Міністерство освіти і науки України
Міжнародний гуманітарний університет
Кафедра комп’ютерної інженерії та інноваційних технологій

Лебедєва О.Ю.

РОЗРОБКА ІГРОВИХ ЗАСТОСУНКІВ

Конспект лекцій
для здобувачів вищої освіти,
які навчаються за спеціальностями
галузі «Інформаційні технології»

Одеса 2025

Затверджено Вченою Радою Міжнародного гуманітарного університету.
Протокол № 5 від 20 січня 2025 р.

Лебедєва О.Ю Розробка ігрових застосунків. Конспект лекцій для здобувачів вищої освіти, які навчаються за спеціальностями галузі «Інформаційні технології» [Електронне видання]. Кафедра комп'ютерної інженерії та інноваційних технологій Міжнародного гуманітарного університету. Одеса, 2025. 116 с.

Дисципліна «Розробка ігрових застосунків» спрямована на формування у здобувачів вищої освіти спеціальності Інженерія програмного забезпечення компетентностей, пов'язаних із проєктуванням та розробкою програмних застосунків ігрового призначення з використанням сучасних інструментальних середовищ створення інтерактивних систем. Вона орієнтована на формування здатності застосовувати технології програмної інженерії під час створення ігрових застосунків, розробляти програмні модулі ігрової логіки, організовувати взаємодію програмних компонентів та реалізовувати інтерактивні програмні системи.

ЗМІСТ

Лекція 1. Введення в play-технології

1. Структура типової ігрової команди
2. Типи користувачів
3. Жанри комп'ютерних ігор
4. Монетизація в іграх
5. Життєвий цикл при розробці ігор

1. Структура типової ігрової команди

Перш ніж поринути у вивчення структури типового ігрового двигуна, коротко розглянемо структуру типової команди розробки гри.

Ігрові студії зазвичай формуються із фахівців п'яти основних напрямів:

- Розробників;
- Художників;
- Геймдизайнерів;
- Продюсерів;
- Іншого управлінського персоналу та персоналу підтримки.

Розробники проектують та створюють програмне забезпечення, що змушує гру та інструменти працювати.

Вони часто поділяються на дві групи:

- розробники середовища виконання (працюють над двигуном та грою як такою);
- розробники інструментів (створюють офлайнові інструменти, що дозволяють решті розробників діяти ефективніше).

Художники створюють весь візуальний та аудіоконтент гри. Для роботи потрібні художники та дизайнери усіх напрямків.

Творці концепт-артів створюють скетчі та замальовки, що дозволяють команді побачити результат розробки гри. Вони починають раніше за інших – на стадії розробки концепції, але зазвичай забезпечують візуальну спрямованість проекту протягом усього життєвого циклу.

Розробники 3D-моделей створюють тривимірну геометрію для всього у віртуальному світі гри. Сюди зазвичай входять:

- розробники моделей переднього плану;
- розробники заднього плану.

Ті, хто працює над переднім планом, створюють об'єкти, персонажів, транспорт, зброю – все, що наповнює ігровий світ.

Розробники моделей заднього плану створюють геометрію статичного тла – рослинність, будівлі, мости тощо.

Художники з текстур створюють двовимірні зображення, звані текстурами, які накладаються на поверхні 3D-моделей для забезпечення деталізації та реалізму.

Фахівці з освітлення створюють в ігровому світі джерела світла, як статичні, так і динамічні, працюють з кольором, інтенсивністю та напрямом освітлення, щоб забезпечити максимальну художність та емоційну дію кожної сцени.

Аніматори надають рухомим персонажам та об'єктам гри динамічність. Ігровий аніматор повинен мати унікальний набір навичок, щоб виконати анімацію, яка бездоганно узгоджуватиметься з технологічними особливостями ігрового двигуна.

Звукорежисери працюють разом з розробниками, щоб створити та змішати звукові ефекти та музику у грі. Актори озвучки наділяють персонажів голосами у більшості ігор.

Робота геймдизайнера полягає в розробці інтерактивної складової взаємодії користувача з грою, званої геймплеєм. Різні дизайнери працюють на різних рівнях деталізації.

Деякі (як правило, старші) геймдизайнери діють на макрорівні, визначаючи сюжет гри, загальну послідовність розділів або рівнів, а також основні цілі та завдання гравця.

Інші дизайнери працюють над певними рівнями або географічними областями всередині віртуального світу гри, створюючи шари геометрії статичного фону, визначаючи, коли і звідки з'являться вороги, розміщуючи такі запаси, як зброя та відновлюючи здоров'я аптечки, розробляючи елементи внутрішньо ігрових загадок тощо.

Деякі ігрові команди наймають одного чи кількох ігрових письменників. Робота цього фахівця може змінюватись від співпраці зі старшими геймдизайнерами з метою конструювання сюжету всієї гри до написання окремих ліній діалогів.

Деякі старші дизайнери відіграють роль менеджерів. У багатьох ігрових командах є геймдиректор, робота якого полягає у спостереженні за всіма аспектами геймдизайну, допомоги у керуванні термінами та забезпеченні того, щоб діяльність усіх дизайнерів була впорядкована протягом усієї роботи над продуктом. Старші дизайнери іноді виростають у продюсерів.

2. Типи користувачів

Для кого ми робимо ігри?

Багато розробників роблять ігри для себе, щоб отримати досвід або тому, що їм самим подобається конкретний жанр.

Цей підхід має право на існування, проте більшість гейм-дизайнерів все ж таки хочуть, щоб з їхнім дітищем познайомилося якнайбільше людей і, що важливо, щоб гравці гідно його оцінили.

Буває, що смаки гейм-дизайнера та аудиторії не збігаються.

Чим більше ви знатимете про переваги вашої аудиторії, тим краще зможете передбачити, які особливості вашої гри можуть принести їм задоволення. Більше того, у вас буде розуміння, де взагалі цю аудиторію шукати, через які канали просувати продукт і так далі.

Сьогодні конкуренція у промисловості більш ніж серйозна. На день виходить кілька тисяч ігор, і яку б чудову гру ви не створили, доведеться докласти зусиль, щоб про неї взагалі хоч хтось дізнався.

Найвідомішою класифікацією гравців за відданим їм типу задоволення є класифікація геймдизайнера і професора Університету Ессекса Річарда Бартла. Він ділить людей, які грають у розраховані на багато користувачів ігри, на чотири групи: ачівери, кілери, соціальники і дослідники. Багато законів взаємодії, помічені Бартлом, можна застосовувати і для однокористувальних ігор.

Головне задоволення Ачіверів (ще їх називають «кар'єристами») – це виклик. Вони виконують різні ігрові цілі в гонитві за радістю досягнення певних результатів, тобто для них важливі саме ігрові дії, що виконуються, і наступні за ними досягнення. Зазвичай такі гравці прагнуть збирання різних ресурсів, умінь, артефактів, нагород. Загалом їм важливий внутрішньоігровий прогрес як можливість досягти успіху. Практично всі ігрові платформи мають систему досягнень, і багато гравців з радістю витрачають час на гру, щоб отримати колекційну картку або значок.

Кілерам потрібна перемога. Цей тип гравців шукає змагань, насамперед з іншими людьми, щоб показати, що вони найкращі. Головне для кілерів – так чи інакше зіставляти себе з іншими гравцями, відчувати свою значущість та перевагу, причому домагатися цього швидко – рутини кілери не люблять.

Соціальники найбільше цінують взаємини з іншими гравцями, почуття товариства та власну популярність. Часто такі люди, знаходячи щось цікаве у вашій грі, наводять своїх друзів та знайомих. Саме вони найчастіше та найактивніше користуються внутрішньоігровим чатом, беруть участь у житті форумів, залишають відгуки, стають лідерами думок.

Дослідники воліють взаємодію з ігровим світом, прагнучи відкрити його у всій повноті. Бої та активні дії для них відходять на другий план. Дослідники цінують різноманіття ігрового контенту, сюжет, різні ігрові механіки, що дають змогу досконало вивчити можливості гри; рейтинги та змагання їм нецікаві.

За чотирма психотипами Бартла зібрано багато корисної інформації (метрики з повернення в гру, відсоток гравців, що платять тощо), тому для визначення своєї аудиторії гейм-дизайнери найчастіше користуються саме цією спрощеною системою.

3. Жанри комп'ютерних ігор

У книзі Рефа Костера «Розробка ігор і теорія розваг» *гра* визначена як інтерактивний досвід гравця, реалізований у вигляді послідовності шаблонних дій, які він або вона освоює і в кінцевому рахунку виконує. Костер стверджує, що дії освоєння та виконання у грі є головними в тому, що ми називаємо «розвагою».

Термін «*ігровий движок*» – це програмне забезпечення, яке розширюється і може застосовуватися як основа для безлічі різних ігор без значних модифікацій. Більшість ігрових двигунів розроблені та ретельно налаштовані для запуску конкретної гри на конкретній апаратній платформі.

Unity развивается с 2005 года. На Unity разработаны тысячи приложений и игр разных жанров на более чем 25 платформах. Изначально Unity создавался для компьютеров Mac, в следующих версиях добавлялись новые платформы: Windows, IPhone, Android, Xbox, Playstation и другие. Для написания скриптов движок использует язык программирования C#. В редакторе Unity простой интерфейс, что позволяет легко производить отладку игры. Главными преимуществами Unity считают кросс-платформенность и наличие визуальной среды разработки.

Unreal Engine випустила у 1998 році компанія Epic Games. Це один з перших універсальних двигунів, що поєднують фізику, рендеринг, штучний інтелект і готове середовище розробки. Спочатку він створювався для роботи над шутерами від першої особи, проте наступні версії застосовуються для ігор різних жанрів. Unreal Engine завжди приділяв багато уваги вражаючому малюнку, ним користуються і в кінематографі; сьогодні він надає просунутий інструментарій для створення фотореалістичної 3D-графіки у реальному часі.

Жанри комп'ютерних ігор, з одного боку, мають досить чіткі межі, але з іншого – нерідко досить складно класифікувати ту чи іншу гру в межах одного жанру. У деяких іграх жанри переплітаються, деякі ігри створюють власні жанри.

Platformers (платформні ігри або платформери) це ігри, де герой подорожує ігровим світом, переходячи з платформи на платформу – стрибаючи над прірвами, перепливаючи річки, дертися на стіни і попутно борючись з лиходіями. Спочатку платформери – це двовимірні ігри з видом збоку.

Action – це ігровий жанр, який цікавий тим, що гравець під час ігрового процесу має постійно щось робити. Найбільш популярне заняття у таких іграх – це знищення ворогів. А найпопулярнішим видом екшенів стали так звані шутери.

Типовий сценарій *шутерів* – ви керуєте будь-яким механізмом (або героєм), блукаєте ігровим світом (або переміщаєтеся в обмежених межах) і знищуєте полчища ворогів.

FPS або *First Person Shooters* – тобто "стрілялки від першої особи" - це ігри, які вже багато років перебувають у стані постійного вдосконалення та розвитку. Як правило, у такій грі популярний вигляд з очей героя – ви бачите ігровий світ так, як бачить його герой. Тривимірний світ у таких іграх дуже важливий – тому перші помітні FPS з'явилися на початку 1990-х з розвитком тривимірної ігрової графіки.

Ігри стилю *Survival Horror* ("жахи", "ігри на виживання") – це ігри, як правило, що мають властивості FPS-ігор, проте крім звичайних для битв елементів, містять елементи

жанру жахів. Гравця оточує похмура атмосфера, монстри, він потрапляє у важкі колотнечі, які повинні тримати його у постійній напрузі

Ігри в стилі *Fighting*, бійки – це ігри, де ви вступаєте в бій з віртуальним противником, який або управляється комп'ютером, або - людиною. Цей жанр виник у середині 1980-х років і популярний досі.

Ще один стиль ігор – це суміш файтингу та платформерів – в англійському варіанті вони називаються *Beat'em up*. Як правило, ціль такої гри – пройти гру, знищивши всіх ворогів. Герой гри зазвичай користується якоюсь зброєю, проте система ударів і рухів у таких іграх зазвичай проста – кілька базових ударів, плюс як мінімум один "суперудар", який дозволить "розкидати" цілу юрбу ворогів. Ці ігри були особливо популярні у 1990-х роках.

RPG – Role Playing Game (рольові ігри) – ігри, де ви повинні відігравати роль будь-якого персонажа. Перші рольові ігри виникли завдяки автоматизації класичних ігор, у які грали без участі комп'ютерів – популярність таких ігор зумовила популярність комп'ютерних RPG. Зазвичай події у таких іграх розвиваються хід за ходом, у сучасних RPG є епізоди (бої, наприклад), які передбачають гру в реальному часі.

Puzzle (головоломки) – це ігри, які зазвичай не вимагають серйозних графічних можливостей системи – їхня головна цінність в ідеї гри.

RTS – Real Time Strategy (стратегії реального часу) – це ігри, де ви, як правило, керуєте будь-якою армією, причому події розвиваються в реальному часі.

Massively Multiplayer Online Games (MMO) – або онлайнові ігрові світи – це ігри, в яких люди буквально живуть день і ніч, підключившись до ігрового серверу через Інтернет. Існує кілька різновидів цих ігор. Перша скорочено називається MMORPG – тобто онлайнова рольова гра, а друга – MMOFPS – онлайнова стрілялка. Перші ігри цього стилю з'явилися торік у 1970-х роках – тоді це були текстові ігри. Пізніше, на початку 1990-х, вони набули графічного інтерфейсу і з того часу розвиваються, займаючи уми все більшої кількості гравців.

Simulator (симулятор) – це гра, покликана як найточніше (чи хоча б правдоподібно) щось імітувати. Популярність симуляторів полягає в тому, що вони дозволяють «не виходячи з дому» відчути себе за штурвалом літака або вертольота, покерувати гоночною машиною або посмикати за важелі танка, стати директором транспортної компанії, політати в космосі, повоювати і таке інше. Розвиток цих ігор полягає у підвищенні якості графічної складової гри, у покращенні імітації реальних подій.

Клас *Adventure* включає ігри пригодницької тематики. Пік популярності цих ігор припав на початок 1990-х років. Під час гри гравець розгадує загадки, вирішує головоломки, спілкується з іншими персонажами. Гра чимось нагадує RPG, проте тут персонаж зазвичай не піддається розвитку – всю гру можна представити у вигляді якогось наперед визначеного плану дій, який ви проходите за допомогою вашого персонажа.

Casual games – це ігри, призначені для широкої аудиторії користувачів комп'ютерів та мобільних пристроїв. Вони не мають чітких жанрових обмежень, однак, це зазвичай ігри досить прості в освоєнні, на проходження рівнів таких ігор не потрібно багато часу. У такі ігри грають користувачі соціальних мереж, власники планшетів, люди, які мають мобільні телефони чи ігрові консолі.

Віртуальна реальність (virtual reality, VR) може бути визначена як багатоаспектна мультимедійна або комп'ютерна реальність, що імітує присутність користувача в середовищі, яке є місцем у реальному чи уявному світі.

Створена комп'ютером віртуальна реальність (computer-generated (CG) VR) є підмножиною цієї технології, в ній віртуальний світ генерується виключно за допомогою комп'ютерної графіки. Користувач може побачити це віртуальне середовище, надягаючи гарнітуру.

Гарнітура відображає контент прямо перед очима користувача, при цьому система відстежує її переміщення в реальному світі, так що рухи віртуальної камери ідеально

відповідають рухам людини, що наділа гарнітуру. Користувач зазвичай тримає пристрої, які дозволяють системі відстежувати рухи обох рук. Це дає можливість гравцеві взаємодіяти із віртуальним світом.

Терміни «*доповнена реальність*» (augmented reality, AR) та «*змішана реальність*» (mixed reality, MR) часто плутають або використовують взаємозамінно. Обидві технології представляють користувачеві реальний світ із комп'ютерною графікою.

В обох пристрій перегляду, такий як смартфон, планшет або технічно допрацьована пара окулярів, виводить статичні зображення реального світу або що змінюється в реальному часі, а поверх нього накладається комп'ютерна графіка.

Деякі розрізняють ці дві технології, використовуючи термін «доповнена реальність» для опису технологій, у яких комп'ютерна графіка накладається зображення реального світу, але з прив'язується до нього.

Термін «змішана реальність» частіше застосовується до використання комп'ютерної графіки для візуалізації уявних об'єктів, які прив'язані до реального світу і здаються існуючими в ньому. Однак ця відмінність у жодному разі не є загальноприйнятою.

4. Монетизація в іграх

Монетизація в іграх – це конвертація гравців у гроші, незалежно від того, заробляємо ми на продажі самої гри або контенту. Вибір монетизаційної стратегії залежить від особливостей проекту, регіону та переваг цільової аудиторії.

Розглянемо деякі моделі монетизації:

- BUY-TO-PLAY – «купи, щоб грати». Гра виставляється в магазинах, і щоб отримати доступ до неї, гравець повинен оплатити покупку.
- PAY-TO-PLAY (підпискова модель) передбачає, що для постійного доступу до ігрового контенту потрібно оформити платну передплату.
- FREE-TO-PLAY – монетизаційна модель, що дозволяє грати без обов'язкового внесення коштів. Такі ігри заробляють, пропонуючи гравцям за гроші отримати ігрову перевагу, унікальні предмети, відкрити новий ігровий контент тощо. Вбудовані покупки допомагають гравцям просунутися в грі: за додаткову плату можна прискорити процес прокачування, отримати потужні ігрові предмети та інші бонуси. Часто розробники пропонують купити чисто декоративні елементи, що дають змогу зовні виділитися серед інших гравців.
- ВБУДОВАНА РЕКЛАМА – ще одна важлива модель монетизації. Вона дозволяє розробникам заробляти завдяки тому, що гравцеві показується реклама інших ігор, додатків, послуг тощо.
- FREEMIUM – монетизаційна модель, що пропонує гравцю безкоштовну обмежену версію гри з можливістю придбати розширену або premium-версію. Відмінність від класичного Free-to-play у тому, що гравцеві до внесення оплати надають доступ до спочатку урізаного геймплей або обмеженого набору контенту.
- PRODUCT PLACEMENT, тобто наявність неявної (прихованої) реклами – також можливість отримати прибуток. Втома гравців від нав'язливої прямої реклами підштовхнула розробників ігор до партнерства з різними брендами.

5. Життєвий цикл при розробці ігор

На рисунку 1.1 продемонстровано робочий процес розробника ігор.

Рисунок 1.1 – Робочий процес розробника ігор

Розглянемо специфічні дії при створенні комп'ютерних ігор в рамках життєвого циклу.

Концептування (Concept) – на цьому першому етапі команда розробляє концепцію гри, і проводить початкове опрацювання ігрового дизайну. Головна мета даного етапу – це геймдизайнерська документація, що включає розгорнутий документ, що описує гру, як кінцевий бізнес-продукт і Concept Document (початкове опрацювання всіх аспектів гри).

Розробка гри починається із створення концепції гри. Власне, *концепція* – це центральна ідея, навколо якої будується все інше. Концепція гри виражається як *концепт-документ*.

Концепт-документ є першим ігровим документом, що дозволяє отримати загальне уявлення про гру, фіксує її основні особливості. Концепт-документ може будуватися за такою схемою:

– Вступ;
– Жанр та аудиторія;
– Основні особливості гри;
– Опис гри;
– Порівняння та передумови створення гри;
– Платформа;
– Контакти.

Дизайн документ містить детальний опис гри, ігрових ресурсів, ігрової логіки. Дизайн-документ можна вважати розширеною версією концепт-документу.

Дизайн-документ може мати таку структуру:

1. Введення
2. Концепція
    2.1. Вступ
    2.2. Жанр та аудиторія
    2.3. Основні особливості гри
    2.4. Опис гри
    2.5. Передумови створення
    2.6. Платформа
3. Функціональна специфікація
    3.1. Принципи гри
        3.1.1. Суть ігрового процесу
        3.1.2. Хід гри та сюжет

3.2. Фізична модель
3.3. Персонаж гравця
3.4. Елементи гри
3.5. "Штучний інтелект"
3.6. Багатокористувацький режим
3.7. Інтерфейс користувача
3.7.1. Блок-схема
3.7.2. Функціональний опис та управління
3.7.3. Об'єкти інтерфейсу користувача
3.8. Графіка та відео
3.8.1. Загальний опис
3.8.2. Двовимірна графіка та анімація
3.8.3. Тривимірна графіка та анімація
3.8.4. Анімаційні вставки
3.9. Звуки та музика
3.9.1. Загальний опис
3.9.2. Звук та звукові ефекти
3.9.3. Музика
3.10. Опис рівнів
3.10.1. Загальний опис дизайну рівнів
3.10.2. Діаграма взаємного розташування рівнів
3.10.3. Графік запровадження нових об'єктів
4. Контакти

У *вступі* міститься коротке формулювання ідеї гри. Введення має містити інформацію, яка дозволить читачеві зрозуміти сутність гри, її жанр, аудиторію. Фактично, введення – це той мінімум інформації, який дозволяє читачеві – видавцеві, майбутньому члену команди розробників, майбутньому гравцю – зрозуміти – чи цікава йому дана гра, чи хоче він мати до неї відношення.

Відомості про *жанр гри* та *цільової аудиторії* дуже важливі, оскільки дозволяють зрозуміти особливості позиціонування гри, особливості майбутнього просування готового продукту. Як правило, цей розділ має сенс включати інформацію про цільові групи користувачів, яким може бути цікава гра. Тут же корисно навести результати попередньо проведеного дослідження.

Таке дослідження можна провести наступним чином: скласти питання, що складається з кількох питань, які призначені для з'ясування ставлення гравців до тих чи інших концепцій, покладених в основу гри. Питання має сенс складати як анкети, що містить варіанти відповіді питання.

Основні особливості гри – цей розділ повинен включати опис ключових особливостей гри, що відрізняють її від ігор того ж жанру, розрахованих на ту ж цільову групу. У цьому розділі слід вказати приблизний обсяг гри. Обсяг гри може бути виміряний у годиннику або в інших одиницях.

Порівняння і передумови створення – цей розділ повинен містити порівняльний аналіз пропонованої гри з існуючими іграми, аналіз ринкових тенденцій. Тут же слід торкнутися питань ліцензування.

Прототипування (Prototyping) – важливий етап проектування будь-якої гри – створення прототипу. Те, що добре виглядає «на папері», зовсім не обов'язково буде цікавим у реальності. Прототип реалізується з метою оцінки основного ігрового процесу, перевірки різних гіпотез, проведення тестів ігрових механік, перевірки ключових технічних моментів.

Дуже важливо на етапі створення прототипу реалізовувати лише те, що потрібно перевірити й у стислий термін. Прототип може бути простим у реалізації, т.к. після досягнення поставлених перед ним цілей, він має бути «викинутий».

Вертикальний зріз (Vertical Slice) – мета вертикального зрізу – отримати мінімально можливу повноцінну версію гри, що включає повністю реалізований основний ігровий процес. При цьому високу якість опрацювання обов'язково потрібно втілити тільки для тих ігрових елементів, які суттєво впливають на сприйняття продукту. При цьому всі базові фічі гри присутні як мінімум у чорновій якості. Реалізовано мінімальний, але достатній реалізації повноцінного ігрового процесу набір контенту (один рівень чи одна локація).

Виробництво контенту (Content production) – на цьому етапі виготовляється достатня кількість контенту для першого запуску на зовнішню аудиторію. Реалізуються всі фічі, які заплановані до закритого бета-тестування. Це найбільш тривалий етап, який може займати для великих клієнтських проектів рік і більше. На цьому етапі задіяна найбільша кількість фахівців, які займаються виробництвом всього основного наповнення гри. Художники створюють усі графічні ресурси, геймдизайнери налаштовують баланс та заповнюють конфіги, програмісти реалізують та полірують усі фічі.

Friends & Family / CBT (закрите бета-тестування) – на етапі CBT продукт вперше демонструється досить широкому загалу, хоча й лояльному продукту чи компанії. Серед найважливіших завдань на цьому етапі виступають: пошук та виправлення гейм-дизайнерських помилок, проблем ігрової логіки та усунення критичних багів. На цьому етапі в грі присутні вже всі ключові фічі, створено достатньо контенту для повноцінної гри тривалий час, налаштовано збір та аналіз статистики. Тестування відбувається за тест-планом, проводяться стрес-тести вже із залученням реальних гравців.

Soft Launch / OBT (відкритий бета-тест) – на цьому етапі продовжується тестування гри, але вже на широкій аудиторії. Йде оптимізація під великі навантаження. Гра має бути готовою для прийому великого трафіку. На цьому етапі повністю завершується розробка нових фічів. Відбувається feature freeze, програмісти перестають реалізовувати щось нове, а повністю переключаються на налагодження та тюнінг наявних фічів. Геймдизайнери, продюсер та аналітики роблять висновки із зібраної на CBT статистики та перевіряють ефективність монетизації. При цьому до початку етапу повинна повністю функціонувати інфраструктура проекту: сайт, групи соц. мереж, канали залучення (User Acquisition), підтримка користувачів.

Release – на цьому етапі має бути повністю налагоджено оперування продукту (технічна підтримка, робота з ком'юніті), дотримуються маркетингові та фінансові плани, ведуться роботи з покращення фінансових показників, активно відпрацьовуються канали залучення трафіку. Команда розробки цьому етапі займається виправленням технічних багів, виявлених у процесі експлуатації та оптимізацією продукту.

Контрольні питання

1. З кого складається типова ігрова команда?
2. Для чого потрібно розуміти які типи користувачів існують?
3. Які жанри комп’ютерних ігор ви знаєте?
4. Що таке монетизація?
5. З яких етапів складається життєвий цикл при розробці ігор?

Лекція 2. Інтерфейс Unity

1. Unity Hub
2. Редактор Unity
3. Сцена

1. Unity Hub

Unity Hub – це корисний інструмент для керування проектами та встановленими версіями Unity. Використовуйте Hub для роботи з кількома версіями редактора Unity та пов'язаних з ними компонентів, для створення нових або відкриття створених проектів.

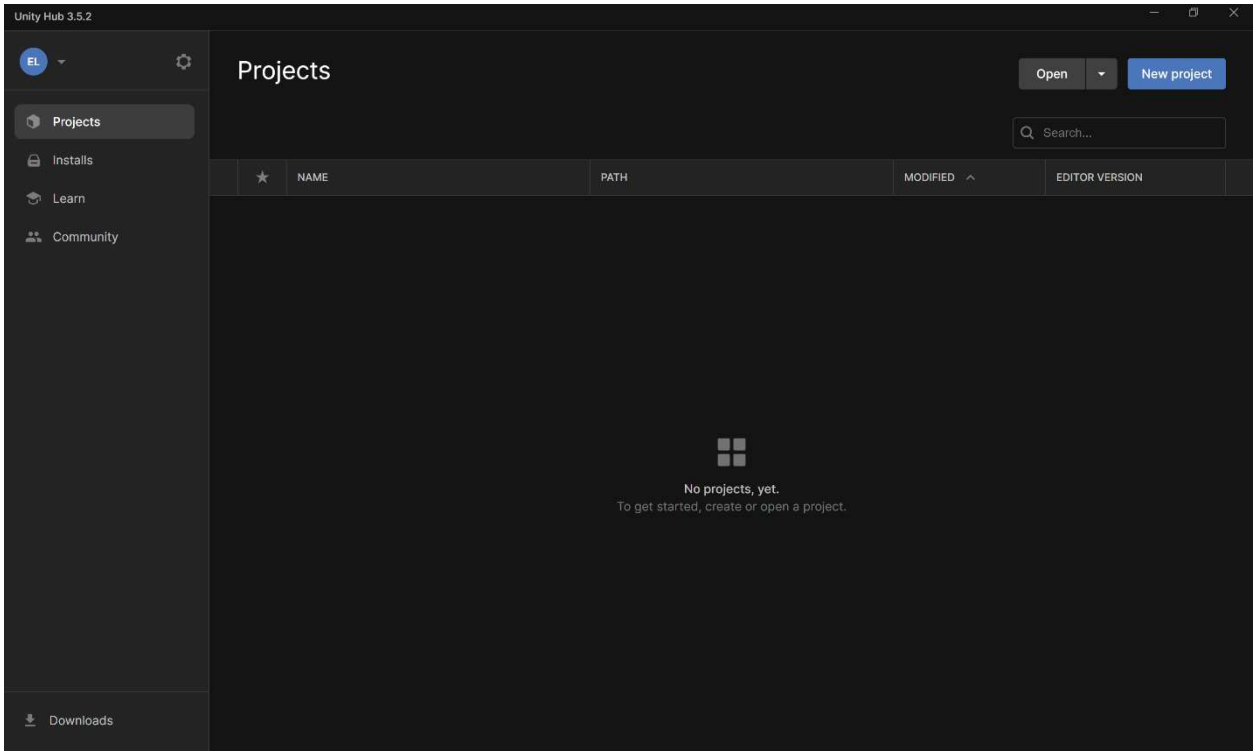


Рисунок 2.1 – Вікно Unity Hub

Unity Hub – це десктопна програма, спроектована для зручної роботи користувачів. З нього відбувається доступ до екосистеми ігрового двигуна Unity, робота з менеджером проектів створених у Unity, управління ліцензіями та встановлення додаткових компонентів.

Unity Hub є своєрідною “точкою старту”, в якій відбувається створення нових проектів (вкладка Projects), встановлення різних версій Unity (вкладка Installs) тощо.

У центральній частині програми вказані проекти (Projects), з якими ви працюєте. Якщо ви використовуєте Unity вперше, то це вікно має бути порожнім.

При натисканні закладки Installs у лівому меню відкриється вікно вибору версій Unity для встановлення або буде показано перелік уже встановлених версій Unity.

LTS (Long Term Support) у Unity означає довгострокову підтримку. Це версія Unity, призначена для розробників, яким потрібна максимальна стабільність та підтримка. LTS версії отримують виправлення помилок та патчі безпеки протягом двох років, що робить їх ідеальними для проектів, які потребують тривалого обслуговування та підтримки.

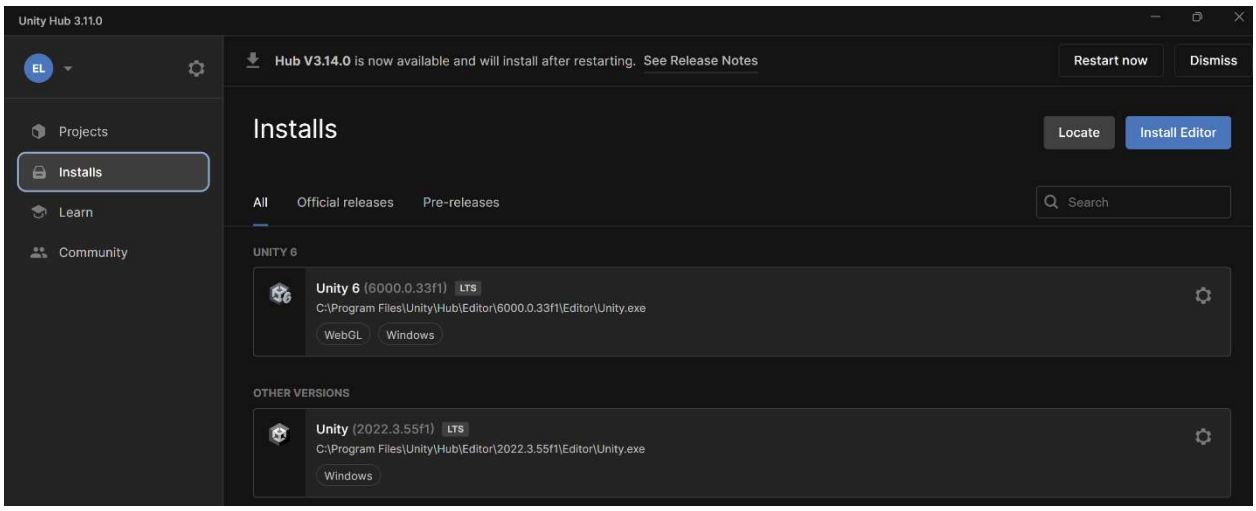


Рисунок 2.2 – Вікно Unity Hub Installs

Якщо надалі вам знадобляться інші версії середовища розробки Unity (наприклад, ви знайдете і захочете подивитися готові проекти, зроблені під попередні версії середовища розробки), – то ви завжди зможете відкрити Unity Hub, перейти у вкладку Install і завантажити версії Unity і модулі .

Для створення порожнього 3D проекту зайдіть у Unity Hub та натисніть кнопку New project.

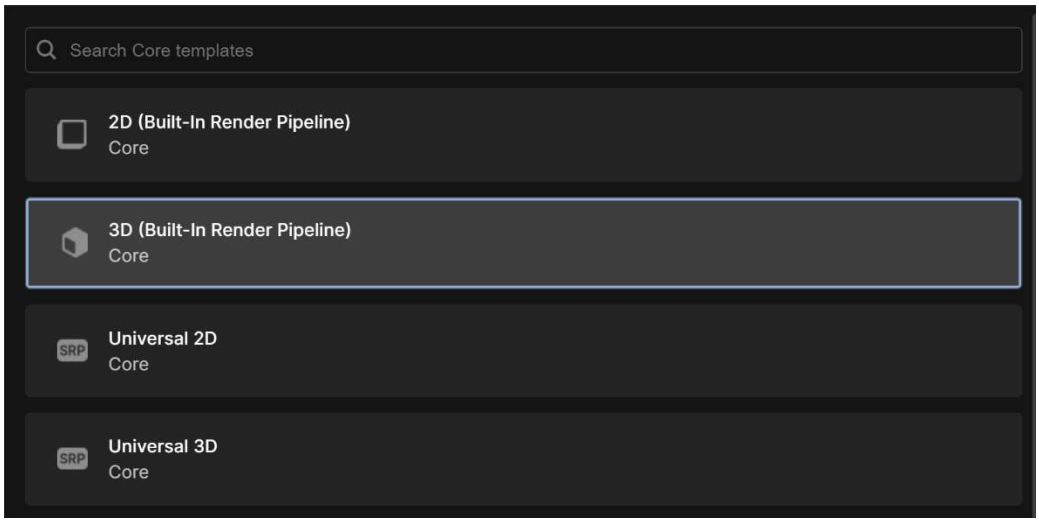


Рисунок 2.3 – Вікно для вибору шаблону проекту, що створюється

Вибираємо кнопку 3D (Built-In Render Pipeline), вказуємо ім'я проекту та його місцезнаходження та натискаємо кнопку Create project. Коли новий проект ініціалізується, з'явиться редактор Unity.

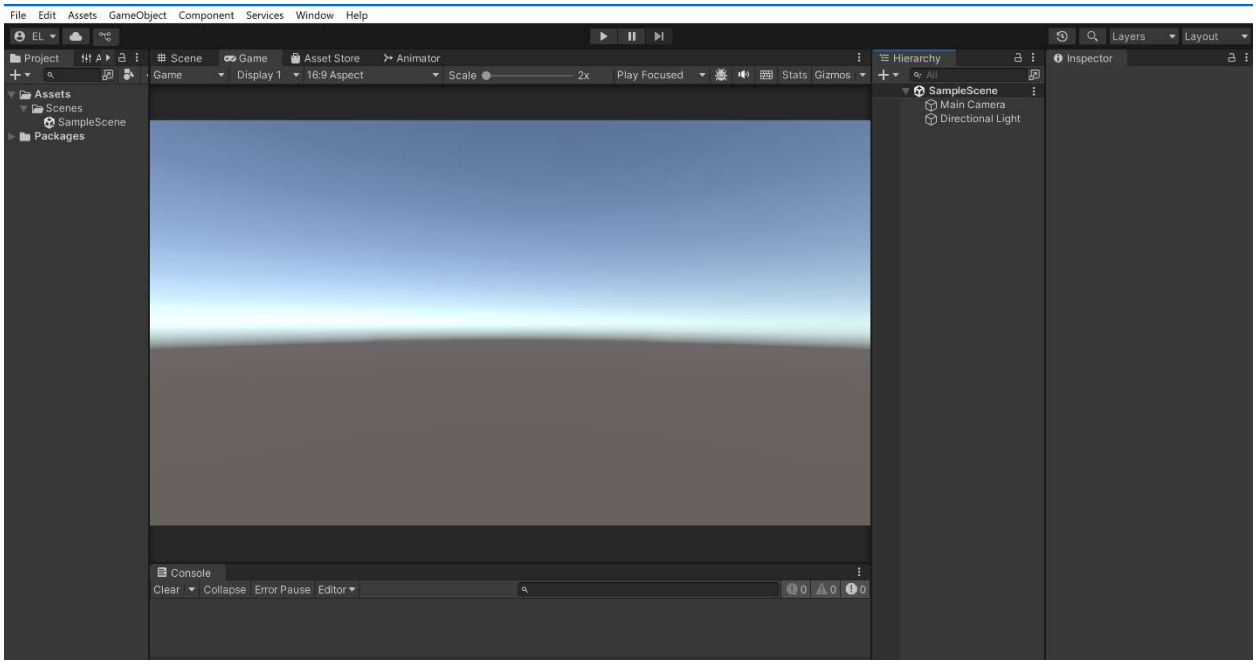
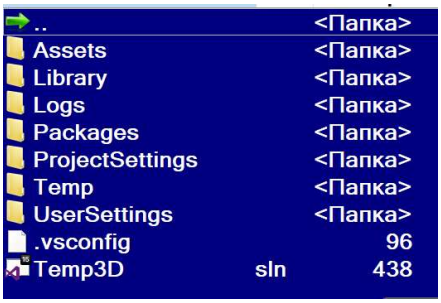


Рисунок 2.4 – Вікно проекту 3D

2. Редактор Unity

Проект Unity – це не єдиний файл, а папка, що містить деяку кількість папок. Розглянемо три найважливіші підпапки:

– Assets,
– ProjectSettings
– Library.



Папка Assets містить усі файли, які використовуються грою: рівні, текстури, звукові ефекти та сценарії. Папка Library містить внутрішні дані для Unity, а папка ProjectSettings – файли з налаштуваннями проекту. В основному не доведеться торкатися файлів у папках Library та ProjectSettings.

Інтерфейс Unity відображається як набір панелей. Кожна панель має вкладку ліворуч угорі, за яку панель можна перемістити і тим самим змінити макет програми. Також, переміщуючи панель за вкладку, її можна перетворити на самостійне вікно. Не всі панелі видимі за замовчуванням, і в міру розробки гри ви відкриватимете нові панелі через меню Window.

Інтерфейс Unity розбитий на кілька частин: вкладка Scene, вкладка Game, панель інструментів, вкладка Hierarchy, панель Inspector, вкладки Project та Console. Кожна частина має власне призначення, при цьому всі вони відіграють важливу роль у циклі створення гри.

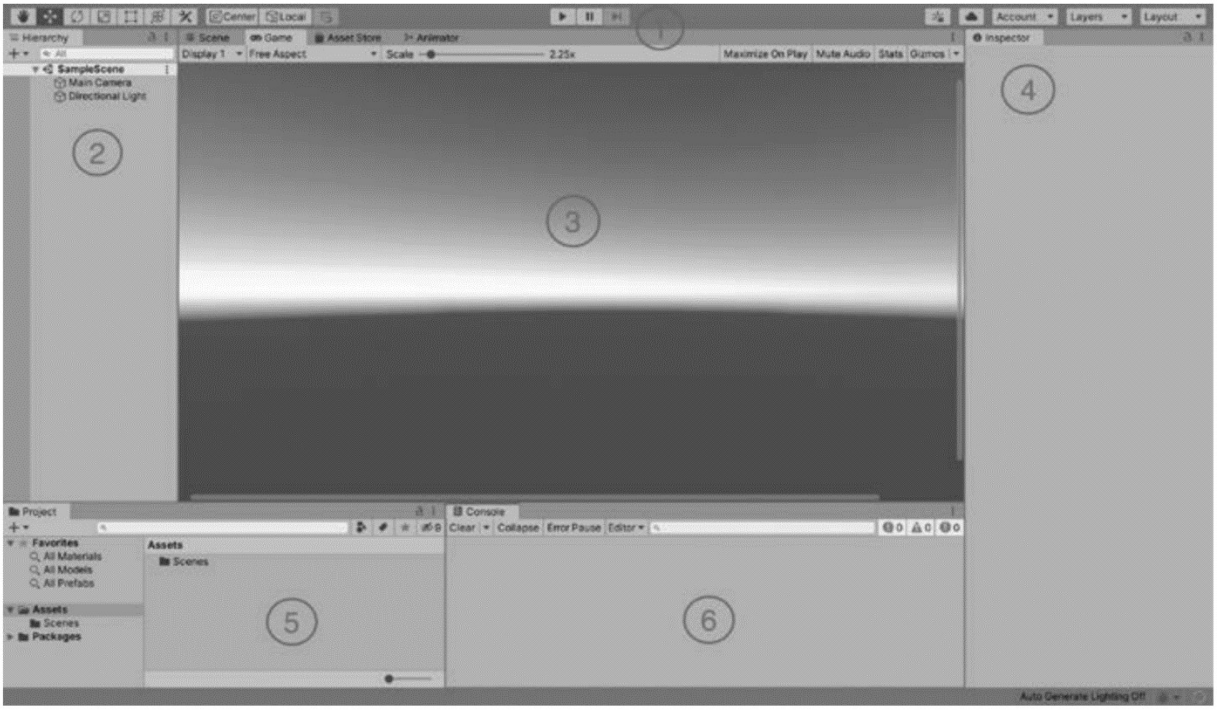
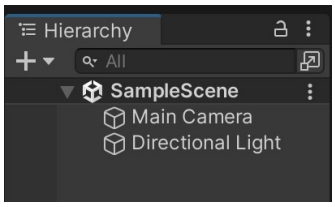


Рисунок 2.5 – Вікно редактору Unity

Панель Toolbar – найвища частина редактора Unity. На цій панелі можна керувати об'єктами (кнопки зліва), а також запускати та призупиняти гру (центральні кнопки). Праворуч розташовані кнопки сервісів Unity, шари-маски та опції макетів.

На панелі Hierarchy відображаються всі елементи, які зараз знаходяться на ігровій сцені. У новоствореному проекті є камера та спрямоване джерело світла, але в міру створення гри ця панель почне наповнюватися об'єктами.



Панель Hierarchy (Ієрархія) відображає список усіх об'єктів у сцені, відкритій на даний момент. Працюючи зі складними сценами ієрархія дозволяє швидко знаходити потрібні об'єкти за іменами.

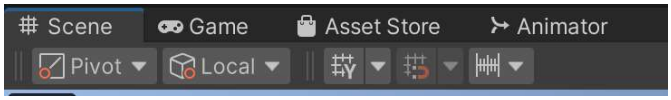
Ієрархія, як випливає з назви, дозволяє також бачити стосунки батьків-нащадок між об'єктами. У Unity об'єкти можуть містити інші об'єкти. Панель ієрархії дозволяє досліджувати такі дерева об'єктів. Також підтримується можливість переупорядковувати об'єкти списки, перетягуючи їх мишею.

Панелі Game і Scene – найвізуальніші наочні частини редактора. Панель Scene – це робоча поверхня, на якій можна розміщувати та переміщувати 2D- та 3D-об'єкти. Коли ви натискаєте кнопку Play, з'являється вікно Game, в якому буде показана та сама сцена, але вже з робочими запрограмованими взаємодіями.

Редактор Unity завжди знаходиться в одному з двох режимів: Edit Mode (Режим редагування) або Play Mode (Режим відтворення). У режимі редагування, що діє за умовчанням, можна створювати сцени, налаштовувати ігрові об'єкти та виконувати інші дії,

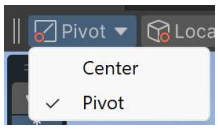
пов'язані зі створенням гри. У режимі відтворення ви можете грати в свою гру і взаємодіяти зі сценою.

На панелі Scene вгорі, поряд з інструментами, можна побачити ще кнопки.



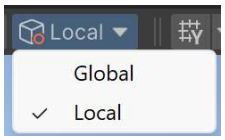
Зараз тут написано Pivot та Local. Ці кнопки це інструменти, це властивості інструментів.

Перша перемикається між пивот та центр. Ця властивість показує нам позицію наших інструментів, точніше центр, звідки виходять стрілки осей або напрямні обертання.



Пивот та центр однакові для куба та для більшості простих об'єктів вони збігатимуться. Але в майбутньому вам будуть зустрічатися об'єкти, в яких пивот не збігається, і якщо ви хочете обертати навколо пива або центру, це те, що ви перемикаєте.

Наступне властивість показує обертання інструментів. Нині воно стоїть локально.

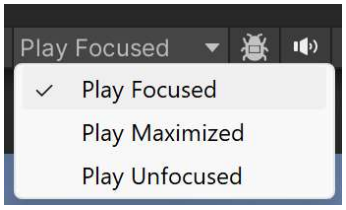


При повороті куба осі інструментів також повертається. Це тому зараз їх обертання локальні. Натиснувши кнопку і зробивши глобальним, можна побачити, що осі повертаються у вихідне становище.

Таким чином, ми можемо вибирати, чи є обертання нашого об'єкта також обертанням осей його інструменту.

Щоб перейти в режим відтворення, клацніть по кнопці Play (Грати) у верхній частині вікна редактора – Unity запустить гру. Щоб залишити режим відтворення, натисніть кнопку Play (Грати) ще раз. Перебуваючи в режимі відтворення, можна зупинити гру, клацнувши піктограму Pause (Пауза) в центрі панелі керування режимом відтворення.

Поведінка в режимі відтворення:



У Unity є кілька різних режимів запуску Play Mode у редакторі. Вони впливають на те, як вікно Game буде поводитися під час відтворення сцени.

Play Focused:

– При натисканні Play вікно Game автоматично отримує фокус.
– Тобто курсор і клавіатурні події спрямовуються одразу в Game View.
– Зручно, коли ви тестуєте управління (клавіатура, мишка, геймпад) і не хочете щоразу клікати у вікно гри.

– Недолік: якщо вам потрібно одночасно дивитися на Scene View чи інші панелі, вони втрачають фокус.

Play Maximized:

– Коли ви запускаєте Play Mode, Game View автоматично розгортається на весь простір редактора (в межах Unity).
– Це імітує гру у «повноекранному режимі», але без виходу за межі редактора.
– Зручно для перевірки UI, адаптивності інтерфейсу, візуальної якості сцени.
– Після зупинки Play Mode вікно повертається до попереднього розміру/розташування.

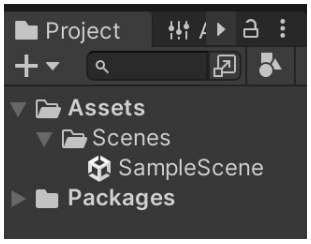
Play Unfocused:

– При запуску Play Mode вікно Game не отримує автоматично фокус.
– Це означає, що ввод із клавіатури/мишки може йти не в Game View, а в інші панелі (наприклад, Scene або Inspector).
– Зручно, коли ви хочете одночасно редагувати налаштування об’єктів у Inspector, змінювати параметри, не перемикаючись постійно між панелями.
– Використовується, коли під час тестів важливі не стільки управління в грі, скільки спостереження за змінами об’єктів у сцені.

Проекти в Unity поділені на сцени. Кожна сцена містить колекцію ігрових об'єктів. Сцену можна розглядати як один рівень у грі, але сцени також допомагають поділити гру на керовані фрагменти. Наприклад, головне меню гри зазвичай реалізується у вигляді окремої сцени, як і кожен її рівень.

Панель Inspector – узагальнений інструмент для перегляду та редагування властивостей ваших об'єктів. Вибравши, наприклад, компонент Main Camera, ви побачите, що він має кілька частин (в Unity вони називаються компонентами) і всі вони знаходяться саме тут.

У вікні Project представлені всі ресурси, які зараз знаходяться у вашому проекті. Якщо простіше, то це всі файли та папки вашого проекту. Додавати нові файли потрібно до папки Assets.

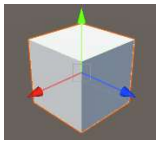


На *панелі Console* з'являться вихідні дані, які виводитимуть створені сценарії.

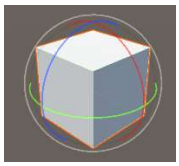
Подання сцени може бути в одному з п'яти різних режимів. Перемикач режимів, що знаходиться у вікні, керує режимом взаємодій з поданням сцени.

Режим захоплення  (Grab mode) – у цьому режимі можна натиснути ліву кнопку миші та зрушити уявлення.

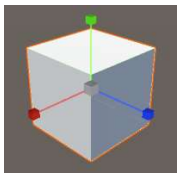
Режим переміщення  (Translation mode) – у цьому режимі можна переміщати виділені об'єкти.



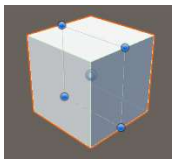
Режим обертання  (Rotation mode) – у цьому режимі можна повертати виділені об'єкти.



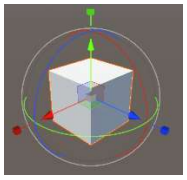
Режим масштабування  (Scale mode) – в цьому режимі можна змінювати розміри виділених об'єктів.



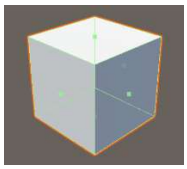
Режим прямокутника  (Rectangle mode) – у цьому режимі можна переміщати вибрані об'єкти та змінювати їх розміри, використовуючи двовимірні маркери. Це особливо зручно при формуванні двовимірних сцен або інтерфейсу користувача.



Наступний інструмент поєднує у собі функціональність всіх попередніх. За допомогою цього інструменту  ми можемо рухати об'єкт, збільшувати його розмір та обертати його.



Останнє  – це кастомні інструменти. Ці інструменти ми можемо встановити по-різному для кожного об'єкта. Для куба він ставатиме як Edit Box Collider.



Перемикання між режимами представлення сцени можна здійснювати за допомогою перемикача або клавішами Q, W, E, R та T.

Обирати об'єкти можна в будь-якому режимі, крім режиму захоплення.

Підтримується кілька способів навігації у поданні сцени:

– Клацнути по піктограмі із зображенням руки ліворуч угорі, щоб перейти в режим захоплення, потім натиснути ліву кнопку миші та зрушити сцену в потрібному напрямку.
– Утримуючи клавішу Alt, натиснути ліву кнопку миші та повернути сцену на потрібний кут.
– Вибрати об'єкт у сцені, клацнувши по ньому лівою кнопкою або вибравши його в панелі ієрархії, помістити вказівник миші у межі панелі з представленням сцени та натиснути F, щоб передати фокус введення вибраному об'єкту.
– Утримуючи праву кнопку, перемістити мишу, щоб озирнутися. Поки права кнопка миші залишається натиснутою, можна використовувати клавіші W, A, S і D, щоб змістити камеру вперед, ліворуч, назад та праворуч. Клавішами Q та E камеру можна зміщувати вгору та вниз. Якщо при цьому ще натиснути та утримувати клавішу Shift, переміщення відбуватимуться швидше.

Панель Hierarchy (Ієрархія) відображає список усіх об'єктів у сцені, відкритій на даний момент. Працюючи зі складними сценами ієрархія дозволяє швидко знаходити потрібні об'єкти за іменами.

Ієрархія, як випливає з назви, дозволяє також бачити стосунки батьків-нащадок між об'єктами. У Unity об'єкти можуть містити інші об'єкти. Панель ієрархії дозволяє досліджувати такі дерева об'єктів. Також підтримується можливість переупорядковувати об'єкти списки, перетягуючи їх мишею.

Інспектор відображає інформацію про об'єкт, обраний в даний момент, і саме тут можна буде здійснювати налаштування своїх ігрових об'єктів. Інспектор відображає список усіх компонентів, підключених до вибраного об'єкта чи ресурсу. Для різних компонентів відображається різна інформація.

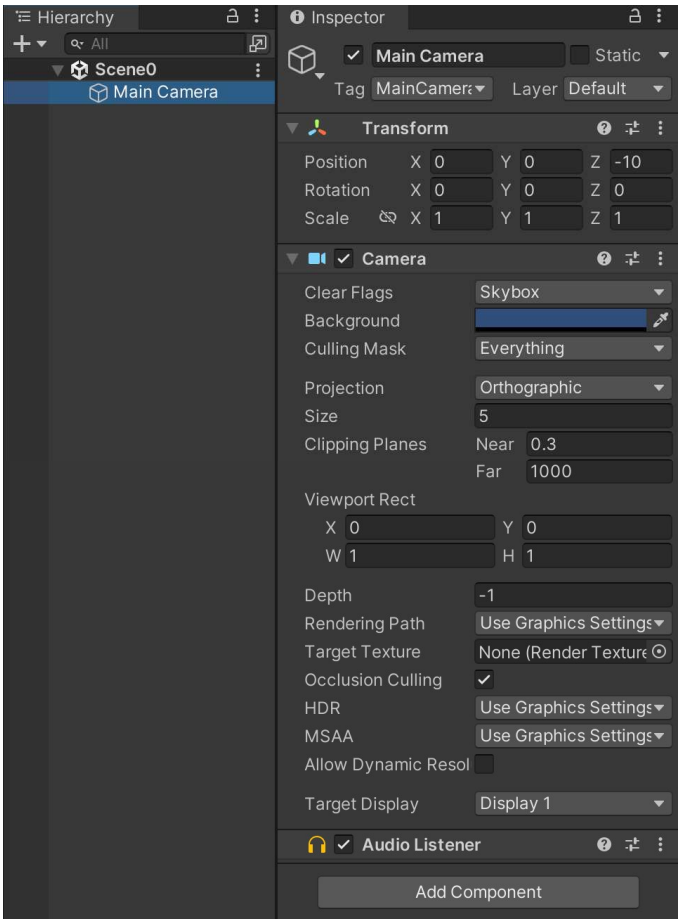


Рисунок 2.6 – Панель Hierarchy та панель Inspector для об’єкту MainCamera

3. Сцена

Сцени містять об'єкти гри. Вони можуть використовуватися для створення головного меню, окремих рівнів та інших цілей. Кожен файл сцени можна вважати окремим ігровим рівнем. У кожній сцені можна розмістити об'єкти оточення, загородження, декорації, шматочками створюючи дизайн і саму гру.

Коли ви створюєте новий проект Unity, у поданні сцени відображатиметься нова сцена. Це сцена без назви та збереження. Сцена буде порожньою, за винятком об'єктів – камери та спрямованого світла.

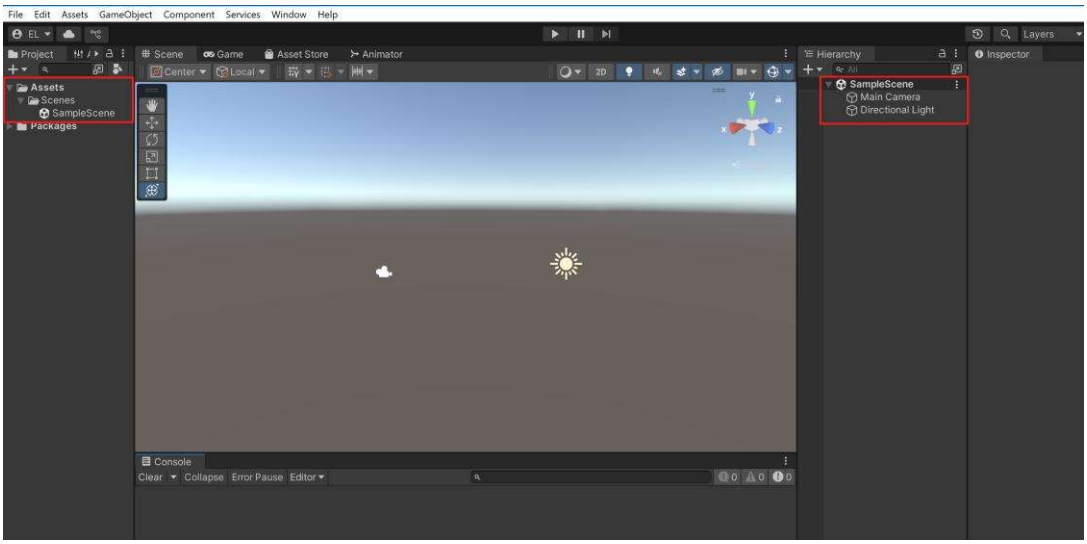


Рисунок 2.7 – Сцена в Unity

Використовуйте діалогове вікно «Нова сцена» для створення нових сцен із певних шаблонів сцен у вашому проекті. Ви також можете використовувати діалогове вікно «Нова сцена» для пошуку шаблонів сцен та керування ними.

За замовчуванням діалогове вікно «Нова сцена» відкривається під час створення нової сцени з меню (Файл → Нова сцена)

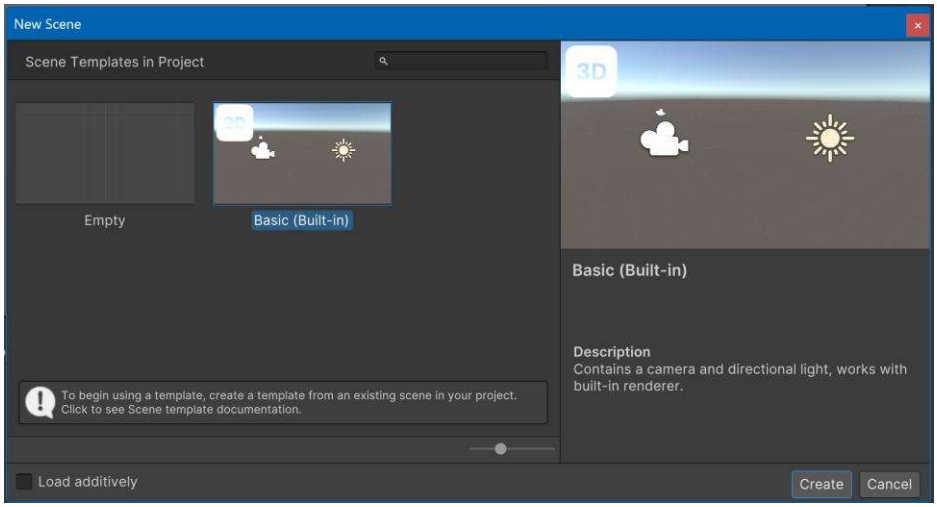
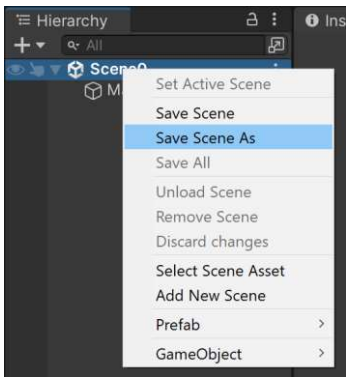


Рисунок 2.8 – Діалогове вікно «Нова сцена»

Коли ви створюєте нову сцену з меню, Unity автоматично копіює базовий шаблон проекту і додає нову сцену до будь-якої папки, відкритої на даний момент у вікні проекту. Нову сцену потрібно зберегти.



Щоб відкрити сцену, виконайте одну з таких дій:

– У вікні «Проект» двічі клацніть об'єкт сцени.
– У меню виберіть Файл → Нова сцена.
– У меню виберіть Файл → Останні сцени → [назва сцени]

Якщо ваша поточна сцена містить незбережені зміни, Unity запропонує зберегти сцену або скасувати зміни.

Існує кілька способів переходу між сценами, а саме:

– зміна сцен за назвою;
– зміна сцен за індексами;
– зміна сцен за допомогою параметрів.

Зміна сцен за назвою – цей спосіб найпростіший, і зустрічається дуже часто. Він використовується тоді, коли нам потрібно перейти точно на певну сцену. Для цього створимо C# скрипт з назвою, наприклад, Scenes. І пропишемо в ньому наступний код:

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class Scenes : MonoBehaviour
{
    public void OpenMenu()
    {
        SceneManager.LoadScene("Menu");
    }

    public void OpenGame()
    {
        SceneManager.LoadScene("Game");
    }

}
```

Зміна сцен за індексами – даний спосіб аналогічний попередньому, за винятком того, що ми замість назв сцен, використовуємо їх індекси. Цей спосіб корисний у тих випадках, коли нам необхідно перейти на наступну або попередню сцену.

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class Scenes : MonoBehaviour
{
    public void OpenMenu()
    {
        SceneManager.LoadScene(0);
    }

    public void OpenGame()
    {
        SceneManager.LoadScene(1);
    }

}
```

Завдяки функціям відкриваються трохи більше можливостей переходу на сцени. Наприклад, ми можемо перейти на наступну чи попередню.

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class Scenes : MonoBehaviour
{
    public void NextLevel()
    {
        var index = SceneManager.GetActiveScene().buildIndex;
        SceneManager.LoadScene(index + 1);
    }

    public void PreviousLevel()
    {
        var index = SceneManager.GetActiveScene().buildIndex;
        SceneManager.LoadScene(index - 1);
    }
}
```

Перед тим як працювати зі сценами за їх індексами, нам необхідно додати всі сцени до Build Settings. Для цього тиснемо на вкладку File → Build Settings. Далі перетягуємо всі свої сцени в область Scenes in Build і розставляємо їх по порядку. Після цього просто закриваємо це вікно.

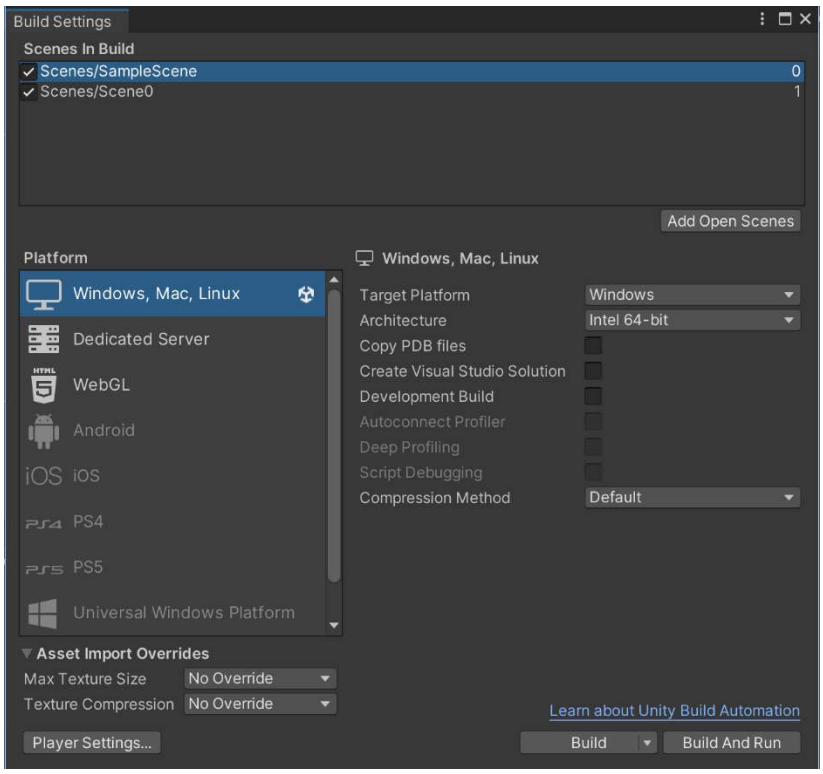


Рисунок 2.9 – Вікно «Build Settings»

Зміна сцен за допомогою параметрів – даний спосіб аналогічний попередньому, тільки тут ми переходимо на певну сцену, виходячи з переданого аргументу в нашу функцію.

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class Scenes : MonoBehaviour
{
    public void GoToLevel(int number)
    {
        SceneManager.LoadScene(number);
    }

}
```

За замовчанням на сцені присутні такі об’єкти як камера та джерело світла.

Camera (камера) – так називають точку в ігровому просторі, з якою гравець бачить ігровий світ. З точки зору положення камери гри можна поділити на ігри від першої особи (камера розташована так, що реалізує вигляд ніби "з очей" персонажа), ігри з виглядом зліва-зверху (стратегії), ігри з виглядом ззаду – камера розташовується зазвичай позаду гравця і трохи вище за нього. Існують і ігри, де камера може приймати різні позиції.

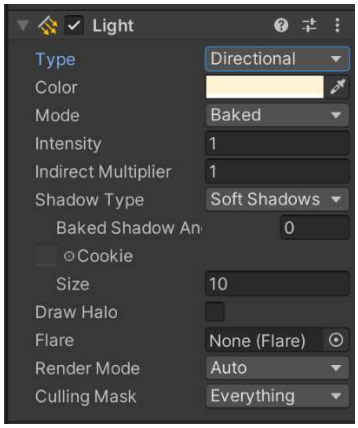
Камера показує нам те, що ми бачимо у нашій грі. У вікні сцени ми можемо бачити зону видимості камери.

Приклад параметрів камери**:**

- Clear Flags — що очищати перед рендером (небо, колір, нічого).
- Background — колір фону.
- Field of View (FOV) — кут огляду.
- Clipping Planes — ближня і дальня площина відсікання.

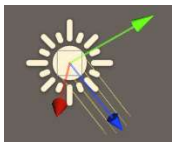
– Projection — перспектива чи ортографія.

Джерела світла – відповідають за освітлення сцени.

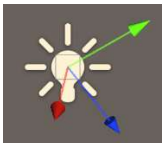


Типи:

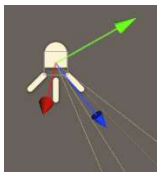
– Directional Light — «сонце», рівномірно світить на всі об’єкти.



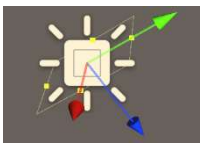
– Point Light — світиться від точки у всі сторони (як лампочка).



– Spot Light — прожектор з кутом світіння.



– Area Light — прямокутне джерело (працює лише з певними рендерами).



Параметри:

– Intensity — яскравість.
– Color — колір світла.
– Range — радіус дії (для Point і Spot).
– Shadows — увімкнені/вимкнені тіні, їхня якість.

Контрольні питання

1. Що таке Unity Hub та для чого він використовується?
2. Для чого використовується папка Assets?
3. Для чого використовується вкладка Hierarchy?
4. Для чого використовується панель Inspector?
5. В яких режимах може бути подання сцени?
6. Як створити нову сцену?
7. Що за замовчанням розташовується на сцені?
8. Для чого використовується камера?
9. Для чого використовується джерело світла?

Лекція 3. 3D. GameObject та його налаштування

1. Ассети
2. GameObjec
2. Матеріал
3. Rigidbody та Prefab
4. Анімація

1. Ассети

Ассети – це компоненти, які являють собою графіку, звуковий супровід або скрипти. Вони прикріплюються до об'єктів і становлять важливу частину гри.

Папка Assets – головна папка, в якій містяться всі ассети, які можуть бути використані проектом на Unity. Вміст Project view відповідає вмісту папки Assets. Більшість функцій API припускають, що все міститься в папці Assets і не вимагають явної вказівки розташування.

Рекомендується під кожен вид ассетів створювати свою окрему папку в Assets.

Хоча єдиного способу організації проекту Unity не існує, кілька ключових рекомендацій:

- Задокументуйте угоди про імена та структуру папок. Посібник із стилю та/або шаблон проекту спрощують пошук та організацію файлів.
- Не використовуйте пробіли в іменах файлів та папок. У інструментів командного рядка Unity виникають проблеми з іменами шляхів, що містять пробіли. Використовуйте CamelCase як альтернативу пробілам.
- Окремі зони тестування чи пісочниці. Створіть окрему папку для невиробничих сцен та експериментів. Підпапки з іменами користувачів можуть розділити вашу робочу область на учасників команди.
- Тримайте свої внутрішні ресурси окремо від сторонніх. Якщо ви використовуєте ресурси з Asset Store або інших плагінів, швидше за все вони мають власну структуру проекту. Тримайте свої активи окремо.

Хоча заданої структури папок немає, розглянемо приклад того, як ви можете налаштувати свій проект Unity.

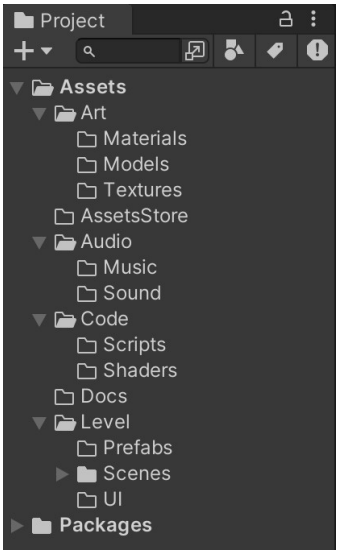
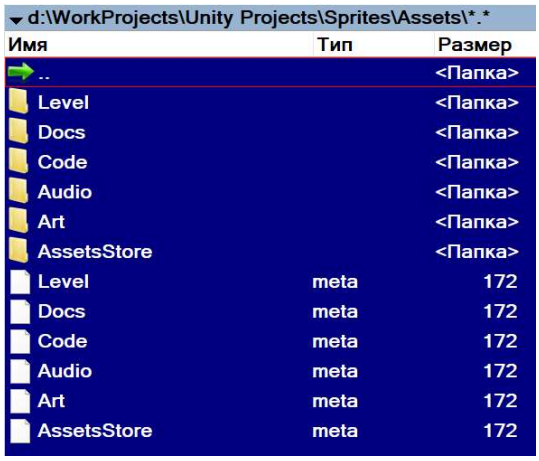


Рисунок 3.1 – Приклад структури папок в папці Assets

Unity генерує мета-файл для кожного файлу усередині проекту. Хоча мета-файл створюється автоматично, він містить багато інформації про файл, з яким він пов'язаний.

Коли Unity створює файл .meta для об'єкта, він записує ідентифікатор об'єкта у файл .meta і зберігає файл .meta у тому місці, що й файл ресурсу.

Приклад:



Метафайли містять важливу інформацію про те, як актив використовується в проекті, і вони повинні залишатися у файлі активу, до якого вони належать. Якщо ви переміщуєте або перейменовуєте ресурс у власному вікні проекту Unity, Unity також автоматично переміщує або перейменовує відповідний файл .meta.

Однак, якщо ви переміщуєте або перейменуєте ресурс за межами Unity, ви повинні перемістити або перейменувати файл .meta, щоб він відповідав.

Якщо ресурс втрачає свій метафайл (наприклад, якщо ви переміщуєте або перейменовуєте ресурс за межами Unity, але не переміщаєте або перейменовуєте відповідний файл .meta), будь-яке посилання на цей ресурс у вашому проекті не працює.

У цій ситуації Unity зауважує, що ресурс не має відповідного метафайлу, створює новий для переміщеного/перейменованого ресурсу, ніби це був абсолютно новий ресурс, і видаляє старий «безхазяйний» файл .meta.

Цей процес може спричинити серйозні проблеми у вашому проекті. Наприклад:

– Якщо ресурс текстури втрачає свій файл .meta, будь-які матеріали, що використовують цю текстуру, втрачають посилання на цю текстуру. Щоб це виправити, вам потрібно буде вручну перепризначити цю текстуру всім матеріалам, для яких вона потрібна.
– Якщо ресурс сценарію втрачає свій файл. Щоб виправити це, вам потрібно буде вручну перепризначити цей скрипт для будь-яких ігрових об'єктів, яким він потрібний.

2. GameObject

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.

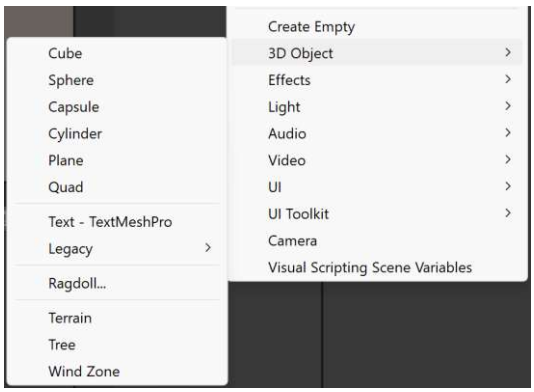


Рисунок 3.2 – Приклад GameObject

Unity 3D-об'єкти включають вбудовані примітиви (куби, сфери тощо) і імпортовані моделі із зовнішніх програм, які зазвичай використовують формати FBX або OBJ для коректного відображення текстур, анімації і сітки. Для додавання об'єктів у сцену використовується меню GameObject або просто перетягування файлів у вікно проекту.

Unity надає набір базових геометричних форм, які можна створювати прямо в редакторі:

– Куб (Cube),
– Сфера (Sphere),
– Циліндр (Cylinder),
– Капсула (Capsule),
– Площина (Plane),
– Квад (Quad).

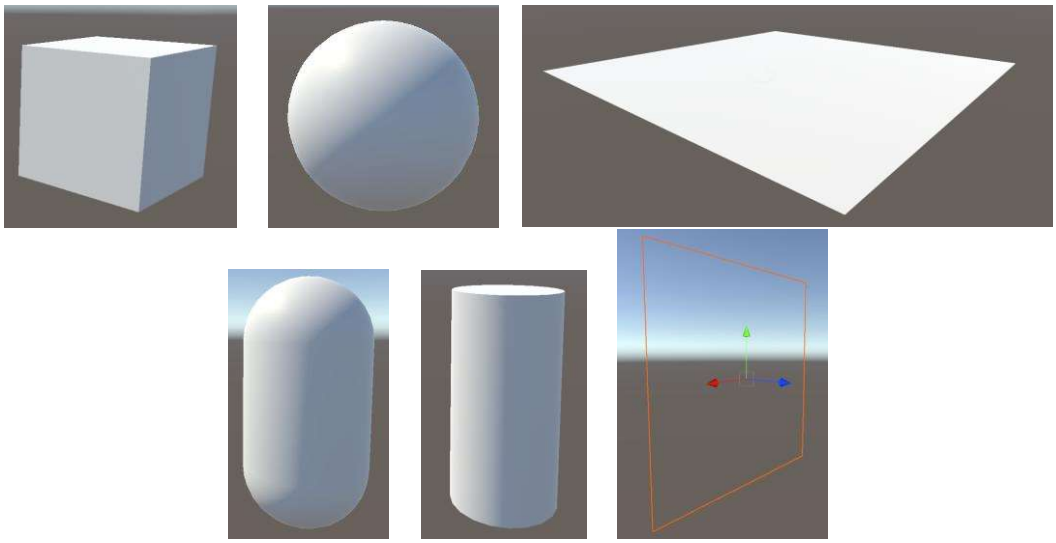


Рисунок 3.3 – Приклад базових 3D-об'єктів

Quad – це примітивний 3D-об'єкт, що є плоскою, двосторонньою поверхнею, створеною з двох трикутників, яка використовується для відображення зображень, текстур або створення простих елементів у 3D-сцені. На відміну від інших примітивів, таких як куби або сфери, Quad не має об'єму і служить лише як поверхня для чогось.

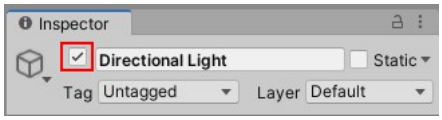
Ключові особливості Quad:

– Двостороння – на відміну від інших об'єктів, які можуть бути односторонніми, Quad має дві сторони, що робить його зручним для відображення зображень з обох сторін;
– Плотська геометрія – він є плоским об'єктом, що відрізняє його від більш об'ємних примітивів, таких як куб.

Ці об'єкти служать для швидкого прототипування, так і для створення базових елементів сцени, наприклад, площину можна використовувати як поверхню землі.

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. Ви можете керувати цим за допомогою GameObject.SetActive.



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.

Ви можете використовувати функцію GameObject.FindWithTag(), щоб знайти GameObject, налаштувавши її на пошук будь-якого об'єкта, що містить потрібний вам тег.

Наприклад:

```
GameObject respawn = GameObject.FindWithTag("Respawn");
```

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.

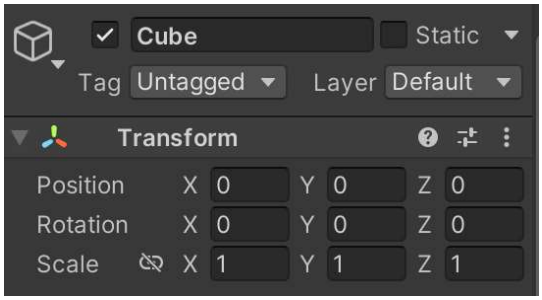


Рисунок 3.4 – Компонент Transform

Інші компоненти, які надають об'єкту його функціональність, можуть бути додані з меню Компонент редактора або скрипта.

Розглянемо компоненти Mesh Filter та Mesh Renderer.

Компонент MeshFilter в Unity використовується для визначення геометрії об'єкта в сцені. Він відповідає за зв'язок об'єкта з певною 3D-сіткою (mesh), яка визначає його форму. Цей компонент працює у парі з MeshRenderer.

Поки MeshFilter визначає, яку саме сітку використовувати, MeshRenderer відповідає за відображення цієї сітки за допомогою матеріалів та світла.

MeshFilter корисний для динамічної зміни форми об'єкта або завантаження сіток під час виконання. Він надає доступ до основних властивостей та методів, що дозволяють розробнику керувати 3D-об'єктами більш гнучко.

Основні характеристики MeshFilter:

1. Дозволяє завантажувати та призначати нові сітки (Mesh) об'єкту під час роботи.

2. Використовується для роботи з довільною геометрією, включаючи процедурно створені моделі.

3. Забезпечує базовий зв'язок між фізичним представленням об'єкта та його візуальною частиною.

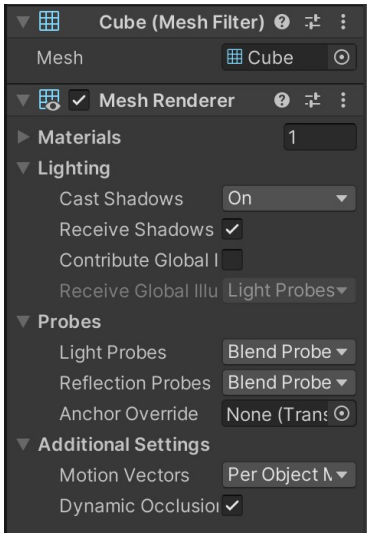


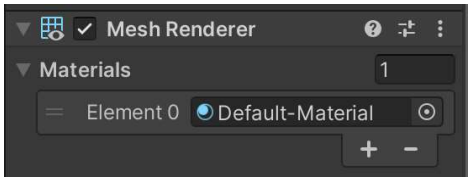
Рисунок 3.5 – Приклад компонентів Mesh Filter та Mesh Renderer

Компонент MeshRenderer в Unity відповідає за візуалізацію 3D-об'єктів у сцені. Його використовують для відображення моделі об'єкта з урахуванням налаштованого матеріалу. Без цього компонента об'єкти з мешем не будуть видимі на екрані.

Матеріали, налаштовані для MeshRenderer, визначають колір, текстуру та інші візуальні властивості відображуваного об'єкта.

Налаштування MeshRenderer дозволяють керувати різними параметрами рендерингу, такими як видимість об'єкта, увімкнення тіней та робота з кількома матеріалами.

У компонента MeshRenderer є властивість матеріалу. За умовчанням тут стоїть default матеріал. З назви зрозуміло, що це стандартний матеріал.



3. Матеріал

Розглянемо створення нового матеріалу.

Для початку в папці Asets ми створимо нову папку та назвемо її Materials.

У створеній папці натискаємо правою кнопкою та вибираємо Create → Material. Тепер можна в параметрі об'єкта перетягнути матеріал, який створили.

Якщо виділити створений матеріал в папці, то в інспекторі можна знайти властивості для налаштування цього матеріалу.

У Unity властивості матеріалу, такі як Albedo, Metallic, Normal Map, Occlusion, Height Map і Detail Map, визначають, як виглядає об'єкт і як він взаємодіє зі світлом.

Albedo відповідає за базовий колір, *Metallic* – за те, наскільки матеріал є металевим, а Normal Map додає деталі поверхні без збільшення полігонів. *Occlusion* визначає області, де світло блокується, *Height* Map створює ефект рельєфу, а *Detail Map* додає дрібні деталі до поверхні, наприклад подряпини або бруд.

Albedo (Альбедо):

- Це базовий колір матеріалу, який залежить від освітлення;
- Він задає відбивну здатність поверхні, але з враховує металеві властивості;
- У цьому полі зазвичай знаходиться текстура, яка визначає колір об'єкта.

Metallic (Металичність):

- Ця властивість визначає, чи є матеріал металевим або діелектричним (неметалевим);
- Значення 1 (або білий колір текстури) означає повністю металевий матеріал, а 0 (чорний) — неметалічний;
- Металеві матеріали відбивають світло по-іншому і менше розсіюють його.

Normal Map (Карта нормалей):

- Текстура, яка "обманює" двигун, створюючи ілюзію дрібних деталей та рельєфу на поверхні, не додаючи при цьому нових полігонів;
- Використовується для надання об'єктам складності, наприклад, вм'ятин, подряпин або складних візерунків.

Occlusion (Карта оклюзії):

- Текстура, яка визначає, де поверхня об'єкта перебуває у тіні через близькість інших об'єктів чи заглиблень;
- Додає реалістичності, затемняючи щілини та поглиблення, які природним чином блокують світло.

Height Map (Карта висот або Displacement Map):

- Текстура, що визначає фактичну висоту поверхні, створюючи справжній рельєф;
- На відміну від Normal Map, вона змінює геометрію об'єкта (хоч і не завжди на рівні сітки), роблячи поверхню об'ємнішою.

Detail Map (Деталізована карта):

- Додаткова текстура, яка додає дрібні деталі (наприклад, бруд, пісок, подряпини) до поверхні, особливо під час перегляду зблизька;
- Вона може використовуватися поверх Albedo, додаючи деталі до кольору та текстури.

Рисунок 3.6 – Приклад властивостей матеріалу в Inspector

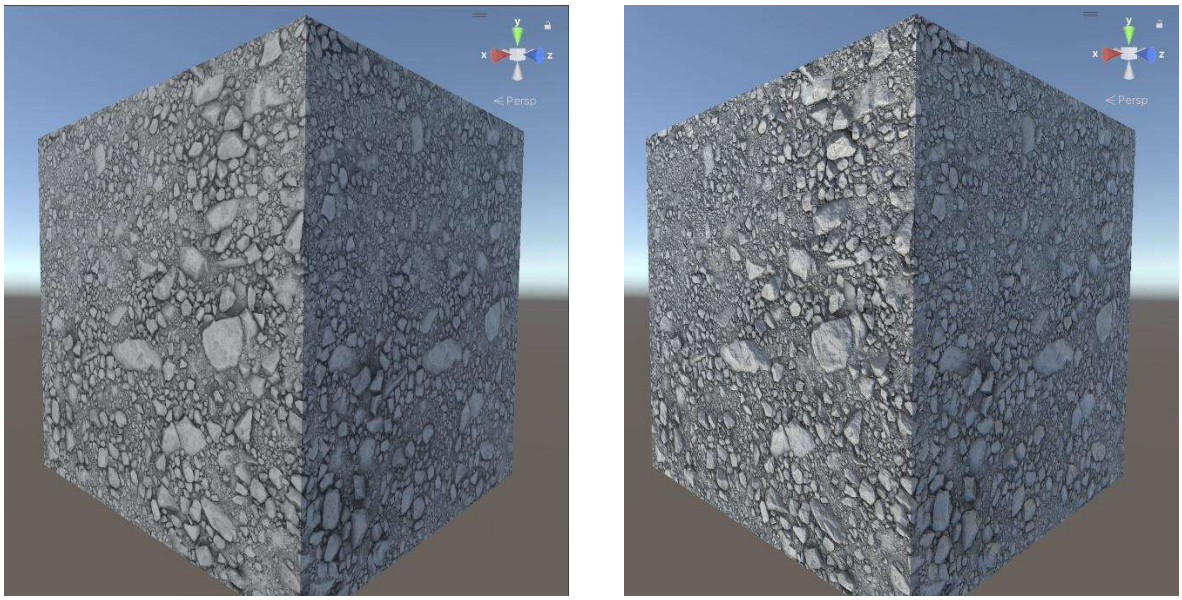
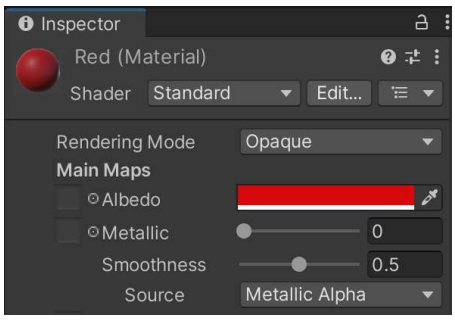


Рисунок 3.7 – Приклад ефекту від додавання Normal Map

Наприклад, необхідно створити матеріал у вигляді певного кольору, наприклад, червоного. Тоді маємо такі налаштування:



У Unity властивості Tiling (тайлінг, мозаїка) та Offset (зміщення) у матеріалах використовуються для керування відображенням текстури на поверхні об'єкта.

Tiling (Мозаїка/Тайлінг) – задає масштаб текстури, визначаючи скільки разів вона повторюватиметься (мозаїкою) по горизонталі (X) і вертикалі (Y) на поверхні об'єкта. Дозволяє створити безшовне повторення текстури та керувати її деталізацією. Наприклад, значення 2,2 для X і Y означатиме, що текстура повторюється вдвічі більше, ніж у вихідному розмірі, займаючи вдвічі менше простору.

Наприклад:

Tiling = (1,1)

Tiling = (3,3)

Offset (Зміщення) – зсув текстури в межах однієї плитки, що визначається значенням Tiling. Використовується для:

- Прокручування текстури – зміна значень Offset з часом дозволяє створити ефект руху, наприклад, для імітації поточної води або хмар, що рухаються;
- Налаштування видимої області – дозволяє вибрати конкретну частину текстури для відображення на об'єкті, не змінюючи її загальний розмір і масштабування.

Наприклад:

Offset = (0,0)

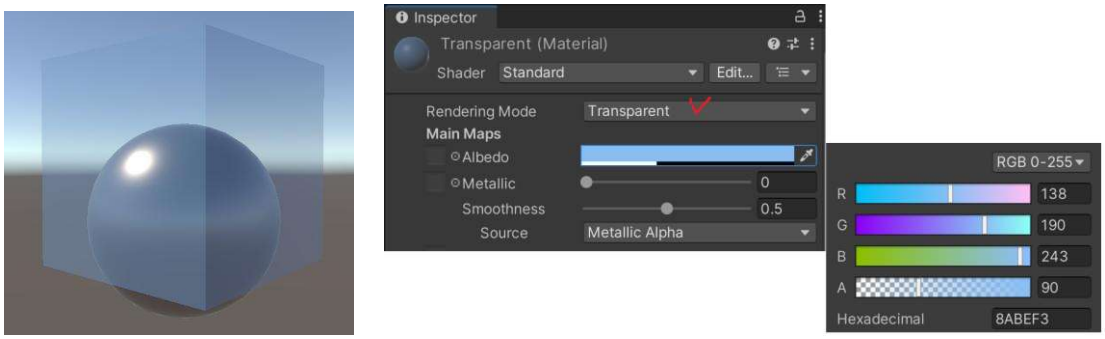
Offset = (1.7,1.5)

У Unity параметр *Rendering Mode* (Режим рендерингу) у властивостях матеріалу контролює, як матеріал відображатиме прозорість об'єкта, і має чотири основні варіанти:

- Opaque – матеріал буде повністю непрозорим, без ефекту прозорості;

– CutOut – застосовується альфа-тестування. Зображення буде або повністю непрозорим або повністю прозорим, залежно від значення параметра «Альфа відсікання» (Alpha Cutoff),
– Fade (напівпрозорий з можливістю повної прозорості) – об'єкт стає напівпрозорим, але може стати повністю прозорим,
– Transparent (напівпрозорий) – об'єкт стає напівпрозорим, дозволяючи бачити крізь нього, що підходить для ефектів, як скло.

Наприклад, зробимо матеріал скло:



У Unity 3D *фізичний матеріал* – це ресурс, що визначає властивості тертя та відскоку при зіткненні об'єктів, а також їхню поведінку при взаємодії (Assets → Create → Physic Material).

Тертя це величина, яка відповідає за запобігання тертю поверхонь один про одного. Ця величина важлива, якщо ви намагаєтеся створити нагромадження з якихось об'єктів.

Тертя буває двох видів, динамічним та статичним. Static friction використовується коли об'єкт перебуває у нерухомому стані. Воно не дасть такому об'єкту зрушити з місця без впливу зовнішніх сил. І в цей момент в дію набуває Dynamic Friction. У разі використання Dynamic Friction об'єкти будуть уповільнюватися при зіткненні один з одним.

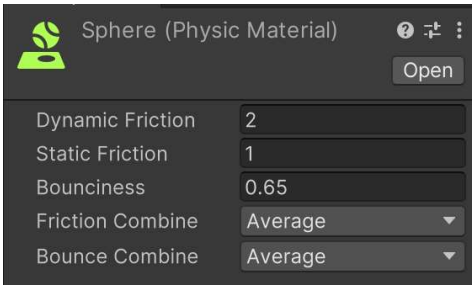


Рисунок 3.8 – Приклад властивостей фізичного матеріалу

Основні властивості фізичного матеріалу:

– Dynamic Friction (Тертя) – визначає, наскільки сильно чинитиме опір ковзанню зіткнення двох поверхонь. Зазвичай значення бувають у діапазоні від 0 до 1. Значення рівне 0 відповідає тертю як на льоду, тоді як значення рівне 1 означає слабке тертя, в результаті якого об'єкту буде складно рухатися без впливу зовнішніх сил.
– Static Friction (Тертя) – тертя, що використовується коли об'єкт лежить на поверхні. Зазвичай значення бувають у діапазоні від 0 до 1. Значення рівне 0 означає відсутність тертя, тоді як значення рівне 1 означатиме абсолютне тертя (тобто об'єктам по такій поверхні буде складно пересуватися).
– Bounciness (Відскок) – визначає, наскільки сильно об'єкт відскочить при зіткненні. Значення 0 – повна відсутність відскоку, 1 – ідеальний відскок без втрати енергії.

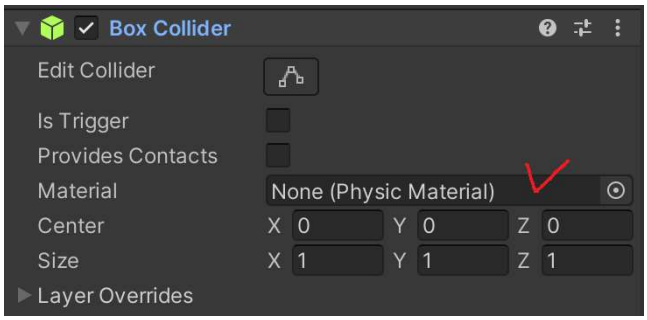
– Friction Combine (Комбінація тертя) - визначає, як комбінуватиметься тертя при зіткненні двох об'єктів з різними фізичними матеріалами. Можливі варіанти:
  - Average (Середнє): обчислює середнє значення тертя обох об'єктів.
  - Minimum (Мінімальний): Приймає найменше значення тертя.
  - Maximum (Максимальне): Приймає найбільше значення тертя.
  - Multiply: Примножує значення тертя обох об'єктів.
– Bounce Combine (Комбінація відскоку) – як комбінується пружність двох об'єктів, що стикаються. Вона підтримує самі режими, як і Friction Combine режим.

Коли два тіла контактують, до обох застосовується однаковий ефект пружності та тертя відповідно до обраного режиму.

Існує особливий випадок, коли два колайдери, що контактують, мають різні встановлені режими комбінування. У цьому конкретному випадку використовується функція з найвищим пріоритетом.

Порядок пріоритетів такий: Середнє < Мінімум < Множення < Максимум. Наприклад, якщо для одного матеріалу встановлено Середнє, а для іншого – Максимум, тоді буде використана функція комбінування Максимум, оскільки вона має вищий пріоритет.

Фізичний матеріал застосовується до колайдеру об'єкта, а не до об'єкта безпосередньо.



4. Rigidbody та Prefab

Rigidbody (Тверде тіло) дозволяють ігровим об'єктам взаємодіяти за допомогою фізики. Для реалістичного переміщення твердих тіл на останні впливають сила обертання та інші сили.

Тверді тіла можуть керуватися силами та обертанням.

Будь-який ігровий об'єкт повинен містити в собі тверде тіло, щоб бути схильним до гравітації, діяти відповідно до призначених шляхом скриптингу сил, або взаємодіяти з іншими об'єктами через фізичний двигун

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.

Оскільки компонент Rigidbody керує переміщенням об'єкта, до якого він прикріплений, вам не слід намагатися впливати на об'єкт із коду за допомогою зміни таких властивостей Transform, як position та rotation. Натомість вам слід застосовувати сили для того, щоб штовхати об'єкт і дозволити фізичному движку розрахувати результати.

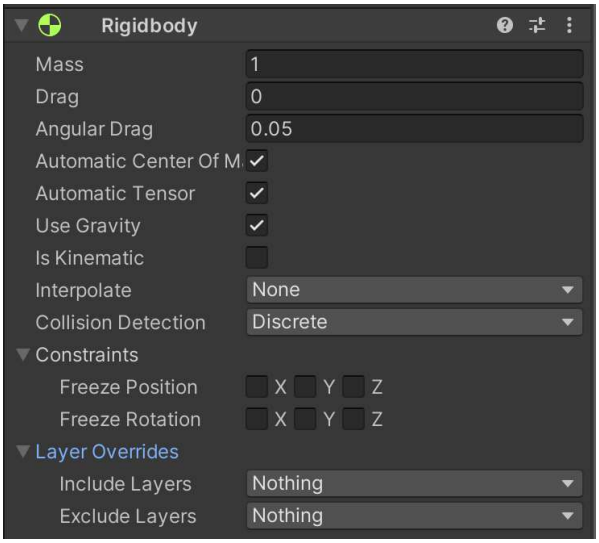


Рисунок 3.9 – Приклад компоненти Rigidbody

Розглянемо основні властивості:

Mass – маса об’єкта (за замовчуванням у кілограмах).

Drag – який повітряний опір виявляється на об'єкт, поки він переміщається під впливом цих сил. 0 означає відсутність опору, а нескінченність (infinity) відразу припиняє переміщення об'єкта.

Angular Drag – який повітряний опір виявляється на об'єкт, поки він обертається під впливом сили обертання. 0 означає відсутність опору. Врахуйте, що ви не можете зупинити обертання об'єкта шляхом встановлення його кутового опору (Angular Drag) в нескінченне (infinity) положення.

Центр мас є критично важливим параметром для фізичного двигуна, оскільки він визначає точку застосування сил і обертання об'єкта. Зміна центру мас безпосередньо впливає на те, як об'єкт реагуватиме на зіткнення, гравітацію та прикладені сили.

Властивість Automatic Center of Mass в компоненті Rigidbody в Unity визначає, чи центр мас об'єкта автоматично розраховуватиметься фізичним двигуном Unity.

Коли властивість Automatic Center of Mass увімкнена (встановлена в true), центр мас об'єкта автоматично обчислюється на основі форми та розподілу маси всіх колайдерів, прикріплених до GameObject. Це забезпечує реалістичну фізичну поведінку, оскільки центр мас відповідатиме геометричному центру об'єкта або його розподілу маси.

Use Gravity – при включенні на об'єкт діє гравітація.

Is Kinematic – при включенні, об'єкт не керуватиметься фізичним двигуном, і зможе керуватися лише за допомогою своєї трансформації.

Freeze Position – выборочно останавливает перемещение Rigidbody по мировым осям X, Y, Z.

Freeze Rotation – выборочно останавливает вращение Rigidbody вокруг оси X, Y, Z.

Unity 3D префаб (Prefab) – це об'єкт гри (GameObject), що містить заздалегідь налаштовані компоненти та властивості, який можна використовувати як шаблон для створення безлічі однакових екземплярів на сцені або під час виконання гри.

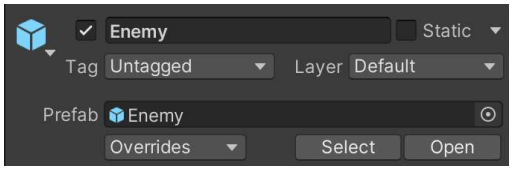
Префаби дозволяють швидко створювати складні об'єкти, застосовувати зміни до всіх копій одночасно і легко модифікувати їх, підвищуючи ефективність робочого процесу.

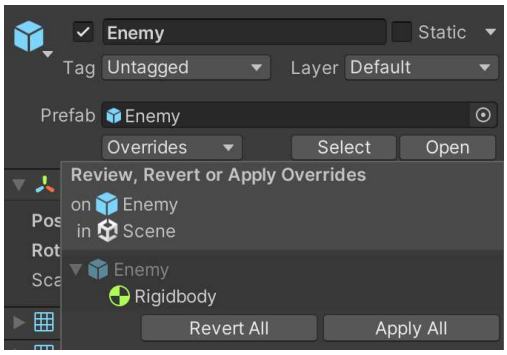
Щоб створити з об'єкта прифаб, все, що потрібно зробити, це перетягнути його в блок до асетів. Таким чином, у папці ассетів у нас з'являється прифаб об'єкта  . Цей прифаб – це копія об'єкта, що знаходиться поза сценою. Протягнувши цей прифаб на сцену, ми можемо створити нову копію на сцені.

Ми також можемо перемістити нашу нову копію об'єкта на сцені. Це дуже простий спосіб створити копію об'єктів. Ми можемо змінювати наш базовий об'єкт. Зміна цього об'єкта призведе до зміни всіх копій.

Тепер нам не потрібно міняти кожен з об'єктів копій чи шукати усі наші об'єкти на сцені. Виділяти їх разом, робити зміни. Все, що потрібно зробити, це змінити базовий об'єкт. І ці зміни будуть застосовані до всіх його копій.

Локальні зміни префабу знаходяться у Overrides.





Вкладені префаби (Nested Prefabs ) в Unity дозволяють створювати складні об'єкти, збираючи їх з більш простих префабів, при цьому зміни в одному префабі автоматично застосовуються до всіх його екземплярів та вкладених частин, що підвищує гнучкість та спрощує підтримку проекту.

Це потужний інструмент для організації проекту, що дозволяє розбивати роботу на префаби, уникати конфліктів і створювати бібліотеки елементів, що перевикористовуються, таких як будівлі, елементи інтерфейсу або набори ворогів.

Prefabs Variants (Варіанти префабів ) у Unity 3D – це функція, яка дозволяє створювати нові префаби, що успадковують властивості від базового префабу, але з деякими перевизначеними характеристиками. Це спрощує створення різних версій одного і того ж ігрового об'єкта, наприклад різних типів ворогів або предметів, зберігаючи при цьому зв'язок з вихідним префабом для полегшення оновлень.

5. Анімація

Анімація в Unity – це потужний інструмент, який дозволяє пожвавити ваші ігрові об'єкти та персонажів.

У Unity анімація реалізується за допомогою різних інструментів та компонентів, таких як Animation Window, Animator Controller та скрипти на C#.

Перш ніж розпочати створення анімації, переконайтеся, що у вас є об'єкт, який ви хочете анімувати. Помістіть його у сцену Unity. Якщо у вас немає готового об’єкту, ви можете створити його з нуля або імпортувати із зовнішнього джерела. Важливо, щоб об'єкт був правильно налаштований та готовий до анімації.

Відкрийте вікно Animation, вибравши Window → Animation → Animation. Це вікно дозволить вам створювати та редагувати анімації для вибраного об'єкта.

Animation Window надає зручний інтерфейс для роботи з ключовими кадрами та часовою шкалою. Ви можете додавати, видаляти та змінювати ключові кадри, а також налаштовувати плавні переходи між ними.

Рисунок 3.10 – Приклад вікна Animation Window

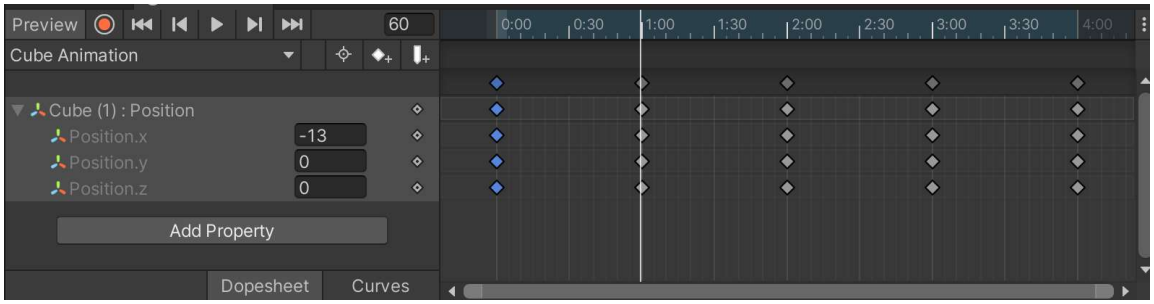
Виберіть об'єкт у сцені та натисніть кнопку Create у вікні Animation. Дайте ім'я нової анімації та збережіть її в папці проекту. Тепер можна почати додавати ключові кадри.

Ключові кадри визначають стан об'єкта певні моменти часу. Переміщуйте повзунок часу та змінюйте властивості об'єкта (позицію, обертання, масштаб тощо), щоб створити ключові кадри. Unity автоматично створить плавні переходи між ними. Ключові кадри дозволяють задавати точні параметри для кожного моменту часу, що забезпечує високий рівень контролю за анімацією.

Для запису анімації натискаємо кнопку  .

Встановлюємо моменти часу на timeline. В кожному моменту часу змінюємо, наприклад місцеположення об’єкту на сцені. Unity автоматично додасть необхідні властивості до вікна Animation Window.

Наприклад:



При перегляді анімації в режимі (закладка Dopesheet) анімовані значення для кожної властивості відображаються лише у вигляді лінійних доріжок, проте в режимі Curves (закладка Curves) ви можете бачити значення властивостей, що змінюються, візуалізовані у вигляді ліній на графіку.

У режимі Curves (Криві) мають кольорові індикатори кривих, кожен колір яких становить значення для однієї з вибраних властивостей у списку властивостей. Крива анімації має кілька ключів, які є контрольними точками, якими проходить крива. Вони візуалізуються у Редакторі кривих як маленьких ромбів на кривих.

Кадр, у якому одна або кілька показаних кривих мають ключ, називається ключовим кадром.

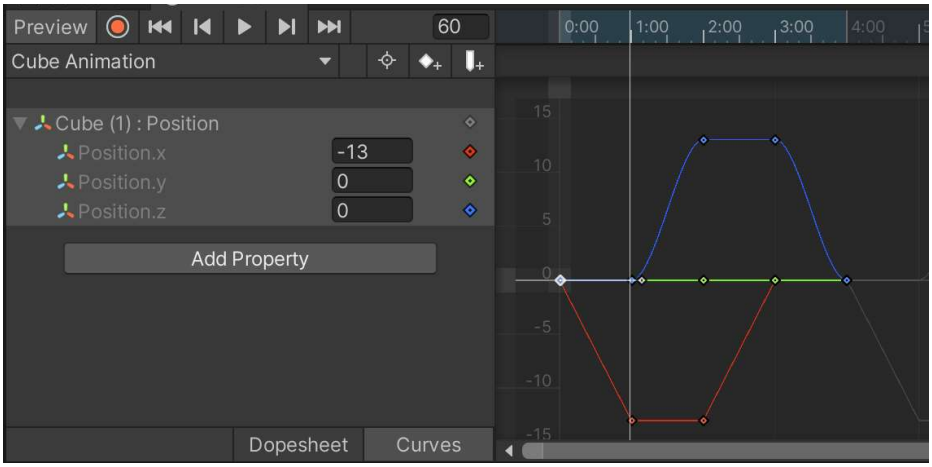
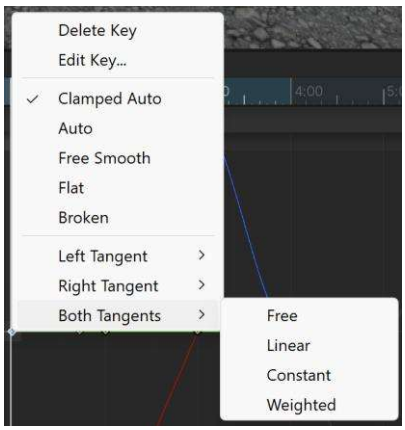


Рисунок 3.11 – Приклад кривих в закладці Curves

Термін «режим тангенсу» стосується алгоритму, який автоматично налаштовує маркери тангентсу, коли ви змінюєте час або значення ключового кадру.



Виберіть тип тангенсу, щоб мати кращий контроль над формою та нахилом кривої анімації до та після ключового кадру. Термін «тип тангенсу» стосується алгоритму інтерполяції, який визначає форму кривої анімації між ключовими кадрами. Термін «інтерполяція» стосується оцінки значень між двома відомими точками.

Може приймати такі значення:

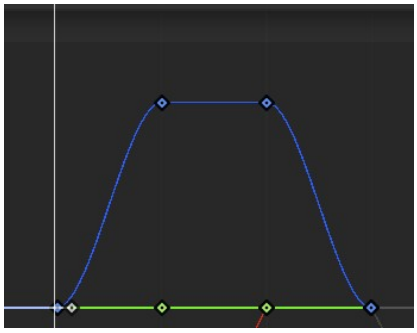
Free – виберіть цей тип тангенсу, щоб вільно обертати лівий маркер тангенсу (Left Tangent), правий маркер (Right Tangent) або обидва маркери тангенсу (Both Tangents). Якщо вибрати «Free», об’єднані маркери тангенсу розриваються та режим тангенсу встановлюється на «Broken».

Linear – виберіть цей тип тангенсу, щоб намалювати пряму лінію до ключового кадру (Left Tangent), від ключового кадру (Right Tangent) або як до, так і від ключового кадру (Both Tangents). Вибір цього типу тангенсу також розриває об’єднані маркери тангенсів та режим тангенсу встановлюється на «Broken».

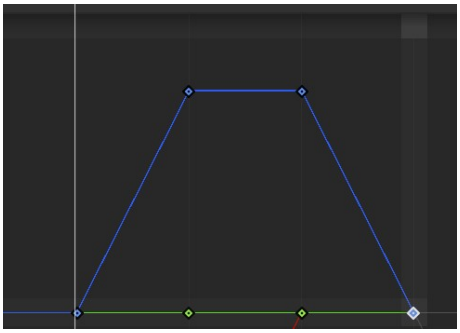
Constant – виберіть цей тип тангенсу, щоб зберегти постійне значення від останнього ключового кадру до цього ключового кадру (Left Tangent), від цього ключового кадру до наступного ключового кадру (Right Tangent) або обох (Both Tangents). Вибір цього типу тангенсу також розриває об’єднані маркери тангенсу та режим тангенсу встановлюється на «Broken».

Weighted – виберіть цей тип тангенсу, щоб використовувати зважений тангенс для зміни форми та нахилу кривої анімації перед ключовим кадром (Left Tangent), після ключового кадру (Right Tangent) або як до, так і після ключового кадру (Both Tangents).

Наприклад, ми бажаємо змінити вид кривих в ключах. Для цього обираємо ключ, натискаємо праву кнопку миші та обираємо, наприклад, Both Tangents → Linear. Якщо так виконати для всіх ключів, наприклад синьої кривої, то в результаті будемо мати:

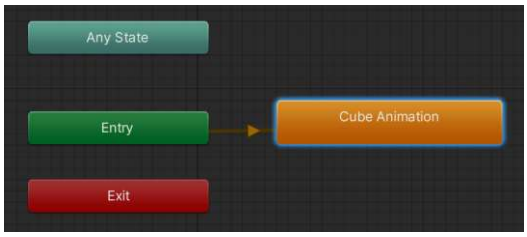


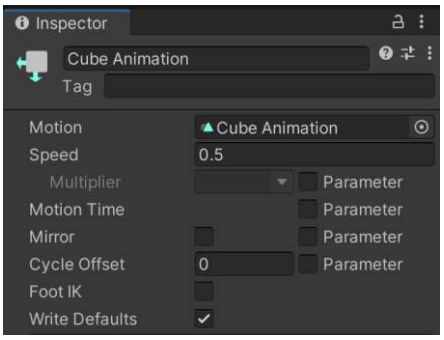
до



після

Збільшити або зменшити швидкість анімації можна вибравши у вікні Window → Animation → Animator відповідну анімацію та у вікні Inspector змінити властивість Speed.





Контрольні питання

1. Що таке сцена та для чого вона призначена?
2. З яких ассетів складається папка Assets?
3. Що таке GameObject?
4. Для чого використовується компоненти?
5. Що робить компонент Rigidbody?
6. Що робить компонент Collider?
7. Для чого використовується матеріал?
8. Що таке Prefab?
9. Як створити анімацію?

Лекція 4. Скрипти в Unity

1. Скрипти в Unity.

1. Скрипти в Unity

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.

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.

Скрипт взаємодіє із внутрішніми механізмами Unity за рахунок створення класу, успадкованого від вбудованого класу, званого MonoBehaviour. Клас MonoBehaviour в Unity - це базовий клас, від якого успадковуються всі скрипти користувача. Він надає доступ до основних функцій Unity, таких як керування ігровими об'єктами, обробка подій, взаємодія з фізикою, а також керування життєвим циклом об'єктів.

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 GameObject, 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.

При створенні нового скрипту на C#, Unity створює файл з розширенням cs та наступним вмістом:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MyScript : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

Фактично можна змінити шаблон сценарію, який Unity використовує під час створення нового сценарію C#. Файл шаблону сценарію, який ми збираємося змінити, називається 81-C# Script-NewBehaviourScript.cs.txt; він знаходиться в папці C:\ProgramFiles\Unity\Hub\Editor\ 2022.2.1f1\Editor\Data\Resources\ScriptTemplates.

Щоб змінити шаблон, запустить Visual Studio Community 2022 як адміністратор і відкрийте цей файл. Для цього запустіть Visual Studio та виберіть Continue without code (Продовжити без коду) внизу праворуч. Після запуску виберіть File → Open → File («Файл»

→ «Відкрити» → «Файл»), перейдіть до файлу, який потрібно редагувати, і двічі клацніть назву файлу.

Щоб зберегти зміни вибираємо File → Save 81-C# Script-NewBehaviourScript.cs.txt As .... У вікні для збереження клацніть стрілку вниз праворуч від кнопки Save («Зберегти») у нижньому правому куті та виберіть Save with Encoding... («Зберегти з кодуванням...») У діалоговому вікні Advanced Save Options («Додаткові параметри збереження») змініть Line Endings на Windows (CR LF) і натисніть кнопку «ОК».

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.

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.

Розглянемо інші функції в Unity.

Функції	Опис
Update	викликається перед малюванням кадру та перед розрахунком анімації
FixedUpdate	викликається перед кожним оновленням фізичних даних. Т.к. оновлення фізики та кадру відбувається не з однаковою частотою, то ви отримаєте більш точні результати від коду фізики
LateUpdate	можливість внести додаткові зміни у момент, коли у всіх об'єктів у сцені відпрацювали функції Update та FixedUpdate та розрахувалися всі анімації.
Start	викликається до оновлення першого кадру чи фізики об'єкту.
Awake	викликається для кожного об'єкта в сцені під час завантаження сцени. Для різних об'єктів функції Start та Awake і викликаються в різному порядку, всі Awake будуть виконані до першого виклику Start.
OnGUI	У Unity є система для відтворення елементів керування GUI поверх того, що відбувається в сцені, і реагування на кліки по цих елементах. Цей код обробляється дещо інакше, ніж звичайне оновлення кадру, тому він повинен бути поміщений у функцію OnGUI, яка буде періодично викликатися.
OnMouseOver, OnMouseDown	дозволяє скрипту реагувати на дії з мишею користувача
OnCollisionEnter, OnCollisionStay OnCollisionExit	будуть викликані на початок, продовження та завершення контакту (зіткненнях) з об'єктом

OnTriggerEnter, OnTriggerStay OnTriggerExit	будуть викликані, коли колайдер об'єкта налаштований як Trigger (тобто цей колайдер просто визначає, що щось перетинає і не реагує фізично). Ці функції можуть бути викликані кілька разів поспіль, якщо виявлено більше одного контакту під час оновлення фізики, тому в функцію передається параметр, що надає додаткову інформацію про зіткнення (координати, "особистість" вхідного об'єкта і т.д.).

Unity дозволяє змінити значення змінних скриптів у запущеній грі. Це дуже корисно, щоб побачити ефекти від змін одразу ж, без встановлення та перезапуску. Коли програвання закінчується, значення змінних змінюються в цьому стані, яке вони мали до натискання кнопки Play.

Атрибути (Attributes) – це маркери, які можуть бути розміщені перед класом, властивостями або функціями в скрипті, щоб забезпечити особливу поведінку.

Наприклад, *атрибут HideInInspector* може бути доданий перед оголошенням властивостей для попереднього відображення відображення цих властивостей в інспекторі, навіть якщо він публічний. В C# він поміщається між квадратними скобами.

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*.

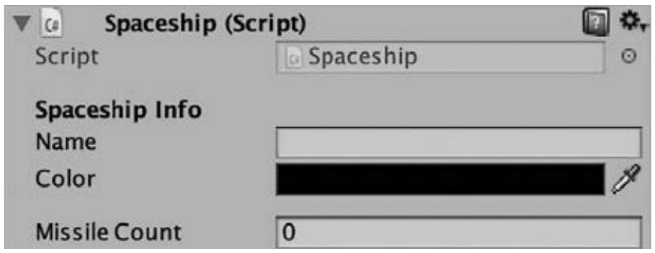
Unity створює мітку у вікні Inspector шляхом додавання пробілу скрізь, де в імені змінної зустрічається велика літера. Однак це виключно для відображення, і ви повинні завжди в коді використовувати ім'я змінної.

Атрибут Header, коли приєднується до поля, змушує Unity виводити напис над полем в інспекторі.

Атрибут Space діє аналогічно, але додає порожній простір. Обидва атрибути зручно використовувати для візуальної організації вмісту інспектора.

Наприклад:

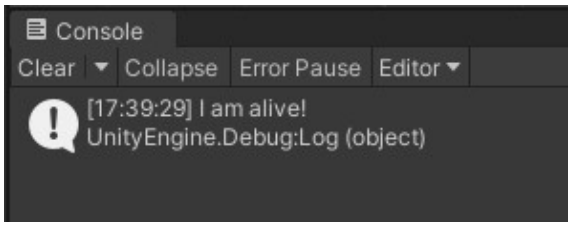
```
public class Spaceship : MonoBehaviour
{
[Header("Spaceship Info")]
public string name;
public Color color;

[Space]

public int missileCount;
}
```



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).



Найпростішим і найпоширенішим є випадок, коли скрипту необхідно звернутися до інших компонентів, приєднаних до того ж *GameObject*. Компонент насправді є екземпляром класу, тому першим кроком буде отримання посилання на екземпляр компонента, з яким ви хочете працювати. Це робиться за допомогою функції *GetComponent*. Типово, об'єкт компонента зберігають в змінну, це робиться C# за допомогою наступного синтаксису:

```
void Start ()
{
    Rigidbody rb = GetComponent<Rigidbody>();
}
```

Як тільки ви маєте посилання на екземпляр компонента, ви можете встановлювати значення його властивостей, тих же, які ви можете змінити у вікні Inspector.

Немає причини, через яку ви не можете мати більше одного скрипту користувача, приєднаного до одного і того ж об'єкта.

Якщо вам потрібно звернутися до одного скрипту з іншого, ви можете використовувати, як завжди, GetComponent, використовуючи при цьому ім'я класу скрипту (або ім'я файлу), щоб вказати, який тип Компоненту вам потрібен.

Якщо ви спробуєте вибрати Компонент, який не був доданий до Ігрового Об'єкта, GetComponent поверне null; виникне помилка порожнього посилання під час виконання (null reference error at runtime), якщо спробуєте змінити будь-які значення у порожнього об'єкта.

Скрипти можуть відстежувати інші об'єкти. Наприклад, ворог, що переслідує, повинен знати позицію гравця. Unity надає кілька шляхів отримання інших об'єктів.

Найпростіший спосіб знайти потрібний ігровий об'єкт – додати до скрипту змінну типу GameObject з рівнем доступу public. Змінну буде видно у вікні Inspector, тепер можна перетягнути об'єкт зі сцени або з панелі Hierarchy в цю змінну, щоб призначити його.

Наприклад:

```
public class Enemy : MonoBehaviour
{
    public GameObject player;

    void Start()
    {
        // Start the enemy ten units behind the player character.
        transform.position = player.transform.position -  Vector3.forward *
10f;
    }
}
```

Часто зручно знаходити об'єкти під час виконання, і Unity надає два основні способи зробити це.

Пошук дочірніх GameObjects. Іноді ігрова сцена використовує декілька ігрових об’єктів одного типу, таких як вороги, шляхові точки та перешкоди. Їх може знадобитися

відстежувати за допомогою певного сценарію, який контролює або реагує на них (наприклад, усі шляхові точки можуть бути доступні для сценарію пошуку шляху).

Використання змінних для зв’язування цих об’єктів GameObject є можливістю, але це робить процес проектування стомлюючим, якщо кожну нову точку маршруту потрібно перетягувати до змінної в сценарії. Так само, якщо шляхову точку видалено, видаляти посилання змінної на відсутній об’єкт GameObject – це незручність.

У подібних випадках часто краще керувати набором GameObjects, роблячи їх усіма нащадками одного батьківського GameObject.

Дочірні GameObjects можна отримати за допомогою батьківського компонента Transform (оскільки всі GameObjects неявно мають Transform)

Наприклад:

```
using UnityEngine;

public class WaypointManager : MonoBehaviour {
    public Transform[] waypoints;

    void Start() {
        waypoints = new Transform[transform.childCount];
        int i = 0;

        foreach (Transform t in transform) {
            waypoints[i++] = t;
        }
    }
}
```

Ви також можете знайти заданий дочірній об'єкт на ім'я, використовуючи функцію `Transform.Find`:

```
transform.Find("Gun");
```

Об'єкт або колекція об'єктів можуть бути знайдені за їх тегом, використовуючи функції GameObject.FindWithTag і GameObject.FindGameObjectsWithTag.

```
void Start() {
    player = GameObject.FindWithTag("Player");
    enemies = GameObject.FindGameObjectsWithTag("Enemy");
}
```

Деякі ігри мають постійну кількість об'єктів на сцені, проте зазвичай персонажі, скарби та інші об'єкти створюються та видаляються під час гри.

У Unity, ігровий об'єкт (GameObject) може бути створений, використовуючи функцію *Instantiate*, яка робить копію існуючого об'єкта.

Об'єкт з якого береться копія не повинен бути присутнім на сцені. Набагато частіше використовується префаб, який був перетягнутий на відкриту змінну (public variable) із файлів проекту на панелі Project.

Також є функція *Destroy*, яка знищить об'єкт після завершення завантаження кадру або опціонально після короткої паузи.

Контрольні питання

1. Скрипти в Unity спадкуються від якого класу?

2. Яка з функцій Start чи Update починає працювати першою?

3. Які атрибути можуть бути розміщені перед класом, властивостями або функціями в скрипті?

4. Що робить атрибут HideInInspector?

5. Що робить атрибут SerializeField?

6. Як вивести інформацію на консоль Unity?

7. Що робить функція GetComponent<>()?

8. Як програмно створити новий ігровий об’єкт?

9. Як програмно знищити ігровий об’єкт?

Лекція 5. UI. Інтерфейс користувача в Unity

1. Інтерфейс користувача
1. Об’єкт Canvas
2. Компонент Rect Transform
3. Створення інтерфейсу під різні роздільні здатності екрану

1. Інтерфейс користувача

У сучасному світі відеоігор користувальницький інтерфейс (UI) відіграє ключову роль у сприйнятті та взаємодії гравця з ігровим світом. UI включає всі візуальні елементи, з якими гравець може взаємодіяти, такі як меню, кнопки, індикатори здоров'я, карти і багато інших елементів. Правильно розроблений інтерфейс може значно покращити досвід гравця, тоді як погано спроектований може призвести до фрустрації та втрати інтересу до гри.

Інтерфейс користувача – це сукупність засобів, за допомогою яких гравець взаємодіє з грою. Це не лише візуальні елементи, а й їхня функціональність. Хороший UI має бути інтуїтивно зрозумілим, легко сприйманим та забезпечувати зручний доступ до всіх необхідних функцій.

Зручність гри багато в чому залежить від якості інтерфейсу користувача. Якщо гравець не може швидко зрозуміти, як керувати персонажем або як використовувати певні функції, це може спричинити негативний досвід. Ось кілька ключових аспектів, які підкреслюють важливість UI:

- Інтуїтивність – гравці повинні легко розуміти як використовувати інтерфейс без необхідності вивчати складні інструкції;
- Естетика – візуальна привабливість UI може значно збільшити загальне враження від гри;
- Функціональність – інтерфейс повинен забезпечувати доступ до всіх необхідних функцій гри без зайвих зусиль;
- Зворотній зв'язок – елементи UI повинні надавати гравцеві необхідну інформацію про його дії та стан гри.

Добре спроектований інтерфейс дозволяє гравцям легко виконувати дії, такі як переміщення персонажа, використання предметів та взаємодія з навколишнім світом. Наприклад, в іграх з відкритим світом наявність зручного меню швидкого доступу може спростити управління інвентарем.

Елементи UI, такі як індикатори здоров'я, міні-карти, допомагають гравцеві залишатися в курсі того, що відбувається в грі. Це дозволяє приймати більш обґрунтовані рішення та покращує загальний ігровий процес.

Дизайн інтерфейсу користувача також може сприяти створенню унікальної атмосфери гри. Наприклад, у хорор-іграх елементи UI можуть бути стилізовані під старовинні або пошкоджені екрани, що посилює почуття страху.

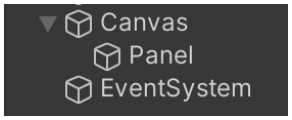
Погано спроектований інтерфейс може викликати у гравців почуття фрустрації. Наприклад, якщо кнопки занадто дрібні або розташовані незручно, це може призвести до помилок та зниження задоволення від гри. Правильне розміщення елементів та їх розмір можуть значно покращити сприйняття гри.

2. Об’єкт Canvas

Всі UI елементи (картинки, кнопки, текст тощо) створюються в Unity всередині спеціального об'єкта – Canvas.

Canvas (полотно) – це область, всередині якої знаходяться всі елементи UI (інтерфейсу користувача). Полотно – це ігровий об'єкт (Game Object) з доданим до нього компонентом Canvas. Всі елементи UI повинні бути дочірніми цьому Canvas.

Коли ви створюєте новий елемент UI, такий як наприклад панель, використовуючи меню GameObject → UI → Panel, разом з ним автоматично створюється і Canvas, якщо до цього на сцені його ще не було. Елемент UI створюється дочірнім Canvas.



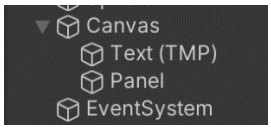
Область Canvas відображається у вигляді прямокутника у вікні Scene View. Це полегшує процес розташування елементів UI без потреби бачити ігрове вікно (Game View).

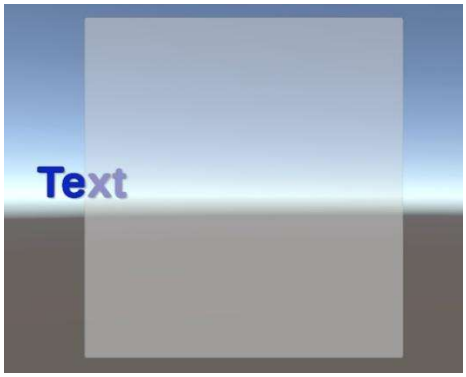
Разом із Canvas створюється Event System.

Event System (Система подій) – спосіб надсилання подій до об'єктів у програмі, заснований на введенні з клавіатури або миші; за допомогою дотиків або персональних пристроїв. Система складається з кількох компонентів, що працюють разом. Ця система за допомогою променів Unity визначає, що знаходиться під покажчиком.

Елементи UI на Canvas виникають у тому порядку, як вони розташовані в ієрархії. Перший дочірній елемент малюється першим, другий – за ним і таке інше. Якщо два елементи UI накладаються один на одного, доданий пізніше буде поверх того, що було додано раніше.

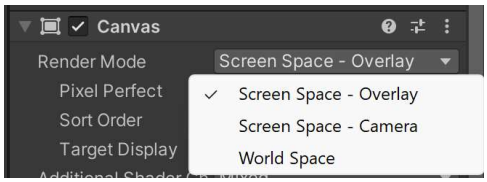
Наприклад:





Щоб змінити те, який елемент буде поверх інших, просто поміняйте місцями елементи в ієрархії шляхом перетягування. Порядком можна керувати за допомогою скриптингу, використовуючи такі методи компонента Transform: SetAsFirstSibling, SetAsLastSibling і SetSiblingIndex.

У полотна є *параметр Render Mode*, який визначає, де воно відображатиметься: у просторі екрана (screen space) чи ігрового світу (world space).



Режими відображення:

– Простір екрану – Перекриття (Screen Space – Overlay) – цей режим відображення поміщає елементи інтерфейсу на екран поверх сцени. Якщо змінюється розмір

екрана або його роздільна здатність, полотно автоматично прийме потрібний розмір разом із ним.

– Простір екрану – Камера (Screen Space – Camera) – це схоже на режим «Простір екрана – Накладання», але в цьому режимі рендерингу полотно розміщується на заданій відстані перед вказаною камерою. Елементи інтерфейсу користувача рендеряться цією камерою, а це означає, що налаштування камери впливають на зовнішній вигляд інтерфейсу користувача.

Якщо для камери встановлено режим «Перспектива», елементи інтерфейсу користувача будуть рендеритися з перспективою, а ступінь спотворення перспективи можна контролювати за допомогою поля зору камери. Якщо розмір екрана змінюється, роздільна здатність або кут розрізу камери змінюється, полотно також автоматично змінює розмір відповідно.

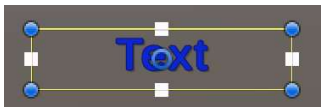
– Простір ігрового світу (World Space) – при цьому режимі відображення Canvas поводиться так само, як і будь-який інший об'єкт на сцені.

Розмір Canvas може бути заданий вручну за допомогою Rect Transform, а елементи інтерфейсу відображатимуться перед або за іншими об'єктами на сцені, залежно від їхнього тривимірного розташування.

Цей режим зручний для тих інтерфейсів, які передбачаються як частина ігрового світу.

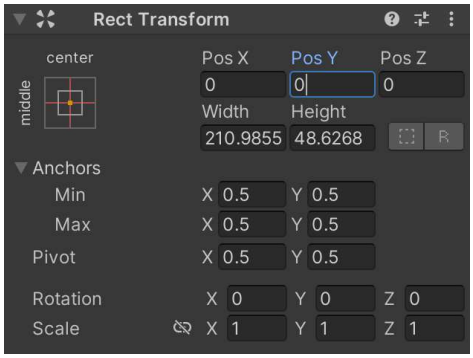
3. Компонент Rect Transform

Кожен елемент інтерфейсу користувача представлений у вигляді прямокутника для цілей макета. Цим прямокутником можна маніпулювати в режимі перегляду сцени за допомогою інструмента «Прямокутник» на панелі інструментів.



Інструмент «Прямокутник» використовується як для 2D-функцій Unity, так і для інтерфейсу користувача, і його можна використовувати навіть для 3D-об'єктів.

Rect Transform – це новий компонент перетворення, який використовується для всіх елементів інтерфейсу користувача замість звичайного компонента Transform.



Rect Transforms мають положення, обертання та масштаб, як і звичайні Transforms, але також мають ширину та висоту, які використовуються для визначення розмірів прямокутника.

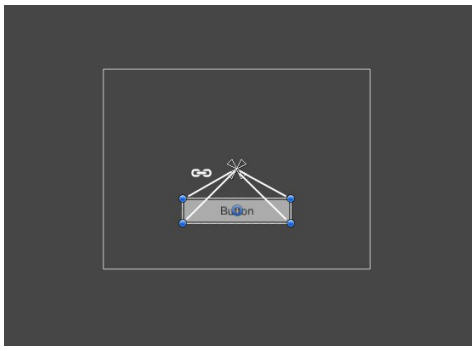
Зміни обертання, розміру та масштабу відбуваються навколо точки опори, тому положення точки опори впливає на результат обертання, зміни розміру або масштабування. Коли кнопка «Pivot» на панелі інструментів встановлено в режим «Pivot», точку опори прямокутного перетворення можна переміщувати в режимі Scene View.

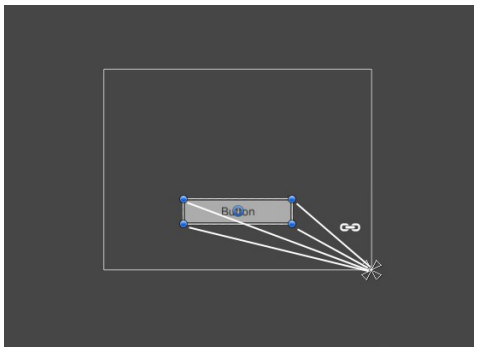
Rect Transforms включають концепцію макета, яка називається *прив'язками*. Прив'язки (Anchors) відображаються у вигляді чотирьох маленьких трикутних ручок у вікні перегляду сцени, а інформація про прив'язку також відображається в Інспекторі.

Якщо батьківський елемент прямокутного перетворення також є Rect Transform, дочірнє Rect Transform може бути прив'язане до батьківського Rect Transform різними способами.

Наприклад, дочірнє перетворення може бути прив'язане до центру батьківського елемента або до одного з кутів.

Наприклад:

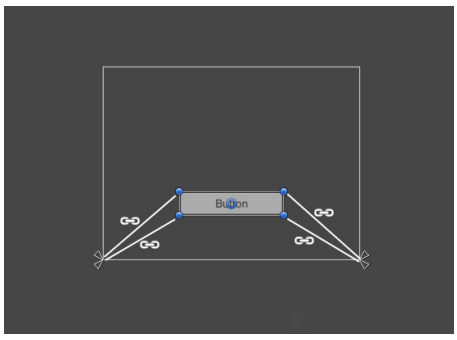




Прив'язка також дозволяє дочірньому об'єкту розтягуватися разом із шириною або висотою батьківського об'єкта.

Кожен кут прямокутника має фіксоване зміщення відносно відповідного йому прив'язувального елемента, тобто верхній лівий кут прямокутника має фіксоване зміщення відносно верхнього лівого прив'язувального елемента тощо.

Таким чином, різні кути прямокутника можуть бути прив'язані до різних точок батьківського прямокутника.



Положення якорів визначаються у частках (або відсотках) ширини та висоти батьківського прямокутника. 0,0 (0%) відповідає лівій або нижній стороні, 0,5 (50%) – середині, а 1,0 (100%) – правій або верхній стороні. Але якорі не обмежуються сторонами та серединою; їх можна прив’язати до будь-якої точки в межах батьківського прямокутника.

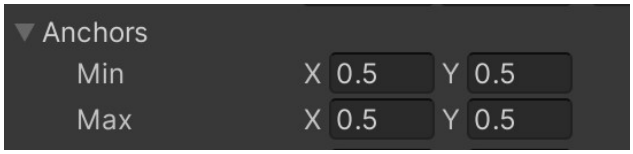
Ви можете перетягувати кожен з якорів окремо, або, якщо вони разом, ви можете перетягнути їх разом, клацнувши посередині між ними та перетягнувши. Якщо ви утримуєте клавішу Shift під час перетягування якоря, відповідний кут прямокутника переміститься разом з якорем.

В Інспекторі кнопку Anchor Preset button можна знайти у верхньому лівому куті компонента Rect Transform. Натискання цієї кнопки відкриває випадаюче меню «Початкові налаштування прив’язки». Звідси ви можете швидко вибрати один із найпоширеніших варіантів прив’язки.

Рисунок 5.1 – Приклад Anchor Preset

Ви можете прив’язати елемент інтерфейсу до боків або посередині батьківського елемента, або розтягнути його разом із розміром батьківського елемента. Горизонтальна та вертикальна прив’язка є незалежними.

Ви можете натиснути стрілку розгортання Anchors, щоб відобразити поля з номерами якорів, якщо вони ще не видно. Anchor Min відповідає нижньому лівому маркеру якоря в області перегляду сцени, а Anchor Max – верхньому правому маркеру.



Поля позиції прямокутника відображаються по-різному залежно від того, чи розташовані якорі разом (що створює фіксовану ширину та висоту), чи окремо (що призводить до розтягування прямокутника разом з батьківським прямокутником).

4. Створення інтерфейсу під різні роздільні здатності екрану

Сучасні ігри та програми часто потребують підтримки широкого спектру різних дозволів екрану, і особливо цієї можливості потребують інтерфейси цих ігор. Система створення інтерфейсів в Unity забезпечена рядом різних інструментів для реалізації цих можливостей, які можна комбінувати між собою масою різних способів.

Елементи інтерфейсу за замовчанням прив'язані до центру батьківського прямокутника. Це означає, що вони зберігають постійне зміщення щодо центру. Якщо з цією настройкою роздільна здатність була змінена під альбомне співвідношення сторін, кнопки можуть випасти зі своїх прямокутних областей, де вони спочатку мали бути розташовані.

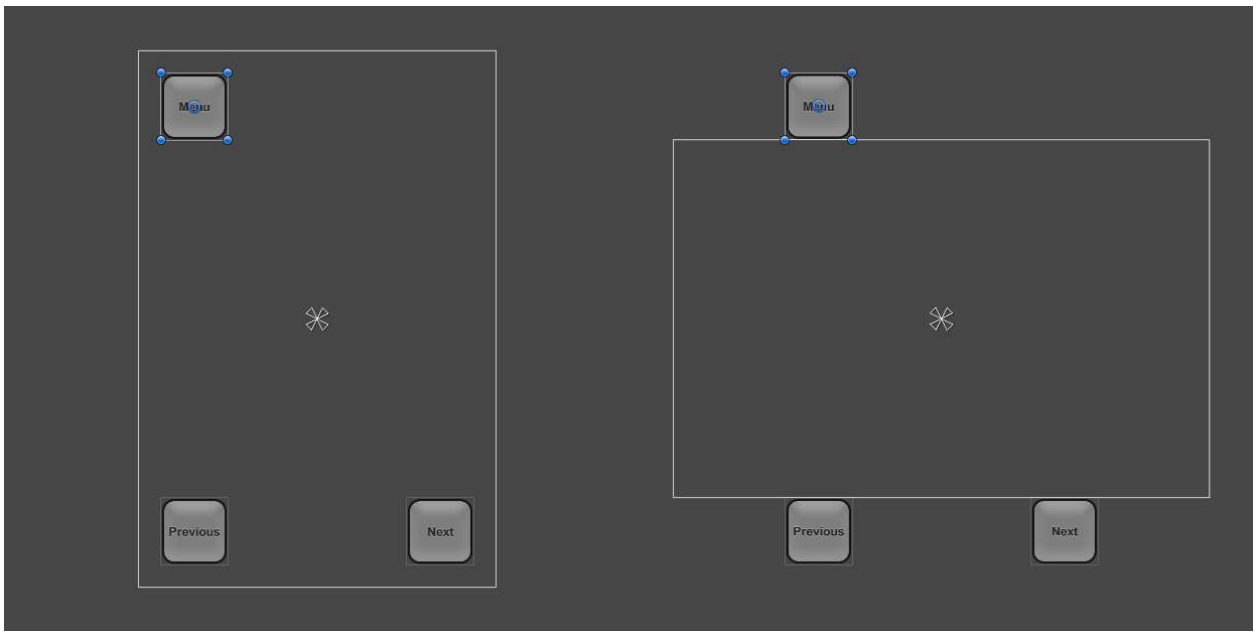


Рисунок 5.2 – Приклад зміни роздільної здатності

Одним із способів зберегти розташування кнопок в області екрану є зміна компонування таким чином, щоб місця розташування були пов'язані з їх відповідними кутами на екрані. Прив'язка лівої верхньої кнопки може бути також встановлена в лівому верхньому кутку при використанні в інспекторі списку Anchors Preset (набори прив'язок), або шляхом перетягування трикутних ручок прив'язок у видовому вікні сцени (Scene View). Так само наприклад, прив'язки для лівої нижньої і правої нижньої кнопок можуть бути виставлені в лівий нижній і правий нижній кут відповідно. Як тільки кнопки були прив'язані до своїх кутів, то при подальших змінах роздільної здатності екрану та співвідношень сторін вони зберігатимуть свою позицію щодо цих кутів.

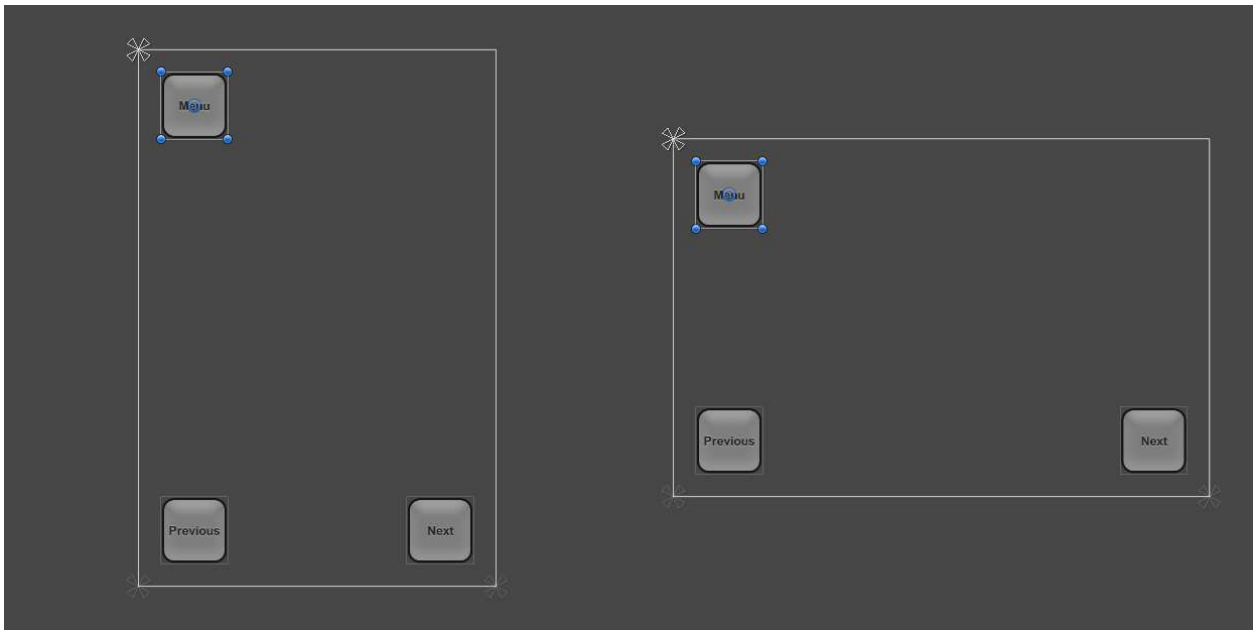


Рисунок 5.3 – Приклад зміни компонування

Коли роздільна здатність екрана змінюється на більшу і меншу щодо поточного, кнопки повинні зберігати своє початкове розташування щодо кутів, до яких вони прив'язані. Однак, зберігаючи свою оригінальну роздільну здатність, задану в пікселях, вони можуть ставати як більше так і менше, відповідаючи пропорціям поточної роздільної здатності

екрана. Все це може бути, а може і не бажано, залежно від того, як ви хочете, щоб ваш інтерфейс реагував на зміну роздільної здатності екрану.

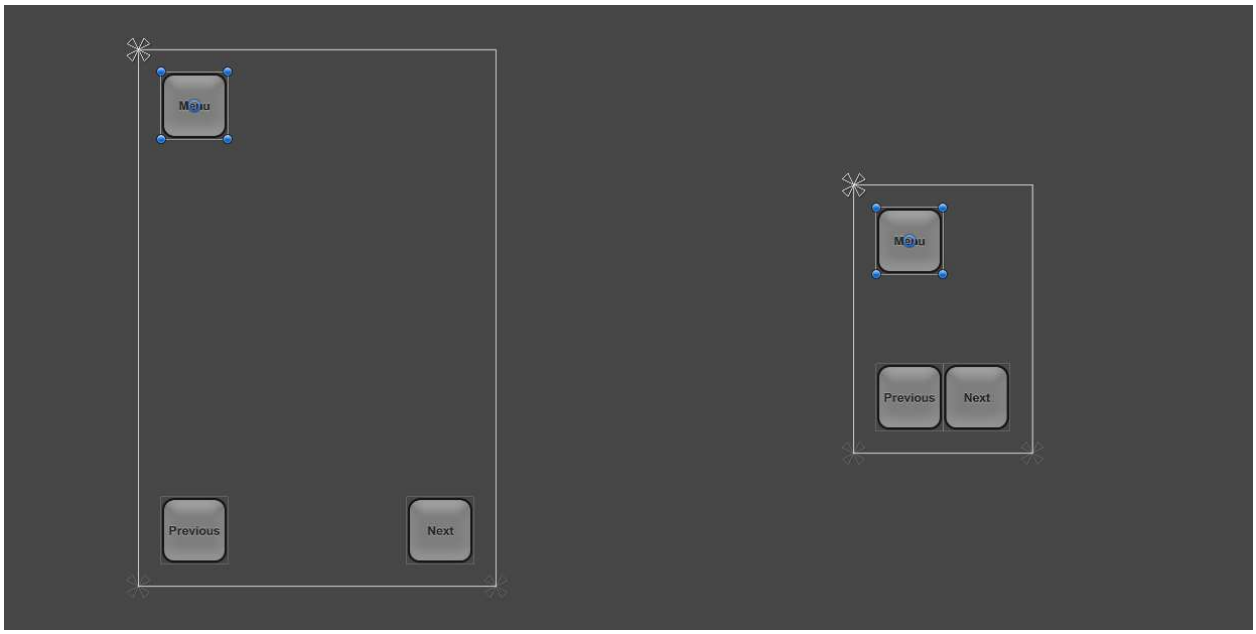


Рисунок 5.4 – Приклад зміни масштабу

У компоненті Canvas Scaler можна встановити його UI Scale Mode в Scale With Screen Size. У цьому режимі масштабування ви можете визначити, який дозвіл використовувати як базовий. Якщо поточна роздільна здатність більша або менша за базовий, фактор масштабування компонента Canvas встановлюється відповідно так, щоб всі елементи інтерфейсу масштабувалися у більшу або меншу сторону разом з роздільною здатністю екрана.

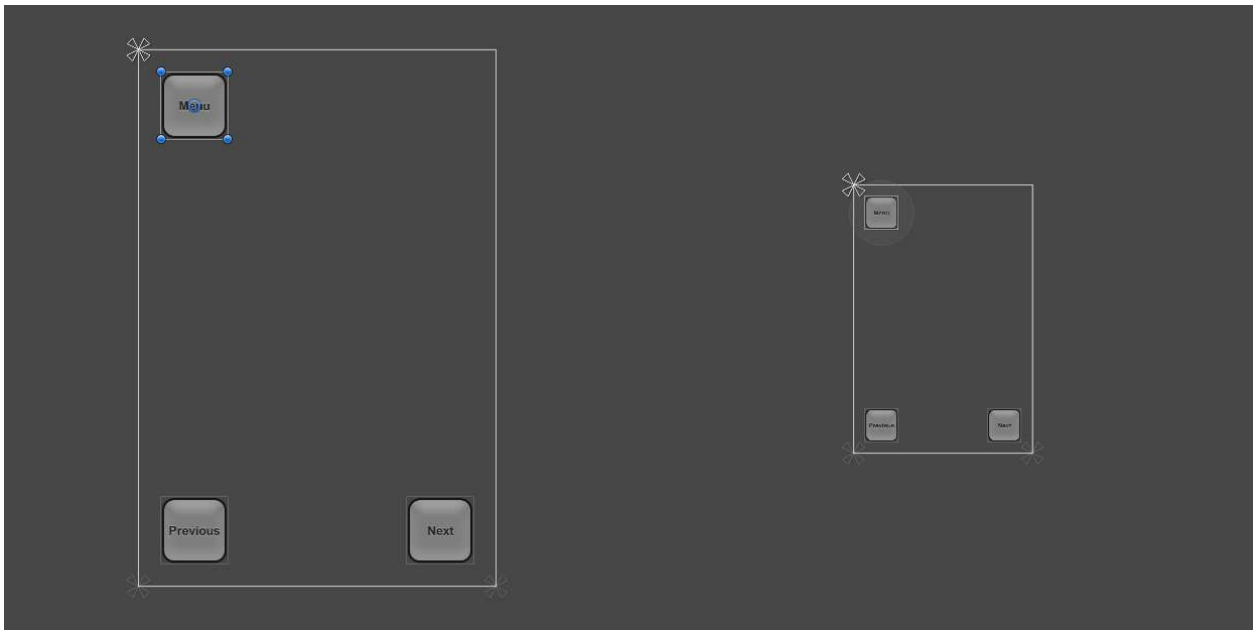


Рисунок 5.5 – Приклад роботи режиму Scale With Screen Size

Компонент Canvas Scaler має властивість під назвою Match, яка може прийняти значення, що дорівнює 0 (ширина), 1 (висота) або будь-яке значення, що лежить в межах між 0 і 1. За замовчуванням воно встановлено в 0, що означає те, що поточна ширина екрану відповідає базовій ширині базової роздільної здатності. Якщо властивість Match має

значення не дорівнює 0.5, воно порівнюватиме поточну ширину з базовою шириною, поточну висоту з базовою висотою, і вибере коефіцієнт масштабу близький і до того і до іншого дозволу.

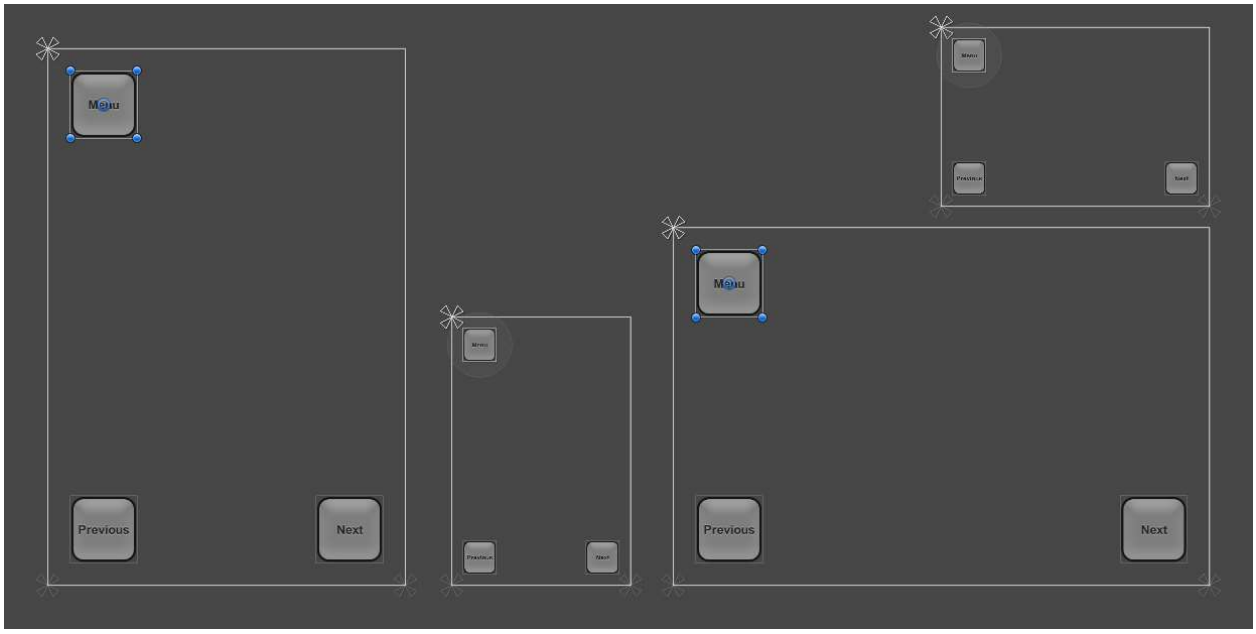


Рисунок 5.6 – Приклад роботи з властивістю Match

Контрольні питання

1. Що таке інтерфейс користувача (UI)?
2. Що залежить від якості інтерфейсу користувача?
3. Що робить об’єкт Canvas?
4. Що робить об’єкт Event System?
5. Як створити елементи UI?
6. Що робить компонент Rect Transform?
7. Що робить Pivot та Anchors?
8. Які є варіанти прив’язок?
9. Як створити інтерфейс під різні роздільні здатності екрану?

Лекція 6. UI. Створення головного меню гри

1. Візуальні компоненти в UI
2. Створення головного меню гри

1. Візуальні компоненти в UI

У Unity, коли йдеться про візуальні компоненти в UI, мають на увазі ті елементи, які відповідають за відображення та роботу користувача на екрані. Всі вони працюють усередині Canvas і використовують Canvas Renderer для малювання.

Основні візуальні UI-компоненти:

– Panel – простий прямокутний контейнер із Image. Зазвичай використовується як тло для інших елементів;
– Image – показує спрайти, іконки, фонові елементи;
– Text (або TextMeshPro - TMP Text) - відображає текст;
– Raw Image – показує текстуру безпосередньо (наприклад, потік з камери чи відео);
– Button – складовий UI-елемент, зазвичай містить Image + Text;
– Slider – повзунок (для гучності, прогресу тощо). Складається з фону, заповненої частини та «бігунка»;
– Scrollbar – вертикальна або горизонтальна смуга прокручування;
– Toggle – перемикач;
– Dropdown – список, що випадає для вибору з декількох опцій;
– Input Field – дозволяє користувачеві вводити текст із клавіатури.

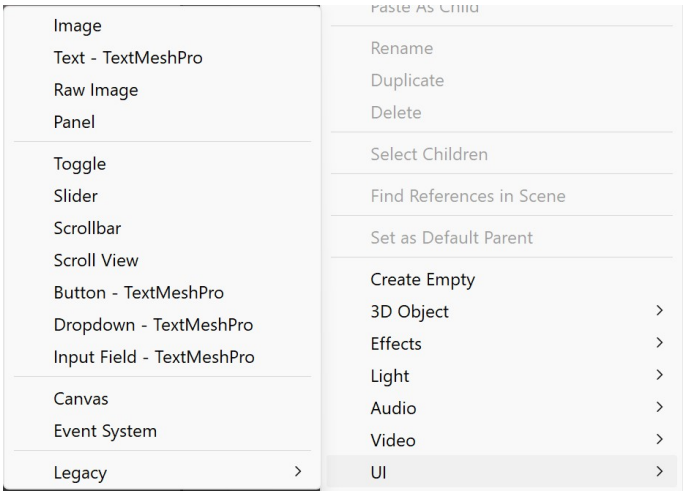


Рисунок 6.1 – Приклад меню для вибору візуальних UI-компонентів

Компонент Text, також відомий як Label, має область для введення тексту, який буде відображено. Є можливість задати шрифт, його стиль, розмір та здатність відображення Rich Text.

Існують опції для встановлення параметрів вирівнювання; налаштування горизонтального та вертикального переповнення, які керують поведінкою тексту, коли він не влазить по ширині або висоті у відведений йому прямокутник.

TextMeshPro – це потужний інструмент для роботи з текстом у Unity, який дозволяє покращити візуальну якість та гнучкість текстових елементів.

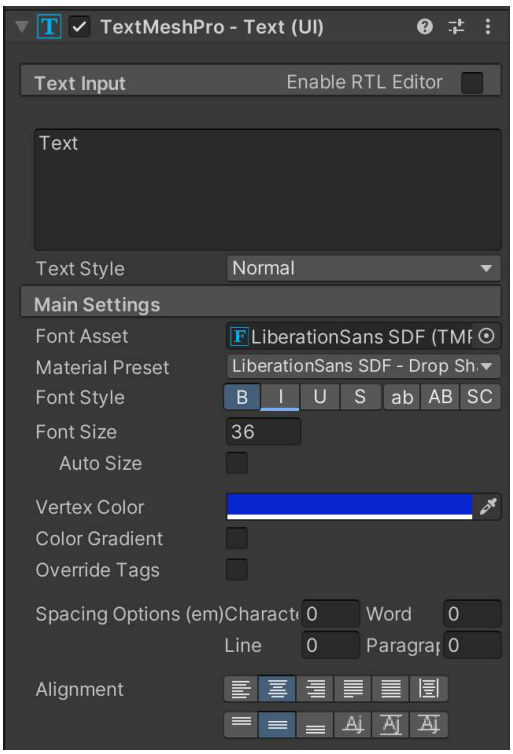


Рисунок 6.2 – Приклад компонента TextMeshPro

При першому використанні TextMeshPro у проекті Unity вам буде запропоновано імпортувати пакети TMP Essentials та Examples & Extras. Переконайтеся, що ви імпортували їх, щоб отримати доступ до всіх функцій.

Текст для елементів інтерфейсу користувача та текстових сіток може містити різні стилі та розміри шрифтів.

Форматований текст (Rich text) підтримується як для системи інтерфейсу користувача, так і для застарілої системи графічного інтерфейсу користувача. Класи Text, GUIStyle, GUIText та TextMesh мають налаштування Rich Text, яке вказує Unity шукати теги розмітки в тексті.

Функція Debug.Log також може використовувати ці теги розмітки для покращення звітів про помилки з коду. Теги не відображаються, але вказують на зміни стилю, які потрібно застосувати до тексту.

Система розмітки тексту в Unity була створена на основі HTML, однак сувора сумісність зі стандартним HTML не передбачається. Основна ідея полягає в тому, що фрагменти тексту можна укладати в пару тегів, що узгоджуються один з одним.

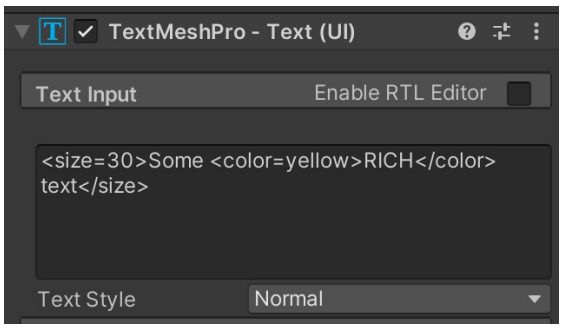
Тегами називаються фрагменти тексту, укладені в кутові дужки: <b>. Текст усередині дужок визначає ім'я тега (у разі b). Тег в кінці фрагмента (закриває) має те саме ім'я, що й тег на початку (відкриває), проте в ньому також міститься символ косої риси /.

```
We are <b>not</b> amused
We are not amused
```

Теги, які підтримуються:

– b – виділяє текст жирним шрифтом;
– i – виділяє текст курсивом;
– size – встановлює розмір тексту відповідно до значення параметра, заданого в пікселях;
– color – встановлює колір тексту відповідно до значення параметра.

Наприклад:

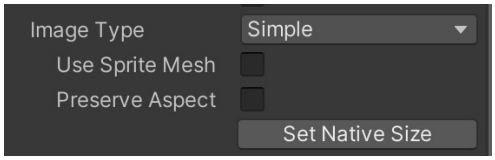


Компонент Image в Unity UI використовується для відображення картинки або кольорової області в інтерфейсі. Це один із найбільш базових візуальних елементів UI, і на його основі будуються багато інших елементів (кнопки, панелі, іконки).

Основні налаштування:

– Source Image – тут вибирається спрайт (наприклад, іконка, фон, картинка);
– Color – визначає колір або прозорість. Якщо є картинка, колір накладається як фільтр (Tint). Якщо картинки немає - малюється просто прямокутник потрібного кольору;
– Material – дозволяє призначити власний матеріал/шейдер;
– Image Type – визначає, яким чином доданий спрайт буде відображено, можливі варіанти:
  • Simple – масштабує весь спрайт рівноцінно.
  • Sliced – використовує поділ спрайту на 3x3 так, щоб масштабування не спотворювало кути, а розтягнута була лише центральна частина.
  • Tiled – схоже на Sliced, але замість розтягування центральної частини вона повторюється (заповнюється тайлами). Для спрайтів без будь-яких кордонів весь спрайт буде заповнений тайлами.
  • Filled – відображає спрайт так, як це робить Simple, за винятком того, що він заповнює спрайт, починаючи з певної точки у встановленому напрямку та кількості заданим методом.

Опция Set Native Size, доступная, когда выбран Simple или Filled, устанавливает изображению изначальный размер спрайта.



Елемент керування Button реагує на натискання користувача та використовується для ініціювання або підтвердження дії. Склададється з:

Розглянемо основні властивості кнопки:

– Interactable – чи прийматиме цей компонент вхідні дані?
– Transition – властивості, що визначають візуальну реакцію елемента керування на дії користувача.
– Navigation – властивості, що визначають послідовність елементів керування.

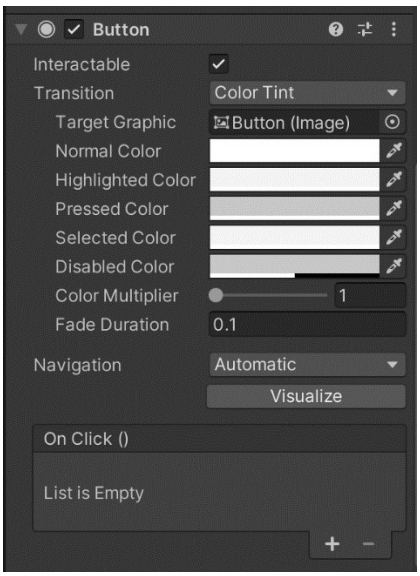
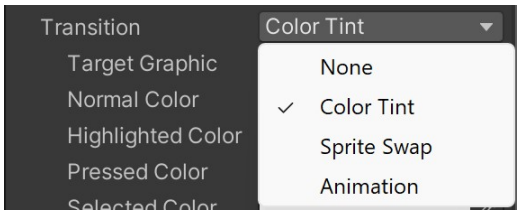


Рисунок 6.3 – Приклад компонента Button

Властивість Transition може приймати такі значення:

– None – цей параметр дозволяє кнопці взагалі не впливати на стан.
– Color Tint – змінює колір кнопки залежно від її стану. Можна вибрати колір для кожного окремого стану. Також можна встановити тривалість переходу між різними станами. Чим більше число, тим повільніше буде переходити між кольорами.
– Sprite Swap – дозволяє відображати різні спрайти залежно від поточного стану кнопки, спрайти можна налаштовувати.
– Animation – дозволяє створювати анімацію залежно від стану кнопки. Для використання переходу анімації має існувати компонент-аніматор.

Важливо переконатися, що рух кореня вимкнено. Щоб створити контролер анімації, натисніть «Згенерувати анімацію» (або створіть свій власний) і переконайтеся, що контролер анімації додано до компонента-аніматора кнопки.



При виборі значення Color Tint маємо такі властивості:

– Target Graphic – графіка, що використовується для компонента взаємодії.
– Normal Color – звичайний колір елемента керування.
– Highlighted Color – колір елемента керування, коли він виділений.
– Pressed Color – колір елемента керування, коли він натиснутий.
– Disabled Color – колір елемента керування, коли він вимкнений.
– Color Multiplier – це множить колір відтінку для кожного переходу на його значення. За допомогою цього ви можете створювати кольори більше 1, щоб зробити кольори (або альфа-канал) на графічних елементах, базовий колір яких менший за білий (або менший за повну альфа-канал).
– Fade Duration – час у секундах, необхідний для плавного переходу з одного стану в інший.

Властивість Navigation може приймати такі значення:

– Navigation – параметри навігації стосуються способу керування навігацією елементів інтерфейсу користувача в режимі відтворення.
– None – навігація за допомогою клавіатури відсутня. Також гарантує, що фокус не переходить у разі натискання/натискання на нього.
– Horizontal – горизонтальна навігація.
– Vertical – вертикальна навігація.
– Automatic – автоматична навігація.
– Explicit – у цьому режимі ви можете явно вказати, куди переміщується елемент керування для різних клавіш зі стрілками.
– Visualize – вибір опції «Візуалізація» надає візуальне представлення навігації, яку ви налаштували у вікні сцени.

2. Створення головного меню гри

Ігрове меню – це інтерфейс, який дозволяє керувати грою, роблячи доступними такі функції, як налаштування, запуск, пауза або вихід з гри, а також вибір режимів гри, керування персонажем та інші опції. Воно може бути основним (головним), коли гра тільки запущена, або меню паузи, яке з'являється під час ігрового процесу, щоб дати гравцеві можливість вибрати, продовжити гру, змінити налаштування або вийти.

Основні функції:

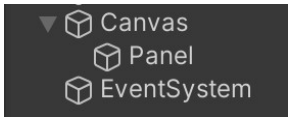
– Установки – дозволяє змінювати настройки гри, такі як мова, гучність, керування або налаштування графіки.
– Вибір гри – дозволяє розпочати нову одиночну або розраховану на багато користувачів гру, приєднатися до існуючого сервера або вибрати рівень.
– Пауза – тимчасова зупинка гри для виконання дій, не перериваючи ігровий процес.
– Інші опції – залежно від гри, може включати доступ до розділів з сюжетом, інвентарем персонажа, місіями або іншими можливостями.

Види ігрових меню:

– Головне меню – відображається під час запуску гри та містить основні опції, такі як початок гри, налаштування, вихід тощо.
– Меню паузи – з'являється під час гри і дозволяє поставити її на паузу, щоб змінити налаштування, зберегти гру або вийти.
– Внутрішньоігрове меню – може відображатися безпосередньо в ігровому просторі та містити специфічну інформацію або команди, наприклад меню карти або інвентарю.

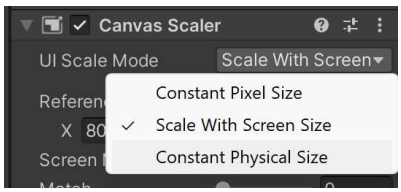
Для створення головного меню гри створимо нову сцену в проекті.

Додаймо на сцену Canvas (Полотно). Полотно – це ігровий об'єкт з доданим до нього компонентом Canvas. Всі елементи UI повинні бути дочірніми цьому Canvas.



В поле Render Mode об’єкту Canvas встановлюємо значення Screen Space – Overlay. Цей режим відображення поміщає елементи інтерфейсу на екран поверх сцени. Якщо змінюється розмір екрана або його роздільна здатність, полотно автоматично прийме потрібний розмір разом із ним.

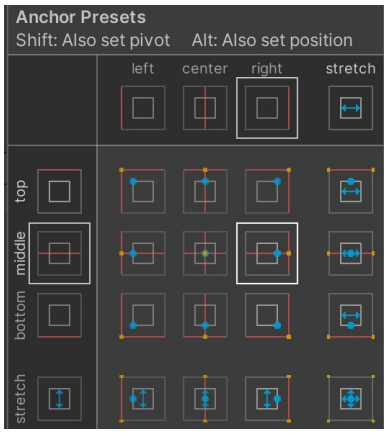
За замовчанням в компоненті Canvas Scaler в полі UI Scale Mode стоїть значення Constant Pixel Size. Якщо зміниться розмір екрана, то всі елементи що входять в Canvas не змінять свого розміру. Замінимо це значення на Scale with Screen Size.



Додаймо в Canvas компонент Panel. Panel це прямокутний контейнер із Image. Назвемо його MainMenu. Налаштуємо розмір компоненти, зробимо його не на весь екран. В компонент Image в поле Source image додаємо картинку фону.

Рисунок 6.4 – Приклад фону головного меню

Змінимо прив'язку панелі MainMenu відносно Canvas. Перетягнемо панель ближче до правої межі Canvas та виберемо Anchor Presets в значення middle-right.

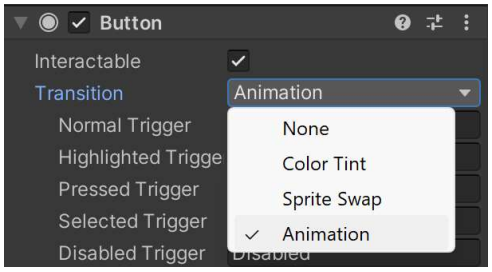


Створюємо дочірнім об'єктом до панелі MainMenu компонент Button. Компонент Button складається з компонента Image та Text. Додаємо до компонента Image зображення для представлення кнопки, в компоненту Text – запишемо текст Start та змінимо колір тексту та зробимо інші налаштунки. Змінимо назву кнопки в Canvas на StartButton.

Рисунок 6.5 – Приклад кнопки в головному меню

Щоб головне виглядало цікавіше додаймо анімацію при наведенні курсору на кнопку.

Поле Transition – це переходи для відтворення кнопки в різних станах. Може мати три варіанти: Color Tint, Sprite Swap та Animation. За замовчанням в компоненті Button в полі Trasition стоїть значення Color Tint. Color Tint, це коли вибираємо один спрайт для кнопки, але змінюємо лише кольори. Тобто колірний перехід має кожен стан. Можна налаштувати кольори для нормального стану, при наведенні курсору, при натисканні, для виділення та для неактивного стану у відповідних полях. Замінимо це значення на Animation. Значення Animation дозволяє створити повноцінний контролер анімації та встановити окремі анімації для різних станів.



При значення Animation натискаємо кнопку Auto Generate Animation та зберігаємо файл анімації до нової папки Animations. Якщо натиснутий на створений контроллер, він відкриється у вікні Animator.

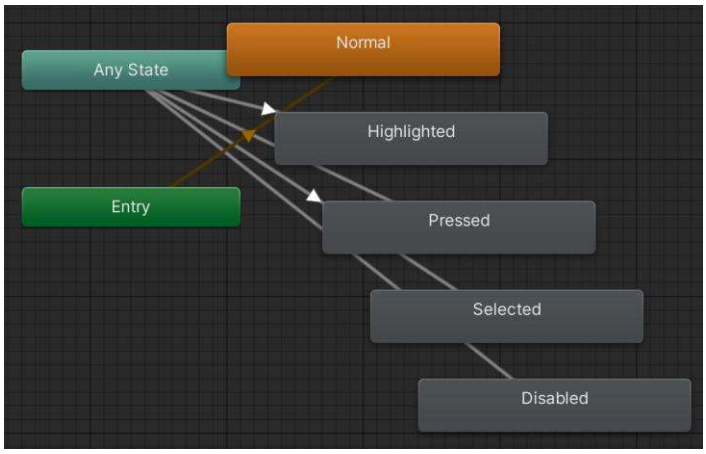
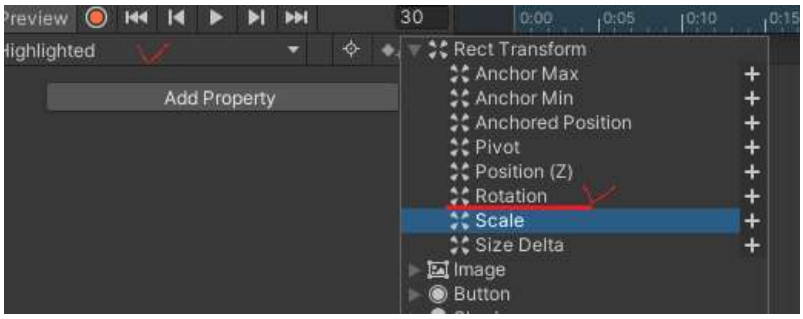


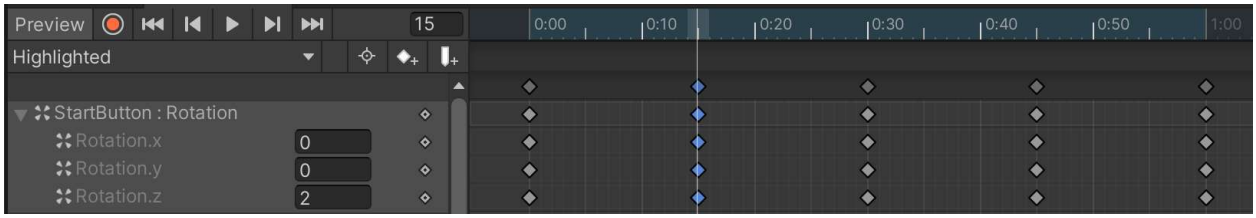
Рисунок 6.6 – Приклад анімаційного контролера у вікні Animator

Додаймо анімацію під час наведення на кнопку мишкою. Нехай трохи похитуватиметься. Виділяємо кнопку та відкриваємо віконце анімації Window → Animation → Animation.

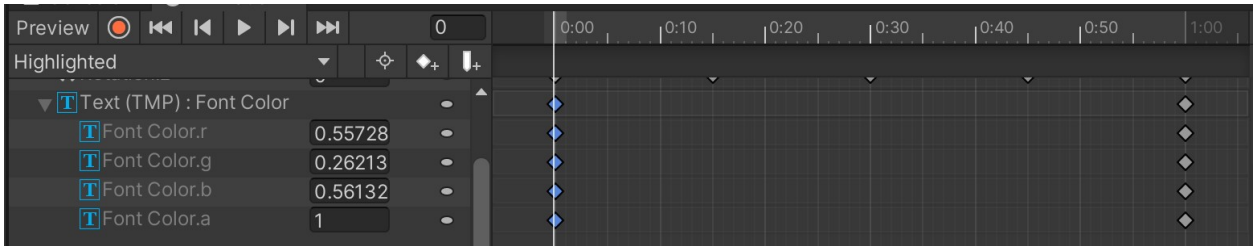
Обираємо Highlighted, натискаємо кнопку Add Property та обираємо Rect Transform → Rotation.



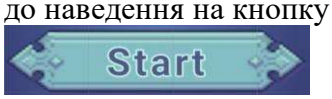
Розставляємо 5 keyframes, за часом 0:00, 0:15, 0:30, 0:45, 1:00. На час 0:00, 0:30 та 1:00 залишаємо Rotation.x, Rotation.y, Rotation.z рівними нулю, а на час 0:15 напишемо Rotation.z = 2, на час 0:45 напишемо Rotation.z = -2.



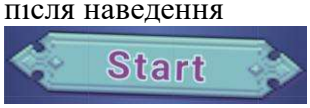
Також можна додати анімацію на колір тексту. В кнопці встановлюємо колір на текст, а у вікні анімації будемо його змінювати:



до наведення на кнопку мишкою



після наведення



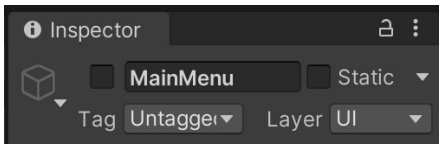
Зробимо 2 копії кнопки, що вийшла та розставимо по вертикалі на панелі. Змінимо текст на кнопках. В результаті отримаємо:

Рисунок 6.7 – Приклад початкового варіанта головного меню

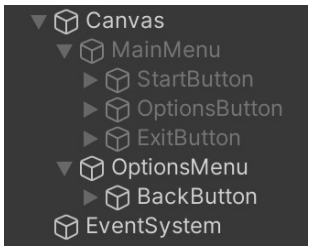
Отже маємо три кнопки: Start – при натисканні якої повинна починатися гра, Options – при натисканні якої повинна з’являтися додаткове меню для налаштувань та Exit – для виходу з програми.

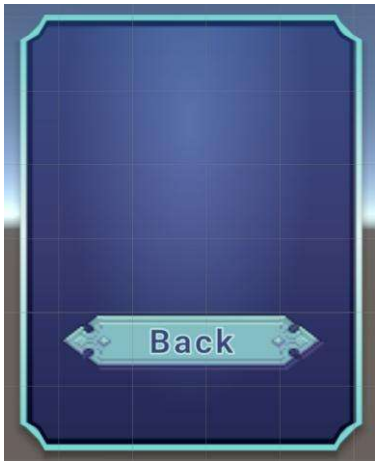
Створимо додаткове меню для налаштувань. В нашій версії воно буде мати тільки одну кнопку Back. Для цього виділяємо елемент MainMenu в ієрархії та робимо його

дублікат. Змінюємо ім’я дубліката на OptionsMenu. Для зручності тимчасово зробимо елемент MainMenu не активним.



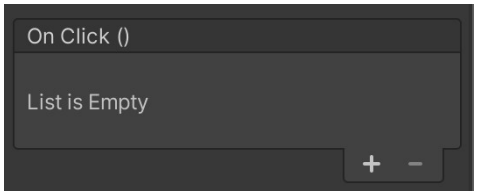
Якщо в ієрархії розкрити елемент OptionsMenu, то він має у підпорядкуванні три кнопки. Видалимо перші дві, а в останній кнопці змінимо текст на Back:

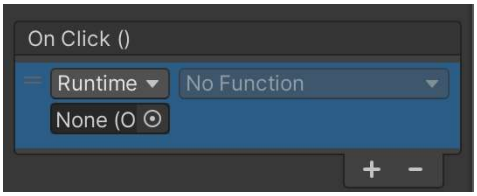




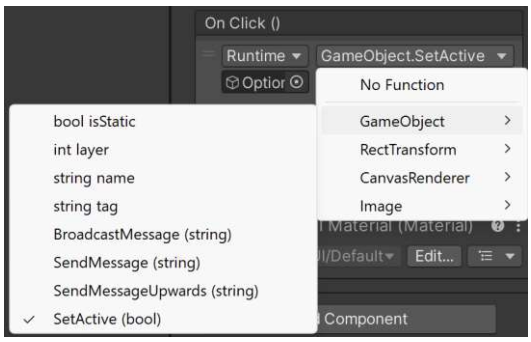
Отже, додаткове меню для налаштувань готово. Необхідно додати функціональність оскільки при натисканні кнопок нічого не відбувається.

Додаймо функціонал при натисканні кнопки Back, а саме елемент OptionsMenu повинен стати не активним, а елемент MainMenu активним. Для цього виберіть із компонента Button розділ «On Click()» і натисніть на плюс.



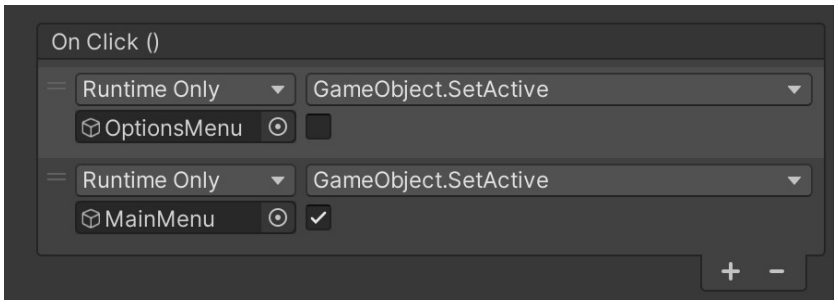


У нижне вікно перетягуємо об’єкт OptionsMenu, тоді справа в полі відкриються доступні функції над цим об’єктом. Обираємо GameObject → SetActive(bool)



Рядом з полем з об'єктом OptionsMenu створиться прапорець. Якщо обрати прапорець, то вікно буде активно (видимо), якщо ні – не активно. Для об'єкта OptionsMenu прапорець не обираємо.

В розділі «On Click()» можна створювати декілька дій одразу. В нашому випадку нам потрібно ще одна дія для показу вікна MainMenu. Зробимо цю дію аналогічно попердій. Тоді в розділі «On Click()» будемо мати наступне:



Повертаємось до головного вікна MainMenu. Виділяємо елемент OptionsMenu в ієрархії та робимо його не активним. Виділяємо елемент MainMenu в ієрархії та робимо його активним. Пропишемо схожі дії для кнопки OptionsButton.

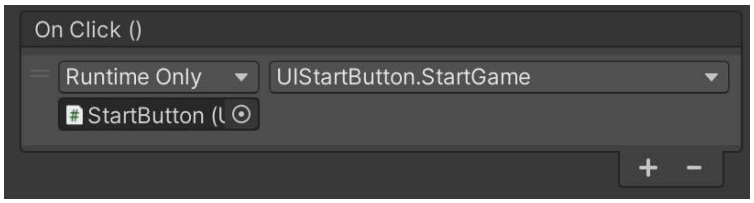
Якщо запустити проект, при натисканні кнопки Options з'явиться додаткове меню з однією кнопкою Back при натисканні на яку, повертається головне меню.

Залишається додати функціональність до кнопки Start. Створимо новий скрипт, наприклад UIStartButton, який має тільки один метод:

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class UIStartButton : MonoBehaviour
{
    public void StartGame()
    {
        SceneManager.LoadScene("Scene1");
    }
}
```

В метод LoadScene необхідно передати назву сцени з грою.

Додймо цей скрипт до кнопки Start. Переходимо в розділ розділі «On Click()» кнопки Start. У нижне вікно перетягуємо кнопку Start тому що до неї прикреплен скрипт з необхідною функцією. В поле функцій обираємо UIStartButton → StartGame().



Тепер при натисканні на кнопку Start програма перейде та запустить сцену, яку прописали в функції LoadScene().

Контрольні питання

1. Які основні візуальні UI-компоненти ви знаєте?
2. Що робить компонент TextMeshPro?
3. Що робить компонент Image?
4. Що робить компонент Button?
5. Що таке ігрове меню та його функції?
6. Які кроки потрібно пройти, щоб створити головне меню гри?
7. Що робить компонент Panel?
8. Як зробити анімацію кнопки?
9. Як створити перехід між головним меню та підменю?

Лекція 7. UI. Вікно інвентаря. Зовнішній вигляд вікна

1. Вікно інвентаря
2. Розробка зовнішнього вигляду вікна інвентаря

1. Вікно інвентаря

Інвентар – це спливаюче меню, яке гравець використовує для керування предметами, які він має при собі.

Вікно інвентаря в іграх – це екран, який показує предмети, що їх має персонаж, розділяючи їх на те, що він носить, і те, що зберігає в рюкзаку.

Рисунок 7.1 – Приклад вікна інвентаря в іграх

Система інвентарю є фундаментальною частиною будь-якої гри, оскільки це дозволяє гравцям керувати придбаними предметами, ресурсами та обладнанням під час прогресу в грі. Це важливо для структури та функціональності гри, оскільки забезпечує плавну та організовану гру.

Система інвентаризації гри це структура, яка керує доступністю, отриманням, зберіганням і використанням об’єктів і ресурсів у грі. Щоб гравці мали доступ до цих предметів, У програмуванні використовуються різні елементи та прийоми. Система інвентарю дозволяє гравцям упорядковувати свої предмети ефективним способом і швидко отримувати доступ до них під час гри.

Одним із важливих елементів системи інвентаризації є *об'єктна база даних*. Ця база містить детальну інформацію про всі об'єкти, обладнання та ресурси в грі. Кожен об'єкт має унікальні атрибути, як-от назва, опис, зображення, рідкість, статистика, вага та інші специфічні атрибути залежно від гри.

Ще одним важливим елементом є *система класифікації інвентар*, який дозволяє гравцеві впорядковувати свої предмети за різними критеріями, наприклад, рідкістю, типом або рівнем потужності. Це полегшує пошук конкретних предметів і допомагає гравцеві приймати стратегічні рішення щодо того, які предмети зберігати чи продавати.

Крім бази даних, система інвентаризації використовує *алгоритми та функції* керувати наявністю та зберіганням об'єктів. Ці алгоритми можуть включати методи

сортування, фільтрації та пошуку, які дозволяють гравцям швидко знаходити потрібні предмети у своєму інвентарі.

Однією з основних функцій системи інвентаризації є *обсяги зберігання*. Гравці можуть зберігати різноманітні предмети, такі як зброя, обладунки, зілля та ключові предмети. Інвентар представлений у вигляді сітки або списку, що дозволяє гравцям упорядковувати предмети так, як їм подобається. Крім того, ця система пропонує можливість розширення сховища шляхом придбання спеціальних предметів або вдосконалення навичок персонажа.

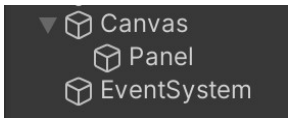
Ще однією важливою функцією системи інвентаризації є можливість *здійснення обміну*. Гравці можуть обмінюватися предметами з іншими гравцями або неігровими персонажами в грі. Це дозволяє гравцям отримувати предмети, яких їм не вистачає, або потрібні для розвитку історії гри. Крім того, деякі предмети можна продати або обміняти на віртуальну валюту, дозволяючи гравцям отримати додаткові ресурси для оновлення свого обладнання або придбання нових предметів.

Окрім зберігання та обміну, система інвентарю також пропонує можливість *оновлювати та налаштовувати предмети*. Гравці можуть комбінувати об'єкти створити нові та потужніші предмети або використовуйте спеціальні компоненти для покращення статистики свого обладнання. Ця функція дозволяє гравцям адаптувати свій інвентар відповідно до бажаного стилю гри та стратегії, покращуючи продуктивність і ефективність у грі. Крім того, деякі предмети можна персоналізувати за допомогою кольорів, візерунків або написів, що додає елемент персоналізації та ексклюзивності до інвентарю гравця.

2. Розробка зовнішнього вигляду вікна інвентаря

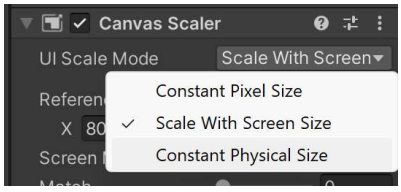
Для створення вікна інвентаря гри створимо на сцені гри новий Canvas.

Додаймо на сцену Canvas (Полотно). Полотно – це ігровий об'єкт з доданим до нього компонентом Canvas. Всі елементи вікна інвентаря повинні бути дочірніми цьому Canvas. Додаймо в Canvas компонент Panel.



В поле Render Mode об’єкту Canvas встановлюємо значення Screen Space – Overlay. Цей режим відображення поміщає елементи інтерфейсу на екран поверх сцени. Якщо змінюється розмір екрана або його роздільна здатність, полотно автоматично прийме потрібний розмір разом із ним.

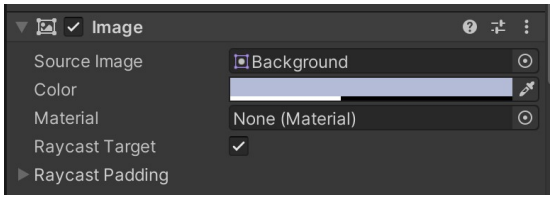
За замовчанням в компоненті Canvas Scaler в полі UI Scale Mode стоїть значення Constant Pixel Size. Якщо зміниться розмір екрана, то всі елементи що входять в Canvas не змінять свого розміру. Замінимо це значення на Scale with Screen Size.



Panel це прямокутний контейнер із Image. Назвемо його InventiryWindow. Налаштуємо розмір компоненти, зробимо його не на весь екран. В компонент Image в поле Source image додаємо картинку фону.

Рисунок 7.1 – Приклад фону вікна інвентаря

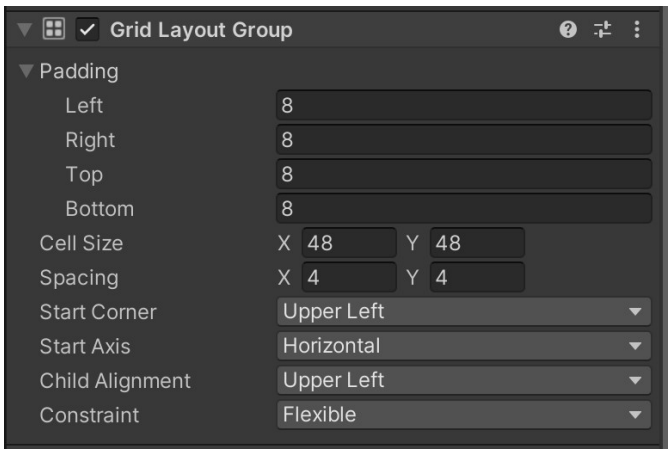
Створюємо дочірнім об’єктом до панелі InventiryWindow компонент Panel та назвемо його ContentView. Саме цю панель будемо розбивати на осередки, в яких будемо показувати картинки об’єктів які підібрали у грі. В компоненті Image змінимо колір фону та зробимо його частково прозорим.



Додаймо до панелі ContentView компонент Grid Layout Group.

Grid Layout Group в Unity – це компонент, який використовується для автоматичного розміщення дочірніх компонентів в рівномірну мережу, де всі клітинки мають однаковий розмір. Розмір та міжклітинкова різниця контролюються компонентом Grid Layout Group самостійно, а дочірні компоненти не впливають на їхні розміри.

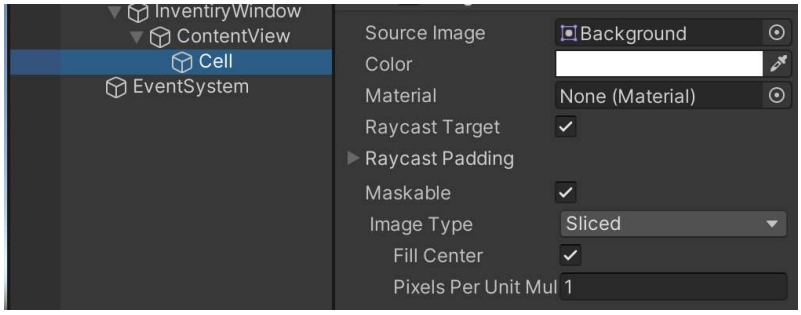
Grid Layout Group дозволяє налаштовувати розміри, отримання та між компонентами мережі, щоб забезпечити правильне розміщення компонентів. Він також дозволяє налаштовувати мінімальні та максимальні розміри компонентів мережі, щоб забезпечити правильне відображення на різних пристроях.



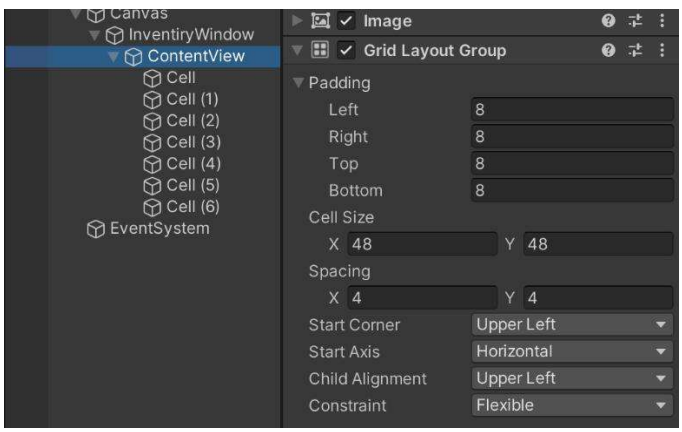
Має такі властивості:

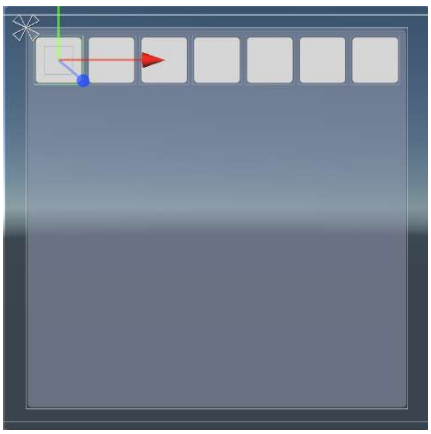
– Padding – відступи всередині країв групи макета.
– Cell Size – розмір, який використовуватиметься для кожного елемента макета в групі.
– Spacing – відстань між елементами макета.
– Start Corner – кут, де розташований перший елемент.
– Start Axis – вздовж якої головної осі розміщувати елементи. Горизонтальне вирівнювання заповнить весь рядок перед початком нового рядка. Вертикальне вирівнювання заповнить весь стовпець перед початком нового стовпця.
– Child Alignment – вирівнювання, яке використовуватиметься для елементів макета, якщо вони не заповнюють весь доступний простір.
– Constraint – обмеження сітки фіксованою кількістю рядків або стовпців для покращення автоматичної системи макета.

Створюємо дочірнім об’єктом до панелі ContentView компонент Image та назвемо його Cell.

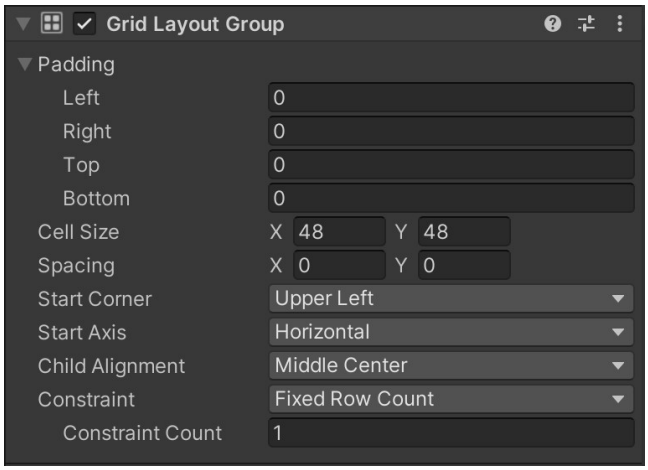


Підберемо значення Padding, Cell Size та Spacing в компоненті Grid Layout Group, щоб в панелі рівномірно розмістилось ціле значення осередків. Для цього тимчасово створіть декілька об’єктів Cell.

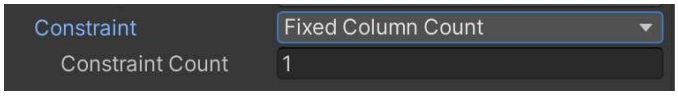




У подальшому до Cell будемо додавати картинку предмету, тому додаймо до об’єкту Cell компонент Grid Layout Group для найкращого розміщення картинки. В компоненті Grid Layout Group зробимо такі налаштування:



Також перевіримо, щоб во властивості Constraint при виборі значення Fixed Column Count було значення 1.



Для надання можливості користувачеві закрити вікно інвентаря, створюємо дочірнім об’єктом до панелі InventiryWindow компонент Button. Компонент Button складається з компонента Image та Text. Додаємо в компоненту Text текст «X» та змінимо колір тексту та зробимо інші налаштунки. Змінимо назву кнопки в Canvas на ButtonClose.

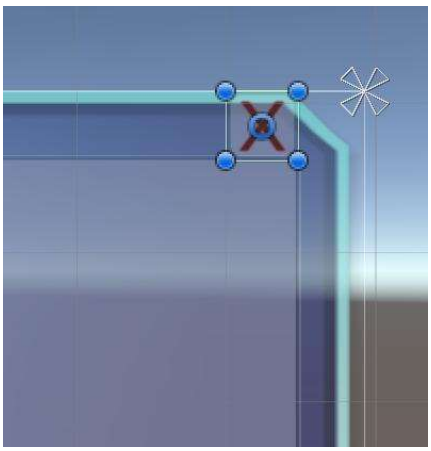
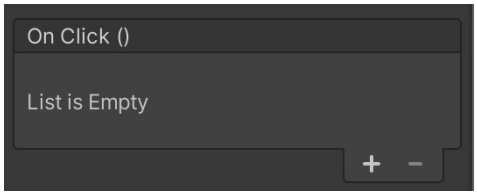
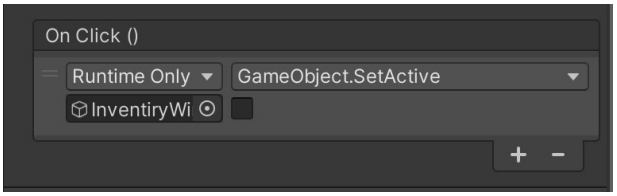


Рисунок 7.2 – Приклад кнопки закриття вікна інвентарю

Додаймо функціонал при натисканні кнопки ButtonClose, а саме елемент InventiryWindow повинен стати не активним. Для цього виберіть із компонента Button розділ «On Click()» і натисніть на плюс.



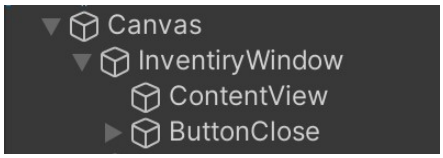


У нижне вікно перетягуємо об’єкт InventiryWindow, тоді справа в полі відкриються доступні функції над цим об’єктом. Обираємо GameObject → SetActive(bool). Рядом з полем з об’єктом InventiryWindow створиться прапорець. Якщо обрати прапорець, то вікно буде активно (видимо), якщо ні – не активно. Для об’єкта InventiryWindow прапорець не обираємо.

Якщо запустити проект, при натисканні кнопки «X» вікно інвентарю закриється.

Залишається додати функціональність. Створимо нові скрипти.

Створюємо скрипт Cell та додаємо його як компонент до об’єкту Cell. Тепер об’єкт Cell не потрібен на сцені, тож зробимо з нього префаб та видалимо зі сцени. Створимо в Assets папку Resources. Тут будуть зберігатися prefabs, тих об'єктів, які необхідно буде створювати динамічна, тобто в процесі гри.



Створюємо скрипт Inventory та додаємо його як компонент до об’єкту InventiryWindow. Для створення необхідної кількості осередків у ContentView створимо поле capacity та поле view для посилання на ContentView. Масив content зберігатиме створені осередки.

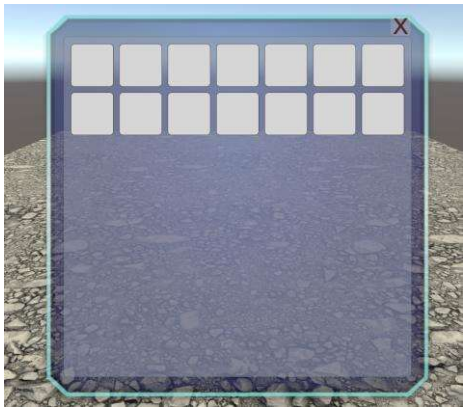
```
[SerializeField]
private int capacity;

[SerializeField]
private Transform view;

public static Cell[] content;
```

Створимо осередки та заповнимо масив content:

```
void Awake()
{
    content = new Cell[capacity];
    CreateCells();
}


void CreateCells()
{
    for (int i = 0; i < capacity; i++)
    {
        GameObject cell = Instantiate(Resources.Load<GameObject>("Cell"));

        cell.transform.SetParent(view);
        cell.transform.localScale = Vector2.one;
        cell.name = string.Format("Cell [{0}]", i);

        content[i] = cell.GetComponent<Cell>();

    }
}
```

В Inspector заповнюємо поля певними значеннями, наприклад, створимо 14 осередків:

Якщо запустити гру, маємо вікно інвентаря з 14 пустими осередками:



Контрольні питання

1. Які робить вікно інвентаря в іграх?
2. Що таке система інвентаризації гри?
3. Як може виглядати вікна інвентаря в грі?
4. Що робить компонент Grid Layout Group?
5. Як створити закриття вікна інвентарю?

Лекція 8. UI. Вікно інвентаря. Зберігання предметів

1. Розташування предметів у вікні
2. Додавання предметів до вікна інвентаря

1. Розташування предметів у вікні

У попередній лекції було створено вікно інвентаря з додаванням пустих осередків. Додаймо можливість показувати в осередках зображення предметів. Нехай маємо такі зображення:



Створимо в Assets папку Sprites та додаймо ці зображення до цієї папки. Виділить в закладці Project перше зображення та в панелі Inspector в поле Texture Type вибрати значення Sprite (2D and UI) та зберегти зміни. Тип Sprite (2D and UI) форматує ресурс таким чином, щоб його можна було використовувати в 3D-додатках як спрайт.

Створимо в панелі ContentView тимчасовий об’єкт типу Image. На панелі Inspector в компоненте Image в поле Source Image перетягуємо, наприклад, перше зображення з папки Sprites. Створимо скрипт Item та приєднуємо його до створеного зображення типу Image. Збережемо об'єкт, що вийшов, як префаб, наприклад, в папці Resources. Перейменуємо, наприклад, в Poison01.

В папці Resources створимо ще дві копії префаба Poison01, назвемо їх Poison02 та Poison03. В компоненті Image в поле Source Image для нових префабів встановимо інші картинки.

В скрипте Item створимо такі поля та їх ініціалізацію:

```
[HideInInspector]
public Cell cell;
private Transform canvas;

void Start()
{
    cell = GetComponentInParent<Cell>();
    canvas = GameObject.Find("Canvas").transform;
}
```

Необхідно додати можливість користувачеві перетягувати картинки у вікні інвентаря між осередками. Для цього зробимо клас Item, що реалізує два інтерфейси IDragHandler, IEndDragHandler та реалізуємо два методи: OnDrag та OnEndDrag.

```
public class Item : MonoBehaviour, IDragHandler, IEndDragHandler
...

public void OnDrag(PointerEventData eventData)
{
    transform.SetParent(canvas);
    transform.position = eventData.position;
}
```

```
public void OnEndDrag(PointerEventData eventData)
{
    float distance = float.MaxValue;
    Cell newCell = cell;

    for (int i = 0; i < Inventory.content.Length; i++)
    {
        float temp = Vector2.Distance(transform.position, Inventory.content[i].transform.position);

        if (temp < distance)
        {
            if (Inventory.content[i].transform.childCount > 0)
                continue;

            distance = temp;
            newCell = Inventory.content[i];
        }
    }

    cell = newCell;

    transform.SetParent(cell.transform);
    transform.position = cell.transform.position;
    transform.localScale = Vector2.one;
}
```

Методи OnDrag та OnEndDrag реалізують переміщення картинок по осередкам. Якщо для картинки вказати не конкретний осередок, то програма знайде найближчий не зайнятий та перенесе картинку до нього.

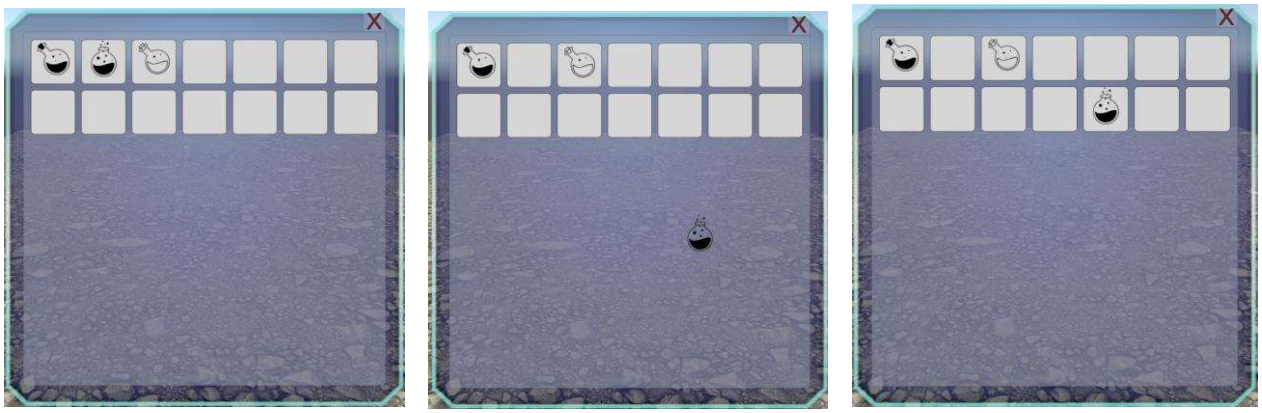


Рисунок 8.1 – Приклад вигляду вікна інвентарю при переміщенні картинок

2. Додавання предметів до вікна інвентаря

Додаймо до сцени площину та персонаж зі скриптом, який керує його переміщенням.

Створимо на сцені декілька об’єктів яких персонаж буде підбирати та зберігати у вікні інвентаря. Для простоти візьмемо в якості таких об’єктів сфери. Створимо скрипт ItemObejct та приєднаємо до сфер.

Рисунок 8.2 – Приклад вигляду сцени

Розглянемо структуру скрипту ItemObejct.

Створимо поле для зберігання картинки з префабів, яка додається до вікна інвентарю у випадку підбирання об’єкту персонажем.

```
public GameObject item;
```

В скрипті Inventory створити метод, який додає картинку до інвентарю:

```
public static bool AddItem(GameObject item)
{
    for (int i = 0; i < content.Length; i++)
    {
        if (content[i].transform.childCount == 0)
        {
            GameObject newItem = Instantiate(item);

            newItem.transform.SetParent(content[i].transform);
            newItem.transform.localScale = Vector3.one;
            newItem.transform.localPosition = Vector3.zero;
            newItem.transform.position = newItem.transform.parent.position;
            newItem.GetComponent<Item>().cell = content[i];

            return true;
        }
    }

    return false;
}
```

Метод AddItem получає на вході об’єкт який необхідно додати. В методі шукається пустий осередок у вікні інвентарю та коли такий знаходиться до нього додається картинка.

В скрипті ItemObejct створимо метод який визиває метод AddItem:

```
void Update()
{
    CheckPickUp();
}

void CheckPickUp()
{
    if (Vector3.Distance(transform.position, MoveHelper.player.position) < 1.0F)
    {
        //Debug.Log("Near");
        if (Inventory.AddItem(item))
        {

            Destroy(gameObject);
        }

    }
}
```

MoveHelper – це скрипт який навішаний на персонажа і в якому поле player зберігає посилання на персонаж на сцені.

Коли персонаж підбере об'єкт на сцені, до інвентаря додається префаб картинки зарезервований за цим об'єктом. У подальшому об'єкт на сцені прибирається.

Контрольні питання

1. Що таке спрайт?
2. Для чого використовуються спрайти у вікні інвентарю?
3. Для чого створюється папака Resources?
4. Що робить інтерфейс IDragHandler та IEndDragHandler?
5. Що роблять методи OnDrag, OnEndDrag?
6. Як реалізувати додавання предметів до вікна інвентаря?

Лекція 9. UI. Додаткові ігрові вікна

1. Ігрові вікна
2. HUD в іграх

1. Ігрові вікна

Ігри – це захоплююче заняття, яке допомагає нам відволіктися від реальності та поринути у світ фантазії та пригод. Але коли ми запускаємо гру на комп'ютері або мобільному пристрої, виникає безліч різних ігрових вікон, які можуть плутати або викликати питання. Розглянемо як називаються ігрові вікна і навіщо вони призначені.

Назви ігрових вікон можуть відрізнятися в різних іграх, але в основному вони виконують одні й ті самі функції – забезпечують кращу ігрову платформу, зручність керування та навігації, а також допомагають взаємодіяти з гравцями по всьому світу. Крім основних вікон, в іграх є додаткові вікна, які надають додаткові можливості та інформацію для більш повного та цікавого ігрового процесу.

Основні ігрові вікна:

- Головне вікно гри – це головне вікно гри, в якому відображається весь ігровий процес. Тут ви керуєте своїм персонажем або об'єктом, взаємодієте з іншими гравцями та виконуєте різноманітні ігрові завдання.
- Вікно налаштувань – тут можна налаштувати графіку гри, управління, звукові ефекти та інші параметри, щоб плавно і комфортно грати.
- Інвентарне вікно – у цьому вікні ви можете переглядати та керувати своїм інвентарем, тобто набором предметів та ресурсів, які ви зібрали у грі. Тут ви можете вибрати потрібні речі для використання чи продажу.
- Вікно завдань – якщо у грі передбачені завдання або квести, то вони відображаються у спеціальному вікні, де ви можете побачити поточні завдання, їх опис та прогрес виконання.
- Чат – багато онлайн-ігор мають чат, в якому ви можете спілкуватися з іншими гравцями, ставити питання, узгоджувати стратегію і просто насолоджуватися спілкуванням.

Додаткові ігрові вікна:

- Вікно магазину – якщо гра має внутрішньоігрову валюту або магазин, ви можете відкрити спеціальне вікно, де можна придбати предмети, поліпшення або додатковий контент.
- Вікно карти – деякі ігри надають карту ігрового світу, в якій ви можете переглядати своє становище, можливості переміщення та цікаві місця.
- Вікно досягнень – тут відображаються всі важливі досягнення та нагороди, які ви здобули у грі, щоб ви могли похвалитися своїми успіхами.
- Вікно лідерів – у режимах гри змагання можна переглянути таблицю лідерів, де відображається рейтинг гравців та їх досягнення.
- Вікно довідки – якщо у грі виникли труднощі або питання, ви можете відкрити вікно довідки, де знайдете відповіді на різні питання та рекомендації щодо проходження.
- Вікно програшу або перемоги – вікно в якому повідомляється, що ви виграли або програли і надає подальші дії в ситуації, що склалася.

Різновиди ігрових вікон:

- Вікно діалогу – це один із найпоширеніших типів ігрових вікон. Воно відкривається при взаємодії з персонажами гри та дозволяє гравцеві вибирати діалогові варіанти, ставити запитання чи отримувати інформацію.

- Меню — це вікно гри, яке містить різні настройки та параметри гри. Тут гравець може змінювати керування, налаштовувати графіку, вибирати рівень складності тощо.
- Інвентар — це вікно, де відображається список предметів, які гравець зібрав протягом гри. Тут він може керувати своїм інструментом — вибирати, використовувати, продавати чи купувати предмети.
- Карта - це ігрове вікно, яке дозволяє гравцеві переглядати ігровий світ та його локації. Тут він може відзначати цікаві місця, знаходити шлях до наступної мети чи дослідити довкілля.

2. HUD в іграх

В іграх існує величезна кількість інтерфейсів: інвентар, діалоги, меню крафту та торгівлі, лобі, карта, дерева прокачування персонажа та його екіпірування та багато інших. Всі вони дозволяють гравцям взаємодіяти із представленими через інтерфейс механіками, які творці гри заклали у свій продукт.

HUD, Heads Up Display – це набір графічних інтерфейсів та декоративних елементів, розташованих на передньому плані та/або в ігровому світі, які покликані явно та швидко доносити до гравця необхідну інформацію від гри у певний момент часу, не заважаючи отриманню ігрового досвіду.

Еволюція інтерфейсу користувача з часом призвела нас до кількох гілок його розвитку. Зі зростанням популярності та складності ігор інтерфейс ставав їх важливою частиною. Поява нових жанрів народжувало і нові типи інтерфейсу: потрібно відображати більше інформації в рамках нових ігрових механік, тому старого доброго лічильника очок і таймера поверх геймплею вже не вистачало.

Згодом ці галузі розвитку були систематизовані в кілька типів інтерфейсів, які сьогодні ми можемо спостерігати в тих чи інших іграх. Залежно від того, чи є інтерфейс частиною оповідання, а також чи є він частиною ігрового простору, ігрові інтерфейси поділяються на такі типи:

- Дієгетичні – коли, наприклад, швидкість автомобіля передається через спідометр, що у кабіні вантажівки. Інтерфейс, який бачить та з яким може взаємодіяти персонаж гравця. Він дозволяє отримати більш наративний досвід у рамках ігрового світу;
- Мета – наприклад, краплі крові та води на «візорі». По суті – постпроцеси, що дозволяють трансформувати досвід персонажа у досвід гравця через ще один шар взаємодії;
- Просторові – це різні виділені області, інтерактивні об'єкти у світі, траєкторії польоту гранат, проекційні повідомлення тощо, створені для гравця, але з персонажа. Для персонажа їх немає і він не може з ними взаємодіяти;
- Недієгетичні – той же витратомір/спідометр і показники цілісності вантажівки – по суті віджети, розташовані поверх геймплей. Дозволяє просто та швидко зчитувати інформацію, але ламає наратив і дуже рідко пояснюється через ігрову історію.

Для HUD цей поділ більш актуальний, тому що в інших інтерфейсах гри використовується в основному недієгетичний тип інтерфейсу. При цьому саме HUD може активно використовувати 3-4 типи одночасно.

Рисунок 9.1 – Приклад трьох типів інтерфейсу на одному екрані, але на різних фічах (1 – недієгетичний, 2 – дієгетичний, 3 – просторовий)

Інформація повинна знаходитись у доступності від точки/області основного фокусу уваги гравця під час ігрового процесу. Для розуміння – розіб'ємо екран на базові блоки HUD:

Рисунок 9.2 – Приклад базових блоків HUD

Тут ми бачимо поділ екрана на такі блоки:

– Головну область фокусу (MFA) – область екрану з найвищою видимістю для гравця, де концентрується його увага під час ігрового процесу. Центр цієї області це центр екрану, точка концентрації основного геймплею.
– Області навколо – з більш низькою видимістю за рахунок свого віддалення від точки максимальної видимості, які найчастіше використовуються для відображення UI.

Якщо раптом ви граєте у щось швидше і когнітивно навантажене вам необхідно отримувати інформацію якнайшвидше та зручніше, для того, щоб бути успішним, наприклад, у шутерах. Саме тому такі елементи, як покажчик напряму атаки, з'являються в момент влучення, знаходяться в головній області фокусування і мають великий розмір.

При розміщенні інтерфейсів в областях навколо MFA вони менш відволікатимуть на себе увагу і при цьому будуть перебувати в достатній близькості від точки фокусування уваги гравця.

Розташування навколо MFA дозволить гравцеві просто і швидко використовувати інформацію, що відображається, витрачаючи на це невелику кількість часу і зусиль. Однак варто зауважити, що видимість інтерфейсів в областях навколо MFA все ж таки буде значно знижена в порівнянні з інтерфейсами всередині MFA.

Зниження видимості інтерфейсу в залежності від його віддалення від точки концентрації уваги та розміру описується законом Фіттса.

Час, необхідне для швидкого переміщення до цільової області (до областей навколо MFA), є функцією відношення між відстанню до мети (зоровий шлях) і розміром мети (цільового інтерфейсу). Якщо простіше: чим далі наш інтерфейс знаходиться від центру екрана, чим він менший, тим менш помітним він стає для уваги гравця, а значить він витрачатиме більше когнітивних зусиль на те, щоб з ним взаємодіяти.

Щоб нівелювати зниження видимості та зменшити необхідні зусилля, UI у цих областях має ініціювати зміщення уваги користувача на себе. Зробити це може через ефекти, анімації та інші можливі показники свого зміненого стану. Це необхідно, щоб гравець не пропустив критично важливу інформацію, і йому було простіше взаємодіяти з інтерфейсом.

Основні елементи HUD:

- Індикатори здоров'я – показують стан здоров'я гравця, часто у вигляді шкали чи чисел. При отриманні пошкоджень індикатор може змінювати колір або мигати, щоб вказати на небезпеку;
- Кількість боєприпасів – відображає, скільки набоїв залишилося у зброї, а також типи доступних боєприпасів;
- Рівень та досвід – наприклад у RPG-іграх, HUD може показувати рівень персонажа та кількість досвіду, необхідного для підвищення рівня;
- Прогрес у грі – інформація про поточний рівень, завдання, кількість очок або грошей, а також відсоток завершення завдань;
- Інвентар – деякі ігри мають панель, що дозволяє швидко використовувати предмети, такі як медичні набори або зілля.

Контрольні питання

1. Що таке основні та додаткові ігрові вікна?
2. Для чого використовуються ігрові вікна?
3. Що таке HUD в іграх?
4. Які базові блоки HUD ви знаєте?
5. Що говорить закон Фіттса?
6. Які основні елементи HUD ви знаєте?

Лекція 10. UI. HUD та вікно перемоги

1. Розробка HUD
2. Створення вікна перемоги

1. Розробка HUD

HUD, Heads Up Display – це набір графічних інтерфейсів та декоративних елементів, розташованих на передньому плані та/або в ігровому світі, які покликані явно та швидко доносити до гравця необхідну інформацію від гри у певний момент часу, не заважаючи отриманню ігрового досвіду.

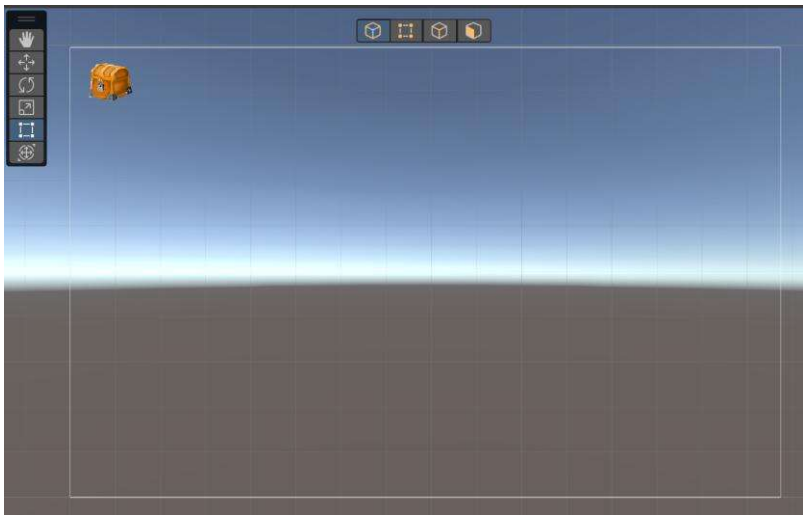
Основними елементи HUD є індикатори здоров'я, кількість боєприпасів, рівень та досвід, прогрес у грі, інвентар тощо.

На попередніх лекціях розглядали створення вікна інвентарю. Вікно інвентарю показується не весь час на екрані, а з’являється при потребі користувача. Доповнимо нашу програму кнопкою для виклика вікна інвентарю.

По-перше, вимкнемо вікно інвентарю. Для цього в Canvas зробимо неактивним панель InventiryWindow.

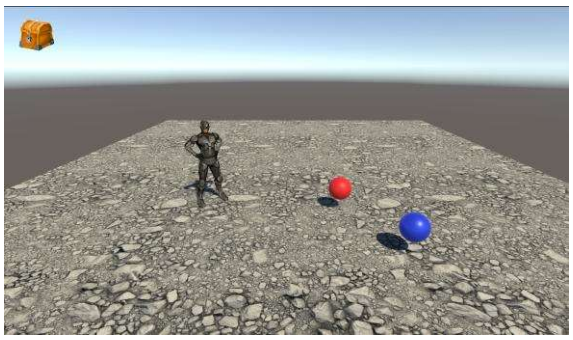
Створимо у Canvas кнопку, наприклад з назвою OpenInventory. В компоненті Image в поле Source Image додаймо зображення, наприклад якоїсь скрині. В компоненті Text (TMP) видалимо надпис.

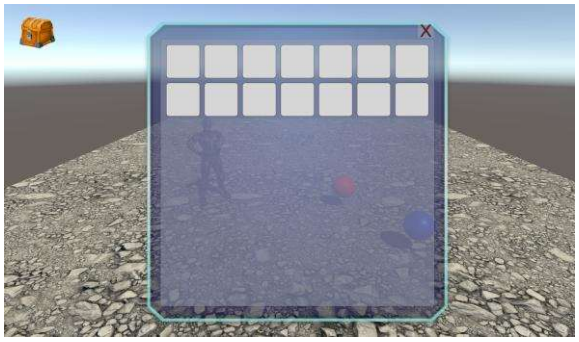
Прикріпимо кнопку до лівого верхнього куту Canvas:



Наприкінці, обробимо подію натискання на кнопку. А саме повинно з’явитися вікно інвентарю, тобто воно повинно стати активним.

Якщо запустити проект, то при натисканні кнопки з скринею з’явиться вікно інвентарю:



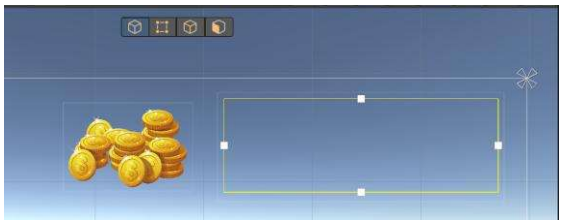


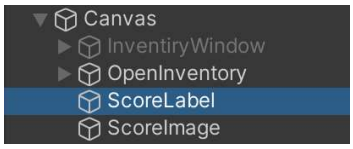
Додаймо у наш інтерфейс можливість бачити прогрес у грі. В якості процесу у грі будемо розуміти інформацію про кількість очок або грошей, які має користувач. Інформація складатиметься з двох частин: картинки, що показує тип інформації та сама кількість. Позиціонувати цю інформацію будемо у правому верхньому кутку.

Створимо у Canvas картинку та в компоненті Image в поле Source Image додаймо зображення, наприклад купки монет та назвемо ScoreImage.

Також створимо Text та назвемо ScoreLabel. Налаштуємо шрифт та колір тексту та видалимо текст.

Нові елементи позиціонуємо до правого верхнього кутку.





Створимо скрипт HUD та прикріпимо його до об’єкту Canvas. В скрипті створимо посилання на екземпляр класу:

```
static private HUD _instance;
public static HUD Instance { get { return _instance; } }

private void Awake()
{
    _instance = this;
}
```

Створимо поле для посилання на об’єкт Text та метод який буде заповнювати цей об’єкт інформацією:

```
[SerializeField]
private TMP_Text scoreLabel;
```

```
public void SetScore(string scoreValue)
{
    scoreLabel.text = scoreValue;
}
```

Далі нам потрібен скрипт який буде керувати подіями на сцені. Для цього створюємо пустий об’єкт на сцені назвемо його GameController. Створимо скрипт з такою ж назвою та прикріпим його до цього об’єкту.

В скрипті GameController створимо посилання на екземпляр класу:

```
static private GameController _instance;
public static GameController Instance { get { return _instance; } }


private void Awake()
{
    _instance = this;
}
```

Створимо поле score, в якому зберігатися поточна кількість очок або грошей.

```
private int score;
private int Score { get { return score; } set { score = value; } }
```

В методі Start() ініціалізуємо цю змінну та визиваємо метод для показу поточної кількості в HUD, а також створюємо метод SetScore() для зміни значення.

```
private void Start()
{
    Score = 0;
    HUD.Instance.SetScore(Score.ToString());
}

public void SetScore(int add)
{
    Score += add;
    HUD.Instance.SetScore(Score.ToString());

}
```

В скрипті ItemObejct в метод CheckPickUp() додаємо рядок передачі кількості очок до GameController:

```
void CheckPickUp()
{
    if (Vector3.Distance(transform.position, MoveHelper.player.position) < 1.0F)
    {
        //Debug.Log("Near");
        if (Inventory.AddItem(item))
        {
            GameController.Instance.SetScore(5);
            Destroy(gameObject);
        }

    }
}
```

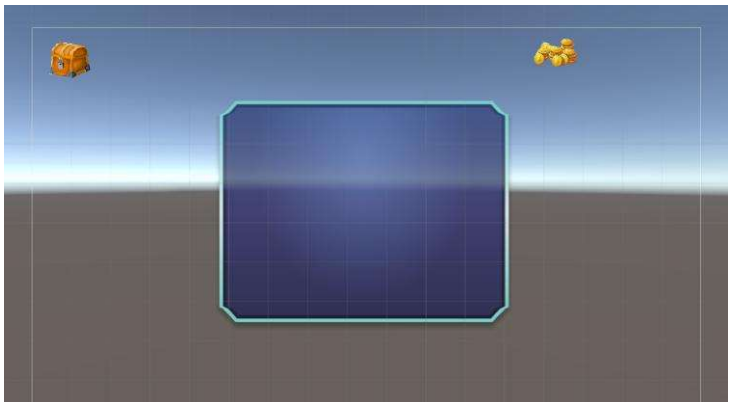
В нашому випадку коли персонаж підбере об’єкт на сцені, до інвентаря додається префаб картинки зарезервований за цим об’єктом, поточна кількість очок збільшиться на 5. У подальшому об’єкт на сцені прибирається. Можна ускладнити програму, зробивши для кожного об’єкту на сцені свою кількість балів які отримає користувач, якщо підбере їх на сцені.

2. Створення вікна перемоги

Нехай коли користувач накопить певну кількість очок гра припиняється і з’являється вікно перемоги. Розглянемо кроки як реалізувати таку задачу.

По-перше створимо вікно перемоги.

Створимо у Canvas нову Panel та назвемо LevelWonWindow. В компоненті Image в поле Source Image додаймо зображення фону вікна та налаштуємо його розмір:



Додаймо до панелі LevelWonWindow такі об’єкти як Text та Button. Налаштуємо їх наступним образом:

Необхідно додати функціональність появи цього вікна та що буде відбуватися при натисканні кнопки у вікні.

У скрипт HUD додаймо поле для зберігання посилання на вікно виграшу та методи показу цього вікна та переходу на іншу сцену:

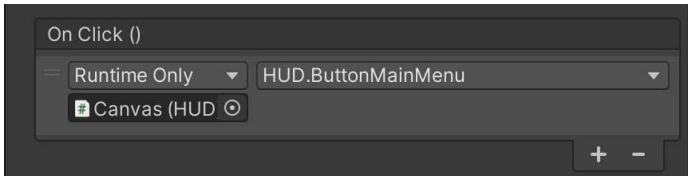
```
[SerializeField]
private GameObject LevelWonWindow;


public void ShowLevelWonWindow()
{
    LevelWonWindow.SetActive(true);
}

public void ButtonMainMenu()
{
    SceneManager.LoadScene("MainMenu");
}
```

В скрипт GameController до методу SetScore() додаймо перевірку кількості набраних очок:

```
public void SetScore(int add)
{
    Score += add;
    HUD.Instance.SetScore(Score.ToString());

    if (Score >= 10)
        HUD.Instance.ShowLevelWonWindow();

}
```

У подальшому в скрипт GameController можна ввести поле для зберігання кількості очок при якому завершується гра та замінити число 10 на цю змінну.

І в завершенні для кнопки на панелі LevelWonWindow додати дію при натисканні:



Запустимо проект, підберемо два об’єкти та бачимо вікно виграшу.

Контрольні питання

1. Як створити HUD для відкриття вікна інвентаря?
2. Як зберігати інформацію про кількість очок або грошей, які має користувач?
3. Які функції несе скрипт GameController?
4. Наведіть основні кроки для створення вікна перемоги?
5. Як перейти з вікна перемоги до головного меню?
6. Як відображається інформація про кількість очок або грошей, які має користувач?

Лекція 11. Огляд популярних атак на комп'ютерні ігри та методи їх проведення

1. Основні визначення
2. Популярні атаки на комп'ютерні ігри

1. Основні визначення

Кібератака – це навмисна спроба окремої особи або групи осіб зламати, пошкодити або порушити роботу комп'ютерних систем, мереж або цифрових пристроїв, часто в зловмисних цілях, таких як крадіжка даних, шпигунство, фінансове шахрайство або саботаж системи.

У період з 2019 по 2025 роки експерти з кібербезпеки виявили тривожне зростання спроб кібератак, пов'язаних з відеоіграми. Ця тенденція впливає як на найпопулярніші ігри, так і на звички ігрової спільноти щодо завантажень, особливо серед молоді. У цей період було зафіксовано понад 2025 мільйонів спроб кібератак із використанням файлів, замаскованих під відеоігри.

Зловмисники користуються довірою та цікавістю геймерів, які часто шукають неофіційні моди, чіти чи версії на неофіційних сайтах. Наслідки цих атак не обмежуються зараженням пристроїв. Один із найсерйозніших наслідків — крадіжка облікового запису відеоігри. Багато з цих облікових записів, які можуть містити скіни, прогрес та цінні предмети, зрештою перепродаються на підпільних ринках.

- Сучасні комп'ютерні ігри – це складні програмні системи, що включають:
- Клієнтську частину – гра на ПК, консолях, мобільних пристроях;
- Серверну частину – онлайн-логіка, бази даних, авторизація;
- Мережеві протоколи обміну даними;
- Економіку – внутрішньоігрова валюта, предмети, донат.

Ігри є привабливою метою атак з наступних причин:

- Фінансова мотивація – продаж читів, ботів, зламаних акаунтів;
- Змагальний фактор – бажання отримати перевагу;
- Низький рівень захисту клієнтської логіки;
- Велика аудиторія та слабка цифрова грамотність користувачів.

Важливо розуміти, що ігрові атаки не відрізняються принципово від атак на звичайні програми, але використовують особливості ігрової логіки.

Статистика кібератак, пов'язаних з відеоіграми, показує, що це явище зачіпає не тільки молодь, але й будь-який гравець, що взаємодіє з іншими гравцями за межами офіційних каналів, може наражатись на ризик втрати даних або поставити під загрозу свою конфіденційність. Найкращим захистом залишається обережність та правильне використання засобів безпеки, щоб насолоджуватися грою без побоювань.

Розглянемо деякі специфічні терміни, які використовуються в геймплеї.

Перші чит-коди у іграх додавали самі розробники й тестувальники. Спочатку це були вбудовані інструменти, які девелопери використовували, щоб легше знайти баги. Чит-коди дозволяли тестувати функції, швидко перевіряти рівні та налаштовувати параметри.

З часом розробники виявили, що чит-коди можуть зацікавити гравців. Так ігрові студії почали залишати деякі чит-коди в іграх. Чит-коди були приховані від геймерів, а деякі можна було розкрити через послідовності дій. Серед яскравих прикладів — відеогра Doom (1993), де можна знайти чит-код після вибору місії та введення коду.

Чит-коди стали популярним елементом у геймдев-культурі. Гравці обмінювалися відомими кодами, а ігрові журнали публікували їх для зацікавлення читачів. Спільноти гравців досі активно вивчають ігри, шукаючи нові чит-коди або неочікувані можливості, щоб розширити свій геймплей.

Щоб геймери не використовували чит-коди, розробники вдаються до античит-систем. Вони виявляють надмірне використання певних функцій (це часто вказує, що тут були чити)

та вживають заходів для їхнього припинення. Серед методів, якими розробники карають за читерство, — карантин, блокування гравців і тимчасові виключення.

Чит-коди створюють заздалегідь, і їхній функціонал спеціально заклали в гру розробники. А чит — це сторонній софт, який знаходить вразливість у відеогрі та дозволяє «змахлювати».

Мод (від англійського "modification" – модифікація) – це доповнення гри, яке змінює її зміст, геймплей, графіку, аудіо, або інші аспекти. Модифікації дозволяють гравцям створювати власні варіанти гри, змінюючи її під свої вподобання або додаючи нові функції.

Моди можуть бути створені спільнотою гравців або розробниками гри. Вони можуть бути досить простими, такими як зміна вигляду персонажів або об'єктів у грі, або складними, такими як додавання повноцінних історій, місій, нових персонажів, режимів гри, або навіть створення зовсім нових світів ігрового процесу. Завдяки модам гравці можуть насолоджуватись більш різноманітним геймплеєм і продовжити ігровий досвід, після того, як пройшли оригінальну гру.

Моддери це ті хто створює моди. Вони можуть брати за основу код гри та інші ресурси, що надаються розробниками, і змінювати їх згідно зі своїми ідеями та вміннями.

Важливо зазначити, що не всі ігри дозволяють модифікації, або вони можуть бути обмежені правами або заборонені політикою розробника чи видавця гри. Крім того, встановлення модів може бути пов'язане з ризиком, так як неправильні модифікації можуть порушити стабільність гри або спричинити проблеми з її функціонуванням.

2. Популярні атаки на комп'ютерні ігри

Геймінг – це мільярдний ринок, де обертаються великі гроші, і саме тому геймери є бажаною ціллю для кіберзлочинців. Адже навіть у грі користувач може втратити персональні дані, гроші або доступ до акаунтів.

Усі атаки умовно поділяються на кілька груп:

- Атаки на клієнтську частину – модифікація пам'яті, підміна логіки, ін'єкції коду;
- Атаки на ігрові дані – підміна збережень, зміна конфігураційних файлів;
- Мережеві атаки – перехоплення пакетів, заміна запитів;
- Атаки на серверну частину – SQL-ін'єкції, логічні уразливості;
- Атаки на ігрову економіку та акаунти – боти, фішинг, перебір паролів.

Memory Hack (атаки на пам’ять) – це зміна значень оперативної пам'яті гри під час її виконання.

Як проводиться атака:

- Гра завантажується на пам'ять (RAM);
- Зловмисник використовує інструменти: Cheat Engine, ArtMoney, Process Hacker;
- Виконується пошук значень (наприклад, кількість здоров'я = 100);
- Значення змінюється (наприклад, 9999);
- Гра продовжує використовувати підмінене значення.

Cheat Engine, ArtMoney, та Process Hacker – усі ці програми використовуються для модифікації пам'яті ігор, дозволяючи змінювати різноманітні ігрові параметри, такі як гроші, здоров'я, боєприпаси тощо. Проте їх функціонал обмежений.

Cheat Engine та ArtMoney вимагають певних навичок та знань для ефективного використання, на відміну від платних читів, які часто пропонують більш розширений функціонал. ArtMoney має інтуїтивно зрозумілий інтерфейс, який легко використовують навіть новачки, але використання програми може призвести до бана облікового запису або інших наслідків у розрахованих на багато користувачів або онлайн-іграх. Cheat Engine, у свою чергу, дозволяє змінювати будь-які параметри, але має складний інтерфейс, який може бути непридатним для новачків.

Атака шкідливі моди та «безплатні чіти» – чимало гравців прагнуть модифікувати ігровий досвід: встановлюють нові карти, візуальні покращення, чіти. Саме на цьому й

грають кіберзлочинці: вони створюють моди, які містять троянські програми, майнери чи кейлогери. Гравець запускає мод, а разом із ним активується шкідливий процес, що викрадає паролі або використовує ресурси ПК.

Кейлогери (keyloggers) – це програми, які призначені для записування та збереження всіх натискань клавіш на клавіатурі комп'ютера або смартфона. Кейлогери можуть використовуватися для злочинних цілей, таких як злам банківських рахунків, крадіжка паролів та інших конфіденційних даних, шпигунство за користувачами, відстеження дій працівників та інше.

Майнери – це програми, що використовують потужність вашого комп'ютера для видобутку криптовалют без вашого відома. У контексті ігор це особливо небезпечне, оскільки вони можуть: значно навантажувати процесор та графічну карту, сповільнюючи роботу ігор і системи; викликати перегрів компонентів та зношування обладнання; встановлюватися приховано, наприклад, через модифіковані ігрові файли або шкідливі завантаження. Користувач може навіть не помітити встановлення майнера, оскільки він працює у фоновому режимі, споживаючи ресурси без прямого втручання.

Атаки на збереження (Save Game Hack) – редагування файлів збережень (JSON, XML, Binary).

Як проводиться:

– Пошук папки збережень;
– Відкриття файлу в текстовому чи hex-редакторі;
– Зміна значень (гроші, рівні, предмети).

Причини вразливості – відсутність шифрування та відсутність підпису цілісності.

Speed Hack (атаки на якийсь час) — зміна швидкості перебігу ігрового часу.

Механізм атаки:

– перехоплення системних викликів часу (Time.time, deltaTime);
– уповільнення чи прискорення таймера гри;
– підміна часових коефіцієнтів.

Ефекти:

– прискорення персонажа;
– прискорений видобуток ресурсів;
– швидке прокачування.

Причини можливості атаки – прив'язка логіки до локального часу клієнта; відсутність серверної синхронізації.

Фішингові атаки – наймасовіший тип шахрайства в геймінгу. Хакери створюють копії відомих сайтів (Steam, Battle.net, Xbox Live тощо), надсилають гравцеві посилання через чат, email або Discord, пропонуючи безплатні скіни, бонуси або участь у розіграші. Після переходу на фейковий сайт гравець вводить логін і пароль – і за секунди втрачає контроль над акаунтом. Часто облікові записи з цінними ігровими предметами одразу перепродаються на чорному ринку.

Мережева атака в комп'ютерних іграх – це спеціальна дія, спрямована на несанкціонований доступ, спотворення або блокування роботи ігрового сервісу або клієнта гри. Мета таких атак зазвичай полягає у контролі над грою, отриманні переваги над іншими гравцями, вимкненні серверів чи перехопленні конфіденційних даних.

Мережеві атаки та Packet Manipulation – перехоплення мережевого трафіку, заміна пакетів, повтор запитів (Replay Attack) тощо.

Інструменти: Wireshark; Fiddler; Burp Suite.

Типові цілі:

– підміна нагород;
– фальшиві запити на купівлю;
– підміна координат.

Wireshark використовується для захоплення та аналізу мережевого трафіку, допомагаючи розуміти роботу мережі і діагностувати проблеми з підключенням та безпекою.

Основні функції Wireshark:

- Захоплення мережевого трафіку – Wireshark дозволяє перехоплювати пакети даних, що передаються через комп'ютерну мережу, у режимі реального часу. Це може бути трафік Ethernet, Wi-Fi, Bluetooth, TCP/IP та інших протоколів.
- Аналіз протоколів – програма розпізнає сотні мережевих протоколів і дозволяє переглядати деталі кожного пакету — адресу відправника та одержувача, заголовки протоколів і вміст переданих даних. Це корисно для налагодження та навчання.
- Виявлення проблем і помилок – Wireshark допомагає знаходити проблеми з мережею, наприклад, втрачені пакети, затримки у з'єднанні або неправильну конфігурацію маршрутизаторів та серверів.
- Безпека та аудит мережі – завдяки аналізу трафіку можна виявляти підозрілі активності, спроби несанкціонованого доступу або атаки на мережу. Wireshark застосовують у тестуванні мереж та аудиті безпеки.
- Фільтри та звітність – програма дозволяє застосовувати фільтри для відбору конкретних протоколів, IP-адрес чи портів, що значно спрощує роботу з великим обсягом даних. Захоплені пакети можна зберігати для подальшого аналізу та створення звітів.

Fiddler – це інструмент для перехоплення, аналізу та модифікації HTTP і HTTPS трафіку, який допомагає розробникам та тестувальникам відстежувати та налагоджувати запити до серверів. Він дозволяє бачити всі HTTP та HTTPS запити та відповіді, включно із заголовками, тілом запитів, параметрами та кукі. Це допомагає виявляти помилки, вразливості та проблеми зі швидкістю завантаження веб-ресурсів.

Burp Suite – це інтегрована платформа для тестування безпеки веб-додатків, яка дозволяє перехоплювати, аналізувати та модифікувати HTTP/HTTPS трафік, автоматично і вручну виявляти вразливості, а також проводити різноманітні атаки для оцінки безпеки веб-сайтів чи API.

Burp Suite використовується для тестування на проникнення веб-додатків. Вона встановлюється між браузером користувача і сервером так, щоб перехоплювати всі запити і відповіді, дозволяючи досліджувати веб-трафік, шукати уразливості, модифікувати запити або вводити автоматизовані сценарії атак.

DDoS-атаки спрямовані на перевантаження ігрових серверів, щоб вивести їх з ладу. Це не тільки шкодить розробникам, але й гравцям – у рейтингових матчах втрата з'єднання може призвести до бану чи втрати досягнень.

IP-спуфінг і перехоплення пакетів – зловмисник підмінює власну IP-адресу на адресу легітимного користувача. Перехоплення або маніпуляція ігровими пакетами дозволяє змінювати стан гри або отримувати конфіденційні дані гравців.

Трояни та мережеві хробаки – програми, що заражають комп'ютери користувачів і автоматично передають управління атакуючому. Можуть викрадати облікові дані чи змінювати поведінку клієнта гри.

Sniffing і перехоплення трафіку LAN – у локальних мережах зловмисник може отримати доступ до всього обміну даними між комп'ютерами. Це дозволяє читати або змінювати пакети гри безпосередньо на маршрутах передачі.

Мережева атака в іграх – це форма віддаленого втручання, яка може порушити конфіденційність, цілісність або доступність ігрових ресурсів. Відмінною особливістю саме ігрових атак є їх орієнтація на ігрову логіку, latency та взаємодію між клієнтом і сервером, що відрізняє їх від загальноприйнятих кібернетичних атак на корпоративні системи. Правильна організація захисту та використання сучасних методів шифрування і аналізу трафіку дозволяє мінімізувати ризик таких атак.

Не всі кібератаки – це технічні трюки. Соціальна інженерія передбачає маніпуляції, коли зловмисник налагоджує «довірливі» стосунки з гравцем або навіть співробітником студії.

Кіберзагрози в геймінгу – це реальність, з якою стикаються як розробники, так і гравці. Щоб убезпечити себе, дотримуйтесь кількох правил:

- Використовуйте двофакторну автентифікацію.
- Не переходьте за підозрілими посиланнями та не вводьте свої дані на незнайомих сайтах.
- Регулярно оновлюйте антивірусне програмне забезпечення та операційну систему.
- Будьте обережні з модами та сторонніми програмами – завантажуйте їх лише з перевірених джерел.

Цікаві факти про кібербезпеку в іграх:

- За даними дослідження, понад 50% гравців стикалися зі спробами злому своїх акаунтів.
- У 2020 році кількість атак на ігрові платформи збільшилась на 300% порівняно з попередніми роками.
- Згідно зі статистикою, 80% користувачів використовують один і той же пароль для кількох облікових записів.
- Двофакторна автентифікація може знизити ризик злому облікового запису на 99,9%.
- Деякі ігри використовують систему блокування IP-адрес для запобігання шахрайству.
- Шкідливі програми можуть ховатися під виглядом популярних ігор чи модифікацій.
- Соціальна інженерія є однією з найпоширеніших тактик шахраїв у ігровій індустрії.
- Багато великих ігрових компаній інвестують мільйони доларів у кібербезпеку для захисту своїх користувачів.

Контрольні питання

1. Що розуміється під терміном кібератака?
2. Які причини привабливості ігор для атак?
3. Що таке чіт-коди в іграх? Для чого використовуються?
4. Що таке моди?
5. Як проводиться атака на пам’ять?
6. Як проводиться атака шкідливі моди та «безплатні чіти»?
7. У чому суть атаки на файли збереження?
8. Як проводиться атака на ігровий час?
9. Що таке фішингові атаки?
10. У чому суть DDoS-атак?

Лекція 12. Зберігання даних

1. Структура та правила створення XML
2. Робота на C# з XML-файлами
3. Структура та правила створення JSON
4. Робота на C# з JSON -файлами

1. Структура та правила створення XML

Слабоструктуровані дані, такі як XML та JSON, використовуються для зберігання та передачі даних, але мають різні характеристики та застосування.

XML (Extensible Markup Language):

- Структура: XML використовує теги для визначення структури даних. Це робить його дуже гнучким, але також може бути громіздким.
- Читабельність: XML легко читається як людьми, так і машинами, але може бути складним для обробки через велику кількість тегів.
- Використання: Часто використовується в веб-сервісах, конфігураційних файлах та для обміну даними між різними системами

JSON (JavaScript Object Notation):

- Структура: JSON використовує пари ключ-значення, що робить його легшим та компактнішим у порівнянні з XML.
- Читабельність: JSON легко читається та обробляється, особливо в JavaScript, що робить його популярним у веб-розробці.
- Використання: Широко використовується в веб-додатках для передачі даних між клієнтом та сервером.

Обидва формати мають свої переваги та недоліки, і вибір між ними залежить від конкретних вимог проекту.

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>
```

За тегами XML зрозуміло, які дані ви переглядаєте. Наприклад, ясно, що це дані про кота, і можна легко визначити його ім'я, вік тощо. Завдяки можливості створювати теги, що визначають майже будь-яку структуру даних, мова XML розширюється.

Правильно сформований XML-файл має відповідати дуже строгим правилам. Якщо він не відповідає цим правилам, не працює XML.

Наприклад, у попередньому прикладі кожен тег, що відкриває, має відповідний тег, що закриває, тому в даному прикладі дотримано одне з правил правильно сформованого XML-файлу.

Базовый синтаксис XML:

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

Декларація (оголошення) XML містить відомості, які готують процесор XML до синтаксичного аналізу документа XML.

Наступний синтаксис показує оголошення XML:

```
<?xml
   version = "version_number"
   encoding = "encoding_declaration"
   standalone = "standalone_status"
?>
```

Декларація XML складається з версії XML, кодування символів та/або автономного статусу. Декларація є необов'язковою. *Якщо декларація XML присутня, вона має бути першою*.

Таблиця 2.1 Атрибути та їх значення в декларації XML

Параметр	Parameter_value	Parameter_description
Версія	1.0	Задає версію стандарту XML.
Кодування	UTF-8, UTF-16, ISO-10646-UCS-2, ISO-10646-UCS-4, ISO-8859-1 до ISO-8859-9, ISO-2022-JP, Shift_JIS, EUC-JP	Він визначає кодування символів у документі. UTF-8 – кодування за замовчуванням.
Автономний	Так чи ні	Він повідомляє синтаксичному аналізатору, чи залежить документ від інформації із зовнішнього джерела, такого як визначення типу зовнішнього документа (DTD) для свого вмісту. Значення за замовчуванням – ні. Встановлення цього значення yes повідомляє процесору, що для синтаксичного аналізу документа не потрібно ніяких зовнішніх оголошень.

Наприклад:

```
<?xml version="1.0" encoding="UTF-8" ?>
```

UTF-8 використовує 8 біт для представлення символів. UTF-16 для представлення символів використовує 16 біт.

Оголошення типу XML-документа, широко відоме як DTD, дозволяє точно описати мову XML. DTD перевіряють словниковий запас та правильність структури XML-

документів на відповідність граматичним правилам відповідної мови XML. XML DTD може бути вказаний усередині документа або може зберігатися в окремому документі.

Приклад внутрішнього DTD:

```
<?xml version = "1.0" encoding = "UTF-8" standalone = "yes" ?>
<!DOCTYPE address [
      <!ELEMENT address (name,company,phone)>
      <!ELEMENT name (#PCDATA)>
      <!ELEMENT company (#PCDATA)>
      <!ELEMENT phone (#PCDATA)>
]>

<address>
      <name> ... </name>
      <company> ... </company>
      <phone> ... </phone>
</address>
```

Коментарі не є обов'язковими. Додавання коментарів допомагає зрозуміти зміст документа. Починається з <!- і закінчується ->.

Наприклад:

```
<!-- Add your comment here -->
```

Теги працюють парами, крім оголошень. Кожна пара тегів складається з тега, що відкриває (початковий тег) і тега, що закриває (кінцевий тег). Імена тегів розташовуються у <>. Для конкретної пари тегів початковий та кінцевий теги мають бути ідентичними, за винятком того, що кінцевий тег має / після <.

Наприклад:

```
<name>...</name>
```

Теги чутливі до регістру. Приклад невірного тегу:

```
<name>...</Name>
```

Все, що знаходиться між тегами, що відкриває і закриває, називається *зміст*.

Відкриваючий тег, контент і тег, що закриває, в цілому називається *елемент*.

Елементи можуть містити *атрибути*.

Наприклад маємо:

```
<age>20</age>
```

age – це ім'я елемента (або елемент); 20 – зміст;

Якщо між тегами немає контенту, це називається *порожні теги*.

Наприклад:

```
<result></result>
```
або
```
<result/>
```

XML – документ повинен містити рівно один кореневий елемент.

Теги мають бути розташовані строго одне всередині одного.

Наприклад:

```
<b><u>This text is bold and italic</u></b>
```

Атрибут елемента розміщується після імені тега у початковому тегу. Можна додати більше одного атрибута до одного елемента з різними іменами атрибутів. Значення атрибутів мають бути поміщені в лапки. Елемент не може містити декілька атрибутів з однаковим назвою.

Атрибут XML має наступний синтаксис

```
<element-name attribute1 attribute2 > ....content.. < /element-name>
```

де attribute1 та attribute2 має наступну форму:

```
name = "value"
```

Наприклад:

```
<teacher subject="English">
   <name>Mr. John</name>.
   <qualification>Graduate </qualification>
</teacher>
```

Ще один приклад:

```
<garden>
   <plants category = "flowers" />
   <plants category = "shrubs"> </plants>
</garden>
```

XML-сутності – це спосіб представлення спеціальних символів. Деякі символи (наприклад, & та < тощо) зарезервовані в XML. Їх називають *спеціальні символи* і вони не можуть бути використані безпосередньо для інших цілей.

Символ	Опис	Ім'я сутності
"	Кавичка (double цитата)	&quot;
&	Амперсант	&amp;
'	Апостроф (одинарна лапка)	&apos;
<	Знак менший	&lt;
>	Знак більше	&gt;

Наприклад:

```
<friend>
    <name>My friends are Alice &amp; Jane</name>
</friend>
```

На додаток до правильно сформованих даних з тегами XML-системи зазвичай використовують два додаткові компоненти:

- схеми;
- перетворення.

Схема XML відома як *XML Schema Definition (XSD)*. Він використовується для опису та перевірки структури та змісту XML-даних. Схема XML визначає елементи, атрибути та типи даних. Елемент схеми підтримує простір імен. Це схоже на схему бази даних, яка описує дані у базі даних.

Основна ідея схем XML полягає в тому, що вони описують допустимий формат, який може приймати документ XML.

Приклад схеми:

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema">
   <xs:element name = "contact">
      <xs:complexType>
         <xs:sequence>
            <xs:element name = "name" type = "xs:string" />
            <xs:element name = "company" type = "xs:string" />
            <xs:element name = "phone" type = "xs:int" />
         </xs:sequence>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

Обумовлені прості типи:

```
xs: integer, xs: boolean, xs: string, xs: date.
```

Складний тип – це контейнер для інших визначень елементів. Це дозволяє вказати, які дочірні елементи може містити елемент, та надати деяку структуру:

```
<xs:element name = "Address">
   <xs:complexType>
   </xs:complexType>
</xs:element>
```

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

XML дозволяє ефективно використовувати та повторно використовувати дані. Механізм повторного використання даних називається *перетворенням XSLT* (або просто перетворенням).

Припустимо, що в базі даних А дані про продаж зберігаються в таблиці, зручній для відділу продажу. У базі даних Б зберігаються дані про доходи та витрати у таблиці, спеціально розробленої для бухгалтерії. База Б може використовувати перетворення, щоб прийняти дані від бази даних А і помістити їх у відповідні таблиці.

Поєднання файлу даних, схеми та перетворення утворює базову систему XML. Файл даних перевіряється на відповідність до правил схеми, а потім передається будь-яким придатним способом для перетворення.

XSLT найчастіше використовується для перетворення даних між різними схемами XML або перетворення даних XML на веб-сторінки або документи PDF.

2. Робота на C# з XML-файлами

Розглянемо приклад створення XML-файлу з результатами гри. Наприклад, в грі в інтерфейсі показується кількість життя гравця у процентах, сума накопичених грошей, кількість підібраних артефактів та назви артефактів. Розробимо функції для створення файлу з такими показниками та отримання даних з файлу.

Створення XML-файлу з результатами гри:

```
static void CreateXMLFile(string xmlName, string name,
                          int life, int money, string[] artifacts)
{
    XmlDocument xDoc = new XmlDocument();
    XmlElement xRoot = xDoc.CreateElement("users");
    xDoc.AppendChild(xRoot);

    XmlElement userNode = xDoc.CreateElement("user");

    XmlElement nameNode = xDoc.CreateElement("name");
    nameNode.InnerText = name;
    userNode.AppendChild(nameNode);

    XmlElement lifeNode = xDoc.CreateElement("life");
    lifeNode.InnerText = life.ToString();
    userNode.AppendChild(lifeNode);

    XmlElement moneyNode = xDoc.CreateElement("money");
    moneyNode.InnerText = money.ToString();
    userNode.AppendChild(moneyNode);

    XmlElement numNode = xDoc.CreateElement("num");
    numNode.InnerText = artifacts.Length.ToString();
    userNode.AppendChild(numNode);

    XmlElement artifactsNode = xDoc.CreateElement("artifacts");
    artifactsNode.InnerText = string.Join(",", artifacts);
    userNode.AppendChild(artifactsNode);

    xRoot.AppendChild(userNode);

    xDoc.Save(xmlName);
}
```

Якщо потрібно доповнювати інформацію про інших гравців, то можна скористатися наступною функцією:

```
static void AddToXMLFile(string xmlName, string name,
                         int life, int money, string[] artifacts)
{
    XmlDocument xDoc = new XmlDocument();
    xDoc.Load(xmlName);
    XmlElement xRoot = xDoc.DocumentElement;

    XmlElement userNode = xDoc.CreateElement("user");

    XmlElement nameNode = xDoc.CreateElement("name");
    nameNode.InnerText = name;
    userNode.AppendChild(nameNode);

    XmlElement lifeNode = xDoc.CreateElement("life");
    lifeNode.InnerText = life.ToString();
    userNode.AppendChild(lifeNode);

    XmlElement moneyNode = xDoc.CreateElement("money");
    moneyNode.InnerText = money.ToString();
    userNode.AppendChild(moneyNode);
```

```
            XmlElement numNode = xDoc.CreateElement("num");
            numNode.InnerText = artifacts.Length.ToString();
            userNode.AppendChild(numNode);

            XmlElement artifactsNode = xDoc.CreateElement("artifacts");
            artifactsNode.InnerText = string.Join(",", artifacts);
            userNode.AppendChild(artifactsNode);

            xRoot.AppendChild(userNode);

            xDoc.Save(xmlName);
        }
```

Читання XML-файлу з результатами гри:

```
static void ReadFromXMLFile(string xmlName)
{
    var doc = new XmlDocument();
    doc.Load(xmlName);

    XmlNodeList userNodes = doc.SelectNodes("/users/user");
    if (userNodes == null || userNodes.Count == 0)
    {
        Console.WriteLine("No nodes found for <user>");
        return;
    }

    foreach (XmlNode userNode in userNodes)
    {
        string name = userNode.SelectSingleNode("name").InnerText;
        string life = userNode.SelectSingleNode("life").InnerText;
        string money = userNode.SelectSingleNode("money").InnerText;
        string num = userNode.SelectSingleNode("num").InnerText;
        string artifacts = userNode.SelectSingleNode("artifacts").InnerText;

        Console.WriteLine($"name={name}, life={life}%, money={money}, artifacts={artifacts}");
    }
}
```

Серіалізація в XML у C# – це перетворення об’єкта (або списку об’єктів) у XML-файл так, щоб його можна було зберегти, передати мережею або прочитати іншою програмою. Зворотній процес називається *десеріалізація* – відновлення об’єкта з XML.

Серіалізація в XML C# виконується за допомогою класу XmlSerializer з простору імен System.Xml.Serialization.

Для цього потрібно створити клас, який описує дані для XML-файлу. Клас має бути public, і зазвичай потрібен порожній конструктор (або авто-ініціалізація). Серіалізуються public властивості/поля (можна керувати атрибутами XmlRoot, XmlElement, XmlAttribute, XmlIgnore).

Для прикладу, який розглянуто вище створимо окремий клас User:

```
using System.Xml.Serialization;
```

```
[XmlRoot("user")]
public class User
{
    [XmlElement("name")]
    public string Name { get; set; } = "";

    [XmlElement("life")]
    public int Life { get; set; } = 0;

    [XmlElement("money")]
    public int Money { get; set; } = 0;

    [XmlElement("artifacts")]
    public string ArtifactsCsv { get; set; } = "";

    [XmlIgnore]
    public string[] Artifacts
    {
        get => string.IsNullOrWhiteSpace(ArtifactsCsv)
            ? Array.Empty<string>()
            : ArtifactsCsv
                .Split(',', (char)StringSplitOptions.RemoveEmptyEntries)
                .Select(x => x.Trim())
                .Where(x => x.Length > 0)
                .ToArray();

        set => ArtifactsCsv = (value == null || value.Length == 0)
            ? ""
            : string.Join(", ", value.Select(x => (x ?? "").Trim()).Where(x => x.Length > 0));
    }

    [XmlElement("num")]
    public int Num
    {
        get => Artifacts.Length;
        set { }
    }
}
```

Серіалізація списку об'єктів:

```
string fileName = "settingsSer.xml";
var players = new List<User>
{
    new User { Name="Olena", Life=100, Money=250, Artifacts = new[] {"Ring","Amulet","Key"} },
    new User { Name="Ivan",  Life=80,  Money=120, Artifacts = new[] {"Map","Potion"} }
};

var serializer = new XmlSerializer(typeof(List<User>));

var writer = XmlWriter.Create(fileName, new XmlWriterSettings { Indent = true });
serializer.Serialize(writer, players);
```

В результаті маємо наступний файл:

```xml
<?xml version="1.0" encoding="utf-8"?>
<ArrayOfUser xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <User>
    <name>Olena</name>
    <life>100</life>
    <money>250</money>
    <artifacts>Ring, Amulet, Key</artifacts>
    <num>3</num>
  </User>
  <User>
    <name>Ivan</name>
    <life>80</life>
    <money>120</money>
    <artifacts>Map, Potion</artifacts>
    <num>2</num>
  </User>
</ArrayOfUser>
```

Десеріалізація списку об'єктів:

```
var serializer = new XmlSerializer(typeof(List<User>));

var fs = File.OpenRead(fileName);
List<User> players = (List<User>)serializer.Deserialize(fs);

foreach (var p in players)
{
    Console.WriteLine($"Name={p.Name}, Life={p.Life}, Money={p.Money}, Num={p.Num}");
    Console.WriteLine("Artifacts: " + string.Join(" - ", p.Artifacts));
    Console.WriteLine();
}
```

3. Структура та правила створення JSON

JSON (JavaScript Object Notation) – це простий, популярний і легкий для читання формат обміну даними. Він був розроблений з метою забезпечення зручного способу передачі інформації між додатками.

Історія JSON бере свій початок 2001 року, коли Дуглас Крокфорд представив цей формат, щоб полегшити роботу з даними в Ajax-додатках.

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.

4. Робота на C# з JSON -файлами

Розглянемо зберігання даних, описаних вище, у JSON-файлу. Розглянемо процес серіалізації і десеріалізації. Клас даних описаний вище.

Потрібно в проекті встановити System.Text.Json.

Серіалізація списку об'єктів:

```
string fileName = "settingsSer.json";
var users = new List<User>
{
    new User { Name="Olena", Life=100, Money=250, Artifacts = new[] {"Ring","Amulet","Key"} },
    new User { Name="Ivan",  Life=80,  Money=120, Artifacts = new[] {"Map","Potion"} }
};
//string jsonString = JsonSerializer.Serialize(users);
var options = new JsonSerializerOptions
{
    WriteIndented = true,
    Encoder = JavaScriptEncoder.UnsafeRelaxedJsonEscaping
};

string json = JsonSerializer.Serialize(users, options);
File.WriteAllText(fileName, json);
```

В результаті маємо наступний файл:

```
[
  {
    "Name": "Olena",
    "Life": 100,
    "Money": 250,
    "ArtifactsCsv": "Ring, Amulet, Key",
    "Artifacts": [
      "Ring",
      "Amulet",
      "Key"
    ],
    "Num": 3
  },
  {
    "Name": "Ivan",
    "Life": 80,
    "Money": 120,
    "ArtifactsCsv": "Map, Potion",
    "Artifacts": [
      "Map",
      "Potion"
    ],
    "Num": 2
  }
]
```

Десеріалізація списку об'єктів:

```
string json = File.ReadAllText(fileName);
List<User> users = JsonSerializer.Deserialize<List<User>>(json);

if (users == null)
{
    Console.WriteLine("No data in JSON.");
    return;
}

foreach (var p in users)
{
    Console.WriteLine($"Name={p.Name}, Life={p.Life}, Money={p.Money}, Num={p.Num}");
    Console.WriteLine("Artifacts: " + string.Join(" - ", p.Artifacts));
    Console.WriteLine();
}
```

Контрольні питання

1. Яка структура XML та JSON?
2. Чим відрізняється структура XML та JSON?
3. Які правила створення XML?
4. Як на C# працювати з XML-файлами?
5. Які правила створення JSON-файлів?
6. Як на C# працювати з JSON-файлами?

Лекція 13. Методи захисту від злому в іграх

1. Захист ігрових даних
2. Обфускація

1. Захист ігрових даних

Якщо Вам потрібна справжня безпека, то найкращий варіант – зберігати дані на сервері, де користувачі не можуть їх змінити. Щоб це працювало, програма не повинна надсилати дані безпосередньо на сервер, оскільки користувачі все одно можуть маніпулювати ними. Замість цього програма може лише надсилати команди на сервер, дозволяти серверу змінювати дані, а потім надсилати результати назад до програми.

Розглянемо простий спосіб збереження даних, тобто локально та можливі методи захисту.

Використання простих json файлів у папці збережень дозволяє користувачам легко маніпулювати характеристиками персонажа. Зловмисник може з легкістю маніпулювати даними свого акаунту, використовуючи внутрішні дані, завантажені на пристрій. Це перша і основна проблема, яку потрібно вирішити.

Шифрування внутрішньо-ігрових даних важливе для забезпечення безпеки та унеможливлення несанкціонованого доступу до конфіденційної інформації. Цей процес забезпечує захист від можливих атак та витоків даних, а також дозволяє зберігати важливу інформацію, таку як паролі або ігровий прогрес, у безпечному стані. Шифрування внутрішньо-ігрових даних включає в себе застосування різноманітних шифрувальних алгоритмів та практик, щоб забезпечити конфіденційність та цілісність інформації в межах гри.

Основні методи захисту JSON в іграх:

- Шифрування (Encryption) – перетворення JSON-тексту в нечитабельний вигляд перед записом на диск;
- Бінарна серіалізація – збереження даних у бінарному форматі (наприклад, MessagePack, Protobuf), що ускладнює пряме редагування;
- Контрольні суми (Checksums/Hashing) – створення унікального хешу файлу. При завантаженні гра перевіряє, чи не був файл змінений (hash mismatch), і відхиляє його;
- Обфускація – заплутування ключів та значень у JSON, щоб зробити їх менш зрозумілими для людини;
- Серверне зберігання – найбільш надійний метод зберігання критичних даних (рівень, валюта) на сервері, а не на пристрої користувача.

Base64-перетворення в іграх використовується для безпечної передачі бінарних даних (зображень, збережень, мережевих пакетів) через текстові протоколи, перетворюючи їх на набір із 64 ASCII-символів. Це забезпечує цілісність даних під час обміну, захищаючи від пошкоджень, але збільшує розмір файлів приблизно на 33%.

Base64 – це метод кодування двійкових (binary) даних у текстовий формат (ASCII), використовуючи лише 64 безпечні друковані символи (A-Z, a-z, 0-9, +, /). Суть полягає у перетворенні 3 байт (24 біта) даних у 4 символи, що дозволяє передавати файли або зображення через текстові протоколи (email, JSON, HTTP) без пошкодження даних.

Ключові аспекти Base64 в ігровій індустрії:

- Кодування збережень – рядки збережень (save strings) часто кодуються в Base64 для використання у хмарних сховищах або обміну між гравцями, що також є елементарною формою «обфускації» (утруднення читання) даних.
- Мережевий обмін – дозволяє передавати бінарні дані, такі як аватари гравців або ігрові об'єкти, через системи, які підтримують лише текст.

- Впровадження ресурсів – маленькі зображення або текстури можуть бути вбудовані прямо в код або JSON-файли за допомогою Base64, що зменшує кількість запитів до сервера.
- Безпека (URL-safe) – у браузерних або онлайн-іграх використовується Base64URL (із заміною символів + та / на - та _), що запобігає помилкам при передачі даних в URL-адресах.

Алгоритм кодування Base64 (покроково):

- Розбиття на байти – вихідні дані діляться групи по 3 байти (3 x 8=24 біта).
- Перегрупування в 6 біт – кожен 24-бітний блок розбивається на 4 блоки по 6 біт (4 x 6=24 біта).
- Індексація – отримані 6-бітові значення (0-63) перетворюються на відповідні символи ASCII-таблиці (A-Z, a-z, 0-9, +, /).
- Доповнення (Padding) – якщо вихідний блок містить менше 3 байт, використовується символ = для заповнення, щоб підсумковий рядок був кратний 4 символам.

Наприклад, маємо слово "Man" (3 байта). Запишемо слово в бітах: M (77), a (97), n (112) → 01001101 01100001 01101110 (24 біта). Розбивка по 6 біт: 010011 (19), 010110 (22), 000101 (5), 101110 (46). Результат: TWFu.

Розглянемо роботу з Base64 на C#:

```
static void CodeToBase64(string fileName)
{
    byte[] bytes = File.ReadAllBytes(fileName);
    string base64 = Convert.ToBase64String(bytes);

    File.WriteAllText("data_b64.txt", base64);

}

static void DecodeFromBase64(string fileName)
{
    string base64 = File.ReadAllText(fileName);
    byte[] bytes = Convert.FromBase64String(base64);

    File.WriteAllBytes("settingsBase64.json", bytes);
}
```

```
...
 string fileName = "settingsSer.json";
 CodeToBase64(fileName);

 fileName = "data_b64.txt";
 DecodeFromBase64(fileName);
```

Файл settingsSer.json:

File Edit View


```
[
  {
    "Name": "Olena",
    "Life": 100,
    "Money": 250,
    "ArtifactsCsv": "Ring, Amulet, Key",
    "Artifacts": [
      "Ring",
      "Amulet",
      "Key"
    ],
    "Num": 3
  },
  {
    "Name": "Ivan",
    "Life": 80,
    "Money": 120,
    "ArtifactsCsv": "Map, Potion",
    "Artifacts": [
      "Map",
      "Potion"
    ],
    "Num": 2
  }
]
```


Файл data_b64.txt:

File Edit View


```
Ww0KICB7DQogICAgIk5hbWUiOiAiT2xlbmEiLA0KICAgICJMaWZlIjogMTAwLA0KICAgICJNb251eSI6IDI1MCwNCiAgICAiQXJ0aWZhY3RzQ3N2IjogIlJpbmcsIEFtdWxldCwgS2V5IiwNCiAgICAiQXJ0aWZhY3RzIjogWw0KICAgICAgIlJpbmciLA0KICAgICAgIkFtdWxldCIsDQogICAgICAiS2V5Ig0KICAgIF0sDQogICAgIk51bSI6IDMNCiAgfSwNCiAgew0KICAgICJOYW1lIjogIkl2YW4iLA0KICAgICJMaWZlIjogODAsDQogICAgIk1vbmV5IjogMTIwLA0KICAgICJBcnRpZmFjdHNDc3YiOiAiTWFwLCBQb3Rpb24iLA0KICAgICJBcnRpZmFjdHMiOiBbDQogICAgICAiTWFwIiwNCiAgICAgICJQb3Rpb24iDQogICAgXSwNCiAgICAiTnVtIjogMg0KICB9DQpd
```


Файл settingsBase64.json:

File Edit View


```
[
  {
    "Name": "Olena",
    "Life": 100,
    "Money": 250,
    "ArtifactsCsv": "Ring, Amulet, Key",
    "Artifacts": [
      "Ring",
      "Amulet",
      "Key"
    ],
    "Num": 3
  },
  {
    "Name": "Ivan",
    "Life": 80,
    "Money": 120,
    "ArtifactsCsv": "Map, Potion",
    "Artifacts": [
      "Map",
      "Potion"
    ],
    "Num": 2
  }
]
```


Хоча Base64 зручний, він не підходить для великих даних через зростання розміру файлу.

Дані, закодовані в Base64, легко декодувати назад, тому що це просто інший формат подання, а не приховування інформації. Для приховування інформації використовують шифрування даних.

Сучасні методи шифрування поділяються на потокові та блочні.

Характерною особливістю блокових криптоалгоритмів є той факт, що в ході своєї роботи вони роблять перетворення блоку вхідної інформації фіксованої довжини і отримують результуючий блок того ж обсягу, але недоступний для прочитання стороннім особам, які не мають ключа.

Таким чином, схему роботу блокового шифру можна описати функціями

Z = EnCrypt(X, Key)

та

X = DeCrypt(Z, Key)

Ключ (Key) є параметром блокового криптоалгоритму і є деяким блоком двійкової інформації фіксованого розміру. Вхідний (X) та зашифрований (Z) блоки даних також мають фіксовану розрядність, рівну між собою, але необов'язково рівну довжині ключа.

Основним принципом побудови сучасних блокових шифрів є умова їхньої практичної стійкості до усіх відомих атак поряд з можливістю ефективної реалізації алгоритмів шифрування на різних платформах.

Стійкість блокового шифру визначається властивостями його раундових функцій, типом ключового суматора, довжиною блоку, довжиною ключа та кількістю раундів шифрування;

Стандарт AES (Advanced Encryption Standard) є стандартом шифрування Сполучених Штатів Америки та включає в себе алгоритм Rijndael.

Алгоритм Rijndael розроблений двома фахівцями з криптографії з Бельгії Rijmen та Daemen, вимовляється як «рейндал». Він є нетрадиційним блоковим шифром, оскільки не використовує мережу Фейштеля для криптоперетворень.

Rijndael є блоковим шифром з довжиною блоку n та довжиною ключа n_k, які можуть приймати одне з трьох значень: 128, 192, 256 бітів. Алгоритм складається з n_r раундів, де параметр n_r визначається за числами n та n_k згідно з таблиці 13.1.

Таблиця 13.1. Кількість раундів в алгоритмі шифрування Rijndael

n_r	$n=128$	$n=192$	$n=256$
$n_k=128$	10	12	14
$n_k=192$	12	12	14
$n_k=256$	14	14	14

В Rijndael використовуються чотири базові перетворення, означення яких наведено нижче:

1) K (AddRoundKey) – додавання до вхідного повідомлення раундо вого ключа;

2) S (ByteSub) – застосування нелінійних вузлів заміни;

3) R (ShiftRow) – циклічний зсув рядків;

4) M (MixColumn) – перемішування стовпців.

Зашифрування відкритого тексту полягає у виконанні таких дій:

1) обчисленні (за допомогою окремої процедури – розкладу ключів) $n_r + 1$ раундових ключів за ключем шифрування;

2) додаванні першого раундового ключа до відкритого тексту;

3) виконанні $n_r - 1$ проміжних раундів;

4) виконанні останнього раунду.

Кожен проміжний раунд полягає в послідовному застосуванні до вхідного повідомлення перетворень S, R, M, K; в останньому раунді здійснюються перетворення S, R, K.

Для реалізації шифрування AES у C# використовується простір імен *System.Security.Cryptography* та клас Aes. Основний підхід полягає у створенні екземпляра Aes, використанні ключа (Key) та вектору ініціалізації (IV) для створення ICryptoTransform через CreateEncryptor або CreateDecryptor, а потім обробку даних за допомогою CryptoStream.

Вектор ініціалізації (Initialization Vector, IV) – це випадкова або псевдовипадкова послідовність байтів, яка використовується разом із секретним ключем у симетричному шифруванні для гарантування унікальності кожного зашифрованого блоку даних. Він запобігає появі однакових шаблонів у зашифрованому тексті, роблячи шифр стійким до атак.

Без IV однаковий текст, зашифрований одним і тим же ключем, завжди виглядатиме однаково. Це дозволяє хакерам знаходити закономірності (наприклад, розпізнати стандартні заголовки повідомлень).

Вектор ініціалізації IV змішується з першим блоком даних (зазвичай через операцію XOR) перед початком шифрування. У сучасних режимах (наприклад, CBC) результат шифрування першого блоку впливає на наступний, створюючи «ланцюжок».

IV не обов’язково тримати в секреті – його часто передають разом із зашифрованим повідомленням у відкритому вигляді. Для кожної нової сесії шифрування потрібно генерувати новий IV. Повторне використання однієї й тієї ж комбінації «ключ + IV» критично знижує безпеку.

В алгоритмах AES використання IV залежить від конкретного режиму роботи. Використовуються такі режими роботи:

- AES-CBC (Cipher Block Chaining) – у цьому режимі кожен блок даних перед шифруванням «змішується» з попереднім зашифрованим блоком. Оскільки для першого блоку ще немає «попереднього», IV виступає в ролі цього першого віртуального блоку. Перший блок тексту проходить операцію XOR з IV, і лише після цього результат шифрується ключем AES.
- AES-CTR (Counter Mode) – цей режим перетворює AES на потоковий шифр. Тут IV – число, що використовується один раз і є частиною «лічильника». AES шифрує не сам текст, а комбінацію «IV + лічильник» (який збільшується для кожного блоку). Отриманий «випадковий» потік даних потім змішується (XOR) з текстом.
- AES-GCM (Galois/Counter Mode) – найсучасніший режим, який не тільки шифрує, а й перевіряє цілісність даних. Працює схоже на CTR, але використовує IV (зазвичай 96 біт) для створення унікального ключа для кожного повідомлення. Якщо дані будуть змінені під час передачі, алгоритм це виявить завдяки «тегу аутентифікації», який генерується разом з IV.

Aes – це абстрактний клас, від якого успадковуються всі реалізації алгоритму AES. У свою чергу, клас Aes сам є спадкоємцем іншого абстрактного класу – SymmetricAlgorithm, який є, як випливає з назви, симетричним алгоритмом шифрування. У .NET всі класи, похідні від SymmetricAlgorithm класу, використовують режим зчеплення блоків тексту (англ. Cipher Block Chaining, CBC). Режим CBC використовується за замовчуванням, але за бажанням його можна змінити.

Щоб отримати об'єкт для шифрування за заданим алгоритмом, достатньо викликати метод Create(). Наприклад, скористаємося цим методом і створимо об'єкт для шифрування даних з використанням AES та подивимося на його налаштування за замовчуванням та деякі інші властивості:

```
Aes aes = Aes.Create();
Console.WriteLine($"{aes.GetType()}");
Console.WriteLine($"Размір ключа: {aes.KeySize}");
Console.WriteLine($"Размір блоку: {aes.BlockSize}");
Console.WriteLine($"Режим роботи: {aes.Mode}");
Console.WriteLine("Розміри ключів, що підтримуються симетричним алгоритмом:");
foreach (var keySize in aes.LegalKeySizes)
{
    Console.WriteLine($"{keySize.MinSize} - {keySize.MaxSize} бит (с шагом {keySize.SkipSize})");
}
Console.WriteLine("Розміри блоків, що підтримуються симетричним алгоритмом:");
foreach (var blockSize in aes.LegalBlockSizes)
{
    Console.WriteLine($"{blockSize.MinSize} - {blockSize.MaxSize} бит");
}
Console.WriteLine($"Вектор ініціалізації: {Convert.ToHexString(aes.IV)}");
Console.WriteLine($"Ключ: {Convert.ToHexString(aes.Key)}");
```

```
System.Security.Cryptography.AesImplementation
Разм?р ключа: 256
Разм?р блоку: 128
Режим роботи: CBC
Розм?ри ключ?в, що п?дтримуються симетричним алгоритмом:
128 - 256 бит (с шагом 64)
Розм?ри блок?в, що п?дтримуються симетричним алгоритмом:
128 - 128 бит
Вектор ?н?ц?ал?зац?ї: 4D7AE664F78D13C1548E61FC303BA2BD
Ключ: A43EB10A60FE6A4D7F088AE499F3457669A5066946DB061C9FAE4865EEC01656
```

Як видно з виведення консолі, для AES ми можемо використовувати ключі розміром 128 біт, 192 біти та 256 біт. За замовчуванням, в отриманому об'єкті вже створено ключ розміром 256 біт і вектор ініціалізації в 128 біт. Але, за бажання, ми можемо змінити ці значення. Зазвичай беруть ключ 32 байти при використанні AES-256, 16 байт – AES-128.

Класи симетричного шифрування (Aes, Des тощо) використовуються зі спеціальним класом потоку – CryptoStream, який шифрує дані, раховані в потік. Клас CryptoStream ініціалізується за допомогою класу керованого потоку та класу, що реалізує інтерфейс ICryptoTransform, а також із зазначенням значення перерахування CryptoStreamMode, яке вказує тип доступу, дозволеного для CryptoStream.

Клас CryptoStream можна ініціалізувати за допомогою будь-якого класу, який є похідним Stream від класу, включаючи FileStream, MemoryStream тощо.

Наприклад, щоб створити об'єкт типу CryptoStream для шифрування даних, можна використовувати такий конструктор:

```
Aes aes = Aes.Create();

string sourceFileName = "settingsSer.json";
string outputFileName = "settingsjson.enc";

FileStream inStream = new FileStream(sourceFileName, FileMode.Open);
FileStream outStream = new FileStream(outputFileName, FileMode.Create);

CryptoStream encStream = new CryptoStream(outStream,
                                          aes.CreateEncryptor(aes.Key, aes.IV),
                                          CryptoStreamMode.Write);
```

Ось як може виглядати процес шифрування та дешифрування файлу з використанням алгоритму AES:

//виклик функцій

```
string sourceFileName = "settingsSer.json";
string outputFileName = "settingsjson.enc";
string destFile = "settings.json";

string key = "secret key";
string ivSecret = "vector";

EcryptFile(sourceFileName, outputFileName, key, ivSecret);
DecryptFile(outputFileName, destFile, key, ivSecret);
```

//перетворення ключа та вектору

```
private static byte[] GetIV(string ivSecret)
{
    MD5 md5 = MD5.Create();
    return md5.ComputeHash(Encoding.UTF8.GetBytes(ivSecret));
}

private static byte[] GetKey(string key)
{
    SHA256 sha256 = SHA256.Create();
    return sha256.ComputeHash(Encoding.UTF8.GetBytes(key));
}
```

//шифрування даних

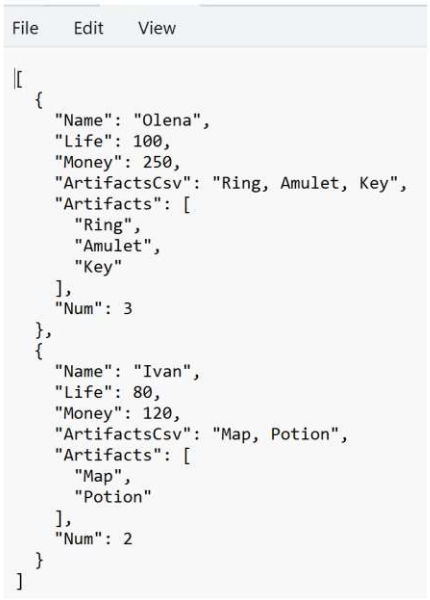
```
static void EcryptFile(string sourceFileName, string outputFileName,
                       string key, string ivSecret)
{
    Aes aes = Aes.Create();

    aes.IV = GetIV(ivSecret);
    aes.Key = GetKey(key);

    FileStream inStream = new FileStream(sourceFileName, FileMode.Open);
    FileStream outStream = new FileStream(outputFileName, FileMode.Create);

    CryptoStream encStream = new CryptoStream(outStream,
                                              aes.CreateEncryptor(aes.Key, aes.IV),
                                              CryptoStreamMode.Write);
    long readTotal = 0;

    int len;
    int tempSize = 100; //розмір тимчасового сховища
    byte[] bin = new byte[tempSize]; //тимчасове Сховище для зашифрованої інформації
```

```
    while (readTotal < inStream.Length)
    {
        len = inStream.Read(bin, 0, tempSize);
        encStream.Write(bin, 0, len);
        readTotal = readTotal + len;
        Console.WriteLine($"{readTotal} байт оброблено");
    }

    encStream.Close();
    outStream.Close();
    inStream.Close();
}
```

//дешифрування даних

```
private static void DecryptFile(string sourceFile, string destFile,
                                string key, string ivSecret)
{
    using FileStream fileStream = new(sourceFile, FileMode.Open);
    using Aes aes = Aes.Create();

    aes.IV = GetIV(ivSecret);
    aes.Key = GetKey(key);

    using CryptoStream cryptoStream = new(fileStream,
                               aes.CreateDecryptor(aes.Key, aes.IV),
                                         CryptoStreamMode.Read);

    using FileStream outStream = new FileStream(destFile, FileMode.Create);

    using BinaryReader decryptReader = new(cryptoStream);
    int tempSize = 10;
    byte[] data;
    while (true)
    {
        data = decryptReader.ReadBytes(tempSize);
        if (data.Length == 0)
            break;
        outStream.Write(data, 0, data.Length);
    }
}
```

Результат:



2. Обфускація

Сканування значень – це процес пошуку відповідних значень у виділеній пам'яті гри та перезапису їх іншими значеннями. Найчастіше використовується для збільшення здоров'я гравця, боєзапасу зброї чи будь-якої іншої цінності, яка могла б дати хакеру несправедливу перевагу у грі.

Як правило, хакери-початківці націлюються лише на значення, які відображаються на екрані та знають, для чого вони використовуються (наприклад, здоров’я гравця, боєприпаси тощо).

Обфускація – це процес ускладнення коду програми або даних з метою утруднення реверс-інжинірингу, тобто забезпечення труднощів у розумінні або відтворенні вихідного коду. Це стандартна практика у сфері програмування та розробки програмного забезпечення для підвищення безпеки та унеможливлення несанкціонованого доступу чи модифікації програмного коду.

Основні аспекти обфускації включають:

- Перейменування – зміна імен змінних, функцій і класів на більш неочевидні та складні для розуміння;
- Заміна констант – заміна числових та рядкових констант на інші значення для утруднення розуміння логіки програми;
- Вставлення непотрібного коду – додавання непотрібного коду, який не впливає на функціональність, але збільшує обсяг та ускладнює аналіз;
- Абстракція умов – заміна зрозумілих умов та логічних виразів більш складними або вигаданими;
- Шифрування рядків – шифрування текстових рядків, таких як рядки, що містять повідомлення про помилки чи ключі;
- Видалення зайвого мета-інформації – вилучення зайвої мета-інформації, яка може бути корисною для реверс-інжинірингу, наприклад, видалення імен методів чи класів.

Обфускація не забезпечує абсолютної безпеки, і великі обфусковані програми все ще можуть бути розібрані. Однак вона створює додаткові труднощі для тих, хто намагається аналізувати або модифікувати вихідний код.

Методи обфускації можна застосовувати для захисту пам'яті програми.

Є різні методи заміни значень змінних, розглянемо один із них з використанням випадкових значень.

Випадкове значення генерується на початку, потім з нього віднімається реальне значення (згодом воно приховується), а потім, коли необхідно, приховане значення віднімається зі згенерованого випадкового значення, при цьому різниця є вихідним числом. Ключ полягає в тому, щоб значення, яке відображається на екрані, мало значення, яке повністю відрізняється від значення змінної, що призведе хакерів до абсолютно неправильного шляху під час сканування.

Наприклад, в попередніх лекціях в скрипте GameController зберігаємо накопичену кількість балів від об'єктів, що піднімаємо, тобто:

```
private int score;
private int Score { get { return score; } set { score = value; } }

private void Awake()
{
    _instance = this;
}

private void Start()
{
    Score = 0;
    HUD.Instance.SetScore(Score.ToString());
}

public void SetScore(int add)
{
    Score += add;
    HUD.Instance.SetScore(Score.ToString());

    if (Score >= 10)
        HUD.Instance.ShowLevelWonWindow();

}
```

Доробимо наш скрипт для використання заміни значень змінних з використанням випадкових значень. Для цього створимо новий скрипт ObfuscateSettings.

```
public class ObfuscateSettings : MonoBehaviour
{
    static int random;

    public static void Initialize()
    {
        random = UnityEngine.Random.Range(10000, 99999);
    }

    public static int Obfuscate(int originalValue)
    {
        return random - originalValue;
    }

    public static int Deobfuscate(int obfuscatedValue)
    {
        return random - obfuscatedValue;
    }
}
```

Зробимо змінив в скрипті GameController:

```
private void Awake()
{
    _instance = this;

    ObfuscateSettings.Initialize();
    Score = ObfuscateSettings.Obfuscate(0);
}

private void Start()
{
    //Score = 0;
    //HUD.Instance.SetScore(Score.ToString());

    HUD.Instance.SetScore(ObfuscateSettings.Deobfuscate(Score).ToString());
}
public void SetScore(int add)
{
    //Score += add;
    //HUD.Instance.SetScore(Score.ToString());

    Score = ObfuscateSettings.Obfuscate(ObfuscateSettings.Deobfuscate(Score) + add);
    HUD.Instance.SetScore(ObfuscateSettings.Deobfuscate(Score).ToString());

    //if (Score >= 10)
    if (ObfuscateSettings.Deobfuscate(Score) >= 10)
        HUD.Instance.ShowLevelWonWindow();

}
```

Замість безпосередньої ініціалізації змінної очок ми ініціалізуємо її на початку в void Awake(). Перш ніж призначати значення за допомогою Obfuscate(value), обов'язково викликайте Initialize().

Тоді під час відображення значень ми деобфускуємо їх на льоту, викликавши Deobfuscate(value) таким чином відображаючи справжні значення. Хакер спробував би знайти 0 або 10 або 20 тощо, але не зміг би їх знайти, оскільки реальні значення зовсім інші.

Honeypot – це вид кібербезпеки, що представляє собою фальшивий об'єкт чи систему, призначений для виявлення, вивчення чи відстеження кіберзлочинців. Honeypot може бути як програмним, так і апаратним засобом, розташованим в мережі з метою привертання уваги потенційних зловмисників.

Основні характеристики honeypot включають:

- Фальшиві дані – honeypot намагається видавати себе за реальну систему чи ресурс, ізолюючи його від решти реальних активів мережі;
- Захоплення даних – honeypot може записувати та аналізувати всю взаємодію зловмисників, зокрема, їхні спроби атак та методи;
- Виявлення загроз – за допомогою honeypot можна виявляти нові та еволюціонуючі загрози, адже вони привертають атаки, які можуть вказувати на нові та не відомі види атак.
- Навчання та аналіз – використання honeypot дозволяє вивчати методи та тактику зловмисників, а також покращувати стратегії кіберзахисту на основі отриманих даних;
- Попередження перед атакою – honeypot може слугувати попередженням перед реальними атаками, дозволяючи вчасно реагувати та захищати реальні системи.

Gameplay-honeypot (ігрові пастки) – це вбудовані у геймплей елементи, які недоступні звичайному гравцю та доступні лише через чити / хаки. Наприклад, предмет за стіною (його «бачать» лише WallHack) або скриня з нульовою нагородою, яку «відкривають» лише боти

або невидимий NPC, з яким взаємодіють лише скрипти. Легальний гравець ніколи не активує honeypot.

Data-honeypot (пастки у даних) – навмисно створені фейкові або контрольні дані. Наприклад, поле в JSON/XML збереженні, яке не використовується грою але змінюється читерами. Наприклад, створення «фейкових» змінних (player_gold_shadow, debug_speed_value), якщо значення змінене це говорить про 100% індикатор злому.

AI / Behavior honeypot – пастки на основі поведінки. Наприклад, ресурс у зоні, до якої неможливо дістатися без teleport hack або ціль, що потребує нелюдської точності реакції або фейкові події, які бачать лише ESP-чити.

Honeypot у іграх – це інтелектуальна пастка, яка дозволяє не боротися з читерами напряму, а змусити їх видати себе самих.

Контрольні питання

1. Як захистити ігрові дані?
2. Для чого використовується шифрування?
3. Що робить Base64-перетворення?
4. Який сучасний стандарт шифрування?
5. Який алгоритм лежить в основі сучасного стандарту шифрування?
6. Що таке режими роботи AES? Для чого використовуються?
7. Зо таке обфускація?
8. Які методи обфускації для захисту ігор ви знаєте?
9. Що таке honeypot?

Рекомендована література

Основна

1. Lanzinger F. 3D Game Development with Unity. Boca Raton: CRC Press, 2022. 414 p.
2. Jackson S. Unity 3D UI Essentials. Birmingham: Packt Publishing, 2015. 280 p.
3. Gupta K. Unity3D Tutorial for Beginners. FutureZenGroup, 2021. 81 p.
4. Creighton R. Unity 3D Game Development by Example. Birmingham: Packt Publishing, 2010. 364 p.

Допоміжна

5. Nakov S., Nakov's Team. Programming Basics with C#. Sofia: SoftUni, 2019. 408 p.
6. Norton T. Learning C# by Developing Games with Unity 3D. Birmingham: Packt Publishing, 2013. 292 p.
7. Miller R. C# for Artists: The Art, Philosophy, and Science of Object-Oriented Programming. Seattle: Pulp Free Press, 2008. 750 p.
8. Coggan A. Unity Game Audio Implementation: A Practical Guide for Beginners. Boca Raton: Taylor & Francis, 2021. 434 p.
9. Sinclair J.-L. Principles of Game Audio and Sound Design. Boca Raton: Taylor & Francis, 2020. 295 p.

Інформаційні ресурси

10. Unity: офіційний вебсайт. URL: https://unity.com/
11. Visual Studio Community: офіційний вебсайт. URL: https://visualstudio.microsoft.com/vs/community/

Unity Asset Store: вебсайт. URL: https://assetstore.unity.com/