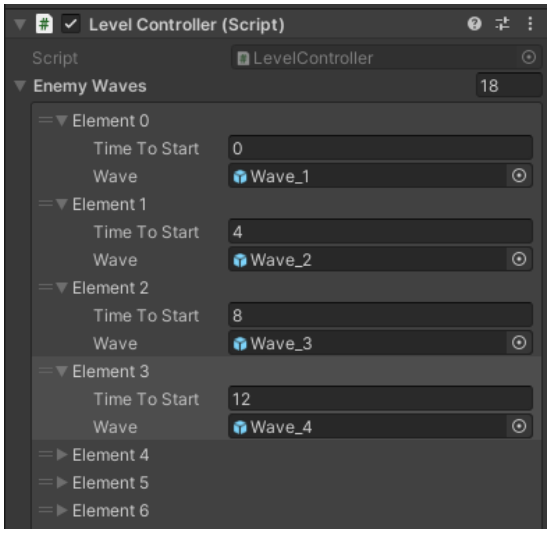


Boundary – об'єкт вказує на об'єкти, що перетинають лінію ігрового поля, знищує чи деактивує їх.

Level_Controller – контролює ігровий процес цього рівня, зокрема виробляє ворогів, бонуси та фонові об'єкти.

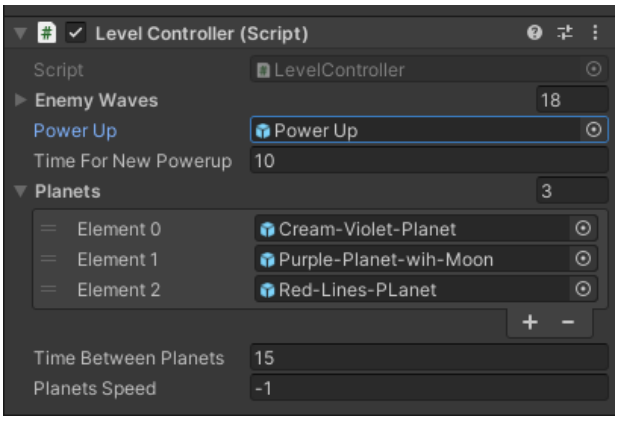
Для запуску рівня необхідно задати поля об'єкта Level_Controller, а саме Enemy Waves. У категорії «Enemy Waves» слід зазначити кількість ворожих хвиль, які з'являться на цьому рівні. Також потрібно вказати точку відліку від початку гри, коли з'являється

кожна хвиля (у секундах). Додаються заздалегідь підготовлені хвилі з папки Prefabs → EnemyWaves.

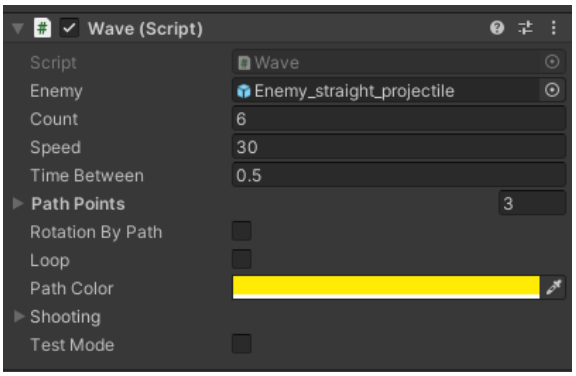


У Time For New Powerup встановіть, як часто повинен з'являтися бонус Power up.

У «Planets» налаштуйте планети-відьми, які ви збираєтеся використовувати, і як часто вони мають з'являтися.



Enemy Waves містить скрипт Wave, який генерує певну кількість ворогів протягом певного періоду часу і запускає їх по певному шляху.

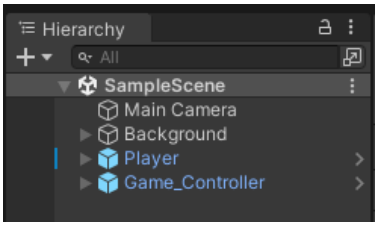


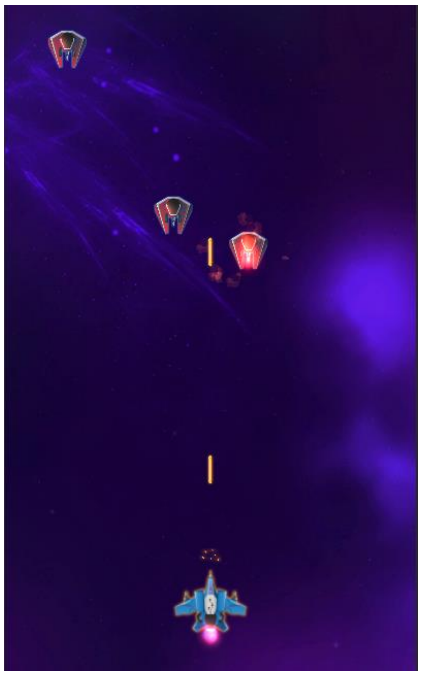
Поля скрипту «Wave»:

– «Enemy» – префаб ворога, який буде створений під час цієї хвилі;
– «Count» – кількість ворогів у цій хвилі;
– «Speed» – швидкість, з якою ворог проходить шлях;

- «Time Between» – період між появою ворогів;
- «Path Points» – точки, які пройде ворог під час проходження шляху;
- «Rotation Bypass» – чи обертатиметься ворог під час проходження шляху.
- «Loop» – якщо цикл активний, після завершення шляху ворог буде перегенерований у початковій точці;
- «Path Color» – колір, яким буде відзначений шлях у Редакторі;
- «Shooting»: «Shot chance» – ймовірність пострілу по противнику при проходженні шляху, а також мінімальний і максимальний час, коли противник спробує вистрілити.
- «Test Mode» – якщо тестовий режим активний, хвиля генеруватиметься постійно з інтервалом у 3 секунди. Ви можете використовувати тестовий режим для налаштування нової хвилі.

Тепер, коли на сцені ми розташували нескінченне тло, гравця та об'єкт, що відповідає за генерацію ворогів і планет можемо запустити гру на виконання:





2. Завдання до практичної роботи №2

Створити гру описану в лабороторній роботі.

Практична робота 3. 2D гра. Основні об'єкти

Мета роботи: познайомитися з основними об’єктами на сцені в Unity. Поняття ігровий об’єкт, сцена, матеріал. Створення ігровий об’єку засобами Unity. Поняття фізичний об’єкт. Додавання фізики до об’єкту. Створення гри на тему Ping pong. Навчитися додавати об’єкти та налаштовувати їх функціональність в програмі Unity.

1. Теоретичний матеріал

Клас Unity GameObject використовується для представлення всього, що може існувати у Сцені. Кожен об'єкт у вашій грі – це GameObject, від персонажів та колекційних предметів до джерел світла, камер та спеціальних ефектів.

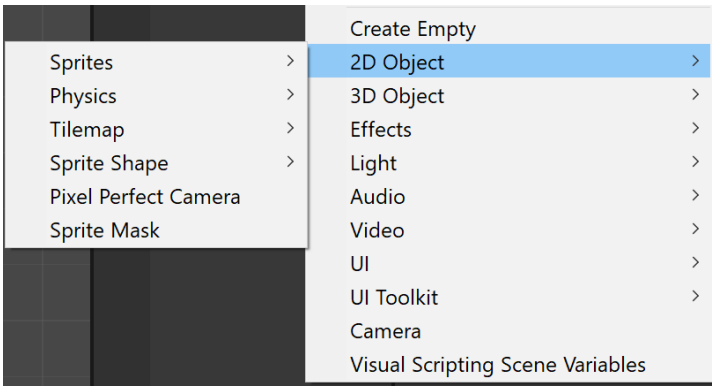
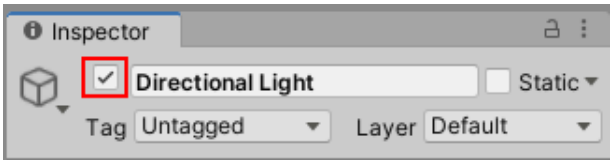


Рисунок 3.1 – Приклад GameObject

Ігрові об'єкти активні за замовчуванням, але їх можна деактивувати, що призведе до вимкнення всіх компонентів, прикріплених до ігрових об'єктів. Зазвичай це означає, що він стане невидимим і не отримуватиме звичайних зворотних викликів або подій, таких як Update або FixedUpdate. Статус ігрового об'єкта активний представлений прапорцем ліворуч від імені об'єкта.



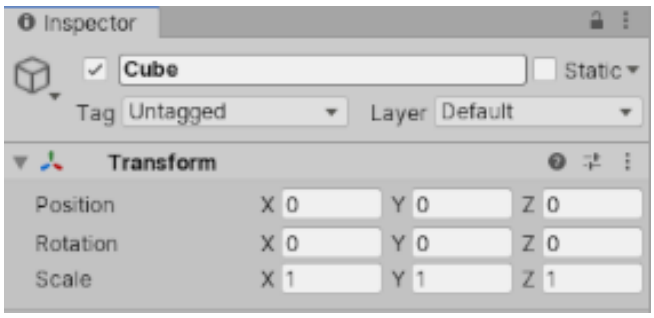
Тег – це довідкове слово, яке можна присвоїти одному або декільком GameObjects. Наприклад, ви можете визначити тег «Гравець» для персонажів, контрольованих гравцем, та тег «Ворог» для персонажів, які не контролюються гравцем. Ви можете визначити предмети, які гравець може збирати у Сцені з позначкою "Колекційний".

Теги допомагають ідентифікувати ігрові об'єкти для сценаріїв. Теги корисні для тригерів у колайдері, за допомогою яких можна з'ясувати, чи взаємодіє гравець, наприклад, з ворогом, реквізитом або предметом колекціонування.

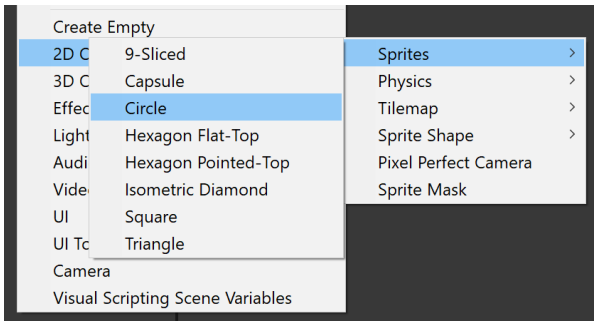
Самі собою GameObject нічого не роблять, але діють як контейнери для компонентів, де реалізована ця функціональність.

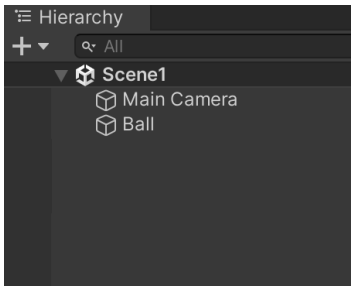
Щоб додати ігровому об'єкту властивості, необхідні для того, щоб він став джерелом світла, деревом або камерою, потрібно додати до нього компоненти. Залежно від того, який об'єкт ви хочете створити, ви додаєте до GameObject різні комбінації компонентів.

До GameObject завжди приєднано компонент Transform (для подання положення та орієнтації), і видалити його неможливо. Інші компоненти, які надають об'єкту його функціональність, можуть бути додані з меню Компонент редактора або скрипта.



Створимо новий проект 2D гри. У новому проекті перейменуємо назву сцени на Scene1. Створимо на сцені простий GameObject у вигляді кола (2D Object → Sprites → Circle) та назвемо його, наприклад Ball. Контекстне меню де можна обрати вид GameObject, що створюється можна визвати правою кнопкою миші в панелі Hierarchy або в меню GameObject.





Компонент Rigidbody – це основний компонент, що включає фізичну поведінку для об'єкта. З прикріпленим Rigidbody об'єкт негайно почне реагувати на гравітацію. Якщо додано один або кілька компонентів Collider, то при колізіях (зіткненнях) об'єкт пересуватиметься.

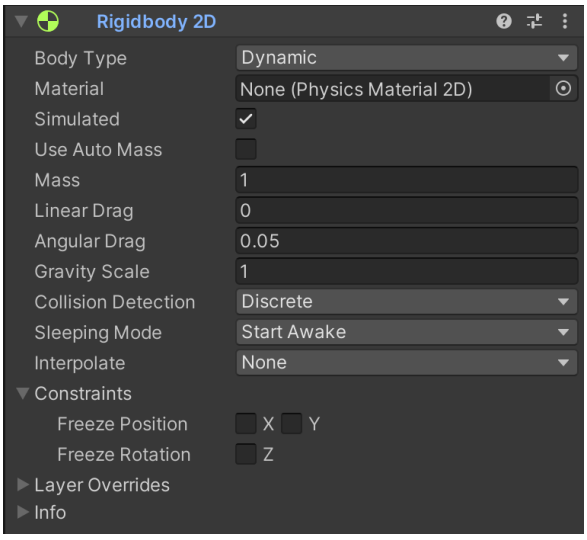


Рисунок 3.2 – Вікно компоненту Rigidbody

Для того щоб об’єкт Ball став фізичним додаємо до нього компонент Rigidbody. Для цього в панелі Inspector натискаємо кнопку Add Component та обираємо Rigidbody 2D.

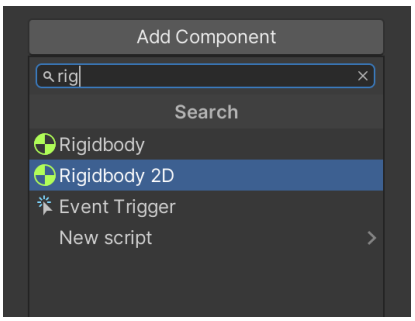


Рисунок 3.3 – Пошук у вікні компоненту Rigidbody

Якщо зараз запустити гру, на об’єкт Ball почне падати вниз та вийде за межі камери так як на нього впливає гравітація.

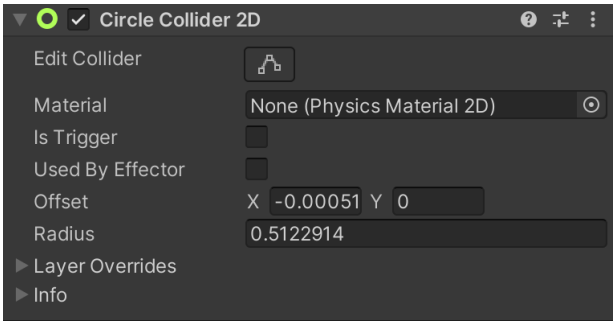
Компоненти Collider визначають форму GameObject для цілей фізичних зіткнень. Коллайдер не обов’язково має бути такої ж форми, як сітка GameObject. Приблизне наближення сітки часто є більш ефективним і нерозрізним у ігровому процесі.

Найпростіші коллайдери є примітивними типами коллайдерів.

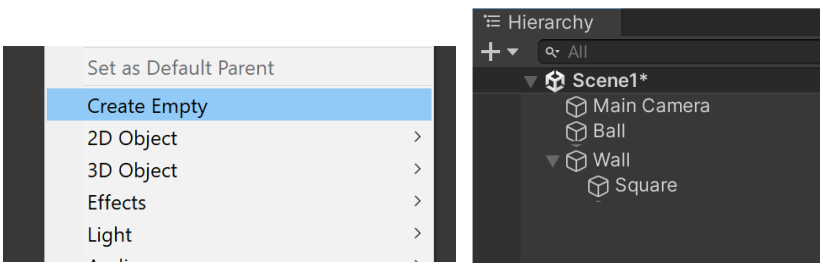
У 3D це Box Collider, Sphere Collider і Capsule Collider. У 2D ви можете використовувати Box Collider 2D і Circle Collider 2D.

Ви можете додати будь-яку їх кількість до одного GameObject, щоб створити складені коллайдери.

Для того щоб об’єкт Ball міг взаємодіяти з іншими об’єктами додаємо до нього компонент Collider. Для цього в панелі Inspector натискаємо кнопку Add Component та обираємо Circle Collider 2D.

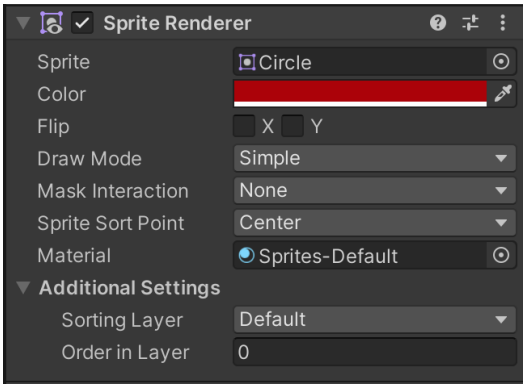


Щоб об’єкт Ball не летів у нескінченність у низ, додаємо на сцені новий об’єкт. Створимо пустий об’єкт GameObject → Create Empty (правою кнопкою миші в панелі Hierarchy). Назвемо його Wall. В ньому створимо ще один об’єкт 2D Object → Sprites → Square.

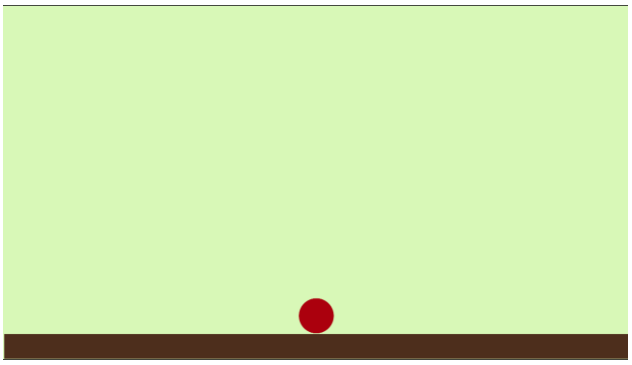


Lодаємо до Wall компонент Collider. Для цього в панелі Inspector натискаємо кнопку Add Component та обираємо Box Collider 2D.

Замінимо колір створених об’єктів Ball та Square. Для цього в панелі Inspector відкриваємо вже існуючий компонент Sprite Renderer та у властивості Color змінюємо колір на потрібний.



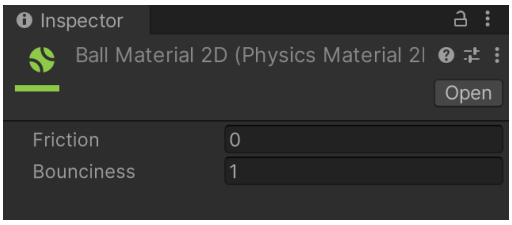
Запустимо гру. Тепер об’єкт Ball зупиниться на верхній поверхні об’єкту Wall.

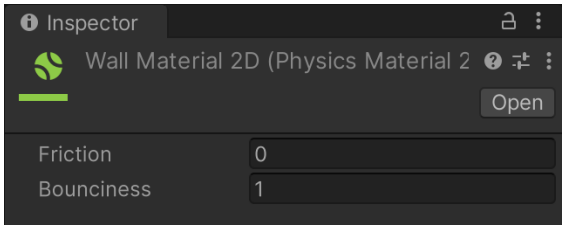


Додаймо до об’єктів Ball та Wall можливість фізично взаємодіяти. Для цього в їх коллайдерах необхідно додати фізичний матеріал.

Physics Material 2D використовується для налаштування тертя та відскакувань, які відбуваються між 2D фізичними об'єктами при їх зіткненні. Створити 2D фізичний матеріал можна через меню Assets (Assets → Create → Physics Material 2D).

Створимо фізичний матеріал для об’єктів Ball та Wall. З початку в папці Assets створимо нову папку Material. В цій папці створимо два матеріалу Ball Material 2D та Wall Material 2D.





Фізичний матеріал має такі властивості:

– Friction – коефіцієнт тертя для даного колайдера.
– Bounciness – ступінь відскоку зіткнень від поверхні. Значення 0 означає, що немає відскоку, в той час як значення 1 означає хорошу відскакуваність без подальшої втрати енергії.

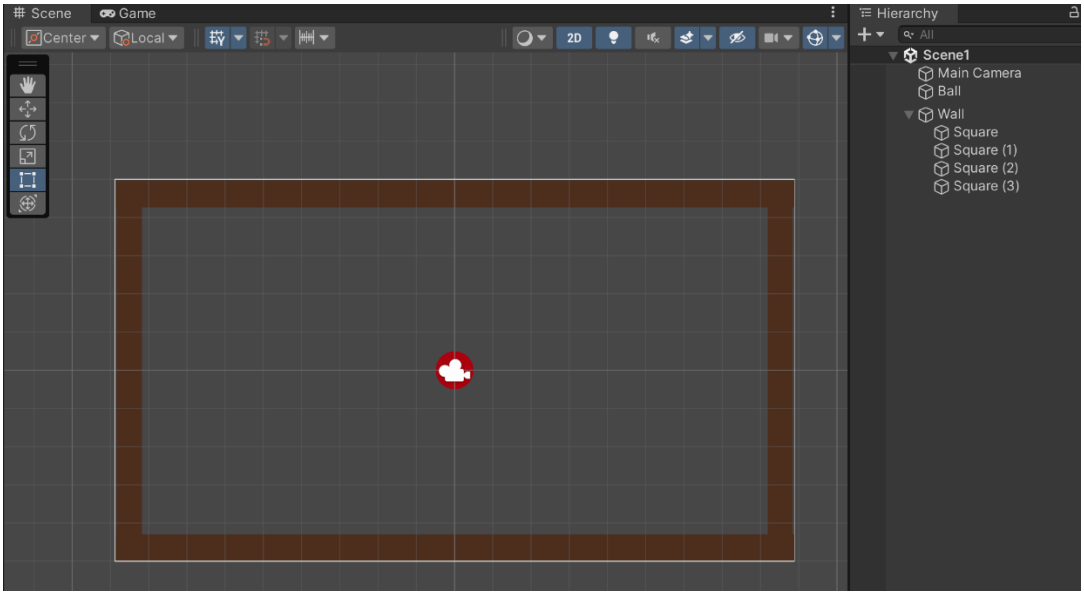
Для використання 2D фізичного матеріалу просто перетягніть його на компонент колайдера в інспекторі.

Якщо тепер запустити гру, об’єкт Ball буде відскакувати перпендикулярно об’єкту Wall. Несемо різноманітність. Створимо скрипт для об’єкту Ball з назвою Ball.

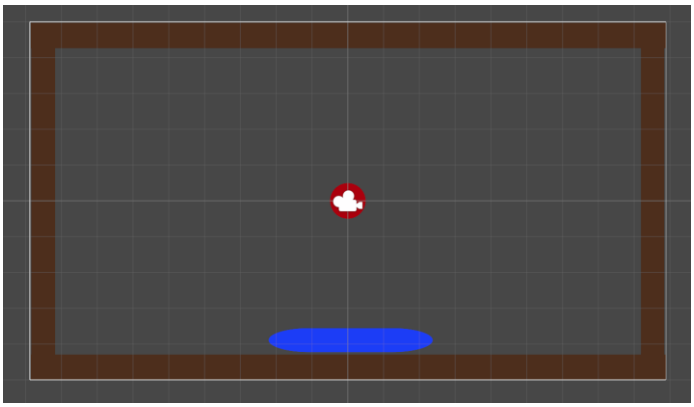
```
void Start()
{
    const float MinImpulse = 3f;
    const float MaxImpulse = 15f;
    float angle = Random.Range(0, 2*Mathf.PI);

    Vector2 direction = new Vector2(Mathf.Cos(angle), Mathf.Sin(angle));
    float magnitude = Random.Range(MinImpulse, MaxImpulse);

    GetComponent<Rigidbody2D>().AddForce(direction* magnitude, ForceMode2D.Impulse);

}
```

Якщо тепер запустити гру, об’єкт Ball буде відскакувати від об’єкту Wall з різною силою та під різними кутами.

Щоб Ball після відскоку не вилітав за ліву та праву границю камери, необхідно об'єкт Wall доповнити ліворуч, праворуч та нагорі об’єктами Square. Кожному створеному об'єкту Square додати компонент і фізичний матеріал Wall Material 2D.



Додаймо до гри платформу, яка буде рухатися вліво та вправо в нижній частині екрану. Створимо ще один об’єкт 2D Object → Sprites → Capsule та назвемо його Racket. Додаймо до нього компонент Capsule Collider 2D. Змінимо колір на інший. Створимо новий матеріал Racket Material 2D з такими ж властивостями, що і попередні. Додаймо матеріал до коллайдера.



Створимо новий скрипт для об’єкту Racket з такою ж назвою:

```
[SerializeField]
private float speed = 10f;

private float lastMouseX;

/// <summary>
/// Start is called before the first frame update
/// </summary>
void Start()
{
    lastMouseX = Input.mousePosition.x;
}

/// <summary>
/// Update is called once per frame
/// </summary>
void Update()
{
    var position = transform.position;

    Vector2 direction = Vector2.zero;

    if (Input.mousePosition.x > lastMouseX)
        direction = Vector2.right * speed;
    else if (Input.mousePosition.x < lastMouseX)
        direction = Vector2.left * speed;

    lastMouseX = Input.mousePosition.x;

    position.x += direction.x * Time.deltaTime;
    transform.position = position;
}
```

Запустимо гру. Об’єкт Racket буде рухатися в напрямку покажчика миші. Таким чином, користувач може управляти об’єктом.

Для складності гри, можна додатково додати декілька об’єктів Ball різного розміру.

2. Завдання до лабораторної роботи №3

Створити 2D – гру, просту версію гри Пінг-понг. Розібратися з налаштуаннями компонент Rigidbody та Collider.

Практична робота 4. 2D-гра. Анімація персонажу

Мета роботи: познайомитися зі створенням 2D-гри на Unity. Поняття спрайту. Види спрайтів. Анімація персонажу Навчитися додавати персонажів на сцену, створювати анімацію персонажів в програмі Unity.

1. Теоретичний матеріал

Робочий процес при створенні двовимірної та тривимірної графіки в Unity приблизно однаковий і включає імпорт графічних ресурсів, перетягування їх у сцену і написання сценаріїв, які потім приєднуються до об'єктів. Основний вид ресурсів, необхідні створення двовимірної графіки, називається спрайтом.

Спрайти (sprites) являють собою двовимірні зображення, що відображаються безпосередньо на екрані, в той час як такі ж зображення, що відображаються на поверхні тривимірних моделей, називаються текстурами.

Створимо новий проект 2D гри. У новому проекті перейменуємо назву сцени на Scene1. Створимо папку Sprits, де будемо зберігати спрайти героїв гри. Добавимо до цієї папки зображення з персонажем.

Іноді текстура спрайту містить лише один елемент графіки, але часто набагато зручніше об'єднати кілька зображень, пов'язаних один з одним в одне зображення. Наприклад, зображення може містити складові персонажа, як для машини колеса якої рухаються незалежно від корпусу.

У двомірних іграх повсюдно поширені анімовані спрайти. Вони створюються малюванням кожного кадру анімації та подальшого відтворення отриманої послідовності кадрів у Unity.

Кадри можна імпортувати у вигляді окремих зображень, але їх зазвичай викладають у вигляді однієї картинки, яка називається *листом спрайтів* (sprite sheet). Листи спрайтів можна автоматично генерувати в Unity. При імпорті аркуша у списку Sprite Mode налаштувань спрайту слід вибрати варіант Multiple.

Рисунок 4.1 – Приклад листу спрайтів

Якщо файл постачався без файлу meta, то для Unity це просто картинка, яку потрібно розбити на кадри. Щоб це зробити виділяємо в папці відповідний файл та в Inspector натискаємо кнопку Sprite Editor. Unity Sprite Editor дозволяє витягувати елементи складеного зображення.

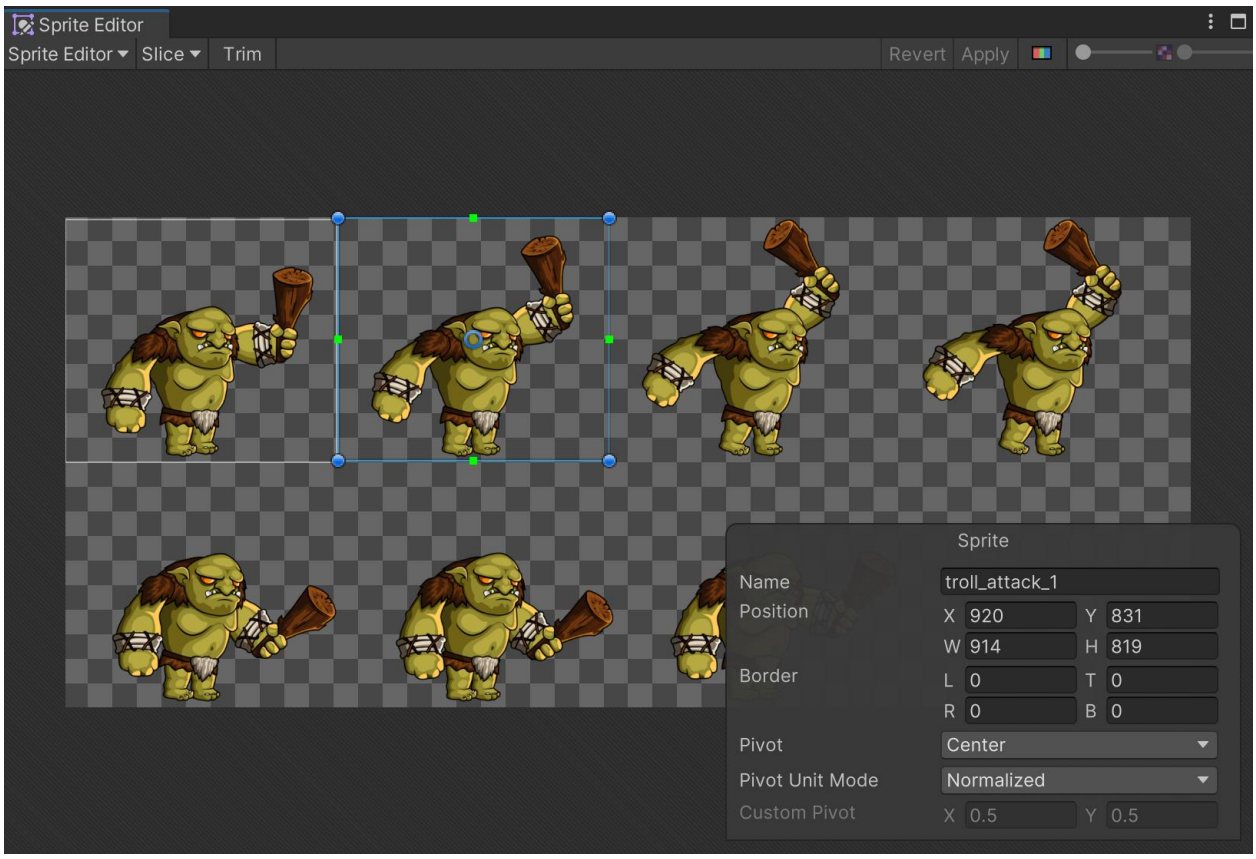


Рисунок 4.2 – Вікно Sprite Editor

Крім складеного зображення, ви побачите різні елементи керування у заголовку вікна редактора. Слайдер у правому верхньому куті керує наближенням, тоді як кнопка з кольоровими смужками ліворуч від нього перемикає режим відображення альфа-каналу та звичайний вигляд зображення.

Найважливіший елемент керування це меню Slice у лівому верхньому кутку, який надає опції для автоматичного нарізування елементів зображення.

Кнопки Apply та Revert дозволяють зберегти або скасувати зроблені зміни.

Найбільш прямим способом використання редактора є ідентифікація елементів вручну. Якщо натиснути на зображення, то з'явиться прямокутна область виділення з маркерами по кутах. Ви можете перетягнути маркери або краї прямокутника, щоб змінити його розмір навколо певного елемента.

Виділивши елемент, можна додати ще один, перетягнувши новий прямокутник в окрему частину зображення. Ви помітите, що коли ви виділите прямокутник, у нижньому правому куті вікна з'явиться панель:

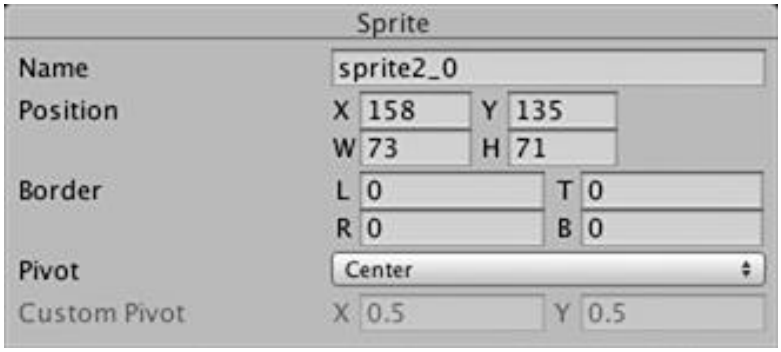
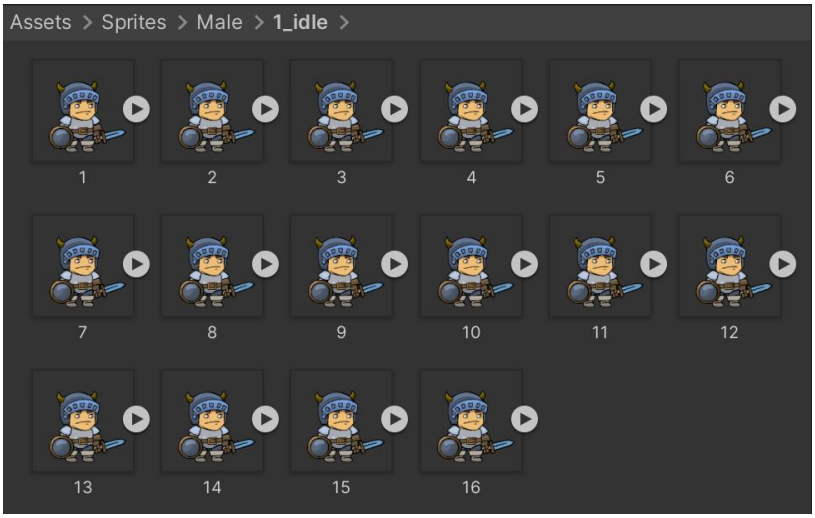


Рисунок 4.3 – Панель для ручного налаштування розміру кадру

Елементи керування на панелі дозволяють вибрати назву для графіки спрайтів та встановити положення та розмір прямокутника за його координатами. Ширина кордону для лівого, верхнього, правого та нижнього краю може бути вказана в пікселях. Також є налаштування для звороту спрайту, який Unity використовує як координату початку координат і головну «опорну точку» (Pivot) графіки.

Розглянемо другий спосіб анімації, коли кадри анімації розташовуються кожний в окремому файлі.

Наприклад маємо такі файли:



Перший кадр додаємо на сцену. Відкриваємо вікно Animation (Window → Animation → Animation).

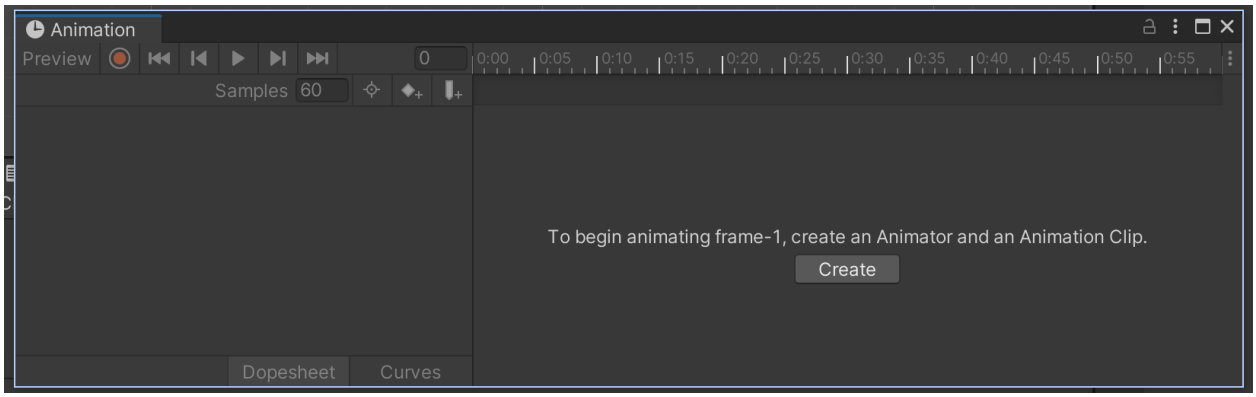
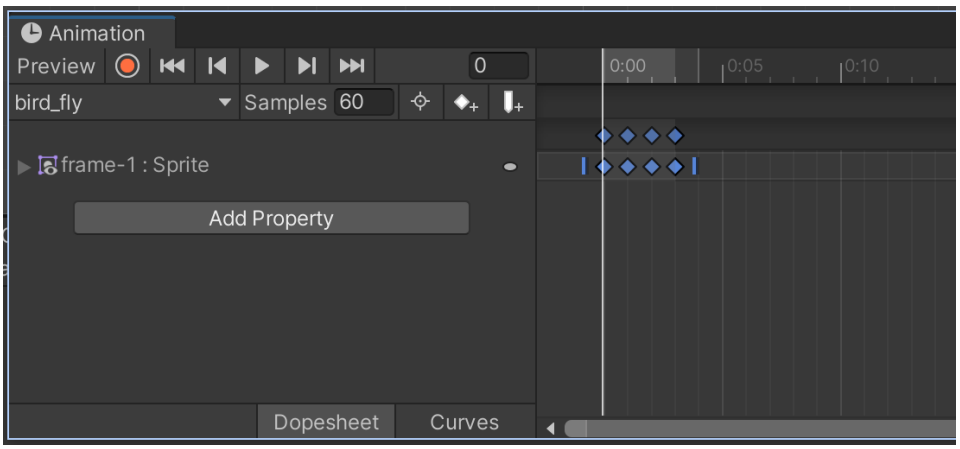


Рисунок 4.3 – Вікно Animation

Вікно Animation (Animation View) дозволяє створити та модифікувати кліпи анімації (Animation Clips) прямо всередині Unity.

Вікно Timeline (шкали часу) дозволяє створювати кінематографічний вміст, ігрові послідовності, аудіоряди та складні ефекти частинок. Ви можете анімувати багато різних ігрових об’єктів в одній послідовності.

У вікні Animation натискаємо кнопку Create. Unity запропонує дати назву та зберегти файл анімації. Далі виділяємо всі спрайти, які є кадрами анімації, що створюється, та переносимо їх на шкалу timeline. Кожний кадр позначиться там окремим маркером.



За замовчанням кількість кадрів в секунду показується в полі Samples. Можна запустити подивитися анімацію натиснувши кнопку Play в окні Animation.

Якщо анімація рухається надто швидко її можна уповільнити, змінивши кількість кадрів на секунду, а саме зменшивши це число. Методом підбору знаходимо необхідну частоту кадрів, у якому анімація виглядає природною.

Коли перетягуємо спрайт з анімацією вперше на сцену, то Unity запропонує зберегти анімацію. Зробимо це в створену папку Animations. Назву файлу даємо як:

назваПерсонажа_типАнімації.anim

Наприклад: troll_walk.anim.

Аналогічно створимо всі необхідні анімації персонажу.

Для зміни розміру персонажу виділяємо в папці проекту файл спрайта, в Inspector змінемо значення в полі Pixels per unit. Зменшення числа – збільшує спрайт та навпаки.

При створенні анімації створилися файли контролери.

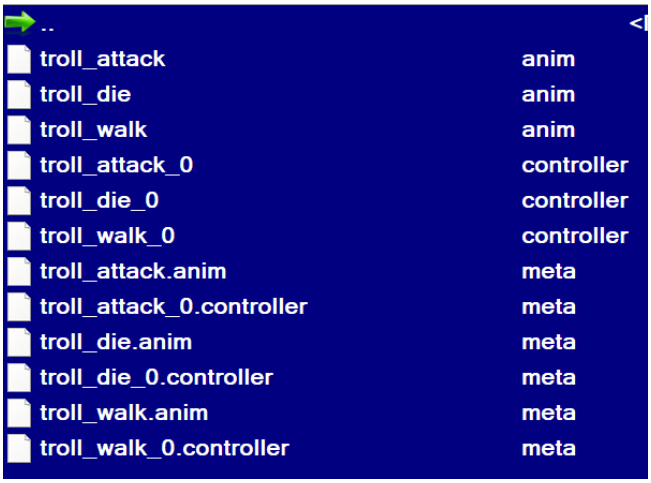


Рисунок 4.5 – Приклад переліку файлів анімації персонажу

2. Завдання до лабораторної роботи №4

Створити 2D – гру, додати персонажу до сцени, створити анімацію для доданого персонажу.

Практична робота 5. 2D-гра. Управління персонажем

Мета роботи: познайомитися зі створенням 2D-гри на Unity. Клас Input. Вікно Animator. Створення скриптів для управління персонажем, та зміни анімацій. Навчитися додавати персонажів та інші ресурси на сцену, створювати анімацію персонажів, обробляти події від користувача в програмі Unity.

1. Теоретичний матеріал

При створенні анімації створилися файли контролери.

Залишемо один головний а інші видалімо. Найчастіше головним буде контролер для анімації Idle (Стояти). Перейменуємо його на troll.controller. Відкриємо його в редакторі анімації.

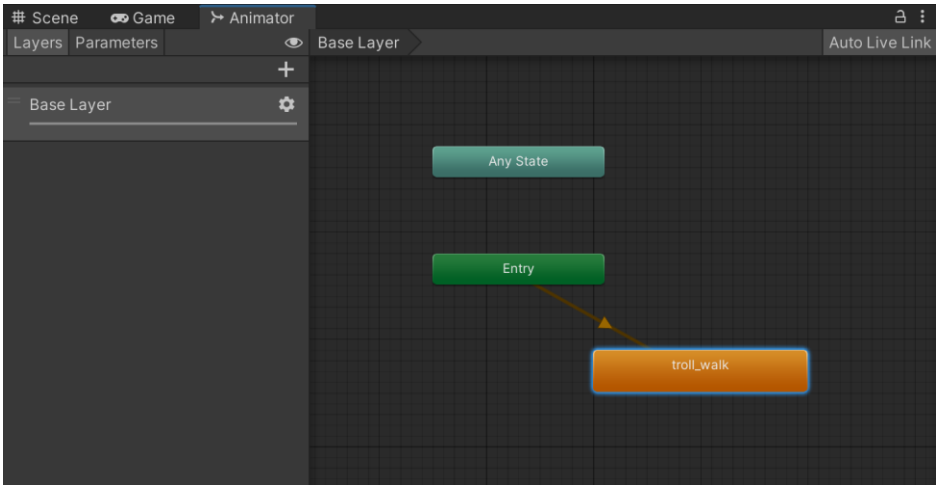


Рисунок 5.1 – Вікно Animator

В вікні редактору анімації – рух по полю натискаємо колесо миші та рухаємо вправо, вліво, вверх, вниз.

Ресурс Animator Controller створюється в Unity і дозволяє впорядковувати та підтримувати набір анімацій для персонажа чи об’єкта. У більшості ситуацій нормально мати кілька анімацій і перемикатися між ними, коли виникають певні умови гри.

Зазвичай персонаж має кілька анімацій, які відповідають різним діям, які можуть бути у грі. Наприклад, він може просто дихати і похитуватися в спокої, йти по команді і скидати руки в паніці при падінні з платформи. У дверей можуть бути анімації відкриття, закриття, заклинювання та злому.

Однак, навіть якщо у вас є лише один анімаційний кліп, вам все одно потрібно розмістити його в контролері аніматора, щоб використовувати його на ігровому об’єкті.

Контролер має посилання на анімаційні кліпи, що використовуються в ньому, і керує різними станами анімації та переходами між ними за допомогою так званого кінцевого автомата, який можна розглядати як певний блок-схему або просту програму, написану на мова візуального програмування в Unity.

Основна ідея полягає в тому, що персонаж бере участь у якомусь конкретному виді дій у будь-який час. Типові дії включатимуть такі речі, як стійка на місці (idling), ходьба (walking), біг (running), стрибок (jumping) тощо. Ці дії називають states, у тому сенсі, що персонаж перебуває у «стані», де він іде, стоїть чи робить ще щось.

Налаштування для наступного стану, в який гравець зможе перейти з поточного стану, називаються state transitions. Разом взяті набори станів, набори змінних та змінні пам'ятають поточну форму стану state machine.

Стани та переходи механізму станів можна уявити за допомогою графіка, де вузли будуть являти собою стани та дуги (стрілки між вузлами), що представляють переходи.

Вікно Animator складається з двох основних розділів: основної області макета з сіткою та лівої панелі «Layers & Parameters».

Ліва панель Parameters дозволяє створювати, переглядати та редагувати змінні. Ці змінні, які ви визначаєте, діють як вхідні дані в кінцевий автомат. Щоб додати параметр, клацніть значок «+» і виберіть тип параметра зі спливаючого меню. Щоб видалити параметр, виберіть параметр у списках і натисніть клавішу видалення.

Автомати стану складаються зі станів, переходів і подій, а менші автомати підстанів можна використовувати як компоненти у великих машинах.

Вузол Any State – це особливий стан, який існує завжди. Воно підходить для ситуацій, коли ви хочете перейти на певний стан, незалежно від поточного стану.

Вузол Entry використовується при переході в кінцевий автомат. Вузол входу буде оцінено та розгалужується до стану призначення відповідно до встановлених умов. Таким чином вузол Entry може контролювати, в якому стані починається кінцевий автомат, оцінюючи стан ваших параметрів, коли починається кінцевий автомат.

Вузол Exit використовується для вказівки, що кінцевий автомат має завершити роботу.

Перетягнемо до вікна Animator анімації персонажа, що маємо. Додаймо змінні, які допоможуть перейти з одного стану в інший.

– Float speed;
– Trigger attack;
– Bool jump;

Щоб додати перехід на новий стан, правою кнопкою миші по поточному стані та обрати Make Transition та дотягнути лінію до нового стану. Далі потрібно вказати умову переходу. Для цього натискаємо на лінію переходу і в Inspector з’явиться наступне вікно:

Знімаємо прапорець з Has Exit Time коли не потрібно щоб попередня анімація спочатку програлась уся, а потім виконався перехід на наступну анімацію. Коли прапорець знято, то попередня анімація програється не до кінця, а як тільки виконується умова переходить до іншої анімації. Натискаємо + в Conditions та додаємо умову для переходу.

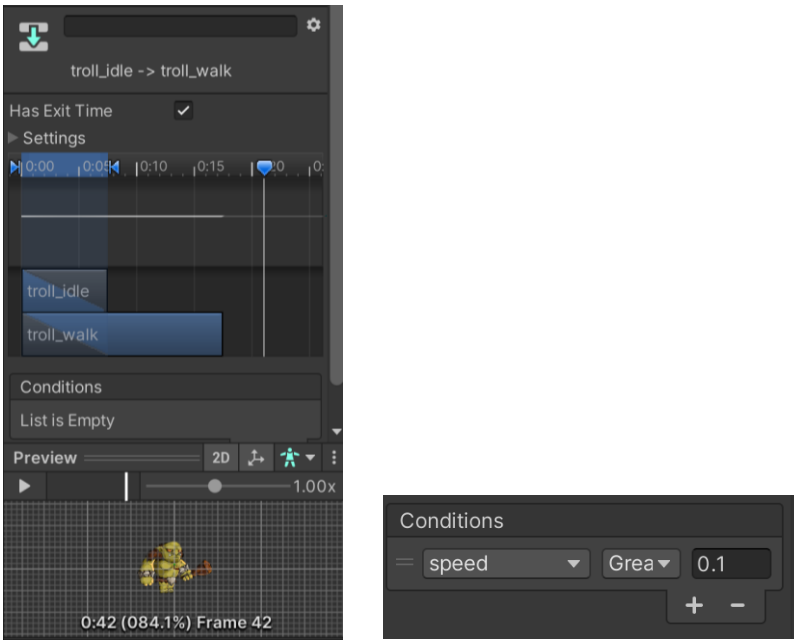


Рисунок 5.2 – Налаштування переходу з одного стану до іншого

Коли персонаж закінчить йти потрібно повернутися в стан Idle. Тоді робимо новий перехід аналогічним чином до стану Idle.

В результаті налаштувань вікно Animator може мати наступний вигляд:

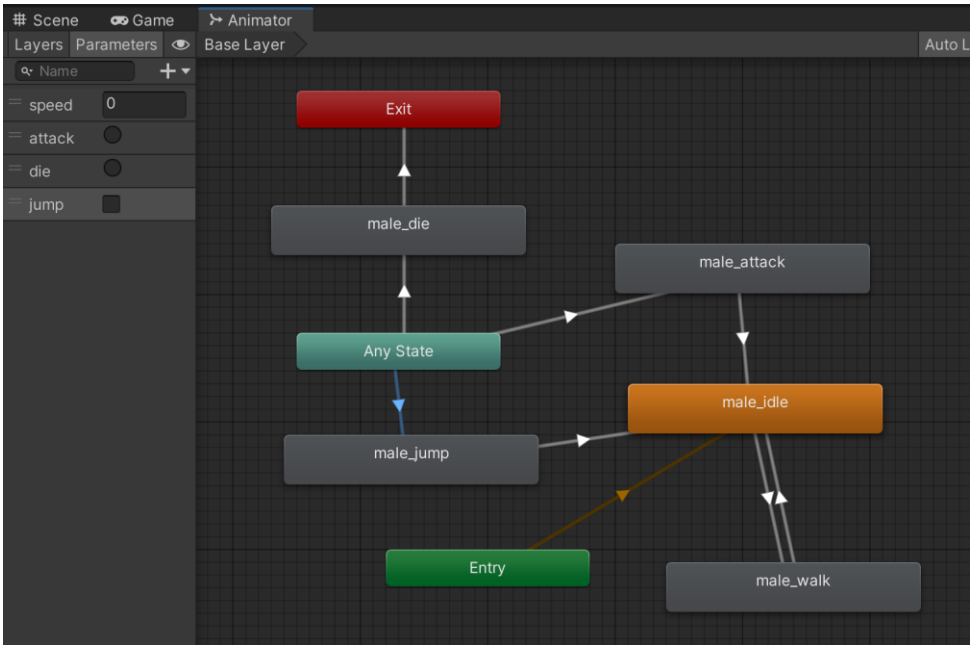


Рисунок 5.3 – Вікно Animator для всіх анімацій персонажу

Додаймо управління до нашого персонажу. Для цього створимо скрипт, наприклад, Hero та прикріпимо його до головного персонажу.

Клас Input у Unity надає доступ до введення на мобільних пристроях, таких як натискання на екран, акселерометр та географічні/локаційні дані. Доступ до клавіатури на мобільних пристроях забезпечується через keyboard iOS. Клас Input також використовується для визначення осей введення та пов'язаних з ними дій у вашому проекті.

Для доступу до менеджера введення перейдіть до головного меню Unity, потім виберіть Edit > Project Settings > Input Manager.

Кожен проект, який ви створюєте, має низку вхідних осей, створених за замовчуванням. Ці осі дозволяють відразу використовувати введення з клавіатури, миші та джойстика у проекті.

Наприклад, щоб запросити поточне значення горизонтальної осі та зберегти його у змінній, можна використовувати Input.GetAxis ось так:

```
float horizontalInput = Input.GetAxis("Horizontal");
```

Для осей, які описують подію, а не рух (наприклад, стрілянина зі зброї у грі), використовуйте Input.GetButtonDown замість цього.

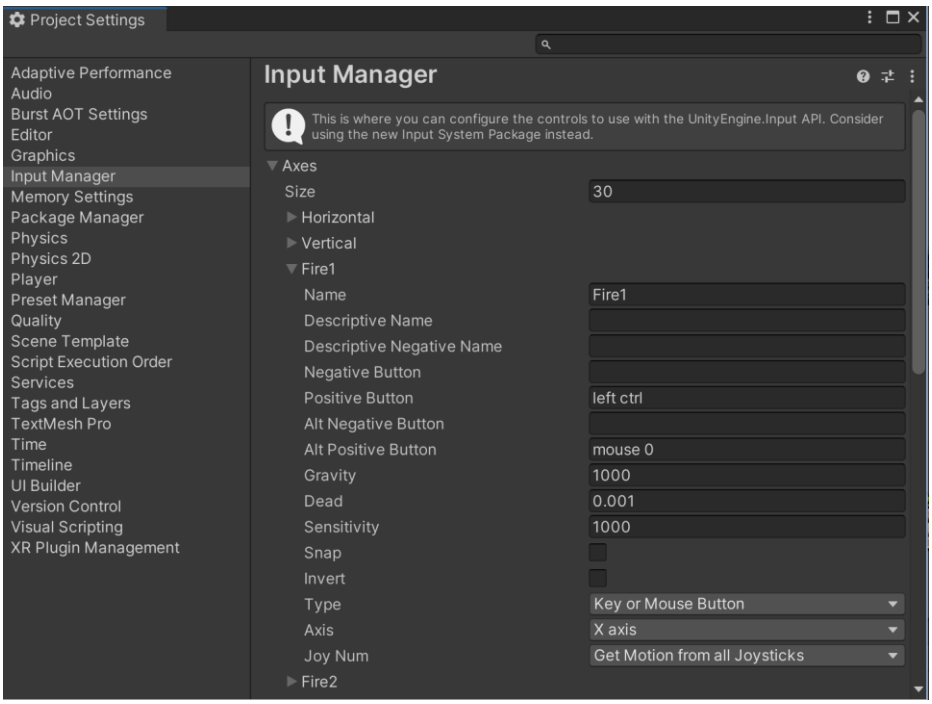


Рисунок 5.4 – Менеджер введення (Input Manager)

Перед тим як додати код до створеного скрипту необхідно персонажам додати такі компоненти:

– Rigidbody2D
– Коллайдер (CircleCollider2D, BoxCollider2D)

Додаємо до персонажу компонент Rigidbody 2D та 2 колайдера. Налаштуйте розміри та місцеположення колайдерів за допомогою кнопки EditCollider.

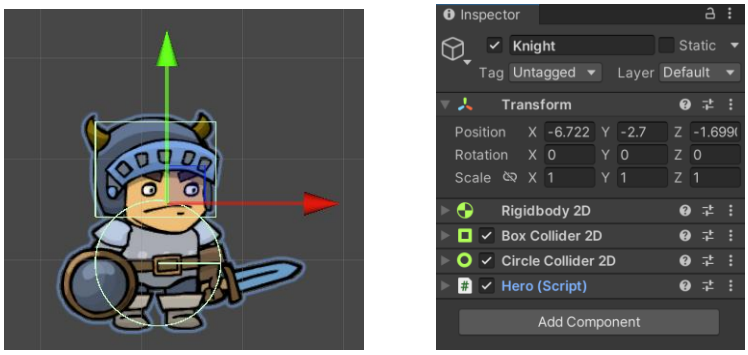


Рисунок 5.5 – Приклад персонажу с Rigidbody 2D та колайдерами

У Rigidbody2D поставте галочку Constraints -> Freeze Rotation -> Z. Відкрийте створений скрипт для персонажа.

Оголосимо змінну типу Animator (назвіть її animator) і ще одну типу Rigidbody2D.

```
[SerializeField]
private Animator animator;

[SerializeField]
private Rigidbody2D rigidbody;
```

Параметрам анімації можна призначати значення зі сценарію за допомогою функцій у класі Animator:

– SetFloat,

– SetInteger,
– SetBool,
– SetTrigger
– ResetTrigger

Використовуйте SetFloat() у сценарії для надсилання дійсних значень до аніматора для активації переходів.

SetFloat (string name, float value);

Використовуйте SetInteger() у сценарії для надсилання цілих значень до аніматора для активації переходів.

SetInteger (string name, int value);

Використовуйте SetBool() у сценарії для надсилання булевих значень до аніматора для активації переходів.

SetBool (string name, bool value);

Метод SetTrigger() дозволяє встановити (тобто активувати) тригер анімації, щоб викликати зміну потоку в кінцевому автоматі аніматора контролера.

SetTrigger (int id)

Аніматор налаштований так, щоб включати анімацію ходьби тоді, коли швидкість більша за нуль. Якщо швидкість менша за нуль, персонаж стоїть на місці. Передаватимемо не швидкість безпосередньо, а модуль швидкості - її абсолютне значення, незалежно від знака:

```
animator.SetFloat("speed", Mathf.Abs(Input.GetAxis("Horizontal")));
```

Далі додамо код, який рухатиме персонаж по горизонталі. Робитимемо це, задаючи значення швидкості Rigidbody2D вручну.

```
Vector2 velocity = rigidbody.velocity;
velocity.x = Input.GetAxis("Horizontal") * speed;
rigidbody.velocity = velocity;
```

Множник speed – це змінна, яку треба оголосити у класі, та вивести в інспектор. Змінюючи її, контролюємо швидкість переміщення персонажа.

```
[SerializeField]
private float speed = 2.0f;
```

Якщо запустити гру, то персонаж повинен ходити вліво-вправо, при цьому залишаючись обличчям в один і той же бік. Щоб персонаж повертався обличчям у потрібну сторону, додаймо наступний код:

```
if (transform.localScale.x < 0)
{
    if (Input.GetAxis("Horizontal") > 0)
        transform.localScale = Vector3.one;
}
else
{
    if (Input.GetAxis("Horizontal") < 0)
        transform.localScale = new Vector3(-1, 1, 1);
}
```

Об’єднаємо увесь цей код у метод Walk(), а метод будемо викликати з методу Update():

```
private void Walk()
{
    animator.SetFloat("speed", Mathf.Abs(Input.GetAxis("Horizontal")));

    Vector2 velocity = rigidbody.velocity;
    velocity.x = Input.GetAxis("Horizontal") * speed;
    rigidbody.velocity = velocity;

    if (transform.localScale.x < 0)
    {
        if (Input.GetAxis("Horizontal") > 0)
            transform.localScale = Vector3.one;
    }
    else
    {
        if (Input.GetAxis("Horizontal") < 0)
            transform.localScale = new Vector3(-1, 1, 1);
    }
}
```

Тепер маємо повну функціональність, щодо руху персонажу вліво-вправо.

Додаймо функціональність виконувати атаку у вигляді метода Attack(), який також будемо викликати з методу Update():

```
private void Attack()
{
    if (Input.GetButtonDown("Fire1"))
        animator.SetTrigger("attack");
}
```

Залишається додати функціональність для виконання стрибка.

Так само як і в 3D геймплеї, щоб персонаж стрибав, до нього треба прикладати силу за допомогою виклику AddForce. При цьому потрібно визначати, чи перебуває персонаж на землі.

Створимо булеву змінну onGround1 і надамо їй значення true. Також створимо і виведемо в інспектор змінну jumpForce – сила стрибка. Потрібно підібрати їй значення для вашого персонажа.

```
[SerializeField]
private float jumpForce = 100.0f;

bool onGround = true;
```

Додаймо функціональність виконувати стрибок у вигляді метода Jump(), який також будемо викликати з методу Update():

```
private void Jump()
{
    animator.SetBool("jump", false);

    if (Input.GetButtonDown("Jump") && onGround)
    {
        animator.SetBool("jump", onGround);
        rigidbody.AddForce(Vector2.up * jumpForce);
    }

}
```

Залишається написати код, який буде визначати перебуває персонаж на землі чи ні. Перед тим відредагуємо наш персонаж. Створимо на сцені пустий GameObject, назвемо його Knight і зробимо персонаж Player підпорядкованим цьому об’єкту.

Щоб знати, чи є під персонажем земля, ми перевірятимемо, чи є відповідний колайдер на лінії між двома точками. Однією буде центр персонажа. Щоб встановити іншу – створимо порожній GameObject, назвемо його GroundCheck, зробимо його дочірнім об'єктом персонажа Knight. Розмістимо цей об'єкт трохи нижче ніг персонажа.

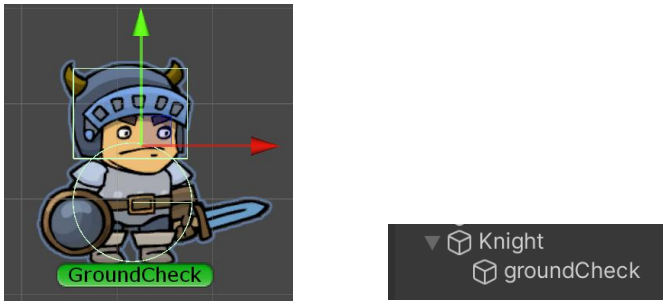
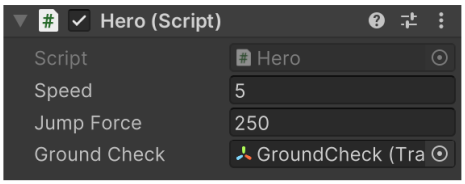


Рисунок 5.6 – Реструктуризація персонажу

Створимо змінну groundCheck типу Transform, і наддамо їй посилання цей об'єкт в інспекторі.

```
[SerializeField]
private Transform groundCheck;
```



Додаймо метод CheckGround(), який повертатиме булеве значення, залежно від того, є земля під персонажем чи ні:

```
private bool CheckGround()
{
    RaycastHit2D[] hits = Physics2D.LinecastAll(transform.position, groundCheck.position);

    for(int i=0; i<hits.Length; i++)
    {
        if (!GameObject.Equals(hits[i].collider.gameObject, gameObject))
        {
            return true;
        }
    }
    return false;
}
```

Тепер метод Update() має наступний вигляд:

```
void Update()
{
    onGround = CheckGround();

    Walk();
    Jump();
    Attack();

}
```

Якщо персонаж у стрибку притиснеться до стіни, він зупиниться і висітиме на місці доти, доки гравець не відпускає кнопку ходьби. Щоб цього не було, створіть новий Physics Material 2D, назвіть його ZeroFriction, задайте йому значення Friction (тертя) рівним нулю, і призначте цей матеріал як фізичний матеріал колайдера персонажа.

2. Завдання до лабораторної роботи №5

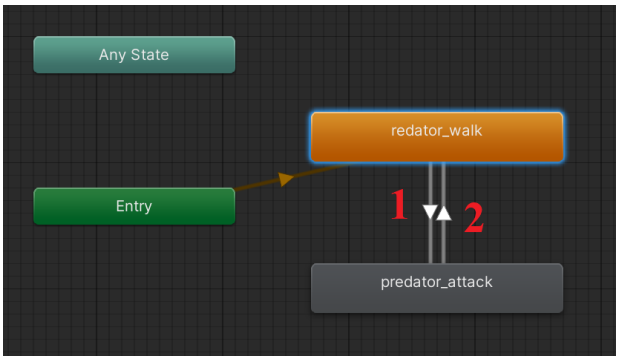
Створити 2D – гру, додати персонажу до сцени, створити анімацію для доданого персонажу та ворогів, додати можливість користувачеві управляти персонажем.

Практична робота 6. 2D-гра. Додавання ворогів

Мета роботи: познайомитися зі створенням ворогів для 2D-гри на Unity. Управління ворогами. Проведення атаки. Створення скриптів для ворогів та проведенню атаки. Життя головного персонажу та ворогів. Навчитися додавати ворогів та їх анімацію на сцену, проводити атаку та нанесення шкоди здоров'ю персонажу та ворогам.

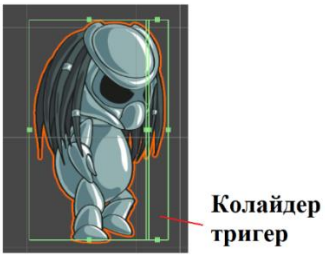
1. Теоретичний матеріал

Додаймо до анімації ворогів анімацію атаки. Налаштуємо контролер анімації ворога, додавши новий стан – атака. Перехід на атаку будемо виконувати зі стану walk. Створимо в Animator змінну типу тригер – Attack.



Для переходу 1 – Has Exit Time галочка знята, для 2 – поставлена, перехід 1 відбувається при активації тригера Attack, перехід 2 – просто після завершення анімації атаки.

Додаймо до ворога ще один колайдер у режимі тригера. Цей колайдер буде фіксувати зіткнення з головним персонажем та показ анімації атаки.



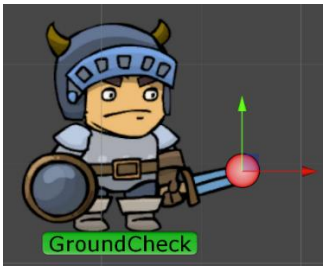
Додаймо до скрипту Enemy до методу OnTriggerEnter2D код, якщо в колайдер - тригер будуть потрапляти об'єкти і якщо це головний персонаж, то розпочати анімацію атаки.

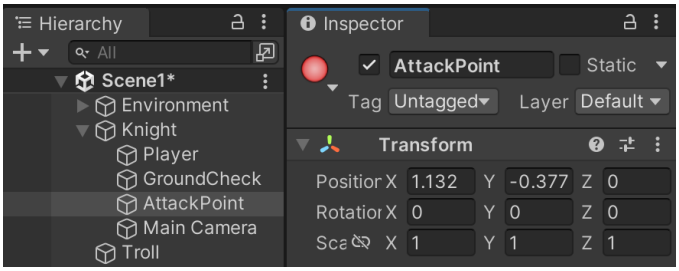
```
Knight knight =
    collider.gameObject.GetComponent();
if (knight != null) {
        animator.SetTrigger("Attack");
}
```

Щоб ворог атакував головного героя неодноразово, а доти, доки головний герой перед ним, змініть назву методу OnTriggerEnter2D на OnTriggerStay2D.

Тепер треба зробити так, щоб головний герой завдав шкоди, а не просто махав зброєю.

Додамо до об'єкта Knight порожній дочірній GameObject – це буде точка, в якій наноситься шкода. Встановимо для цього об'єкта іконку, перейменуємо AttackPoint і помістимо її на кінець зброї.





У скрипті Hero оголосимо змінну attackPoint типу Transform для цього об'єкта, виведемо її в інспектор і перетягнемо об'єкт у поле, щоб зберегти посилання на цей об'єкт.

Також оголосимо змінну attackRange типу float, і задаємо її значення інспекторі, наприклад, рівним 0.5. Це буде радіус навколо об'єкту AttackPoint, в якому будуть вражати ворожі колайдери.

Тепер треба навчити ворога та головного персонажу завдавати один одному ушкоджень. Спробуймо реалізувати це за допомогою інтерфейсів.

Інтерфейс – це список методів та властивостей, які мають бути реалізовані у класі, який успадковує цей інтерфейс (або реалізовує). Якщо ми знаємо, що якийсь клас реалізує певний інтерфейс, то ми можемо бути впевнені, що в цьому класі є набір методів та властивостей, які є обов'язковими для даного інтерфейсу.

Кожен клас може бути успадкований тільки від одного іншого класу, але може при цьому мати скільки завгодно успадкованих інтерфейсів

Створимо новий скрипт IDestructable, і видалимо з нього весь код і додаймо наступний код:

```
interface IDestructable
{
        float Health { get; set; }
        void Hit(float damage);
        void Die();
}
```

Тепер будь-який клас, який реалізує цей інтерфейс, повинен мати публічну властивість Health типу float, публічний метод Hit, який повертає void, приймає параметр типу float, і публічний метод Die, який так само повертає void, і не приймає параметрів.

Створимо клас Creature. Винесемо до нього загальні для персонажу та ворогів поля та методи та зробимо його класом, що реалізує інтерфейс IDestructable.

поля

```
protected Animator animator;
protected Rigidbody2D rigidbody;

[SerializeField]
protected float speed = 2.0f;

[SerializeField]
protected float damage;

[SerializeField]
protected float health = 100;

public float Health
{
    get { return health; }
    set { health = value; }
}
```

методи

```
private void Awake()
{
    animator = gameObject.GetComponent<Animator>();
    rigidbody = gameObject.GetComponent<Rigidbody2D>();
}

public void Hit(float damage)
{

}

public void Die()
{

}
```

Поля, які перенесені в цей клас видаляємо в скриптах персонажу та ворогів. Класи персонажів та ворогів робимо спадкоємцями класу Creature.

В метод Die() додаємо:

```
Destroy(gameObject);
```

В метод Hit() додаємо:

```
Health -= damage;
if (Health <= 0)
{
    Die();
}
```

2. Завдання до лабораторної роботи №6

Створити 2D – гру, додати ворогів до сцени, створити можливість проведення атаки та нанесення шкоди здоров'ю персонажу та ворогам

Практична робота 7. 2D гра. Додавання оточення

Мета роботи: познайомитися зі поняттям оточення для 2D-гри на Unity. Оточення сцени. Рух сходами. Створення підвісних мостів. Анімація для предметів. Створення скриптів для оточення. Навчитися створювати оточення.

1. Теоретичний матеріал

Tile – невелике зображення, яке використовується для конструювання рівнів у іграх. Tile можна перекласти як «мозаїка» або «черепиця».

Додаємо на сцену елементи сцени. Наприклад:

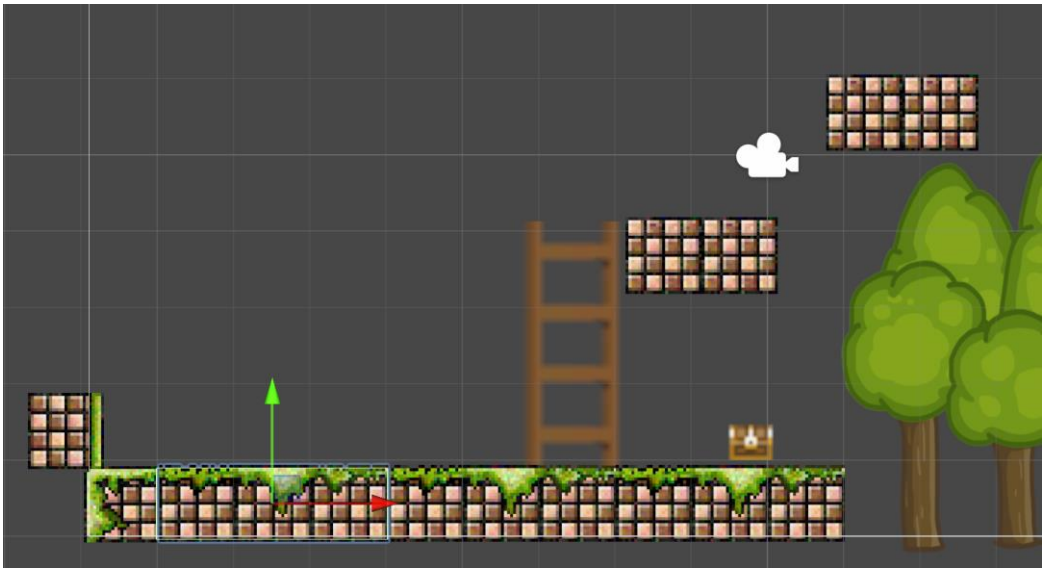


Рисунок 7.1 – Приклад сцени

Щоб елементи оточення щільно притискалися один до одного, натискаємо на клавіатурі кнопку **v** та тягнемо елемент до відповідного кута.

Бувають такі ситуації у грі, коли персонажу треба забратися кудись, наприклад сходами. Розглянемо яку функціональність потрібно додати персонажу, щоб він міг це робити.

Спочатку перетягнемо в сцену спрайт сходів. Якщо необхідно отримати сходи потрібної довжини, то створимо новий GameObject назвемо його Stair, додаймо як підлеглий об'єкт спрайт сходів. За потреби спрайт сходів розмножимо стільки разів, щоб отримати сходи потрібної довжини. Додаймо на об'єкт Stair BoxCollider2D. Налаштуємо розміри та положення колайдера, і поставимо галочку Is Trigger.

Потрібно, щоб, коли персонаж входить до зони тригера сходів, у нього включався режим “на сходах”, а коли виходить із зони – вимикався. Для цього є два стандартні методи MonoBehaviour: OnTriggerEnter2D та OnTriggerExit2D. Як аргумент у них автоматично передається посилання на колайдер, який увійшов/вийшов у/з зони тригера.

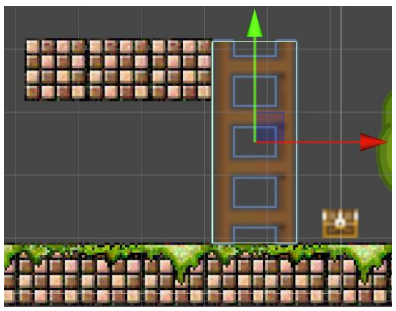


Рисунок 7.2 – Приклад використання сходів

У скрипт персонажа додаймо булеву змінну onStair, і створимо для неї public властивість:

```
private bool onStair = false;

public bool OnStair
{
    get
    {
        return onStair;
    }
    set
    {
        if (value == true)
            rigidbody.gravityScale = 0;
        else
            rigidbody.gravityScale = 1;

        onStair = value;
    }

}
```

Щоб персонаж не сповзав зі сходів під дією гравітації, у властивості OnStair додали код включення/вимкнення гравітації.

Створимо новий скрипт Stair та приєднаємо його до об’єкту Stair. Додаймо до цього скрипту методи OnTriggerEnter2D та OnTriggerExit2D з наступним кодом:

```
public class Stair : MonoBehaviour
{
    private void OnTriggerEnter2D(Collider2D collision)
    {
        Hero knight = collision.gameObject.GetComponent<Hero>();
        if (knight != null )
            knight.OnStair = true;
    }

    private void OnTriggerExit2D(Collider2D collision)
    {
        Hero knight = collision.gameObject.GetComponent<Hero>();
        if (knight != null)
            knight.OnStair = false;
    }
}
```

У скрипт персонажа додаймо змінну stairSpeed, яка визначатиме швидкість руху сходами.

Для того, щоб персонаж міг пересуватися вгору-вниз, знадобиться наступний код:

```
Vector2 velocity = rigidbody.velocity;
velocity.y = Input.GetAxis("Vertical") * stairSpeed;
rigidbody.velocity = velocity;
```

Створимо метод Stairs() в якому, якщо персонаж переткнув тригер сходів буде виконуватися код пересування вгору-вниз. Виклик цього методу зробимо в Update().

```
private void Stairs()
{
    if (OnStair)
    {
        Vector2 velocity = rigidbody.velocity;
        velocity.y = Input.GetAxis("Vertical") * stairSpeed;
        rigidbody.velocity = velocity;
    }
}
```

Додаймо до сцени підвісний міст.

Створимо на сцені пустий GameObject та дамо йому назву Bridge. Помістим всередину нього спрайт ділянки мосту.

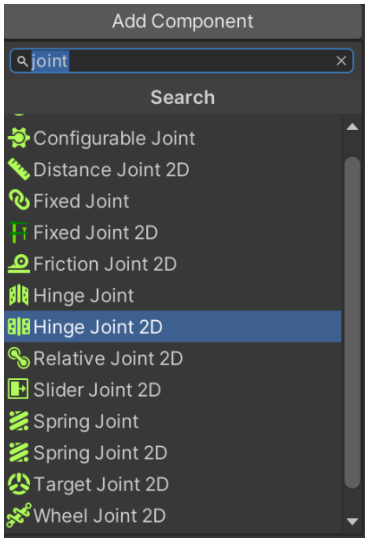
До спрайту ділянки мосту додати Rigidbody2D та BoxCollider2D.

2D джоінти являють собою звичайні фізичні об’єкти, яким можна придавати силу, переміщати, кидати, зтиснути тощо, в загальному – робити все те, що ми зазвичай робимо з фізичними об’єктами в Unity.

Особливість же джоінтів в тому, що з їх допомогою можна створити зв'язок з іншими фізичними об'єктами на сцені.

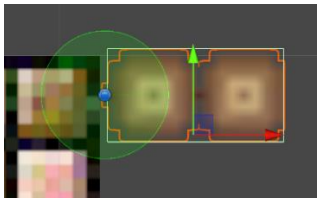
В цьому випадку всі дії, які ми здійснюємо над одним об’єктом, будуть також впливати на інші – пов’язані з ним об’єкти.

Всі компоненти джоінтів можна знайти у вкладці Component -> Physics 2D.

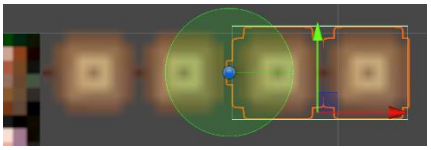


Hinge Joint 2D (або шарнірна петля) – це компонент, що контролюється фізикою Rigidbody, який дозволяє приєднати Sprite до точки в просторі, навколо якої цей об'єкт може обертатися. Обертання може бути пасивним (наприклад, у відповідь на зіткнення) або активним, коли двигун приводить його в рух.

Після додавання компоненту Hinge Joint 2D на спрайті з’явиться мітка у вигляді кола. Центр цієї мітки переносимо до місця де деталь буде примикати до іншої деталі.



Створимо копію спрайту ділянки мосту зі всіма компонентами. В копії в полі Connected Rigid Body додаймо посилання на Rigidbody2D попередньої ділянки мосту. Так продовжуємо поки не будемо мати необхідну довжину мосту.



В останній спрайт ділянки мосту додаймо ще один компонент Hinge Joint 2D та в полі Connected Rigid Body нічого не додаємо.

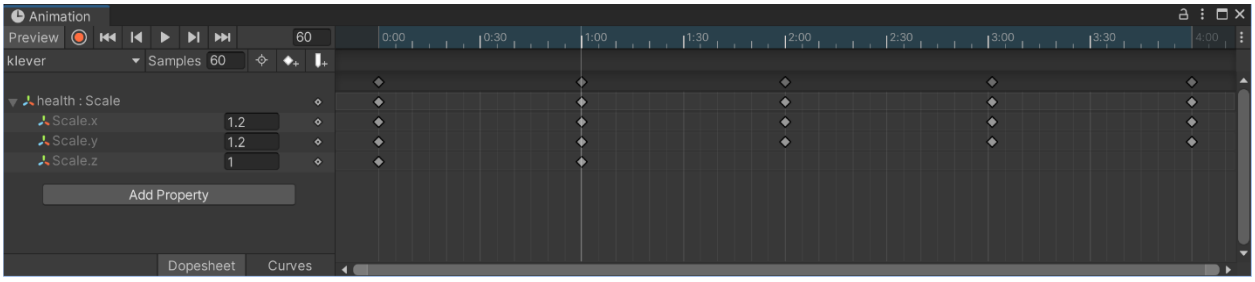
Пульсуюча анімація предмету.

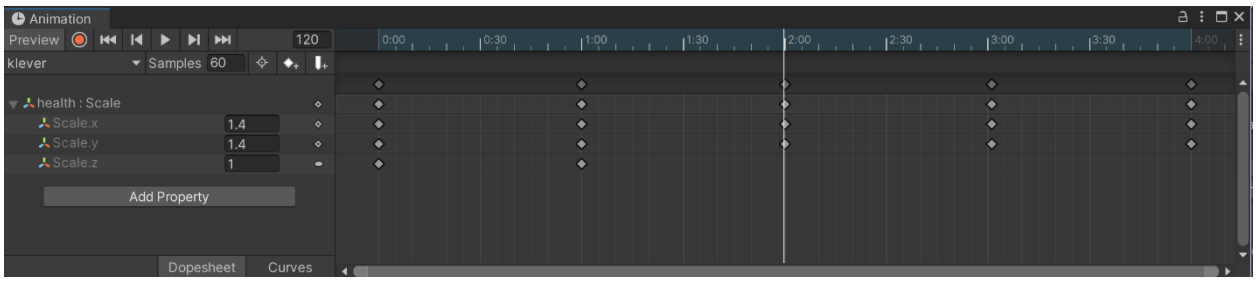
Обираємо об’єкт який бажаємо зробити пульсуючим. Додаємо до нього компонент Collider2D. Викликаємо вікно Animation для цього об’єкту.

Для створення нового кліпу натискаємо кнопку Create та вказуємо назву анімації, що створюється. Для запису анімації натискаємо кнопку 

У вікні Animation додаємо властивості що будуть змінюватися. Наприклад, місцеположення, масштаб тощо. Для цього натискаємо кнопку Add property.

Встановлюємо моменти часу на timeline. В каждому моменту часу змінюємо обрані властивості.





2. Завдання до лабораторної роботи №7

Створити 2D – гру, додати оточення. Додати можливість персонажу рухатися сходами та по мосту.

Практична робота 8. Формати для зберігання даних

Мета роботи: познайомитися зі формати для зберігання даних XML та JSON. Навчитися створювати файли даних формату XML та JSON.

1. Теоретичний матеріал

XML з'явився ще 1996 року і вперше був опублікований 1998 року. WWW Consortium (W3C) є розробником XML, і це стало Рекомендація W3C у 1998 році. XML виглядає дуже схожим на HTML, але XML є структурованішою мовою.

Документи у форматах HTML і XML містять дані, укладені в теги, але подібність між двома мовами закінчується. У форматі HTML теги *визначають оформлення даних* – розташування заголовків, початок абзацу тощо. У форматі XML теги *визначають структуру та зміст даних* – те, чим вони є.

Можна використовувати одну систему для генерації даних та позначки їх тегами у форматі XML, а потім обробляти ці дані в будь-яких інших системах незалежно від клієнтської платформи або операційної системи. Завдяки такій сумісності XML є основою однієї з найпопулярніших *технологій обміну даними*.

Правила XML дозволяють створювати будь-які теги, необхідні опису даних та його структури. Припустимо, що вам необхідно зберігати та спільно використовувати відомості про домашніх тварин. Для цього можна створити наступний XML-код:

```
<?xml version="1.0"?>
<CAT>
    <NAME>Izzy</NAME>
    <BREED>Siamese</BREED>
    <AGE>6</AGE>
    <OWNER>Colin Wilcox</OWNER>
</CAT>
```

Базовый синтаксис XML:

```
<?xml version = "1.0" encoding = "UTF-8" ?>
<root>
    <child>
        <subchild>.....</subchild>
    </child>
</root>
```

Для перевірки XML-файлів можна скористатися онлайн-додатком, наприклад https://www.freeformatter.com/xml-validator-xsd.html.

Структура JSON – вся інформація зберігається у вигляді *пар ключ-значення*.

Ключі в JSON це рядки, укладені в подвійні лапки. Ключі відіграють роль ідентифікаторів для значень і допомагають структурувати дані. Вони є унікальними в межах одного об’єкта JSON, що означає, що кожен ключ в об’єкті має бути унікальним.

Значення в JSON можуть бути різними типами даних, включно з рядками (текст), числами, логічними значеннями (true або false), null (порожнє значення), масивами або вкладеними об’єктами. Значення можуть бути укладені в подвійні лапки (для рядків), без лапок (для чисел, логічних значень і null) або в квадратні дужки чи фігурні дужки (для масивів і об’єктів відповідно).

Приклад JSON-файлу:

```
{
     "name": "John",       // Строкове значення
     "age": 30,            // Числове значення
     "isStudent": true     // Логічне значення
     "address": null       // значення null (пусто)
```

```
}
```

де name, age, isStudent та address – це ключі, а їхні значення відповідно «John», 30 і true.

Масиви в JSON являють собою впорядковані списки елементів. Вони укладаються в квадратні дужки і можуть містити значення різних типів даних, включно з іншими масивами та об'єктами.

Наприклад:

```
{
  "fruits": ["apple", "banana", "orange"]
}
```

де fruits – це ключ, а його значення – масив з елементами apple, banana і orange.

Об'єкти являють собою невпорядковані набори ключів і значень. Вони містяться у фігурних дужках і дають змогу структурувати дані складніше, вкладаючи об'єкти один в одного.

Наприклад:

```
{
      "person":
      {
            "name": "Alice",
            "age": 25
      }
}
```

person – ключ, а його значення являє собою вкладений об'єкт із ключами name і age.

Серіалізація і десеріалізація – це два важливі процеси, пов'язані з JSON, які дають змогу перетворити дані у формат JSON і назад.

Серіалізація JSON – це процес перетворення даних з об'єктів і структур даних у рядок JSON. Коли ми серіалізуємо дані, ми робимо їх придатними для передавання мережею або збереження на диску в текстовому форматі.

Десеріалізація JSON – це зворотний процес, під час якого рядок JSON перетворюється назад в об'єкти і структури даних. Коли ми отримуємо дані у форматі JSON з мережі або файлу, нам потрібно перетворити їх назад на об'єкти, щоб використовувати їх у програмі.

2. Завдання до лабораторної роботи №8

Створіть файл даних для створеної вами гри в форматі XML. Перевірьте правільність його створення. Створіть файл даних в форматі JSON.

Практична робота 9. Криптографічний захист

Мета роботи: познайомитися зі стандартами кібербезпеки та методами криптографії. Навчитися шифрувати данні.

1. Теоретичний матеріал

Стандарт ISO/IEC 27001 — міжнародний стандарт, який встановлює певні вимоги до створення, впровадження, підтримки та вдосконалення системи управління інформаційною безпекою (ISMS) підприємства, а також до поточної оцінки інформаційних ризиків та управління ними.

Сертифікація ISO 27001 значною мірою актуальна для великих організацій, але з розвитком технологій вона стає все більш популярним рішенням для малих і середніх компаній.

ISO 27001 описує структуру системи управління інформаційною безпекою (СУІБ) для компаній незалежно від організаційної структури, розміру або орієнтації. Стрижнем тут є управління ризиками.

Мінливі кіберзагрози постійно використовують нові потенційні вразливості в компаніях з метою атаки та компрометації інформаційних потоків, а отже, і бізнес-процесів. Ризики, що виникають внаслідок цього механізму для трьох основних цілей захисту інформаційної безпеки конфіденційності, цілісності та доступності повинні бути ідентифіковані та керовані.

Стандарт ISO 27001 складається з двох частин:

- Основна частина – різновид стратегії, основна частина стандарту;
- Додаток А – перелік із 114 потенційно застосовних засобів контролю.

Дуже суттєвою зміною є доповнення контексту організації в п. 4.4 вимогою щодо визначення необхідних процесів та їх взаємодій в рамках СУІБ, які потрібні для її впровадження та підтримки в робочому стані.

Ця чітка вимога приводить стандарт ISO/IEC 27001:2022 у відповідність до підходу найкращих практик інших систем управління відповідно до HS (HLS). Система управління інформаційною безпекою повинна ґрунтуватися на встановлених процесах, що відслідковуються, та їх взаємодії. На основі цих процесів розробляються та адаптуються засоби управління інформаційною безпекою, зазначені в Додатку А.

Міжнародний стандарт ISO/IEC 17799 Інформаційна технологія - Методи захисту - Практичний посібник для управління інформаційною безпекою. З 2007 року запропоновано включити нове видання стандарту ISO/IEC 17799 до нової схеми нумерації, як ISO/IEC 27002.

Стандарт визначає загальну організацію, класифікацію даних, систему доступу, напрями планування, відповідальність співробітників, використання оцінки ризику тощо у контексті інформаційної безпеки. Зарекомендував себе як практичний стандартний посібник у багатьох відділах IT та безпеки як визнаний інструмент.

У процесі впровадження стандарту створюється так звана система управління інформаційною безпекою, мета якої – скорочення матеріальних втрат, пов'язаних з порушенням інформаційної безпеки.

ISO 17799 – це модель системи управління, і в цьому сенсі не є технічним стандартом. Наприклад, він не наказує використання якихось певних алгоритмів шифрування. Єдиний пункт стандарту, що безпосередньо регламентує шифрування, містить буквально таку вимогу: "Особливо важлива інформація має бути зашифрована". Що вважати важливою інформацією та як робити шифрування, підприємство має вирішити самостійно, ґрунтуючись на загальних засадах застосування стандарту.

ISO 17799 містить сто з лишком елементів управління інформаційною безпекою, розподілених за кількома групами:

- політика у сфері безпеки. Завдання елементу – забезпечити чітке управління та підтримку політики в галузі інформаційної безпеки з боку керівництва підприємства;
- організація системи безпеки. Завдання елементу – створити організаційну структуру, яка впроваджуватиме і забезпечуватиме працездатність системи інформаційної безпеки в організації;
- класифікація ресурсів та управління. Завдання елементу – підтримувати адекватну інформаційну безпеку організації шляхом покладання персональної відповідальності, а також класифікації інформаційних ресурсів за потребою та пріоритетом захисту;
- безпека та персонал. Завдання елементу – зменшити ризик людських помилок, розкрадань та неправильного використання обладнання, у тому числі шляхом ефективного навчання та впровадження механізму відстеження інцидентів;
- фізична та зовнішня безпека. Завдання елементу – запобігти несанкціонованому доступу, пошкодженню та порушенню роботи інформаційної системи організації;
- менеджмент комп'ютерів та мереж. Завдання елементу – забезпечити безпечне функціонування комп'ютерів та мереж;
- керування доступом до системи. Завдання елемента – керувати доступом до ділової інформації, запобігати несанкціонованому доступу та виявляти несанкціоновану діяльність;
- розробка та обслуговування системи. Завдання елемента – забезпечити виконання вимог безпеки під час створення чи розвитку інформаційної системи організації, підтримувати безпеку додатків та даних;
- забезпечення безперервності роботи. Завдання елемента – підготувати план дій у разі надзвичайних обставин задля забезпечення безперервності роботи організації;
- відповідність законодавству. Завдання елемента – забезпечити виконання вимог відповідного цивільного та кримінального законодавства, включаючи закони про авторські права та захист даних.

Така структура дозволяє вибрати ті засоби управління, які стосуються конкретної організації чи сфери відповідальності всередині організації.

Виділено також десять так званих ключових елементів управління, які є фундаментальними:

- політика з інформаційної безпеки;
- розподіл відповідальності за інформаційну безпеку;
- освіта та тренінг з інформаційної безпеки;
- звітність щодо інцидентів з безпекою;
- захист від вірусів;
- забезпечення безперервності роботи;
- контроль копіювання програмного забезпечення, що ліцензується;
- захист архівної документації організації;
- захист персональних даних;
- виконання політики щодо інформаційної безпеки.

З вищевикладеного видно, що поряд з елементами управління для комп'ютерів та комп'ютерних мереж стандарт приділяє велику увагу питанням розробки політики безпеки, роботи з персоналом (прийом на роботу, навчання, звільнення з роботи), забезпечення безперервності виробничого процесу, юридичних вимог.

1 грудня 1999 року видано стандарт iso/iec 15408 "Критерії оцінки безпеки інформаційних технологій".

Цей міжнародний стандарт став підсумком майже десятирічної роботи фахівців кількох країн, він увібрав у себе досвід документів національного та міжнаціонального масштабу, що існували на той час.

Базовими документами, які лягли в основу стандарту iso/iec 15408, є:

- Помаранчева книга (TCSEC) 1985р.;
- Європейські критерії (ITSEC) 1991р.;
- Канадські критерії 1993р.;
- Федеральні критерії США 1993р.

Стандарт ISO/IEC 15408 визначає критерії безпеки інформаційних технологій. У ньому не наводиться список вимог безпеки, але положення стандарту дозволяють сформулювати цілі безпеки, спрямовані на забезпечення протистояння загрозам та виконання політики безпеки.

Стандарт описує інфраструктуру, в якій користувачі системи можуть сформулювати вимоги, а експерти з безпеки визначити, чи має продукт заявлені властивості.

З історичних причин цей стандарт часто називають "Загальними критеріями" (ЗК).

«Загальні критерії» є стандартом, що визначає інструменти оцінки безпеки ІС та порядок їх використання. На відміну від "Помаранчевої книги", ЗК не містять певних «класів безпеки».

Загальні критерії отримали широке визнання у світі. У 1998 році урядовими організаціями Канади, Франції, Німеччини, Великобританії і США було підписано угоду про взаємне визнання оцінок (The International Mutual Recognition Arrangement – MRA), отриманих на основі Загальних критеріїв.

Стандарт складається з 5 частин.

ДСТУ ISO / IEC 15408-1 встановлює загальний підхід до формування вимог до оцінки безпеки (функціональні та довіри), основні конструкції (профіль захисту, завдання з безпеки) подання вимог безпеки в інтересах споживачів, розробників і оцінювачів продуктів і систем ІТ.

Вимоги безпеки об’єкта оцінки (ОО) за методологією Загальних критеріїв визначаються виходячи з цілей безпеки, які, у свою чергу, грунтуються на аналізі призначення об’єкта оцінки і умов середовища його використання (загроз, припущень, політики безпеки).

ДСТУ ISO / IEC 15408-2 містить універсальний систематизований каталог функціональних вимог безпеки і передбачає можливість їх деталізації і розширення за певними правилами.

ДСТУ ISO / IEC 15408-3 включає в себе систематизований каталог вимог довіри, визначають заходи, які повинні бути прийняті на всіх етапах життєвого циклу продукту або системи ІТ для забезпечення впевненості в тому, що вони задовольняють пред’явленим до них функціональним вимогам.

Тут же містяться оціночні рівні довіри, що визначають шкалу вимог, які дозволяють з зростаючої ступенем повноти і строгості оцінити проектну, тестову та експлуатаційну документацію, правильність реалізації функцій безпеки об’єкта оцінки, уразливості продукту або системи ІТ, стійкість механізмів захисту і зробити висновок про рівень довіри до безпеки об’єкта оцінки.

ISO / IEC 15408-4 – попереднє визначення профілі захисту.

ISO / IEC 15408-5 – процедури реєстрації профілів захисту.

Знайомство із середовищем Cryptool 2

CrypTool 2 – безкоштовне програмне забезпечення з відкритим вихідним кодом, що реалізує концепцію візуального програмування та 28 виконання каскадів криптографічних процедур. CrypTool 2 є однією із складових великого проекту CrypTool, призначеного в першу чергу для електронного навчання криптографії та криптоаналізу. Завантажими можна за посиланням https://www.cryptool.org/en/ct2/downloads , актуальна версія Stable

Version → CrypTool 2.1 (Stable Build 9778.2). Програмний засіб CrypTool 2 (рис. 6.2) на даний час доступний німецькою та англійською мовами, має інтуїтивно зрозумілий сучасний графічний інтерфейс та зручне меню, за допомогою якого користувач у робочій області програми може перетворювати повідомлення з використанням найвідоміших криптографічних алгоритмів.

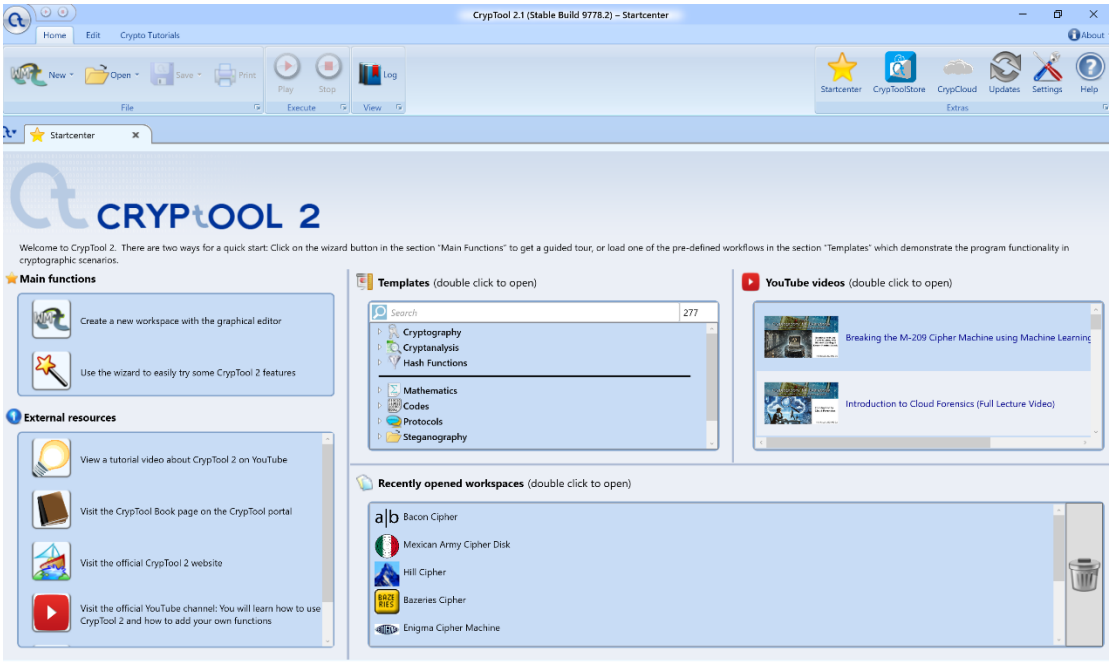


Рисунок 9.1 – Вікно CrypTool 2

Розділ «Templates» містить як готові шаблони проектів, що реалізують криптографічні та криптоаналітичні алгоритми, математичні та інші функції, протоколи тощо, так і набір інструментів, котрі можуть бути використані для модифікації готових або створення стеганографічних способів перетворення даних, тобто такі, при яких повідомлення не шифрується, а приховується сам факт його передачі чи існування.

Для використання готового шаблону у розділі «Templates» із меню, необхідно обрати потрібний алгоритм. Після чого відкриється нова вкладка, що складатиметься із окремих модульних компонентів, пов’язаних між собою.

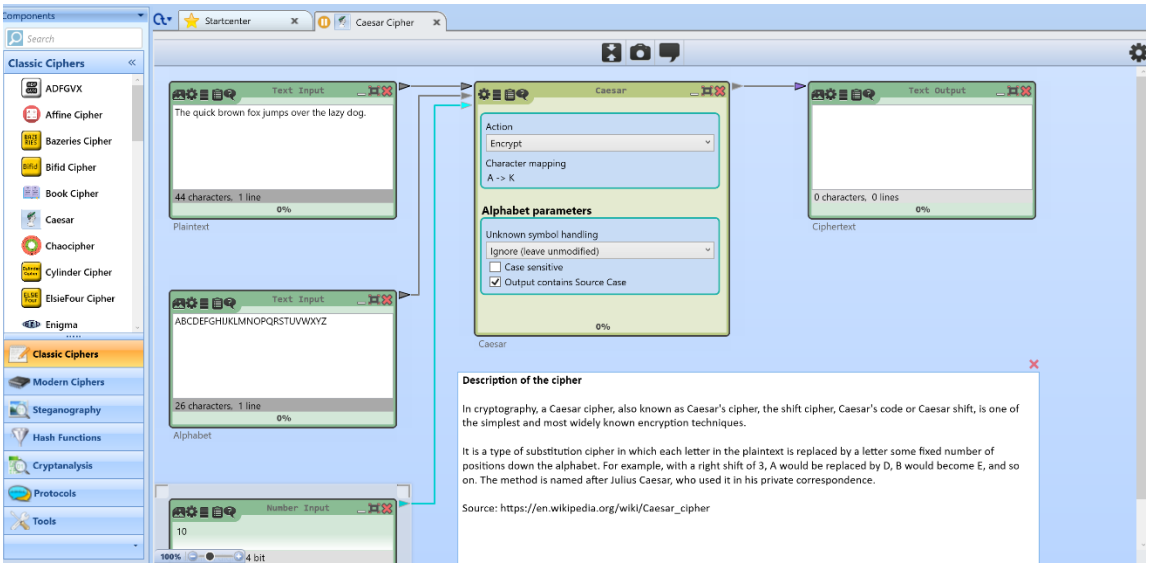


Рисунок 9.2 – Нова вкладка при виборі шифру Цезаря

Вони мають властивості подібні до діалогового вікна операційної системи Windows. Активізація компоненту відбувається шляхом натискання по ньому лівою клавішею миші. Кожен компонент у лівому верхньому кутку містить меню, що дає змогу налаштування дій, відкриття довідки тощо.

Панель модульних компонентів «Components», що розміщена ліворуч робочого вікна, дає змогу додавати до проекту нові складові, таким чином удосконалюючи його.

2. Завдання до лабораторної роботи №9

Ознайомитися зі стандартами кібербезпеки. Написати реферат на тему, як інформація зі стандартів може застосовуватися в play-технологіях.

Виконати зашифрування блоку даних відкритого тексту за допомогою алгоритму AES на основі навчальної програми CrypTool 2 (Templates → Cryptography → Modern → Symmetric → AES Visualization). Сформувати раундові ключі для зашифрування даних. У звіті описати зі скріншотами кроки алгоритму генерації ключів.

Рекомендована література

Основна

1. Lanzinger F. 2D Game Development with Unity: CRC Press, 2020. – 444 p. URL: https://www.google.com.ua/books/edition/2D_Game_Development_with_Unity/hjwDEAAAQBAJ?hl=ru&gbpv=0

2. Felicia P. The Ultimate Guide to 2D Games with Unity: Amazon Digital Services LLC – Kdp, 2020. – 488 p. URL: https://www.google.com.ua/books/edition/The_Ultimate_Guide_to_2D_games_with_Unit/9AYBEAAAQBAJ?hl=ru&gbpv=0

3. Halpern J. Developing 2D Games with Unity. Independent Game Programming with C#: Apress, 2018. – 383 p. URL: https://www.google.com.ua/books/edition/Developing_2D_Games_with_Unity/loF8DwAAQBAJ?hl=ru&gbpv=0

4. Murray Jeff W. 2D Unity. Your First Game from Start to Finish: No Starch Press, 2016. – 312 p. URL: https://www.google.com.ua/books/edition/2D_Unity/0-8cswEACAAJ?hl=ru

Допоміжна

5. Sapio F. Unity 2D Game Development Blueprints / F. Sapio, A. Saher : Packt Publishing, Limited, 2016. – 252 p. URL: https://www.google.com.ua/books/edition/Unity_2D_Game_Development_Blueprints/KMzpjwEACAAJ?hl=ru

6. Nakov S. Programming Basics with C# / S. Nakov, Nakov's Team: SoftUni, 2019. – 408 p. URL: https://www.google.com.ua/books/edition/Programming_Basics_with_C/vzKyDwAAQBAJ?hl=ru&gbpv=0

7. Miller R. C# for Artists. The Art, Philosophy, and Science of Object-Oriented Programming: Pulp Free Press, 2008. – 750 p. URL: https://www.google.com.ua/books/edition/C_for_Artists/K0ICo6r29W4C?hl=ru&gbpv=0

8. Zoubir Z. Mammeri. Cryptography: Algorithms, Protocols, and Standards for Computer Security : John Wiley & Sons, 2024. – 624 p. URL: https://www.google.com.ua/books/edition/Cryptography/ecb0EAAAQBAJ?hl=ru&gbpv=1&dq=Cryptography&printsec=frontcover

Інформаційні ресурси

9. Офіційний сайт Unity [Електронний ресурс] URL: https://unity.com/ (Дата звернення: 01.09.2024)

10. Офіційний сайт Visual Studio Community [Електронний ресурс] URL: https://visualstudio.microsoft.com/ru/vs/community/ (Дата звернення: 01.09.2024)

11. Офіційний сайт Scratch [Електронний ресурс] URL: https://scratch.mit.edu/ (Дата звернення: 01.09.2024)

12. ISO/IEC 27001: What’s new in IT security? [Електронний ресурс] URL: https://www.iso.org/contents/news/2022/10/new-iso-iec-27001.html (Дата звернення: 01.09.2024)

13. Про прийняття національних стандартів, зміни до національного стандарту та скасування національних стандартів [Електронний ресурс] URL: https://zakon.rada.gov.ua/rada/show/v0210774-23#Text (Дата звернення: 01.09.2024)

14. Офіційний сайт програми CrypTool 2 [Електронний ресурс] URL: https://www.cryptool.org/en/ct2/downloads/ (Дата звернення: 01.09.2024)