



Приватний заклад вищої освіти
Одеський технологічний університет «ШАГ»
Кафедра інформаційних технологій та фундаментальної підготовки

Випускна кваліфікаційна робота бакалавра

**Розробка 3D інтерфейсу людино-машинної взаємодії
(на прикладі Escape Chronicles)**

на здобуття бакалаврського рівня вищої освіти
зі спеціальності «122 Комп'ютерні науки»

Виконавці проекту

№	П.І.Б.	Група
1	Семчук В. Б.	КН-П-211
2	Шеленгівський А. Л.	КН-П-211
3	Врублевська Ю. Д.	КН-П-211

Автор звіту

_____ Семчук Владислав Богданович

Науковий керівник	Доцент кафедри ІТ та ФП, к.т.н., Суліма М.М
-------------------	--

АНОТАЦІЯ

Семчук В. Б. Р Розробка 3D інтерфейсу людино-машинної взаємодії (на прикладі Escape Chronicles): випускна кваліфікаційна робота на здобуття бакалаврського рівня вищої освіти зі спеціальності «122 Комп'ютерні науки». - Одеса: ОТУШ, 2022 - 13 с.

Дана кваліфікаційна робота присвячена розробці кооперативної 3D-гри, яка симулює спільну втечу гравців із віртуальної в'язниці. Метою проєкту є створення інтерактивного середовища, що сприяє розвитку командної взаємодії, стратегічного мислення та логіки.

Актуальність теми зумовлена зростаючим попитом на багатокористувацькі кооперативні ігри, які дозволяють гравцям взаємодіяти у реальному часі, долаючи фізичну відстань. Ринок відеоігор останніми роками демонструє стійкий інтерес до продуктів з елементами соціальної взаємодії, гнучкого геймплею та можливістю проходження у команді. Розробка даного проєкту відповідає цим тенденціям, поєднуючи захопливу тематику втечі з в'язниці із сучасними підходами до дизайну ігрового процесу та мережевої взаємодії.

ЗМІСТ

ВСТУП	3
Технічне завдання до проекту	5
РОЗДІЛ 1. Загальна характеристика системи	11
РОЗДІЛ 2. Gameplay Ability System та Input	14
РОЗДІЛ 3. Характеристики персонажу, ролі та теги	16
РОЗДІЛ 4. GameplayAbilityComponents	19
РОЗДІЛ 5. Пересування персонажу	22
РОЗДІЛ 6. Система збережень та ідентифікація гравців	24
РОЗДІЛ 7. Власники акторів	28
РОЗДІЛ 8. Інвентар та предмети	31
РОЗДІЛ 9. Пошкодження об'єктів	34
РОЗДІЛ 10. Ігровий час та зміна дня і ночі	36
РОЗДІЛ 11. Ввімкнення/вимкнення освітлення за розкладом	38
РОЗДІЛ 12. Ігрові події	40
РОЗДІЛ 13. Музика для різних подій	46
РОЗДІЛ 14. Клієнт-серверне підключення	49

РОЗДІЛ 15. UI	53
РОЗДІЛ 16. Створення ботів	57
РОЗДІЛ 17. Nameplates	61
РОЗДІЛ 18. Штучний інтелект	64
РОЗДІЛ 19. Результати випробувань	68
ВИСНОВКИ	72
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ	74

ТЕХНІЧНЕ ЗАВДАННЯ ДО ПРОЄКТУ
Тривимірний гра-симулятор втечі з в'язниці
(за зразком The Escapists 2)

Загальний опис.

Програмне забезпечення (ПЗ) є тривимірною грою, що моделює життя ув'язненого в умовах охоронюваної в'язниці. Гравець має дотримуватись розпорядку, взаємодіяти з неігровими персонажами (ув'язненими та охоронцями), збирати предмети, працювати, створювати інструменти та планувати втечу. Всі персонажі та об'єкти діють за власними правилами, формуючи симулятивну ігрову систему. Кінцева мета гравця — втекти з в'язниці, уникаючи підозри та переслідування охоронців.

Мета.

Метою нашої розробки є створення інформаційної технології для моделювання предметної галузі ігрових симуляцій на основі розроблення архітектури та реалізації автоматизованої багатокомпонентної системи з елементами мультиплесру, збереження станів, штучного інтелекту, та управління ігровим процесом у реальному часі.

Вимоги до функціоналу.

Головна сторінка:

Гравець при запуску гри потрапляє до головного меню, яке повинне надавати такі функції:

- Запуск гри (нової або збереженої) у якості хоста;
- Перегляд глобального рейтингу рекордів гравців;
- Налаштування гучності;
- Налаштування якості картинки.

Система збереження:

- Автоматичне збереження ігрових даних кожні декілька секунд та при виході гравця з сесії.
- Автоматичне завантаження збереження при старті гри, якщо гравець є хостом.
- Можливість передати збереження іншому гравцю для продовження гри в іншій сесії.

NPC:

- Заручники (окрім гравців), охоронці та медики — керуються AI (в основному, на основі State Trees).
- Охоронці патрулюють, перевіряють підозрілих, атакують гравців при високій підозрілості.
- Медики знаходять непритомних та несуть їх до лікарні.

Система часу та розкладу:

- Ігровий час ділиться на дні, години, хвилини. Одна хвилина = одна секунда реального часу.
- Час впливає на положення сонця/місяця та зміну подій.
- Порушення розпорядку (пропуск подій, перебування поза камерою) викликає зростання підозрілості або загальну тривогу.

Ігрові події:

- "Тиха година" — протягом однієї ігрової години гравець має повернутися до своєї камери. Після цього двері до камер зачиняються на вихід, а багато інших дверей —

повністю.

- "Перекличка" — заручники повинні з'явитися в загальному залі, інакше почнеться тривога.
- "Прийом їжі" — заручники сідають за столи в їдальні.
- "Душ" — заручники користуються душем.
- "Перерва" — вільний час гравця для дій.
- "Тривога" — ставить інші події на паузу (але боти заручники продовжують їм слідувати), викликається, якщо гравця немає у камері вночі, пропущено перекличку або у в'язниці виявлені пошкодження. Після тривоги всі пошкодження мапи усуваються. Тривога підіймається для конкретного списку персонажів в залежності від причини початку.

Пошкодження об'єктів:

- Деякі об'єкти мапи можна ламати інструментами.
- У зонах, де NPC не з'являються, можна безпечно робити дірки.

Характеристики персонажа:

- Здоров'я — при 0 персонаж падає, його відносять до ліжка медики.
- Енергія — витрачається при бігу, ударах, блоках, пошкодженні об'єктів.
- Підозрілість — при >90 NPC охоронці атакують заручника.
- Урон — залежить від обраної зброї в руках та її наявності.

Рух персонажа:

Гравець може ходити, бігти підцюзцем, бігти швидко (витрачає енергію), присідати, стрибати

Інвентар та предмети:

- Обмежена кількість слотів + слот для одягу.
- Гравець має доступ до системи крафту: предмети можна поєднувати відповідно до фіксованих рецептів для створення нових об'єктів.
- Деякі рецепти дозволяють створювати копії ключів або одяг маскуванню без втрати оригінального предмета.
- Крафт відбувається через інтерфейс, доступний у спеціальному меню гравця, де відображаються доступні рецепти на основі наявних у інвентарі предметів.
- Є предмети звичайні, контрабандні, витратні, крафтові, інструменти, зброя.
- Контрабанда реагує на металодетектори. Цього можна уникнути якщо в одному зі слотів є контрабандний мішок.
- Втрата свідомості веде до втрати контрабанди (для гравців).
- Деякі предмети мають міцність та ламаються.
- Існують кукли для ліжка, що обманюють охоронців під час перевірок уночі.

Можна відкривати інвентар непритомних персонажів.

Взаємодія з об'єктами:

- Ліжка, лікарняні ліжка, душі, сидіння в їдальні — дають різні ефекти відновлення.
- Тумба — зберігає предмети.

Генерація предметів:

- Щоденне оновлення предметів у тумбах (окрім тих, якими володіють гравці).
- Щоденне оновлення інвентарів ботів.

Одяг:

- Заручники, охоронці, медики мають різний одяг.
- Одяг можна скрафтити або вкрасти.
- У нічний час замаскований гравець не викликає тривоги, коли його бачать поза його камери.

Бій:

- Можна атакувати та ставити блок.

- Блок захищає від урону, але витрачає енергію.

Двері та ключі:

- Двері бувають відкриті повністю, або з одностороннім доступом.
- Якщо двері закриті, то для проходу потрібен ключ відповідного кольору.
- Ключі можна знайти, вкрасти або скрафтити копію. Оригінал не витрачається при використанні та не пропадає при крафті копії.
- При втраті оригінального ключа після того, як охоронець приходить до свідомості, цей охоронець піднімає тривогу.

Втеча з в'язниці:

- Для втечі потрібно вийти за межі тюрми через зламану ділянку мапи, не будучи поміченим охоронцями.

Імена персонажів:

- Усі персонажі мають імена. У гравців — зі Steam.
- Імена відображаються над головами та в інтерфейсі інвентаря.

Кінець гри та рекорди:

- Гра завершується при втечі одного з гравців.
- Результат потрапляє до глобального рейтингу рекордів часу втечі.

Мережевий режим:

- До 4 гравців одночасно через Steam P2P.
- Один з гравців є хостом. Інші підключаються через запрошення у Steam.

Технології

Зберігання вихідних кодів ПЗ здійснюється за допомогою системи контролю версій Git.

Структура файлів та каталогів повинна відображати архітектуру ПЗ із розбиттям на асети з блюпрінтами та код з .h і .cpp файлами. Приклад:

- Content
 - AbilitySystem
 - Abilities
 - GA_Example
 - Cues
 - GCNB_Example
 - Effects
 - GE_Example
 - AI
 - AIControllers
 - BP_ExampleAIController
 - SmartObjects
 - SOD_Example
 - StateTrees
 - ST_Example
 - Characters
 - Common
 - Animations
 - ABP_Example
 - Materials
 - M_Example
 - Meshes

- SKM_Example
 - Textures
 - T_Example
 - BP_ExampleCharacter
- PlayerControllers
 - BP_ExamplePlayerController
- UI
 - Widgets
 - WBP_Example
- Source
 - ProjectName
 - Private
 - AbilitySystem
 - Abilities
 - ExampleGameplayAbility.cpp
 - Characters
 - ExampleCharacter.cpp
 - Common
 - Structs
 - Example.cpp
 - Enums
 - Example.cpp
 - Controllers
 - PlayerControllers
 - ExamplePlayerController.cpp
 - AIControllers
 - ExampleAIController.cpp
 - StateTree
 - Conditions
 - ExampleStateTreeCondition.cpp
 - Evaluators
 - ExampleStateTreeEvaluator.cpp
 - Tasks
 - ExampleStateTreeTask.cpp
 - UI
 - Widgets
 - ExampleWidget.cpp
 - Public
 - AbilitySystem
 - Abilities
 - ExampleGameplayAbility.h
 - Characters
 - ExampleCharacter.h
 - Common
 - Structs
 - Example.h

- Enums
 - Example.h
- Controllers
 - PlayerControllers
 - ExamplePlayerController.h
 - AIControllers
 - ExampleAIController.h
- StateTree
 - Conditions
 - ExampleStateTreeCondition.h
 - Evaluators
 - ExampleStateTreeEvaluator.h
 - Tasks
 - ExampleStateTreeTask.h
- UI
 - Widgets
 - ExampleWidget.h

Для розробки необхідні наступні технології створення:

- Ігровий процес: Ігровий рушій Unreal Engine 5, мова програмування C++, Gameplay Framework, Replication, Remote Procedure Calls (RPC), Mover, Gameplay Ability System, Subsystems, State Trees, Smart Objects, Common UI, інше.
- Бекенд рейтингу гравців: Node.js, MySQL, Express, Nodemon, MySQL2, TypeORM, Dotenv, Nginx, Swagger.
- Дизайн: Ігровий рушій Unreal Engine 5, Sythe, Autodesk Maya, Adobe Substance 3D Painter, ZBrush, Marmoset Toolbag, Figma.

Команда проєкту

Для виконання поставлених завдань робота команди передбачається окремо за кожною частиною ігрової функціональності.

При команді в 3 учасника, двоє учасників відповідають за створення програмних систем, 1 відповідає за візуальну частину, усі відповідають за наповнення ігрового контенту.

Мінімальна команда проєкту - два учасники, у такому складі один учасник виконує модуль Backend та програмну частину Web, другий - модель Moskip та розміткову частину Web.

Рекомендована команда проєкту - три учасники, у такій команді кожен з учасників відповідає за одну з частин: Moskip, Backend, Web.

Дозволяється включення до команди більшої кількості учасників, у такому разі модуль Mobile App набуває обов'язкової реалізації. Також модуль Moskip може бути поділений на моделі Web та Mobile частин.

РОЗДІЛ 1
ЗАГАЛЬНА ХАРАКТЕРИСТИКА СИСТЕМИ

Дипломний проєкт являє собою повноцінну 3D-гру, реалізовану на базі рушія Unreal Engine 5 із використанням C++ та внутрішніх систем рушія, таких як Gameplay Framework, Gameplay Ability System (GAS), State Trees, Smart Objects, Subsystems, Common UI, та інші. Архітектура застосунку побудована відповідно до сучасних принципів розробки програмного забезпечення, що забезпечує масштабованість, повторне використання компонентів, зручність підтримки та розширення функціоналу.

Усі системи гри реалізовані з максимальною модульністю — через активне використання композиції, де це можливо, з чітким розділенням відповідальностей між об'єктами. Там, де це було доцільно, застосовувалося успадкування для розширення базової логіки, однак базова архітектура завжди зберігала гнучкість і незалежність систем за рахунок делегування та інтерфейсів.

Код відповідає офіційним стандартам Unreal Engine, зокрема:

- дотримано конвенцій іменування (U, A, F, E, I);
- розмежовано публічний та приватний API;
- усі властивості оголошено з використанням UPROPERTY з відповідними специфікаторами (VisibleAnywhere, EditDefaultsOnly, Transient, SaveGame тощо);
- застосовано GetNetMode() та HasAuthority() для відокремлення серверної та клієнтської логіки.

Значна частина логіки гри розрахована на багатокористувацьку взаємодію. Для цього реалізовано підтримку реплікації змін (Replicated, ReplicatedUsing) та віддалених викликів процедур (Remote Procedure Calls, Client, Server, Multicast). Реплікація використовувалася як на рівні Actor'ів та компонентів, так і на рівні атрибутів та інших об'єктів. Окремі системи передбачають клієнтські передбачення — наприклад, у Ability System — для зменшення затримок при виконанні дій.

Усі вказівники на об'єкти в .h файлах реалізовано за допомогою TObjectPtr та TSubclassOf — як рекомендовано у рушії п'ятої версії. Для збереження зв'язків із іншими класами без жорсткого завантаження об'єктів у пам'ять активно використовувалися м'які посилання — TSoftObjectPtr та TSoftClassPtr. Це дозволяє уникнути зайвого завантаження об'єктів у пам'ять до того моменту, поки в них дійсно не виникне потреба.

Усі soft-посилання завантажуються асинхронно за допомогою StreamableManager, і після завершення використання — вивантажуються з пам'яті рушієм. Це дає змогу тримати в оперативній пам'яті лише актуальні об'єкти. Завантаження таких ресурсів відбувається асинхронно з використанням StreamableManager, що, у свою чергу, дозволяє уникнути фризів та блокування головного потоку гри. Після завершення використання такі об'єкти вивантажуються з пам'яті, що додатково оптимізує використання ресурсів.

```
LoadSomeObjectHandle = UAssetManager::GetStreamableManager().RequestAsyncLoad(
    SomeSoftObject.ToSoftObjectPath(),
    Callback: FStreamableDelegate::CreateUObject(this, &ThisClass::OnSomeObjectLoaded));
```

У процесі розробки активно використовувалися макроси перевірки check, checkf, ensure, ensureAlways — для виявлення критичних помилок під час виконання у режимі розробника. Це дозволяло фіксувати проблеми безпосередньо під час тестування гри в редакторі, замість застосування окремих unit-тестів, які не є типовою практикою в Unreal Engine.

```
if (!ensureAlways(IsValid(Character)))
{
    CancelAbility(Handle, ActorInfo, ActivationInfo, bReplicateCancelAbility: false);

    return;
}
```

```
#if DO_CHECK
    check(IsValid(NewPlayer));
    check(IsValid( Test: ↻ NewPlayer->GetPawn())));
#endif #if DO_CHECK
```

Документація до коду реалізована у форматі JavaDoc, із докладним описом класів, методів, параметрів і прикладів використання. Усі основні модулі, включно з системою здібностей, збережень, подій, інвентарю тощо, мають докладні пояснення для сторонніх розробників.

Таким чином, архітектура проєкту вирізняється високою якістю реалізації, дотриманням стандартів рушія, оптимізацією пам'яті та ресурсоємних операцій, а також чіткою підтримкою багатокористувацької взаємодії. Це дозволяє системі бути не лише функціональною, але й стабільною, масштабованою та зручною для підтримки в майбутньому.

РОЗДІЛ 2
GAMEPLAY ABILITY SYSTEM ТА ІНПУТ

У представленому проєкті реалізована модульна система здібностей на основі Gameplay Ability System (GAS), яка була запозичена з офіційного проєкту від Epic Games — *Lyra Starter Game*, однак суттєво модифікована та вдосконалена з урахуванням кращих архітектурних практик. Основною метою стало створення масштабованої системи, що дозволяє централізовано призначати здібності, ефекти та атрибути будь-якому персонажу залежно від його ролі, та пов'язувати ці здібності з діями користувача за допомогою розширеної системи вводу.

AbilitySystemSet

Для централізації логіки надання здібностей було реалізовано клас `UAbilitySystemSet`, який є нащадком `UPrimaryDataAsset`. Цей клас визначає такі структури:

- `FAbilitySystemSet_GameplayAbility` — містить тип здібності, рівень та `InputTag`.
- `FAbilitySystemSet_GameplayEffect` — визначає ефекти з рівнями.
- `FAbilitySystemSet_AttributeSet` — набір атрибутів (наприклад, здоров'я, енергія).
- `FAbilitySystemSet_BlockedAttributes` — дозволяє заблокувати зміну певних атрибутів залежно від `GameplayTag`.

При виклику `GiveToAbilitySystem(...)` компонента `UAbilitySystemSet` автоматично:

- додає `AttributeSets`,
- надає `GameplayAbilities`,
- застосовує `GameplayEffects`,
- виконує прив'язку `InputTag` до `InputID`.

Коли здібності додаються через GAS, кожна з них отримує унікальний `InputID`, згенерований методом `GenerateInputId()`. Потім, через `RegisterInputTag(...)` цей ID зберігається у `TMap InputIDsAssociatedWithInputTags`, що дозволяє надалі активувати здібності за `GameplayTag`, а не за raw-інпутами.

Система вводу

Для конфігурації вводу створено окремий клас `UInputConfig` — похідний від `UDataAsset`, який визначає три основні групи:

- `InputMappingContexts` — набір контекстів вводу (Enhanced Input),
- `NativeInputActions` — інпут-дії, пов'язані з діями, які не є здібностями (наприклад, рух чи керування камерою),
- `AbilityInputActions` — інпут-дії, що пов'язані з `InputTag` здібностей.

Прив'язка інпутів виконується в `AEscapeChroniclesPlayerController` через метод `BindInputConfig(...)`. Для `AbilityInputActions` передбачається автоматична реєстрація `InputTag` у `Ability System Component` з наступним відслідковуванням подій `ETriggerEvent::Started` та `Completed`, які відповідно викликають `PressInputTag(...)` та `ReleaseInputTag(...)`.

Таким чином, кожна здібність пов'язується з конкретною кнопкою за допомогою зв'язки: `InputAction` → `InputTag` → `InputID` → `Ability`.

Цей підхід є архітектурно правильним способом використання GAS, на відміну від *Lyra*, де `InputAction` щотік перевіряється вручну, що не є оптимальним для масштабованих систем.

Висновок

Дана реалізація забезпечує:

- централізовану конфігурацію здібностей через `UAbilitySystemSet`,
- декларативне управління інпутами за допомогою `UInputConfig`,
- чисту архітектуру з чітким розділенням відповідальностей,
- дотримання стандартів GAS API (з використанням `InputID` замість кастомної логіки),
- простоту розширення та налаштування нових персонажів, ролей та здібностей без дублікації логіки.

РОЗДІЛ 3

ХАРАКТЕРИСТИКИ ПЕРСОНАЖУ, РОЛІ ТА ТЕГИ

Для реалізації системи характеристик персонажів у проєкті було використано Gameplay Ability System та створено три окремі AttributeSet класи:

- CombatAttributeSet — для зберігання бойових параметрів (наприклад, поточний урон);
- VitalAttributeSet — для параметрів здоров'я та енергії (зокрема, Health, MaxHealth, Energy, MaxEnergy);
- SharedRelationshipAttributeSet — для підозрілості (Suspicion та MaxSuspicion), що відображає відношення NPC до гравця.

Усі ці класи є нащадками від UEscapeChroniclesAttributeSet, який, у свою чергу, успадковується від UAttributeSet. В ньому було реалізовано базову логіку для обмеження значень атрибутів (ClampAttribute), а також хелпери для доступу до значень через макроси ATTRIBUTE_ACCESSORS, ATTRIBUTE_GETTERS, ATTRIBUTE_SETTERS. Наприклад, у класі USharedRelationshipAttributeSet встановлено логіку, яка не дозволяє перевищити підозрілість понад її максимум:

```
void USharedRelationshipAttributeSet::ClampAttribute(const FGameplayAttribute& Attribute, float& NewValue) const
{
    Super::ClampAttribute(Attribute, [&] NewValue);

    if (Attribute == GetSuspicionAttribute())
    {
        // Do not allow the Suspicion to drop below 0 and above MaxSuspicion
        NewValue = FMath::Clamp(NewValue, Min:0.f, GetMaxSuspicion());
    }
    else if (Attribute == GetMaxSuspicionAttribute())
    {
        // Do not allow MaxSuspicion to drop below 1
        NewValue = FMath::Max(NewValue, 1.f);
    }
}
```

Атрибути позначено як ReplicatedUsing, тож при оновленні значення викликається відповідна OnRep_ функція для клієнтів, наприклад:

```
void USharedRelationshipAttributeSet::OnRep_Suspicion(const FGameplayAttributeData& OldValue) const
{
    GAMEPLAYATTRIBUTE_REPNOTIFY(ThisClass, Suspicion, OldValue);
}
```

Щоб додати гнучкість при ініціалізації персонажів, було реалізовано можливість підставляти різні PlayerState залежно від контролера. Клас AEscapeChroniclesPlayerController (як і AIController) має поле TSubclassOf PlayerStateClassOverride, яке дозволяє задати потрібний клас PlayerState прямо з редактора. Метод InitPlayerState перевіряє цю змінну та створює відповідний об'єкт.

Цей підхід дозволяє нам створювати незалежні набори характеристик і здібностей для:

- гравця-ув'язненого (BP_PlayerPrisonerPlayerState);
- бота-охоронця (BP_BotGuardPlayerState);
- медика (BP_BotMedicPlayerState та ін.

Кожен з таких PlayerState отримує свій набір AbilitySystemSet (наприклад, DA_GuardAbilitySystemSet, DA_PrisonerAbilitySystemSet тощо), що містить:

- список здібностей (GameplayAbilitiesToGrant),
- ефекти (GameplayEffectsToGrant),
- набори атрибутів (AttributeSetsToGrant),
- обмеження на атрибути (AttributesToBlockByTags).

Додатково в грі реалізовано систему тегів ролей через GameplayTags:

- Role.Prisoner

- Role.Guard
- Role.Medic
- Role.NPC
- Role.Player

Ці теги додаються через `GameplayEffect` безпосередньо в `AbilitySystemSet` кожного персонажа, що дозволяє легко фільтрувати поведінку за ролями у `State Tree`, при цільових ефектах, поведінці AI тощо.

Завдяки такій модульній структурі проєкт легко масштабується — можна без змін у коді додати нові типи персонажів, наприклад, бота-кухаря або гравця-охоронця, просто створивши нові `PlayerState`'и, `AbilitySystemSet`'и, та відповідні `Controller`'и.

РОЗДІЛ 4
GAMEPLAY ABILITY COMPONENTS

У проєкті було реалізовано власну систему GameplayAbilityComponents — модульних компонентів, які розширюють логіку здібностей Gameplay Ability System без необхідності створювати нові похідні класи здібностей. Це дало змогу зменшити кількість дубльованого коду, а також спростити розширення логіки здібностей при збереженні базової реалізації в UEscapeChroniclesGameplayAbility.

Кожен UGameplayAbilityComponent:

- є екземпляром UObject, який зберігається у властивості Components здібності;
- має маркування UCLASS(Abstract, Const, DefaultToInstanced, EditInlineNew, CollapseCategories, Within=EscapeChroniclesGameplayAbility), що дозволяє створювати його екземпляри безпосередньо всередині здібності;
- повністю дублює весь public та protected інтерфейс UGameplayAbility, включаючи методи CheckCost, CommitCheck, ApplyCooldown, InputPressed, EndAbility тощо.

Кожен з цих методів у UEscapeChroniclesGameplayAbility викликає відповідні методи усіх компонентів у циклі. Наприклад, метод CheckCost:

```
bool UEscapeChroniclesGameplayAbility::CheckCost(const FGameplayAbilitySpecHandle Handle,
    const FGameplayAbilityActorInfo* ActorInfo, FGameplayTagContainer* OptionalRelevantTags) const
{
    if (!Super::CheckCost(Handle, ActorInfo, OptionalRelevantTags))
    {
        return false;
    }

    // Check if all components satisfy the cost check
    for (const UGameplayAbilityComponent* Component : Components)
    {
        if (!Component->CheckCost(Handle, ActorInfo, OptionalRelevantTags))
        {
            return false;
        }
    }

    return true;
}
```

Це дозволяє декомпонувати логіку здібностей на маленькі ізольовані частини. Наприклад, одна зі здібностей може містити компонент URequireEnergyGameplayAbilityComponent, що перевіряє наявність енергії в гравця у методі CheckCost, а також автоматично відписується та скасовує здібність, якщо енергія падає до нуля. Він також підписується на делегат зміни атрибута енергії через GetGameplayAttributeValueChangeDelegate і при спрацюванні викликає CancelAbility у здібності.

Такий підхід дозволив:

- реалізувати здібності у вигляді базових Blueprint-класів з можливістю редагування складу компонентів напряму в редакторі;
- забезпечити модульність та повторне використання логіки у різних здібностях;
- значно зменшити дублювання коду;
- спростити тестування та підтримку здібностей.

Кожен компонент не має прямого доступу до AbilitySystemComponent, ActorInfo чи інших внутрішніх об'єктів — вони передаються явно у всі відповідні методи. Це дозволяє повністю контролювати життєвий цикл компонента та гарантує його коректну інтеграцію в будь-який контекст виконання здібності.

Важливо зазначити, що всі внутрішні методи GameplayAbility було відкрито через friend class, зокрема UGameplayAbilityComponent, щоб мати доступ до потрібних делегатів,

властивостей та можливість завершувати здібність безпосередньо з компоненту.

Таким чином, реалізація `GameplayAbilityComponents` забезпечила масштабовану та гнучку архітектуру здібностей у проєкті без втрати контролю над внутрішньою логікою GAS.

РОЗДІЛ 5
ПЕРЕСУВАННЯ ПЕРСОНАЖУ

Система пересування у грі реалізована на основі експериментального плагіна Mover, який було інтегровано у рушії Unreal Engine 5. За основу було взято приклад персонажа з демонстраційного плагіну MoverExamples, що надалі був розширений та вдосконалений для підтримки більш гнучких режимів пересування.

Було реалізовано кастомний інтерфейс інпуту у вигляді структури FEscapeChroniclesCharacterExtendedDefaultInputs, що включає прапор bWantsToBeCrouched (чи хоче персонаж присісти) та змінну DesiredGroundSpeedModeOverride типу EGroundSpeedMode, яка вказує бажаний режим пересування: Walking, Jogging, або Running.

Щоб зв'язати обраний режим пересування зі швидкістю, було реалізовано клас UGroundSpeedModeSettings, який належить до типу IMovementSettingsInterface та зберігає максимальні швидкості для кожного режиму. Параметри задаються через змінні WalkingMaxSpeed, JoggingMaxSpeed, RunningMaxSpeed, та обираються за допомогою функції GetSelectedMaxSpeed() відповідно до поточного GroundSpeedMode.

Для модифікації швидкості пересування на основі обраного режиму було реалізовано окремий модифікатор FGroundSpeedModeModifier, що додається у методі OnMoverPreSimulationTick компонента UEscapeChroniclesCharacterMoverComponent.

Цей компонент є спадкоємцем UCharacterMoverComponent та відповідає за всю логіку керування швидкістю і реакцію на зміни режиму. У випадку зміни GroundSpeedMode, активується делегат OnGroundSpeedModeChanged, що дозволяє відслідковувати переходи між режимами.

Положення персонажа синхронізується з системою здібностей (Ability System) через GameplayTags, які додаються або видаляються у методах SyncMovementModesTagsWithAbilitySystem, SyncStancesTagsWithAbilitySystem, SyncGroundSpeedModeTagsWithAbilitySystem класу AEscapeChroniclesCharacter. Наприклад, коли персонаж біжить, до нього додається тег Status.Movement.Mode.Running, а якщо він присідає — Status.Movement.Crouching. Це дозволяє здібностям реагувати на зміну стану руху персонажа.

Присідання реалізовано за допомогою вже існуючих можливостей UCharacterMoverComponent, де достатньо викликати метод присідання в ProduceInput, зчитавши прапор з кастомного інпуту. Самі методи присідання (Crouch, UnCrouch) викликають зміну внутрішнього стану bWantsToBeCrouched.

Інтерфейс IMoverInputProducerInterface реалізовано безпосередньо в класі персонажа AEscapeChroniclesCharacter, де формується об'єкт інпуту на кожному тiku в методі ProduceInput. Тут також реалізована логіка обертання персонажа в напрямку камери гравця або напрямку руху, в залежності від конфігурації.

Таким чином, система пересування побудована як повністю модульна та масштабована, із чіткою інтеграцією з інпутом, режимами анімації та системою здібностей.

РОЗДІЛ 6
СИСТЕМА ЗБЕРЕЖЕНЬ ТА ІДЕНТИФІКАЦІЯ ГРАВЦІВ

У рамках гри була реалізована універсальна система збережень, яка дозволяє зберігати стан усіх важливих об'єктів гри, включаючи:

- підсистеми світу (*WorldSubsystems*),
- статичні актори на сцені,
- дозволені динамічні актори (наприклад, актори, що з'являються на рівні вручну або спавняються у межах геймплею),
- ігрові класи ядра: *GameMode*, *GameState*, *PlayerState*, *Controller*, *Pawn*,
- компоненти зберезуваних акторів.

Для того, щоб об'єкт міг бути збереженим, достатньо реалізувати інтерфейс **ISaveable**, який має методи:

- OnPreSaveObject / OnPostSaveObject
- OnPreLoadObject / OnPostLoadObject

Ці методи викликаються відповідно до фази серіалізації/десеріалізації. Для акторів також достатньо просто додати FName тег (Saveable), якщо вони не потребують складної логіки.

Компоненти акторів можуть зберігатися лише якщо його власник-актор теж може зберігатися.

Архітектура побудована на основі кастомного класу підсистеми світу — USaveGameSubsystem, який відповідає як за централізовану логіку збереження та завантаження гри, так і за ініціалізацію системи ідентифікації гравців.

Збереження здійснюється до кастомного об'єкта класу UEscapeChroniclesSaveGame, що має окремі поля із різними TMap для різних типів об'єктів.

Збереження та завантаження виконується через стандартні засоби серіалізації Unreal Engine. Зокрема, використовується FObjectAndNameAsStringProxyArchive з виставленим прапором IsSaveGame. Це дозволяє автоматично серіалізувати тільки поля, помічені як UPROPERTY(SaveGame).

Приклад функції, яка відповідає за серіалізацію даних у байтовий масив:

```
void USaveGameSubsystem::SaveObjectSaveGameFields(UObject* Object, TArray<uint8>& OutByteData)
{
    // Pass the array to be able to fill with data from an object
    FMemoryWriter MemoryWriter([&] OutByteData);

    // Create an archive to serialize the data from an object
    FObjectAndNameAsStringProxyArchive Ar([&] MemoryWriter, bInLoadIfFindFails: true);

    // Serialize only properties marked with "SaveGame"
    Ar.ArIsSaveGame = true;

    // Finally, serialize the object's properties
    Object->Serialize([&] Ar);
}
```

Аналогічна функція використовується для десеріалізації даних:

```
void USaveGameSubsystem::LoadObjectSaveGameFields(UObject* Object, const TArray<uint8>& InByteData)
{
    // Pass the array to read the data for an object from
    FMemoryReader MemoryReader(InByteData);

    // Serialize the passed data to an archive
    FObjectAndNameAsStringProxyArchive Ar( [&] MemoryReader, bInLoadIfFindFails: true);

    // Serialize only properties marked with "SaveGame"
    Ar.ArIsSaveGame = true;

    // Finally, serialize the object's properties from the archive
    Object->Serialize( [&] Ar);
}
```

Система підтримує повну ієрархію об'єктів гри, включно з WorldSubsystem'ами, динамічно створеними акторами, контролерами, PlayerState, Pawn тощо. Динамічні актори допускаються до збереження лише в тому випадку, якщо входять до списку дозволених через метод IsAllowedDynamicallySpawnedActor. Це забезпечує контроль над тим, які динамічні об'єкти можуть бути збережені, наприклад, через категорії геймплейних об'єктів.

Кожен SaveGameObject прив'язується до конкретного рівня: як при збереженні, так і при завантаженні до імені слоту додається назва рівня. Це дозволяє уникнути помилкового завантаження збережень іншого рівня. Окрім цього, назва рівня також зберігається всередині UEscapeChroniclesSaveGame для додаткової перевірки.

Окрему увагу приділено системі ідентифікації гравців. Для цього було реалізовано власну структуру — FUniquePlayerID, яка складається з:

- внутрішнього ID (PlayerID);
- мережевого ID (NetID) — наприклад, Steam ID;
- локального ID (LocalPlayerID) — використовується у випадках Split Screen або гри в редакторі.

Цей ID зберігається в PlayerState гравця та використовується для точного визначення того, яким гравцем було створено те чи інше збереження. Завдяки цьому можна окремо зберігати й відновлювати стан як онлайн-гравців, так і офлайн-гравців та ботів.

При збереженні PlayerState або бота, система аналізує, чи має він дійсний NetID:

- якщо має — збереження йде до мапи OnlinePlayerSaveData;
- якщо це бот — залежно від ролі (Prisoner, Guard, Medic) збереження відбувається до відповідної мапи;
- інакше — збереження йде до OfflinePlayerSaveData.

При завантаженні гра намагається знайти збереження за NetID, або, якщо гравець локальний, — за LocalPlayerID. У разі невдачі генерується новий FUniquePlayerID. Для ботів завантаження здійснюється за роллю, і кожному боту присвоюється перший доступний ID із відповідної мапи.

Автоматичне збереження гри відбувається кожні N секунд після OnWorldBeginPlay, завдяки встановленню таймера. Також гра зберігається синхронно при закритті (EndPlay у GameMode) та при виході гравця (OnUnPossess у контролері), де викликається SavePlayerOrBotChecked з оновленням даних гравця в поточному SaveGameObject.

Завантаження нового гравця реалізовано в AEscapeChroniclesGameMode::InitNewPlayer. Після завершення ініціалізації PlayerState та Pawn, гра викликає LoadPlayerOrGenerateUniquePlayerIdChecked, після чого викликає InitNewPlayer (перезавантажений метод), що сигналізує, чи гравець був завантажений зі збереження, чи

приєднався вперше.

Наприклад, у `AbilitySystemComponent` зберігаються всі атрибути з усіх `AttributeSets` у `TMap` «назва → значення» перед збереженням об'єкта, і відновлюються після завантаження об'єкта.

Завдяки такому підходу, система збережень забезпечує:

- Надійне відновлення повного стану гри;
- Підтримку як одиночної гри, так і мультиплеєру;
- Відокремлення даних між рівнями;
- Підтримку ботів та різних ролей персонажів;
- Можливість гнучкого розширення (через інтерфейс `ISaveable`).

РОЗДІЛ 7
ВЛАСНИКИ АКТОРІВ

У рамках гри проєкту була реалізована система прив'язки акторів до конкретних гравців або ботів. Основною метою цієї системи є визначення того, хто володіє якими об'єктами на сцені: ліжками, камерами, тумбами, шафами, камерами відеоспостереження тощо. Завдяки цій системі забезпечується контроль доступу до об'єктів на основі приналежності, а також персоналізована взаємодія кожного гравця з ігровим світом.

Система базується на компоненті `UPlayerOwnershipComponent`, який додається до кожного актора, що повинен мати власника. Власником актора виступає гравець або бот, визначений через структуру `FUniquePlayerID` — унікальний ідентифікатор, який містить внутрішній ID гравця, `NetID` платформи (якщо такий є) і `LocalPlayerID`, який використовується для локальної ідентифікації, зокрема в режимах `PIE` або `Split Screen`. Компонент зберігає дані про власника в реплікованій структурі `FPlayerOwnershipComponentOwningPlayerContainer`, що включає сам ID та тип гравця (реальний гравець або бот).

Власність у системі визначається не на рівні окремого актора, а через групу компонентів. Кожен `UPlayerOwnershipComponent` має прив'язку до певної групи, яка описується у спеціальному `DataTable`, що містить рядки типу `FPlayerOwnershipComponentGroupTableRow`. Ці рядки визначають назву групи, дозволені типи гравців, яким можна належати до цієї групи (через перелік значень `EControlledCharacterType`), а також перелік тегів (`GameplayTags`), якими повинен володіти гравець, щоб мати право стати її власником. Таким чином, система дозволяє не лише задати приналежність об'єктів, а й накладати додаткові обмеження на доступ до них.

Прив'язка гравця до компонентів відбувається автоматично після повної ініціалізації гравця — після того як завершиться завантаження його стану з сейву, буде створено `Pawn`, `Controller` та `PlayerState`. Для цього використовується статичний метод `UPlayerOwnershipComponent::RegisterPlayer`, який викликається окремо для кожного нового гравця або бота із `GameMode`, коли цей гравець або бот повністю ініціалізований та завантажений. Метод здійснює повний обхід усіх акторів на рівні й знаходить компоненти з валідними групами. Далі система перевіряє, чи вже є власник у групі. Якщо власник відсутній і переданий гравець задовольняє вимоги (тип і теги), він призначається власником для всіх компонентів цієї групи. Якщо власник вже є, але не встановлений у деяких компонентах, ці компоненти оновлюються. Один гравець може володіти лише однією групою — ця логіка контролюється кодом, що не дозволяє мати кілька груп з однаковим `OwningPlayer`.

Для встановлення власника компонент викликає метод `InitializeOwningPlayer`, який оновлює відповідні поля та викликає мультиделегат `OnOwningPlayerInitialized`. Таким чином, інші частини гри можуть підписатися на подію ініціалізації власника й реагувати на неї: наприклад, `UI` може активувати елементи керування, а механіки — змінити поведінку об'єкта залежно від гравця.

Щоб отримати актуальні дані про власника, можна викликати методи `GetOwningPlayer()` та `GetControlledCharacterType()`. Якщо ж потрібно дочекатись моменту, коли власник буде встановлений, передбачений метод `CallOrRegister_OnOwningPlayerInitialized`, що дозволяє або одразу викликати делегат, або зареєструвати його для подальшого виклику після ініціалізації.

Система також інтегрована із системою збережень. Компонент `UPlayerOwnershipComponent` реалізує інтерфейс `ISaveable`, і зберігає дані про власника в сейві. Під час завантаження (`OnPostLoadObject`) компонент автоматично викликає `OnOwningPlayerInitialized`, якщо було збережено валідний `FUniquePlayerID`. Таким чином, прив'язка зберігається між сесіями, і гравець завжди має доступ до своїх об'єктів.

Крім того, компоненти можуть бути створені динамічно під час гри. У таких випадках виклик `InitializeOwningPlayer` дозволяє вручну встановити гравця як власника для таких об'єктів. Це забезпечує підтримку геймплейних механік, де нові об'єкти створюються під час сесії.

У випадку, коли у компонента не задано групу (`GroupData.RowName == None`), він не

братиме участі в системі власників і не отримає жодного гравця як власника. Це дозволяє створювати нейтральні або загальні об'єкти, які не прив'язані до конкретного гравця.

Загалом, система власників акторів у забезпечує централізовану, контрольовану та масштабовану інфраструктуру для прив'язки об'єктів до гравців або ботів. Вона дозволяє розмежовувати доступ, зберігати власність між сесіями, забезпечує персоналізацію взаємодії й є важливою частиною архітектури гри.

РОЗДІЛ 8
ІНВЕНТАР ТА ПРЕДМЕТИ

У рамках проєкту було реалізовано декілька геймплейних систем, які розширюють функціональність інвентаря та предметів. Зокрема, додано предмети, що змінюють персонажа або взаємодіють зі світом, підтримку використання предметів, збереження інвентаря та предметів на сцені, а також спеціальні механіки, пов'язані з класифікацією предметів.

UEscapeChroniclesInventoryManagerComponent, що є нащадком базового компонента інвентаря, зберігає та відновлює всі предмети гравця при збереженні гри, включно з їхніми характеристиками. Також була реалізована підтримка використання предметів, екіпірування одягу, класифікація предметів (наприклад, контрабанда), логіка видалення предметів у разі втрати свідомості, та взаємодія з об'єктами у світі, зокрема металодетекторами та акторами ActivitySpots. Окремо була додана система збережень предметів, що розкидані по карті.

Для збереження інвентаря гравця реалізовано серіалізацію всіх слотів через структуру FInventorySlotsSaveData. Під час збереження

UEscapeChroniclesInventoryManagerComponent проходиться по кожному типізованому масиву слотів (наприклад, основні, екіпірування, швидкий доступ) та зберігає для кожного предмета його клас (ItemDefinition) і статистику (InstanceStats). Після завантаження гри інвентар очищується, і всі предмети реконструюються за збереженою інформацією. Це дозволяє зберігати інвентар індивідуально для кожного гравця чи бота.

Підтримку використання предметів реалізовано через систему фрагментів. Базовим є клас UUsableItemInventoryItemFragment, який надає інтерфейс TryUseItem. Конкретні реалізації перевизначають метод OnTriedToUseItem. Наприклад, аптечка реалізована як UConsumableInventoryItemFragment, яка при використанні застосовує GameplayEffect до AbilitySystemComponent гравця. Ефект завантажується асинхронно через UAssetManager та після застосування вивільняється з пам'яті. Для запуску використання з інтерфейсу інвентаря було реалізовано фрагмент UInventoryManagerUseItemsFragment, який звертається до SelectorFragment та ініціює TryUseItem для поточного слота.

Додано систему класифікації предметів через фрагмент UClassificationInventoryItemFragment, що зберігає тип предмета у вигляді переліку (EItemClassification). Це дозволило реалізувати поведінку, коли певні предмети (наприклад, контрабанда) автоматично зникають з інвентаря при втраті свідомості. Відповідний фрагмент URemoveItemsOnFaintedByClassificationInventoryManagerFragment підписується на зміни GameplayTag Status.Fainted у AbilitySystemComponent персонажа та видаляє предмети із зазначеними класифікаціями. Це дозволяє підтримувати логіку втрати небезпечних чи заборонених предметів після непритомності.

Система одягу реалізована через UClothingInventoryItemFragment. Якщо предмет із таким фрагментом екіпіровано у слот із відповідним тером, персонаж отримує новий SkeletalMesh, що завантажується асинхронно, та GameplayEffect, який може впливати на поведінку NPC. У AEscapeChroniclesCharacter додано логіку для заміни моделі через метод SetMesh та її відновлення через ResetMesh. Наприклад, якщо гравець одягне форму охоронця вночі, охоронці не підійматимуть тривогу при зустрічі.

Для взаємодії з предметами в середовищі реалізовано систему, яка дозволяє використовувати певні предмети в конкретних умовах. Наприклад, предмет ляльки реалізований як UDollUsableInventoryItemFragment, який дозволяє гравцю покласти ляльку в ActivitySpot (наприклад, у ліжку), щоб створити ілюзію присутності в камері під час перевірки вночі. Цей фрагмент перевіряє, чи вибраний актор є ActivitySpot, і якщо так — викликає метод SpawnOccupyingDoll. У класі AActivitySpot реалізовано логіку асинхронного спавну актора ляльки, збереження її трансформації, знищення при перезавантаженні гри та повторного спавну при завантаженні. Таким чином, система підтримує повноцінне збереження і взаємодію з такими об'єктами.

Особливу увагу було приділено системі виявлення контрабандних предметів. Металодетектори реалізовані як актори AScanner, які перевіряють гравця при вході у тригер. Якщо в інвентарі знайдено предмети з класифікацією Contraband та немає предмета з фрагментом UContrabandBagInventoryItemFragment (тобто

"сумка-контрабанда"), сканер застосовує до гравця `GameplayEffect` (наприклад, миттєво підвищує підозрілість до максимуму) і відтворює звук на всіх клієнтах. Якщо контрабандна сумка присутня, її міцність зменшується через фрагмент витривалості (`UDurabilityInventoryItemFragment`), а покарання не застосовується.

Окремо була реалізована повноцінна система збереження предметів, розкиданих по рівню. Для цього створено сабсистему `USaveInventoryPickupItemsSubsystem`, яка реєструє всі актори типу `AEscapeChroniclesInventoryPickupItem` при їхньому спавні. Перед збереженням гри компонент проходиться по всіх активних `pickup`-акторах, зберігає їхній клас, трансформацію, клас предмета та його характеристики. При завантаженні гри система спочатку знищує всі наявні `pickup`-актори на рівні, після чого асинхронно завантажує їхні класи, реконструює `item instance` і спавнить відповідні актори в тих самих позиціях. Це забезпечує повноцінну підтримку збереження предметів, які були покинуті або спеціально розміщені в оточенні.

Усі ці системи були реалізовані як розширення базової архітектури інвентаря через фрагменти, що гарантує їхню модульність, масштабованість та незалежність. Вони дозволяють створити реалістичний та глибокий ігровий процес із динамічною реакцією світу на дії гравця та контекстні умови.

РОЗДІЛ 9
ПОШКОДЖЕННЯ ОБ'ЄКТІВ

У грі реалізована система пошкодження об'єктів, що базується на Boolean-операціях з геометрією через Geometry Script. Система дозволяє створювати отвори довільної форми в мешах на рівні, які представлені у вигляді DynamicMeshActor, — шляхом вирізання сфер із заданим радіусом у вказаній точці. Уся логіка реалізована у вигляді компонента UDestroyableComponent, який можна додати до будь-якого DynamicMeshActor. Компонент дозволяє не лише модифікувати меш, а й підтримує синхронізацію цих змін у мережевому середовищі та їх збереження у файл гри.

Кожен отвір описується структурою FDynamicMeshHoleData, що містить відносну позицію отвору (відносно компонента) та його радіус. Створення отвору відбувається шляхом побудови сферичного допоміжного меша (через Geometry Script) та віднімання його із основного меша за допомогою ApplyMeshBoolean з операцією Subtract. Це дозволяє створити реалістичну дірку у поверхні.

Щоб уникнути постійного накопичення модифікацій і дати можливість відновити початкову геометрію, компонент зберігає копію оригінального меша у вигляді об'єкта UDynamicMesh. Після цього кожна модифікація застосовується до поточної копії. Якщо потрібно відновити стан об'єкта (наприклад, при очищенні отворів або при завантаженні нового збереження), меш замінюється на збережену копію, після чого модифікації можуть застосовуватись знову.

Для підтримки збережень UDestroyableComponent реалізує інтерфейс ISaveable. У функції OnPreLoadObject видаляються всі попередні отвори, після чого в OnPostLoadObject усі збережені отвори повторно застосовуються до меша через AddHoleAtRelativeLocation_Internal. Усі отвори реплікуються через властивість Holes, яка має ReplicatedUsing, що дозволяє автоматично оновлювати стан меша на клієнтах у разі змін на сервері.

Крім основної функціональності, компонент має додаткові параметри:

- bAutomaticallyMakeActorSaveable — дозволяє автоматично зробити актор зберезуваним, додаючи тег (SaveableActorTag) у його список тегів.
- bAutomaticallyMakeActorReplicated — активує реплікацію для актора.
- bAutomaticallyAddAIPerceptionStimuliSourceComponent — додає компонент для сповіщення системи штучного інтелекту (Perception), що дозволяє, наприклад, охоронцям помічати пошкоджені об'єкти.

Також реалізовано підтримку фільтрації типу інструменту, яким дозволено завдавати шкоди об'єкту. Це налаштовується через властивість DestructiveToolType, що визначає тип дозволеного інструмента (наприклад, кирка). Усі інші інструменти не повинні мати змоги модифікувати актор з цим компонентом. Це дозволяє реалізовувати диференціацію інструментів у геймплеї.

У результаті дана система дозволяє модифікувати геометрію на рівні динамічно під час гри, з урахуванням обмежень, синхронізації в мережі, можливості збереження/завантаження та інтеграції зі штучним інтелектом. Такий підхід дозволяє реалізовувати механіки підкопів, руйнування, прихованих проходів та інших взаємодій з оточенням, що змінюється.

РОЗДІЛ 10
ІГРОВИЙ ЧАС ТА ЗМІНА ДНЯ І НОЧІ

У проєкті реалізована повноцінна система ігрового часу, що поділяє час на дні, години та хвилини. Одна хвилина ігрового часу відповідає одній реальній секунді, що дозволяє синхронізувати візуальні ефекти та ігрову логіку в реальному часі без надмірної складності.

Для представлення ігрового часу створені дві структури:

- `FGameplayTime` — представляє лише години та хвилини в межах одного дня. Містить численні оператори для порівняння, арифметики, нормалізації, інкременту/декременту та конвертації в загальну кількість хвилин.
- `FGameplayDateTime` — розширює `FGameplayTime`, додаючи поле «день» і дозволяючи працювати з часом у довготривалій перспективі. Також має зручні оператори додавання, віднімання, нормалізації та перетворення.

Поточний ігровий час зберігається в `GameState`, де щосекунди (тобто щохвилини ігрового часу) відбувається оновлення. Кожне оновлення супроводжується викликом делегата, до якого можуть підписатися інші системи, такі як день/ніч, розпорядок дня, події тощо. Це дозволяє централізовано синхронізувати зміни часу по всій грі.

Система дня і ночі реалізована в окремому акторі `Sky`, що містить два компоненти світла:

- `SunDirectionalLightComponent` — сонце, яке освітлює світ протягом дня.
- `MoonDirectionalLightComponent` — місяць, що світить вночі, обертаючись синхронно із сонцем, але з поворотом у 180° по осі `pitch`.

`Sky` підписується на делегата оновлення ігрового часу в `GameState`, після чого щосекунди (на `Tick`) обчислює плавне інтерпольоване значення між поточним часом та передбаченим наступним (через одну хвилину) і змінює кут освітлення сонця відповідно до цього значення. Таким чином, рух сонця та місяця виглядає безперервним та реалістичним.

Схід і захід сонця задаються через змінні типу `FGameplayTime`: `SunriseTime` і `SunsetTime`. Вони конвертуються у значення від 0 до 1 (як альфа), які відповідають прогресу протягом дня. В залежності від поточного часу визначається, чи є зараз день чи ніч, і обчислюється відповідне значення кута обертання сонця:

- Вдень (від сходу до заходу) сонце повертається з 180° до 360° .
- Вночі (від заходу до сходу) воно «перемотується» назад від 0° до 180° , забезпечуючи повний цикл.

Інтенсивність та температура місяця відрізняються від сонця: місяць має меншу інтенсивність і подвійну температуру, що дозволяє створити реалістичне нічне освітлення.

Загалом, система забезпечує:

- Чітку та зручну роботу з часом у межах одного дня або багатьох днів.
- Централізовану обробку подій, пов'язаних з часом.
- Реалістичну та плавну зміну освітлення залежно від часу доби.
- Повну підтримку збережень (через `SaveGame`-макриси у структурах).
- Просту інтеграцію в інші системи гри, включаючи ШІ, розклад активностей, NPC-логіку тощо.

Така реалізація дозволяє формувати живий, реактивний світ, де події чітко прив'язані до часу доби та прогресу гри.

РОЗДІЛ 11
ВВІМКНЕННЯ/ВИМКНЕННЯ ОСВІТЛЕННЯ ЗА РОЗКЛАДОМ

У проєкті реалізовано універсальний компонент `USceneComponentVisibilitySchedulerComponent`, що дозволяє автоматично вмикати або вимикати видимість певних компонентів актора відповідно до ігрового часу. Цей механізм дозволяє, зокрема, автоматизувати роботу освітлення — наприклад, увімкнути ліхтарі вночі та вимкнути їх удень — без ручного втручання чи скриптових рішень.

Компонент працює наступним чином:

- У ньому задається час початку та завершення, у який компоненти мають бути видимими (через поля типу `FGameplayTime`).
- Також у нього передається множина компонентів (`TargetComponentsReferences`), видимість яких слід контролювати.
- При запуску гри (в `BeginPlay`) компонент отримує доступ до поточного ігрового часу з `GameState`, виконує перше оновлення видимості та підписується на делегат зміни ігрового часу.
- Після кожного оновлення ігрового часу компонент перевіряє, чи поточний час потрапляє в заданий інтервал, і вмикає/вимикає видимість цільових компонентів.

Компонент також враховує випадки, коли час увімкнення починається пізніше часу вимкнення, наприклад, з 22:00 до 01:30 (тобто коли проміжок перетинає межу доби).

Крім того, компонент має опцію `bPropagateVisibilityToChildren`, яка дозволяє рекурсивно застосувати зміну видимості до дочірніх компонентів.

Цей механізм працює повністю автоматично та не потребує жодного додаткового втручання після встановлення — достатньо лише задати інтервал і перелік компонентів у редакторі. Таким чином, дана система є ефективним рішенням для сценаріїв, що потребують розкладу роботи компонентів, зокрема — освітлення.

РОЗДІЛ 12

ІГРОВІ ПОДІЇ

Система подій у грі є ключовим елементом управління ігровим процесом, що забезпечує структуровану організацію активностей для гравців та неігрових персонажів (NPC). Вона базується на двох основних типах подій: заплановані події (*Scheduled Events*) та активні події (*Active Events*). Ця система дозволяє динамічно керувати поведінкою персонажів, синхронізувати ігровий час із подіями та створювати різноманітні сценарії взаємодії між заручниками та охоронцями. У цьому розділі детально описано архітектуру системи подій, зосереджуючись на компоненті `UScheduleEventManagerComponent`, структурі `FScheduleEventData`, базовому класі `UScheduleEvent`, а також коротко розглянуто спеціалізовані типи подій.

Загальний огляд системи подій

Система подій побудована таким чином, щоб забезпечити чітке розмежування між запланованими подіями, які визначають розклад дня для заручників, та активними подіями, які диктують поведінку охоронців у певний момент. Заплановані події (*Scheduled Events*) синхронізуються з ігровим часом і автоматично змінюються залежно від нього, тоді як активні події (*Active Events*) можуть переривати заплановані, наприклад, у разі тривоги. Лише одна подія може бути активною в будь-який момент часу, а інші, якщо вони є, перебувають у стані паузи.

Ключовим елементом системи є компонент `UScheduleEventManagerComponent`, який керує стеком подій, їх створенням, активацією, паузою та завершенням. Події представлені екземплярами класів, похідних від базового класу `UScheduleEvent`, а їхні дані зберігаються у структурі `FScheduleEventData`. Деякі події підтримують механізм відмітки присутності (*Presence Mark*), який дозволяє відстежувати, чи виконав гравець необхідні дії, наприклад, прибув до певного місця.

Компонент `UScheduleEventManagerComponent`

Компонент `UScheduleEventManagerComponent` є центральним елементом системи подій і призначений для використання на сервері (authority). Він відповідає за створення, управління та перемикання між подіями, а також за їх синхронізацію з ігровим часом. Компонент працює зі стеком подій (`EventsStack`), де остання подія в стеку завжди є активною, а перша — запланованою.

Основні функції компонента

1. Управління стеком подій:

- Поле `EventsStack` типу `TArray<FScheduleEventData>` зберігає список усіх поточних подій. Перший елемент стеку завжди відповідає запланованій події, яка залежить від поточного ігрового часу, а останній елемент — це активна подія, яка визначає поточну поведінку в грі.
- Метод `PushEvent` додає нову подію на верх стеку, роблячи її активною, тоді як попередня активна подія (якщо вона існує) переходить у стан паузи.
- Метод `RemoveEvent` видаляє подію за її тегом (`FGameplayTag`), завершуючи її та активуючи або відновлюючи попередню подію в стеку, якщо вона існує.
- Метод `RemoveEventByIndex` дозволяє видалити подію за індексом у стеку, із можливістю вказати, чи потрібно зменшувати розмір масиву (`EAllowShrinking`) та чи слід запускати/відновлювати останню подію.

2. Синхронізація з ігровим часом:

- Компонент реагує на оновлення ігрового часу через метод `OnCurrentGameDateUpdated`. Цей метод перевіряє, чи потрібно змінити заплановану подію на основі нового часу, порівнюючи його з мапою `ScheduledEvents` (типу `TMap<FGameplayTime, FScheduleEventData>`), яка містить розклад подій.
- Якщо нова подія знайдена, вона замінює попередню заплановану подію в стеку, а стара подія завершується. Якщо подія є єдиною в стеку, вона автоматично стає активною.

3. Створення та ініціалізація подій:

- Метод `CreateEventInstanceAndStartOrPauseIt` створює екземпляр події на основі даних `FScheduleEventData`. Подія може завантажуватися синхронно або асинхронно залежно від параметра `bLoadEventClassSynchronously`. Після завантаження класу події викликається метод `OnEventClassLoaded`, який ініціалізує екземпляр і запускає його (або ставить на паузу, якщо подія не є активною).
- Компонент підтримує делегати (`OnCurrentScheduledEventChanged` та `OnCurrentActiveEventChanged`), які сповіщають про зміну запланованої або активної події, а також делегати для відстеження створення екземплярів подій (`OnCurrentScheduledEventInstanceCreated`, `OnCurrentActiveEventInstanceCreated`).

4. Життєвий цикл компонента:

- У методі `BeginPlay` компонент ініціалізується, підписуючись на оновлення ігрового часу через `AEscapeChroniclesGameState`.
- У методі `EndPlay` усі події в стеку завершуються, а їхні дані очищаються (за винятком даних, пов'язаних із збереженням).

Збереження даних стеку подій

Поле `EventsStack` позначене як `SaveGame`, що дозволяє зберігати поточний стан стеку подій при збереженні гри. Це забезпечує можливість відновлення послідовності подій після завантаження гри. Перед завантаженням гри компонент зберігає копію стеку в `EventsStackBeforeLoadingGame`, щоб порівняти її з новим станом після завантаження. Події, які більше не присутні в стеку після завантаження, завершуються, а нові створюються та ініціалізуються.

Архітектурні особливості

- **Одночасна активність лише однієї події:** Система гарантує, що лише остання подія в стеку є активною, що спрощує керування логікою гри.
- **Гнучкість пріоритетів:** Додавання нової події через `PushEvent` дозволяє тимчасово перервати заплановану подію, що корисно для таких сценаріїв, як тривога.
- **Асинхронне завантаження:** Події можуть завантажуватися асинхронно через `UAssetManager`, що оптимізує продуктивність гри, особливо при роботі з великими класами подій.
- **Делегати для сповіщень:** Використання делегатів забезпечує гнучку взаємодію з іншими системами гри, дозволяючи, наприклад, сповіщати клієнтів про зміну подій через `GameState`.

Структура `FScheduleEventData`

Структура `FScheduleEventData` є контейнером для даних, необхідних для створення та управління подіями. Вона включає:

- **Тег події** (`FGameplayTag EventTag`): Унікальний ідентифікатор події, який використовується для її пошуку та видалення.
- **Клас події** (`TSoftClassPtr<UScheduleEvent> EventClass`): М'яке посилання на клас події, що дозволяє асинхронне завантаження.
- **Екземпляр події** (`UScheduleEvent* EventInstance`): Вказівник на створений екземпляр події після її ініціалізації.

Ця структура забезпечує уніфікований спосіб зберігання інформації про події, дозволяючи компоненту `UScheduleEventManagerComponent` ефективно керувати стеком подій.

Базовий клас `UScheduleEvent`

Клас `UScheduleEvent` є базовим для всіх подій у грі та реалізує інтерфейс `FTickableGameObject`, що дозволяє подіям виконувати логіку щокадру, якщо це необхідно.

Клас визначає основний життєвий цикл події через методи:

- **StartEvent:** Запускає подію, викликаючи віртуальний метод OnEventStarted. Якщо подія запускається в паузі (bStartPaused = true), вона одразу переходить у стан паузи.
- **EndEvent:** Завершує подію, викликаючи OnEventEnded і передаючи причину завершення (EScheduleEventEndReason).
- **PauseEvent:** Ставить подію на паузу, викликаючи OnEventPaused і вимикаючи тікання.
- **ResumeEvent:** Відновлює подію, викликаючи OnEventResumed і вмикаючи тікання.

Ключові особливості

- **Тікання:** Події можуть бути налаштовані для тікання через поле bCanEverTick, яке встановлюється в конструкторі. Тікання активується лише для активних і непаузених подій.
- **Дані події:** Кожна подія пов'язана з екземпляром FScheduleEventData, доступним через метод GetEventData.
- **Перевірки безпеки:** Використовуються макроси DO_ENSURE і DO_CHECK для забезпечення коректності стану подій, наприклад, що подія не запускається повторно або що bCanEverTick встановлюється лише в конструкторі.

Цей клас забезпечує гнучку основу для створення спеціалізованих подій, таких як UAlertScheduleEvent, UBedtimeScheduleEvent, URollCallScheduleEvent тощо.

Спеціалізовані типи подій

Нижче коротко описано основні типи подій, які розширюють базовий клас UScheduleEvent або його похідні, такі як UScheduleEventWithPresenceMark. Кожен тип має унікальну логіку, яка інтегрується з загальною системою подій.

Події з відміткою присутності (UScheduleEventWithPresenceMark)

Клас UScheduleEventWithPresenceMark є базовим для подій, які вимагають від гравців фізичної присутності в певних зонах, визначених тригерами (PresenceMarkTriggers).

Основні особливості:

- **Відстеження присутності:** Гравці відзначаються як присутні через перетин із тригерами класу, визначеного в PresenceMarkTriggerClass. Метод OnPresenceMarkTriggerComponentBeginOverlap обробляє початок перетину, а OnPresenceMarkTriggerEndOverlap — його завершення.
- **Геймплейні ефекти:** Присутнім гравцям застосовується ефект CheckedInGameplayEffectClass із тером Status_CheckedIn, а гравцям, які пропустили подію, — ефект MissedPlayerGameplayEffectClass.
- **Список присутніх і відсутніх:** Поле CheckedInPlayers зберігає ідентифікатори присутніх гравців, а PlayersThatMissedAnEvent — тих, хто пропустив подію. Метод CollectPlayersThatMissedAnEvent викликається при завершенні події для обробки відсутніх гравців.
- **Віртуальні методи:** Клас дозволяє налаштування через перевизначення методів, таких як CanCheckInPlayer (перевірка, чи може гравець бути відзначений) і NotifyPlayerMissedEvent (обробка пропуску події).

Подія "Тиха година" (UBedtimeScheduleEvent)

Ця подія вимагає, щоб заручники дісталися до своїх камер протягом певного часу (TimeForPlayersToCheckIn). Основні особливості:

- **Часовий ліміт:** Після закінчення часу, визначеного в TimeForPlayersToCheckIn, двері, зазначені в DoorsToLock, зачиняються, а гравці, які не встигли, викликають тривогу через AlertEventData.
- **Перевірка камер:** Метод CanCheckInPlayer перевіряє, чи гравець перебуває у своїй камері, використовуючи компонент UPlayerOwnershipComponent.
- **Блокування дверей:** Поле CachedDoorsInstancesToLock містить екземпляри

дверей, які блокуються через метод `SetDoorsLocked`.

- **Прискорення часу:** Якщо всі гравці перебувають у своїх камерах, ігровий час може прискорюватися до завершення події.

Подія "Переключка" (`URollCallScheduleEvent`)

Ця подія вимагає присутності заручників у загальному залі. Основні особливості:

- **Запуск тривоги:** Якщо гравець пропускає подію, метод `NotifyPlayerMissedEvent` запускає подію тривоги через `AlertEventData`.
- **Охорона:** Охоронці супроводжують заручників до залу, що реалізується через логіку AI, інтегровану з подією.

Подія "Прийом їжі" (`UMealScheduleEvent`)

Ця подія відповідає за організацію прийому їжі в їдальні. Основні особливості:

- **Оновлення зон активності:** Поле `ActivitySpotsClassesToUpgrade` визначає, які зони активності (`AActivitySpot`) отримують підвищений рівень геймплейних ефектів під час події. Метод `SetActivitySpotsUpgraded` застосовує або скидає ці ефекти.
- **Охорона:** Охоронці чергують поблизу, що керується логікою AI.

Подія "Душ" (`UShowerScheduleEvent`)

Ця подія аналогічна "Прийому їжі", але відбувається в душових. Основні особливості:

- **Відмітка присутності:** Гравці повинні зайняти місця в душових зонах, позначених тригерами.
- **Охорона:** Частина охоронців патрулює поблизу, що реалізується через інтеграцію з системою AI.

Подія "Перерва"

Ця подія надає гравцям вільний час без строгих вимог до присутності. Основні особливості:

- **Відсутність тригерів:** Подія не вимагає відмітки присутності, дозволяючи гравцям вільно пересуватися.
- **Пасивна охорона:** Охоронці можуть патрулювати територію, але без активного втручання, якщо немає порушень.

Подія "Тривога" (`UAlertScheduleEvent`)

Подія тривоги є особливою, оскільки вона перериває інші події, ставлячи їх на паузу.

Основні особливості:

- **Позначення гравців:** Гравці, позначені як "розшукувані" (`WantedPlayers`), отримують геймплейний ефект `WantedGameplayEffectClass` із тером `Status_Wanted`. Методи `AddWantedPlayer` і `SetWantedPlayers` додають гравців до списку розшукуваних, а `RemoveWantedPlayer` видаляє їх.
- **Очищення пошкоджень:** При нормальному завершенні події (`EScheduleEventEndReason::Normal`) усі отвори в `UDestructibleComponent` об'єктів `ADynamicMeshActor` очищаються через метод `ClearAllHoles`.
- **Завершення події:** Подія автоматично завершується, коли список розшукуваних гравців (`WantedPlayersInGame`) стає порожнім.
- **Реакція на підключення/відключення гравців:** Методи `OnPlayerOrBotInitialized` і `OnPlayerOrBotLogout` обробляють підключення або відключення гравців, оновлюючи списки `WantedPlayers` і `WantedPlayersInGame`.

Висновки

Система подій забезпечує гнучке та надійне керування ігровими сценаріями, дозволяючи створювати складні взаємодії між заручниками та охоронцями. Компонент `UScheduleEventManagerComponent` є основою системи, ефективно керуючи стеком подій і забезпечуючи їх синхронізацію з ігровим часом. Структура `FScheduleEventData` уніфікує дані подій, а базовий клас `UScheduleEvent` надає гнучку основу для реалізації різноманітних

типів подій. Спеціалізовані події, такі як "Тиха година", "Перекличка", "Прийом їжі", "Душ", "Перерва" та "Тривога", додають унікальну логіку, інтегруючись із системами AI, геймплейними ефектами та фізичними тригерами. Ця архітектура дозволяє легко розширювати систему новими типами подій, зберігаючи її масштабованість і продуктивність.

РОЗДІЛ 13
МУЗИКА ДЛЯ РІЗНИХ ПОДІЙ

Система музичного супроводу подій у грі відіграє важливу роль у створенні атмосфери та зануренні гравців у ігровий процес. Вона забезпечує динамічний підбір та відтворення музики залежно від активної події (*Active Event*), а також плавні переходи між музичними композиціями при зміні подій. У цьому розділі описано архітектуру системи музичного супроводу, її інтеграцію з компонентом `UScheduleEventManagerComponent` та механізми управління звуковими ефектами.

Система музичного супроводу базується на кастомному компоненті `UAudioComponent`, який розміщений на об'єкті `AEscapeChroniclesGameState`. Цей компонент відповідає за відтворення фонові музики, прив'язаної до тега активної події (`FGameplayTag` у структурі `FScheduleEventData`), а також за відтворення коротких звукових ефектів під час переходу між подіями. Система інтегрована з компонентом `UScheduleEventManagerComponent`, що дозволяє синхронізувати музику з поточним станом ігрових подій.

Кастомний `UAudioComponent`, розташований на `AEscapeChroniclesGameState`, є центральним елементом системи. Він налаштований для автоматичного відстеження змін активної події через делегат `OnCurrentActiveEventChanged` компонента `UScheduleEventManagerComponent`.

Основні функції компонента:

1. Відтворення музики для подій:

- Кожна подія в грі має унікальний тег (`FGameplayTag`), який асоціюється з певною аудіокомпозицією. Ці асоціації зберігаються в конфігураційній структурі або таблиці даних, доступній для `UAudioComponent`.
- При зміні активної події компонент отримує новий тег через делегат `OnCurrentActiveEventChanged` і завантажує відповідну аудіокомпозицію.

2. Звукові ефекти переходу:

- Під час перемикавання між подіями компонент відтворює короткий звуковий ефект (наприклад, звуковий сигнал або мелодійний акцент), який позначає зміну ігрового контексту.
- Цей ефект відтворюється перед початком нової композиції, щоб забезпечити плавний перехід і уникнути різких змін у звуковому ландшафті.

3. Управління звуковими параметрами:

- Компонент підтримує налаштування гучності, затухання (*fade-in/fade-out*) та інших аудіопараметрів для забезпечення комфортного слухового досвіду.
- Наприклад, при завершенні події музика може поступово затихати, а при початку нової — плавно наростати.

Компонент `UScheduleEventManagerComponent` є джерелом даних про поточну активну подію. Його делегат `OnCurrentActiveEventChanged` передає об'єкту `UAudioComponent` інформацію про стару та нову подію (`FScheduleEventData OldEventData` та `FScheduleEventData NewEventData`). Це дозволяє компоненту:

- Визначити тег нової події (`NewEventData.EventTag`).
- Завантажити та відтворити асоційовану аудіокомпозицію.
- Відтворити звуковий ефект переходу перед зміною музики.

Для оптимізації продуктивності аудіокомпозиції завантажуються асинхронно через `UAssetManager`, подібно до того, як це реалізовано для класів подій у `UScheduleEventManagerComponent`. Це зменшує затримки при перемиканні подій і забезпечує плавний ігровий процес навіть на менш потужних системах.

Логіка роботи системи

1. Ініціалізація:

- Під час запуску гри (`BeginPlay` у `AEscapeChroniclesGameState`) `UAudioComponent` ініціалізується та підписується на делегат `OnCurrentActiveEventChanged` компонента `UScheduleEventManagerComponent`.
- Компонент завантажує початкову композицію, що відповідає першій активній події.

2. Обробка зміни події:

- При виклику делегата `OnCurrentActiveEventChanged` компонент отримує тег нової події.
- Якщо стара композиція ще відтворюється, вона затихає (*fade-out*).
- Відтворюється короткий звуковий ефект переходу.
- Завантажується та відтворюється нова композиція з поступовим наростанням гучності (*fade-in*).

3. Завершення роботи:

- У методі `EndPlay` компонент припиняє відтворення музики та очищає всі активні звукові ресурси, щоб уникнути витоків пам'яті.

Особливості реалізації

- **Гнучкість налаштувань:** Асоціації між тегами подій і аудіокомпозиціями можуть налаштовуватися через редактор Unreal Engine, що полегшує додавання нових подій та музики без змін у коді.
- **Підтримка різноманітних подій:** Система адаптована для всіх типів подій, включаючи "Тиху годину", "Перекличку", "Прийом їжі", "Душ", "Перерву" та "Тривогу". Наприклад, подія "Тривога" може мати напружену музику, тоді як "Перерва" — спокійну.
- **Оптимізація продуктивності:** Асинхронне завантаження та кешування аудіоресурсів зменшують навантаження на систему.
- **Плавні переходи:** Використання *fade-in/fade-out* та звукових ефектів переходу створює природний і приємний слуховий досвід.

Система музичного супроводу подій забезпечує динамічне та атмосферне звукове оформлення гри, тісно інтегроване з системою подій. Кастомний `UAudioComponent` на `AEscapeChroniclesGameState` ефективно відстежує зміни активних подій через `UScheduleEventManagerComponent`, відтворюючи відповідну музику та звукові ефекти переходу. Гнучка архітектура системи дозволяє легко розширювати її для нових подій, а асинхронне завантаження та оптимізоване управління звуком гарантують високу продуктивність. Ця система значно підвищує занурення гравців, підкреслюючи унікальний настрій кожної ігрової події.

РОЗДІЛ 14
КЛІЄНТ-СЕРВЕРНЕ ПІДКЛЮЧЕННЯ

Система клієнт-серверного підключення у грі забезпечує стабільне та гнучке мережеве взаємодію між гравцями через модель Peer-to-Peer (P2P) з одним гравцем у ролі хоста (Listen Server). Для реалізації використано абстрактний інтерфейс OnlineSubsystem з інтеграцією Steamworks, що дозволяє використовувати Steam Sockets для надійного з'єднання без необхідності відкривати порти. Уся логіка управління сесіями, підключенням, переходом між рівнями та відключенням клієнтів централізована в класі UEscapeChroniclesGameInstance. Цей розділ описує архітектуру системи, її ключові компоненти, налаштування та принципи роботи.

Гра використовує P2P-модель, де один гравець виступає хостом (Listen Server), а інші гравці підключаються до нього як клієнти через систему запрошень Steam. Для мережевої взаємодії застосовується транспортний рівень Steam Sockets, який забезпечує надійне з'єднання та автоматичне керування мережею. Система сесій побудована на основі OnlineSessionInterface, що дозволяє створювати, приєднуватися та знищувати сесії. Абстракція через OnlineSubsystem забезпечує можливість заміни платформи (наприклад, на Epic Online Services) без значних змін у коді. Налаштування на стороні проекту, зокрема в DefaultEngine.ini, активує Steam як основну платформу та встановлює тестовий SteamAppId=480.

Клас UEscapeChroniclesGameInstance є центральним компонентом для управління мережевою логікою. Він відповідає за створення та знищення сесій, переміщення гравців між рівнями, обробку запрошень і відключення клієнтів.

Основні функції класу:

1. Ініціалізація:

- У методі Init клас підписується на делегат OnSessionUserInviteAccepted через IOnlineSessionPtr, щоб обробляти запрошення до сесій. Також ініціалізується зв'язок із UEscapeChroniclesGameUserSettings для управління звуковими налаштуваннями.

2. Створення сесії (StartHostSession):

- Викликається, коли гравець хоче стати хостом. Метод створює сесію з налаштуваннями (FOnlineSessionSettings), що включають максимальну кількість гравців (MaxPlayersPerSession), підтримку запрошень і приєднання через присутність у Steam.
- Делегати OnCreateSessionComplete та OnStartSessionComplete дозволяють відстежувати успішність створення та запуску сесії.
- Після успішного створення сесія стартує, дозволяючи клієнтам приєднатися через запрошення.

3. Перехід між рівнями (ServerTravelByLevelSoftObjectPtr):

- Використовується для переміщення хоста та клієнтів на новий рівень. Метод конвертує TSoftObjectPtr<UWorld> у URL-адресу рівня та додає параметр ?listen, якщо потрібен Listen Server.
- Викликається після створення сесії або для повернення до головного меню.

4. Відключення клієнтів (KickPlayer):

- Переміщує віддаленого клієнта до головного меню через ClientTravelToMainMenu. Метод перевіряє, чи можна вигнати гравця (CanKickPlayer), щоб уникнути помилкового відключення хоста чи локального клієнта.

5. Знищення сесії (DestroyHostSession та DestroySession):

- DestroyHostSession викликається хостом для завершення гри. Спочатку виганяються всі віддалені клієнти, після чого сесія знищується.
- DestroySession знищує сесію для хоста або клієнта, викликаючи делегат OnDestroySessionComplete. Якщо вказано bServerTravelToMainMenu, після знищення сесії виконується перехід до головного меню.

6. Обробка запрошень (OnSessionUserInviteAccepted):

- Викликається, коли гравець приймає запрошення. Якщо гравець уже хостить сесію, його сесія знищується через DestroyHostSession. Для клієнта знищується локальна сесія через DestroySession, після чого викликається JoinSession.

7. Приєднання до сесії (OnJoinSessionComplete):

- Після успішного приєднання клієнти переміщуються на рівень хоста через ClientTravel з використанням ConnectString, отриманої з GetResolvedConnectString.

Налаштування в DefaultEngine.ini забезпечують інтеграцію зі Steam:

- DefaultPlatformService=Steam: Встановлює Steam як основну платформу.
- SteamDevAppId=480: Використовується для тестування (Spacewar). У релізній версії замінюється унікальним AppId гри.
- bInitServerOnClient=true: Дозволяє ініціалізацію серверної логіки на клієнті для P2P.
- NetConnectionClassName: Вказує клас для мережових з'єднань через Steam Sockets.

```
[OnlineSubsystem]
DefaultPlatformService=Steam

[OnlineSubsystemSteam]
bEnabled=true
SteamDevAppId=480
SteamAppId=480

bInitServerOnClient=true

[/Script/OnlineSubsystemSteam.SteamNetDriver]
NetConnectionClassName="OnlineSubsystemSteam.SteamNetConnection"
```

Використання Steam Sockets через OnlineSubsystemSteam забезпечує:

- **Надійне з'єднання:** Steam обробляє NAT traversal, усуваючи потребу у відкритих портах.
- **Система запрошень:** Гравці можуть надсилати запрошення через Steam Overlay, що спрощує підключення.
- **Абстракція платформи:** Логіка гри використовує IOnlineSessionPtr, що дозволяє замінити Steam на іншу платформу без переписування коду.

Логіка роботи системи

1. **Створення сесії:**
 - Гравець викликає StartHostSession, створюючи сесію з унікальним ім'ям (NAME_GameSession).
 - Після успішного створення сесія стартує, і хост може надсилати запрошення через Steam.
2. **Приєднання клієнтів:**
 - Клієнт приймає запрошення, викликаючи OnSessionUserInviteAccepted.
 - Якщо клієнт уже в сесії, його поточна сесія знищується.
 - Клієнт приєднується до сесії хоста через JoinSession і переміщується на його рівень.
3. **Перехід між рівнями:**
 - Хост викликає ServerTravelByLevelSoftObjectPtr для зміни рівня. Клієнти автоматично слідують за хостом.
4. **Завершення гри:**
 - Хост викликає DestroyHostSession, виганяючи клієнтів і знищуючи сесію.
 - Клієнти повертаються до головного меню через ClientTravelToMainMenu.
 - Якщо вказано bServerTravelToMainMenu, хост також переходить до головного меню.
5. **Обробка помилок:**

- Використовуються макроси DO_ENSURE і DO_CHECK для перевірки валідності об'єктів і умов.
- У разі невдачі (наприклад, якщо не вдалося отримати ConnectString) сесія знищується, щоб уникнути неконсистентного стану.

Особливості реалізації

- **Централізація логіки:** Уся мережева логіка зосереджена в UEscapeChroniclesGameInstance, що спрощує підтримку та розширення.
- **Гнучкість платформи:** Використання абстрактного OnlineSubsystem дозволяє адаптувати гру до інших платформ.
- **Оптимізація делегатів:** Делегати очищаються після виконання, щоб уникнути витоків пам'яті.
- **Тестування через SteamAppId=480:** Використання тестового AppId дозволяє розробникам перевіряти мережеву функціональність без релізного AppId.
- **Підтримка до 4 гравців:** Параметр MaxPlayersPerSession обмежує кількість гравців у сесії, що відповідає дизайну гри.

Система клієнт-серверного підключення забезпечує надійне та зручне P2P-з'єднання через Steam Sockets, інтегроване з OnlineSubsystem. Централізація логіки в UEscapeChroniclesGameInstance спрощує керування сесіями, підключенням і переходами між рівнями. Абстракція через OnlineSessionInterface робить систему гнучкою для майбутніх змін платформи, а налаштування в DefaultEngine.ini забезпечують легку інтеграцію зі Steam. Використання запрошень і автоматичне керування з'єднаннями створюють комфортний досвід для гравців, тоді як ретельна обробка помилок і оптимізація делегатів гарантують стабільність системи.

РОЗДІЛ 15

UI

Користувачський інтерфейс (UI) у грі відіграє ключову роль у забезпеченні зручної взаємодії гравця з грою. Цей розділ зосереджується на трьох основних компонентах UI, розроблених автором: головному меню, меню налаштувань та головному інтерфейсі гравця. Інші елементи UI, створені іншими розробниками, тут не розглядаються. Опис включає архітектуру, функціональність та інтеграцію з іншими системами гри, зокрема з `UEscapeChroniclesGameUserSettings` та `UEscapeChroniclesGameInstance`.

1. Головне меню

Головне меню є початковою точкою взаємодії гравця з грою. Воно забезпечує доступ до основних функцій, таких як початок нової гри, приєднання до сесії, налаштування параметрів та вихід із гри. Меню реалізовано як віджет без застосування стилів, що передбачає використання стандартного вигляду Unreal Engine для прототипування.

Архітектура

- **Реалізація:** Головне меню реалізовано як віджет на основі `UCommonUserWidget` (або похідного класу), що інтегрується з `UEscapeChroniclesGameInstance` для управління мережевими сесіями.
- **Функціональність:**
 - Кнопка "Нова гра" викликає `StartHostSession` у `UEscapeChroniclesGameInstance`, створюючи нову сесію, де гравець стає хостом.
 - Кнопка "Приєднатися" обробляє запрошення через Steam, викликаючи `OnSessionUserInviteAccepted` для приєднання до існуючої сесії.
 - Кнопка "Налаштування" відкриває віджет `USettingsMenuWidget` для зміни параметрів гри.
 - Кнопка "Вихід" завершує виконання гри.
- **Інтеграція:** Меню взаємодіє з `UEscapeChroniclesGameInstance` для обробки мережових подій та переходів між рівнями. Наприклад, після створення сесії викликається `ServerTravelByLevelSoftObjectPtr` для завантаження ігрового рівня.

Особливості

- Меню не містить кастомних стилів, що спрощує розробку та тестування.
- Логіка підключення до сесій централізована в `UEscapeChroniclesGameInstance`, що забезпечує модульність.
- Використання делегатів (`OnJoinSession`, `OnCreateSessionComplete`) дозволяє відстежувати статус мережових операцій.

2. Меню налаштувань

Меню налаштувань дозволяє гравцям змінювати звукові та графічні параметри гри. Воно складається з двох вкладок: звукових налаштувань (`USoundSettingsMenuWidget`) та графічних налаштувань (`UGraphicsSettingsMenuWidget`), які перемикаються через `UCommonActivatableWidgetSwitcher`. Налаштування зберігаються через кастомний клас `UEscapeChroniclesGameUserSettings`.

Архітектура

- **Базовий віджет (`USettingsMenuWidget`):**
 - Використовує `UCommonActivatableWidgetSwitcher` для перемикання між вкладками звуку та графіки.
 - Містить кнопки "Застосувати" (`ApplyButton`) та "Назад" (`BackButton`), які викликають делегат `OnApply` або деактивують віджет.
 - Ініціалізує дочірні віджети (`GraphicsSettingsMenuWidget` та `SoundSettingsMenuWidget`) у `NativePreConstruct`.
- **Звукові налаштування (`USoundSettingsMenuWidget`):**
 - Містить три слайдери (`USliderWidgetContainer`) для регулювання гучності: загальної (`MasterVolumeSlider`), музики (`MusicVolumeSlider`) та звукових ефектів (`SFXVolumeSlider`).
 - Ініціалізує значення слайдерів із `UEscapeChroniclesGameUserSettings` у

- InitializeSlidersValuesFromGameUserSettings.
- При натисканні кнопки "Застосувати" оновлює значення в GameUserSettings, викликає ApplySoundSettings і зберігає налаштування.
- **Графічні налаштування (UGraphicsSettingsMenuWidget):**
 - Використовує UComboBoxStringWidgetContainer для вибору параметрів, таких як режим вікна, роздільна здатність, VSync та різні рівні якості графіки (наприклад, тіні, текстури, антиаліасинг).
 - Мапи (TMap) типу enum або bool до FText забезпечують конвертацію між текстовими значеннями та внутрішніми типами.
 - Спеціальний перелік EGraphicsQuality конвертується в int32, сумісний з API Unreal Engine, для точного налаштування графічних параметрів.
 - Ініціалізує значення комбо-боксів із поточних налаштувань GameUserSettings.
 - При натисканні "Застосувати" оновлює всі параметри через GameUserSettings->ApplySettings.

Інтеграція

- **Звук:**
 - Налаштування гучності зв'язані з USoundClass (MasterSoundClasses, MusicSoundClasses, SFXSoundClasses), визначеними в UEscapeChroniclesGameInstance.
 - GameUserSettings->ApplySoundSettings викликає делегат OnApplySoundSettings, який оновлює AudioDevice через SetSoundMixClassOverride у GameInstance.
- **Графіка:**
 - Використовує методи GameUserSettings для встановлення роздільної здатності, режиму вікна, VSync та рівнів якості графіки.
 - UKismetSystemLibrary::GetSupportedFullscreenResolutions заповнює список доступних роздільних здатностей.
- **Збереження:** Налаштування зберігаються через GameUserSettings->SaveSettings, що записує їх у конфігураційний файл.

Особливості

- **Модульність:** Використання UCommonActivatableWidgetSwitcher дозволяє легко додавати нові вкладки налаштувань.
- **Гнучкість графіки:** EGraphicsQuality забезпечує точне керування якістю через конвертацію в int32.
- **Оптимізація:** Кешування GameUserSettings у UGraphicsSettingsMenuWidget зменшує кількість викликів.
- **Інтуїтивність:** Слайдери та комбо-бокси забезпечують зручне керування параметрами.

3. Головний інтерфейс гравця

Головний інтерфейс гравця відображає ключову інформацію про стан гри та персонажа. Він включає три віджети: UGameplayAttributeValueWidget (характеристики персонажа), UCurrentGameDateTimeWidget (ігровий час і день) та UCurrentActiveEventWidget (поточна активна подія та статус "переключки"). Ці віджети об'єднані в єдиний Blueprint (WBP), а не в коді, для гнучкого розташування та стилізації.

Архітектура

- **Віджет характеристик (UGameplayAttributeValueWidget):**
 - Відображає значення ігрового атрибута (FGameplayAttribute) з UAbilitySystemComponent гравця.
 - Ініціалізується через OnOwningPlayerStateInitialized, де підключається до PlayerState та слухає зміни атрибута через GetGameplayAttributeValueChangeDelegate.
 - Оновлює текст (UCommonTextBlock) із значенням атрибута та постфіксом (AttributeValueTextPostfix).
- **Віджет часу (UCurrentGameDateTimeWidget):**

- Відображає поточний ігровий день і час (FGameplayDateTime) через два текстові блоки (DayTextWidget, TimeTextWidget).
- Підключається до AEscapeChroniclesGameState у NativeConstruct і слухає оновлення часу через OnCurrentGameDateTimeUpdated.
- Форматує час у вигляді "Година:Хвилина" (наприклад, "9:00").
- **Віджет активної події (UCurrentActiveEventWidget):**
 - Відображає назву поточної активної події (FScheduleEventData) через UCommonTextBlock та статус "переключки" через UCommonLazyImage.
 - Назви подій беруться з TMap<FGameplayTag, FText> EventsNames.
 - Статус "переключки" оновлюється через OnCheckedInStatusChanged, реагуючи на тег Status_CheckedIn у UAbilitySystemComponent.
 - Деякі події (з EventsHidingCheckedInStatusImageWidget) приховують іконку статусу.
 - Використовує TMap<bool, TSoftObjectPtr<UTexture2D>> CheckedInStatusImages для відображення іконок залежно від статусу.

Інтеграція

- **3 GameState:** UCurrentGameDateTimeWidget та UCurrentActiveEventWidget слухають оновлення часу та подій через делегати OnCurrentGameDateTimeUpdated та OnCurrentActiveEventChanged у AEscapeChroniclesGameState.
- **3 PlayerState:** UGameplayAttributeValueWidget та UCurrentActiveEventWidget взаємодіють із UAbilitySystemComponent через AEscapeChroniclesPlayerState для отримання атрибутів та статусу.
- **Стилізація:** Стилі застосовані в Blueprint (WBP), що дозволяє легко змінювати розташування та зовнішній вигляд віджетів.

Особливості

- **Динамічність:** Віджети автоматично оновлюються при зміні стану гри чи персонажа.
- **Гнучкість:** Об'єднання в WBP дозволяє легко налаштовувати UI без змін у коді.
- **Оптимізація:** Використання UCommonLazyImage для іконок статусу зменшує витрати на завантаження текстур.
- **Контекстність:** Статус "переключки" приховується для подій, де він нерелевантний.

РОЗДІЛ 16
СТВОРЕННЯ БОТІВ

Система створення ботів у грі відповідає за генерацію та ініціалізацію ботів (штучного інтелекту) різних типів, таких як в'язні, охоронці та медики, з урахуванням їхньої ролі в грі та місця появи. Ця система реалізована через компонент `UBotSpawnerComponent`, який розміщено на `AEscapeChroniclesGameMode` і викликається у методі `BeginPlay`. Вона також інтегрується з системою вибору точок появи (`AEscapeChroniclesPlayerStart`), яка використовує теги `FGameplayTag` для визначення відповідності між гравцем і точкою появи. Цей розділ описує архітектуру, логіку роботи, інтеграцію та особливості системи створення ботів.

Система створення ботів забезпечує асинхронне завантаження класів персонажів, їхній спавн на відповідних точках появи (`PlayerStart`) та ініціалізацію через `AEscapeChroniclesGameMode`. Кожен бот отримує унікальне ім'я з набору, визначеного в `UBotNames`, та унікальний ідентифікатор через `USaveGameSubsystem`. Для точного розміщення ботів використовується кастомна логіка вибору `PlayerStart`, яка враховує теги, застосовані до `PlayerState` через `UAbilitySystemComponent`. Система підтримує три типи ботів: в'язні, охоронці та медики, кожен із власним класом персонажа та кількістю для спавну.

`UBotSpawnerComponent` є основним компонентом для створення ботів. Він розташований на `AEscapeChroniclesGameMode` і викликає спавн ботів у `BeginPlay`.

Основні елементи:

1. Властивості компонента:

- `PrisonerCharacterClass`, `GuardCharacterClass`, `MedicCharacterClass`: `TSoftClassPtr<AEscapeChroniclesCharacter>` для асинхронного завантаження класів персонажів в'язнів, охоронців та медиків.
- `PrisonerBotsCount`, `GuardBotsCount`, `MedicBotsCount`: Кількість ботів кожного типу для спавну.
- `BotNamesDataAsset`: Асет даних (`UBotNames`), що містить список імен для ботів.
- `ClaimedBotNames`: `TSet<FString>` для відстеження вже використаних імен, позначений як `SaveGame` для збереження між сеансами.

2. Методи спавну:

- `SpawnBots()`: Запускає асинхронне завантаження класів персонажів через `UAssetManager::GetStreamableManager`. Після завантаження викликаються методи `OnPrisonerCharacterClassLoaded`, `OnGuardCharacterClassLoaded` або `OnMedicCharacterClassLoaded`.
- `SpawnBots(uint8 NumberOfBotsToSpawn, TSubclassOf<AEscapeChroniclesCharacter> CharacterClass)`: Викликає `SpawnOneBot` потрібну кількість разів для заданого класу.
- `SpawnOneBot(TSubclassOf<AEscapeChroniclesCharacter> CharacterClass)`: Спавнить одного бота, викликає `SpawnDefaultController` і підписується на делегат `ReceiveControllerChangedDelegate` для обробки ініціалізації контролера.

3. Обробка контролерів:

- `OnCharacterControllerChanged`: Викликається після ініціалізації контролера бота. Перевіряє, чи контролер є `AAIController`, і чекає, поки пешка буде встановлена (`OnControllerPossessedPawnChanged`).
- `OnControllerPossessedPawnChanged`: Виконує рестарт бота через `GameMode->RestartPlayer`, призначає ім'я через `SelectBotName` та ініціалізує `PlayerState` через `GameMode->LoadAndInitBot`.

4. Управління іменами:

- `SelectBotName`: Вибирає випадкове ім'я з `BotNamesDataAsset`, яке ще не використано (`ClaimedBotNames`). Якщо всі імена зайняті, вибирається будь-яке ім'я з доступних.

`AEscapeChroniclesPlayerStart` розширює стандартний `APlayerStart` для підтримки тегової системи спавну:

1. Властивості:

- `RequiredGameplayTagsToSpawn`: `FGameplayTagContainer`, який визначає теги,

необхідні для спавну пешки на цій точці.

2. Метод **CanSpawnPawn**:

- Перевіряє, чи має PlayerState (AEscapeChroniclesPlayerState) усі необхідні теги через UAbilitySystemComponent::HasAllMatchingGameplayTags.
- Якщо AbilitySystemComponent недоступний, дозволяє спавн лише за відсутності необхідних тегів.

AEscapeChroniclesGameMode відповідає за вибір точки появи та рестарт гравців/ботів.

Ключові методи:

1. **ChoosePlayerStart_Implementation**:

- Модифікована версія методу з AGameModeBase, яка враховує AEscapeChroniclesPlayerStart::CanSpawnPawn для фільтрації точок появи.
- Перевіряє, чи точка вільна (UnOccupiedStartPoints) або може бути використана з телепортацією (OccupiedStartPoints).
- Віддає перевагу APlayerStartPIE у режимі Play-In-Editor.

2. **RestartPlayerAtPlayerStart**:

- Скидає позицію та ротацію пешки до значень PlayerStart, навіть якщо пешка вже існує, на відміну від базової реалізації.
- Обробляє створення пешки, якщо вона відсутня, через SpawnDefaultPawnFor.

3. **RestartPlayerAtTransform**:

- Аналогічно скидає позицію та ротацію пешки до заданого FTransform, забезпечуючи гнучкість для сценаріїв без PlayerStart.

Логіка роботи системи

1. Ініціалізація:

- У BeginPlay ігрового режиму викликається UBotSpawnerComponent::SpawnBots.
- Компонент асинхронно завантажує класи персонажів (PrisonerCharacterClass, GuardCharacterClass, MedicCharacterClass) через UAssetManager.

2. Спавн ботів:

- Після завантаження класів викликаються методи On[Type]CharacterClassLoaded, які спавняють потрібну кількість ботів через SpawnBots.
- Кожен бот створюється через SpawnOneBot, де:
 - Спавниться персонаж (AEscapeChroniclesCharacter) із параметром AlwaysSpawn.
 - Викликається SpawnDefaultController для створення контролера (AEscapeChroniclesAIController).
 - Підписка на ReceiveControllerChangedDelegate забезпечує подальшу обробку.

3. Призначення точки появи:

- Після ініціалізації контролера викликається GameMode->RestartPlayer, який використовує ChoosePlayerStart_Implementation для вибору PlayerStart.
- AEscapeChroniclesPlayerStart::CanSpawnPawn перевіряє відповідність тегів бота (заданих через GameplayEffect у UAbilitySystemComponent) з RequiredGameplayTagsToSpawn.

4. Ініціалізація бота:

- Боту призначається унікальне ім'я через SelectBotName.
- GameMode->LoadAndInitBot ініціалізує PlayerState через USaveGameSubsystem, завантажуючи збережені дані або генеруючи унікальний ID.

5. Очищення:

- Асинхронні дескриптори (FStreamableHandle) звільняються після спавну для економії пам'яті.
- Делегати (ReceiveControllerChangedDelegate, OnPossessedPawnChanged) очищаються після використання.

Особливості реалізації

- **Асинхронне завантаження:** Використання `UAssetManager::RequestAsyncLoad` зменшує затримки при завантаженні класів персонажів.
- **Тегована система спавну:** `AEscapeChroniclesPlayerStart` дозволяє точно контролювати, де з'являються боти, залежно від їхньої ролі (наприклад, в'язні спавняються у камерах, охоронці — у зонах патрулювання).
- **Управління іменами:** Система унікальних імен через `UBotNames` і `ClaimedBotNames` забезпечує різноманітність і збереження між сеансами.
- **Обхід помилки движка:** Тимчасове відключення колізії мешу вирішує проблему з вибором `PlayerStart`.
- **Модульність:** Компонентна архітектура дозволяє легко додавати нові типи ботів, змінюючи лише властивості в `UBotSpawnerComponent`.

Система створення ботів забезпечує гнучке та ефективне створення ботів різних типів (в'язні, охоронці, медики) з використанням асинхронного завантаження, тегової системи спавну та інтеграції з `AEscapeChroniclesGameMode`. Компонент `UBotSpawnerComponent` централізує логіку спавну, а `AEscapeChroniclesPlayerStart` забезпечує точне розміщення ботів на основі тегів, заданих через `GameplayEffect`. Обхід помилки колізії та управління іменами додають стабільності та різноманітності. Система є модульною, оптимізованою та готовою до розширення для нових типів ботів чи сценаріїв гри, що сприяє створенню динамічного та занурювального ігрового досвіду.

РОЗДІЛ 17
NAMEPLATES

Система неймплейтів у грі забезпечує відображення імен персонажів (гравців і ботів) над їхніми головами, що полегшує ідентифікацію у багатокористувацькому середовищі. Неймплейти реалізовані через компонент `UNameplateWidgetComponent`, який розміщується на персонажах (`AEscapeChroniclesCharacter`) і завжди орієнтується на камеру локального гравця. Для ботів імена генеруються випадково з асера даних (`UBotNames`), уникаючи повторень, коли це можливо, тоді як імена гравців автоматично беруться з профілю Steam. Цей розділ описує архітектуру, функціональність, інтеграцію та особливості системи неймплейтів.

Система неймплейтів відображає ім'я персонажа через віджет, який є підкласом `UTextContainerWidget`. Компонент `UNameplateWidgetComponent` відповідає за ініціалізацію, оновлення тексту та орієнтацію віджета відносно камери гравця. Неймплейти не відображаються для локально керованих персонажів, щоб уникнути захаращення екрана. Для ботів імена призначаються через `UBotSpawnerComponent`, а для гравців — через `AEscapeChroniclesPlayerState`, який синхронізується з даними Steam.

Компонент `UNameplateWidgetComponent`

`UNameplateWidgetComponent` є похідним від `UWidgetComponent` і відповідає за відображення неймплейта персонажа. Основні елементи:

1. Властивості та ініціалізація:

- `WidgetClass`: Встановлюється як `UTextContainerWidget::StaticClass()` для відображення текстового вмісту.
- `ManuallyRedraw`: Увімкнено для ручного оновлення віджета при зміні тексту.
- `TickMode` та `TickWhenOffscreen`: Налаштовані для автоматичного тикю лише коли віджет видимий, що оптимізує продуктивність.
- `LocalPlayerCameraManager`: Зберігає слабкий вказівник на `APlayerCameraManager` локального гравця для орієнтації віджета.

2. Методи:

- `BeginPlay`: Ініціалізує компонент:
 - Перевіряє, чи персонаж (`AEscapeChroniclesCharacter`) не є локально керованим. Якщо так, неймплейт приховується.
 - Отримує `AEscapeChroniclesPlayerState` і викликає `OnOwningPlayerStateChanged`, якщо `PlayerState` уже доступний.
 - Підписується на делегат `OnPlayerStateChangedDelegate` для обробки ініціалізації або зміни `PlayerState`.
 - Отримує `LocalPlayerCameraManager` через `UGameplayStatics::GetPlayerCameraManager`.
- `OnOwningPlayerStateChanged`: Оновлює неймплейт при зміні `PlayerState`:
 - Відписується від делегата `OnPlayerNameChanged` старого `PlayerState`, якщо він був.
 - Встановлює текст неймплейта через `SetNameplateText` для нового `PlayerState`.
 - Підписується на делегат `OnPlayerNameChanged` нового `PlayerState` для оновлення імені.
- `SetNameplateText`: Оновлює текст віджета (`UTextContainerWidget`) новим ім'ям і викликає `RequestRedraw`.
- `TickComponent`: Оновлює орієнтацію неймплейта:
 - Використовує `UKismetMathLibrary::FindLookAtRotation` для обчислення повороту до камери гравця.
 - Встановлює ротацію компонента, зберігаючи лише yaw і roll для коректного відображення.

Інтеграція з іншими системами

1. `AEscapeChroniclesPlayerState`:

- Неймплейт отримує ім'я через PlayerState->GetPlayerName.
 - Для гравців ім'я синхронізується з профілем Steam через OnlineSubsystemSteam.
 - Делегат OnPlayerNameChanged дозволяє оновлювати неймплейт при зміні імені.
2. **UBotSpawnerComponent:**
 - Для ботів імена призначаються через SelectBotName, який вибирає випадкове ім'я з UBotNamesDataAsset, уникаючи повторень, якщо можливо.
 - Ім'я встановлюється в PlayerState->SetPlayerName під час ініціалізації бота в GameMode->LoadAndInitBot.
 3. **AEscapeChroniclesCharacter:**
 - Компонент розміщується на персонажі та отримує доступ до PlayerState через GetPlayerState.
 - Перевіряє IsLocallyControlled, щоб приховати неймплейт для локального гравця.
 4. **3 камерою гравця:**
 - Використовує LocalPlayerCameraManager для орієнтації неймплейта, забезпечуючи його видимість із будь-якого кута огляду.

Логіка роботи системи

1. **Ініціалізація:**
 - У BeginPlay компонент перевіряє, чи персонаж є локально керованим. Якщо так, неймплейт вимикається.
 - Якщо PlayerState уже доступний, викликається OnOwningPlayerStateChanged для встановлення імені.
 - Підписка на OnPlayerStateChangedDelegate забезпечує обробку ініціалізації PlayerState.
2. **Оновлення імені:**
 - При ініціалізації або зміні PlayerState викликається OnOwningPlayerStateChanged, який встановлює ім'я через SetNameplateText.
 - Зміна імені в PlayerState (наприклад, через Steam або збереження) викликає делегат OnPlayerNameChanged, оновлюючи текст неймплейта.
3. **Орієнтація:**
 - У TickComponent неймплейт обчислює ротацію до камери гравця через FindLookAtRotation.
 - Ротація застосовується лише по осях yaw і roll, щоб уникнути неправильного нахилу.
4. **Оптимізація:**
 - Неймплейт не тикає, якщо невидимий (TickWhenOffscreen=false).
 - Ручне перемальовування (RequestRedraw) викликається лише при зміні тексту.

Особливості реалізації

- **Орієнтація на камеру:** Неймплейт завжди повернутий до камери гравця, що забезпечує читабельність імені з будь-якого ракурсу.
- **Уникнення повторень імен ботів:** Використання ClaimedBotNames у UBotSpawnerComponent мінімізує дублікати імен, якщо є достатньо унікальних варіантів у UBotNamesDataAsset.
- **Автоматичне ім'я гравця:** Інтеграція з OnlineSubsystemSteam забезпечує автоматичне отримання імені з профілю Steam.
- **Оптимізація продуктивності:**
 - Відключення неймплейта для локального гравця зменшує кількість віджетів на екрані.
 - Автоматичний тик і ручне перемальовування знижують обчислювальне навантаження.
- **Гнучкість віджета:** Використання UTextContainerWidget дозволяє легко змінювати стиль неймплейта через Blueprint або дочірні класи.

Система неймплейтів забезпечує чітке та зручне відображення імен персонажів через

UNameplateWidgetComponent. Неймплейти автоматично орієнтуються на камеру гравця, приховуються для локально керованих персонажів і підтримують як імена гравців із Steam, так і випадкові імена ботів із UBotNamesDataAsset. Інтеграція з AEscapeChroniclesPlayerState і UBotSpawnerComponent забезпечує динамічне оновлення імен, а оптимізації, такі як ручне перемальовування та умовний тик, підвищують продуктивність. Система є гнучкою, модульною та сприяє покращенню ігрового досвіду шляхом легкої ідентифікації персонажів у багатокористувацькому середовищі.

РОЗДІЛ 18
ШТУЧНИЙ ІНТЕЛЕКТ

Система штучного інтелекту (ШІ) у грі побудована на основі State Tree, що забезпечує модульне та гнучке керування поведінкою ботів різних типів: в'язнів, охоронців і медиків. Кожен тип бота має власне дерево поведінки, адаптоване до його ролі та ігрових завдань. Система використовує компоненти Unreal Engine, такі як Smart Objects, AI Perception, NavLinkProxies, NavAreas, NavigationQueryFilters і DetourCrowdAIController, а також численні кастомні завдання, оцінники та умови State Tree, які лише частково представлені в документації як приклади. Цей розділ описує загальну архітектуру, функціональність, інтеграцію та ключові особливості системи ШІ, підкреслюючи, що вона значно ширша, ніж представлені приклади коду.

Загальний огляд системи

Система ШІ забезпечує логічну та реалістичну поведінку ботів, які динамічно реагують на ігрові події, взаємодіють із середовищем і виконують рольові завдання. В'язні беруть участь у перекличках, їдять, сплять або намагаються втекти; охоронці патрулюють, реагують на підозрілу поведінку чи діри в стінах; медики лікують непритомних персонажів. Основні принципи системи:

- **State Trees:** Кожен тип бота має унікальне дерево поведінки для виконання специфічних завдань. Надані приклади, такі як `FFindSmartObjectByClassStateTreeTask` або `FGetRandomReachablePointInRadiusStateTreeTask`, є лише частиною широкого набору завдань, які формують поведінку.
- **Smart Objects:** Використовуються для уникнення конфліктів між ботами, забезпечуючи зайняття унікальних місць (ліжка, столи, душі).
- **AI Perception та Stimuli Source:** Дозволяють ботам виявляти гравців, інших ботів і об'єкти, наприклад, діри в стінах, як показано в `FAIPerceptionFunctionLibrary::IsBoxVisible`.
- **NavLinkProxies та NavAreas:** Забезпечують коректне пересування через двері, вузькі проходи та рольові зони.
- **DetourCrowdAIController:** Запобігає скупченню ботів, оптимізуючи навігацію.
- **Кастомні компоненти State Tree:** Завдання, оцінники та умови, такі як `FCharacterStatusStateTreeEvaluator` чи `FDoesAISeeAnyDestructedHoleStateTreeCondition`, є прикладами значно ширшої системи, яка включає десятки подібних компонентів для різних сценаріїв.

Система ШІ інтегрується з іншими підсистемами гри, такими як `AEscapeChroniclesGameMode`, `UAbilitySystemComponent` і `AEscapeChroniclesPlayerState`, для обробки подій, статусів персонажів і навігації. Надані приклади коду ілюструють лише частину функціональності, тоді як повна система охоплює численні додаткові компоненти, які забезпечують комплексну поведінку ботів.

Архітектура системи

1. State Tree та AEscapeChroniclesAIController

`AEscapeChroniclesAIController`, похідний від `ADetourCrowdAIController`, керує поведінкою ботів через `UStateTreeAIComponent`. Основні аспекти:

- **Ініціалізація:**
 - У `PostInitializeComponents` контролер налаштовує `StateTreeAIComponent` для виконання дерева поведінки.
 - Метод `InitPlayerState` створює `AEscapeChroniclesPlayerState` (з перевизначеним класом `PlayerStateClassOverride`) для збереження даних бота, таких як ім'я та унікальний ідентифікатор.
 - Делегат `OnBotInitialized` викликається після ініціалізації `PlayerState` через `GameMode->LoadAndInitBot`.
- **DetourCrowdAIController:**
 - Оптимізує навігацію в натовпі, уникаючи скупчень ботів.
- **State Tree:**
 - Кожне дерево поведінки адаптовано до ролі бота (в'язень, охоронець, медик). Наприклад, `FFindSmartObjectByClassStateTreeTask` ілюструє пошук об'єктів,

але подібних завдань значно більше для інших дій, таких як патрулювання чи взаємодія з об'єктами.

2. Smart Objects

Smart Objects є основою для організації взаємодії ботів із об'єктами та місцями, уникаючи конфліктів. Вони використовуються в багатьох сценаріях, таких як:

- **Переключка, прийом їжі, душ:** В'язні шукають вільний Smart Object (місце в строю, стіл, душ) і виконують дію. Охоронці займають обмежену кількість позицій.
- **Ліжка під час тихої години:** В'язні йдуть до своїх ліжок, визначених через теги Smart Object.
- **Непритомні персонажі:** Smart Object дозволяє лише одному медику взаємодіяти з персонажем.
- **Лікарняні ліжка:** Медики шукають вільне ліжко або чекають, якщо всі зайняті.

Реалізація (на прикладі `FFindSmartObjectByClassStateTreeTask` і `FFindUnoccupiedActivitySpotSmartObjectByClassStateTreeTask`):

- Пошук Smart Objects за класом актора з методами `First`, `Nearest`, `NearestInAcceptableRadius`.
- Фільтрація незайнятих об'єктів, як показано в `FilterFoundActors`.
- Кожен Smart Object має власне дерево поведінки для керування діями бота. Надані завдання є прикладами, а система включає значно більше подібних компонентів для інших сценаріїв.

3. AI Perception та Stimuli Source

Система сприйняття дозволяє ботам виявляти гравців, інших ботів і об'єкти. Приклади реалізації:

- **Оцінник `FAIPerceptionStateTreeEvaluator`:**
 - Відстежує видимі актори (`SeenActors`) через `UAIPerceptionComponent` і викликає делегат `OnSeenActorsChangedDispatcher` при змінах.
- **Умова `FArrayContainsSuspiciousCharactersStateTreeCondition`:**
 - Перевіряє наявність персонажів із атрибутом `Suspicion` вище 90.
- **Умова `FDoesAISeeAnyDestructedHoleStateTreeCondition`:**
 - Виявляє діри в стінах через `FAIPerceptionFunctionLibrary::IsBoxVisible`, який обчислює видимість кутів `BoundingBox` діри.
- Ці компоненти є лише частиною системи сприйняття, яка включає додаткові оцінники та умови для інших подій, таких як виявлення атак чи інших об'єктів.

4. Навігація

Навігація ботів використовує стандартні та кастомні компоненти:

- **`NavLinkProxies`:** Забезпечують проходження через двері та вузькі місця.
- **`NavAreas` та `NavigationQueryFilters`:** Обмежують зони пересування (наприклад, лікарня для медиків, камери для в'язнів).
- **Завдання `FGetRandomReachablePointInRadiusStateTreeTask`:**
 - Шукає випадкову досяжну точку в радіусі з урахуванням `QueryFilterClass`.
 - Має запасні механізми (`bGetRandomPointInNavigableRadiusIfFailed`, `bProjectPointOnNavigationIfFailed`) для роботи за межами навігаційної сітки.
- **`DetourCrowdAIController`:** Уникає скупчень ботів.

Ці компоненти є прикладами ширшої навігаційної системи, яка включає додаткові фільтри та завдання для специфічних сценаріїв.

5. Реакція на статуси персонажів

Система обробляє статуси персонажів, такі як непритомність:

- **Оцінник `FCharacterStatusStateTreeEvaluator`:**
 - Слухає події `OnCharacterFaintedStatusChanged` і надсилає їх до `State Tree`.

- **Завдання FGetNearestFaintedCharacterStateTreeTask:**
 - Знаходить найближчого непритомного персонажа для дій медика.
- Ці компоненти ілюструють лише частину системи обробки статусів, яка включає додаткові оцінники для інших подій.

6. Динамічна поведінка

Боти реагують на ігрові події через State Tree:

- **Бойова система:** Охоронці атакують за високої підозри (Suspicion > 90) або при виявленні атаки.
- **Медики:** Переносять непритомних до лікарняних ліжок через Smart Objects.
- **Реакція на діри:** Охоронці піднімають тривогу при виявленні діри.
- **Idle-режим:** Боти обирають випадкові точки в дозволених зонах через FGetRandomReachablePointInRadiusStateTreeTask.

Ці поведінки є частиною ширшої системи, яка включає додаткові сценарії, наприклад, реакцію на інші типи подій чи взаємодії.

Інтеграція з іншими системами

- **3 AEscapeChroniclesGameMode та UBotSpawnerComponent:**
 - Створення ботів і призначення тегів ролі через GameplayEffect.
- **3 UAbilitySystemComponent:**
 - Управління атрибутами (Suspicion) і статусами (непритомність).
- **3 навігаційною системою:**
 - Інтеграція з NavLinkProxies, NavAreas і NavigationQueryFilters.
- **3 AI Perception:**
 - Використання UAIPerceptionComponent і UAIPerceptionStimuliSourceComponent для сприйняття.

Особливості реалізації

- **Масштабність:** Система включає десятки кастомних завдань, оцінників і умов State Tree, з яких надані приклади є лише частиною.
- **Модульність:** State Tree дозволяє легко додавати нові поведінки.
- **Smart Objects:** Уникають конфліктів і забезпечують логічну взаємодію.
- **Гнучке сприйняття:** Кастомні рішення, як IsBoxVisible, долають обмеження стандартного AI Perception.
- **Оптимізована навігація:** DetourCrowdAIController і NavigationQueryFilters забезпечують плавний рух.
- **Робота зі статусами:** Оцінники та завдання, як FCharacterStatusStateTreeEvaluator, дозволяють реагувати на динамічні події.

Система штучного інтелекту є комплексною та модульною, побудована на основі State Tree із використанням Smart Objects, AI Perception, NavLinkProxies, NavAreas та DetourCrowdAIController. Надані приклади, такі як FFindSmartObjectByClassStateTreeTask чи FAIPerceptionFunctionLibrary::IsBoxVisible, ілюструють лише частину широкої системи, яка включає численні завдання, оцінники та умови для різних сценаріїв. Система забезпечує логічну поведінку ботів, уникає конфліктів через Smart Objects, оптимізує навігацію та реагує на динамічні події, створюючи занурювальний ігровий досвід.

РОЗДІЛ 19
РЕЗУЛЬТАТИ ВИПРОБУВАНЬ

Результатом розробки проєкту *Escape Chronicles* є повноцінна тривимірна кооперативна гра, створена на базі рушія Unreal Engine 5, яка успішно реалізує симуляцію втечі з віртуальної в'язниці з елементами багатокористувацької взаємодії, штучного інтелекту, динамічних ігрових подій і комплексного управління ігровим процесом. Усі системи гри, описані в попередніх розділах, були ретельно протестовані для забезпечення їхньої стабільності, відповідності технічному завданню (ТЗ) і здатності створювати занурювальний ігровий досвід. Тестування проводилося в редакторі Unreal Engine, у режимі Play-In-Editor (PIE), а також у скомпільованій версії гри на платформі Windows із підключенням до Steam для перевірки мережевих функцій. Результати випробувань підтвердили працездатність усіх ключових компонентів і їхню інтеграцію в єдину систему.

Процес тестування

Тестування охоплювало всі основні підсистеми гри, включаючи ігровий процес, штучний інтелект, мережеву взаємодію, збереження даних, інтерфейс користувача, аудіосупровід і управління подіями. Основні методи тестування включали:

- **Функціональне тестування:** Перевірка відповідності кожної системи вимогам, зазначеним у ТЗ, шляхом виконання сценаріїв, що охоплюють основні механіки гри.
- **Інтеграційне тестування:** Перевірка коректної взаємодії між підсистемами, наприклад, між системою штучного інтелекту та Gameplay Ability System.
- **Мережеве тестування:** Перевірка стабільності багатокористувацького режиму через Steam P2P із підключенням до 4 гравців, включаючи створення сесій, приєднання, перехід між рівнями та обробку відключень.
- **Тестування збереження:** Перевірка коректності автоматичного збереження та завантаження ігрового стану, включаючи інвентар, позиції персонажів і стан об'єктів.
- **Продуктивність:** Оцінка оптимізації гри в різних сценаріях, зокрема при великій кількості ботів і динамічних об'єктів.
- **Тестування UI:** Перевірка функціональності головного меню, меню налаштувань і головного інтерфейсу гравця.

Для виявлення помилок використовувалися макроси Unreal Engine (check, checkf, ensure, ensureAlways), які дозволяли фіксувати критичні проблеми під час виконання в режимі розробника. Додатково проводилося ручне тестування для перевірки ігрового досвіду, зокрема логічності поведінки ботів, зручності інтерфейсу та відповідності аудіосупроводу ігровим подіям.

Результати тестування ключових систем

1. **Загальна архітектура та ігровий процес (Розділ 1):**
 - Успішно реалізовано модульну архітектуру з використанням Gameplay Framework, Subsystems, State Trees і Smart Objects. Тестування підтвердило стабільність і масштабованість системи, а також коректну роботу реплікації в багатокористувацькому режимі.
 - Асинхронне завантаження ресурсів через StreamableManager забезпечило відсутність затримок під час гри.
2. **Gameplay Ability System та інпут (Розділ 2):**
 - Система здібностей на базі GAS продемонструвала гнучкість у призначенні здібностей і ефектів різним ролям персонажів. Тестування підтвердило коректну прив'язку інпутів через UInputConfig і стабільну роботу клієнтських передбачень.
 - Централізована конфігурація через UAbilitySystemSet спростила додавання нових здібностей.
3. **Характеристики персонажів і ролі (Розділ 3):**
 - Атрибути (Health, Energy, Suspicion) коректно реплікуються та обмежуються через ClampAttribute. Тестування показало правильну роботу тегів ролей (Role.Prisoner, Role.Guard, Role.Medic) для фільтрації поведінки.
4. **Gameplay Ability Components (Розділ 4):**
 - Модульні компоненти здібностей (UGameplayAbilityComponent) успішно зменшили дублювання коду та спростили тестування. Наприклад, URequireEnergyGameplayAbilityComponent коректно обробляє перевірку

енергії.

5. **Пересування персонажа** (Розділ 5):
 - Система пересування на базі плагіна Mover забезпечила плавне перемикання між режимами (Walking, Jogging, Running, Crouching). Тестування підтвердило синхронізацію з GAS через теги стану.
6. **Система збережень та ідентифікація гравців** (Розділ 6):
 - Автоматичне збереження кожні кілька секунд і при виході з сесії працювало без помилок. Тестування підтвердило коректне відновлення стану гри, включаючи інвентар, позиції ботів і стан об'єктів.
 - Унікальні ідентифікатори (FUniquePlayerID) забезпечували точну ідентифікацію гравців і ботів.
7. **Система власників акторів** (Розділ 7):
 - Прив'язка об'єктів (ліжка, тумби) до гравців і ботів через UPlayerOwnershipComponent працювала стабільно. Тестування показало коректне збереження власності між сесіями.
8. **Інвентар та предмети** (Розділ 8):
 - Система інвентаря підтримувала крафт, використання предметів, класифікацію (наприклад, контрабанда) і взаємодію з об'єктами (металодетектори, ляльки). Тестування підтвердило втрату контрабанди при непритомності та коректне збереження предметів на рівні.
9. **Пошкодження об'єктів** (Розділ 9):
 - Система пошкодження через UDestroyableComponent забезпечувала створення отворів у стінах із синхронізацією в мережі та збереженням. Тестування показало коректне виявлення дірок охоронцями через AI Perception.
10. **Система подій та розкладу** (Розділи 10–11):
 - Події ("Перекличка", "Тиха година", "Прийом їжі", "Тривога") коректно впливали на поведінку ботів і стан гри. Тестування підтвердило синхронізацію часу та подій у мультиплеєрі.
11. **Бойова система та здоров'я** (Розділ 12):
 - Бойові механіки (атака, блок) і система здоров'я працювали стабільно, із коректною реакцією медиків на непритомних персонажів.
12. **Музичний супровід подій** (Розділ 13):
 - Аудіосистема забезпечувала плавні переходи між композиціями залежно від подій. Асинхронне завантаження аудіоресурсів не викликало затримок.
13. **Клієнт-серверне підключення** (Розділ 14):
 - Мережевий режим через Steam P2P підтримував до 4 гравців без значних затримок. Тестування показало стабільне створення сесій, приєднання клієнтів і обробку відключень.
14. **Користувацький інтерфейс** (Розділ 15):
 - Головне меню, меню налаштувань і головний інтерфейс гравця функціонували коректно. Тестування підтвердило зручність налаштування звуку та графіки, а також динамічне оновлення віджетів (час, події, атрибути).
15. **Створення ботів** (Розділ 16):
 - Система спавну ботів через UBotSpawnerComponent забезпечувала коректне розміщення в'язнів, охоронців і медиків на основі тегів. Унікальні імена ботів зберігалися між сесіями.
16. **Неймплейти** (Розділ 17):
 - Неймплейти відображали імена гравців (із Steam) і ботів, орієнтуючись на камеру. Тестування показало оптимізовану продуктивність завдяки умовному тику.
17. **Штучний інтелект** (Розділ 18):
 - Система ШІ на базі State Trees забезпечувала логічну поведінку ботів (патрулювання, реакція на діри, лікування). Тестування підтвердило уникнення конфліктів через Smart Objects і оптимізовану навігацію через DetourCrowdAIController.

Результати випробувань підтвердили, що гра *Escape Chronicles* відповідає всім вимогам, зазначеним у технічному завданні. Усі підсистеми, від ігрового процесу до мережевої взаємодії та штучного інтелекту, продемонстрували стабільну роботу та високу якість реалізації. Модульна архітектура, використання сучасних інструментів Unreal Engine 5 (Gameplay Ability System, State Trees, Mover, Smart Objects) і ретельна оптимізація (асинхронне завантаження, реплікація, кешування) забезпечили високу продуктивність і масштабованість проєкту.

Гра створює занурювальний ігровий досвід завдяки динамічним подіям, логічній поведінці ботів, зручному інтерфейсу та атмосферному аудіосупроводу. Багатокористувацький режим через Steam P2P дозволяє гравцям легко взаємодіяти, а система збережень забезпечує безперервність ігрового прогресу. Успішне тестування всіх компонентів підтверджує готовність проєкту до використання та потенціал для подальшого розвитку, наприклад, додавання нових рівнів, ролей чи механік.

Для ознайомлення з вихідним кодом і повною документацією проєкту можна звернутися до репозиторію: <https://github.com/Driver2579/Escape-Chronicles>.

ВИСНОВКИ

Висновки

На основі технічного завдання було розроблено проєкт *Escape Chronicles* — тривимірну кооперативну гру, яка повністю відповідає усім зазначеним вимогам. У процесі створення проєкту було проведено детальний аналіз предметної галузі ігрових симуляцій, досліджено сучасні тенденції в розробці багатокористувацьких ігор і складено план реалізації системи. Результатом стала інтерактивна гра, що забезпечує занурювальний ігровий досвід завдяки поєднанню динамічних механік, логічної поведінки штучного інтелекту, зручного інтерфейсу та атмосферного аудіосупроводу.

Система була спроектована з урахуванням сучасних принципів розробки програмного забезпечення, що забезпечило її модульність, масштабованість і зручність підтримки. Для реалізації використано ігровий рушій Unreal Engine 5 із застосуванням таких ключових технологій, як Gameplay Ability System, State Trees, Smart Objects, Mover, а також інтеграція з мережевою платформою Steam для багатокористувацького режиму. Архітектура гри складається з кількох взаємопов'язаних компонентів:

- **Ігровий процес:** Реалізовано симуляцію втечі з в'язниці з підтримкою кооперативної взаємодії, системою подій, інвентарем, пошкодженням об'єктів і бойовими механіками.
- **Штучний інтелект:** Забезпечено логічну поведінку ботів (в'язнів, охоронців, медиків) через State Trees і Smart Objects, що дозволяє уникати конфліктів і створювати реалістичне ігрове середовище.
- **Мережева взаємодія:** Інтегровано стабільний багатокористувацький режим через Steam P2P із підтримкою до 4 гравців.
- **Користувацький інтерфейс і аудіосупровід:** Розроблено інтуїтивно зрозумілі віджети для головного меню, налаштувань і відображення ігрової інформації, а також динамічний музичний супровід, що реагує на ігрові події.
- **Система збережень:** Забезпечено надійне збереження ігрового стану, включаючи інвентар, позиції персонажів і стан об'єктів.

Гра була успішно протестована в редакторі Unreal Engine, у режимі Play-In-Editor, а також у скомпільованій версії на платформі Windows із підключенням до Steam. Результати тестування підтвердили працездатність усіх систем, їхню інтеграцію та відповідність вимогам ТЗ. Система продемонструвала стабільність у багатокористувацькому режимі, високу продуктивність завдяки асинхронному завантаженню ресурсів і оптимізованій навігації ботів, а також зручність для користувачів завдяки інтуїтивному інтерфейсу.

Розроблений проєкт має значний потенціал для подальшого розвитку, зокрема додавання нових рівнів, типів персонажів, механік або інтеграції з іншими платформами. Успішна реалізація *Escape Chronicles* підтверджує можливість створення складних ігрових систем, які відповідають сучасним стандартам індустрії відеоігор і забезпечують якісний ігровий досвід для користувачів.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

- Source code Unreal Engine 5
- <https://dev.epicgames.com/>
- <https://forums.unrealengine.com/>
- Lyra Starter Game
- <https://www.reddit.com/>
- <https://chatgpt.com/>
- <https://grok.com/>
- GitHub Copilot
- <https://www.youtube.com/>