AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE
WYDZIAŁ INFORMATYKI, ELEKTRONIKI I TELEKOMUNIKACJI
INSTYTUT ELEKTRONIKI

PROJEKT DYPLOMOWY

Gra kooperacyjna 2D typu horror

2D cooperative horror game made in Unity

Autorzy: **Bartosz Jaromin, Marcel Cisowski**

Kierunek studiów: Nowoczesne Technologie w Kryminalistyce

Opiekun pracy: dr inż. Marek Frankowski

Kraków, 2023 r.

STRESZCZENIE

Tematem niniejszej pracy inżynierskiej jest opracowanie kooperacyjnej gry komputerowej 2D przy pomocy silnika Unity oraz serwisu Photon. Gra należy do gatunku przygodowych oraz zawiera w sobie motywy horroru. Przeznaczona jest na urządzenia z systemem operacyjnym Windows. Rozgrywka zaprojektowana jest dla dwóch graczy, których zadaniem jest przechodzenie etapów i przezwyciężanie utrudnień w postaci potworów oraz braku oświetlenia. Projekt jest wersją demonstracyjną i zawiera wiele gotowych mechanik zapewniających pełne doświadczenie rozgrywki, a została ograniczone jedynie jej długość. Poprzez stosowanie licznych wzorców projektowych oraz technik pisania czystego kodu, projekt zorientowany jest na dalszą rozbudowę, bez potrzeby modyfikacji istniejących już rozwiązań. Praca składa się z 6 rozdziałów, które grupują najważniejsze informacje dotyczące procesu powstawania gry. Rozdział pierwszy skupia się na generalnym zamyśle gry. W kolejnym opisane są narzędzia i ich elementy, z których korzystamy do stworzenia projektu i które są kluczowe do zrozumienia kolejnych rozdziałów. Dwa kolejne rozdziały skupiają się na implementacji mechanik w grze, oraz implementacji rozgrywki wieloosobowej, czyli zasadniczej części naszego projektu. Rozdział piąty przedstawia zastosowane rozwiązania uwzględnione w napisanym przez nas kodzie. Ostatni rozdział skupia się na procesie eksportu projektu i dalszej perspektywie rozbudowy gry.

The subject of this engineering thesis is the development of a cooperative 2D computer game using the Unity engine and the Photon service. The game belongs to the adventure genre and includes horror themes. It is designed for devices with the Windows operating system. The game is designed for two players, whose task is to complete stages and overcome obstacles in the form of monsters and darkness. The project is a demonstration version of the game and contains many fully functional mechanics ensuring a full gameplay experience, with the only thing being limited is its length. By using numerous design patterns and clean code writing techniques, the project is oriented towards further development, without the need to modify existing solutions. The work consists of 6 chapters, which group the most important information about the process of creating the game. The first chapter focuses on the general idea of the game. The next chapter describes the tools and their elements that we use to create the project and which are key to understanding the following chapters. The next two chapters focus on the implementation of mechanics in the game, and the implementation of multiplayer gameplay, which is an essential part of our project. The fifth chapter presents the applied solutions included in the code we wrote. The last chapter focuses on the project export process and the further development of the game.

Spis treści

Wstęp

Gry komputerowe na przestrzeni ostatnich lat stały się gigantycznym przemysłem zrzeszającym wiele dziedzin. [1] W procesie tworzenia gier można wykorzystać między innymi umiejętności z zakresu ich projektowania, programowania, tworzenia grafik i animacji lub modeli 3D, a także projektowania dźwięku. Rodzajów gier jest bardzo wiele, mogą się one różnić mechanikami, motywem, stylem, obsługą graczy. Nasz projekt skupia się na sposobach implementacji poszczególnych elementów, które mogą zostać wykorzystane w grze z gatunku przygodowych z motywami horroru oraz zapewniającą możliwość rozgrywki graczom przez internet.

Celem projektu jest stworzenie wersji demonstracyjnej gry horrorowej w przestrzeni dwuwymiarowej, w której dwóch graczy musi kooperować ze sobą, aby przechodzić kolejne poziomy. Gra będzie przeznaczona na system operacyjny Windows i będzie umożliwiała graczom rozgrywkę przez internet w czasie rzeczywistym. Rozwiązania zastosowane w projekcie są wykonane w taki sposób, aby rozgrywka była płynna.

W pierwszym rozdziale zostanie opisany charakter rozgrywki stworzonej przez nas gry kooperacyjnej.

W kolejnym rozdziale poruszony zostanie temat wykorzystanego oprogramowania oraz narzędzi, z których korzystaliśmy podczas całego procesu tworzenia gry. Ma to na celu przybliżenie istotnych pojęć, w szczególności elementów silnika gry, gdyż nazwy i koncepty ich działania są konieczne do zrozumienia dalszej części pracy.

Trzeci rozdział posłuży nam do opisania w jaki sposób zostały stworzone i zaimplementowane mechaniki w grze. Ponadto poruszone zostaną niektóre zagadnienia związane ze spełnieniem założeń projektu tj. intuicyjność rozgrywki lub zbudowanie odpowiedniego klimatu. Opisuje on kluczową część naszej pracy inżynierskiej związaną z projektowaniem gier, zaimplementowanymi rozwiązaniami i napisanym przez nas kodem.

Następny rozdział przybliży sposób zaimplementowania rozgrywki sieciowej, a w szczególności kooperacji dwóch graczy w czasie rzeczywistym. Zostanie dogłębnie opisane działanie jednego z narzędzi o nazwie Photon PUN. Wytłumaczone zostanie również, w jaki sposób dostosować należało odpowiednie mechaniki gry do działania w trybie wieloosobowym.

W rozdziale piątym opisane zostaną rozwiązania zastosowane w niektórych elementach naszego projektu, co będzie poprzedzone koniecznymi definicjami. Są to techniki

pisania kodu w określony sposób, aby zapewniać daną funkcjonalność, a jednocześnie zachowując przy tym jego przejrzystość, hermetyczność i umożliwić prostą rozbudowę.

Ostatni rozdział opisuje finalne etapy tworzenia gry, w szczególności eksport gry na określoną platformę oraz proces. Wskazuje on również na elementy gry, które zostały dostosowane dzięki informacjom zwrotnym od graczy testujących grę.

1. Charakter rozgrywki

Część projektowa naszej pracy inżynierskiej jest wersją demonstracyjną przygodowej gry z motywami horroru z dwuosobowym trybem kooperacji. Projekt posiada zatem wiele gotowych mechanik, które pokazują jakie czynności może wykonywać gracz, oraz inne podstawowe elementy rozgrywki w tego typu grze. Wersja demonstracyjna nie skupia się jednak na długości rozgrywki a jedynie przedstawia jej zarys. Sposób w jaki zaprojektowane i wykonane są mechaniki pozwala na szybkie budowanie nowych poziomów tj. dorobienie zawartości gry, jednak odwzorowywanie środowisk lub miejsc po których porusza się gracz wiązałoby się z koniecznością zwiększenia liczby plików graficznych, co nie jest głównym elementem tego projektu.

Celem gracza będzie przemieszczanie się po dwuwymiarowej przestrzeni symulującej fragmenty opuszczonej posiadłości, zbierając klucze oraz unikając potworów, aby opuścić określony obszar. Dynamikę w grze dodają przeszkody w postaci rozmieszczonych w określonych miejscach potworów, natomiast klimat horrorowy gry buduje ponure i ciemne otoczenie zapewnione przez zaimplementowanie systemu świateł oraz odpowiednie ich ustawienie. [2]

Sama grafika jest w stylu pixelowym, to znaczy o niskiej rozdzielczości, przypominającym stare produkcje z czasów, kiedy pamięć komputerowa była ograniczona rozmiarem. Jest to świadomy wybór, ponieważ dzięki temu również wymagania sprzętowe do uruchomienia gry są niższe niż gdyby pliki graficzne miały wyższą rozdzielczość.

Rys. 1.1: Zrzut ekranu przedstawiający styl gry

2. Opis wykorzystanego oprogramowania i narzędzi

W niniejszym rozdziale zostaną opisane narzędzia wykorzystane do stworzenia wykonanego przez nas projektu, uwzględniając te, które umożliwiały jego przemyślane planowanie i usprawniały proces jego tworzenia. Rozdział w znacznym stopniu skupia się na silniku gry, na którym bazuje nasz projekt. Zostaną opisane najważniejsze elementy silnika, które są jego podstawową częścią, a ich znajomość jest kluczowa do zrozumienia dalszej części pracy.

2.1 Oprogramowania do zarządzania projektem

Celem ułatwienia planowania i usprawnienia procesu tworzenia gry, zdecydowaliśmy się na użycie **oprogramowania do zarządzania projektami** o nazwie ClickUp. Umożliwia on tworzenie list zadań, którym można przypisać status, na jakim etapie wykonania znajduje się czynność, osobę, która ma wykonać daną czynność, lub termin, do kiedy dana rzecz powinna zostać wykonana. We wspomnianym programie można również tworzyć dokumenty tekstowe. Zostały one użyte do opisania niektórych mechanik, aby ich późniejsze wykonanie było zgodne z pierwotnym konceptem. Korzystanie z tego oprogramowania znacznie ułatwiło planowanie projektu oraz śledzenie postępów.

2.2 Kontrola wersji

W procesie wytwarzania dowolnego oprogramowania istotne jest korzystanie z **systemu kontroli wersji**. Jest to oprogramowanie zapewniające możliwość śledzenia zmian w kodzie źródłowym oraz niosące pomoc deweloperom w łączeniu zmian dokonanych w tych samych plikach przez różne osoby. Umożliwia to tym samym śledzenie wszystkich modyfikacji dokonywanych na plikach oraz łatwe przywracanie poprzednich wersji. W tym celu wykorzystaliśmy oprogramowanie GitHub, które zapewnia również możliwość tworzenia repozytoriów w chmurze i udostępnia oprogramowanie do kontroli wersji, co pozwoliło nam na bezproblemowe aktualizowanie całego projektu.

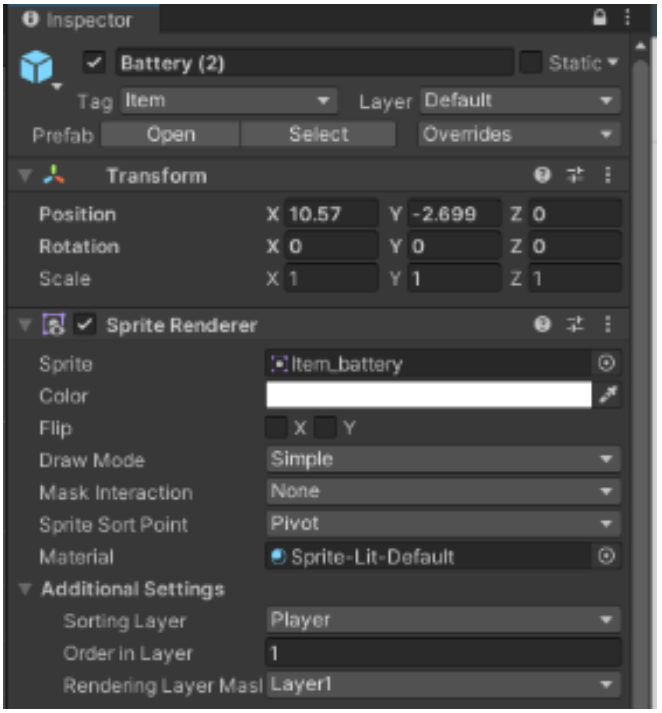
2.3 Silnik gry

Silnik gry, jako przeważająca część kodu powiązana z zintegrowanym środowiskiem programistycznym, to podstawowy element każdej gry komputerowej, który obsługuje interakcje między elementami w niej zawartymi. Projekt został wykonany przy użyciu silnika Unity, a precyzyjniej mówiąc, zestawu narzędzi Unity 2D. [3] Obsługuje on między innymi kolizje między obiektami i renderowanie grafiki, ponadto posiada wiele narzędzi pozwalających na wtórne wykorzystywanie kodu oraz jest kompatybilne z platformą .NET, co pozwala na tworzenie własnych mechanik w postaci skryptów napisanych w języku C#.

2.3.1 Komponent

Komponent to niezależny moduł programowy, który za pomocą jednoznacznie zdefiniowanego interfejsu udostępnia swą funkcjonalność. Jest on zdolny do współdziałania z innymi komponentami. [4]

Jednym z komponentów jest **Skrypt**, który za pomocą języka C# pozwala na wprowadzenie logiki lub dodatkowych mechanik w projekcie.



Rys. 2.1: Przykładowe komponenty (Transform i Sprite Renderer)

2.3.2 Obiekt Gry

Podstawową klasą każdej jednostki gry jest klasa GameObject (zwana dalej „obiekt gry” lub „obiekt”). Obiekty gry same w sobie nie wykonują żadnego zadania, natomiast służą one jako kontenery na komponenty, które implementują jego prawdziwą funkcjonalność. Podstawowym, nierozłącznym komponentem każdego obiektu jest Transform, który odpowiada za reprezentację pozycji oraz orientacji tego obiektu w przestrzeni. Można więc powiedzieć, że pusty obiekt jest punktem w przestrzeni 3D. Przestrzeń ta jest opisana przez układ współrzędnych kartezjańskich w przestrzeni 3-wymiarowej, tj. za pomocą osi X, Y oraz Z. Każdy obiekt gry może również być kontenerem na inne obiekty.

Parent Object to obiekt gry będący niepustym zbiorem innych obiektów gry, którego elementy określamy mianem Child Object. Te drugie będą miały swój Transform względny do tego posiadanego przez Parent Object. Tym samym przemieszczający się Parent Object powoduje poruszanie się swoich ChildObject razem z sobą. Na obrazku poniżej można zaobserwować przykładową hierarchię obiektów. Obiekt o nazwie Player reprezentujący logikę protagonisty (opisanego w rozdziale 3.4) jest Parent Object obiektu PlayerGFX, który odpowiada za wyświetlanie odpowiednich grafik i animacji.

Rys. 2.2: Przykładowa hierarchia obiektów gry

To jak każdy obiekt będzie wykorzystany, zależy wyłącznie od dodanych do niego komponentów, których limit, określany jest jedynie przez pamięć naszego komputera. Każdy GameObject może również posiadać określoną etykietę. Można o niej myśleć jako o zmiennej przechowującej tekst, która identyfikuje rodzaj obiektu.

2.3.3 Scena

Scena jest nierozłącznym elementem Unity, używanym do przechowywania obiektów składających się na całość lub część zawartości gry. W momencie załadowania w grze konkretnej sceny, załadowywana jest rówież cała jej zawartość. Mogą być one używane w dowolny sposób, zależny od potrzeb dewelopera i zaplanowanej architektury gry. Cała gra składać się może z jednej sceny, wielu scen rozdzielonych na poszczególne poziomy, lub każdy element gry może być osobną sceną.

2.3.4 Hierarchia

Istotnym elementem organizacji obiektów w Unity jest **hierarchia obiektów**, czyli możliwość umieszczania jednego obiektu jako dziecko drugiego. Głównym powodem takiego działania jest to, że obiekt dziecka zawsze zachowuje relatywną pozycję względem swojego rodzica, dzięki czemu zmiana pozycji rodzica o dany wektor zmienia również pozycję wszystkich dzieci o ten sam wektor. Zmiana pozycji obiektu dziecka natomiast, zmienia jego lokalną pozycję względem pozycji obiektu rodzica.

2.3.5 MonoBehaviour

MonoBehaviour to podstawowa klasa, po której domyślnie dziedziczy każdy skrypt stworzony w Unity.

W silniku Unity wszystkie metody zapewniane przez klasę MonoBehaviour mają swoją określoną kolejność wykonywania lub warunek konieczny do ich wykonania. Ich kolejność i cykl w jakim wykonują nazywa się Pętlą Gry. Najważniejsze lub najczęściej wykorzystywane w projekcie funkcje klasy MonoBehaviour to:

- **Awake** - wywoływane, gdy instancja skryptu zostanie załadowana do sceny. Dla danej instancji Unity wywołuje Awake tylko raz.
- **Start** - wywoływane jest po Awake, w pierwszej wygenerowanej klatce. Tak samo jak w przypadku Awake, zdarzenie Start wywoływane jest tylko raz dla danej instancji skryptu.
- **Update** - wywoływane jest w każdej wygenerowanej klatce, gdy skrypt jest aktywny. Częstotliwość wywoływania zdarzenia Update jest zmienna i zależy ona od optymalizacji gry oraz zasobów jakimi dysponuje sprzęt, na którym jest ona włączona.

- **FixedUpdate** - wywoływane jest w stałych interwałach czasowych ustawionych w systemie fizyki Unity. Używane jest ono do wszelkich kalkulacji fizyki w grze.
- **OnCollisionEnter** - wywoływane jest, gdy jeden Collider dotknie drugiego.
- **OnTriggerEnter** - wywoływane jest, gdy jeden Collider dotknie drugiego i co najmniej jeden z nich to Trigger.
- **OnTriggerExit** - wywoływane jest, gdy jeden Collider opuści drugi i co najmniej jeden z nich to Trigger.
- **OnDestroy** - wywoływane jest gdy obiekt lub scena zostaje zniszczona oraz gdy gra została wyłączona.

Klasa MonoBehaviour pozwala także na używanie korutyn, które omówione zostaną w następnym paragrafie.

2.3.6 Korutyny

Klasa MonoBehaviour pozwala również na rozpoczynanie, zatrzymywanie oraz zarządzanie **korutynami**, które pozwalają na pisanie asynchronicznego kodu. Mogą one zatrzymywać wywoływanie fragmentu kodu pod warunkiem czekania na upłynięcie określonego okresu lub wywołania odpowiednich akcji, pozwalając reszcie kodu na kontynuację swojego działania. Po spełnieniu określonych warunków, kod zostaje kontynuowany od momentu, w którym został zatrzymany.

2.3.7 Kamera

Komponent **kamery** służy do reprezentacji obiektów położonych w trójwymiarowej przestrzeni sceny, na dwuwymiarowym wyświetlaczu gracza. Posiada ona dwa tryby wyświetlania:

- perspektywistyczny, który symuluje zachowanie ludzkiego oka, zachowując głębie obrazu.
- ortograficzny, który nie zachowuje głębi obrazu, przez co wszystkie obiekty mają taką samą skalę, jednostki długości są takie same, a wszelkie linie równoległe pozostają równoległe.

Kamera może śledzić pewien obiekt, utrzymując go w środku wyświetlanego obrazu.

2.3.8 Cinemachine confiner

Cinemachine confiner odpowiedzialny jest za ograniczanie pozycji kamery do określonego obszaru.

2.3.9 Postprocessing

Kamery w Unity umożliwiają **Postprocessing**, czyli nakładanie wybranych efektów specjalnych na kamerę, które następnie wyświetlane są graczowi na ekranie.

2.3.10 Sprite

Sprite jest graficznym obiektem 2D. Aby zostać wygenerowane na scenie, załączany jest on do komponentu o nazwie Sprite Renderer, który następnie dołączany jest do obiektu.
Sprite renderer to komponent, który zapewnia możliwość umieszczania grafik w różnych warstwach. Mogą one być generowane przez kamerę w odpowiedniej kolejności, aby jedna była generowana nad drugą tym samym ją przysłaniając.

W omawianym projekcie wszystkie grafiki, w postaci plików .png, przygotowane zostały przez studenta Akademii Sztuk Pięknych im. Jana Matejki w Krakowie, Jaremę Chyrzyńskiego.

2.3.11 Światła

Aby wyliczyć zacienienie obiektu, Unity musi znać intensywność, kierunek oraz kolor światła na niego padającego. Informacje te dostarczane są przez **źródła światła**, czyli komponenty dołączone do obiektów na scenie. Mogą być punktowe, kierunkowe, lub powierzchniowe. Istnieje również komponent o nazwie ShadowCaster, dzięki któremu można ustalić kształt obiektu w kontekście rzucanego przez niego cienia.

2.3.12 Rigidbody

Rigidbody jest komponentem, który umożliwia włączenie zachowania opartego na fizyce silnika. Takimi zachowaniami mogą być: poruszanie obiektu, symulacja grawitacji na obiekcie oraz kolizje obiektów.

2.3.13 Colliders

Unity zarządza kolizjami między obiektami dzięki komponentowi **Collider**. Są one niewidzialne i definiującymi kształt obiektu dla zastosowań fizycznych kolizji. Mogą przyjmować one kształt w dwuwymiarze lub trójwymiarze, w zależności od naszych potrzeb.

2.3.14 Trigger

Specjalnym rodzajem komponentu zderzaczy Collider jest **Trigger**. Nie symuluje on zderzenia tak jak zwyczajny Collider, natomiast zachowując swój pierwotny kształt, informuje nas gdy inny zderzacz znajdzie się, pozostanie lub wyjdzie z obrębu jego objętości.

2.3.15 Audio Source

Audio Source to komponent, który odgrywa klip dźwiękowy w scenie. Może być on odgrywany globalnie, na całą scenę, lub punktowo z określonym zasięgiem.

2.3.16 Audio Listener

Audio Listener jest komponentem działającym na zasadzie mikrofonu. Odbiera on wszystkie dźwięki ze źródeł, w których zasięgu się znajduje, a następnie odgrywa je w zewnętrznym urządzeniu dźwiękowym gracza.

2.3.17 ScriptableObject

ScriptableObject jest wygondym w użyciu kontenerem, który może przechowywać duże ilości danych, niezależne od instancji klasy. Głównymi powodami ich stosowania, są redukcja zasobów pamięci zużywanych przez projekt oraz unikanie kopii wartości.

2.4 Oprogramowanie do realizacji rozgrywki sieciowej

Do umożliwienia graczom rozgrywki w sieci istnieje wiele narzędzi. W naszym projekcie wykorzystaliśmy darmowy serwis z usługą dla wielu graczy o nazwie Photon. Zapewnia on możliwość stworzenia dedykowanego serwera i zapewnia SDK kompatybilne z Unity, a konkretniej, paczkę do Unity o nazwie Photon Unity Networking (w skrócie Photon PUN), dzięki której z poziomu aplikacji można komunikować się z serwerem. Za pomocą tego frameworka gracze dołączają do oddzielnych pokoi zapewniających im możliwość rozgrywki. Photon obsługuje główny serwer, który jest odpowiedzialny za śledzenie liczby pokoi oraz graczy się w nich znajdujących, jak i osobne serwery dedykowane dla każdego pokoju z osobna, które obsługują poszczególne pokoje.

3. Implementacja mechanik rozgrywki

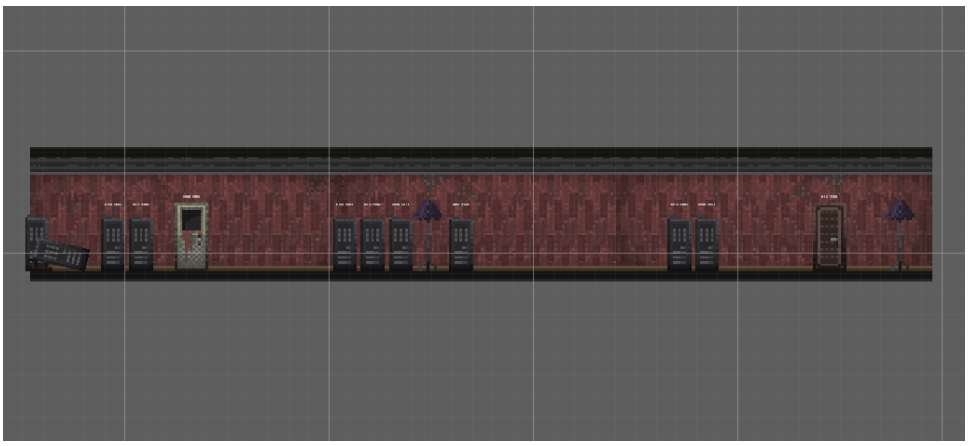
W tym rozdziale zostanie opisane, w jaki sposób zostały stworzone odpowiednie obiekty, a także jak zostały zrealizowane mechaniki wykorzystywane w grze. Elementy opisane w tym rozdziale zostały zaprojektowane wzorując się na wskazówkach przedstawionych w „The Fundamentals of Designing a Horror Game”. [5]

Logika każdej z wymienionych mechanik realizowana jest za pomocą skryptu napisanego w języku C#, dostosowanego do indywidualnych potrzeb każdego z nich. Są one napisane w taki sposób, aby przestrzegały zasad SOLID oraz zasad enkapsulacji - tym samym w jak najlepszym stopniu zapewniały możliwość dalszego rozbudowywania kodu, zapewniały jego czytelność oraz możliwość ponownego użycia lub ułatwiały wykrywanie potencjalnych błędów. Opisane w poniższym rozdziale elementy z wyłączeniem płyty naciskowej oraz drzwi na płytę naciskową stworzył Marcel Cisowski.

3.1 Sceny

Sceny są to zasoby zawierające całość lub część gry, zawierające przygotowane wcześniej zestawienia obiektów gry. Ich zastosowanie może być dowolne, a samych scen w projekcie można stworzyć nieograniczoną ilość. Sceny można w prosty sposób zmieniać, jednocześnie zmieniając aktualnie pokazywaną zawartość graczowi.

W naszym projekcie scen używamy przede wszystkim do rozdzielenia zawartości gry na mniejsze, osobno załadowywane fragmenty. Reprezentują one zatem kolejne poziomy do przejścia w grze. Każdy taki poziom zawiera odpowiednio rozmieszczone w przestrzeni obiekty reprezentujące przedmioty, kryjówki, potwory oraz inne elementy potrzebne do poprawnego działania etapów. Na takich scenach są również rozstawione pojemniki oraz kamery, które włączają lub wyłączają się w zależności po której części sceny porusza się gracz.



Rys. 3.1: wygląd przykładowej sceny z naszej gry z etapem w edytorze Unity

Dodatkowo ze scen korzystamy do oddzielenia zawartości gry związanej etapami oraz innych funkcjonalności gry. W omawianym projekcie scenami nie będącymi poziomami do przejścia są:

- Menu główne, scena na której za pomocą przycisków możemy rozpocząć grę lub wyjść z aplikacji.
- Scena łączenia z serwerem, opisana w rozdziale 4.1.1
- Lobby, scena opisana w rozdziale 4.1.2

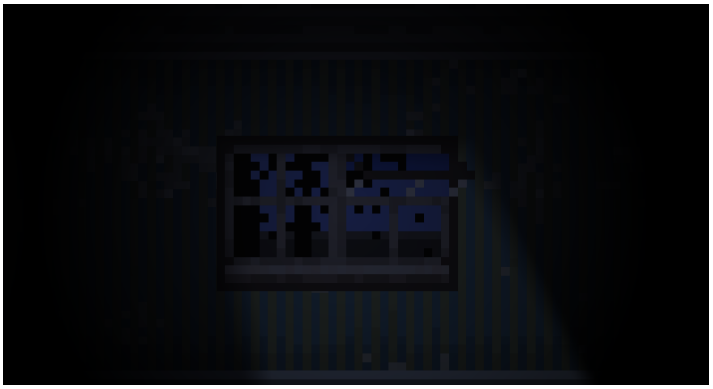
3.2 Oświetlenie

Elementami oświetlającymi sceny są różnego rodzaju lampy, okno, latarka gracza oraz sam gracz emituje niewielkie światło symulujące jego wizję. Bez nich wszystkie elementy sceny byłyby kompletnie czarne. Tekstury widoczne są jedynie wtedy, gdy są w zasięgu któregoś ze źródeł światła. Wszystkie elementy oświetlenia są zatem obiektami z komponentem źródła światła lub posiadają ChildObject z tym komponentem, natomiast w zależności od potrzeb w każdym z nich zmieniane są między innymi:

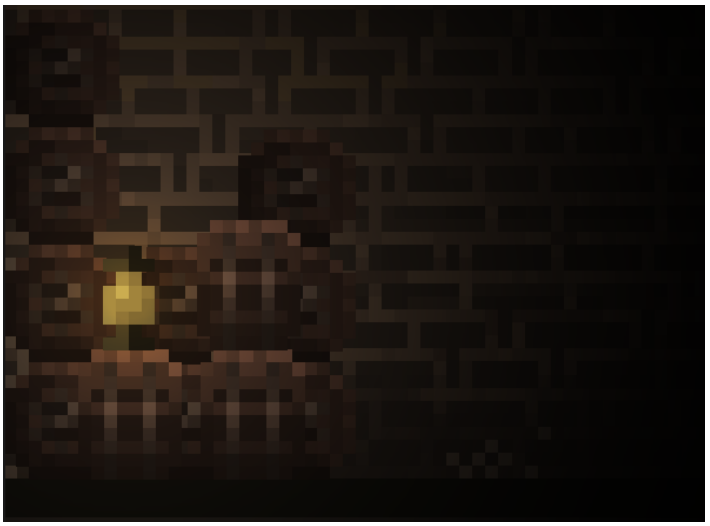
- kolor
- intensywność
- kąt oświetlania
- zasięg światła

Dzięki modyfikacji tych elementów wraz z odpowiednimi teksturami w grze oświetlenie tworzy ponurą, tajemniczą atmosferę pożądaną w grach z motywami horroru.

Lampy oraz okno w grze są osobnymi obiektami nie zmieniającymi swojej pozycji na scenie. Wyjątkiem jest latarka oraz wizja gracza, ponieważ każda z nich jest ChildObject protagonisty, aby przemieszczały się wraz z nim.



Rys. 3.2: Element oświetlenia w postaci okna



Rys. 3.3: Element oświetlenia w postaci lampy naftowej ustawionej na beczkach

3.3 Dźwięki

Kolejnym istotnym elementem gry, pozwalającym na budowanie napięcia są dźwięki zaimplementowane do rozgrywki. Możemy je podzielić na dwie kategorie:

- dźwięki ciągłe, na które składa się muzyka, lub tło muzyczne w postaci dźwięków imitujących obecność w starym domu. Pozwalają one częściowe odwzorowanie otoczenia oraz budowanie napięcia, które jest bardzo istotne w grach z elementami horroru.

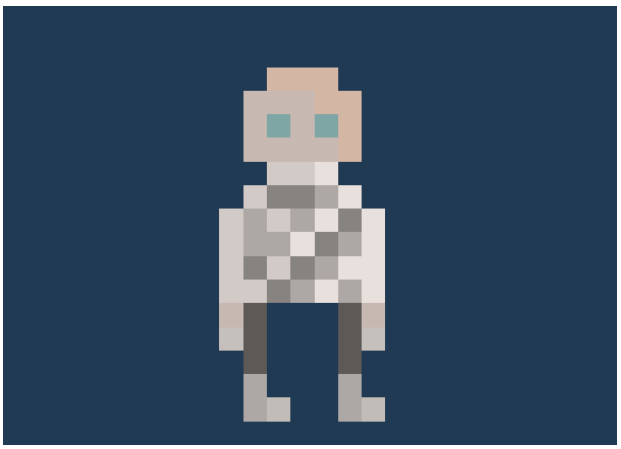
- dźwięki chwilowe, wyzwalane jedynie w momencie zaistnienia danego zdarzenia, składają się na nie dźwięki wydawane przez protagonistę, potwora oraz wszystkie przedmioty znajdujące się w projekcie. Dodają one do gry wrażenie większej interaktywności oraz pozwalają na wystraszenie gracza.

3.4 Protagonista

W gatunku gier przygodowych osoba grająca najczęściej wciela się w jakąś postać, kieruje nią i może dokonywać decyzji związanych z owym charakterem. W projekcie gracz za pomocą klawiatury steruje **protagonistą**, który może poruszać się, wchodzić w interakcję z innymi obiektami w grze oraz zapalać lub gasić latarkę. Jest on zatem najważniejszym i jednocześnie najbardziej rozbudowanym obiektem w całym projekcie.

Gracz sterowanym protagonistą może wykonać następujące akcje:

- poruszanie się
- podnoszenie przedmiotów do ekwipunku
- wyrzucanie przedmiotów z ekwipunku
- przechodzenie przez drzwi
- odblokowywanie drzwi zamkniętych na klucz
- otwieranie zamkniętych drzwi
- otwieranie zamkniętych drzwi płytką naciskową
- chowanie się w kryjówkach
- wchodzenie i wychodzenie z kanałów wentylacyjnych
- zapalanie i gaszenie latarki



Rys. 3.4: Graficzna reprezentacja protagonisty

Obiekt protagonisty może wykonywać powyższe czynności dzięki dołączonym do jego niego skryptów. Ponieważ akcji, które może wykonać gracz, jest wiele, kod związany z logiką ich wykonywania w jednym skrypcie byłby bardzo obszerny pod kątem liczby linijek kodu. Aby rozwiązać ten problem, zaimplementowany został wzorzec projektowy stanu dla każdej interakcji z osobna. Dodatkowo, dzięki temu w prosty sposób w razie potrzeby można dodać nowe stany, nie zmieniając poprzednio napisanego kodu. Protagonista posiada maszynę stanów, która odpowiedzialna jest za ich zmianę oraz wykonywanie logiki tego, w którym aktualnie znajduje się gracz.

Każdy **stan** jest klasą dziedziczącą po abstrakcyjnej klasie bazowego stanu i nadpisuje wszystkie jego abstrakcyjne metody odpowiedzialne za:

- wejście do stanu
- aktualizację stanu co klatkę
- kolizję z Collider
- kolizję z Trigger

Stany zmieniane są wewnątrz maszyny stanów, natomiast dzięki zastosowaniu słownika w którego kluczem jest publiczny typ wyliczeniowy, a wartością klasa stanu, można zainicjować zmianę stanu w maszynie stanów z wnętrza każdego innego stanu, w zależności od spełnionych w kodzie warunków.

Elementem każdego protagonisty jest wspomniana w rozdziale 3.2 latarka. Jest to jedyny element oświetlenia który można włączać i wyłączać. Dodatkowo działa ona przez określony w skrypcie czas. Po upływie tego czasu, latarka przestaje świecić a do ponownego jej włączenia konieczna jest bateria. Poziom zużycia baterii w latarce jest reprezentowany w interfejsie użytkownika.

Ekwipunek protagonisty to lista przedmiotów do zbierania (zobacz 3.8), w której przechowywane są informacje na ich temat. Każdy protagonista posiada także reprezentację graficzną swojego ekwipunku w interfejsie użytkownika. Jest to lista przechowująca dane o przedmiotach do zbierania z którymi gracz wszedł w interakcję, tym samym powodując ich zniknięcie ze sceny. Ekwipunek, a właściwie jego zawartość jest istotna podczas działania niektórych czynności jak otwieranie drzwi lub do wymiany baterii w latarce. Liczba przedmiotów w ekwipunku gracza jest ustalona przez nas na trzy, jeśli gracz spróbuje wejść

w interakcję z kolejnym przedmiotem do zbierania nie zostanie od zapisany w liście. Gracz może również wyrzucić przedmiot z ekwipunku, powodując jego pojawienie się na scenie w miejscu w którym aktualnie znajduje się jego protagonista.

Gracz porusza się protagonistą, wykonuje interakcje z innymi obiektami oraz zapala lub gasi latarkę za pomocą klawiszy na klawiaturze. Za wykrywanie wciśnięcia, przytrzymania oraz puszczenia odpowiednich klawiszy odpowiedzialny jest osobny skrypt o nazwie **Input Manager** dołączony do obiektu gracza. Dzięki zawartych w nim zmiennym, informację o odpowiednim stanie klawisza można w prosty sposób przekazać do innych skryptów. Za informacje o kierunku poruszania się odpowiada zmienna typu integer, która przyjmuje wartość 0 w spoczynku, 1 dla poruszania się zgodnie z osią X i -1 dla przeciwnego kierunku. Następnie w stanie odpowiedzialnym za poruszanie się obiektowi protagonisty nadajemy wektor prędkości ze współrzędną X przemnożoną przez wartość tej zmiennej. Informacje o pozostałych akcjach są przekazywane za pomocą zmiennych typu bool.

3.5 Potwór

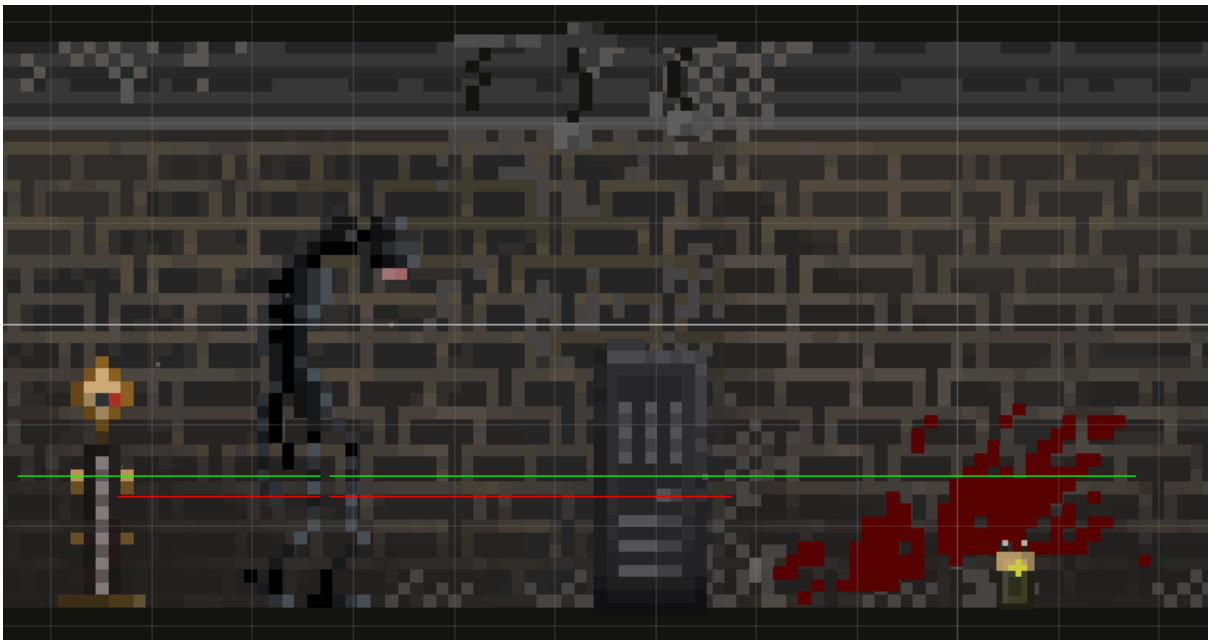
W grach komputerowych, aby rozgrywka była dynamiczna, zapewniała wyzwanie lub wywoływała zaplanowane emocje, wprowadza się postaci przeciwników, które utrudniają graczom wykonywanie określonych zadań.

W projekcie taką rolę pełni **potwór**, i jest to jedyny element rozgrywki, który może ją przedwcześnie zakończyć i zmusić graczy do rozpoczęcia obecnego poziomu od początku. Jest to zatem element, który buduje napięcie, zmusza gracza do ostrożniejszego podejmowania decyzji i w połączeniu z klimatem zapewnionym przez działanie świateł oraz postprocessing może wywołać nawet strach, co jest oczywiście pożądanym efektem w grze z motywami horroru. Zachowanie potwora opiera się na trzech stanach:

1. **Stan patrolowania** - stan, w którym potwór spokojnie przemieszcza się pomiędzy określonymi punktami spoczynku, poszukując po drodze gracza.
1. **Stan spoczynku** - stan, w którym znajduje się potwór podczas przebywania w punkcie spoczynku. Trwa on kilka sekund i podczas jego trwania potwór pozostaje czujny i wyszukuje gracza w pobliżu.
2. **Stan pogoni** - stan, w którym znajduje się potwór po wykryciu gracza. W momencie rozpoczęcia pogoni, potwór przygotowuje się do biegu, a następnie ze zdwojoną prędkością rozpoczyna bieg w kierunku, gdzie znajdował się gracz w momencie wykrycia. Istnieją dwa warianty zakończenia pogoni, pierwszym jest kolizja potwora

z graczem, co odpowiada jego złapaniu i jest równoznaczne z ponownym rozpoczęciem poziomu. Drugim jest zniknięcie gracza z pola widzenia potwora. W momencie, gdy potwór przestanie wykrywać gracza, będzie kontynuować swój bieg w kierunku, w którym ostatnio widoczny był gracz jeszcze przez kilka sekund, aż zatrzyma się i wróci do patrolowania.

Wykrywanie gracza odbywa się na zasadzie wystrzeliwania ze środka obiektu potwora laserów w obie strony. Jeśli któryś z laserów trafi obiekt gracza i spełni odpowiednie warunki, potwór rozpoczyna pogoń. Długość laserów zostaje wydłużona, jeśli potwór znajduje się w stanie pogoni lub gracz znajdujący się w pobliżu ma włączoną latarkę. Zielony laser odpowiada za wykrywanie graczy z włączoną latarką, czerwony natomiast wykrywanie gracza z wyłączoną latarką, biały natomiast za wykrywanie gracza w stanie pogoni.



Rys. 3.5: Potwór oraz graficzna reprezentacja laserów w edytorze Unity

Każdy potwór na początku rozgrywki jest nieaktywny. Aktywowany zostaje dopiero w momencie otwierania się drzwi do pokoju, w którym się znajduje. Rozwiązanie to służy kontrolowaniu pozycji potwora w pomieszczeniach, a jego celem jest uniknięcie sytuacji, w której gracz, nie widząc co jest po przeciwnej stronie drzwi, wszedł przez nie do pokoju i natknął się od razu na potwora stojącego za nimi.

Kolejnym elementem dołączonym do obiektu potwora jest Trigger oddziaływujący na kamerę i wyzwalający na niej filtry zaginające, rozmywające oraz zmieniające kolor obrazu renderowanego przez kamerę. Zastosowanie tych filtrów posiada w projekcie dwa

zastosowania. Pierwszym z nich jest informowanie gracza o zbliżającym się zagrożeniu zagrożeniu ze strony potwora, który może być niewidoczny w ciemności, jeśli gracz nie ma włączonej latarki. Drugim zastosowaniem jest wyzwalanie u gracza niepokoju i stresu poprzez trudności w zrozumieniu co dzieje się na ekranie.

Rys. 3.6: Działanie filtrów kamery wyzwalanych przez potwora znajdującego się w bliskiej odległości.

3.6 Kryjówki

Jedną z możliwości ukrycia się przed szukającym lub goniącym nas potworem jest użycie którejś z wielu **kryjówek** rozmieszczonych w pokojach. Wyróżniamy 4 rodzaje kryjówek:

- szafkę
- szafę
- długi stół (wkleić zdjęcia)
- krótki stół

Kryjówki sprawiają, że jesteśmy kompletnie niewidoczni dla potwora, sekundę po wejściu do jednej z nich.

W momencie gdy jesteśmy schowani, potwór może obok kryjówki, patrolować oraz biegać, a protagonista nie zostanie przez niego zauważony. Dzieje się tak dzięki wyłączeniu jego

odpowiednich komponentów za pomocą skryptu PlayerStateHiding, gdy ten ukrywa się. Istnieje jednak możliwość, aby gracz został wykryty wewnątrz kryjówki, jeśli wejdzie do niej z włączoną latarką lub włączy ją po wejściu.
W każdej kryjówce znajdować się może wyłącznie jeden gracz, więc w momencie wystąpienia zagrożenia w postaci napotkania potwora, każdy gracz musi znaleźć kryjówkę dla siebie.

3.7 Drzwi

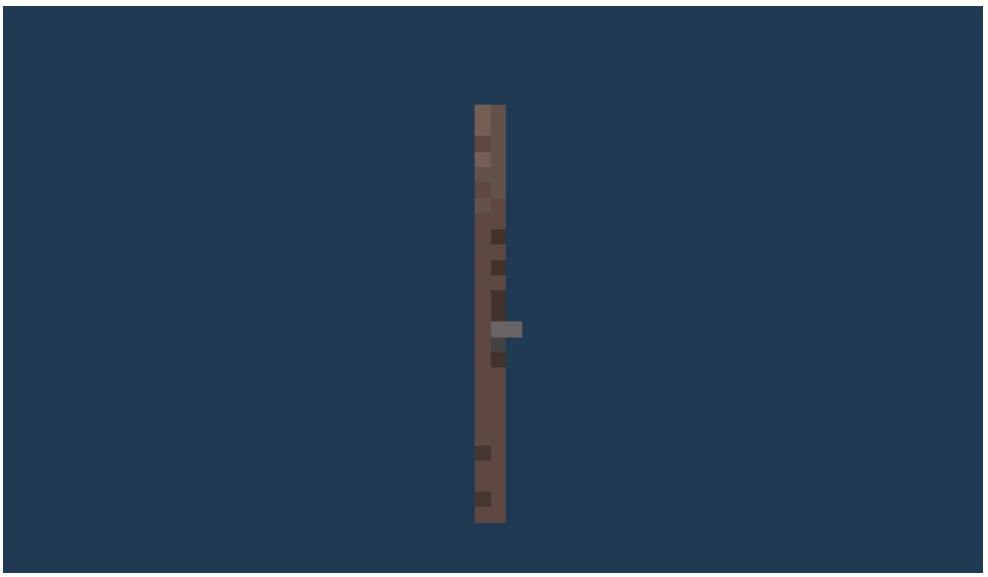
Sceny, w których odbywa się rozgrywka przygotowanym przez nas stylem graficznym oraz układem odzwierciedlają scenerię, przypominającą wnętrze nawiedzonej lub opuszczonej posiadłości, tajemniczej i pełnej zagadek. Każda z takich scen dzielona jest na pomieszczenia, a te oddzielone są od siebie drzwiami. To właśnie one są elementem który pozwala na wcześniejsze zaprojektowanie ścieżki którą musi podążać gracz aby dojść do celu. Odblokowywanie może wymagać posiadania konkretnego klucza (w przypadku HorizontalDoors opisane w 3.5.1 lub VerticalDoors opisane w 3.5.2) lub użycia płytki naciskowej (PressurePlateDoors opisane w 3.5.3) Pozwala to na odpowiednie zaprojektowanie poziomów aby gracz w celu przejścia danego fragmentu gry musiał wejść w interakcję z odpowiednimi elementami gry, co przyczynia się do konieczności odkrywania nowych lokalizacji, czym cechują się gry z gatunku przygodowych. Jest to również, tak jak kryjówki, element mogący zapewnić bezpieczeństwo przed potworem, który nie potrafi przechodzić przez drzwi. Zasada działania drzwi opiera się na włączaniu komponentu Collider kiedy są zamknięte lub wyłączaniu go kiedy są otwarte. Ze względu na styl graficzny 2D oraz charakter rozgrywki polegający na kooperacji dwóch graczy, w grze zaimplementowane zostały różne rodzaje drzwi których logika różni się między sobą. Poniżej zostaną opisane ich rodzaje oraz zastosowania.

3.7.1 Drzwi horyzontalne (HorizontalDoors)

Pierwszym rodzajem drzwi są nazwane przez nas drzwi horyzontalne które są przedstawione jako drzwi na które patrzy się z perspektywy wnętrza ściany. Jest to obiekt mający na celu oddzielenie dwóch pomieszczeń które sąsiadują między sobą w osi X przestrzeni w której ulokowany jest każdy GameObject. Najważniejszymi elementami drzwi horyzontalnych odpowiedzialnymi za ich działanie są:

- ➢ Komponent Collider ustawiony jako Trigger
- ➢ ChildObject posiadający Collider odpowiadający za kolizję z innymi obiektami
- ➢ Dwa puste ChildObject z odpowiednimi ustawieniami Transform
- ➢ Komponent skryptu o nazwie DoorScript

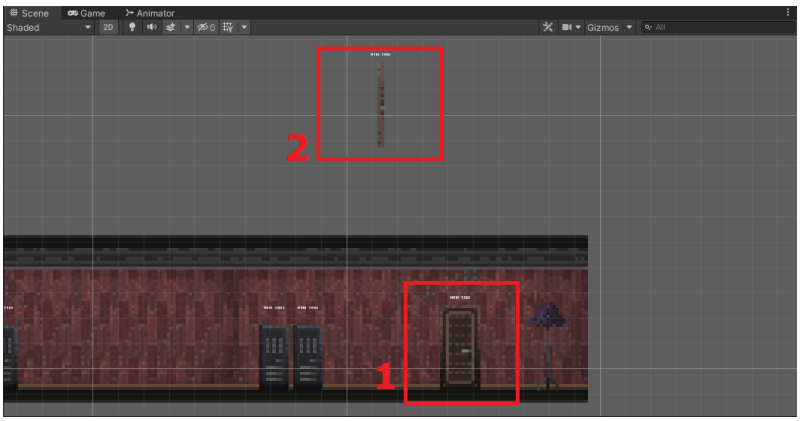
Skrypt HorizontalDoorScript jest komponentem trzymającym informacje o obecnym stanie drzwi za pomocą zmiennych publicznych. Jedna zmienna typu bool określa czy drzwi są zamknięte na klucz czy odblokowane. Jeśli drzwi są zablokowane, w momencie próby interakcji sprawdzany będzie ekwipunek gracza w poszukiwaniu przedmiotu o odpowiednim ID do ich odblokowania. ID takiego przedmiotu również przechowywane jest jako publiczna zmienna int w skrypcie HorizontalDoorScript. Gracz posiadając taki przedmiot i wciskając klawisz interakcji wywoła funkcję która sprawdza czy klawisz interakcji będzie odpowiednio długo trzymany i po odpowiednim czasie odblokuje drzwi. Dzięki komponentowi Collider ustawionemu jako Trigger, funkcja OnTriggerExit może restartować czas odblokowywania drzwi przez gracza gdy ten odsunie się zbyt daleko opuszczając jego obszar. Puste ChildObject symbolizują punkt po lewej oraz prawej stronie drzwi. Transform tych punktów wykorzystywany jest do przenoszenia gracza w odpowiednie miejsce podczas wykonywania logiki stanu PlayerStateUsingHorizontalDoors.



Rys. 3.7: Wygląd drzwi horyzontalnych

3.7.2 Drzwi Wertykalne (VerticalDoors)

Ten rodzaj drzwi łączy pomieszczenie, które wizualnie mieściłoby się w głąb tego, w którym aktualnie przebywa gracz. Wchodząc do takiego pomieszczenia, chcąc zachować styl wizualny, gry postanowiliśmy połączyć je z Drzwiami Horyzontalnymi, aby gracz z powrotem przeniesiony został do pomieszczenia w którym poruszać się będzie wzdłuż osi X. W rzeczywistości jednak przenoszony on jest jednocześnie w inne miejsce na osi Y. Dzięki takiemu rozwiązaniu pomieszczenia po jakich porusza się gracz nie muszą być ustawione w jednej linii. Elementy potrzebne do działania tego typu drzwi są bardzo podobne do poprzedniego typu drzwi. Drzwi Wertykalne (na obrazku oznaczone numerem 1) również posiadają Komponent Collider ustawiony jako Trigger i komponent skryptu o nazwie DoorScript, a zatem zasada działania jest niemal identyczna. W odróżnieniu do poprzedniego typu, te posiadają też Drzwi Horyzontalne jako ChildObject (na obrazku oznaczone numerem 2), a do przemieszczania protagonisty w odpowiednie miejsce wykorzystywane są trzy puste ChildObject z odpowiednio ustawionymi komponentami Transform (te w Drzwiach Horyzontalnych są wówczas wyłączone).



Rys. 3.8: Wygląd drzwi wertykalnych. 1 - ParentObject, 2 - ChildObject

3.7.3 Drzwi na płytę naciskową i płyta naciskowa (PressurePlateDoors i PressurePlate)

Jest to rodzaj drzwi który wymaga współpracy graczy do ich otwarcia i zostały one zaprojektowane ze względu na rozgrywkę wieloosobową. Drzwi te nie mogą zostać otwarte za pomocą klawisza interakcji, a jedynym sposobem na ich otwarcie jest znalezieniu się gracza na odpowiedniej płycie naciskowej. Płyta naciskowa to GameObject który za pomocą Collidera ustawionego na Trigger w skrypcie PressurePlateScript za pomocą funkcji OnTriggerEnter i OnTriggerExit wykrywa liczbę stojących na niej rotagonistów. Jeśli liczba

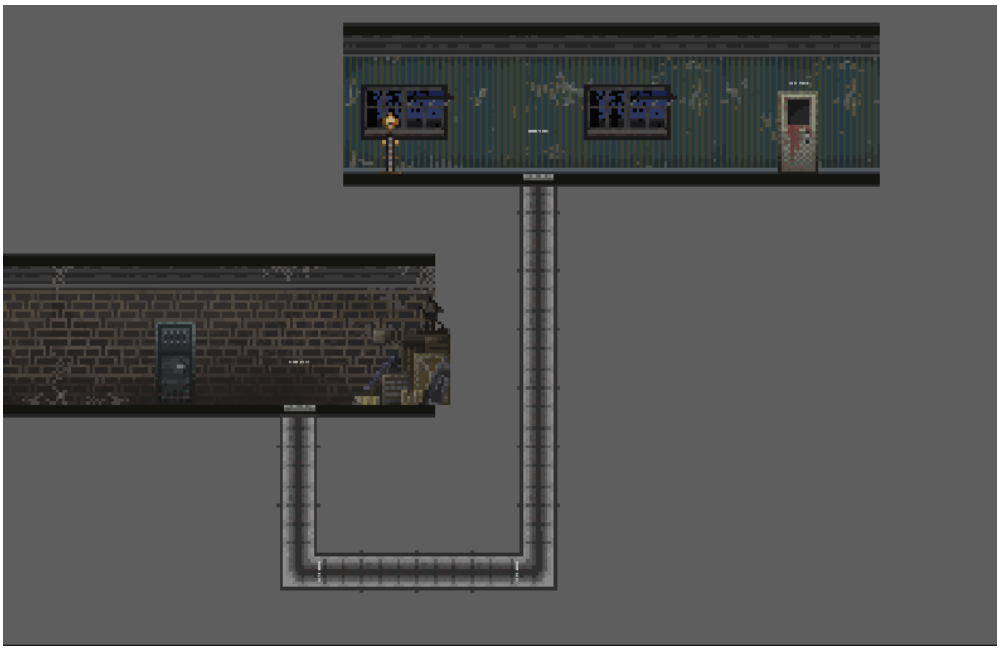
ta jest równa 0, drzwi będą zamknięte, a jeśli będzie większa od zera drzwi zostaną otwarte. Drzwi na płytę naciskową implementują wzorzec projektowy o nazwie obserwator, a komunikacja między nimi a płytą odbywa się za pomocą zdarzeń w języku C#. Dzięki takiemu rozwiązaniu logika płyty naciskowej z inną grafiką mogłaby również służyć do wywoływania innego zdarzenia niż otwieranie drzwi.

Rys. 3.9: Wygląd płyty naciskowej

3.8 System Wentylacji

System wentylacji w grach oraz filmach jest częstym obiektem po którym ktoś może się poruszać. Dodatkowo osoba taka najczęściej stara się gdzieś zakraść niezauważenie lub schować, dlatego nadaje się do gry z motywami Horroru. W naszym projekcie System Wentylacji umożliwia graczom wchodzenia i wychodzenia z jego środka, umożliwiając przy tym zmianę pokoju w orientacji zgodnej z osią X lub zgodnej z osią Y. Najważniejszym elementami Systemu Wentylacji są jego otwory (w tym rozdziale dalej zwane wejściami lub wyjściami) i to w nich zawarta jest cała logika. Do obiektu wejścia dodany jest między innymi komponent skryptu o nazwie LadderScript i przechowuje on publiczne zmienne które wpłyną na zachowanie się protagonisty w stanie PlayerStateUsingLadder. Jedną z nich jest zmienna bool która określa czy obiekt do którego podpięty jest skrypt jest wejściem. Jeśli tak, będzie to miało wpływ na animator protagonisty - w wentylacji stany bezczynności i poruszania się mają inną animację niż poza nią. Również skrypt przechowuje publiczne zmienne typu transform które wyznaczają na jaką wysokość ma zostać przeniesiony protagonista przy zmianie współrzędnej Y swojego obiektu.

Rys. 3.10: Wygląd systemu wentylacji w grze, ukazujący zmienioną animację gracza

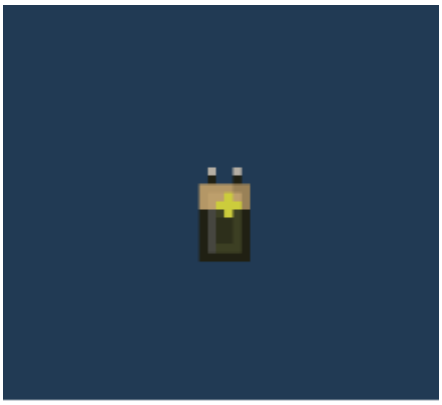


Rys. 3.11: Wygląd wentylacji w edytorze Unity

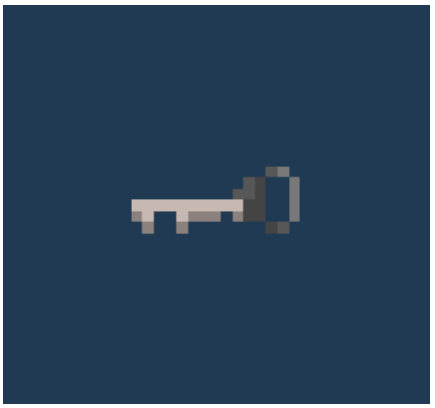
3.9 Przedmioty do zbierania

Przedmioty do zbierania są bardzo częstym elementem gier, mogą one mieć bardzo różne zastosowania. Są więc bardzo elastycznym elementem który może być zastosowany w niemal każdym gatunku, w szczególności w grze przygodowej. Zwykle są one ściśle związane z ekwipunkiem gracza, i tak jest również w przypadku tego projektu. Każdy przedmiot w projekcie posiada komponent w postaci skryptu który implementuje interfejs IPickableObject. Dzięki temu muszą posiadać konkretne zmienne lub lub metody ale mogą

mieć również swoją unikalną logikę. Każdy przedmiot musi posiadać między innymi swój identyfikator w postaci zmiennej typu integer, lub nazwę do wyświetlania w elementach interfejsu użytkownika w postaci zmiennej typu string. Interfejs wymaga również zaimplementowanie metod do ustawienia nowej pozycji przedmiotu i do jego zniszczenia. Są one wywoływane, gdy gracz zdecyduje wyrzucić przedmiot z ekwipunku lub gdy przedmiot zostanie zużyty.



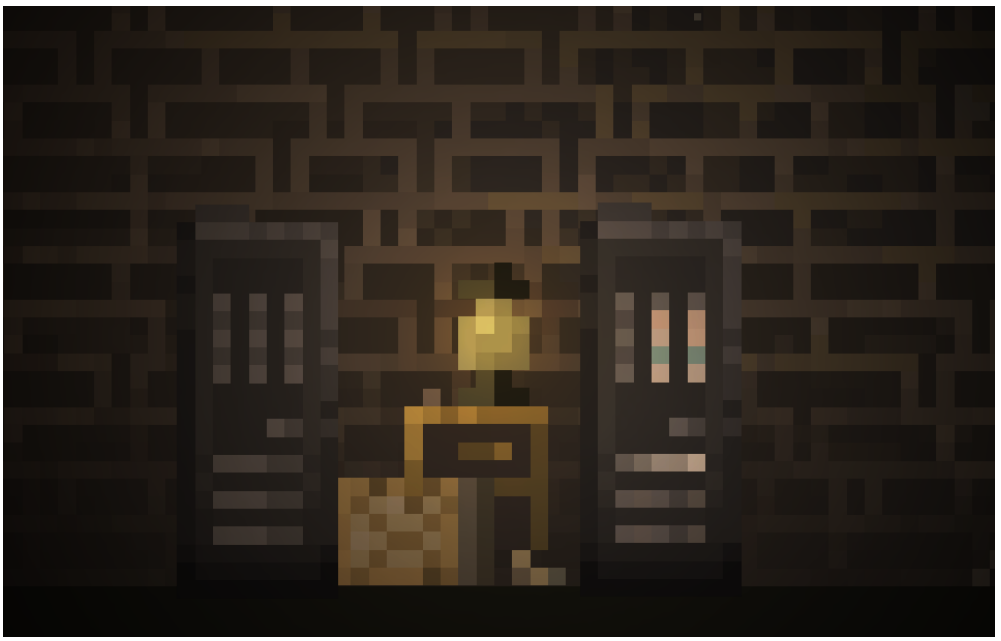
Rys. 3.12: Wygląd baterii



Rys. 3.13: Wygląd klucza

3.10 Czytelność mechanik i interfejs użytkownika

W projekcie zostało zastosowane wiele elementów, które oprócz wprowadzania pewnej funkcjonalności mają za zadanie ułatwiać graczowi zrozumienie pewnych mechanik w grze. Jednym z podstawowych elementów, które wydają się nieodłączną częścią gry są animacje postaci i przedmiotów. W przypadku protagonistów są one konieczne do informowania gracza, w jakim stanie znajduje się każdy z nich. Identyczny wygląd postaci podczas spoczynku, poruszania się oraz stanie ukrycia byłby mylący dla graczy, a podczas każdego z nich gracz powinien spodziewać się nieco innego działania pewnych mechanik. Obiekty, które mają złożone mechaniki lub kilka animacji mają rozdzieloną logikę działania od części graficznej. Na poniższym rysunku dzięki odpowiednim animacjom widać różnicę, w której kryjówce znajduje się protagonista.



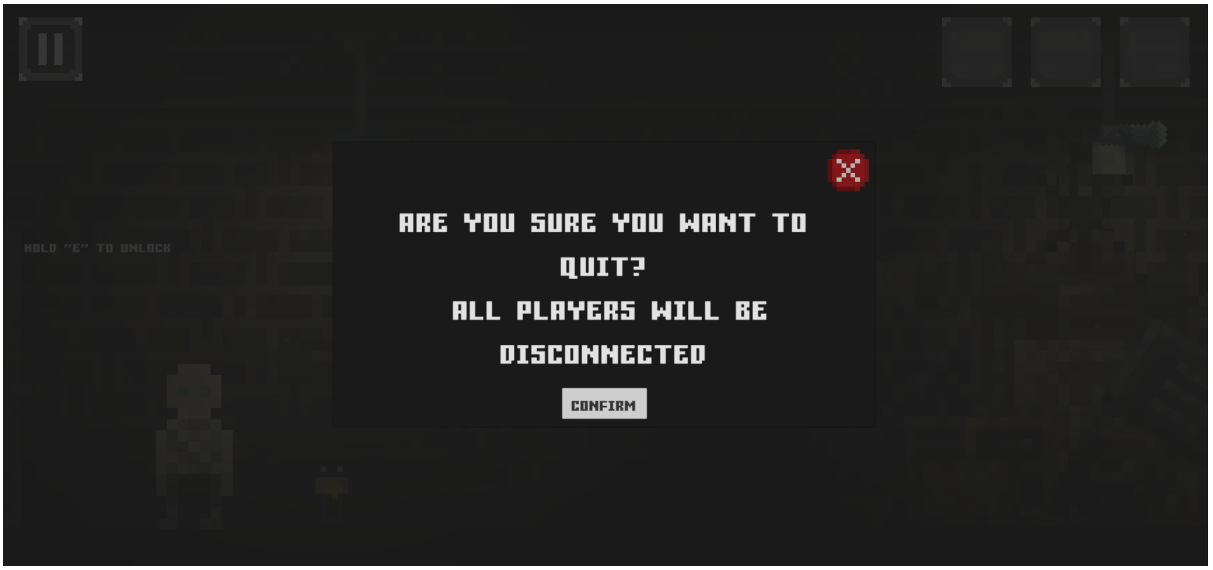
Rys. 3.14: Dwie kryjówki. Kolejno od lewej: pierwsza pusta, druga z protagonistą w środku

Interfejs użytkownika to część gry, z którą gracz wchodzi w interakcje. Skupia się ona na przekazywaniu informacji, mówiącej o tym jaki będzie rezultat wykonanej przez niego czynności. Interfejs użytkownika w omawianym projekcie można rozdzielić na dwie główne części - HUD (z angielskiego Heads-up display) oraz części interfejsu na scenach z etapami do przejścia.

HUD to panel zawierający obrazki, przyciski, które nie są częścią etapu a on sam jest wyświetlany niezależnie od aktualnie używanej kamery. W jego skład wchodzą:

1) Przycisk pauzy, który po jego naciśnięciu pokazuje menu gry. Menu gry posiada przyciski do kontynuowania rozgrywki (czyli zamknięcia menu), przycisk wyjścia do menu, oraz przycisk wyjścia z aplikacji. Aby uniknąć przypadkowego wyjścia dodany został panel z koniecznością potwierdzenia wyboru opuszczenia rozgrywki.
2) Tekst informujący o przedmiocie do zbierania, wyświetlany na określony czas po jego podniesieniu.
3) Obrazki reprezentujące ekwipunek. Są one aktualizowane wraz ze zmianą listy przedmiotów do podnoszenia którą posiada protagonista.
4) Ikona latarki oraz poziom jej zużycia. Gdy latarka jest włączona przez gracza pojawia się indykator, który radialnie zanika wraz z zużyciem baterii.

Rys. 3.15: Elementy HUD



Rys. 3.16: Panel potwierdzający chęć zakończenia rozgrywki

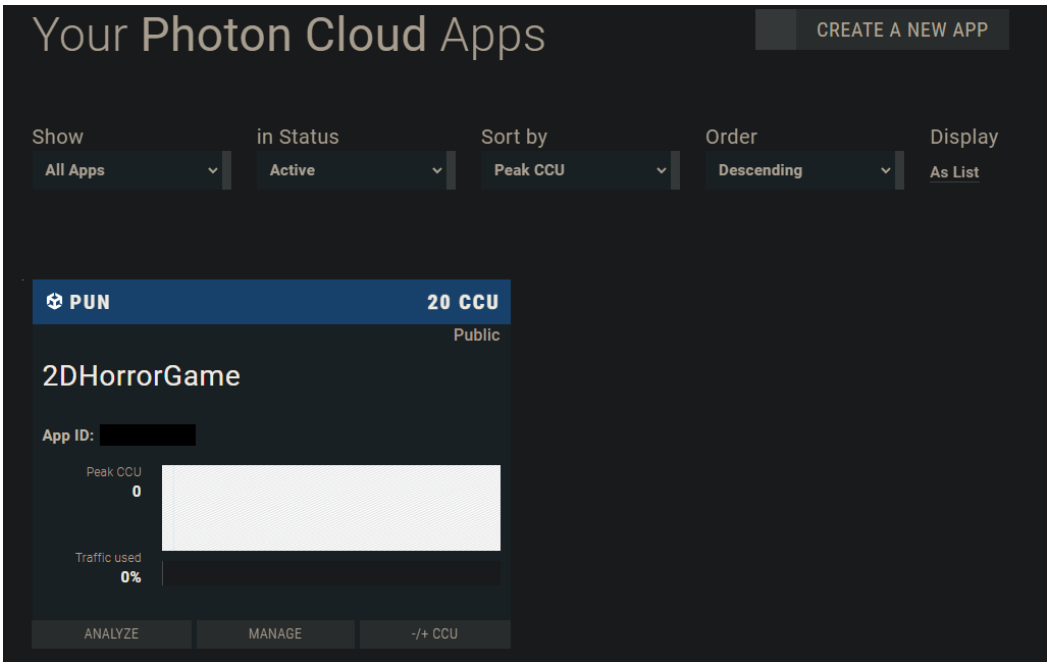
Częścią interfejsu jest stworzone przez nas **Podświetlenie Interakcji**. Jest to element interfejsu przytwierdzony do przedmiotów w grze. Gdy gracz znajduje się w określonej przez wyzwalacz odległości, przedmiot lekko podświetla się oraz wyświetlany jest odpowiedni tekst pasujący do interakcji. Ma on na celu naprowadzenie gracza na możliwość dokonania interakcji z podświetlonym przedmiotem.

Rys. 3.17: Podświetlenie Interakcji przytwierdzony do drzwi (po lewej) i do baterii (po prawej)

4. Implementacja rozgrywki sieciowej

Zgodnie z założeniem naszego projektu gra powinna zapewniać możliwość rozgrywki graczom przez internet w trybie kooperacji, czyli takim gdzie gracze muszą wspólnie wykonywać określone zadania. W tym celu został użyty Photon Unity Network. Za część projektu opisaną w poniższym rozdziale odpowiedzialny był Bartosz Jaromin.

W celu integracji biblioteki Photon.Pun należało i korzystania z serwera w pierwszej kolejności należało stworzyć konto w serwisie Photon, dodać aplikację wybierając odpowiednie ustawienia i otrzymać identyfikator aplikacji. [6] Służy on do rozróżniania aplikacji działających w sieci serwera Photon. Dzięki temu infrastruktura serwera Photon kieruje ruch sieciowy do właściwej aplikacji i efektywnie zarządza zasobami każdej aplikacji. Na poniższym rysunku widać panel w serwisie Photon. Identyfikator aplikacji ze względów bezpieczeństwa został zamazany.



Rys. 4.1: Panel aplikacji w serwisie Photon

Dalsza część rozdziału w szczegółowy sposób opisuje w jak udostępnione przez bibliotekę Photon.Pun narzędzia zostały wykorzystane do zapewnienia graczom rozgrywki przez sieć w czasie rzeczywistym.

4.1 Początkowe obsługiwanie graczy

Po uruchomieniu programu z grą sceną która zostanie załadowana jest menu główne. Gra uruchamiana jest lokalnie na urządzeniu gracza, zatem przed załadowaniem etapów w celu umożliwienia rozgrywki wieloosobowej, należy nawiązać połączenie z serwerem, stworzyć odpowiednie sesje a następnie w odpowiedni sposób klienci muszą wymieniać między sobą najważniejsze dane o podejmowanych w grze akcjach. W tym rozdziale opisane zostanie opisany sposób implementacji wyżej wymienionych czynności.

4.1.1 Łączenie z serwerem i lobby

Aby zacząć grę w trybie wieloosobowym w pierwszej kolejności nasza gra musi połączyć się z głównym serwerem. [7] W tym celu została stworzona scena ekranu ładowania, na której między innymi znajduje się obiekt gry z kodem odpowiedzialnym za tę funkcjonalność. Znajduje się na niej obiekt posiadający skrypt z klasą o nazwie ConnectToServerScript. Klasa ta dziedziczy po MonoBehaviourPunCallbacks dostarczaną przez bibliotekę Photon.Pun. Dzięki temu poprzez skrypt za pomocą metody ConnectUsingSettings() wywoływanej w metodzie Start() możemy połączyć się z głównym serwerem gry przy załadowaniu sceny.

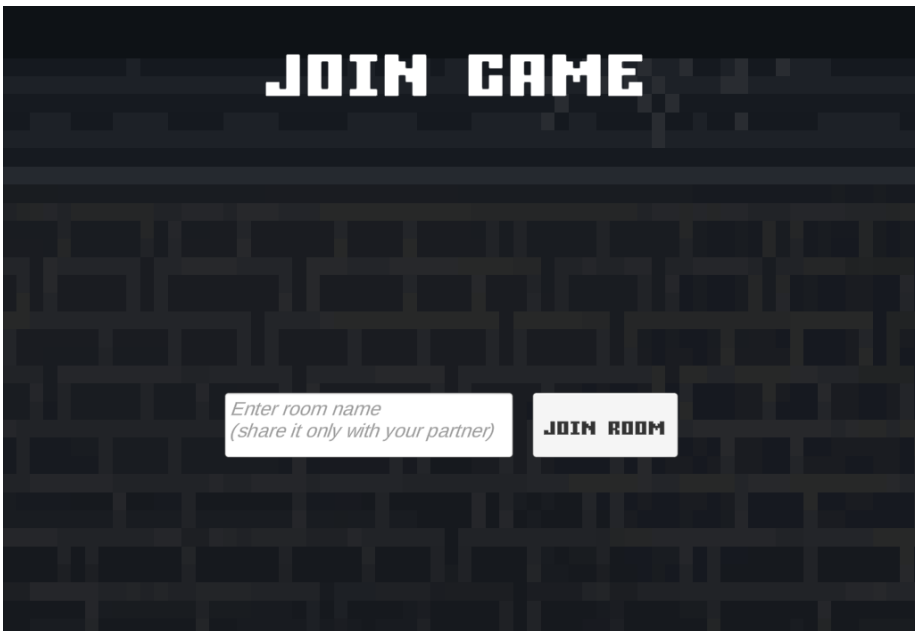
Rys. 4.2: Ekran wyświetlany podczas łączenia z serwerem

Jeśli połączenie będzie udane, klient zostanie połączony z lobby w którym znajduje się lista wszystkich pokoi. [8] Dzięki nadpisanej funkcji OnJoinedLobby(), która wykonuje się po udanej próbie dołączenia gracza do lobby, gracz przenoszony jest na kolejną scenę

reprezentującą lobby opisaną w rozdziale 5.1.2. Proces ten nie ma wizualnej reprezentacji w grze, a kiedy zachodzi na ekranie gracza wyświetla się informacja o ładowaniu.

4.1.2 Dołączanie do pokoi

Po udanym połączeniu z serwerem graczom załadowuje się scena Lobby, posiadająca obiekt ze skryptem LobbyManager. Jego zadaniem jest umożliwić dostęp do stworzenia nowego lub dołączenia nowego pokoju. Rozgrywka dostarczana w projekcie bazuje na kooperacji dwóch graczy, zatem w naszym rozwiązaniu nie chcieliśmy zapewnić możliwości dołączania do losowego pokoju. Również etapy zaprojektowane są dla dokładnie dwóch graczy, co wymaga ograniczenia liczby graczy mogących dołączyć do pokoju. Z powodu wymienionych warunków zdecydowaliśmy aby lista pokoi nie była jawna dla wszystkich graczy. Chcąc stworzyć pokój, gracz musi wpisać ciąg znaków w wyznaczone pole tekstowe i nacisnąć przycisk znajdujący się obok niego. Po jego naciśnięciu wykonywana zostaje funkcja CreateRoom() napisana w skrypcie LobbyManager.



Rys. 4.3: Fragment sceny lobby przedstawiający pole do wpisania nazwy pokoju

Za pomocą narzędzi dostarczanych przez Photon, podany w polu tekstowym ciąg znaków, wykorzystywany jest do nazwania tworzonego pokoju oraz ograniczania liczby graczy mogących dołączyć do pokoju o tej nazwie do dwóch. Jeśli pokój o wpisanej nazwie istniał poprzednio i liczba graczy w tym pokoju jest mniejsza od dwóch, gracz zamiast

stworzyć pokój dołączy do istniejącego. Nadpisana funkcja OnJoinedRoom(), która wykonuje się po udanej próbie dołączenia gracza do pokoju, przenosi gracza na scenę gdzie będzie odbywać się rozgrywka.

4.2 Obsługiwanie graczy podczas rozgrywki

Połączenie się graczy do tego samego lobby jest kluczowym elementem aby umożliwić im wspólną rozgrywkę, jednak oprócz tego należy również dostosować mechaniki obiektów zawarte w skryptach. W tym rozdziale opisane zostaną elementy, które wymagały dostosowania do rozgrywki wieloosobowej.

4.2.1 Obiekty lokalne i dzielone między graczami

Wysyłanie zbyt wielu informacji do serwera nie jest możliwe ze względu na wielkość pakietu, zatem domyślnie, żadne informacje o obiektach na scenie nie są wysyłane do innych klientów. Domyślnie więc, cała zawartość gry zostaje załadowana u gracza lokalnie, a to oznacza, że nawet jeśli obaj gracze dołączyli do tego samego pokoju, jeśli jeden wykona dowolną akcję ta nie będzie miała odzwierciedlenia na scenie u drugiego gracza. Trzeba zatem zidentyfikować, jakie informacje trzeba będzie wysyłać do innego gracza przez serwer, aby zaimplementowane w projekcie mechaniki i narzędzia działały poprawnie w trybie wieloosobowym. Elementy o których informacje są wysyłane między klientami przez serwer to:

- Protagonista
- Potwór
- Kryjówki
- Drzwi
- System Wentylacji
- Przedmioty do zbierania
- Trigger do zmiany scen w których odbywa się rozgrywka

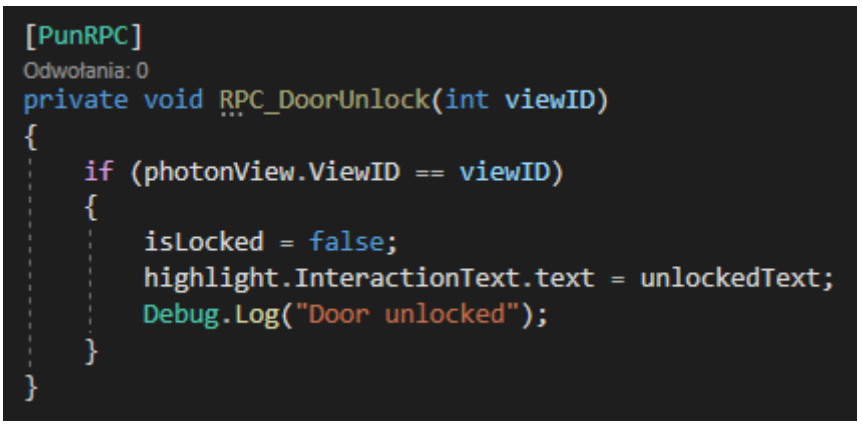
Zmiana stanu każdego z tych obiektów jest istotna dla rozgrywki zatem ich pozycja lub stan w jakim się znajdują muszą być przesyłane między sesjami graczy. W rozdziale 4.2.2 jest opisany sposób w jaki dokonuje się synchronizacja odpowiednich elementów gry.

4.2.2 Synchronizacja gry między klientami

W celu umożliwienia synchronizacji obiektów między sesjami graczy, każdy posiada swój własny unikalny identyfikator w postaci numeru. Dzięki niemu odpowiednie obiekty mogą być rozpoznawane przez każdego uczestnika rozgrywki odbywającej się w danym pokoju. Jest za to odpowiedzialny komponent **PhotonView**. Pozwala on także na skonfigurowanie sposobu, w jaki klienci kontrolują instancje. [8]

Do scen w których znajdują się etapy do przechodzenia, umieszczany jest obiekt z dołączonym skryptem o nazwie SpawnPlayers. Dzięki niemu, w momencie dołączenia do pokoju, każdy z graczy tworzy instancję obiektu własnego protagonisty, a ponieważ ma on nadane swój własny identyfikator zapewniony przez PhotonView, pojawia się również u drugiego gracza. Ponieważ pozycja i rotacja obiektów protagonistów nieustannie zmienia się podczas ich poruszania, dodany został również komponent o nazwie **PhotonTransformViewClassic**. Klasa ta umożliwia synchronizowanie pozycji, rotacji i skali obiektu, a także upłynnienie wyglądu zsynchronizowanych wartości, nawet jeśli dane są wysyłane tylko kilka razy na sekundę. Komponent ten posiada również potwór, którego pozycja musi być identyczna dla obu graczy.

Obiekty wymienione w poprzednim rozdziale mają swoje własne mechaniki oprócz ustalonej pozycji i rotacji, które działają dzięki skryptom, a również muszą być synchronizowane między graczami. W tym celu zastosowana została jedna z funkcji Photon PUN o nazwie **Zdalne Wywoływanie Procedur** (z angielskiego Remote Procedure Calls) w skrócie **RPC**. Są to wywołania metod zdalnie u klientów w tym samym pokoju. W celu włączenia Zdalnego Wywoływania Procedur dla wybranych metod, należy opatrzyć je odpowiednim atrybutem. Następnie za pomocą funkcji zapewnianej przez PhotonView można wysłać do określonych klientów informację jaką metodę należy w danym momencie wykonać.



Rys. 4.4: Przykładowa metoda opatrzona atrybutem PunRPC

Nie wszystkie funkcje powinny być opatrzone atrybutem “PunRPC”. Mogłoby to spowodować nieoczekiwane lub niechciane zachowania. Przykładowo drzwi powinny przenieść do innego pokoju tylko gracza który ich używa, a nie obu. Dlatego kluczowe jest zdalne wywoływanie jedynie niektórych funkcji.

Pierwszorzędnym elementem który należy synchronizować między klientami to animacje obiektów. Atrybutem PunRPC są objęte zatem funkcje wysyłające zdarzenia dotyczące animacji. W kontekście synchronizowania samych mechanik, każdy obiekt będzie różnił się pod kątem zdalnego wywoływania jego metod. Funkcje, które są wywoływane zdalnie u klientów (z wyłączeniem tych odpowiedzialnych za animacje) są odpowiedzialne za:

- Zmianę stanów każdego z protagonistów
- Zmianę stanów u każdego z potworów
- Odblokowanie kluczem zamkniętych drzwi
- Przenoszenie przedmiotów podczas podnoszenia lub upuszczenia z ekwipunku
- Aktualizowanie liczby graczy stojących na płytkach naciskowych
- Zmiana scen etapów

W każdym miejscu w kodzie, w którym wykonywana była metoda objęta wspomnianym atrybutem, jej wywołanie zostało podmienione na funkcję zapewnianą przez klasę PhotonView, dzięki której do wybranych klientów w pokoju zostaje wysłana informacja o jej wykonaniu.

5. Zastosowane rozwiązania programistyczne

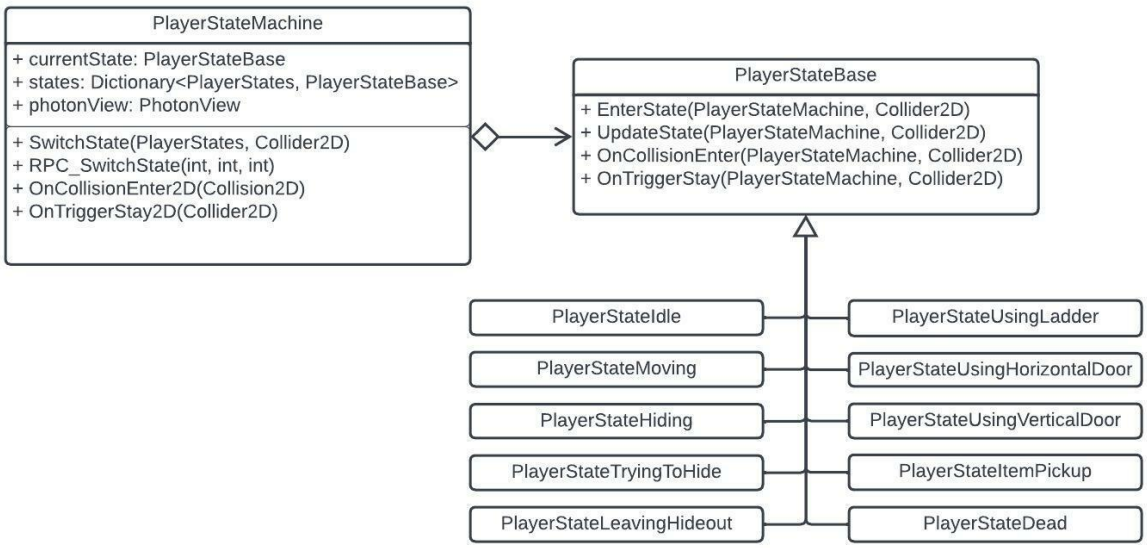
5.1 Wzorzec projektowy Stan

Podczas implementacji mechanik obiektu protagonisty pojawił się problem związany z ilością instrukcji warunkowych, ujawniających się w miarę dodawania kolejnych jego zachowań. Większość metod miała olbrzymie rozmiary oraz były bardzo nieczytelne, przy zastosowaniu w nich jedynie struktur warunkowych if lub switch. Trudno było także przewidzieć wszystkie możliwe zachowania i przejścia między nimi na etapie projektowania.

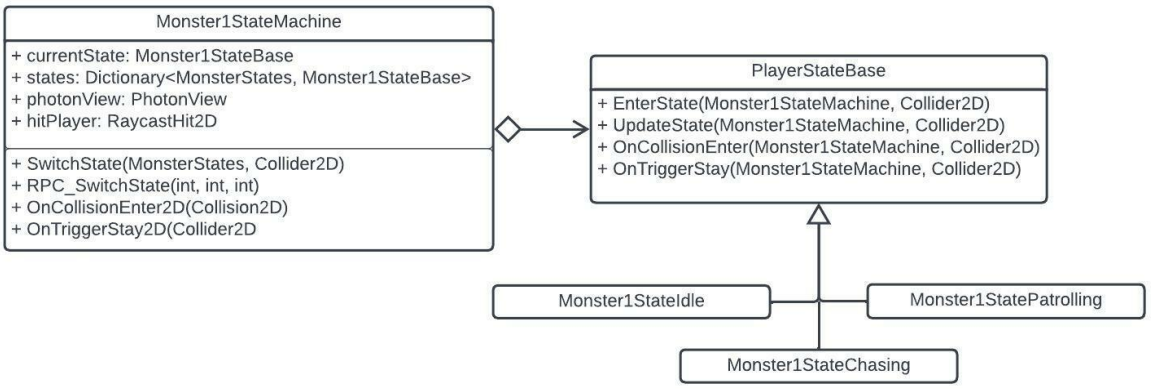
Z pomocą przyszedł wzorzec projektowy Stan, który zamiast implementować wszystkie zachowania samodzielnie, proponuje stworzenie nowych klas dla każdego z możliwych stanów obiektu oraz ekstrakcję wszystkich zachowań zależnych od stanu do tychże klas. [9]

Maszyna stanów przechowuje w sobie odniesienie do aktualnie reprezentującej ją stanu i deleguje mu związane z nim zadania.

Znaczną zaletą takiego rozwiązania jest to, że poszczególne stany mogą być świadome siebie nawzajem i inicjować przejście do innego stanu, bez wiedzy o jego implementacji.



Rys. 5.1: Maszyna stanów protagonisty zwizualizowana w UML



Rys. 5.2: Maszyna stanów potwora zwizualizowana w UML

5.2 Wzorzec projektowy Singleton

Singleton jest wzorcem projektowym gwarantującym istnienie tylko jednego obiektu danej klasy. Udostępnia także tylko jeden punkt dostępowy do tego obiektu z dowolnego miejsca w programie.

Wzorzec ten został zastosowany w projekcie między innymi w klasie CoroutineHandler, która odpowiada za obsługiwanie korutyn. Najczęściej wykorzystywany jest przez nas w poszczególnych stanach gracza, które ze względu na brak dziedziczenia po klasie MonoBehaviour, nie mogą wykonywać korutyn, które są w projekcie często wykorzystywane do automatycznej zmiany pozycji obiektu protagonisty podczas interakcji z kryjówkami lub drzwiami.

5.3 Wzorzec projektowy Obserwator

Obserwator jest behawioralnym wzorcem projektowym pozwalającym zdefiniować mechanizm subskrypcji, a następnie zawiadamiać subskrybentów o zdarzeniach zachodzących w obserwowanym obiekcie. Głównymi zaletami stosowania omawianego wzorca są:

- ➢ brak potrzeby aktywnego sprawdzania przez klasę interesującego ją obiektu
- ➢ odizolowanie od siebie obiektów
- ➢ możliwość informowania przez obiekt obserwowany wielu obserwatorów na raz

Wzorzec Obserwator został wielokrotnie zastosowany w projekcie, ze względu na dwa aktywne obiekty protagonisty, których zachowanie wpływa na wiele innych obiektów.

Jednym z istotnych elementów rozgrywki, który obserwuje zachowanie graczy jest klasa GameOverScreenManager, odpowiedzialna za zakończenie, a następnie zrestartowanie danej sceny u obu graczy w momencie złapania przez potwora dowolnego obiektu protagonisty. Innym elementem wykorzystującym ten wzorzec projektowy są drzwi na płytę naciskową, które na jego podstawie śledzą stan płyt naciskowych wymaganych do ich otwarcia.

5.4 System obsługi dźwięków

Aby usprawnić zarządzanie dźwiękami oraz ułatwić dodawanie kolejnych w przyszłości wraz z rozwojem projektu, zastosowano wiele rozwiązań technicznych pomagających w ich implementacji.

5.4.1 Sound

Klasa **Sound** jest podstawową klasą systemu dźwiękowego projektu. Posiada ona w sobie wyłącznie dwa pola: plik dźwiękowy, oraz typ numeryczny reprezentujący ten plik. W celu ułatwienia dodawania do gry kolejnych dźwięków, klasa sound dziedziczy po klasie ScriptableObject, co pozwala na bardzo proste tworzenie kolejnych obiektów klasy.

5.4.2 SoundPlayerPool

Klasa **SoundPlayerPool** odpowiada w systemie dźwiękowym za optymalizację.
Najprostszym możliwym rozwiązaniem, które mogłoby być zastosowane w systemie dźwiękowym, byłoby utworzenie w razie potrzeby obiektu, który odgrywałby potrzebny dźwięk, a następnie po odegraniu był niszczony. Przy większej ilości dźwięków, mogłoby się to jednak okazać dużym obciążeniem dla procesora, ze względu na potrzebę ciągłego zwalniania nieużywanej pamięci obiektu przez <garbage collector>, czyli automatycznego zarządcę pamięci języka C#.

Aby uniknąć spadku wydajności podczas ogrywania dużej ilości dźwięków na raz, w systemie dźwiękowym zastosowany został Object Pooling. Jest to kreacyjny wzorzec projektowy polegający na stworzeniu na początku gry puli nieaktywnych obiektów, które w trakcie rozgrywki zostają aktywowane i dostosowane do aktualnych potrzeb projektu, a po zakończeniu swojej pracy, ponownie zostają dezaktywowane i czekają na kolejne instrukcje. Dodatkowo, gdy wszystkie obiekty z puli zostaną wykorzystane, tworzony jest kolejny, aby uzupełnić ten brak, jednak nie jest on usuwany po zakończeniu swojego działania, lecz pozostaje w puli, aby taka sytuacja nie wydarzyła się ponownie. [10]

5.4.3 SoundsManager

Klasą pozwalającą na zarządzanie wszystkimi dźwiękami jest klasa **SoundsManager**. Jest ona singletonem oraz posiada publiczne metody odpowiedzialne za odgrywanie plików dźwiękowych, przez co jest ona dostępna w każdym miejscu w kodzie, co pozwala na precyzyjne odgrywanie wszystkich dźwięków oraz ułatwia dodawanie kolejnych wraz z rozwojem projektu.

Znacznym uproszczeniem zastosowanym w opisywanej klasie, są metody, które jako argumenty przyjmują jedynie typ numeryczny reprezentujące dany plik dźwiękowy oraz pozycję, w której dany dźwięk ma zostać odegrany. Następnie na podstawie przekazanego typu numerycznego odpowiedni plik dźwiękowy jest wyszukiwany w liście obiektów klasy Sound znajdującej się wewnątrz opisywanej klasy. Pozwala to na uniknięcie dużej liczby niepotrzebnych referencji.

5.5 Zastosowanie słowników w maszynach stanów

Jednym z wyzwań implementacyjnych w projekcie podczas dodawania mechaniki wieloosobowej była odpowiednia synchronizacja zmiany stanów przez maszyny stanów. Podczas testów manualnych, okazało się, że ze względu na niewielkie różnice obliczeniowe, stany nie zmieniały się tak samo u obu graczy, więc potrzebne okazało się zaimplementowanie mechanizmu ich synchronizacji.

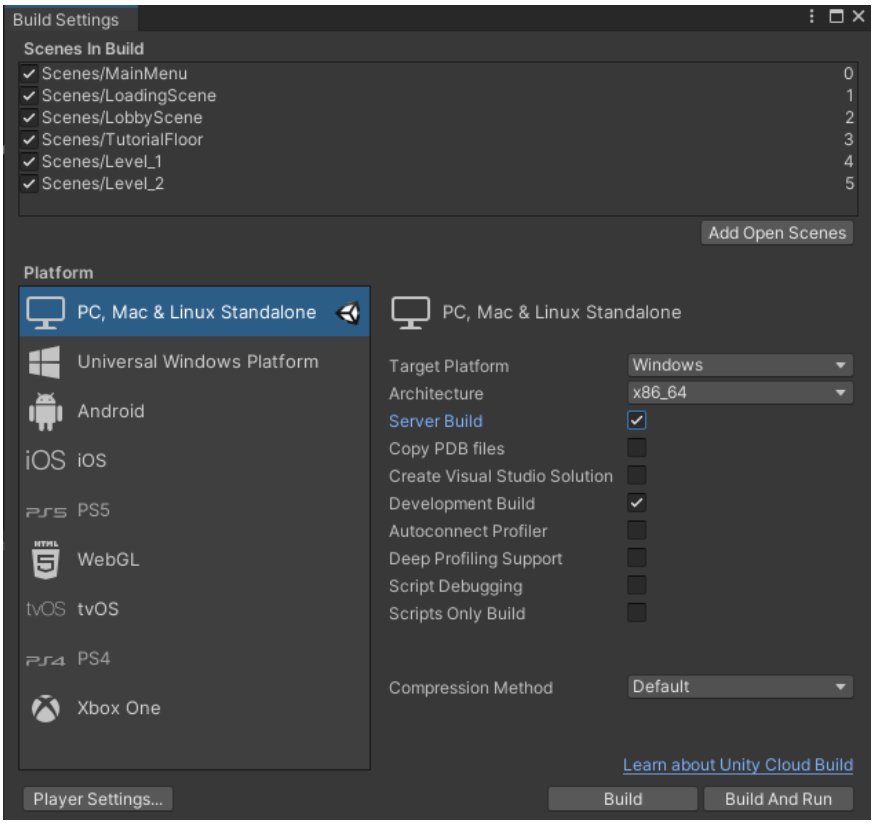
Photon, podczas przesyłania pakietów danych korzysta z wysoko zoptymalizowanych protokołów przesyłania binarnego, więc wszystkie dane które mają być przesłane są najpierw serializowane, czyli zmieniane na strumienie bajtów. Oznacza to że im bardziej złożone typy danych będziemy przesyłać, tym strumienie bajtów będą dłuższe, co wpłynie na szybkość komunikacji między graczami.

Aby zoptymalizować szybkość komunikacji sieciowej między graczami, zamiast przesyłać całe obiekty stanów maszyny, zastosowane zostały słowniki, które jako klucze, posiadają typy wyliczeniowe, a jako wartości odpowiadające im stany. Dzięki takiemu rozwiązaniu, w pakiecie dotyczącym zmiany stanu przesyłany jest tylko jeden integer, który następnie przez odbiorcę wykorzystywany jest do znalezienia odpowiedniego stanu i przypisaniu go jako obecnego.

Rozwiązanie to ułatwia zarządzanie stanami, sprawiając że kod staje się bardziej czytelny i podatny na prostą rozbudowę maszyny o kolejne stany w przyszłości.

6. Rozwój gry po eksporcie projektu

Aby z plików projektu w Unity stworzyć grę, którą można uruchomić należy w edytorze Unity ją **zbudować.** Zbudowanie jest procesem w którym Unity zamienia pliki projektu na inną, odpowiednią strukturę plików które umożliwiają uruchomienie gry na wybranej platformie. W trakcie procesu budowania można dostosować wiele elementów gry, między innymi sceny uwzględnione w projekcie lub platformę na którą chcemy dokonać eksportu projektu. Projekt aktualnie posiada 6 scen: menu, 2 związane z umożliwieniem rozgrywki sieciowej oraz 3 etapy zawierające etapy do przejścia.

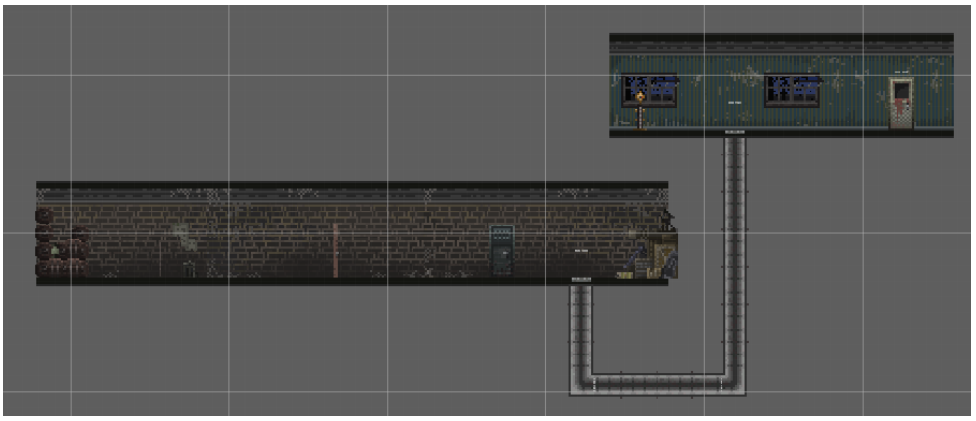


Rys. 6.1: menu ustawienia budowania projektu w Unity

Pracę nad projektem rozpoczęliśmy we wrześniu 2021, zatem w trakcie procesu rozwijania gry bardzo wiele elementów było na bieżąco dostosowywanych. Gra dwukrotnie była eksportowana i testowana na niewielką skalę, zarówno wśród osób grających regularnie w gry jak i nowicjuszy. Przeprowadzone zostały testy A/B scen pierwszego oraz drugiego poziomu. W wyniku tych testów kompletnie przerobione zostały poziomy gry oraz dostosowane pewne elementy. Głównie zmieniła się długość poziomów i poziom ich

skomplikowania, ponieważ sprawiały one graczom zbyt dużą trudność. Skrócenie etapów i redukcja poziomu skomplikowania zaowocowała zwiększeniem liczby osób, które były w stanie przejść etap. Poniższe grafiki przedstawiają etap zbudowany przed przeprowadzeniem testów i po ich wykonaniu.

Rys. 6.2: Pierwszy poziom w grze przed testowaniem



Rys. 6.3: Pierwszy poziom w grze po testowaniu

W wyniku otrzymywanych informacji zwrotnych powstało także opisane wcześniej Podświetlenie Interakcji które pozwoliło graczom na szybsze zapoznanie się ze sterowaniem. Zmieniły się również graficzne reprezentacje przedmiotów do podnoszenia. Poprzednio były one zaprojektowane bardzo minimalistycznie i zgodnie ze skalą wielkości, natomiast po zmianach ich rozmiar modeli został znacznie powiększony celem przejrzystości i umiejętności szybszego dostrzegania lub rozpoznawania przedmiotu.

Znaczna część mechanik gry została zaprojektowana w taki sposób, aby ich dalsza rozbudowa była prosta, czemu służą rozwiązania opisane w rozdziale piątym. W przyszłości można zatem bez problemu zwiększyć ilość poziomów i rozwinąć rozgrywkę. Dzięki rozgraniczeniu logiki obiektów od ich reprezentacji graficznych, w razie chęci wizualnego stylu gry również nie trzeba będzie modyfikować napisanego kodu ani modyfikować działania mechanik. Również dzięki wyizolowaniu metod odpowiedzialnych za wykrywanie sygnału wejściowego używanego do sterowania protagonistą w osobnym skrypcie, ułatwione będzie dostosowanie tej części projektu pod eksport na platformy mobilne jak Android czy IOS.

Podsumowanie

Rezultatem naszej pracy jest wersja demonstracyjna gry horrorowej w 2D z rozgrywką sieciową w postaci kooperacji dwóch graczy. Na początku pracy został krótko opisany koncept i podstawowe elementy gry. Następnie celem lepszego zrozumienia zaimplementowanych przez nas mechanik gry, zostały wymienione i opisane narzędzia, które były konieczne do wykonania części projektowej, w szczególności Unity, czyli silnik gry. W kolejnym rozdziale, opisany został sposób stworzenia tych mechanik, czyli kluczowej części projektu ściśle związanej z naszym wkładem własnym w pracy. W rozdziale czwartym wydzielona została część dotycząca implementacji rozgrywki sieciowej. Nawiązuje ona również do poprzednio opisanych mechanik, tłumacząc ich najistotniejsze modyfikacje wymagane do poprawnego funkcjonowania rozgrywki sieciowej. Rozdział piąty opisuje zastosowane techniki pisania kodu i rozwiązania zapewniające jego przejrzystość i hermetyczność, jednocześnie umożliwiając dalszą jego rozbudowę. Ostatni rozdział opisuje proces eksportu gry na system operacyjny Windows oraz wskazuje na obszary gry, które zostały poprawione dzięki informacjom zwrotnym od graczy.

Największą trudnością projektu było zsynchronizowanie mechanik z częścią graficzną, co wymagało rozszerzenia logiki o metody związane z automatycznym zarządzaniem obiektem gracza podczas interakcji. Dodatkowo dużą przeszkodą okazało się dostosowywanie istniejącego już kodu do rozgrywki sieciowej.

Mechaniki gry, takie jak ekwipunek gracza, przedmioty do zbierania, system dźwięków oraz maszyna stanów protagonisty, są kompletne i napisane w sposób umożliwiający łatwą rozbudowę wraz z dalszym rozwojem projektu. Wszystkie mechaniki zostały również w pełni dostosowane do rozgrywki wieloosobowej,

Bibliografia

[1] Newzoo video game market analysis
https://newzoo.com/insights/articles/new-gaming-boom-newzoo-ups-its-2017-global-games-market-estimate-to-116-0bn-growing-to-143-5bn-in-2020
[2] Creating Horror through Level Design - Jared Mitchell
https://jaredemitchell.com/articles/creating-horror-through-level-design/
[3] Unity manual:
https://docs.unity3d.com/Manual/index.html
[4] Komponent:
https://pl.wikipedia.org/wiki/Komponent_(informatyka)
[5] The Fundamentals of Designing a Horror Game - Thomas Eaves
https://www.linkedin.com/pulse/fundamentals-designing-horror-game-thomas-eaves/?trk=public_profile_article_view
[6] Creating a Photon Account:
https://www.kodeco.com/1142814-introduction-to-multiplayer-games-with-unity-and-photon#toc-anchor-006
[7] Photon - Introduction/Photon Cloud:
https://doc.photonengine.com/fusion/current/getting-started/fusion-intro
[8] Photon Unity Networking: General Documentation:
https://documentation.help/Photon-Unity-Networking-1.91/general.html
[9] Design Patterns:
https://refactoring.guru/pl/design-patterns
[10] Object Pooling Design Pattern:
https://sourcemaking.com/design_patterns/object_pool