



WYŻSZA SZKOŁA EKONOMII I INFORMATYKI

KATEDRA INFORMATYKI

The Cultist

Dokumentacja techniczna

Imię i nazwisko:

Jakub MAGIERA

10 lutego 2024

Spis treści

1	Wstęp	6
2	Instalacja Gry	6
3	Część 1 - Oprogramowanie	6
3.1	Silnik Gry - Unity 2021.3.12f1	6
3.1.1	Wykorzystane narzędzia:	7
3.2	Środowisko Programistyczne - JetBrains Rider 2022.2.4	7
3.3	Modele 3D - Blender 3.6.5	7
3.4	Teksturowanie - Blender 3.6.5, Substance Painter 8.3.1	7
3.5	Tworzenie Tekstur - Substance Designer 13.0.0	8
3.6	Projektowanie UI - Figma	8
3.7	Logika Dialogów - Ink 0.14.1	8
3.8	Grafika 2D - Photoshop 2019, Illustrator 2019	8
3.8.1	Adobe Illustrator 2019	8
3.8.2	Adobe Photoshop 2019	8
3.9	Pomoc ze skryptami - ChatGPT-3.5	9
4	Część 2 - Dodatkowe zależności	9
4.1	DOTween	9
4.2	TextMeshPro	10
4.3	Unity Input System	10
5	Część 3 - Architektura	11
5.1	Diagram zależności	11
5.1.1	Silnik Gry (Unity)	12
5.1.2	Systemy Gry	12
5.1.3	Interfejs Użytkownika (UI)	12
5.1.4	System Dialogów (INK)	13
5.2	Quest System	13
5.2.1	Campaign	13
5.2.2	Questline	15
5.2.3	Quest	16
5.2.4	Decyzje gracza	19
5.2.5	Nagrody	19
5.3	Narrative Event	20
5.4	Dialogue System	21

5.5	Interactive Objects	22
6	Część 4 - Algorytmy	23
6.1	Lista Skryptów	23
6.1.1	Game Manager	23
6.1.2	Location Manager	27
6.1.3	Quest Manager	30
6.1.4	Narrative Event Handler	31
6.1.5	Test Player	34
6.1.6	Player Data	36
6.1.7	Player Events	36
6.2	Lista Systemów	37
6.2.1	System Dialogów	37
6.2.2	System Questów	42

Spis rysunków

1	Diagram zależności między modułami	11
2	Konfiguracja Kampanii	14
3	Konfiguracja Questline'a	16
4	Konfiguracja Zadania	18
5	Wygląd Narrative Event'u	21
6	Przykładowy Container	22
7	Przykładowy Travel Point	23

Spis fragmentów kodu

1	Pauzowanie rozgrywki	24
2	Zarządzanie ekwipunkiem gracza	25
3	Obliczanie nowej wartości dla statystyki gracza	27
4	Wyszukiwanie domyślnego spawn pointa	28
5	Dwie opcje ładowania sceny	29
6	Quest Manager	30
7	Wyszukiwanie warunków narrative event'u	31
8	Wyszukiwanie komend narrative event'u	32
9	Ładowanie sceny z komendy narrative eventu	33
10	Dodawanie i usuwanie przedmiotu z komendy narrative eventu	33
11	Kontrolowanie statusu zadań z komendy narrative eventu	34
12	Testowanie statystyki gracza	34
13	Zmiana wartości PlayerData	36
14	Przykładowe eventy związane z postacią gracza	37
15	Podstawowa klasa dialogu	37
16	Klasa DialogController - inicjalizacja i kontrola dialogu	38
17	Klasa DialogController - opcje dialogowe	39
18	Klasa DialogController - zarządzanie zadaniami z dialogu	39
19	Klasa DialogController - ładowanie zmiennych z zadań do dialogu	40
20	Rozpoczęcie kampanii	42
21	Ukończenie kampanii	43
22	Rozpoczęcie questline'u	43
23	Ukończenie questline'u	44
24	Sprawdzenie statusu zawartych w questline'ie zadań	44
25	Rozpoczęcie zadania	45
26	Zakończenie zadania	45
27	Zadanie nie zostało wykonane pomyślnie	46
28	Przyznanie nagrody za wykonanie zadania	47

1 Wstęp

Witaj w dokumentacji technicznej gry "The Cultist" – fascynującej produkcji, która przeniesie Cię w mroczny świat sekretów, magii i niebezpiecznych tajemnic. To RPG, które zapewni Ci unikalne doświadczenia w pełnym napięcia świecie, gdzie przyjdzie graczowi zdecydować o losie wielu istnień

Cel i Wizja Gry:

The Cultist przenosi gracza w rolę nieustraszonego badacza tajemnic, który staje do walki z kultem czczącym starożytne moce. Gracz będzie eksplorować mroczne zakamarki świata, podejmować decyzje wpływające na fabułę, rozwijać postać, często korzystając z demonicznych mocy, i stawiać czoła potężnym wrogom. Celem jest stworzenie wciągającego świata, który łączy w sobie głęboką narrację z dynamicznym światem, reagującym na poczynania gracza.

Technologia i Platformy:

The Cultist zostało stworzone w oparciu o silnik gry Unity. Docelowo produkcja ma trafić na platformę Steam i być dostępna wyłącznie na komputerach osobistych.

2 Instalacja Gry

The Cultist nie posiada własnego instalatora. Aby uruchomić grę, należy pobrać folder z plikami gry i kliknąć na ikonę gry dwukrotnie.

3 Część 1 - Oprogramowanie

3.1 Silnik Gry - Unity 2021.3.12f1

The Cultist oparto o silnik Unity, gdzie skorzystano ze standardowego projektu URP.

3.1.1 Wykorzystane narzędzia:

1. Analytics - paczka stworzona przez Unity, która służy do zbierania danych gracza i przetwarzania ich zgodnie z potrzebą twórcy gry.
2. JetBrains Rider Editor - paczka pozwala zintegrować program JetBrains Rider z silnikiem Unity.
3. ProBuilder - szybkie i przystępne narzędzie do prototypowania poziomów.
4. Ink Integration for Unity - pozwala zintegrować program Ink, w którym tworzone są dialogi, z silnikiem gry oraz edytorem kodu.

Aby zainstalować paczki do projektu, należy w silniku otworzyć Package Manager i wybrać potrzebne narzędzia, a następnie dokonać instalacji. Jeśli narzędzie nie widnieje w spisie, należy pobrać go ze sklepu z assetami Unity.

3.2 Środowisko Programistyczne - JetBrains Rider 2022.2.4

W projekcie wykorzystano środowisko programistyczne Rider¹. Oprogramowanie zakupiono z wykorzystaniem licencji studenckiej, która uprawnia użytkownika do darmowego użytkowania programu na okres roku.

3.3 Modele 3D - Blender 3.6.5

Blender, to bezpłatne i otwarte oprogramowanie do modelowania 3D, charakteryzujące się wszechstronnością i potężnymi funkcjami. Dzięki temu narzędziu, możliwe było precyzyjne modelowanie oraz animowanie trójwymiarowych obiektów, a także renderowanie ich w różnych warunkach oświetleniowych.

3.4 Teksturowanie - Blender 3.6.5, Substance Painter 8.3.1

Adobe Substance Painter to narzędzie dedykowane malowaniu tekstur bezpośrednio na modelach 3D. Jego interaktywny interfejs zapewnił mi swobodę w tworzeniu realistycznych tekstur, nadając moim modelom autentyczny wygląd.

¹JetBrains Rider - <https://www.jetbrains.com/rider/features/>

3.5 Tworzenie Tekstur - Substance Designer 13.0.0

Adobe Substance Designer to zaawansowane narzędzie do projektowania tekstur proceduralnych. Jego wizualny interfejs umożliwia projektowanie skomplikowanych tekstur, co było kluczowe dla uzyskania realistycznego wyglądu modeli 3D.

3.6 Projektowanie UI - Figma

Figma stała się podstawowym narzędziem do projektowania interfejsu użytkownika w czasie rzeczywistym. Figma umożliwiła mi również łatwe tworzenie interaktywnych prototypów, co było niezwykle przydatne podczas testowania i iteracji projektu. Użytkownicy mieli szansę ocenić funkcjonalność interfejsu przed pełną implementacją w grze.

3.7 Logika Dialogów - Ink 0.14.1

Program Inky jest dedykowanym oprogramowaniem do pisania dialogów, historii lub opisu scen przedstawionych w grze. Wykorzystuje on stworzony przez autorów język programowania Ink.

3.8 Grafika 2D - Photoshop 2019, Illustrator 2019

3.8.1 Adobe Illustrator 2019

Adobe Illustrator był jednym z głównych narzędzi, które wykorzystałem do projektowania elementów UI. Program ten jest znany z doskonałych możliwości rysowania wektorowego, co pozwala na precyzyjne tworzenie ikon, przycisków i innych elementów interfejsu. Dzięki funkcji warstw, mogłem łatwo organizować projekty i dostosowywać elementy graficzne zgodnie z potrzebami gry.

3.8.2 Adobe Photoshop 2019

Photoshop stanowił ważne uzupełnienie w procesie projektowania UI, szczególnie jeśli chodzi o prace związane z teksturami, cieniowaniem i efektami specjalnymi. Tworzyłem w nim również koncepcje na wygląd elementów UI.

3.9 Pomoc ze skryptami - ChatGPT-3.5

ChatGPT-3.5 był wykorzystywany do pomocy przy programowaniu oraz projektowaniu najważniejszych elementów. Przy pomocy AI mogłem rozwiązać wiele problemów z kodem.

4 Część 2 - Dodatkowe zależności

4.1 DOTween

DOTween to popularna paczka do animacji i tweenu w środowisku Unity, oferująca wydajne, łatwe w użyciu narzędzia do animowania obiektów i wartości. Zaprojektowana z myślą o prostocie i wydajności, DOTween umożliwia płynne i zaawansowane animacje, co sprawia, że jest często wybierana przez deweloperów do projektów 2D i 3D.

Główne cechy paczki DOTween:

1. **Prostota Użycia:** DOTween oferuje intuicyjne API, co czyni tworzenie animacji prostym i zrozumiałym nawet dla początkujących programistów.
2. **Wszechstronność:** Możliwość animowania różnych typów wartości, takich jak pozycje, rotacje, skale, kolory, a także właściwości dźwięku czy materiałów.
3. **Zaawansowane Easingi:** Dostęp do szerokiej gamy funkcji Easing, pozwalających na łatwe dodawanie efektów płynnych przejść i animacji.
4. **Sekwencje i Pętle:** Tworzenie sekwencji animacji, umożliwiających skomplikowane zestawienia ruchów i efektów. Obsługa pętli, co pozwala na powtarzanie animacji z łatwością.
5. **Obsługa Działań Przy Zakończeniu Animacji:** Dodawanie działań do wykonania po zakończeniu animacji, co umożliwia płynne zarządzanie logiką gry w trakcie animacji.
6. **Wsparcie dla Unity Events:** Integruje się z Unity Events, co ułatwia reakcję na zdarzenia w trakcie animacji.

Instalacja:

DOTween dostępne jest w Unity Asset Store. Po pobraniu, paczkę można zainstalować za pomocą menedżera paczek Unity.

4.2 TextMeshPro

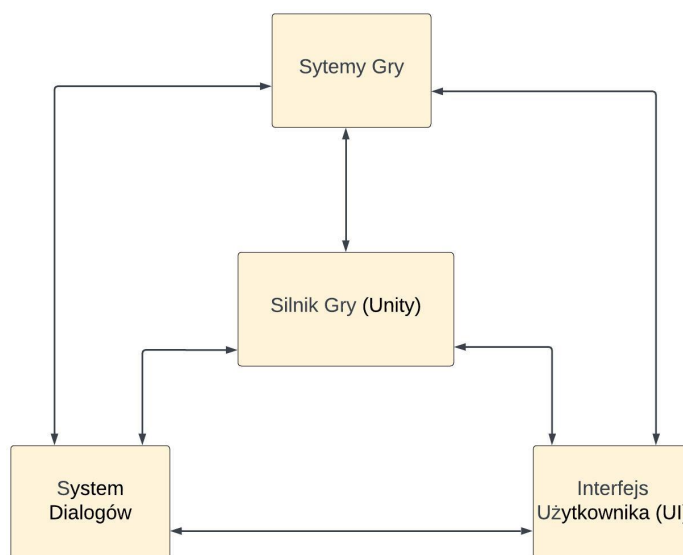
Narzędzie TextMeshPro jest potężnym narzędziem, wykorzystywanym do tworzenia UI gry. Potężny i łatwy w użyciu TextMeshPro (znany również jako TMP) wykorzystuje zaawansowane techniki renderowania tekstu wraz z zestawem niestandardowych shaderów; zapewniając znaczną poprawę jakości wizualnej, jednocześnie dając użytkownikom niesamowitą elastyczność, jeśli chodzi o stylizację i tekstutowanie tekstu.

4.3 Unity Input System

Input System to nowszy, bardziej elastyczny system, który umożliwia korzystanie z dowolnego urządzenia wejściowego do sterowania zawartością Unity. Ma on zastąpić klasyczny menedżer wprowadzania danych Unity. Jest on określany jako "The Input System Package" lub po prostu "The Input System". Aby z niego skorzystać, należy zainstalować go w projekcie za pomocą Menedżera pakietów.

5 Część 3 - Architektura

5.1 Diagram zależności



Rysunek 1: Diagram zależności między modułami

Powyższy diagram przedstawia zależności między poszczególnymi modułami, z których składa się projekt.

5.1.1 Silnik Gry (Unity)

- **Relacja z Systemami Gry:** Silnik zarządza podstawowymi funkcjonalnościami, takimi jak renderowanie grafiki, obsługa dźwięku, fizyki. Systemy gry korzystają z funkcji silnika do realizacji swoich zadań.
- **Relacja z Interfejsem Użytkownika (UI):** Silnik wspiera interfejs użytkownika poprzez dostarczenie narzędzi do projektowania i renderowania elementów UI. Komponenty interfejsu są integrowane z silnikiem w celu efektywnego zarządzania interakcjami z graczem.
- **Relacja z Systemem Dialogów:** Silnik umożliwia obsługę dialogów poprzez dostarczenie narzędzi do zarządzania plikami tekstowymi, kompilację plików Ink, by zarządzać przepływem dialogu i reagować na zmiany oraz decyzje gracza.
- **Relacja z Systemem Fizyki:** Silnik obsługuje silnik fizyki, który jest używany przez systemy gry do systemu nawigacji postaci, interakcji z otoczeniem oraz zjawisk fizycznych w grze.

5.1.2 Systemy Gry

- **Relacja z Silnikiem Gry:** Systemy gry korzystają z funkcji silnika do renderowania grafiki, obsługi dźwięku, zarządzania fizyką i innych podstawowych aspektów gry.
- **Relacja z Interfejsem Użytkownika (UI):** Systemy gry komunikują się z interfejsem użytkownika, informując go o zmianach w stanie gry, prezentując informacje o zdrowiu postaci, aktualnych zadaniach, itp.
- **Relacja z Systemem Dialogów:** Systemy gry korzystają z systemu dialogów do uruchamiania dialogów z postaciami niezależnymi, dostarczania informacji fabularnych oraz aktywacji zadań.

5.1.3 Interfejs Użytkownika (UI)

- **Relacja z Silnikiem Gry:** Interfejs użytkownika jest renderowany przez silnik gry. Elementy UI są zintegrowane z silnikiem w celu obsługi interakcji z graczem.
- **Relacja z Systemami Gry:** Elementy UI reagują na zmiany w stanie gry, takie jak zdrowie postaci, poziom doświadczenia, dostępność zadań. Komunikują się z systemami gry, aby uzyskać aktualne dane.

- **Relacja z Systemem Dialogów:** Interfejs użytkownika może wyświetlać okna dialogowe, wybory dialogowe, ikony postaci i inne elementy związane z dialogami, współpracując z systemem dialogów.

5.1.4 System Dialogów (INK)

- **Relacja z Silnikiem Gry:** System dialogów wykorzystuje funkcje silnika do renderowania postaci, obsługi animacji i efektów dźwiękowych podczas dialogów.
- **Relacja z Systemami Gry:** System dialogów integruje się z systemami gry w celu aktywacji zadań, zmiany stanu postaci, a także uzyskiwania informacji o postępach w grze.
- **Relacja z Interfejsem Użytkownika (UI):** System dialogów komunikuje się z interfejsem użytkownika, aby wyświetlać informacje tekstowe, opcje dialogowe i inne elementy związane z dialogami.





5.2 Quest System

System zadań został podzielony na trzy części. Największą jest kampania, która scala ze sobą mniejsze fabuły w całość. Drugim jest linia zadań, czyli questline, w którym umieszczono kolejne, najmniejsze składniki historii. Pojedyncze zadanie składa się z wyraźnej instrukcji, zrozumiałej dla gracza.

5.2.1 Campaign

Kampania jest największą składową fabułą. Jest pojemnikiem na pomniejsze fabuły, powiązane ze sobą i przeplatającymi się nieustannie. Kampania ma swój klimat, założenia i cel. Można myśleć o jej strukturze jak o książce. Pojedyncza kampania jest całą opowieścią, na którą składają się rozdziały w formie linii zadań.

Wszystkie kampanie muszą mieć unikalną nazwę oraz identyfikator, dzięki którym można je będzie rozróżnić. Gracz w trakcie wyboru kampanii widzi jej opis, który zapewnia początkowy wgląd w tło fabularne, klimat i tematykę przygody.


Port Investigation (Campaign)




Open

Script
Campaign

Campaign Id
1

Campaign Name
Shadows in the lower city

Campaign Image
Background1

Campaign Desc

First Scene To Load
Small_port (9)

Starting Event
PortInvestigationNarrative

▼ Campaign Questlines
1

= Element 0
InvestigateLowerCityPort
+ -

▼ Required Questlines
1

= Element 0
InvestigateLowerCityPort
+ -

▼ Required Campaigns
0

List is Empty
+ -

Has Started
☒

Is Completed
☐

▼ Campaign Rewards
3

=▼ Element 0

Type
Give Stat Level

Stat
Forensics

Increase Va
1

Rysunek 2: Konfiguracja Kampanii

Struktura fabuły wymaga czasem ograniczeń. By tworzyła ona spójną całość, kampania może mieć wymagania dotyczące ukończonych przygód oraz questline'ów, które musi ukończyć gracz, by wybrana kampania mogła zostać zamknięta.

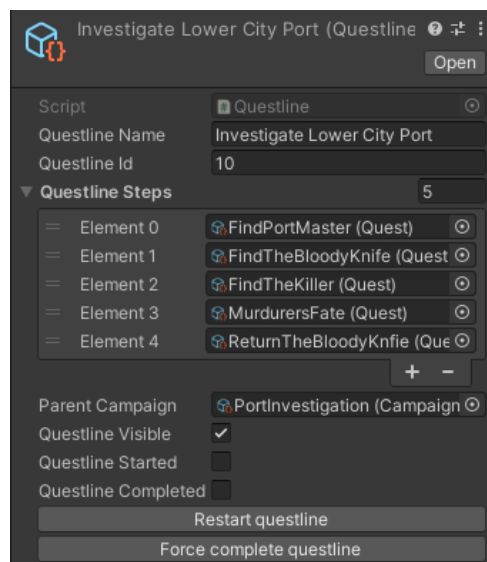
Innym elementem ważnym z perspektywy gracza są nagrody. Ukończenie kampanii gwarantuje wynagrodzenie za czas i przejście przygody.

5.2.2 Questline

Linia questów pozwala łączyć mniejsze questy w ciąg opowiadający krótki fragment historii zawartej w kampanii. Inną nazwą może być łańcuch zadań (z ang. Quest chain).

Questline można traktować jak rozdział książki. Tyczy się pojedynczego wątku i jest wstępem do kolejnych. W The Cultist część questline'ów musi zostać ukończona, by zamknąć kampanię i przejść do kolejnych. Inne, nazywane pobocznymi, są opcjonalne i tylko do gracza należy decyzja, czy je wykonać.

Możliwości linii zadań są ograniczone, gdyż mają spajać ze sobą mniejsze zadania. Dlatego za jej ukończenie nie przewidziano nagród.



Rysunek 3: Konfiguracja Questline'a

5.2.3 Quest

Najmniejsza składowa fabuły. Zadanie można traktować jak proste polecenie: idź, przynieś, porozmawiaj. Instrukcja ma być prosta i zrozumiała, by gracz nie miał problemu ze znalezieniem rozwiązania.

Questy, choć najmniejsze, determinują przebieg gry oraz historii. Jako jedyna składowa mogą zostać przerwane i oznaczone jako failed (z ang. Nieudany).

Jako instrukcja i kierunek dalszych poczynań gracza, questy są wyświetlane na ekranie gry w specjalnym oknie, gdzie gracz ma szybki podgląd na aktualnie wykonywane zadanie. Jeśli gracz otworzy questlog (z ang. Dziennik zadań), wyświetlone zostają wszystkie questy w linii zadań, zarówno te wykonywane jak i zakończone. Dzięki temu użytkownik ma pełniejszy wgląd w aktualną historię i może przypomnieć sobie zakończone zadania.

Pojedyncze zadanie składa się z czterech obiektów typu string: krótkiego opisu, domyślnego, opisu po zaliczeniu zadania oraz gdy graczowi nie uda się go ukończyć.

Ten pierwszy wyświetla się na ekranie jako polecenie w najprostszej postaci.

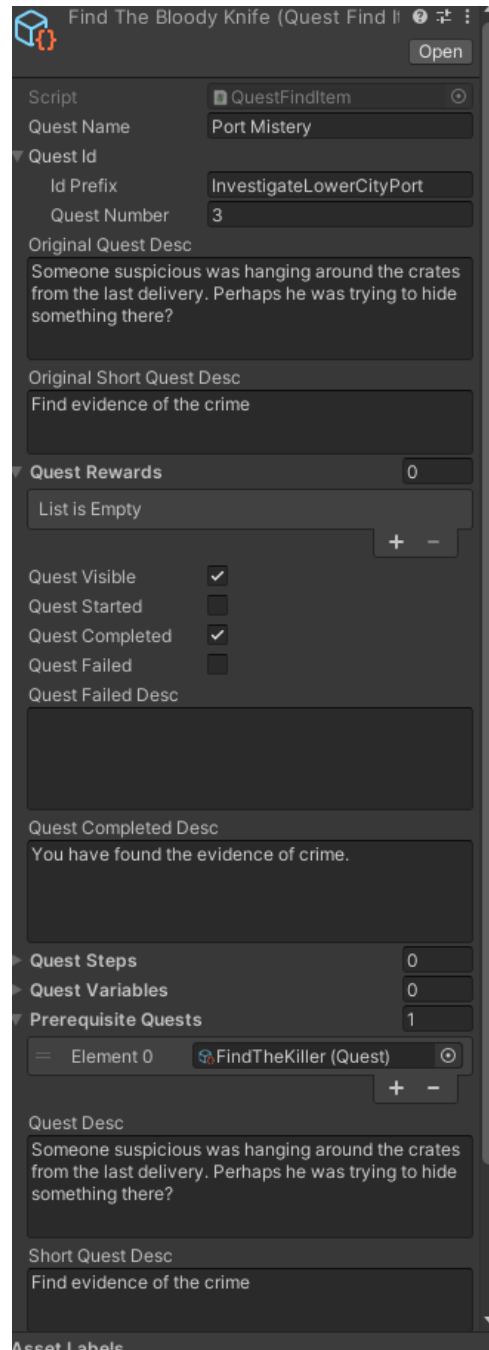
Drugi znajduje się w dzienniku zadań i może zawierać więcej szczegółów fabularnych, jak imię osoby zlecającej graczowi zadanie, czy jego przemyślenia.

Trzeci string zastępuje domyślny, gdy gracz ukończy zadanie. W ten sposób, gdy zobaczy zadanie w dzienniku, od razu będzie wiedział, że zostało ono ukończone w sposób pomyślny.

Ostatni string również zastępuje domyślny opis, ale tylko w przypadku, gdy graczowi nie uda się ukończyć zadania i zostanie ono oznaczone jako failed.

Ukończenie quest'a może dać graczowi nagrody, które zostały przypisane do zadania. Są one jednak znacząco mniejsze od tych na koniec kampanii.

Na całość zadania składa się jeszcze jeden typ obiektu. Quest variables (z ang. Zmienne zadań) są niewielkimi obiektami danych, które przechowują informacje na temat decyzji gracza podjętych w czasie gry. Służą one do zmieniania stanu świata w zależności od kroków gracza. Przykładowo: zmienna, która przechowuje informacje o tym, czy gracz spotkał daną postać na wcześniejszym etapie gry, może zostać wykorzystana przy kolejnym spotkaniu. Jeśli przechowuje ona wartość true, NPC przywita gracza, znając go z wcześniejszych etapów opowieści. Niewielka zmiana pozwala na pokazanie użytkownikowi jego wkładu w grę oraz tego, że nawet najmniejsza jego decyzja ma znaczenie.



Rysunek 4: Konfiguracja Zadania

5.2.4 Decyzje gracza

Decyzje gracza są kluczowe zarówno w grach z gatunku RPG, jak i wielu innych gatunkach, które coraz częściej są połączeniem RPG oraz innych gatunków. Gdy gracz widzi, że jego akcja i wcześniejsza rozgrywka mają odzwierciedlenie na kolejnych etapach gry, będzie bardziej zaangażowany i ma szansę na zbudowanie silniejszej więzi z produktem.

W projekcie rolę decyzji spełniają głównie zmienne zadań, które przechowują kluczowe informacje o sposobie, w jaki gracz ukończył zadanie.

Pod koniec kampanii na ekranie podsumowania wyświetlone zostają wszystkie linie zadań wraz z dostępnymi w nich decyzjami oraz ich opisem. Gracz może przejrzeć, co udało, a co nie udało się osiągnąć i otrzymać informacje o potencjalnych konsekwencjach swojej decyzji.

Decyzji nie można zmienić. Jest to element, który zapewnia regrywalność i poddaje użytkownikowi pod zastanowienie, co stałoby się, gdyby zagrał inaczej.

5.2.5 Nagrody

Motywacja gracza skłania go do częstszego i dłuższego posiedzenia przy grze. Nie dając graczowi nic, prócz produktu, przy którym spędza długie godziny, szybko utracimy odbiorcę.

System nagród jest metodą na stymulację motywacji gracza i wciągnięcie go w rozgrywkę. Ma on wiele wspólnego z systemem decyzji, gdyż różne wybory mogą nagrodzić gracza zupełnie innym przedmiotem, umiejętnością lub informacją.

W The Cultist nagrody podzielone są na dwa typy:

- Grand award
- Quest award

Pierwszy typ nagród gracz otrzymuje tylko przy zakończeniu kampanii. Ma być to wynagrodzenie współmierne z ukończeniem dużego rozdziału historii, więc i nagroda musi być większa. Odblokowanie nowych kampanii, lokacji w świecie gry, niepowtarzalne przedmioty czy zdolności, których gracz nie może zdobyć w żaden inny sposób.

Drugi typ nagród to mające mały wpływ na aktualną rozgrywkę bonusy. Może to być niewielka suma pieniędzy do wydania w sklepie na nowe wyposażenie postaci lub kilka

punktów umiejętności, by lepiej radzić sobie z wyzwaniami. Ten rodzaj nagród ma zapobiec szybkiej utracie motywacji do gry i sprawić, by gracz dotarł do głównej wygranej – końca kampanii.

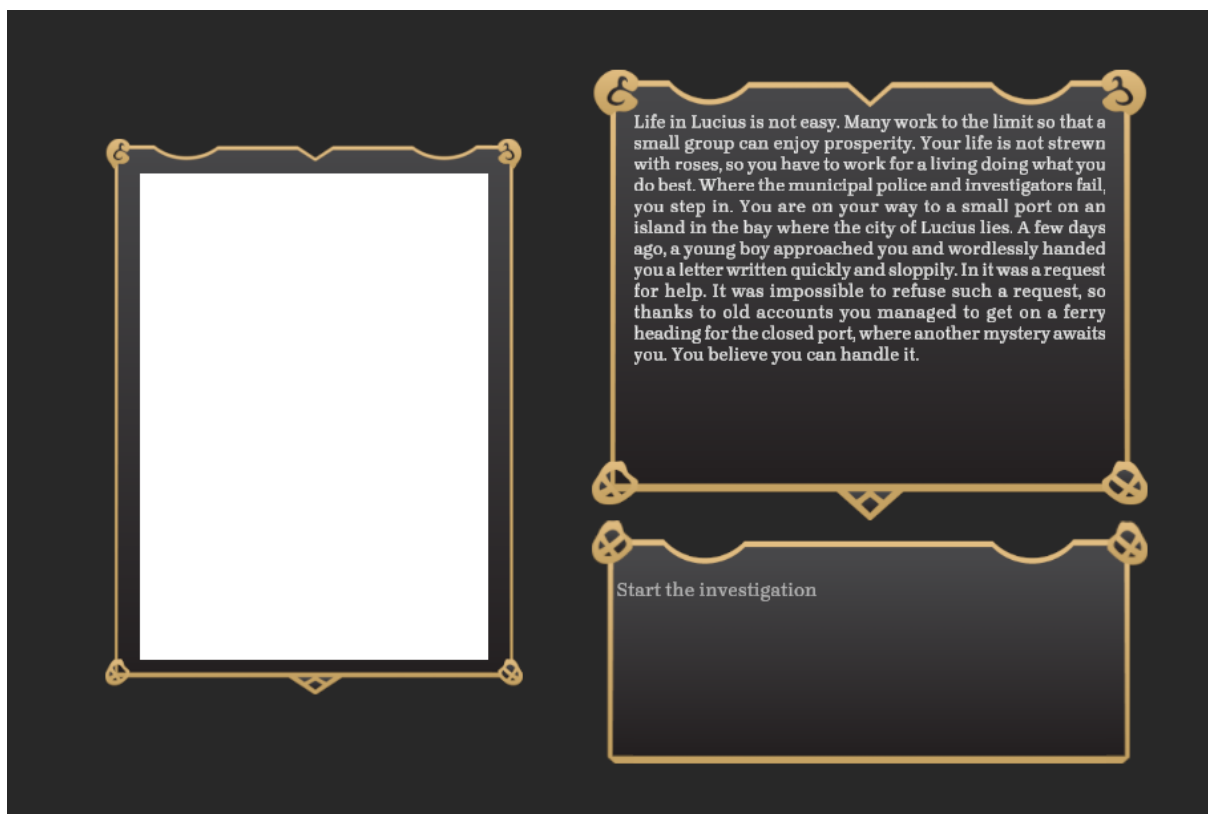
5.3 Narrative Event

Narrative event (z ang. Wydarzenie opisowe) jest funkcją stosowaną, by bardziej nakreślić wydarzenie w świecie gry, wprowadzić gracza w akcję. Wydarzenie opisowe ma charakter zbliżony do książki – gracz czyta wycinek tekstu i podejmuje na jego podstawie decyzję. To szybki i łatwy sposób na wprowadzenie mniejszych zdarzeń, które nie potrzebują zadań, dialogów czy zaprogramowanych spotkań w grze.

Narrative event spełnia również funkcje starych gier paragrafowych, gdzie gracz miał do dyspozycji jedynie fragment tekstu i dostępne dla niego opcje akcji, co powinien zrobić dalej.

W trakcie wyświetlania wydarzenia, można wyświetlić również przypisane do niego grafiki, by nieco lepiej umiejscowić gracza w nowej sytuacji.

Inną funkcją jest sterowanie rozgrywką z poziomu wydarzenia. W czasie programowania istnieje szereg poleceń, które służą do ładowania nowych plansz, usuwania, bądź dodawania przedmiotu do ekwipunku gracza oraz wielu innych przydatnych z punktu widzenia designera poleceń.



Rysunek 5: Wygląd Narrative Event'u

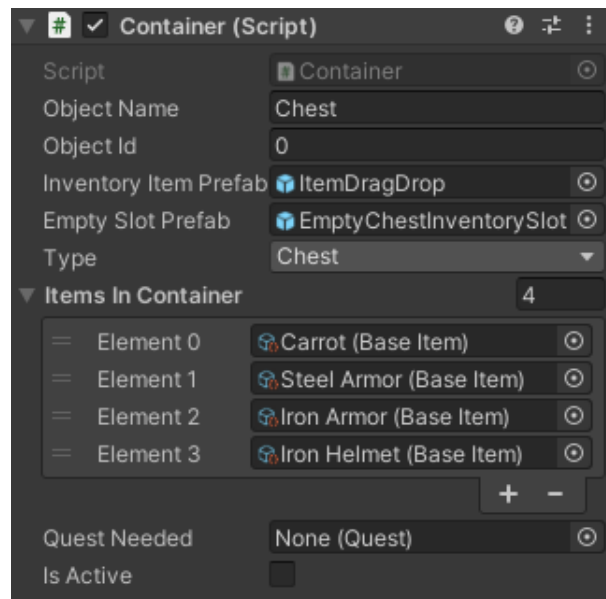
5.4 Dialogue System

Dialogi w grze RPG są drugą najważniejszą funkcją i stanowią podstawę do rozwoju fabuły. Są to rozmowy lub interakcje z NPC, które imitują zwykłą rozmowę, gdzie dwoje lub więcej rozmówców wymienia swoje poglądy oraz dyskutuje na bieżące tematy.

Po wyświetleniu pierwszej linii dialogu, gracz może kontrolować przełączanie na kolejne kwestie, by dostosować tempo rozmowy do swoich potrzeb. Z czasem będzie miał do wyboru jedną lub kilka opcji dialogowych, które prowadzą do innych kwestii.

Dialog ma dać poczucie żyjącego świata, a dzięki kontroli nad zmiennymi zdań na bieżąco dostosowywane są linie tekstu. W trakcie rozmowy NPC może wspomnieć akcje gracza, wydarzenie do którego się przyczynił czy zadanie, którego nie udało mu się rozwiązać.

W trakcie rozmowy można również kontrolować przebieg zadań. Część z nich polega na rozmowie z danym NPC, co jest wykonywane, gdy przypiszemy do kwestii odpowiednią funkcję z identyfikatorem zadania.



Rysunek 6: Przykładowy Container

5.5 Interactive Objects

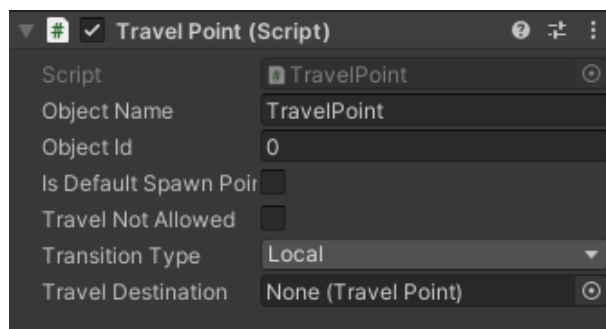
Gra pokroju The Cultist nie może być statyczna. Gracz ma mieć możliwość wpływania na świat i wchodzenia z nim w interakcję, tak jakby mógł to robić w rzeczywistym świecie.

Wypełnieniu tego założenia służą obiekty w świecie gry, które gracz może zbadać, otworzyć itd. Przykładem może być drewniana skrzynia, która ma w sobie cenny przedmiot, więc gracz przeszukuje kolejne skrzynie, by w końcu trafić na ukryty skarb. Innym przykładem może być obiekt nazwany w architekturze gry travel point, który umożliwia graczom przemieszczanie się między planszami, gdzie toczy się rozgrywka.

Obiekty, które są pojemnikami na przedmioty, jak skrzynie czy beczki, mają zaprogramowany system ekwipunku podobny do tego używanego przez gracza. Można wyciągnąć z nich przedmiot, który doda się do przedmiotów gracza.

Punkty podróży (z ang. Travel point) pozwalają przemieszczać się w obrębie jednej planszy, między dwoma różnymi planszami i pozwala również przejść do widoku mapy miasta, z którego gracz sam wybiera, w który rejon się uda.

Interaktywne obiekty służą również jako element fabuły. Być może gracz otrzyma zadanie odszukania zgubionego plecaka lub dowie się, że pewna część obszaru rozgrywki została zablokowana i nie może użyć travel point'u.



Rysunek 7: Przykładowy Travel Point

6 Część 4 - Algorytmy

W tej części zostały opisane najważniejsze elementy kodu. Wyjaśniono ich zadania, budowę oraz wykorzystanie skryptu lub systemu.

6.1 Lista Skryptów

6.1.1 Game Manager

Game Manager odpowiada za kontrolowanie przebiegu gry - wstrzymuje i wznowia grę - modyfikuje ekwipunek postaci oraz jej statystyki. Przypisano do niego scriptableObject, który odpowiada za przechowywanie informacji o graczu.

Listing 1: Pauzowanie rozgrywki

```
1 private static void HandleGamePause(bool boolean)
2     {
3         if (boolean)
4         {
5             PauseGame();
6         }
7         else
8         {
9             ResumeGame();
10        }
11    }
12 private static void PauseGame()
13     {
14         Time.timeScale = 0;
15     }
16
17 private static void ResumeGame()
18     {
19         Time.timeScale = 1;
20     }
```

Game Manager nasłuchuje odpowiedniego eventu, który informuje go, czy wstrzymać, czy wznowić grę. Po otrzymaniu boola, skrypt decyduje, co zrobić z flow gry.

Listing 2: Zarządzanie ekwipunkiem gracza

```
1 private static void AddItemToList (ICollection<BaseItem> listOfItems,
    BaseItem itemToAdd)
2     {
3         Dodanie
4     }
5
6 private static void RemoveItemFromList (IList<BaseItem>
    listOfItems, BaseItem itemToRemove)
7     {
8         for (var i = 0; i < listOfItems.Count; i++)
9         {
10             if (listOfItems[i] != itemToRemove) continue;
11             listOfItems.Remove(listOfItems[i]);
12             break;
13         }
14     }
15
16 private void CheckForItemInInventory (BaseItem itemToFind)
17     {
18         var quantityOfItemsFound = playerData.playerInventoryItems.
            Count(item => item == itemToFind);
19
20         OnReturnQuantityOfItems?.Invoke(quantityOfItemsFound);
21     }
22
23 private void RemoveQuestItem (BaseItem itemToRemove, int
    quantityOfItemToRemove)
24     {
25         for (var i = 0; i < quantityOfItemToRemove; i++)
26         {
27             for (var j = 0; j < playerData.playerInventoryItems.Count
                ; j++)
28             {
29                 if (playerData.playerInventoryItems[j] !=
                    itemToRemove) continue;
30
31                 playerData.playerInventoryItems.Remove(playerData.
                    playerInventoryItems[j]);
32
33                 break;
34             }
35         }
36     }
```

Game Manager modyfikuje listę z wyposażeniem postaci oraz ekwipunkiem gracza po otrzymaniu żądania o usunięcie lub dodanie do odpowiedniej listy nowego przedmiotu. Wcześniej jednak należy sprawdzić, czy jeśli trzeba usunąć przedmiot, czy znajduje się on w liście ekwipunku, by został usunięty.

Listing 3: Obliczanie nowej wartości dla statystyki gracza

```
1 private static int CalculateStatValue(int playerDataStat, ItemEffect
   effect, bool isEffectActive)
2     {
3         var updatedStat = playerDataStat;
4         if (isEffectActive)
5         {
6             if (effect.typeOfInfluence == ItemEffect.
               TypesOfInfluenceOnStat.IncreaseStat)
7             {
8                 updatedStat += effect.pointsAffecting;
9             }
10            else
11            {
12                updatedStat -= effect.pointsAffecting;
13            }
14        }
15        else
16        {
17            if (effect.typeOfInfluence == ItemEffect.
               TypesOfInfluenceOnStat.IncreaseStat)
18            {
19                updatedStat -= effect.pointsAffecting;
20            }
21            else
22            {
23                updatedStat += effect.pointsAffecting;
24            }
25        }
26        return updatedStat;
27    }
```

Gdy następuje modyfikacja statystyki, Game Manager oblicza nową wartość, biorąc starą i modyfikując ją o zmienną. Następnie przypisuje nową wartość odpowiedniemu polu w Player Data, gdzie trzymane są statystyki gracza.

6.1.2 Location Manager

Location Manager odpowiada głównie za zmianę scen. Po otrzymaniu żądania o zmianę sceny, skrypt ładuje nową scenę i odładowuje poprzednią. Manager służy również do wczytania scen takich jak: główne menu, scena z UI, kamerą i postacią gracza oraz scenę podsumowania

kampanii. Kolejnym ważnym elementem Location Managera jest wyszukanie domyślnego punktu, w którym pojawia się postać gracza po wczytaniu nowej sceny.

Listing 4: Wyszukanie domyślnego spawn pointa

```
1 private void SetPlayerAtTravelPoint(Vector3 interactorPosition)
2     {
3         _newSpawnPoint = interactorPosition;
4
5         _pointFound = true;
6
7         SpawnPlayerAtPosition(_newSpawnPoint);
8
9         Debug.Log("Travel point set: " + _newSpawnPoint);
10    }
11 private void SetNewDefaultSpawnPoint(Vector3 interactorPosition)
12    {
13        _defaultSpawnPoint = interactorPosition;
14
15        _pointFound = true;
16
17        SpawnPlayerAtPosition(_defaultSpawnPoint);
18
19        Debug.Log("Default point set: " + _defaultSpawnPoint);
20    }
```

Listing 5: Dwie opcje ładowania sceny

```
1 private void OnChangeLocation(string scene, bool setActive)
2     {
3         _currentLocation = SceneManager.GetActiveScene();
4
5         _locationToLoad = scene;
6
7         _pointFound = false;
8
9
10        StartCoroutine(LoadSceneAsync(_locationToLoad, setActive));
11
12        UnloadScene();
13    }
14 private void OnChangeLocation(string scene, int travelPointId)
15     {
16         _currentLocation = SceneManager.GetActiveScene();
17
18         _locationToLoad = scene;
19
20         _travelPointIdToSpawnAt = travelPointId;
21
22         _pointFound = false;
23
24        StartCoroutine(LoadSceneAsync(_locationToLoad, true));
25
26        UnloadScene();
27    }
```

Pierwsza metoda służy do ładowania scen takich jak: UI, główne menu, podsumowanie kampani itd. Są to sceny, które można ustawić jako aktywna scena. Druga metoda służy do przenoszenia gracza między scenami, na których toczy się rozgrywka.

6.1.3 Quest Manager

Quest Manager śledzi przebieg historii i zadań wykonywanych przez gracza. Przechowuje questLog, gdzie znajdują się pliki z wykonanymi i aktywnymi zadaniami. Śledzi również aktywny narrative event.

Listing 6: Quest Manager

```
1 [SerializeField] private QuestLog currentQuestLog;
2
3
4 [SerializeField] private Campaign ongoingCampaign;
5
6 [SerializeField] private NarrativeEvent currentNarrativeEvent;
7
8 private void UpdateCurrentNarrativeEvent (NarrativeEvent
   narrativeEvent)
9 {
10     currentNarrativeEvent = narrativeEvent;
11
12     OnNarrativeEventPass?.Invoke (narrativeEvent);
13 }
14
15 private void UpdateCampaign (Campaign newCampaign)
16 {
17     ongoingCampaign = newCampaign;
18 }
19
20 private void PassQuestLog ()
21 {
22     OnQuestLogPass?.Invoke (currentQuestLog.ReturnActiveQuestlines (),
   currentQuestLog.ReturnCompletedQuestlines ());
23 }
```

6.1.4 Narrative Event Handler

Zadaniem skryptu jest obsługa narrative event'ów. Wyświetla on event, gdy zostaje wywołany, ładuje pliki graficzne, jeśli jakieś są dołączone do eventu oraz obsługuje logikę za interakcją z opcjami eventu.

Listing 7: Wyszukiwanie warunków narrative event'u

```
1 private void SearchForChoiceFlag(Choice choice)
2     {
3         var conditionsPassed = new List<bool>();
4
5         if (choice.tags == null)
6         {
7             CreateChoicePrompt(choice);
8             return;
9         }
10        foreach (var flag in choice.tags)
11        {
12            var array = flag.Split(":");
13
14            if(!array[0].Contains("flag")) continue;
15
16            var choiceCanBeSeen = false;
17
18            switch (array[1])
19            {
20                case "hasItem":
21                    StartCoroutine(CheckForItemInInventory(
22                        ReturnItemFromList(int.Parse(array[2])),
23                        result => { choiceCanBeSeen = result; }));
24                    break;
25
26                case "statValue":
27                    StartCoroutine(CheckForStatValue(array[2], int.Parse(
28                        array[3]),
29                        result => { choiceCanBeSeen = result; }));
30
31            }
32            conditionsPassed.Add(choiceCanBeSeen);
33        }
34
35        if (conditionsPassed.Any(condition => condition == false))
36        {
37            return;
38        }
39    }
```

```

36     }
37
38     CreateChoicePrompt(choice);
39 }

```

Metoda ustala, jaki warunek posiada aktualnie wyświetlany event i jeśli spełnione zostaną wszystkie warunki, wyświetla odpowiednią opcję dostępną dla gracza.

Listing 8: Wyszukiwanie komend narrative event'u

```

1 private void SearchForCommand()
2 {
3     if (_currentStory.currentTags == null) return;
4
5     var commandList = _currentStory.currentTags;
6
7     foreach (var array in commandList.Select(command => command.Split
8         (':')))
9     {
10         if (!array[0].Contains("command") || array[0] == null) return
11             ;
12
13         switch (array[1])
14         {
15             case "loadLevel":
16                 LoadLevelFromNarrativeEvent(array[2]);
17                 break;
18
19             case "addItem":
20                 AddItemFromNarrativeEvent(int.Parse(array[2]), int.
21                     Parse(array[3]));
22                 break;
23
24             case "removeItem":
25                 RemoveItemFromNarrativeEvent(int.Parse(array[2]), int
26                     .Parse(array[3]));
27                 break;
28
29             case "startQuest":
30                 StartQuest(array[2], int.Parse(array[3]));
31                 break;
32
33             case "completeQuest":
34                 CompleteQuest(array[2], int.Parse(array[3]));
35                 break;
36         }
37     }
38 }

```



```

32         case "failQuest":
33             FailQuest(array[2], int.Parse(array[3]));
34             break;
35     }
36 }
37 }

```

Podobnie jak poprzednim przypadku, skrypt sprawdza, czy wyświetlonemu na ekranie eventowi przypisano komendę, którą należy wykonać po zamknięciu eventu.

Listing 9: Ładowanie sceny z komendy narrative eventu

```

1 private void LoadLevelFromNarrativeEvent(string sceneName)
2 {
3     var locationChange = new CallLocationChange();
4
5     locationChange.ChangeLocation(sceneName, true);
6 }

```

Listing 10: Dodawanie i usuwanie przedmiotu z komendy narrative eventu

```

1 private void AddItemFromNarrativeEvent(int itemId, int quantity)
2 {
3     foreach (var item in _currentEventItems)
4     {
5         if (item.itemId != itemId) continue;
6
7         var playerEvent = new PlayerEvents();
8
9         for (int i = 0; i < quantity; i++)
10        {
11            playerEvent.AddItem(item);
12        }
13    }
14 }
15 private void RemoveItemFromNarrativeEvent(int itemId, int quantity)
16 {
17     foreach (var item in _currentEventItems)
18     {
19         if (item.itemId != itemId) continue;
20
21         var playerEvent = new PlayerEvents();
22
23         for (int i = 0; i < quantity; i++)
24        {

```

```

25         playerEvent.RemoveItem(item);
26     }
27 }
28 }

```

Listing 11: Kontrolowanie statusu zadań z komendy narrative eventu

```

1 private void StartQuest(string questlineName, int questNumber)
2 {
3     var questEvent = new CallQuestEvents();
4
5     questEvent.StartQuest(ReturnQuestId(questlineName, questNumber));
6 }
7
8 private void CompleteQuest(string questlineName, int questNumber)
9 {
10    var questEvent = new CallQuestEvents();
11
12    questEvent.CompleteQuest(ReturnQuestId(questlineName, questNumber));
13 }
14 private void FailQuest(string questlineName, int questNumber)
15 {
16    var questEvent = new CallQuestEvents();
17
18    questEvent.FailQuest(ReturnQuestId(questlineName, questNumber));
19 }

```

6.1.5 Test Player

Skrypt wykonuje obliczenia na bazie statystyki gracza. Pobiera odpowiednią statystykę, a następnie losuje liczbę między 1 i 20 na wzór rzutu kością o dwudziestu ściankach. Ostatnim krokiem jest dodanie wylosowanej wartości do pobranej statystyki i zwrócenie wyniku z porównaniem do wartości trudności testu.

Listing 12: Testowanie statystyki gracza

```

1 public static event Action<int> OnDiceRoll;
2 public static bool TestAbility(int basePlayerNumber, int
    baseTestDifficulty)
3 {
4     var random = new Random();
5

```

```
6     var testResult = random.Next(1, 20);
7
8     testResult += basePlayerNumber;
9     Debug.Log("Test result: " + testResult + " => " + "dice roll result
10         : "
11             + (testResult - basePlayerNumber) + ", player value of
12               ability added: " + basePlayerNumber + ". Test
13               difficulty: " + baseTestDifficulty);
14
15     OnDiceRoll?.Invoke(testResult);
16
17     return testResult >= baseTestDifficulty;
18 }
```

6.1.6 Player Data

Klasa `PlayerData` jest rozszerzeniem klasy `BaseCharacter`, która przechowuje podstawowe statystyki gracza, jego ekwipunek oraz dostępne zdolności, rozszerzając klasę o umiejętności dostępne tylko dla gracza.

Listing 13: Zmiana wartości `PlayerData`

```
1 private void CopyData(PlayableCharacter newStats)
2     {
3         Type playableCharType = typeof(PlayableCharacter);
4         Type playerDataType = typeof(PlayerData);
5
6         FieldInfo[] playableFields = playableCharType.GetFields(
7             BindingFlags.Public | BindingFlags.Instance);
8         FieldInfo[] playerFields = playerDataType.GetFields(
9             BindingFlags.Public | BindingFlags.Instance);
10
11         foreach (FieldInfo playableField in playableFields)
12         {
13             FieldInfo correspondingPlayerField = Array.Find(
14                 playerFields, field => field.Name == playableField.
15                 Name);
16
17             if (correspondingPlayerField != null)
18             {
19                 correspondingPlayerField.SetValue(this, playableField
20                     .GetValue(newStats));
21             }
22         }
23
24         Debug.Log("Data copied!");
25     }
```

Funkcja pobiera statystyki z klasy `PlayableCharacter` i przypisuje je do odpowiednich pól w `PlayerData`. Dzięki temu jeden obiekt typu `PlayerData`, może obsługiwać wiele plików z innymi postaciami.

6.1.7 Player Events

Player Events służą do wszelkich interakcji postaci gracza ze światem. Używamy ich, gdy chcemy dać znać, by dodać lub usunąć dany przedmiot z ekwipunku, uleczyć postać

lub przetestować statystykę. Klasa służy jedynie do wywołania odpowiedniego eventu z parametrem.

Listing 14: Przykładowe eventy związane z postacią gracza

```
1 public void AddItem(BaseItem item)
2     {
3         OnAddItemToInventory?.Invoke(item);
4     }
5
6     public void RemoveItem(BaseItem item)
7     {
8         OnRemoveItemFromInventory?.Invoke(item);
9     }
10
11     public void CheckForItem(BaseItem item)
12     {
13         OnCheckForItem?.Invoke(item);
14     }
15
16     public void CheckForStatValue(Stat stat)
17     {
18         OnCheckForStatValue?.Invoke(stat);
19     }
```

6.2 Lista Systemów

6.2.1 System Dialogów

Listing 15: Podstawowa klasa dialogu

```
1 public class Dialogue
2     {
3         public TextAsset dialogueInstance;
4
5         public bool oneTimeDialogue;
6
7         public bool wasSeen;
8
9         public List<DialogueRequiredQuest> RequiredQuests;
10    }
```

Klasa służy do przypisania dialogu do interakcji z NPC oraz determinowania, czy dany dialog może zostać wywołany. Jeśli jest to dialog pojedynczy, po zakończeniu zostaje on usunięty.

Listing 16: Klasa DialogController - inicjalizacja i kontrola dialogu

```
1 private void InitializeStory(DialogueInteraction dialogueInteraction)
2     {
3         _inkAsset = dialogueInteraction.dialogueAsset;
4         _inkStory = new Story(_inkAsset.text);
5         HandleInkError();
6
7         _originConversationPoint = dialogueInteraction;
8     }
9 private void NextDialogue(InputAction.CallbackContext context)
10    {
11        if (_inkStory.canContinue && _inkStory.currentChoices.Count == 0)
12        {
13            ContinueStory();
14        }
15        if (!_inkStory.canContinue && _inkStory.currentChoices.Count == 0
16            || _npcTextBox.text == "")
17        {
18            EndDialogue();
19        }
20        if (_inkStory.currentChoices.Count == 0 ||
21            _playerChoicesContainer.transform.childCount != 0) return;
22
23        _listOfChoices?.RemoveRange(0, _listOfChoices.Count);
24        ClearInfoToClick();
25        _listOfChoices = _inkStory.currentChoices;
26        DisplayChoices(_listOfChoices);
27    }
28
29 private void ContinueStory()
30    {
31        if (!_inkStory.canContinue) return;
32
33        _npcTextBox.text = $"<color=yellow>{_charName}</color> " +
34            _inkStory.Continue();
35        HandleQuestManagement();
36    }
```

Gdy dialog zostanie wywołany, następuje inicjalizacja pliku z dialogiem, w którym znajduje się plik ink. Kolejne kwestie dialogowe ładowane są do okna UI, a kontrola nad postępem dialogu następuje po przez przyciśnięcie przycisku kontynuacji lub wyboru opcji dialogowej.

Listing 17: Klasa DialogController - opcje dialogowe

```

1 private void DisplayChoices (List<Choice> list)
2     {
3         foreach (var choice in list)
4         {
5             var dialogueOption = DisplayDialogueOption();
6
7             var testToPassInfo = AddTestInfoToChoice(choice);
8
9             LoadDialogueOptionContent(dialogueOption, choice.index,
              testToPassInfo + choice.text);
10            dialogueOption.GetComponent<DialogueSendChoice>().choice =
              choice;
11
12        }
13    }
14 private void SubmitChoice (Choice choice)
15     {
16         var wasPlayerTested = TestPlayerWithChoice(choice);
17
18         _inkStory.ChooseChoiceIndex(_listOfChoices.IndexOf(choice));
19         ClearChoices(_playerChoicesContainer.transform.
              GetComponentsInChildren<DialogueSendChoice>());
20         if (_inkStory.canContinue)
21         {
22             ContinueStory();
23             if (wasPlayerTested && _inkStory.canContinue)
24             {
25                 ContinueStory();
26             }
27         }
28         DisplayInfoToClick();
29     }

```

Opcje dialogowe są interakcją gracza z historią przedstawioną w dialogu. Skrypt wyświetla dostępne opcje, zdefiniowane w pliku ink z dialogiem. Gracz następnie wybiera odpowiednią opcję, która zostaje wywołana przez skrypt.

Listing 18: Klasa DialogController - zarządzanie zadaniami z dialogu

```

1 private void HandleQuestManagement ()
2     {
3         var currentTags = CheckForQuestTags(_inkStory.currentTags);
4         var questIdToReturn = new QuestId();
5

```

```

6      foreach (var tagContent in currentTags.Select (questTag =>
7          questTag.Split (':'))
8      {
9          questIdToReturn.idPrefix = tagContent [1];
10         questIdToReturn.questNumber = int.Parse (tagContent [2]);
11
12         var questEvent = new CallQuestEvents ();
13
14         switch (tagContent [0])
15         {
16             case "questStart":
17                 questEvent.StartQuest (questIdToReturn);
18                 break;
19             case "questComplete":
20                 questEvent.CompleteQuest (questIdToReturn);
21                 break;
22             case "questFail":
23                 questEvent.FailQuest (questIdToReturn);
24                 break;
25         }
26     }
27
28     private List<string> CheckForQuestTags (List<string>
29         currentTagsAtContinue)
30     {
31         List<string> list = new List<string> ();
32
33         foreach (var questTag in currentTagsAtContinue)
34         {
35             if (questTag.Contains ("quest")) list.Add (questTag);
36         }
37
38         return list;
39     }

```

Z poziomu dialogu można zarządzać też zadaniami. Można rozpocząć, zakończyć lub niewykonać zadanie, co zostaje przekazane do pliku z zadaniem.

Listing 19: Klasa DialogController - ładowanie zmiennych z zadań do dialogu

```

1 private void CallForVariable ()
2 {
3     var listOfVariables = LoadNeededVariablesFromDialogue ();
4 }

```



```

5         OnCallVariables?.Invoke(listOfVariables);
6     }
7
8     private List<string> LoadNeededVariablesFromDialogue()
9     {
10         var listOfNeededVariables = new List<string>();
11
12         foreach (var inkVariable in _inkStory.variablesState)
13         {
14             listOfNeededVariables.Add(inkVariable);
15         }
16
17         return listOfNeededVariables;
18     }
19     private void LoadQuestVariablesToDialogue(List<QuestVariables>
20         listOfReturnQuestVariables)
21     {
22         _listOfQuestVariables = new List<QuestVariables>();
23
24         _listOfQuestVariables = listOfReturnQuestVariables;
25
26         foreach (var variable in _listOfQuestVariables)
27         {
28             var variableContent = variable.conditionPassed;
29
30             _inkStory.variablesState[variable.variableCodeName] =
31                 variableContent;
32
33             Debug.Log("Variable with name: " + variable.variableCodeName
34                 + " has been passed to story!");
35         }
36         SaveStoryState();
37         LoadStoryState(_originConversationPoint.dialogueSaved);
38     }
39     private void SendNewVariablesToDialogueLog()
40     {
41         var questVariablesList = new List<QuestVariables>();
42
43         var listOfVariables = LoadNeededVariablesFromDialogue();
44
45         foreach (var variable in listOfVariables)
46         {
47             var newVariableToSet = new QuestVariables
48             {
49                 variableCodeName = variable,

```

```

47         conditionPassed = (bool)_inkStory.variablesState[variable
48             ];
49
50         questVariablesList.Add(newVariableToSet);
51     }
52
53     OnSetNewVariables?.Invoke(questVariablesList);
54 }

```

Zmienne zadań mogą determinować, czy dana opcja dialogowa pojawi się w trakcie gry. Skrypt daje znać, które zmienne trzeba pobrać i sprawdzić ich wartość. Następnie nowe wartości zostają przypisane do dialogu i zadania.

6.2.2 System Questów

System questów, inaczej zadań, tworzą typy klas: campaign, questline, quest, reward oraz questId. Pierwsze cztery omówiono w wcześniejszym rozdziale. Klasa questId służy do identyfikacji zadania.

Listing 20: Rozpoczęcie kampanii

```

1 private void StartCampaign(Campaign startCampaign)
2     {
3         if(startCampaign.campaignId != campaignId) return;
4
5         if (requiredCampaigns.Any(campaign => !campaign.isCompleted))
6         {
7             return;
8         }
9
10        hasStarted = true;
11
12        OnCampaignStart?.Invoke(this);
13
14        SendStartedCampaignData(campaignId.ToString(), campaignName);
15    }

```

Skrypt ma za zadanie sprawdzić, czy wszystkie warunki są spełnione, by można było rozpocząć kampanię przez gracza. Metoda zostaje wywołana wraz z otrzymaniem informacji z eventu, którego nasłuchuje.

Listing 21: Ukończenie kampanii

```
1 private void CompleteCampaign(Questline recentlyCompletedQuestline)
2     {
3         if (campaignQuestlines.Any(questline => questline.questlineName
4             != recentlyCompletedQuestline.questlineName))
5         {
6             return;
7         }
8
9         if (requiredQuestlines.Any(questline => !questline.
10             questlineCompleted) || !hasStarted)
11         {
12             return;
13         }
14
15         isCompleted = true;
16
17         OnCampaignComplete?.Invoke(this);
18         Debug.Log(campaignName + " completed!");
19
20         SendCompletedCampaignData(campaignId.ToString(), campaignName);
21     }
```

Skrypt ma za zadanie sprawdzić, czy wszystkie warunki są spełnione, by można było zakończyć kampanię przez gracza. Na koniec wysyłane są dane do analizy, która kampania została ukończona przez gracza. Metoda zostaje wywołana, gdy wykryje się odpowiedni event z parametrem questline'u, który został przed chwilą ukończony.

Listing 22: Rozpoczęcie questline'u

```
1 private void StartQuestline(Quest startedQuest)
2     {
3         Instantiate(this);
4         if (!questlineSteps.Any(quest => quest == startedQuest && !
5             questlineStarted)) return;
6
7         questlineStarted = true;
8
9         OnQuestlineStart?.Invoke(this);
10
11         SendStartedQuestlineData(questlineId.ToString(), questlineName);
12     }
```

Działanie jest podobne jak w przypadku kampanii. Jeśli zostaną spełnione wszystkie warunki, następuje aktywacja linii zadań.

Listing 23: Ukończenie questline'u

```
1 private void CompleteQuestline ()
2     {
3         questlineCompleted = true;
4
5         OnQuestlineCompleted?.Invoke (this);
6
7         SendCompletedQuestlineData (questlineId.ToString (), questlineName);
8
9         Quest.OnQuestCompleted -= CheckForRemainingOnQuests;
10
11        Quest.OnQuestStarted -= StartQuestline;
12
13        Quest.OnQuestFailed -= CheckForRemainingOnQuests;
14    }
```

Metoda oznacza questline jako ukończony i wywołuje event z parametrem questline'u. Można ją wykonać dopiero, gdy w poniższym kodzie wszystkie warunki zostaną spełnione.

Listing 24: Sprawdzenie statusu zawartych w questline'ie zadań

```
1 private void CheckForRemainingOnQuests (Quest completedQuest)
2     {
3         var allQuestsCompleted = true;
4
5         foreach (var quest in questlineSteps)
6         {
7             if (quest.questCompleted) continue;
8
9             allQuestsCompleted = false;
10
11            break;
12        }
13
14        if (allQuestsCompleted)
15        {
16            CompleteQuestline ();
17        }
18    }
```

Sprawdzanie, czy wszystkie potrzebne warunki są spełnione. Jeśli tak, wywoływana jest metoda CompleteQuestline.

Listing 25: Rozpoczęcie zadania

```
1 protected virtual void StartQuest (QuestId questIdFromEvent)
2     {
3         if (questIdFromEvent.idPrefix != questId.idPrefix ||
4             questIdFromEvent.questNumber != questId.questNumber)
5             return;
6     }
7
8     if (questStarted)
9     {
10        return;
11    }
12
13    if (prerequisiteQuests != null && prerequisiteQuests.Any(
14        requiredQuest => !requiredQuest.questCompleted))
15    {
16        return;
17    }
18
19    questDesc = originalQuestDesc;
20
21    shortQuestDesc = originalShortQuestDesc;
22
23    questStarted = true;
24
25    SendStartedQuestData (questId.ToString(), questName);
26
27    OnQuestStarted?.Invoke (this);
28 }
```

Jeśli wywołano rozpoczęcie odpowiedniego zadania, następuje aktywacja. Następnie sprawdzane są wszystkie warunki potrzebne do rozpoczęcia zadania. Opisy zadań, które prezentowane są graczowi, zostają zaktualizowane, by gracz wiedział, co ma zrobić.

Listing 26: Zakończenie zadania

```
1 protected virtual void CompleteQuest (QuestId questIdFromEvent)
2     {
3         if (questIdFromEvent.ToString() != questId.ToString())
4         {
5             return;
6         }
7     }
```

```

6      }
7
8      if (questCompleted || questFailed) return;
9
10     if (prerequisiteQuests != null)
11     {
12         if (questSteps.Any(requiredQuest => !requiredQuest.
13                               questCompleted))
14         {
15             return;
16         }
17
18         if (!questStarted) return;
19
20         questDesc = questCompletedDesc;
21
22         questCompleted = true;
23
24         StopListeningToQuestEvents();
25
26         RewardPlayer();
27
28         SendCompletedQuestData(questId.ToString(), questName);
29
30         OnQuestCompleted?.Invoke(this);
31     }

```

Gdy zadanie zostaje ukończone, następuje sprawdzanie warunków oraz zmiana opisów zadań, by odpowiadały wykonanemu zadaniu, a gracz otrzymuje nagrodę za wykonane zadanie.

Listing 27: Zadanie nie zostało wykonane pomyślnie

```

1 protected virtual void FailQuest(QuestId questIdFromEvent)
2 {
3     if (questIdFromEvent.ToString() != questId.ToString())
4     {
5         return;
6     }
7
8     if (questFailed || !questStarted || questCompleted) return;
9
10    if (prerequisiteQuests != null)
11    {

```

```

12         foreach (var quest in prerequisiteQuests.Where(quest => quest
13             .questStarted))
14         {
15             quest.FailQuest(quest.questId);
16         }
17
18         questDesc = questFailedDesc;
19
20         questFailed = true;
21
22         questCompleted = true;
23
24         StopListeningToQuestEvents();
25
26         OnQuestFailed?.Invoke(this);
27     }

```

Gdy graczowi nie uda się spełnić określonych wymagań do ukończenia zadania lub gdy z przyczyn fabularnych zadanie zostanie oznaczone jako zakończone, zadanie otrzymuje status nieudanego.

Listing 28: Przyznanie nagrody za wykonanie zadania

```

1 private void RewardPlayer()
2     {
3         foreach (var reward in questRewards)
4         {
5             switch (reward.rewardType)
6             {
7                 case RewardType.GiveItem:
8                     GiveItemReward(reward.rewardItem);
9                     break;
10                case RewardType.GiveClue:
11                    break;
12                case RewardType.GiveMoney:
13                    GiveMoneyReward(reward.rewardMoney);
14                    break;
15                case RewardType.GiveStatExp:
16                    break;
17                case RewardType.GiveStatLevel:
18                    GiveStatLevelReward(reward.statToModify, reward.
19                        rewardStatLevel);
20                    break;
21                case RewardType.GiveAbility:

```

```
21         GiveAbilityReward(reward.rewardAbility);
22         break;
23     default:
24         throw new ArgumentOutOfRangeException();
25     }
26 }
27 }
28
29 private static void GiveItemReward(BaseItem rewardItem)
30 {
31     var playerEvent = new PlayerEvents();
32
33     playerEvent.AddItem(rewardItem);
34 }
```

Skryt zczytuje nagrody przypisane do zadania i zleca wykonanie ich na podstawie jednej z kilku opcji. Następnie wywołany jest player event z odpowiednim parametrem.