

Politechnika Wrocławsk

Wydział Informatyki i Telekomunikacji

Kierunek: IST

**ZESPOŁOWE PRZEDSIEWZIĘCIE
INFORMATYCZNE**

**Dynamiczna gra komputerowa 3D z elementami
eksploracji podziemi**

Wojciech Begierski

Piotr Dudziak

Paweł Kułaczkowski

Łukasz Małecki

Opiekun pracy

dr inż., Grzegorz Popek

Słowa kluczowe: Unity, dynamiczna, gra, hack&slash

WROCŁAW (2023)

Spis treści

1. Wykaz symboli, oznaczeń i akronimów (opcja).....	5
2. Cel i zakres przedsięwzięcia.....	5
3. Słownik pojęć (opcja).....	5
4. Stan wiedzy w obszarze przedsięwzięcia (opcja).....	6
5. Założenia wstępne.....	8
6. Specyfikacja i analiza wymagań na produkt programowy.....	10
7. Projekt produktu programowego.....	10
8. Implementacja (opcja).....	12
8.1 System statystyk.....	12
8.1.1 Wprowadzenie.....	12
8.1.2 Implementacja.....	12
8.1.3 Rekalkulacja statystyk.....	14
8.1.4 StatisticsInput.....	15
8.1.5 UI.....	15
8.2 System przedmiotów i ekwipunki.....	17
8.2.1 Logika.....	17
8.2.2 UX/UI.....	20
8.3 System skalowania obrażeń (Łukasz Małecki).....	21
8.3.1 Idea.....	21
8.3.2 Skalowanie ze statystykami podstawowymi - koncept.....	22
8.3.2 Konkretnie informacje dotyczące skalowania.....	22
8.4 Sztuczna inteligencja, przeciwnicy (Łukasz Małecki).....	24
8.4.1 Dynamiczne patrole przeciwników.....	24
8.4.2 Typy przeciwników (Łukasz Małecki).....	26
8.5 Algorytm generowania poziomów.....	39
8.5.1 Założenia.....	39
8.5.2 Parametry konfiguracyjne.....	41
8.5.3 Kroki algorytmu.....	41
8.6 Obsługa działania poziomów.....	42
8.6.1 Funkcjonowanie poziomów.....	42
8.6.2 Integracja z graczem.....	45
8.6.3 Integracja z przeciwnikami.....	46
8.6.4 Quality of Life.....	46
8.7 Optymalizacja.....	47
8.7.1 Memoizacja.....	47
8.7.1 Przetwarzanie równoległe.....	48
8.8 System dropów.....	49
8.8.1 Wprowadzenie.....	49

8.8.2 Działanie.....	50
8.9 System Poruszania się i życia.....	51
8.9.1 System poruszania się.....	51
8.9.2 System Życia.....	55
8.10 HDRP oraz Visual effect Graph.....	57
8.10.1 Zalety HDRP:.....	57
8.10.2 Wady HDRP:.....	58
8.10.3 Alternatywa HDRP - URP.....	58
8.10.4 Decyzja i wybór Render Pipeline'a.....	59
8.11 Particle System vs Visual Effect Graph.....	59
8.10 System Magii.....	61
8.10.1 Efekty nakładane na gracza i przeciwników.....	64
8.10.2 Pomocnicze klasy do zaklęć.....	66
8.10.3 Ataki strzelające.....	71
8.10.4 Defensywne.....	74
8.10.5 Strzelanie naprowadzane.....	75
8.10.6 Ataki obszarowe.....	76
8.10.6 Ataki bliskiego zasięgu.....	76
8.11 Menu & Score.....	77
8.11.1 Pierwsze uruchomienie.....	77
8.12 Fabuła & Film intro (promocyjny).....	77
8.12.1 Początki gry, pomysł i potrzeba na fabułę.....	77
8.12.2 Fabuła:.....	78
8.12.3 Film promocyjny/ wstęp do gry:.....	78
9. Testy produktu programowego/Wyniki i analiza badań.....	79
9.1 Testy jednostkowe.....	79
9.2 Testy użytkowników końcowych.....	80
10. Podsumowanie.....	80
1. Wprowadzenie Kontrolowanie postaci:.....	81
2. Instalacja produktu programowego.....	81
2.1. Wymagania systemowe.....	82
System: Windows 7/8/10/11 64-bitowy.....	82
2.2. Opis procesu instalacji.....	82
Gra nie wymaga instalacji. Dostępna jest w postaci pliku wykonywalnego .exe... 82	
2.3. Opis realizacji typowych zadań z podziałem na ich typy i/lub aktorów.....	82
2.3.1 Realizacja kolizji.....	82
2.3.2 Główne cele.....	83
2.3.3 Interakcje gracza.....	83

DOKUMENTACJA PROJEKTOWA

1. Wykaz symboli, oznaczeń i akronimów

SO - Scriptable Object

2. Cel i zakres przedsięwzięcia

Celem naszego projektu jest wytworzenie prototypu jednoosobowej gry komputerowej 3D. Gra ta ma zawierać generowane proceduralnie podziemia, szeroki zestaw zakleć, dynamiczną rozgrywkę, system przedmiotów i statystyk. Gra ma być napisana tak, aby łatwo dało się ją rozwijać.

3. Słownik pojęć (opcja)

- boss - przeciwnik znacznie silniejszy od reszty, często strzegący czegoś (skarbu, przejścia)
- dash - mechanika z gier, która pozwala postaci na krótki, ekspresowy bieg w określonym kierunku, często zwiększający jej prędkość w trakcie rozgrywki.
- drop - przedmiot lub zbiór przedmiotów, które gracz otrzymuje po pokonaniu przeciwnika.
- element – typ ataku, żywioł, np. ogień, woda, fizyczny, ciemność
- entity - byt reprezentowany przez gracza lub przeciwnika
- mana - Energia w światach fantasy, używana przez magów do czynienia czarów
- maszyna stanów - model obliczeń zakładający skończoną liczbę stanów i zawsze bycie
- prefab - “szablon opisujący obiekt gry (z zestawem komponentów), na podstawie którego można tworzyć wiele instancji” ~ wyk 1 Programowanie gier dr Bogumiła Hnatkowska
- skalowanie – wpływ grupy parametrów na wartość innego parametru
- statystyki defensywne – statystyki elementowe wpływające na zmniejszenie otrzymywanych obrażeń danego typu, np. FireResistance, PhysicalResistance, ElectricityResistance (nie defense!)
- statystyki elementowe – statystyki dotyczące konkretnego elementu, np. FireDamage, AirResistance, PhysicalDamage
- statystyki ofensywne – statystyki elementowe wpływające na zwiększenie zadawanych obrażeń danego typu, np. WaterDamage, DarknessDamage, PhysicalDamage

- statystyki podstawowe – statystyki niepowiązane jednoznacznie z żadnym elementem t.j. siła, zręczność, inteligencja, obrona
- w dokładnie jednym stanie.
- zasób (asset) - reprezentacja dowolnego elementu, który może być wykorzystany w grze (np. plik graficzny, plik audio)

4. Stan wiedzy w obszarze przedsięwzięcia (opcja)

np. analiza istniejących rozwiązań z podsumowaniem, opis porównywanych metod

Założeniem projektu było stworzenie dynamicznej komputerowej gry 3D. Gra ma być pełna akcji, szybkiego ruchu i intensywnych walk w kompleksowych, interaktywnych podziemnych lochach, gdzie gracze odkrywają tajemnice, skarby i rozwijają swoje postacie.

Rodzajami gier jakie pasują do naszego opisu to: **Hack & Slash**, **Roguelike** i **FPS**.

1. **Hack & Slash:** Gatunek gier komputerowych charakteryzujący się szybkimi i dynamicznymi starciami, w których gracz koncentruje się głównie na eliminowaniu wielu wrogów poprzez powtarzalne ataki. Wymaga to umiejętności strategicznego podejścia do walki oraz szybkich reakcji. Gry tego typu często prezentują intensywne akcje, wykorzystując różnorodność umiejętności i efektowne ruchy postaci. Przykładami takich gier są: "**Diablo**", "**Path of Exile**" oraz "**God of War**".
2. **Roguelike:** Główne cechy tego gatunku gier obejmują proceduralnie generowane poziomy, gdzie struktura świata gry jest losowa, co zapewnia unikalne doświadczenia podczas każdej rozgrywki. Istotnym elementem są również stała śmierć postaci, co oznacza utratę postępu w grze po niepowodzeniu. Grając, gracze eksplorują te losowo generowane poziomy, zdobywając losowe przedmioty i rozwijają umiejętności, co stanowi integralną część rozgrywki. Przykładami gier tego rodzaju są "**Rogue Legacy**" oraz "**The Binding of Isaac**".
3. **FPS (First-Person Shooter):** Gatunek gier, w którym gracz obserwuje świat gry z perspektywy pierwszej osoby, często skupiający się na intensywnych starciach z przeciwnikami za pomocą różnorodnych broni palnych. Gra oferuje dynamiczną rozgrywkę, szybkie tempo oraz silny nacisk na precyzję i szybkie reakcje gracza. Przykładem takiej gry jest "**Doom**".

Wyżej wymienione gry stanowiły inspirację do naszego projektu i są jego głównymi rywalami

Podsumowując analizę krytyczną istniejących rozwiązań można porównać te typy gier do naszej w niżej przedstawionej tabeli. Oczywistym jest, że gry jednego typu będą się różniły i nie wszystkie posiadają cechy zaprezentowane tak jak poniżej, jednak może być dobrą reprezentacją klasyfikacji do poszczególnych typów:

Cecha	System umiejętności ości i rozwoju postaci	Pierwszo osobowa perspektywa (FPP)	Elementy skradanki	Szybka i dynamiczna rozgrywka	System statystyk	Zaawansowany system magii
Roguelike	✓	✗	✗	✗	?	✗
Hack and Slash	✓	✗	✗	✓	✓	✓
FPS	✗	✓	✓	✓	?	?
Nasza gra	✓	✓	?*	✓	✓	✓

Cecha	Proceduralne generowanie poziomów	Permadeath (śmierć kończy grę)	Losowe przedmioty i nagrody	Faza eksploracji i odkrywania świata	Walka z hordami przeciwników
Roguelike	✓	✓	✓	✓	✗

Hack and Slash	✗	✗	✗	?	✓
FPS	✗	✗	✗	✗	✓
Nasza gra	✓	✓	✓	✓	✓

✓ - dany typ gry posiada cechę

✗ - dany typ gry nie posiada danej cechy

? - dany typ gry niekoniecznie posiada te cechy i zależy od gry lub stylu rozgrywki gracza.

* - mówi o tym, że są nie było zamierzone dostarczenie tej funkcjonalności w całości w pierwszej wersji gry.

5. Założenia wstępne

np. wysokopoziomowa lista wymagań i użytkowników, dobór technologii, przyjęte ograniczenia

Zdecydowaliśmy, że do stworzenia gry wykorzystamy gotowy silnik gry. Decyzja ta jest poparta następującymi argumentami:

- gotowe silniki zapewniają większą niezawodność
- skorzystanie z gotowego silnika zapewnia znacznie szybsze tworzenie samej gry
- tworzenie własnego silnika jest skomplikowanym i bardzo długim procesem
- obecne na rynku silniki gier są wysokiej jakości

Zdecydowaliśmy się stworzyć grę przy użyciu silnika gier Unity. Wybór padł na ten silnik z kilku powodów:

- większość grupy ma doświadczenie z tym silnikiem gry
- silnik ten jest darmowy
- Unity jest dostępne od 2005 roku, zatem dostępne jest wiele materiałów w internecie, do których można się odnieść w razie potrzeby
- dostęp do Unity Asset Store, w którym znajdują się także bezpłatne zasoby, takie jak modele postaci 3D z animacjami
- większość grupy ma doświadczenie z językiem programowania C#

Lista elementów gry, na których zdecydowaliśmy się nie skupiać w ramach projektu:

- Dźwięki
- Walki z bossami na końcu poziomu
- Widocznego awatara gracza
- Widocznej róźdżki poza ekwipunkiem
- Systemu poziomów i drzewka umiejętności gracza
- Niepłaskie pokoje z przeciwnikami + poruszania się przeciwników w osi Y (np. skakanie)
- Tryb wieloosobowy
- Dobrze zbalansowany i przetestowany system skalowania obrażeń

Assets:

- Część assetów była pobrana z asset store oraz innych stron, które udostępniały modele oraz grafiki na darmowej licencji.
- Tworzenie własnoręczne zasobów 3D, takich jak modele oraz animacje ataków magicznych
- Tworzenie własnoręczne własne zasobów 2D, takich jak tekstury do zaklęć magicznych
- Generowanie obrazów i tekstur 2D za pomocą darmowych modeli sztucznej inteligencji, takich jak ikony zaklęć oraz przedmiotów

Prace nad grą podzieliliśmy na kilka ogólnych modułów, w ramach których mieliśmy dostarczyć pewien zestaw funkcjonalności, modułami tymi są:

- Poruszanie i życie (Wojciech Begierski) - moduł obejmujący kontroler gracza, sposób interakcji gracza ze światem, system punktów życia i many dla Entities, pasek życia i many, przeładowanie damage indicator.
- System magii (Wojciech Begierski) - moduł obejmujący tworzenie graficzne zaklęć, tworzenie systemu interakcji zaklęć z graczem i z przeciwnikami, stworzenie uniwersalnego kodu pozwalającego na korzystanie z zaklęć zarówno przez gracza, jak i przeciwników, tworzenie efektów zaklęć, spell whell.

- Menu główne (Wojciech Begierski) - stworzenie menu głównego, menu dźwięku, ekranu pauzy, komunikatu śmierci. Zarządzanie poziomem dźwięku, dodanie filmu oraz jego pomijanie i ponowne odtwarzanie.
- Fabuła i film wstępny (Wojciech Begierski) - napisanie fabuły, nagranie i stworzenie filmiku wstępnego/ promocyjnego do gry.
- System statystyk (Paweł Kułaczkowski) - planowanie konceptu statystyk w grze, UI dla statystyk, przeliczanie się statystyk z przedmiotów, typy magii, system wpływu statystyk na obrażenia, system dostarczania statystyk dla Entities, system skalowania obrażeń ze statystykami i jego wypełnienie (Łukasz Małecki), wypełnienie statystykami przedmiotów i przeciwników (Łukasz Małecki)
- System przedmiotów (Paweł Kułaczkowski) - planowanie konceptu przedmiotów w grze, UI dla przedmiotów, ikonki dla przedmiotów, nazwy przedmiotów, opisy przedmiotów, system dropienia przedmiotów, system interakcji gracza z dropami, UI do zakładania zaklęć, przedmioty pozwalające na rzucanie zaklęć, wypełnienie przedmiotów statystykami (Łukasz Małecki), wypełnienie drop rate'ów dla przeciwników (Łukasz Małecki)
- Generowanie proceduralne (Piotr Dudziak) - planowanie konceptu na generowanie poziomów, planowanie sposobu reprezentacji pokojów i całego piętra w projekcie, generator poziomów, spawnery przeciwników, system komunikacji przeciwników z mechanizmem otwierania drzwi, modele pokojów, wypełnienie pokojów, system zależności zawartości pokojów/typów pokojów od numeru piętra
- Sztuczna inteligencja (Łukasz Małecki) - koncept na działanie przeciwników, wyszukiwanie assetów przeciwników, planowania działania i implementacja maszyn stanów dla przeciwników, planowanie systemów dynamicznej orientacji w terenie dla przeciwników, system animacji przeciwników, implementacja maszyn stanów, tworzenie wariantów przeciwników, system losowego wyboru zaklęć, system dynamicznych patroli, organizacja prefabów i obiektów ze statystykami, aby umożliwić dynamiczne pojawianie przeciwników z pożądanymi cechami

6. Specyfikacja i analiza wymagań na produkt programowy

np. definicja wymagań funkcjonalnych/niefunkcjonalnych i ich analiza; możliwe formy: diagram wymagań, diagram przypadków użycia, lista historyjek (może być pogrupowana w epiki) oraz ich uszczegółowienie w postaci tekstowych specyfikacji przypadków użycia lub diagramów aktywności lub testów akceptacyjnych lub opisów tekstowych

7. Projekt produktu programowego

np. opis decyzji architektonicznych, projekt architektury, bazy danych, zastosowane wzorce projektowe, inne (definicja zachowania w postaci diagramów sekwencji)

W projekcie dominuje component-based architecture, którego użycie wynika bezpośrednio ze stosowanego przez nas silnika gier Unity.

W kodzie zastosowano wiele wzorców projektowych, przede wszystkim w celu komunikacji między modułami.

- Wzorzec obserwator - użyty został wielokrotnie do komunikacji między elementami systemu. Przykładem użycia jest EnemySpawner i EnemySpecific, gdzie spawner obserwuje każdego przeciwnika, którego pojawił. W momencie, gdy przeciwnik umiera, wywołuje on odpowiednią metodę, która informuje spawner o swojej śmierci. Gdy każdy spawner w pokoju będzie wiedział, że wszyscy podlegli mu przeciwnicy nie żyją, gracz będzie miał możliwość otwarcia drzwi i przejścia dalej
- Wzorzec fasada - użyty został do ujednolicenia komunikacji z przeciwnikami, klasa EnemySpecific daje dostęp do korzystania z metod komponentów przeciwnika, bez konieczności samodzielnego wywoływania funkcji głęboko zaszytych. Pozwala na bezpieczne korzystanie z metod klas EnemyMagicSystem, Animator, EnemySM. HealthAndMana wywołuje metodę Die() z zawartą w klasie EnemySpecific bez konieczności wiedzy, jakim konkretnie przeciwnikiem jest dany obiekt.
- Maszyna stanów - do zarządzania zachowaniami przeciwników posłużyliśmy się maszyną stanów, której użycie zapewnia znacznie większą kontrolę i łatwość w znajdowaniu błędów. Każdy typ przeciwnika ma nie więcej niż 9 stanów, zatem praktycznie żadne problemy, związane z użyciem tego rozwiązania, nie występują. Potencjalnym problemem jest powtórzenie kodu, w pokrywających się stanach, takich jak Rest, Patrol, Chase.

Obsługa przeciwników została rozwiązana w sposób zapewniający możliwie najmniejsze powtarzanie kodu i spójność między różnymi typami przeciwników:

- Każdy przeciwnik komponent EnemySM i odpowiadający za konkretny rodzaj przeciwnika (np. MeleeSM, ShamanSM)
- EnemySM służy do przechowywania elementów niepowiązanych ściśle z typem przeciwnika, takimi jak prędkość poruszania, referencja do obiektu gry gracza, navMeshAgent
- EnemySM służy przede wszystkim do kontroli ruchu przeciwnika, udostępnia metody zmiany stanu (np. podążanie za celem (graczem), losowe przemieszczanie się po mapie, postój, obracanie się w kierunku celu)
- EnemySM udostępnia metody zwracające informacje konieczne do podejmowania decyzji przez konkretne maszyny stanów (np. odległość od celu, odległość od celu z wyłączeniem wysokości, czy przeciwnik znajduje się w podanej odległości od celu, czy droga w podanym wektorze jest wolna)
- EnemySM udostępnia jednolity mechanizm znajdowania ścieżki do patrolu, który preferuje ścieżki nieprowadzące do kolizji z innymi przeciwnikami
- Konkretne maszyny stanów korzystają z informacji udostępnianych przez EnemySM i własnych parametrów, aby odpowiednio zmieniać stany, np. dla MeleeSM:

```

3 references
public bool IsGoalInRange()
{
    if(enemySM.GetGoalDynamic() == null)
    {
        return false;
    }
    return enemySM.IsInRangeXZ(detectionRange);
}

if (mainStateMachine.IsGoalInRange())
{
    mainStateMachine.ChangeState(mainStateMachine.chase);
    return;
}

```

- Konkretnie maszyny stanów najczęściej ustawiają konkretny stan EnemySM przy wejściu do nowego własnego stanu, np. dla stanu Chase:

```

public void OnEnter()
{
    stateMachine.SetStateFollow();
}

```

- EnemySpecific jest klasą abstrakcyjną, po której dziedziczą wszystkie konkretne maszyny stanu
- EnemySpecific zawiera zestaw zbiera referencje do komponentów używanych przez każdy rodzaj przeciwnika i zestaw metod używanych przez każdy rodzaj przeciwnika (np. Die(), ChangeState(), CastSelectedSpell())
- EnemySpecific udostępnia abstrakcyjną metodę RandomizeStateTimes(), która ma jednorazowo pozwolić konkretnym maszynom stanu zmienić wartości czasu niektórych stanów, aby przeciwnicy nie robili wszystkiego w tym samym czasie
- Instancje stanów są przechowywane jako zmienne w maszynach stanu, aby nie trzeba było tworzyć nowej instancji przy każdym wywołaniu

8. Implementacja (opcja)

8.1 System statystyk

8.1.1 Wprowadzenie

W grze istnieje system statystyk, stworzony w celu zapewnienia większej różnorodności rozgrywki. Każdy entity posiada swoje własne statystyki. Statystyki te to wartości liczbowe reprezentujące podatność lub odporność na dany typ magii, obrażenia zadawane przez konkretny typ magii, punkty życia, punkty many, siła, zręczność, odporność i inteligencja.

8.1.2 Implementacja

Implementacja została przedstawiona na diagramie klas poniżej:

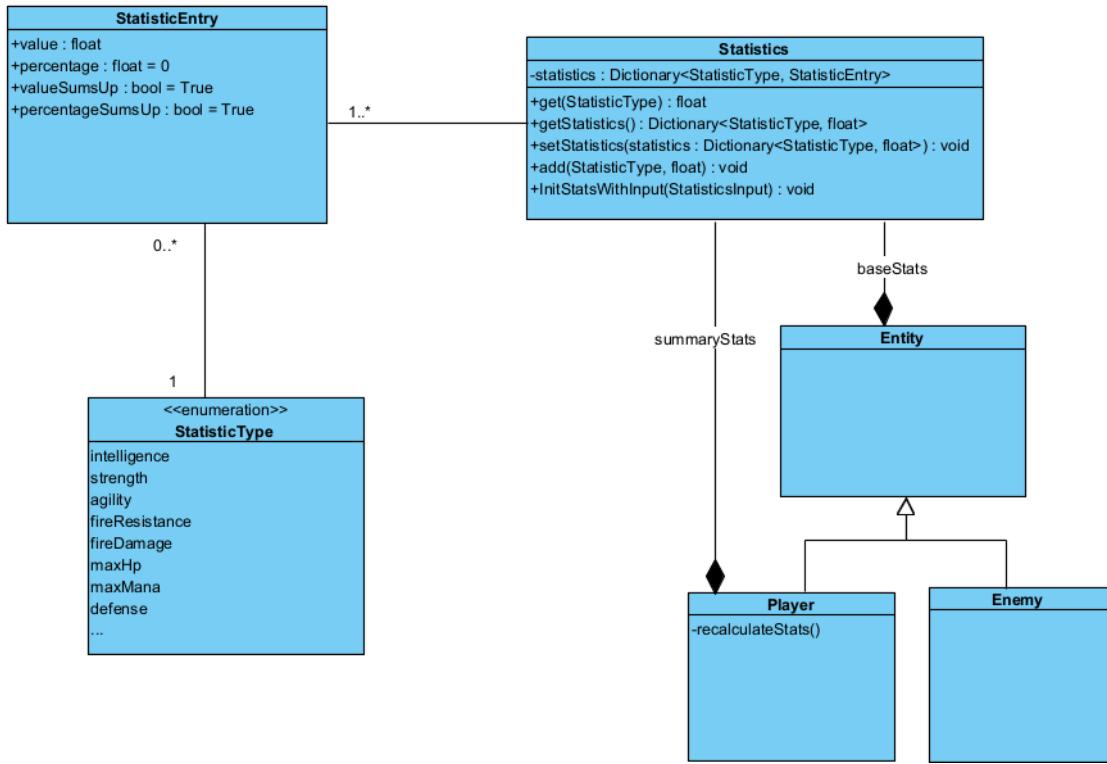


Diagram klas - system statystyk

Każdy typ statystyki (nazywany dalej *StatisticType*) zdefiniowany jest jako pozycja w enum *StatisticType*.

StatisticEntry reprezentuje dane dotyczące konkretnej statystyki (np. Obrażeń od ognia - *fireDamage*) i zawiera następujące informacje:

- *StatisticType* - typ statystyki którą reprezentuje
- *value* - wartość liczbową
- *percentage* - wartość procentowa
- *valueSumsUp* - informuje o tym, czy wartość liczbową (*value*) statystyki sumuje się z innymi (więcej w punkcie 8.1.3)
- *percentageSumsUp* - informuje o tym, czy wartość procentowa (*percentage*) statystyki sumuje się z innymi (więcej w punkcie 8.1.3)

Klasa *Statistics* posiada zbiór *StatisticEntry* realizowany jako mapa łącząca poszczególne *StatisticType*'y z odpowiadającymi im *StatisticEntry*.

Każdy *Entity* posiada swą własną instance klasy *Statistics* - *baseStats* które są statystykami podstawowymi, przydzielanymi każdemu entity. Pozostają one niezmienne. Dodatkowo, *Player* (gracz) posiada także *summaryStats* które jest wypadkową *baseStats* oraz statystyk wynikających ze statystyk używanych przez gracza przedmiotów i mogą dynamicznie zmieniać się w trakcie rozgrywki.

8.1.3 Rekalkulacja statystyk

Ponieważ gracz w trakcie gry może używać różnego rodzaju wyposażenia (więcej w punkcie 8.2) które również posiada swoje statystyki, statystyki gracza dynamicznie ulegają zmianie. W momencie założenia lub zdjęcia z gracza elementu wyposażenia, następuje rekalkulacja statystyk. Przebiega ona w następujący sposób:

8.1.3.1 - Opis ogólny

Zarówno *value* jak i *percentage* statystyk bazowych, jak i tych uzyskanych z przedmiotów używanych przez gracza są - dla każdego *StatisticType* osobno - sumowane jeżeli odpowiednia *valueSumsUp=true* i *percentageSumsUp=true*.

Następnie dla każdego *StatisticType* wybierana jest wartość maksymalna z:

- sumy sumowalnych wartości
- każdej wartości z niesumowalnych wartości

Analogicznie proces przebiega dla procentów.

Jest to mechanizm, który ma dostarczyć w grze elementu strategii - pewne przedmioty dają duże bonusy do statystyk, ale może być to obarczone tym, że wzmacnienia te nie będą sumowały się z innymi. W takim wypadku zostanie wybrany najbardziej korzystny dla gracza wariant.

8.1.3.2 - Opis szczegółowy - pseudokod w języku naturalnym

- 1.** pobierz wszystkie instancje *Statistics* z *Item*'ów będących na wyposażeniu gracza i zapisz do listy *statisticsArray*
- 2.** Zainicjalizuj obiekt *Statistics* o nazwie *result*. Będzie on używany do przechowywania wyników sumowania statystyk.
- 3.** Dla każdego obiektu *Statistics* w tablicy *statisticsArray*:

3.1 Dla każdego wpisu entry typu *StaticsicEntry* w tym obiekcie:

3.1.1 Jeśli wartość *entry.valueSumsUp* jest prawdą zsumuj wartość *entry.value* z aktualną wartością dla tego samego typu statystyki w obiekcie *result*.

3.1.2 Jeśli wartość `entry.percentageSumsUp` jest prawdą, zsumuj procent `entry.percentage` z aktualnym procentem dla tego samego typu statystyki w obiekcie `result`.

3.2 Dla każdego `StatisticType`:

3.2.1 Znajdź wszystkie wartości `value` z `statisticsArray`, które nie są sumowalne (`valueSumsUp` jest fałszem), i zapisz je do tablicy float'ów `notSummableValue`.

3.2.2 Jeśli tablica `notSummableValue` nie jest pusta:

3.2.2.1 znajdź w niej największą wartość i porównaj ją z obecną wartością `value` dla `statType` w `result`.

3.2.2.2 Przypisz większą z tych wartości do `value` w `result`.

3.2.3 Znajdź wszystkie procenty `percentage` z `statisticsArray`, które nie są sumowalne (`percentageSumsUp` jest fałszem), i zapisz je do tablicy `notSummablePercentage`.

3.2.4 Jeśli tablica `notSummablePercentage` nie jest pusta:

3.2.4.1 Znajdź w niej największy procent i porównaj go z obecnym `percentage` dla `statType` w `result`.

3.2.4.2 Przypisz większy z tych procentów do `percentage` w `result`.

4. Zwróć obiekt `result`

8.1.4 StatisticsInput

`StatisticsInput` to `ScriptableObject` zawierający informacje o bazowych statystykach dla wszystkich `Entities`. Wykorzystywany jest on w metodzie `InitStatsWithInput` z klasy `Statistics` do inicjalizacji statystyk.

8.1.5 UI

W grze dostępny jest ekran statystyk (uruchamiany klawiszem U) pozwalający na podgląd obecnych statystyk gracza (`summaryStats`).

Składa się on z trzech części - paneli:

- *Basic* - podstawowe statystyki gracza
- *Damage* - informacje o wzmacnieniach ataków zadawanych przez gracza
- *Resistance* - informacje o odporności na dane typy ataków



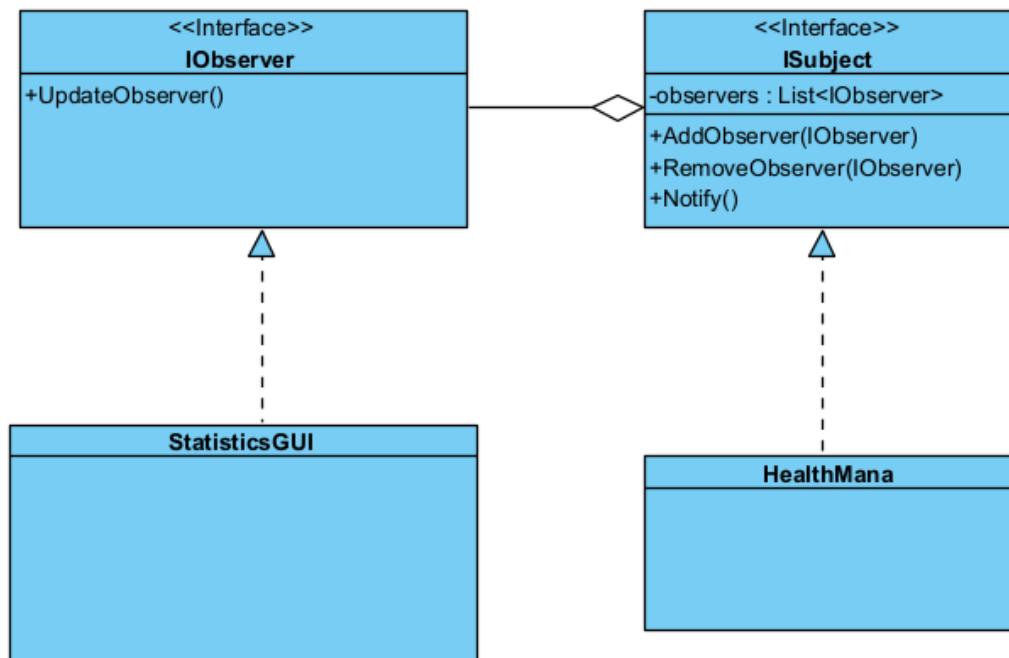
The screenshot shows a game statistics interface titled "Statistics". It is divided into three main sections: "Basic", "Damage", and "Resistance".

Basic		Damage		Resistance	
Health	120/100	Fire	0	Fire	20
Mana	100/100	Water	0	Water	20
Base Health	100	Earth	0	Earth	20
Base Mana	100	Air	0	Air	0
Intelligence	10	Darkness	0	Darkness	20
Strength	10	Lightness	0	Lightness	20
Agility	10	Electricity	0	Electricity	20
Defense	0	Physical	0	Physical	20

At the bottom right of the screen, there is a large green number "0".

Ekran statystyk gracza

Pozycje *Health* oraz *Mana* odświeżane są na bieżąco, ponieważ zmieniają się one szybko i dynamicznie. Wykorzystany został tu wzorzec obserwatora - klasa *StatisticsGui* odpowiedzialna za zarządzanie wspomnianym ekranem obserwuje obiekt odpowiedzialny za zarządzanie punktami życia i maną gracza (*HealthMana*) i powiadomiony reaguje na zmiany:



Reszta statystyk w celu optymalizacji odświeżana jest jedynie, gdy w ekwipunku (*Equipment*) nastąpi zmiana przedmiotów (*Items*) w które wyposażony jest gracz.

8.2 System przedmiotów i ekwipunki

8.2.1 Logika

Przedmioty dzielą się na dwie kategorie:

- *Items* - reprezentuje wszelkie przedmioty możliwe do założenia przez gracza.
Dzieli się na kategorie:
 - *Hat* (kapelusz)
 - *Coat* (płaszcz)
 - *staff* (kostur)
 - *pants* (spodnie)
 - *ring* (pierścień)
 - *boots* (buty)

Każdy *Item* posiada swoje statystyki.

- *Spell items* - reprezentują czary w postaci przedmiotu który można podnieść i wyposażyć się weń, w celu jego używania

Ponieważ oba te typy są podobne z punktu widzenia logiki, dziedziczą one po klasie abstrakcyjnej *Equipable* (kolekcjonowalne / możliwe do wyposażenia).

Samo *Equipable* dziedziczy po *ScriptableObject*, ponieważ możemy potraktować to jako asset - pewną niezmienialną rzecz (żadna ze składowych *Equipable* ani klas pochodnych nie zmienia się w trakcie rozgrywki. Obiekty tych klas są pewnymi konfiguracjami - definiują możliwe do zdobycia obiekty)

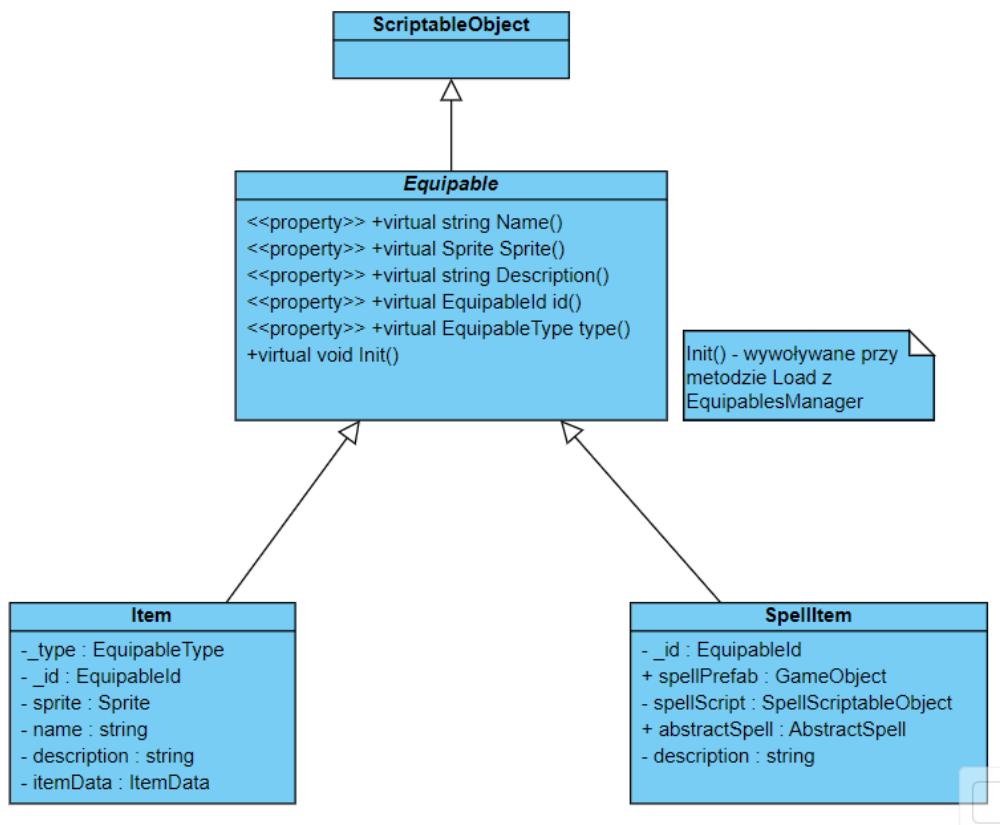


Diagram klas dla *Item* i *SpellItem*

W celu identyfikacji przedmiotów, każdy z nich posiada *EquipableId* będący enumem. Przykładowe wartości to:

fire_staff,

mystic_breeches,

Fireball.

Na początku gry odpowiednia klasa statyczna *EquipableManager* wczytuje z folderu *Resources/Equipables* wszystkie *Items* oraz *SpellItems*.

```

void Awake()
{
    LoadEquipable<Item>("Equipable/Items/Boots");
    LoadEquipable<Item>("Equipable/Items/Rings");
    LoadEquipable<Item>("Equipable/Items/Coats");
    LoadEquipable<Item>("Equipable/Items/Pants");
    LoadEquipable<Item>("Equipable/Items/Hats");
    LoadEquipable<Item>("Equipable/Items/Staffs");

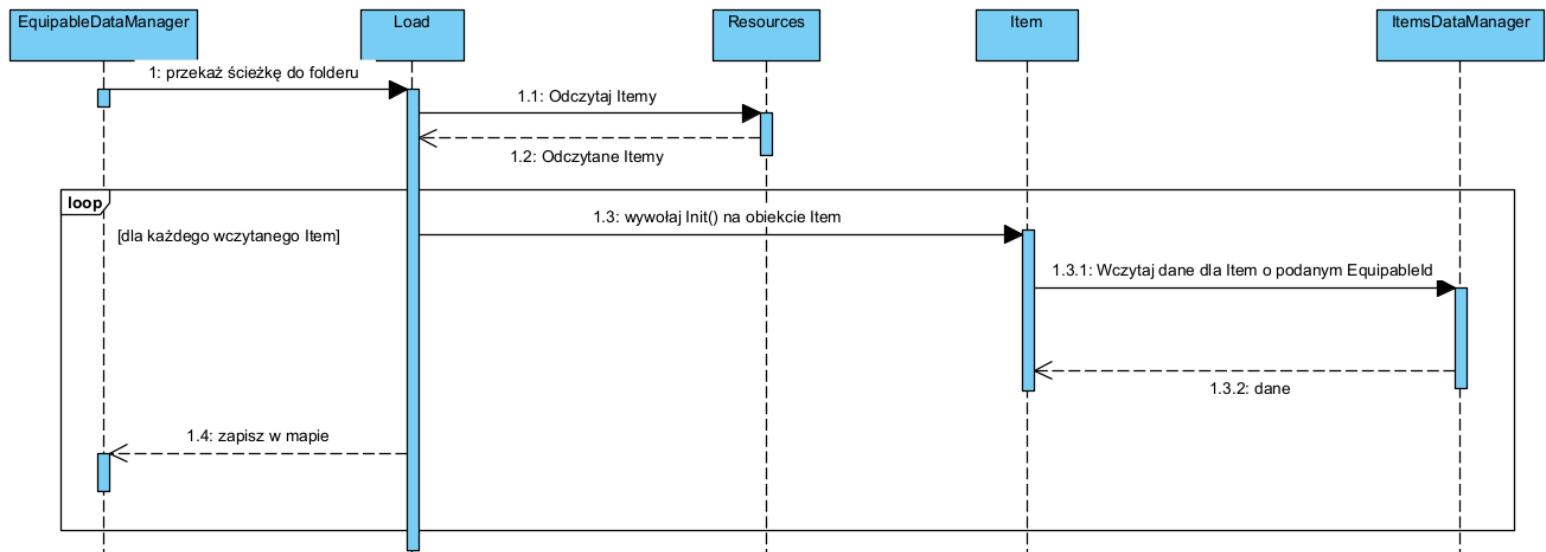
    LoadEquipable<SpellItem>("Equipable/SpellItems/Offensive");
    LoadEquipable<SpellItem>("Equipable/SpellItems/Defensive");
}

```

Wczytywanie Items i SpellItems

Obiekty zostają wczytane do słownika mapującego EquipableId do Equipable. Przy pomocy metod *GetItem*, *GetSpellItem* możliwe jest uzyskanie danego obiektu po przekazaniu *EquipableId*.

Jak już wspomniano, *Items* posiadają statystyki. Jednak statystyki zaimplementowane są jako mapa, a *ScriptableObject* nie pozwala na serializację map. W związku z tym same statystyki nie są serializowane wraz z *Item*, a przechowywane w klasie *ItemData*, która te statystyki przechowuje. Gdy *Items* są wczytywane, we wskazanej wyżej metodzie *Load* wywoływana jest na każdym *Equipable* metoda *init*, która - w przypadku *Item* - odczytuje ze statycznej klasy *ItemDataManager* odpowiednie *ItemData* i przypisuje do



wczytanego obiektu. Ostatecznie obiekt zapisywany jest w mapie.

Sytuacja jest podobna dla *SpellItem*, jednak w tym przypadku nie są wczytywane statystyki, gdyż *SpellItem* ich nie posiada.

- Gracz posiada dwa rodzaje ekwipunków:
- Ekwipunek na przedmioty (zwany dalej *Equipment*)
 - Ekwipunek na czary (zwany dalej *SpellEquipment*)

8.2.2 UX/UI

8.2.2.1 Equipment



Ekwipunek dzieli się na trzy moduły:

- Lewy - Statistics - pozwala na szybki podgląd podstawowych statystyk gracza
- Środkowy - Equipment - zawiera miejsca (*Sloty*) na przedmioty. Przedmiot może być przypisany do slotu tylko jeżeli jest tego samego typu co slot - np. pierścionek może być przypisany tylko do jednego z dwóch slotów na pierścienie.
W centralnej części panelu znajduje się opis statystyk oferowanych przez zaznaczony przez kursor przedmiot. Poniżej znajduje się jego zdjęcie oraz opis. Na górze znajduje się ikona kosza, która pozwala na wyrzucenie z ekwipunku niepotrzebnych przedmiotów.
- Prawy - to miejsce na wszystkie posiadane przez gracza przedmioty *Items*.

8.2.2.2 Spells Equipment

Jest to ekwipunek przeznaczony jedynie na obiekty SpellItem - a więc zaklęcia i czary.

Jego centralną część stanowi ekran w którym dostosować można aktualnie używane czary - po ich doborze pojawią się one w *SpellWheel*. Jeżeli nie zostanie przypisany żaden czar, w SpellWheel znajdzie się na tej pozycji czar domyślny.

Po prawej stronie umieszczona jest lista zebranych przez gracza, dostępnych czarów. Podobnie jest w *Equipment* na górze widnieje ikona kosza, umożliwiająca wyrzucenie niechcianego SpellItem.



Na dole widoczne jest zdjęcie i opis aktualnie zaznaczonego kursorem *SpellItem*.

8.3 System skalowania obrażeń (Łukasz Małecki)

8.3.1 Idea

Statystyki Entity muszą wpływać na obrażenia zadawane przez ich zaklęcia oraz na obrażenia otrzymywane od zaklęć innych Entities. Skalowanie to jest rozróżniane według typu (np. ogień, fizyczne, światło) oraz od tego, czy są to obrażenia otrzymywane, czy zadawane innym.

Statystyki podstawowe mają być mniej więcej równo użyteczne. Poza oczywistym skalowaniem się obrażeń zgodnych z typem (czyli np. obrażeń magii ognistej z fireDamage), będą również dodatkowe bonusy wynikające ze statystyk podstawowych.

Bonusy te mają zachęcić gracza do eksperymentowania z różnymi kombinacjami ekwipunku w zależności od używanych czarów. Same skalowania danych elementów ze statystykami podstawowymi są po części inspirowane systemem magii w Avatar: The Last Airbender, gdzie inteligencja nie jest jedynym czynnikiem mocy użytkownika magii (np. ziemia jest mocno powiązana z siłą fizyczną, a powietrze ze zręcznością). Oczywiście nie jest to dokładna kalka, bo w samej serii nie jest dokładnie określone, co wpływa na moc użytkownika magii, ale sam koncepcja i filozofia jest dobrą bazą, aby na niej zbudować system nieopierający się jedynie na inteligencji (jak w większości gier).

Inteligencja będzie mieć pewien wpływ na każde ofensywne obrażenia (z wyjątkiem fizycznych). Wpływ ten będzie niewielki, ale ma to zapewnić istnienie statystyki o bardziej ogólnym zastosowaniu, dla graczy, którzy nie lubią się specjalizować w typach magii i chcą pozostać bardziej elastycznymi.

8.3.2 Skalowanie ze statystykami podstawowymi - koncepcja

W każdym podpunkcie będzie wymieniony element, którego on dotyczy oraz statystyki w kolejności od mającego największy wpływ, do mającego najmniejszy wpływ. Dobór statystyk ma pewne uzasadnienie pseudofilozoficzne, które może mieć wpływ na fabułę, ale nie będzie wymieniane w ramach dokumentacji.

Warto zaznaczyć, że nie będą wymienione oczywiste statystyki wpływające, np. FireDamage dla ognia. Ponadto na każdy z elementów będzie w małym stopniu wpływać inteligencja, jako statystyka, którą można wykorzystać chcąc mieć najbardziej elastyczną kombinację.

- Ogień - inteligencja, siła
- Woda - zręczność, siła
- Ziemia - siła, obrona
- Powietrze - zręczność, inteligencja
- Elektryczność - siła, zręczność
- Ciemność - siła, statystyki ofensywne każdego typu
- Jasność - zręczność, statystyki defensywne każdego typu
- Fizyczne - siła, zręczność

8.3.2 Konkretne informacje dotyczące skalowania

Statystyki mają się zawierać mniej więcej w przedziale [-200, 200], wyższa wartość bezwzględna statystyk do pewnego momentu nadal będzie miała wpływ, ale znacznie mniejszy.

Statystyki podstawowe mają z zasady wpływać na obrażenia w sposób płaski, czyli niezależny od bazowych obrażeń zaklęcia. Natomiast statystyki elementowe mają wpływać na obrażenia na zasadzie zwiększenia/zmniejszenia o procent. Zatem im bazowo więcej obrażeń zadaje zaklęcie, tym większy wpływ na nie mają statystyki elementowe i vice versa.

Statystyki defensywne mają takie same przeliczniki, lecz wartość ich statystyk jest traktowana jako ujemna.

Statystyki po całym wyliczeniu nie mogą zwiększyć obrażeń o więcej niż 200% ani zmniejszyć o więcej niż 80% (zabezpieczenie to jest nakładane dopiero po wykonaniu wszystkich obliczeń związanych ze statystykami, zatem nie zakłóci obliczeń w przypadku, gdy jedna statystyka jest dodatnia, a druga ujemna).

8.3.2.1 - Statystyki elementowe

- Przy wartości statystyk elementowych 0 nie wywierają one żadnego wpływu na obrażenia.
- Przy wartości statystyk elementowych -200 zmniejszają one obrażenia o około 80% (mnożone są przez 0.2)
- Przy wartości statystyk elementowych 200 zwiększają one obrażenia o około 150% (mnożone są przez 2.5)
- Przy wartości statystyk elementowych -100 zmniejszają one obrażenia o 50% (mnożone są przez 0.5)
- Przy wartości statystyk elementowych 100 zwiększają one obrażenia o 75% (mnożone są przez 1.75)
- Przy wartości statystyk elementowych -700 zmniejszają one obrażenia o 90% (mnożone są przez 0.1)
- Przy wartości statystyk elementowych 700 zwiększają one obrażenia o 300% (mnożone są przez 4.0)

8.3.2.2 - Statystyki podstawowe

Modyfikatory do obrażeń od statystyk podstawowych są aplikowane przed modyfikatorami od statystyk elementowych.

- Np.
- bazowe obrażenia czaru: 100
- zmiana obrażeń ze względu na statystyki podstawowe: -10
- mnożnik obrażeń ze względu na statystyki elementowe: 1.5
- ostateczne obrażenia: $(100-10) \cdot 1.5 = 90 \cdot 1.5 = 135$

Modyfikatory do obrażeń od statystyk podstawowych są mniej ujednolicone i mocno zależne od danego typu magii.

Na ten moment inteligencja wzmacnia zaklęcia o z mnożnikiem 0.1 (atem 100 inteligencji daje nam 10 dodatkowych obrażeń).

Na ten moment obrona osłabia otrzymywane obrażenia z mnożnikiem 0.15 (zatem 100 obrony zmniejsza obrażenia o 15 punktów)

Zadawane obrażenia:

- Ogień - inteligencja*0.35, siła*0.25
- Woda - zręczność*0.4, siła*0.2
- Ziemia - siła*0.35, obrona*0.2
- Powietrze - zręczność*0.35, inteligencja*0.25
- Elektryczność - siła*0.4, zręczność*0.2

- Ciemność - siła*0.35, suma statystyk ofensywnych każdego typu *0.03
- Jasność - zręczność*0.35, suma statystyk defensywnych każdego typu *0.03
- Fizyczne - siła*0.4, zręczność*0.2

Otrzymywane obrażenia:

- Ogień - inteligencja*0.05
- Woda - zręczność*0.05
- Ziemia - siła*0.05
- Powietrze - zręczność*0.05
- Elektryczność - siła*0.05
- Ciemność - siła*0.05
- Jasność - zręczność*0.05
- Fizyczne - zręczność*0.05, siła*0.02 (oraz + 0.02 do mnożnika od obrony, czyli 0.17 obrona)

8.4 Sztuczna inteligencja, przeciwnicy (Łukasz Małecki)

8.4.1 Dynamiczne patrole przeciwników

8.4.1.1 - Zarys problemu

Problem: przeciwnik potrzebuje możliwości znalezienia punktu, w odpowiednim promieniu od niego, do którego może swobodnie się dostać (nie ma po drodze ściany, w danym punkcie znajduje się navmesh), preferowane jest, aby zminimalizować liczbę kolizji pomiędzy różnymi przeciwnikami

Ograniczenia: przeciwnik nie zna geometrii pomieszczenia, w którym się znajduje, nie wie również, gdzie znajdują się inni przeciwnicy

Rozwiązanie: przeciwnik wywołuje funkcję MakePath w X kierunkach wokół siebie, w tych samych kierunkach wywołuje Raycast wykrywający przeciwników, zapisujemy poprawne punkty zwrócone przez MakePath, ale preferujemy te, w których Raycast nie wykrył przeciwników, w przypadku nieznalezienia takiego punktu przeciwnik wróci do pozycji, z której ostatnio przyszedł, jeśli takiej pozycji nie ma, to będzie stał w miejscu.

8.4.1.2 - Opis algorytmu

1. Funkcja bool FindPatrolSpot(out Vector3 destination, float rotation=30f, float pathLength=8f, int sampleStep=8, float minimumPathPercent = 0.5f)
 - a. bool - informacja, czy udało się znaleźć cel patrolu
 - b. destination - zwraca tu cel patrolu, jeśli znaleziono, w przeciwnym wypadku wektor pozycji przeciwnika
 - c. rotation - ilość stopni o które chcemy obracać kierunek w każdej iteracji (definiuje liczbę iteracji)
 - d. pathLength - max odległość destination od obecnej pozycji przeciwnika
 - e. sampleStep - liczba kroków kontrolnych dla metody MakePath
 - f. minimumPathPercent - minimalna długość ścieżki, aby została zaakceptowana jako możliwa

2. Podaną wartość rotacji zmieniamy na wartość bezwzględną i jeżeli == 0, to zwracamy false
3. Tworzymy dwie puste listy wektorów 3D: List<Vector3> potentialSpots, List<Vector3> potentialBetterSpots
4. W pętli for(float currentRotation = 0f; currentRotation <= 360f; currentRotation += rotation)
 - a. zbieramy wektor kierunku, do którego zwrócony jest przeciwnik do tempVector
 - b. tempVector obracamy o currentRotation korzystając z funkcji RotateVector z EnemySM
 - c. ustalamy wartość Vector3 tempDestination na Vector3.zero
 - d. if(stateMachine.MakePathDestination(out tempDestination, tempVector, pathLength, sampleStep,minimumPathPercent))
 - i. tempDestination - tutaj zwróci potencjalny cel patrolu
 - ii. tempVector - wektor kierunku po rotacji
 - iii. pathLength - maksymalna długość ścieżki
 - iv. sampleStep - liczba kroków kontrolnych
 - v. minimumPathPercent - minimalny procent długości ścieżki, aby ją zaakceptować
 - vi. then dodaj tempDestination do potentialSpots
 - e. else
 - i. continue;
 - f. if(!stateMachine.Raycast(tempVector, pathLength*minimumPathPercent))
 - i. raycast tutaj sprawdza, czy w podanym kierunku są przeciwnicy
 - ii. then dodaj tempDestination do potentialBetterSpots
5. if(potentialSpots.Count == 0)
 - a. then return false;
6. if(potentialBetterSpots.Count != 0)
 - a. destination ustalamy losowy element z listy potentialBetterSpots
 - b. return true
7. destination ustalamy losowy element z listy potentialSpots
8. return true;

8.4.1.3 - Opis słowny MakePathDestination

Funkcja zwraca informację, czy da się zrobić ścieżkę w podanym kierunku o danej długości.

Funkcja przyjmuje za argumenty destination, do którego zwróci wynik ostateczny punkt, do którego może dostać się przeciwnik bez przeszkód; direction - kierunek, w którym mamy szukać ścieżki; pathLength - pożądana długość ścieżki; sampleStepCount - liczba punktów, w których uruchamiamy SamplePosition; minimumPathLength - minimalny procent długości ścieżki, aby uznać ją za stosowną do zwrócenia.

Normalizujemy wektor kierunku i w pętli for (int i = 1; i <= sampleStepCount; i++) przypisujemy destination wartość pozycji przeciwnika z dodanym wektorem kierunku przemnożonym przez i, pathLength oraz podzielonym przez sampleStepCount. W tejże pozycji uruchamiamy SamplePosition od navmesh, które sprawdza, czy w małej odległości od danego punktu znajduje się niewycięty navmesh. Jeżeli zwróci on true, to zapisujemy nasz destination jako dotychczasowy najlepszy, w przeciwnym wypadku

jeśli wartość zmiennej i jest większa/równa od sampleStepCount*minimumPathLength, wtedy break, w przeciwnym wypadku ustawiamy destination jako obecną pozycję gracza i zwracamy false.

Po pętli dla destination przypisujemy bestDestination.

Następnie korzystamy z navmesh.CalculatePath, który oblicza kawałek ścieżki (na ogół nie całą), jeśli ścieżka okaże się invalid, to zwracamy false, w przeciwnym wypadku zwracamy true.

8.4.2 Typy przeciwników (Łukasz Małecki)

8.4.2.1 - Wstęp

W grze mamy 4 główne typy przeciwników, którym odpowiadają konkretne maszyny stanów:

- kontaktowi - MeleeSM (szkielety)
- zasięgowi - RangedSM (gobliny)
- szamani - ShamanSM (szamani)
- golemy - GolemSM (kamienny golem, rogaty golem)

Wszystkie powyższe maszyny stanów dziedziczą po abstrakcyjnej klasie EnemySpecific, która zbiera uniwersalne elementy wszystkich 4 maszyn oraz ułatwia komunikację z resztą elementów systemu.

Proces tworzenia przeciwnika na ogół zawierał następujące elementy:

- Miałem/znajdowałem model postaci 3D z użytecznymi animacjami, pasujący do naszej gry
- Biorąc pod uwagę możliwości, jakie dawał asset, tworzyłem plan maszyny stanów zaczynając od ogólnej idei
- Wymyślałem konkretne stany i potrzebne dla nich warunki przechodzenia między sobą
- Ustalałem listę potrzebnych parametrów dla działania maszyny stanów
- Tworzyłem kod maszyny stanów i każdego jej stanów
- Importowałem model do projektu
- Dostosowywałem collider i system animacji do moich potrzeb
- Łączyłem model przeciwnika i jego system animacji z maszyną stanów

Wszelkie prace związane z łączeniem maszyn stanów konkretnie z systemem magii miały miejsce w późniejszym etapie wytworzania gry.

Później wytworzone maszyny stanów miały bardziej rozbudowany proces planowania.

8.4.2.2 - Przeciwnik kontaktowy

Plan działania przeciwnika kontaktowego opiera się przede wszystkim na opisie dostępnych dla niego stanów i przejść między nimi. Jest to najprostsza z maszyn stanów, zatem wymagała najmniej refleksji.

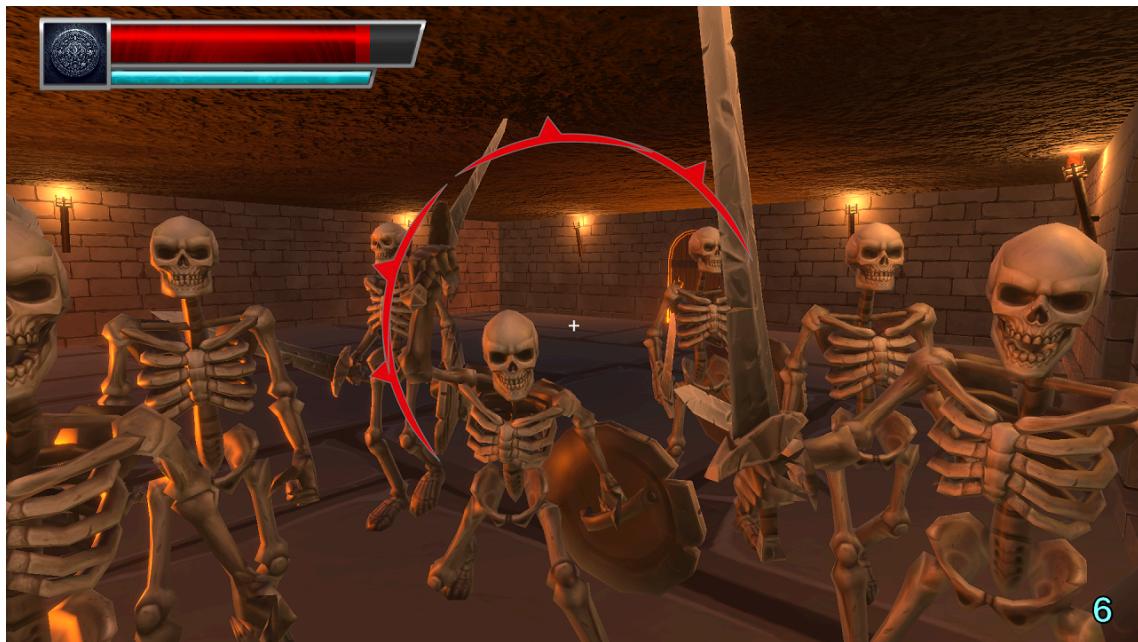
Fragmenty oryginalnego dokumentu:

- **Rest** – przeciwnik stoi w miejscu, po upłynięciu czasu, gdy **restTimePassed > restTime**, powrót do stanu **Patrol**, zmienna **isRestForced** będzie decydować, czy przeciwnik będzie mógł zmienić stan na **Chase**, gdy gracz będzie w pobliżu, zmienna **reactionTime** będzie decydować po jakim czasie przeciwnik wejdzie w stan **Chase** po tym, jak gracz wejdzie w jego zasięg widzenia
- **Patrol** – przeciwnik porusza się z punktu A do punktu B, co jakiś czas wchodzi wchodzi w stan **Rest**, przeciwnik zmienia stan na **Chase**, jeżeli gracz będzie w pobliżu
- **Chase** – przeciwnik porusza się w kierunku gracza aż znajdzie się w odpowiednim od niego zasięgu, aby wykonać atak, wtedy zmienia stan na **Attack**, jeżeli gracz jest niewykrywalny, to przeciwnik stoi w miejscu przez pewien czas, jeżeli gracz się znowu pojawi, to od razu zaczyna go gonić, w przeciwnym wypadku wchodzi w stan **Patrol**
- **Attack**– przeciwnik zaczyna wykonywać atak, jeśli gracz jest nadal w zasięgu po wykonaniu ataku, to po cooldownie na atak wykonuje kolejny, w przeciwnym wypadku przechodzi do stanu **Chase**

Przeciwnik ten nie jest szczególnym wyzwaniem w pojedynkę, jednak zostanie przez niego osaczony z wielu stron bez możliwości radzenia sobie z grupami przeciwników może być zabójcze.

Atak przeciwnika został zrealizowany jako niewidzialny trigger pojawiający się przed przeciwnikiem w miejscu wykonywania animacji ataku mieczem, który szybko znika po zakończeniu animacji.

Istniała również koncepcja, aby przeciwnik nie wytraçał prędkości w momencie wyprowadzenia ataku, jednak zrezygnowaliśmy z tego, z powodu nienaturalnego wyglądu takiego zdarzenia oraz zbyt dużego poziomu trudności w unikaniu.



State times	
Rest Time	3
Patrol Time	8
Chase Time	4
Hurt Time	0.3
Reaction Time	0.5
Range	
Attack Range	2
Detection Range	19
Attack	
Attack Cooldown	0.2

```

public void UpdateState()
{
    if (!mainStateMachine.IsGoalInRange())
    {
        stateTimePassed += Time.deltaTime;
    }
    else
    {
        stateTimePassed = 0f;
    }
    if(stateTimePassed >= stateTime)
    {
        mainStateMachine.ChangeState(mainStateMachine.patrol);
        return;
    }

    if (mainStateMachine.IsGoalInAttackRange())
    {
        mainStateMachine.ChangeState(mainStateMachine.attack);
        return;
    }
}

```

Prostota MeleeSM, lewo - parametry maszyny stanów, prawo - metoda update stanu Chase

Przeciwnik pojawia się w dwóch wariantach:

- SkeletonWeak - wersja bez tarczy, słabsza, pojawia się tylko na 1-ym piętrze, dla ułatwienia, gdy gracz nie ma przedmiotów
- SkeletonBase - wersja z tarczą, standardowa, pojawia się na każdym piętrze

Oba warianty są wrażliwe na magię ognia i ziemi.

8.4.2.3 - Przeciwnik zasięgowy

Plan działania przeciwnika zasięgowego zawiera 2 dodatkowe stany w porównaniu do kontaktowego. Pomysł na uciekanie przeciwników przed graczem wziął się z tego, że w kulturze goblinów nierzadko postrzegane są jako tchórzliwe istoty, ponadto taki dodatkowy typ responsywności na działania gracza zapewnia lepszą immersję. W samym planie są również zawarte plany na temat sposobu implementacji ucieczki oraz sposobu wybierania celu ataku. Po dyskusji planu z resztą zespołu wybrane zostały rozwiązania zaznaczone na zielono. O decyzjach przede wszystkim zadecydowała prostota implementacji, którą PM uznał za kluczową, aby dotrzymać terminu. Ostateczna wersja działa praktycznie tak jak opisane w planie, z zastrzeżeniem do sekcji opisu ucieczki przeciwnika przed graczem, która działa w bardziej wyrafinowany sposób. Różnice te zostaną wymienione na fioletowo.

Fragmenty oryginalnego dokumentu:

- **Rest** – przeciwnik stoi w miejscu, po upłynięciu czasu, gdy **restTimePassed > restTime**, powrót do stanu **Patrol**, zmienna **isRestForced** będzie decydować, czy przeciwnik będzie mógł zmienić stan na **Chase**, gdy gracz będzie w pobliżu, zmienna **reactionTime** będzie decydować po jakim czasie przeciwnik wejdzie w stan **Chase** po tym, jak gracz wejdzie w jego zasięg widzenia

- **Patrol** – przeciwnik porusza się z punktu A do punktu B, co jakiś czas wchodzi wchodzi w stan **Rest**, przeciwnik zmieni stan na **Chase**, jeżeli gracz będzie w pobliżu
- **Chase** – przeciwnik porusza się w kierunku gracza aż znajdzie się w odpowiednim od niego zasięgu, aby wykonać atak, wtedy zmienia stan na **Prepare**, jeżeli gracz jest niewykrywalny, to przeciwnik stoi w miejscu przez pewien czas, jeżeli gracz się znowu pojawi, to od razu zaczyna go gonić, w przeciwnym wypadku wchodzi w stan **Patrol**
- **Attack**– przeciwnik zaczyna wykonywać atak, jeśli gracz jest nadal w zasięgu po wykonaniu ataku i **attacksNumberLeft** > 0, to po cooldownie na atak wykonuje kolejny, jeśli gracz nie jest w zasięgu, to przechodzi do stanu **Chase**, jeżeli **attacksNumberLeft** <= 0, to przeciwnik przechodzi w stan **Prepare**, zmienna **attacksNumber** określa maksymalną liczbę ataków w „serii”
- **Prepare** – przeciwnik jest w odpowiednim dystansie od celu i przygotowuje się do wykonania ataku, po odpowiednim czasie przechodzi do stanu **Attack**, jeżeli przed upływnięciem tego czasu cel będzie za blisko (w **dangerRange**), to jeżeli zmienna **isAttackForced** nie będzie true, to przeciwnik wchodzi w stan **Flee**, jeżeli cel wyjdzie z zasięgu ataku, to przeciwnik przechodzi w stan **Chase**
- **Flee** – przeciwnik oddala się od celu, po określonym czasie/dotarciu do odpowiedniego punktu ustawia zmienną **forcedAttack** na true, jeżeli cel jest w zasięgu, to również wchodzi w stan **Prepare**, w przeciwnym wypadku wchodzi w stan **Chase**

Kolejność od najbardziej, do najmniej prawdopodobnych w wykonaniu

1. **Flee** odbywać może się na różne sposoby, propozycje:

- a. W pewnym promieniu od przeciwnika przeszukiwana jest część punktów na okręgu (np. co kilka stopni) i wybierany jest punkt, do którego może zostać utworzona ścieżka (preferowanie jak najdalej od celu) (Zalety: elastyczność, indywidualność przeciwników; Wady: potencjalne znalezienie się punktu w innym pokoju? (rozwiązywanie, branie pod uwagę długości potencjalnej ścieżki, ustalenie maksymalnej długości ścieżki, np. 1.5*promień), potencjalna trudność w implementacji, potencjalne uciekanie w kierunku gracza? (rozwiązywanie: nieprzeszukiwanie okręgu, ale półokręgu, potencjalny problem w przypadku, gdy przeciwnik otoczony jest ścianami z 3 stron potencjalnie rozwiązyany przez 1.5*promień))
- b. Na mapie będzie znajdować się kilka punktów, do których przeciwnik może uciec i wybierany punkt będzie na podstawie kryteriów: możliwie blisko przeciwnika + możliwie daleko od celu + aby cel był w zasięgu + aby cel nie był za blisko (Zalety: łatwość implementacji; Problemy: ucieczka wielu przeciwników do tego samego miejsca, silne powiązanie przeciwnika z danym pokojem)
- c. Przeciwnik ma uciekać w tym samym kierunku, w którym porusza się gracz, aż nie na trafi na ścianę, w tym momencie ma się obrócić w stronę gracza i próbować atakować, ewentualnie uderza w ścianę i się stunuje (Zalety: łatwość implementacji, łatwa przewidywalność dla gracza; Problemy: potencjalne częste napotykanie bycia pod

ścianą, A.I. postrzegane jako głupie) - przeciwnik próbuje uciekać w kierunku zgodnym z wektorem pozycja gracza -> pozycja przeciwnika na dystans o maksymalnej długości **fleeDistance**, jeżeli w tym kierunku nie ma możliwości utworzenia ścieżki dłuższej niż **fleeDistance*0.5**, to przeciwnik ustawia **isAttackForced** i wchodzi do stanu **Prepare**, jeżeli ścieżkę udało się utworzyć to przeciwnik ucieka w kierunku celu i gdy dojdzie do niego lub minie **maxFleeDistance**, to ustawia **isAttackForced** na true i wchodzi do stanu **Prepare**

2. Cel ataków przeciwnika

- a. Przeciwnik może strzelać dokładnie tam, gdzie gracz jest w momencie strzału (Zalety: bardzo prosta implementacja, łatwa do zrozumienia mechanika; Wady: potencjalnie bardzo niska skuteczność przeciwników zasięgowych)
- b. Przeciwnik może brać pod uwagę wektor prędkości gracza i odpowiednio zmienić miejsce strzału o ustaloną liczbę stopni (Zalety: potencjalnie lepsza skuteczność przeciwnika; Wady: wymagany sposób na określenie wektora prędkości celu)
- c. Przeciwnik może brać pod uwagę wektor prędkości gracza i proporcjonalnie do niego zmienić miejsce strzału, aby trafić go jeżeli wektor prędkości zostanie zachowany (Zalety: zdecydowanie lepsza skuteczność przeciwnika; Wady: wymagany sposób na określenie wektora prędkości celu, potencjalne bycie niefair wobec gracza, gdy gracz ma bardzo wysoką prędkość (potencjalne rozwiązanie: limit prędkości lub zmiany kąta wystrzału), potencjalnie zbyt duże utrudnienie dla gracza)

Stworzenie przeciwnika zasięgowego zaowocowało utworzeniem metody **MakePathDestination** (opisanej w **8.4.1.3**), która była konieczna do zapewnienia osiągalności punktu wybranego w stanie **Flee**.

Przeciwnik stanowi rzeczywiste zagrożenie dla gracza (zwłaszcza, że zazwyczaj nie występuje w pojedynkę) i zmusza go do bycia w ciągłym ruchu.

Strzały goblina są w rzeczywistości zakleciem wykonanym w ramach modułu Systemu magii. Na ogół gobliny mają szansę na wystrzelanie jednej z dwóch strzał: jedna zwykła, druga mająca efekt statusu (np. ślepota, podpalenie). Stosunek prawdopodobieństwa tych strzał wynosi na ogół 6:1, choć może się różnić między wariantami.



State times	
Rest Time	3
Patrol Time	5
Chase Time	4
Prepare Time	2.5
Hurt Time	0.3
Reaction Time	0.5
Max Flee Time	5
Range	
Attack Range	14
Detection Range	17
Danger Range	5
Attack	
Attack Cooldown	1
Attack Series Number	2
Flee	
Flee Distance	3
Flee Steps	8
Cast Point	CastPoint (Transform)

```

public void UpdateState()
{
    if (mainStateMachine.IsGoalInRange())
    {
        stateTimePassed += Time.deltaTime;
    }
    else
    {
        stateTimePassed -= Time.deltaTime;

        if (stateTimePassed < 0)
        {
            mainStateMachine.ChangeState(mainStateMachine.chase);
            return;
        }
    }
    if(mainStateMachine.IsGoalInDangerRange() && !isAttackForced)
    {
        mainStateMachine.ChangeState(mainStateMachine.flee);
        return;
    }

    if (stateTimePassed >= stateTime)
    {
        mainStateMachine.ChangeState(mainStateMachine.attack);
        return;
    }
}

```

Lewo - odpowiednia możliwość parametryzacji przeciwnika, Prawo - stan prepare ma możliwość przejścia do jednego z aż 3 stanów (return konieczne po każdym ChangeState, aby zapobiec błędowi w przypadku, gdy warunki dla więcej niż 1 stanu są spełnione)

Mamy 4 warianty goblińskiego łucznika, 2 wykorzystują zmodyfikowaną w programie GIMP pod kątem koloru wersję assetu Goblin Necro:

- GoblinBase (zielony) - bazowy goblin, odporny na magię powietrza, wrażliwy na magię ziemi i elektryczność, drugi typ ataku nakłada truciznę
- GoblinFire (czerwony/pomarańczowy) - ognisty goblin, odporny na magię ognia i ciemność, wrażliwy na magię wody, ziemi i elektryczność, drugi typ ataku nakłada podpalenie

- GoblinDark (fioletowy) - goblin ciemności, odporny na magię powietrza i ciemność, niewielka wrażliwość na ziemię i elektryczność, niższe obrażenia, wyższa obrona, drugi typ ataku nakłada oślepienie
- GoblinElite (żółty) - elitarny goblin, odporny na magię powietrza, światła, elektryczności, wrażliwość na magię wody, ziemi, ciemności, najlepsze statystyki ze wszystkich goblinów, nie ma specjalnego ataku, ale w serii jest w stanie wystrzelić 3 strzały

Wersja fioletowa i żółta zostały osiągnięte przy użyciu podmiany kolorów tekstu w programie do obróbki graficznej GIMP.

8.4.2.4 - Szaman

Plan tworzenia szamana powstał jeszcze zanim został skrytalizowany system działania zaklęć dla przeciwników oraz nie było pewności, co do możliwości istnienia zaklęć obniżających/wzmacniających czasowo statystyki entities. Ostatecznie BattleCry wykorzystywany jest do rzucania rzadziej, ale potężniejszych zaklęć, a FocusCast do bardziej typowych, ale nadal bardziej zjawiskowych. QuickCast wystrzeliwuje prosty pocisk zadający obrażenia odpowiedniego typu. Za knockbackSpell służą zaklęcia blisko zasięgowe lub defensywne. Zrealizowana została idea zwiększenia odporności szamana w trakcie trwania stanu **Focus**. Ale znaczna większość informacji jest aktualna.

Fragmenty oryginalnego dokumentu:

Ogólny pomysł

Szaman ma specjalizować się w zasięgowych atakach. Różni się tym, że jest w stanie on korzystać z zaklęć. Zaklęcie rzucane podczas grania animacji BattleCry ma być albo buffem dla przeciwników lub debuffem dla gracza Curse(?). Poza tym zaklęciem będzie posiadał QuickCast, szybki w użyciu wystrzał pocisku (animacja z laską do przodu). Ponadto będzie posiadał FocusSpell, rzucenie tego czaru ma zajmować więcej czasu, ale powinien być potężniejszy, np. duży pocisk podążający za graczem, obszarowe zaklęcie, coś na kształt lasera, etc.

Potencjalny pomysł, może dałoby się zrobić tak, aby szamanowi dałoby się equipnąć z poziomu edytora dowolne zaklęcie? Ewentualnie mógłby mieć metodę sprawdzającą jego kompatybilność z danym zaklęciem, np. czy ma odpowiednie metody, które mogłyby mu pozwolić je rzucić etc.

Stany

Rest – przeciwnik stoi w miejscu, po upłynięciu czasu, gdy **restTimePassed > restTime**, powrót do stanu **Patrol**, zmienna **isRestForced** będzie decydować, czy przeciwnik będzie mógł zmienić stan na **Chase**, gdy gracz będzie w pobliżu, zmienna **reactionTime** będzie decydować po jakim czasie przeciwnik wejdzie w stan **Chase** po tym, jak gracz wejdzie w jego zasięg widzenia

Patrol – przeciwnik porusza się z punktu A do punktu B, co jakiś czas wchodzi wchodzi w stan **Rest**, przeciwnik zmieni stan na **Chase**, jeżeli gracz będzie w pobliżu

Chase – przeciwnik porusza się w kierunku gracza aż znajdzie się w odpowiednim od niego zasięgu, aby wykonać atak, wtedy zmienia stan na **Focus**, jeżeli gracz jest niewykrywalny, to przeciwnik stoi w miejscu przez pewien czas, jeżeli gracz się znowu pojawi, to od razu zaczyna go gonić, w przeciwnym wypadku wchodzi w stan **Patrol**

Attack– przeciwnik ma do wyboru kilka zaklęć o różnym efekcie po użyciu:

- Szybki atak – wyrzuca prosty pocisk w kierunku gracza, po użyciu szaman odczeka połowę **focusTime** i może użyć kolejnego losowego zaklęcia
- FocusSpell – używa silniejszego zaklęcia, po użyciu szaman wchodzi w stan **Focus**
- BattleCry – używa buffa lub debuffa, po użyciu wchodzi w stan **Focus**
- KnockbackSpell(?) – używa zaklęcia odrzucającego gracza, po użyciu wchodzi w stan **Focus**

przeciwnik po wejściu w stan **Attack** wybiera jedno z zaklęć (poza Knockback) i go używa, jeżeli jednak zmienna **isInDanger** == true, to używa Szybkiego ataku lub KnockbackSpell(?), jeśli po Szybkim ataku gracz nie jest w zasięgu, to przechodzi do stanu **Chase**, w przeciwnym wypadku odczeka 50% **focusTime** i wybiera kolejne losowe zaklęcie, w przypadku innych zaklęć przechodzi do stanu **Focus**

Focus (animacja Block)– przeciwnik jest w odpowiednim dystansie od celu i przygotowuje się do wykonania ataku, po odpowiednim czasie przechodzi do stanu **Attack**, jeżeli przed upłynięciem tego czasu cel będzie za blisko (w **dangerRange**), to zmienna **isInDanger** zmienia się na true, jeżeli zmienna **isInDanger** == true, to przeciwnik potrzebuje tylko 50% **focusTime**, aby wejść w stan **Attack**, ale będzie mieć dostęp tylko do szybkiego ataku (lub odrzucenia gracza), jeżeli cel wyjdzie z zasięgu ataku, to przeciwnik przechodzi w stan **Chase**

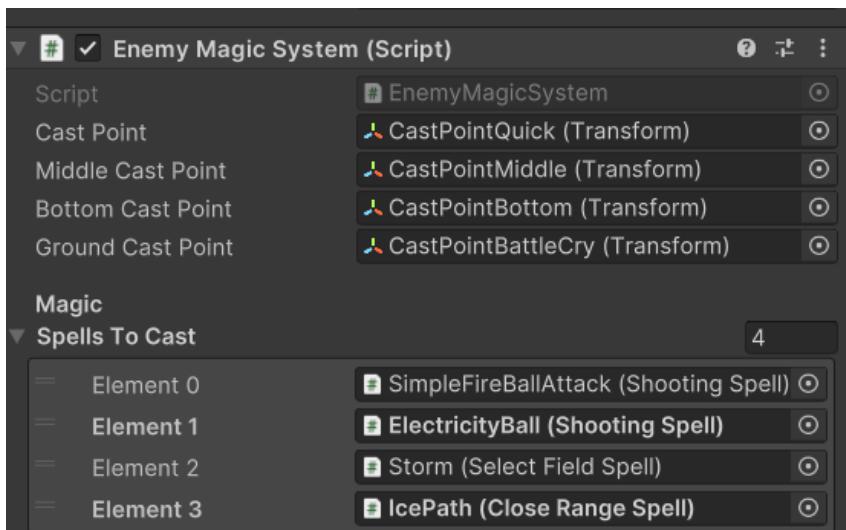
Szaman był pierwszym przeciwnikiem, który z założenia miał korzystać z systemu magii, w czasie tworzenia nie było pewności, jak ten system będzie wykorzystywany przez przeciwników, zatem przez pewien czas przeciwnik aktywował odpowiednie animacje zgodnie z planem, ale w rzeczywistości nie rzucał żadnych zaklęć.

Szaman jest przeciwnikiem, który może stanowić poważne zagrożenie dla nieuważnego gracza, ich podstawowe ataki są dosyć słabe, ale reszta ich arsenalu potrafi poważnie uszkodzić gracza, ponadto jest w stanie nałożyć nieprzyjemne efekty statusu.

Szamani z idei mieli różnić się od goblinów byciem niewzruszonym, ich reakcją na zagrożenie miała być walka. Gdy gracz podejdzie za blisko albo zostanie przebitý silnym zaklęciem blisko zasięgowym lub szaman rozsądnie osłoni się przed natarciem.



▼ Spell Chance Weights	
= Element 0	4
= Element 1	2
= Element 2	1
+ -	
State times	
Rest Time	3
Patrol Time	7
Chase Time	4
Focus Time	3
Hurt Time	0.3
Reaction Time	0.5
Range	
Attack Range	14
Detection Range	17
Danger Range	4.5
Attack	
Attack Cooldown	0.2
Quick Cast Point	CastPointQuick (Transform)
Focus Cast Point	CastPointFocus (Transform)
Battle Cry Point	CastPointBattleCry (Transform)
Knockback Cast Point	CastPointKnockback (Transform)



SpellChanceWeights pochodzi z EnemySpecific i służy do nadawania wag prawdopodobieństwu wystąpienia zaklęcia o odpowiadającym indeksie, KnockbackSpell nie ma swojej wagi, bo szansa na niego ma być tylko, gdy gracz jest blisko. Szaman ma dodane wiele punktów wylotu zaklęć typu Shooting Spell, aby w zależności od animacji zmienić CastPoint w EnemyMagicSystem (middle, bottom i ground CastPoints zostają zawsze te same)

Mamy 2 warianty szamana:

- ShamanBase - bazowy szaman posługujący się magią ognia i elektryczności, odporny na ogień, ciemność i elektryczność, wrażliwy na wodę, ziemię i jasność
- ShamanWater - szaman wodny posługujący się magią wody i lodu, odporny na wodę, ziemię, ciemność, wrażliwy na ogień, powietrze, elektryczność, jasność, generalnie bardziej wytrzymały

8.4.2.5 - Golem

Dla golema znalezione zostały świetne 2 assety i zamiast tworzyć 2 różne maszyny stanów, postanowiono, że jedna maszyna stanów będzie obsługiwać oba typy. Spowodowało to sporo usprawnień, które nie były wspomniane w dokumencie. Powstały osobne zmienne boolowskie, których ustawienie decydowało o działaniu golema w pewnej sytuacji. Ponieważ jeden z golemów ma animację rzucania, a drugi ma świetną animację szarża, zdefiniowano 2 różne zachowania w zależności od tego, czy golem może szarżować czy rzucać.

Fragmenty oryginalnego dokumentu:

Ogólny pomysł

Golem powinien być wytrzymały, lecz dość powolnym przeciwnikiem. Jednakże, aby nadal mógł stanowić zagrożenie będzie posiadał stan **Rage**. Kiedy w trakcie gonienia celu oddali się on na zbyt daleki dystans od golema wchodzi on w stan **Focus** (ładuje się) i po tym wchodzi w stan **Rage**. Sam stan **Rage** może być różnie rozwiązany, jednym z pomysłów jest **Charge**, aby golem zaczął poruszać się w stronę celu z coraz to wyższą prędkością (np. na czas bycia w tym stanie zwiększamy speed agenta i zmieniamy odpowiednio przyspieszenie, można pobawić się z angular speed). Przy

osiągnięciu prędkości granicznej lub odpowiedniego dystansu od celu można ustawić, że cel golema nie będzie się już aktualizował (aby przesadnie nie skręcił na gracza lub nie zwolnił). Innym pomysłem jest po prostu ogólny buff do prędkości, odporności, może szybkości animacji ataku, etc. Innym konceptem jest również rzucenie jakiegoś silnego zaklęcia/rzucenie kamieniem (mini legion rock golem).

Stany

Rest – przeciwnik stoi w miejscu, po upłynięciu czasu, gdy **restTimePassed > restTime**, powrót do stanu **Patrol**, zmienna **isRestForced** będzie decydować, czy przeciwnik będzie mógł zmienić stan na **Chase**, jeżeli gracz będzie w pobliżu, zmienna **reactionTime** będzie decydować po jakim czasie przeciwnik wejdzie w stan **Chase** po wykryciu przeciwnika, jeżeli **isRestForced == true**, to niezależnie czy gracz jest blisko, golem po upłynięciu czasu wchodzi w stan **Chase**

Patrol – przeciwnik porusza się z punktu A do punktu B, co jakiś czas wchodzi w stan **Rest**, przeciwnik zmieni stan na **Chase**, jeżeli gracz będzie w pobliżu

Chase – przeciwnik porusza się w kierunku gracza aż znajdzie się w odpowiednim od niego zasięgu, aby wykonać atak, wtedy zmienia stan na **Attack**, jeżeli gracz będzie przez pewien czas za daleko od Golema, to wchodzi w stan **Focus** (chyba, że jest już pod wpływem buffa ze stanu **Focus**), jeżeli gracz jest niewykrywalny, to przeciwnik stoi w miejscu przez pewien czas, jeżeli gracz się znowu pojawi, to od razu zaczyna go gonić, w przeciwnym wypadku wchodzi w stan **Rest** lub **Patrol**

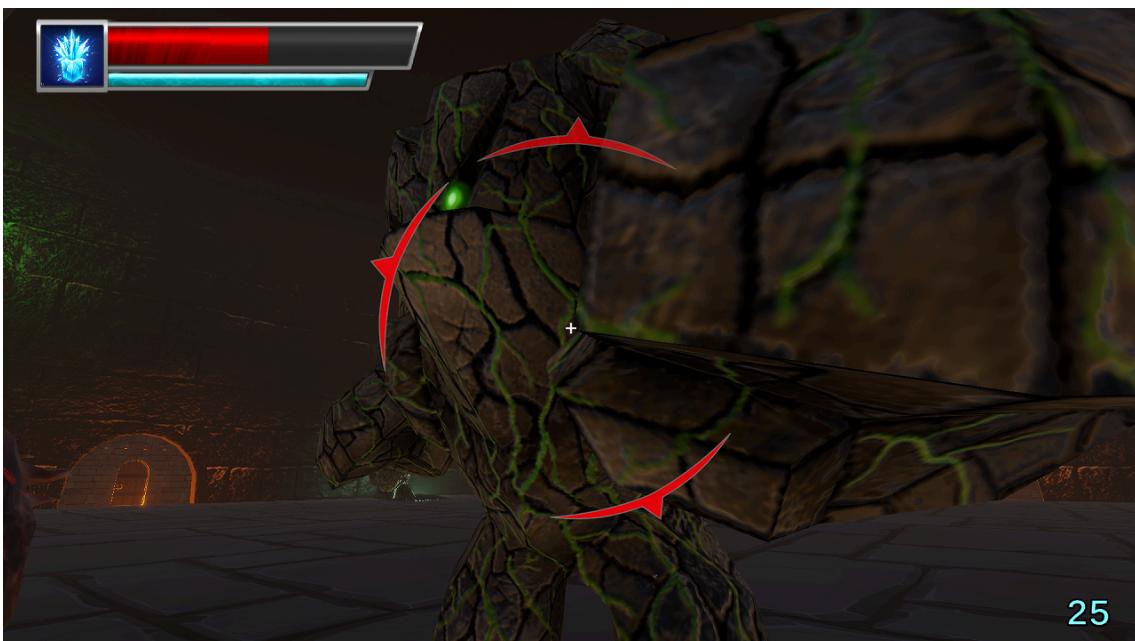
Attack – przeciwnik wykonuje atak wręcz, jeśli po wykonaniu ataku gracz nadal będzie w zasięgu to po odczekaniu cooldownu wykona on kolejny atak, jeżeli gracz nie będzie już w zasięgu to przejdzie do stanu **Chase**

Focus (animacja Victory/Rage) – przeciwnik przygotowuje się do wykonania ataku, po odpowiednim czasie przechodzi do stanu **Rage**, jeżeli przed upłynięciem tego czasu cel będzie w zasięgu zwykłego ataku, to przejdzie do stanu **Attack**, w stanie **Focus** golem powinien mieć zmniejszone otrzymywane obrażenia, czas trwania tego stanu powinien być warunkowany zmienną **focusTime**, jeżeli wartość ta będzie wynosić ≤ 0 , to wykorzystany zostanie oryginalny czas animacji

Rage – golem wykonuje jedną z silnych akcji (przy pomocy booli możemy ustalić, czy dany golem ma do konkretnej akcji dostęp). Po wykonaniu akcji wchodzi w stan **Rest** (forced) lub **Chase**:

- **Charge** – golem zaczyna iść w stronę gracza stopniowo przyspieszając do znacznie wyżej prędkości niż podstawowa i uderza gracza, po tym wchodzi w stan **Rest** (forced)
- **Throw** – golem dokonuje rzutu kamieniem (lub zaklęciem) w gracza po czym wchodzi w stan **Chase**
- **Reinforce** – golem wzmacnia się (nie jest grana dodatkowa animacja, ale może jakiś efekt graficzny się pojawia typu poświata) na czas działania zwiększana jest prędkość, obrażenia etc. od razu wchodzi do stanu **Chase**

Plan był tworzy, gdy nadal nie było pewności, co do zakresu możliwości zakleć, zatem istniał pomysł na zaklęcie wzmacniające, który ostatecznie został zażegnany. Pomysł na stan rage, jak opisane w wycinku z oryginalnego dokumentu, poparty był umożliwieniem golemom bycia rzeczywistym zagrożeniem. Szarża nie jest niezawodna i bardzo często nie trafia gracza, ale nadal daje mu poczucie zagrożenia, gdy rogaty golem jest z nim w pokoju.



▼ Spell Chance Weights		4
= Element 0	0	
= Element 1	0	
= Element 2	2	
= Element 3	1	
		+
		-
State times		
Rest Time	3	
Patrol Time	7	
Reaction Time	0.5	
Rage Time	3	
Chase Time	5.5	
Range		
Attack Range	4	
Detection Range	17	
Rage Range	13	
Attack		
Attack Cooldown	0.3	
Hurt Time	0.3	
Cast Point Attack	 FistAttackPoint (Transform) 	
Rage		
Focus Animation Count	1	
Can Throw	<input checked="" type="checkbox"/>	
Throw Point	 Index_Proximal_R (Transform) 	
Can Headbutt	<input type="checkbox"/>	
Rush Speed Percent	3	
Max Rush Time	12	
Focus Ongoing	<input type="checkbox"/>	

Przykładowe wypełnienie parametrów dla kamiennego golema, który jest w stanie rzucić zaklęcia. W SpellWeights pierwsze 2 miejsca są zarezerowane na podstawowy atak i potencjalny wariant tego ataku nakładający efekt. Reszta to zaklęcia, które może wyrzucić golem.

Mamy 2 warianty golema:

- RockGolem - kamienny golem, bardzo wysoka odporność na ziemię, elektryczność, fizyczne, pewna odporność na światło, wrażliwość na ogień, wodę, ciemność, wysoka obrona i siła, korzysta z magii ziemi
- HornedGolem - rogaty golem, bardzo wysoka odporność na wiatr, pewna odporność na wodę i fizyczne, wrażliwość na ogień, elektryczność, lekka wrażliwość na ziemię, wysoka obrona i siła, używa wyłącznie ataków o typie fizycznym

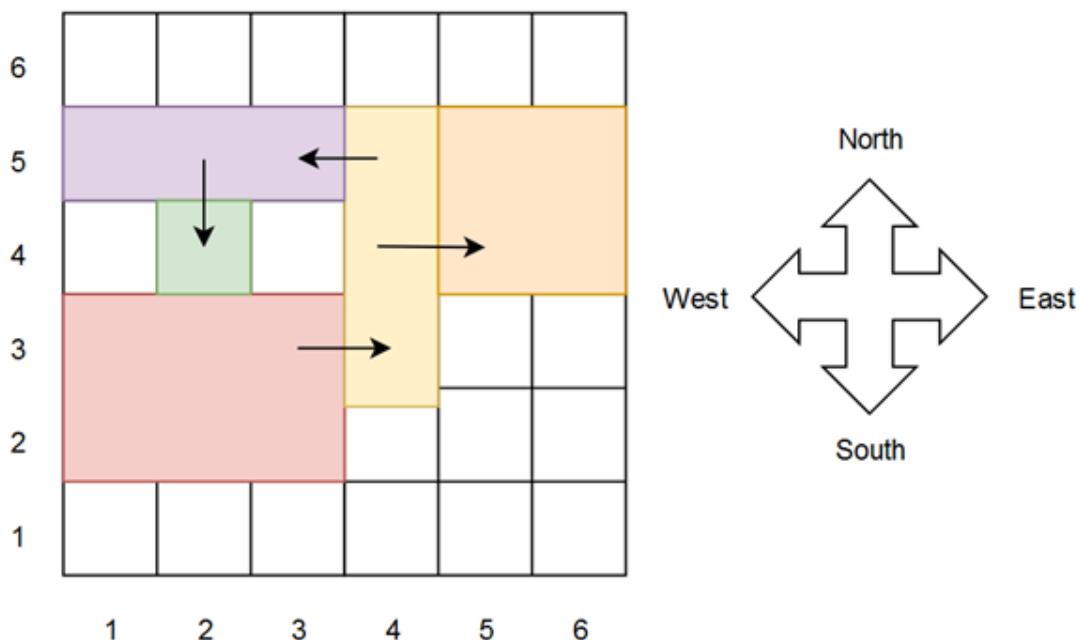
8.5 Algorytm generowania poziomów

8.5.1 Zalożenia

Z uwagi na ograniczenia czasowe przyjęliśmy że generowanie poziomów będzie polegało na losowym wybraniu zbudowanych przez nas pomieszczeń (nazywanych też pokojami) i ułożeniu ich w przestrzeni gry. Ułożenie to będzie odbywać się w sposób losowy, czyli każdy pokój będzie miał losową rotację i losowe położenie.

Dla uproszczenia każdy pokój będzie reprezentowany przez prostokąt o pewnych wymiarach „x” i „y”. Takie podejście pozwala na traktowanie przestrzeni jako dwuwymiarową macierz.

Przyjmujemy, że x i y to liczby naturalne. Wtedy każda komórka macierzy będzie reprezentować kwadrat o boku 1, a każde pomieszczenie będzie można podzielić na kwadraty, jak przedstawiono na rys. 8.5.1-1

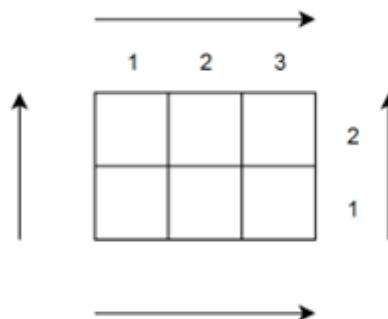


rys. 8.5.1-1 Schemat przestrzeni wypełnionej pokojami

Na rys. 8.5.1-1 przestrzeń przedstawiona jest jako macierz 6×6 . Każda komórka macierzy może być pusta lub posiadać pewien kolor. Komórki należące do tego samego pomieszczenia mają ten sam kolor. Strzałki reprezentują miejsca w których można przejść z pomieszczenia do pomieszczenia.

Aby ułatwić określenie wzajemnego położenia pokoi oraz ich rotacji, będą stosowane kierunki geograficzne zgodnie z rys. 1. Rotacja będzie określana przez kierunek zwrotu. Przyjmuje się, że każde pomieszczenie zaczyna ze zwrotem w kierunku północnym.

Każdy pokój będzie posiadał drzwi. Będą one rozłożone wzdłuż krawędzi reprezentującego pokój prostokąta w sposób równomierny.

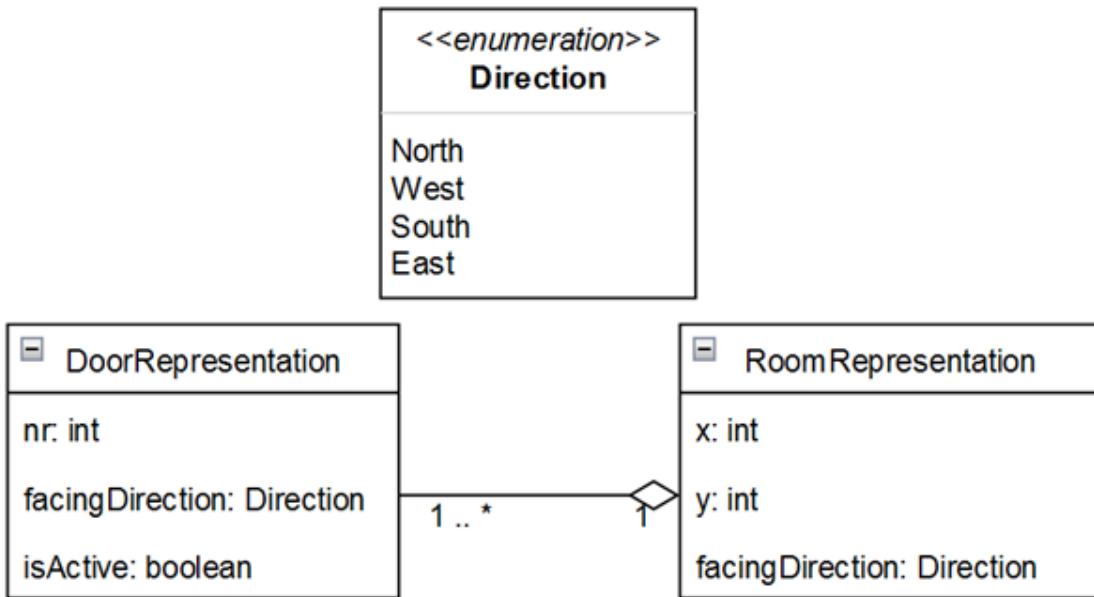


rys. 8.5.1-2 Numerowanie Drzwi w pokoju

Na rys. 8.5.1-2 jest widoczny przykładowy pokój o wymiarach $x = 3$ i $y = 2$. Na każdej krawędzi tego prostokąta są ułożone drzwi. Stosując kierunki geograficzne można powiedzieć, że w kierunku północnym zwrócone są 3 pary drzwi (ponumerowane 1, 2, 3 zgodnie z numerami na rys. 2), a w kierunku wschodnim zwrócone są 2 pary drzwi (ponumerowane 1, 2). Analogicznie wyglądają drzwi zwrócone w kierunku zachodnim i południowym. Kierunek numeracji jest ważny (zgodnie ze strzałkami na rysunku), aby w łatwy sposób określać na podstawie numeru i kierunku zwrotu, gdzie są dane drzwi w przestrzeni.

Każde drzwi mogą być aktywne lub nieaktywne. Aktywne są wtedy, gdy mogą łączyć 2 pokoje ze sobą. Nieaktywne są gdy nie mogą zostać użyte w generatorze. Decyzję, czy drzwi są aktywne, podejmuje osoba modelująca wnętrze danego pokoju.

Na podstawie powyższych założeń został stworzony podstawowy model reprezentacji danych stosowany w algorytmie (rys. 8.5.1-3)



rys. 8.5.1-3 Diagram klas przedstawiający model danych używany w algorytmie generowanie poziomów

8.5.2 Parametry konfiguracyjne

Maksymalna długość ścieżki – maksymalna ilość pokoi pomiędzy pierwszym i ostatnim pokojem

Maksymalna długość ścieżki pobocznej – maksymalna ilość połączonych pokoi, w jednej ścieżce bez rozgałęzień, nie zawierających się w głównej ścieżce.

N – maksymalna ilość prób znalezienia pasujących drzwi wejściowych w nowym pokoju

M – maksymalna ilość prób obrotu nowego pokoju (**M** < 4 – obrót 4 razy 90 stopni oznacza powrót do początkowej pozycji)

P - prawdopodobieństwo tego, że drzwi mogą zostać wybrane do stworzenia nowej ścieżki (**P** – liczba naturalna. Prawdopodobieństwo jest liczone według wzoru: 100% / **P**)

8.5.3 Kroki algorytmu

W poniższym algorytmie „spróbuj postawić pokój” oznacza następujące kroki:

1. Dokonaj walidacji czy pokój może być postawiony, jeśli może zwrócić informację o sukcesie.
2. Jeśli walidacja zakończyła się porażką wybierz inne drzwi z pokoju .
3. Przesuń pokój tak, aby wybrane drzwi znajdowały się naprzeciwko drzwi z poprzedniego pokoju.
4. Powtórz krok 1. W przypadku porażki powtarzaj krok 3 co najwyżej **N** razy.
5. Jeśli pokoju nie udało się postawić, obróć pokój o 90 stopni i powtórz kroki 1 – 4.

6. Jeśli pokoju nie udało się postawić powtórz 5 co najwyżej **M** razy.
7. Zwróć informację o porażce

Algorytm będzie składał się z dwóch faz.

Faza 1 – generowanie głównej ścieżki

Kroki Fazy 1:

0. Postaw pokój początkowy, wybierz losowe drzwi. obecny punkt = punkt za wybranymi drzwiami
1. Jeśli długość ścieżki = maksymalna długość ścieżki wykonaj 2, w przeciwnym razie przejdź do 3
2. Spróbuj postawić pokój końcowy w obecnym punkcie. Jeśli się to udało zwróć aktualną ścieżkę. W przeciwnym razie zwróć informację o porażce.
3. Weź losowy pokój z losową rotacją
4. Spróbuj postawić pokój w obecnym punkcie
5. Jeśli postawienie pokoju nie jest możliwe wykonaj 2.
6. Wybierz losowe drzwi z postawionego pokoju, obecny punkt = punkt za wybranymi drzwiami, długość ścieżki $\leftarrow 1$
7. Wykonaj kroki 1 – 9 dla losowego pokoju
8. Jeśli w obecnym punkcie nie udało się postawić kolejnego pokoju powtórz 6 i 7 co najwyżej **K** razy.
9. Jeśli w obecnym punkcie udało się postawić kolejny pokój zwróć zbudowaną ścieżkę.
10. Zwróć informację o porażce.

Faza 2 – generowanie ścieżek pobocznych

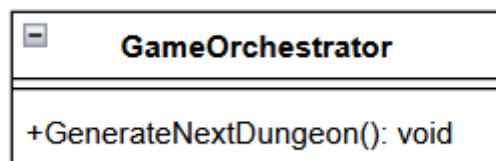
Kroki Fazy 2:

0. aktualna długość ścieżki pobocznej = 0
1. Dla każdego pokoju postawionego w fazie 1, poza pierwszym i ostatnim, wykonuj 2 – 5
2. Dla każdych nieużytych drzwi w pokoju wykonuj 3 – 5
3. Jeśli maksymalna długość ścieżki pobocznej = aktualna długość ścieżki pobocznej zakończ
4. Z prawdopodobieństwem **P** spróbuj postawić pokój za aktualnymi drzwiami
5. Jeśli udało się postawić pokój: aktualna długość ścieżki pobocznej $\leftarrow 1$. Dla tego pokoju wykonaj kroki 2 – 5

8.6 Obsługa działania poziomów

8.6.1 Funkcjonowanie poziomów

Za generowanie kolejnych poziomów odpowiada klasa GameOrchestrator.



rys 8.6.1-1 Klasa odpowiadająca za generowanie poziomów

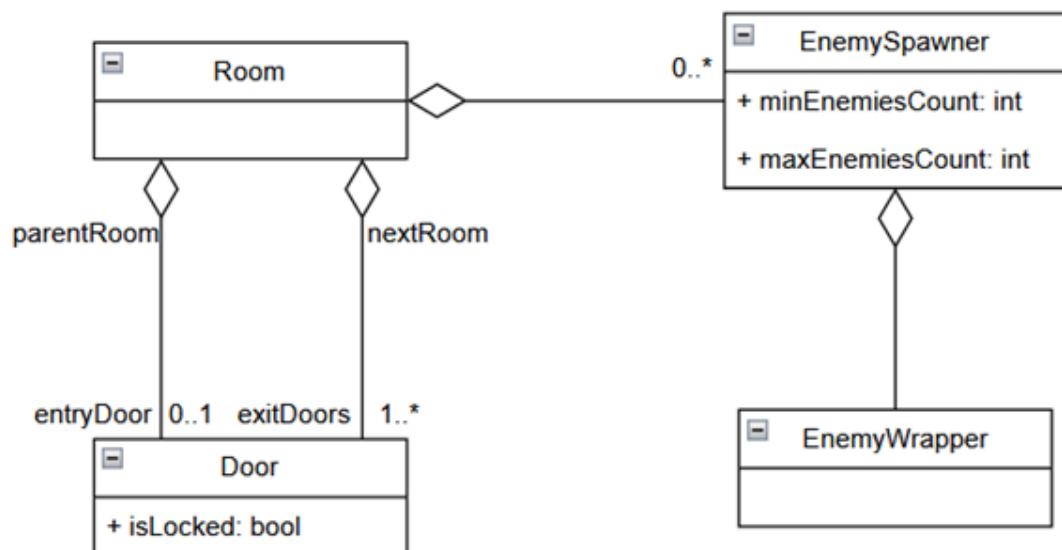
Hierarchię obiektów w pokoju opisuje rys. 8.6.1-2

Room – klasa reprezentująca pomieszczenie w ramach poziomu

Door – klasa reprezentująca drzwi, którymi można wejść / wyjść z pokoju

EnemySpawner – klasa, której zadanie to przyzwanie przeciwników i informowanie pokoju o ich śmierci

EnemyWrapper – klasa, będąca mostem pomiędzy przyzwanyim przeciwnikiem a EnemySpawnerem



rys 8.6.1-2 Diagram klas opisujący hierarchię obiektów w pokoju

W momencie generowania poziomu, po ustawieniu pokojów w odpowiednich miejscach i w odpowiedniej rotacji następuje proces konfiguracji każdego z pomieszczeń.

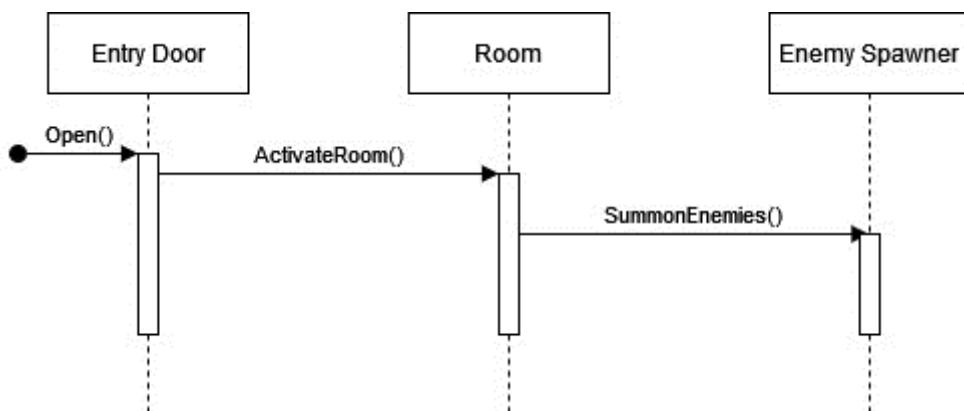
1. Znalezienie wszystkich EnemySpawnerów wśród dzieci pokoju
2. Wylosowanie przeciwników, których EnemySpawnerów przyzwie po aktywacji pokoju
3. Usunięcie nieużywanych drzwi i drzwi wejściowych
4. Zablokowanie pokoju – wyłączenie wszystkich przeciwników i zablokowanie drzwi wejściowych

W przypadku pokojów pierwszego i ostatniego konfiguracja obejmuje tylko krok 3.

Po zakończeniu konfiguracji następuje zbudowanie NavMesh'u - służącego do poprawnego funkcjonowania przeciwników.

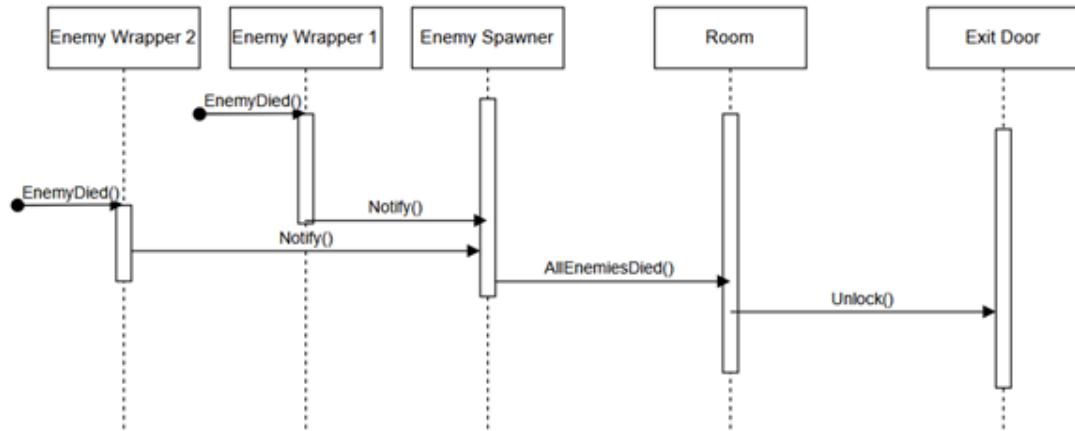
- Pokoje z przeciwnikami:

W trakcie trwania gry, gdy gracz otworzy drzwi wejściowe następuje aktywacja pokoju. Diagram sekwencji aktywacji pokoju jest przedstawiony na rys. 8.6.1-3.



rys. 8.6.1-3. Diagram sekwencji przedstawiający proces aktywacji pokoju

Po aktywacji pokoju gracz pokonuje przyzwanych przeciwników. O śmierci przeciwnika informowany jest przypisany EnemyWrapper. Gdy wszyscy przeciwnicy zginą, EnemySpawner informuje o tym pokój, który odblokowuje drzwi. Jest to przedstawione na diagramie sekwencji na rys 8.6.1-4.

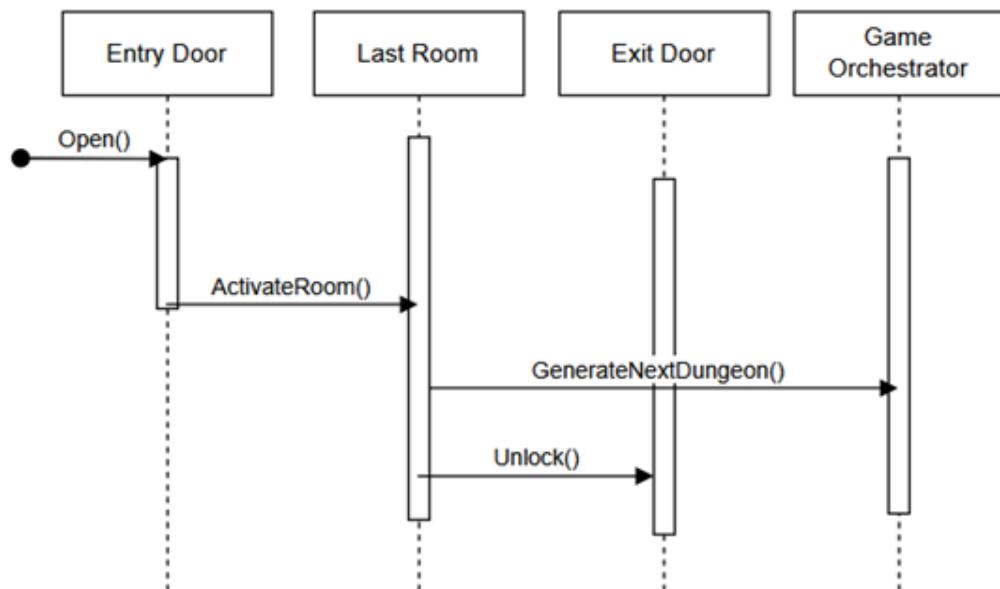


rys 8.6.1-4 Diagram sekwencji przedstawiający proces odblokowywania drzwi wyjściowych w pokoju

- Pierwszy i ostatni pokój

Pierwszy pokój jest aktywny cały czas i nie ma drzwi wejściowych.

W przypadku ostatniego pokoju aktywacja pokoju wygląda następująco:



rys 8.6.1-5 Diagram sekwencji przedstawiający proces aktywacji pokoju

8.6.2 Integracja z graczem

8.6.2.1 - Collidery

Aby gracz mógł poprawnie poruszać się po pomieszczeniach należało ustawić na wszystkich obiektach gry (powierzchniach i przeszkodach) komponenty Collider.

W przypadku podłóg ważne jest, żeby grubość collidera wynosiła przynajmniej 0.5 jednostki unity. Zapobiega to przypadkowemu przenikaniu przez podłogi podczas specyficznych kombinacji akcji gracza zawierających szybkie kucanie i skakanie w niskiej przestrzeni.

Każda przeszkoda i ściana zawiera collider w kształcie prostopadłościanu (ang. Box Collider) dopasowany możliwie najbliżej do kształtu obiektu. Zrezygnowaliśmy z używania komponentów „Mesh Collider” (Generujących Collider na podstawie wyświetlanej tekstury), ponieważ powodowały problemy z poruszaniem, a w niektórych przypadkach collider znajdował się poza obiektem.

8.6.2.2 Drzwi

Aby otworzyć drzwi muszą być spełnione następujące warunki

- Gracz musi mieć zwróconą kamerę na drzwi. W każdej klatce gry, za pomocą wbudowanej w Unity metody OnMouseEnter gra sprawdza czy promień wychodzący ze środka kamery przecina model drzwi.

- Gracz musi znajdować się przynajmniej „,5” jednostek od drzwi. W tym warunku następuje obliczenie jaką jest odległość pomiędzy graczem a środkiem modelu drzwi. Następuje to tylko jeśli warunek 1 jest spełniony.

- Drzwi muszą być odblokowane i zamknięte.

- Gracz musi nacisnąć przycisk interakcji z przedmiotami.

Jeśli wszystkie warunki są spełnione drzwi zostaną otworzone i następuje aktywacja kolejnego pomieszczenia.

8.6.3 Integracja z przeciwnikami

AI przeciwników do poprawnego poruszania potrzebuje komponentów NavMeshSurface i NavMeshObstacle nałożonych odpowiednio na podłogi i przeszkody.

Z uwagi na różne rozmiary przeciwników podłogi muszą mieć nałożone 2 komponenty NavMeshSurface, jeden skonfigurowany pod przeciwników normalnych

rozmiarów, a drugi skonfigurowany pod dużych przeciwników. Pozostała część integracji odbywa się za pomocą opisanej wcześniej klasy “Enemy Wrapper”.

8.6.4 Quality of Life

Aby ułatwić graczowi poruszanie się po poziomie, w każdym pokoju nieotwarte przez niego drzwi mają pomarańczową poświata (rys. 8.6.4-1). Po otwarciu drzwi poświata znika



rys. 8.6.4-1 Pomarańczowa poświata drzwi, które jeszcze nie zostały otwarte

8.7 Optymalizacja

8.7.1 Memoizacja

W trakcie procesu generowania i funkcjonowania poziomu obiekty gry wywołują na innych obiektach metodę „GetComponents” dostarczaną przez interfejs Unity. Metoda ta jest dosyć kosztowna i w przypadku zbyt wielu wywołań może powodować spowolnienie aplikacji. Zdarza się, że na pewnych gameobiektach metoda GetComponents wykonywana jest wielokrotnie przez różne komponenty.

Aby nie wywoływać tej metody z tymi samymi parametrami została zastosowana technika Memoizacji. Polega ona na zapamiętywaniu par argumentów i wyników

funkcji. W przypadku języku programowania C# zapamiętywanie odbywa się poprzez słownik do którego dodawane są nowe pary argument, wynik, jeśli argument nie istnieje już w słowniku. Implementacja polega na użyciu fabryki funkcji. Fabryka jako argument przyjmuje funkcje która będzie używana do tworzenia nowych wyników dla nowych argumentów.

```
public static class Memoized
{
    [9 usages]
    public static Func<T1, TRet> Of<T1, TRet>(Func<T1, TRet> f)
    {
        var cache = new ConcurrentDictionary<T1, TRet>();
        return arg1:T1 => cache.GetOrAdd(arg1, valueFactory:xarg:T1 =>f(xarg));
    }
}
```

rys 8.7.1-1 Fabryka funkcji stosując technikę memoizacji

Na rys. 8.7.1-1 widać przykład deklaracji metody używając fabryki Memoized.Of. Posiadając nieoptymalną metodę „FindBorderF”, która dla obiektu gry zwraca komponent o nazwie „Border”, możemy stworzyć optymalną metodę „FindBorder”, która będzie zapamiętywać wyniki dla argumentów funkcji. Użycie optymalnej metody wygląda tak samo jak użycie zwykłej metody w języku C#.

```
private Border GetBorder()
{
    return FindBorder(gameObject);
}

private static readonly Func<GameObject, Border> FindBorder =
    Memoized.Of<GameObject, Border>(FindBorderF);

[1 usage]
private static Border FindBorderF(GameObject go)
{
```

Rysunek 8.7.1-2 Przykład użycia memoizacji

8.7.1 Przetwarzanie równolegle

Generowanie nowego poziomu odbywa się po otwarciu drzwi wejściowych ostatniego poziomu. Nie może się ono odbywać na głównym wątku, ponieważ wtedy gracz musiałby czekać aż generowanie się zakończy, zanim będzie mógł się ruszyć. W ekosystemie Unity przetwarzanie równolegle odbywa się za pomocą systemu tzw. „korutyn”. Korutyna pozwala na przetwarzanie operacji na innych wątkach niż głównym, stosując przy tym optymalizację, że jedna operacja (odpowiednio zdefiniowana) jest wykonywana w jednej klatce gry. Wtedy np. jeśli chcemy na kolekcji

3-elementowej wykonać pewną operację to zamiast 3 operacji wykonanych w jednej klatce otrzymamy 1 operację w każdej z 3 kolejnych klatek.

W przypadku generowania poziomu wiele operacji jest wykonywanych wielokrotnie:

- stawianie pokoju
- przyzwanie przeciwnika
- zablokowanie drzwi
- usunięcie starych przedmiotów po przejściu do nowego pokoju

Wszystkie te operacje mogą odbywać się w ramach korutyn przetwarzając jeden obiekt w trakcie jednej klatki

8.8 System dropów

8.8.1 Wprowadzenie

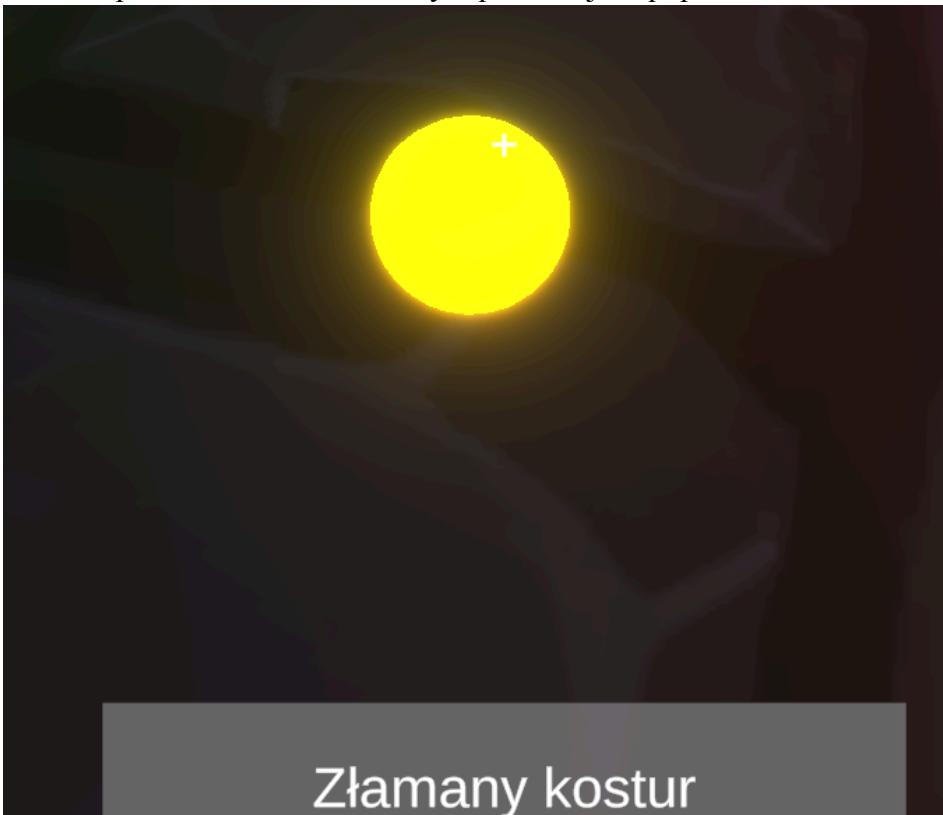
Każdy przeciwnik w momencie śmierci pozostawia po sobie pewne Equipables możliwe do podniesienia przez gracza. Aby jednak było to możliwe, muszą mieć one fizyczną reprezentację na scenie - odpowiada za to klasa *EquipableOnScene*.

EquipableOnScene to obiekt reprezentujący *SpellItem* lub *Item* jako byt na scenie w postaci kolorowej kulki. Pozwala to na uniknięcie modelowania 3D modeli wielu przedmiotów.

EquipableOnScene pozostawiane są po martwych wrogach.

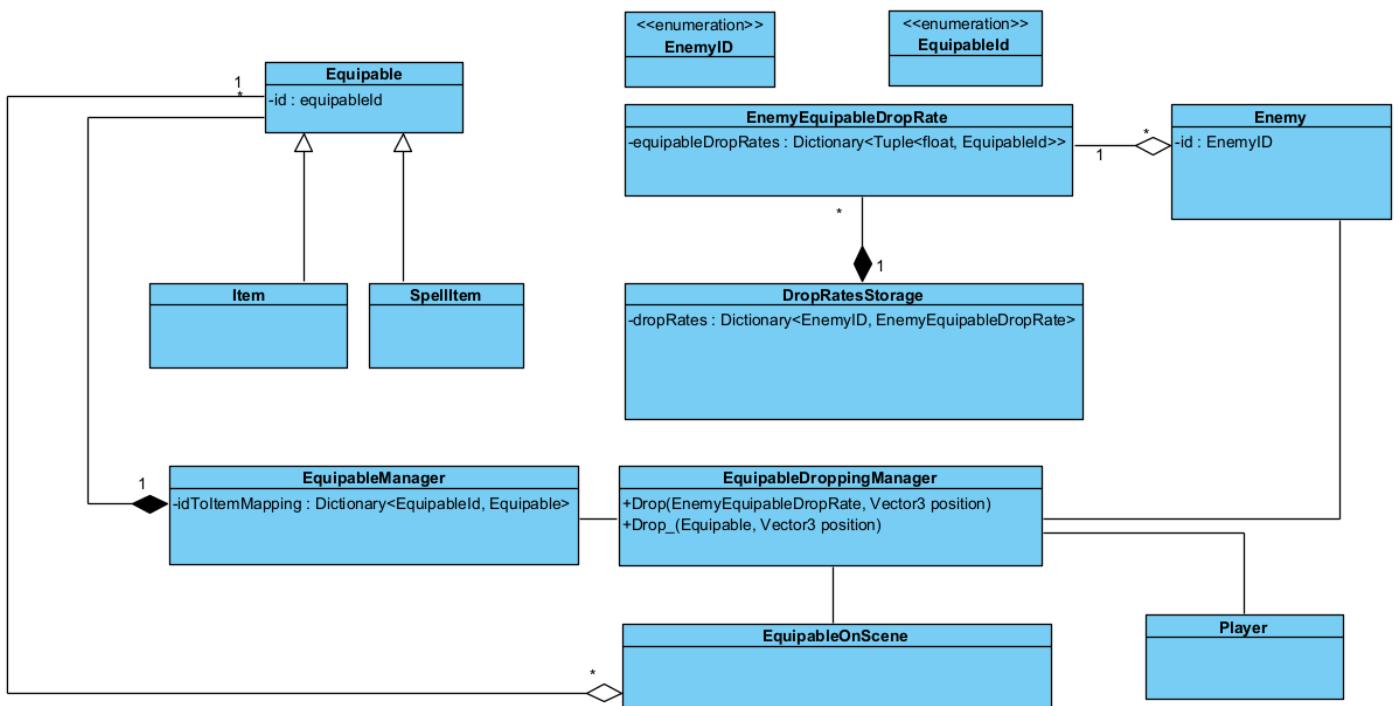


Gracz po podejściu do kulki i najechaniu na nią celownikiem otrzyma informacje o nazwie SpellItemu lub Itemu który reprezentuje EquipableOnScene:



O ile ma miejsce w odpowiednim ekwipunku, może go podnieść za pomocą klawisza R.

8.8.2 Działanie



System dropów - diagram klas

DropRatesStorage przechowuje informacje o EnemyEquipableDropRate, które jest słownikiem par $\langle\text{float}, \text{EquipableId}\rangle$ gdzie float to liczba $[0;1]$ definiująca prawdopodobieństwo otrzymania z przeciwnika określonego przez EquipableId przedmiotu.

W momencie śmierci przeciwnika, EquipableDroppingManager pobiera z Enemy EnemyEquipableDropRate i losuje przedmioty, które ten powinien pozostawić po sobie. Następnie tworzy odpowiednią liczbę instancji EquipableOnScene i do każdego przypisuje po jednym upuszczonym przedmiocie.

8.9 System Poruszania się i życia

8.9.1 System poruszania się

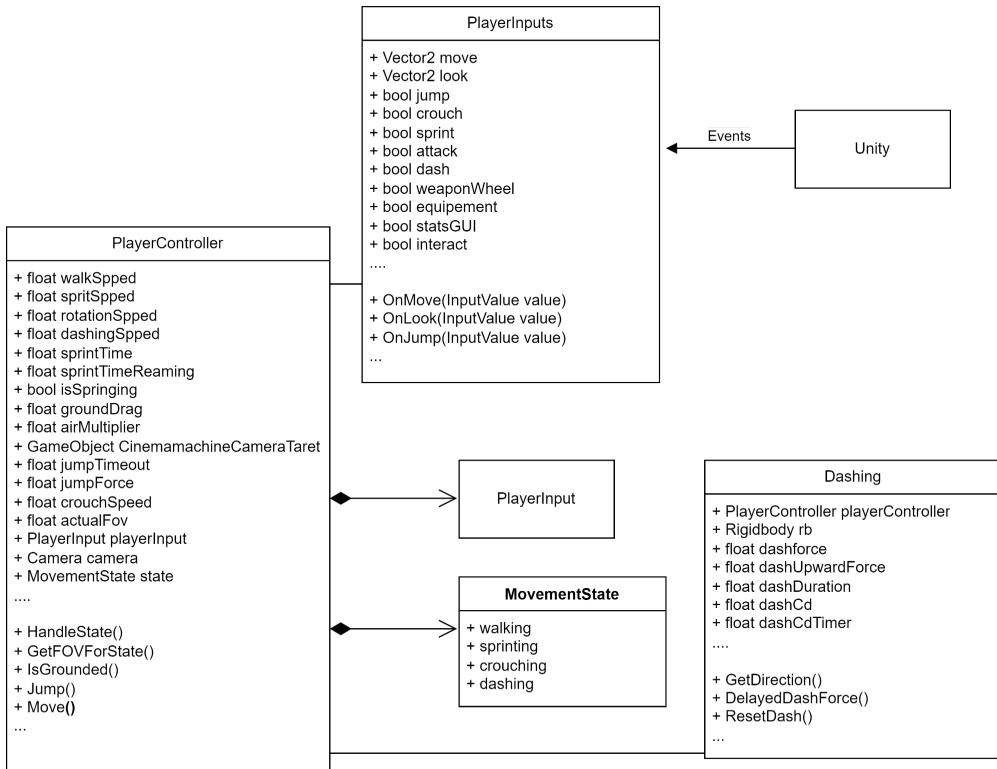
Postanowiono zaimplementować system używając nowych, dobrych praktyk projektowania gier w Unity. W związku z tym użyto **The Input System**.

Ten system jest alternatywnym podejściem do starego i wbudowanego w silnik **The Input Manager**. Nowy system jest również oficjalną produkcją Unity, jednak nie jest domyślnie wgrywany do każdego projektu, ale trzeba go zainportować przez **Package Manager**.

The Input System w Unity przynosi szereg korzyści, w tym:

- **Większa elastyczność:** System nowego wejścia w Unity to zaawansowany framework umożliwiający obsługę wielu rodzajów urządzeń wejściowych, takich jak kontrolery, klawiatury, myszki czy ekrany dotykowe. Jest to znaczące ulepszenie w porównaniu z poprzednim systemem wejść w Unity, oferującym bardziej elastyczną i intuicyjną obsługę wejść użytkownika. Również dzięki niemu programiści mogą łatwo tworzyć gry, które działają na różnych platformach i urządzeniach, bez konieczności tworzenia oddzielnych rozwiązań dla każdego z nich. Nasz projekt w aktualnej fazie obejmował tworzenie gry tylko na platformę PC, jednak dzięki zastosowanej wyżej wymienionej technologii dodanie następnych urządzeń nie będzie problematyczne.
- **Wyższa wydajność:** Nowy system oferuje lepszą wydajność w porównaniu z poprzednimi rozwiązaniami, co oznacza mniej opóźnień w reakcjach na wejścia użytkownika.
- **Elastyczność i personalizacja obsługi wejść.** System nowego wejścia pozwala na definiowanie niestandardowych akcji i przypisywanie ich do różnych urządzeń. To daje większą kontrolę nad interakcją z grą i umożliwia lepsze dopasowanie do preferencji użytkownika. Niestety w związku z krótkim okresem trwania projektu ustaliliśmy, że nie będziemy dawali opcji personalizacji sterowania wejściami przez gracza. Nie zmienia to jednak faktu, że z naszych testów wynika, że byłoby to prostsze do zaimplementowania.

W skład systemu Poruszania się wchodzą skrypty, które odpowiadają za odczytywanie danych z urządzeń wejścia gracza (np. myszka, klawiatura). Następnie dokonywana jest wstępna analiza danych i ich przetworzenie, żeby inne klasy mogłyby z nich korzystać.



8.9.1.1 PlayerInputs

PlayerInputs jest kodem, który odbiera eventy od unity z wartościami jakie dane przyszły oraz robi wstępne przetwarzanie ich.

Najważniejszym skryptem obsługującym poruszanie się postaci jest **PlayerController**. Pozwala na łatwą parametryzację atrybutów odpowiedzialnych za szybkość poruszania się, fov (Field of View), skoku itp.

Movement state opisuje w jakim stanie gracz się teraz znajduje. Dzięki temu łatwo określamy jaki ma mieć FOV oraz prędkość poruszania się itp.

8.9.1.2 Bieganie

Nasza gra jako, że jest dynamiczna musi pozwolić graczowi na “bieганie” w powietrzu. Oznacza to tyle, że gracz nie dotykając ziemi, będąc w stanie skoku może zmienić kierunek swojego lotu. Nie było to jednak prostsze od implementacji bez tego, wręcz przeciwnie. Należało w kilku sytuacjach dopisać dodatkowe warunki, aby gracz nie mógł utknąć przyklejony do ściany lub przypadkiem nie wspinał się na nie. Nasza mechanika skakania obejmuje także ustalenie limitu czasowego między kolejnymi skokami (**jumpTimeout**), aby uniemożliwić wielokrotne skakanie w krótkim czasie lub skakanie w powietrzu przez dłuższy czas po opuszczeniu podłoża.

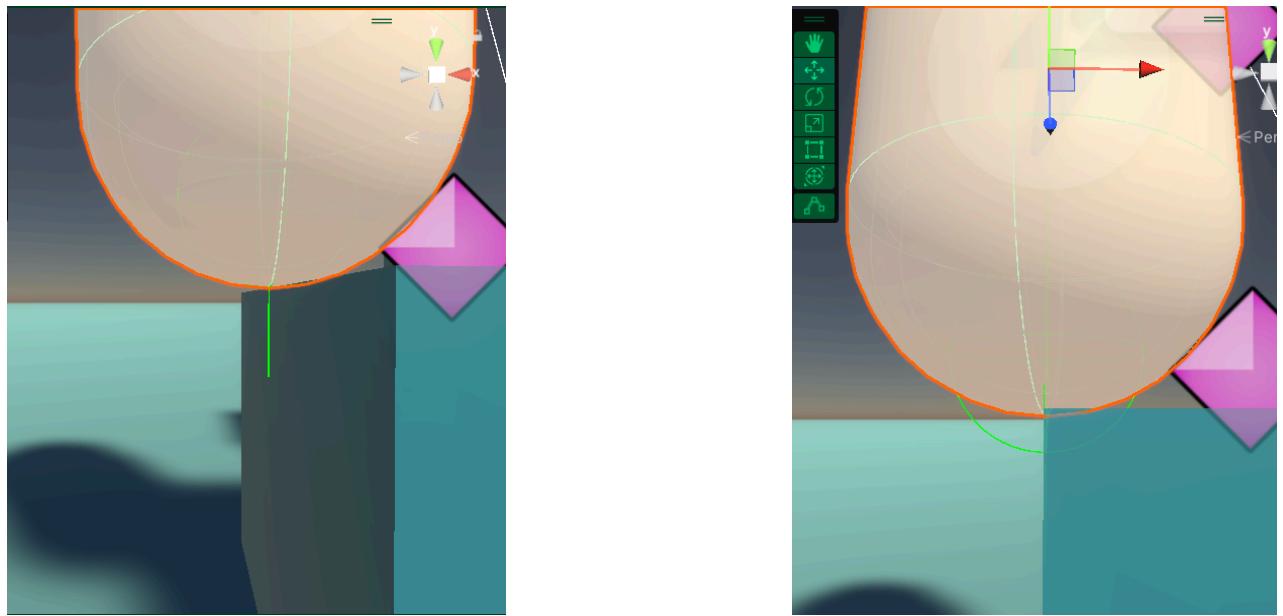
8.9.1.3 Sprawdzanie podłoża

Metoda **IsGrounded()** wykorzystuje **Physics.SphereCast** w celu sprawdzenia, czy sfera w okolicy stóp postaci styka się z obiektami oznaczonymi jako ziemia

(whatIsGround). Promień sfery reprezentującej obszar stóp postaci jest rzutowany w dół, a jeśli w obszarze tym znajduje się obiekt uznawany za ziemię, to zmienna `grounded` zostaje ustawiona na true, w przeciwnym razie na false.

Dzięki temu mechanizmowi postać wie, kiedy jest na ziemi i może wykonać skok oraz inne czynności związane z byciem w kontakcie z podłożem.

W pierszej wersji gracz nie używał SphereCast a jedynie promień, przez to czasem występowały błędy, gdy postać była na krawędzi i promień stwierdzał, że gracz nie doryka ziemi.



Wersja po lewej sprawiała problemy z krawędziami, podczas gry wersja po prawej dużo lepiej sobie z nimi radzi.

8.9.1.4 Kucanie

Funkcja `Crouch()` zarządza stanem kucania postaci.

Jeśli zostanie wykryte wcisnięcie klawisza kucania lub postać nie może wstać z powodu kolizji z otoczeniem (`!canStandUp`), zmienia się skala postaci w osi Y za pomocą `transform.localScale`.

Zmiana skali wpływa na wizualne schylenie postaci, co symuluje efekt kucania.

`canStandUp` jest sprawdzane poprzez promień (**Raycast**), który jest rzucany w górę od pozycji postaci. Jeśli wykryje on kolizję z obiektami zdefiniowanymi jako przeszkody `crouchingLayers`, którymi są na przykład ściany, podłogi czy sufit, to postać nie może wstać.

8.9.1.4 Dashing

Innym ważnym elementem systemu poruszania się jest mechanika znana z innych gier tzw **Dash**.

Dashowanie w grach komputerowych odnosi się do szybkiego, krótkotrwałego przemieszczenia postaci w określonym kierunku. Może być to szybki sprint, teleporcja albo inna forma szybkiego przemieszczenia.

Zazwyczaj dashowanie może być używane w różnych kontekstach w grach, od unikania ataków, przez szybkie przemieszczanie się przez obszary, aż po akcje ofensywne, takie jak szybkie ataki czy zaskakujące manewry.

Mechanika dashowania jest bardzo znana i lubiana w grach jak np. Doom, ponadto pasowała do typów naszej gry i wprowadziła przyjemniejszy system poruszania się.

8.9.1.5 Usunięte mechaniki

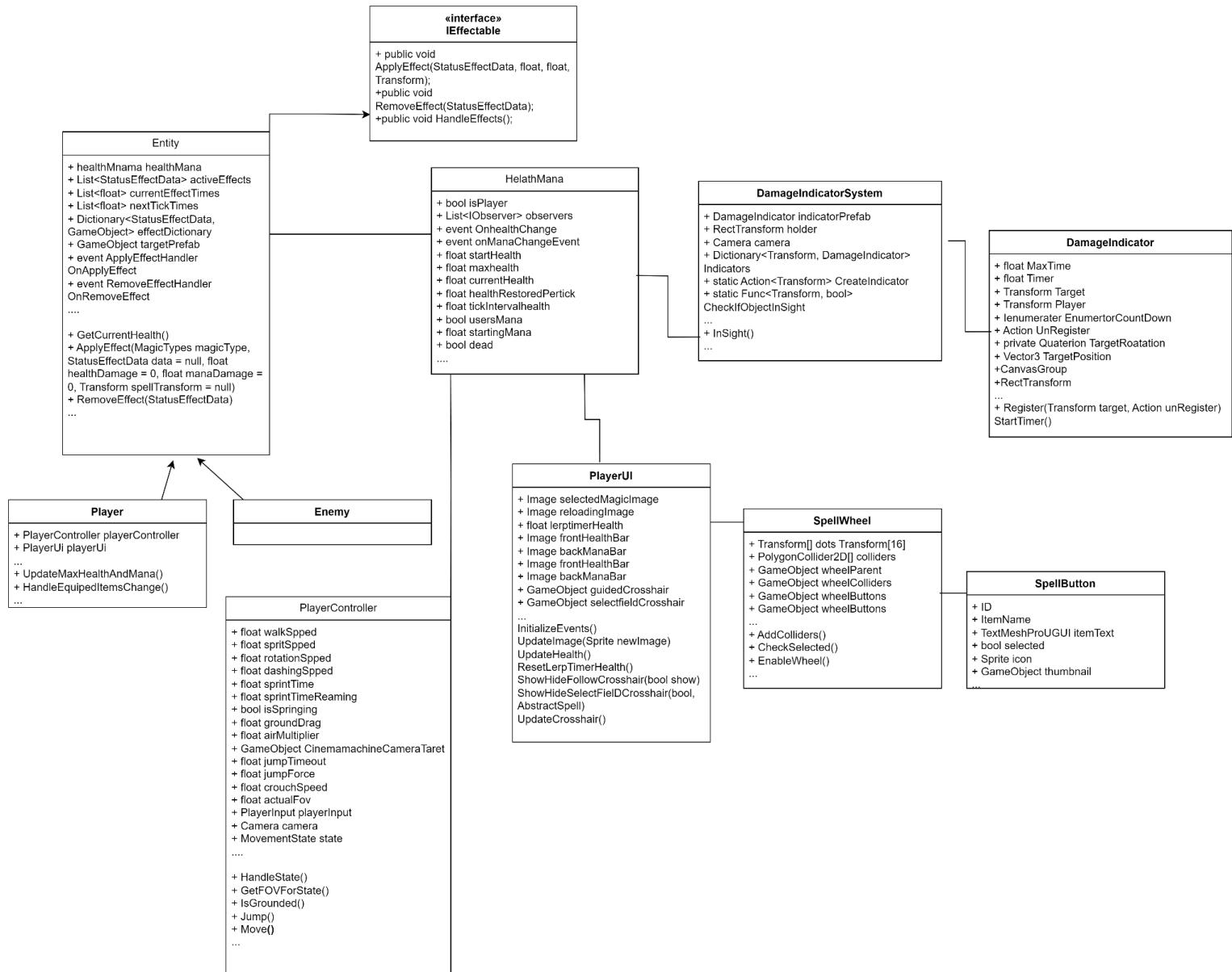
W trakcie rozwoju projektu, początkowo zrobiliśmy implementację dwóch zaawansowanych mechanik ruchu - wallrunu oraz slidingu (ślizgania się). Wallrun polegał na umożliwieniu postaci biegu po ścianach, podczas gdy sliding zapewniałby płynne przemieszczanie się po powierzchniach równych lub pochyłych z chwilową zwiększoną prędkością oraz zmienionym poziomem widoczności.

Jednakże po serii testów oraz analizie naszych map i głębszej refleksji nad konwencją naszej średniowiecznej fabuły, doszliśmy do wniosku, że te mechaniki mogłyby wprowadzić dodatkowe trudności i nie pasować do ogólnej estetyki oraz przeznaczenia naszej gry.

Wallrun, choć fascynujący pod względem dynamizmu rozgrywki, wymagałby znaczących modyfikacji map, takich jak odpowiednie geometryczne ukształtowanie ścian, aby umożliwić płynne poruszanie się postaci. Podobnie sliding, czyli ślizganie się po ziemi i równiach pochyłych, również wymagałby dopasowania terenu, co kolidowało z naszym pierwotnym projektem map.

W związku z tym, w duchu zachowania spójności fabularnej i ograniczeń projektowych, zdecydowaliśmy się usunąć te zaawansowane mechaniki z naszego projektu. Koncentrując się na istniejących aspektach rozgrywki, mogliśmy lepiej skupić się na opracowaniu głównych elementów gry, zapewniając graczom satysfakcjonujące i spójne doświadczenie.

8.9.2 System Życia



Klasa "HealthMana" odpowiada za zarządzanie zdrowiem i maną postaci w grze. Jest to istotny element, który wpływa na interakcję postaci z otoczeniem i umożliwia kontrolę nad ich stanem zdrowia oraz ilością many. Ta klasa jest kluczowa dla mechaniki zdrowia i many postaci w grze, umożliwiając precyzyjną kontrolę nad ich stanem oraz reakcje na zmiany w trakcie rozgrywki.

Klasa "PlayerUI" w naszym projekcie Unity pełni istotną rolę w interfejsie użytkownika oraz wizualizacji stanu zdrowia i many postaci. Posiada referencje do różnych elementów UI, takich jak paski życia (frontHealthBar, backHealthBar) i many (frontManaBar, backManaBar), które są aktualizowane w czasie rzeczywistym w zależności od stanu zdrowia i many postaci.

Wykorzystuje zmienne chipSpeedHealth i chipSpeedMana do płynnego aktualizowania pasków zdrowia i many.

Funkcje **ShowHideFollowCrosshair()** oraz **ShowHideSelectFieldCrosshair()** odpowiedzialne są za pokazywanie i ukrywanie elementów interfejsu, takich jak celownik czy pole wyboru ataku, w zależności od wykonywanej akcji przez postać.

Klasy **DamageIndicator** i **DamageIndicatorSystem** są kluczowymi elementami systemu wskaźników obrażeń w grze.



Pokazany damage indicator na obrazku ze strony z której przybył do do nas atak. Wskazuje poprawą pozycję nawet jak ten przeciwnik umrał.

DamageIndicator posiada elementy interfejsu użytkownika, takie jak **CanvasGroup** i **RectTransform**, które są odpowiedzialne za wyświetlanie wskaźnika obrażeń. Wykorzystuje timer do wygaszania wskaźnika po określonym czasie (**MaxTime**). Metoda **Register()** rejestruje nowy wskaźnik obrażeń dla określonego celu - wrogiej postaci, która nas zaatakowała, rozpoczynając licznik i rotację wskaźnika w kierunku celu. Metoda **RotateToTheTarget()** zapewnia ciągłą rotację wskaźnika w kierunku celu.

DamageIndicatorSystem posiada metodę **Create()** tworzy nowy wskaźnik obrażeń dla określonego celu, lub restartuje istniejący, jeśli cel już ma wskaźnik. Wykorzystuje słownik **Indicators**, by przechowywać i zarządzać wskaźnikami dla poszczególnych celów. **InSight()** ocenia, czy cel znajduje się w polu widzenia kamery, wykorzystując przekształcenia ekranu w punkty widoku.

Te klasy wraz z powiązanymi metodami tworzą system wskaźników obrażeń, który umożliwia śledzenie i wyświetlanie informacji o obrażeniach dla różnych celów w grze, zapewniając intuicyjny interfejs dla gracza.



Klasa **SpellWheel** w grze odpowiada za interakcję z kołem zaklęć oraz wybieranie odpowiedniego zaklęcia w zależności od pozycji kurSORA.

Klasa **SpellWheelButton** Odpowiada za interakcje z przyciskiem koła zaklęć, takie jak wyświetlanie nazwy i ikony danego zaklęcia. Te elementy łącznie tworzą interaktywny system wyboru zaklęć w grze.

8.10 HDRP oraz Visual effect Graph

Rozpoczynając pracę nad projektem, przeprowadziłem gruntowne rozeznanie w celu wyboru odpowiedniej technologii, a ostatecznie zdecydowałem się na wykorzystanie **Unity High Definition Render Pipeline (HDRP)** zamiast standardowego **Buildin Rendering (BR)**. Ta decyzja opierała się na starannym zbadaniu zalet i wad obu rozwiązań, co umożliwiło dokładne zrozumienie ich potencjału oraz dopasowanie do potrzeb projektu. Poniżej przedstawiam analizę wyboru HDRP w kontekście jego korzyści oraz ograniczeń.

8.10.1 Zalety HDRP:

- **Wizualna jakość:** HDRP oferuje zaawansowane efekty wizualne, w tym realistyczne oświetlenie, cienie i refleksy, efekty środowiskowe (takie jak mgła, dym czy mgła wolumetryczna), co pozwala na tworzenie bardziej imponujących i realistycznych scen.
- **Wsparcie dla najnowszych technologii:** HDRP umożliwia korzystanie z najnowszych technologii graficznych, takich jak ray tracing, co przekłada się na bardziej realistyczne odwzorowanie światła i cieni.
- **Elastyczność i personalizacja:** Ta technologia zapewnia dużą elastyczność w dostosowaniu renderingu do indywidualnych potrzeb projektu. Dzięki niemu

można tworzyć niestandardowe efekty i wygląd scen.

- **Zgodność z nowoczesnymi platformami:** HDRP jest zoptymalizowany pod kątem działania na różnych platformach, co sprawia, że projekt będzie bardziej uniwersalny i łatwiejszy w przenoszeniu na różne urządzenia.

8.10.2 Wady HDRP:

Wymagania sprzętowe: Korzystanie z HDRP może być wymagające pod względem zasobów sprzętowych. Starsze lub mniej wydajne urządzenia mogą mieć trudności z płynnym działaniem projektu.

Uczenie się i przystosowanie: Przejście na HDRP może wymagać czasu na naukę nowych funkcji i narzędzi, co może spowodować początkowe trudności i opóźnienia w projekcie.

Możliwe problemy z wydajnością: Nieprawidłowe skonfigurowanie ustawień HDRP może prowadzić do spadku wydajności lub wystąpienia błędów renderingu, co wymagać będzie dodatkowego czasu na optymalizację.

Kompatybilność ze starszymi urządzeniami: Built In Rendering może być bardziej kompatybilny ze starszymi lub mniej wydajnymi urządzeniami, co może być istotne, jeśli projekt ma działać na różnorodnych platformach.

Wsparcie społeczności i dokumentacji: Z uwagi na dłuższy czas istnienia, Buildin Rendering może mieć bogatszą bazę dokumentacji i wsparcia społecznościowego, co może ułatwić rozwiązywanie problemów i zdobywanie wiedzy.

Rozmiar plików i wydajność w czasie rzeczywistym: HDRP może generować większe pliki i wymagać więcej zasobów w porównaniu do Buildin Rendering, co może mieć wpływ na wydajność aplikacji w czasie rzeczywistym.

8.10.3 Alternatywa HDRP - URP

Podczas wyboru pomiędzy BR a HDRP rozważano również pośrednie rozwiązanie. **Unity Universal Render Pipeline (URP)** to kolejne narzędzie renderowania w Unity, które oferuje alternatywę dla zarówno Buildin Rendering jak i HDRP.

Charakterystyka URP:

Wyważony stosunek między wydajnością a jakością: URP oferuje lepszą wydajność w porównaniu do HDRP, zachowując jednocześnie poprawną jakość wizualną. Jest to szczególnie korzystne dla projektów, które nie wymagają najwyższej jakości grafiki, ale wymagają płynnej rozgrywki na różnych urządzeniach.

Lepsza kompatybilność z różnymi platformami: URP może być bardziej uniwersalny i łatwiejszy w przenoszeniu między różnymi platformami, co sprawia, że jest to atrakcyjna opcja dla projektów, które docelowo mają działać na wielu urządzeniach.

Optymalizacje dla mobilnych i starszych urządzeń: Jest bardziej zoptymalizowany pod kątem urządzeń mobilnych oraz starszych komputerów, co sprawia, że może być lepszym wyborem dla projektów skierowanych na te platformy.

8.10.4 Decyzja i wybór Render Pipeline'a

W zespole przetestowaliśmy zarówno BR, URP oraz HDRP. Podczas testów nie wykryliśmy sporej różnicy w wydajności i płynności pomiędzy naszymi ustawieniami URP oraz HDRP.

BR niestety zostało odrzucone ze względu na ogromną różnicę w wyglądzie. Generowany obraz w BR był znacznie gorszej jakości pod względem światel, które miały również pełnić u nas ważną rolę.

Wybierając pomiędzy URP a HDRP zdecydowaliśmy się wybrać HDRP.

Dlaczego wybraliśmy HDRP:

- **Brak potrzeb wsparcia innych platform:** Aktualnie nie mamy w planach wydania aplikacji na platformę inną niż komputery z windowsem. Oznacza to, że nie będziemy się martwili co do wydajności aplikacji na konsolach czy telefonach, zyskując jednocześnie duży zapas możliwości i efektów specjalnych.
- **Wizualna jakość i realizm:** Nasz projekt skupia się na potrzebie uzyskania najwyższej jakości wizualnej. HDRP oferuje zaawansowane efekty, takie jak realistyczne oświetlenie, cienie czy refleksy, co pozwala na stworzenie imponujących i bardzo realistycznych scen. Dzięki temu duża ilość światel sprawi, że gra będzie bardzo nasycona pobudzająca dla gracza.
- **Potrzeba zaawansowanych efektów graficznych:** W projekcie istniała potrzeba eksperymentowania z zaawansowanymi efektami wizualnymi podczas tworzenia magii, a HDRP dostarcza narzędzi do ich tworzenia. To kluczowe, biorąc pod uwagę cel artystyczny projektu.
- **Zgodność z nowoczesnymi technologiami:** HDRP oferuje wsparcie dla najnowszych technologii graficznych, takich jak ray tracing, co przekłada się na bardziej realistyczne odwzorowanie światła i cieni. Zespół chciał przetestować czy uda się użyć tych technologii, jednak ze względu na niewielką ilość czasu testy tych technologii zostały przeniesione na później.
- **Wizja końcowego efektu:** Decyzja o wyborze HDRP wynikała z wizji końcowego efektu - chcemy stworzyć projekt, który nie tylko będzie funkcjonalny, ale również wizualnie imponujący, co jest zgodne z założeniami i priorytetami naszego projektu.

8.11 Particle System vs Visual Effect Graph

Rozpoczęcie procesu tworzenia projektu wymaga odpowiedniego rozeznania się w dostępnych narzędziach i technologiach celem wyboru optymalnego rozwiązania. W kontekście mojego projektu, podjąłem decyzję o wykorzystaniu Unity Visual Effect Graph zamiast tradycyjnego Particle System, co stało się przedmiotem pogłębionych rozważań oraz analiz.

Zalety Unity Visual Effect Graph:

- **Złożone Efekty Wizualne:**
Visual Effect Graph umożliwia bardziej zaawansowane i złożone efekty wizualne. Dzięki temu narzędziu można tworzyć bardziej skomplikowane animacje i efekty, co jest trudniejsze lub niemożliwe do osiągnięcia za pomocą Particle System.
- **Wykorzystanie GPU zamiast CPU:**
GPU jest specjalnie zaprojektowane do wykonywania równoległych obliczeń związanymi z grafiką. Dzięki temu Visual Effect Graph może działać znacznie szybciej niż tradycyjne rozwiązania wykorzystujące CPU, szczególnie przy bardziej złożonych efektach wizualnych. Oznacza to, że generowanie bardzo dużej ilości cząsteczek nie stanowi dużego problemu.
- **Modularność i Wielość Warstw:**
VFX Graph pozwala na tworzenie efektów składających się z wielu warstw i modułów, co daje większą elastyczność w projektowaniu efektów wizualnych. To znacząco zwiększa kontrolę nad wyglądem i zachowaniem efektów.
- **Wykorzystanie Shader Graph:**
Integracja z Shader Graph pozwala na niestandardowe modyfikacje i manipulacje efektami wizualnymi, co otwiera drzwi do kreatywnych eksperymentów z teksturami, oświetleniem i innymi elementami wizualnymi.
- **Animacja i Interakcja:**
VFX Graph umożliwia dynamiczną zmianę i interakcję efektów w czasie rzeczywistym, co jest szczególnie przydatne w interaktywnych projektach gier.

Wady Unity Visual Effect Graph:

- **Wymagania Sprzętowe:**
Złożoność i zaawansowane możliwości Visual Effect Graph mogą wymagać większych zasobów sprzętowych. Starsze urządzenia lub te z ograniczonymi zasobami mogą napotkać trudności w obsłudze bardziej zaawansowanych efektów.
- **Dokładna Symulacja Kolizji:**
Particle System działający na CPU może zapewniać bardziej szczegółową symulację kolizji między cząstkami. Ta dokładność jest istotna, szczególnie gdy potrzebne są precyzyjne interakcje między elementami gry, co może być trudniejsze do osiągnięcia na GPU.
- **Gorsza Implementacja Fizyki:**
CPU oferuje elastyczność w implementacji i zarządzaniu logiką fizyki cząstek, co może być szczególnie przydatne w przypadku efektów, gdzie interakcje fizyczne są kluczowe dla zachowania realistycznego wyglądu.

- **Krzywa Uczenia:**

VFX Graph może wymagać czasu na przyswojenie, szczególnie dla osób początkujących. Jest to narzędzie o bardziej zaawansowanej funkcjonalności, co może skutkować dłuższym okresem nauki i przystosowania się.

8.11.1 Decyzja

Osoba, która podjęła się stworzenia systemu magii podjęła decyzję o wykorzystaniu **Visual Effect Graph** zamiast Particle System. Powodem były wcześniej wspomniane wymagania dotyczące ładnego i zaawansowanego wyglądu gry.

Dużym problemem z pewnością jest brak posiadania informacji odnośnie kolizji, gorsza fizyka oraz duża krzywa uczenia się. Jednak z pewnością są to problemy do obejścia, które można jakoś rozwiązać. Temat rozwiązywania tych problemów zostanie poruszony w dziale dotyczący systemu magii w podrozdziale opisującym pojedyncze ataki i mechaniki zaklęć.

Do nauki efektów stosowałem ten kanał:

<https://www.youtube.com/@GabrielAguiarProd>

8.10 System Magii

System magii jest bardzo rozbudowaną częścią naszej gry, co było ustalone w założeniach projektowych.

Gra aktualnie posiada aż **58 różnych ataków magicznych** (z czego **6 jest dostępnych tylko dla przeciwników**) o różnym wyglądzie, działaniu i funkcjonalnościach.

Dodatkowo każdy ze spelli może nakładać na przeciwnika lub gracza jeden z **4 efektów**. Dzięki wprowadzeniu dobrze rozwiniętego, parametryzowanego systemu, rozwinięcie i dodanie nowych czarów i efektów zajmuje bardzo krótko.

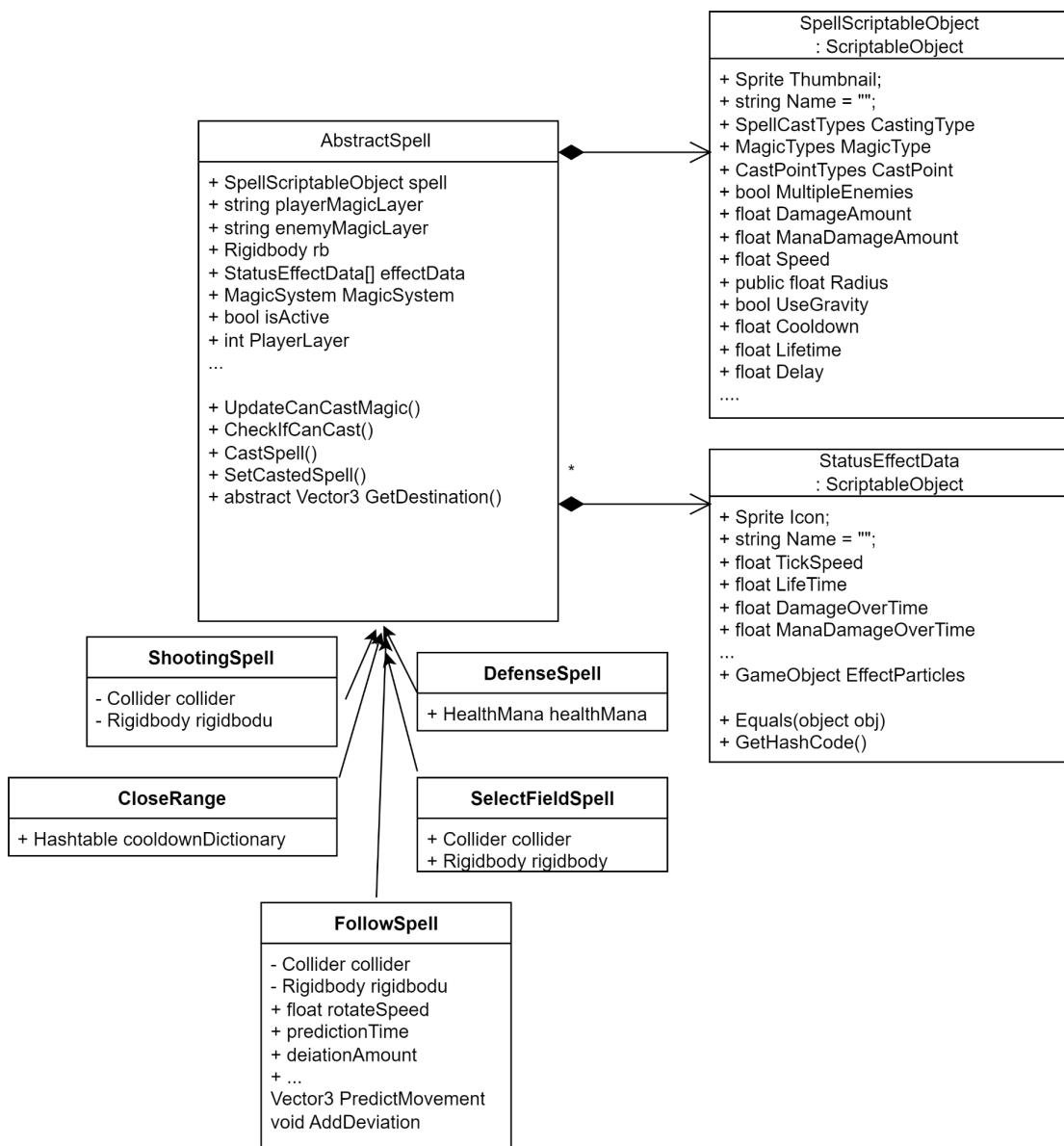
Wyróżnienie poszczególnych kategorii nie jest zadaniem prostym, ponieważ zaklęcia tego samego typu potrafią się znacznie różnić co do oddziaływanego oraz sposobu w jaki się przemieszczają.

Ogólny podział zaklęć na kategorie i podkategorie:

- Strzelające
 - Normalne
 - Wybuchające
 - Przyklejające się
 - Przyzywanie magicznych stworzeń
 - Shotgun
- Defensywne
 - Do położenia
 - Do nałożenia
- Naprowadzające
- Obszarowe
- Bliskiego zasięgu

Każda kategoria zostanie omówiona niżej w dokumencie, podczas przedstawiania charakterystycznych reprezentantów zaklęć danego typu, albo takich, które reprezentują ciekawe lub trudne mechaniki.

Na początek należy zacząć od omówienia klas podstawowych, które przetrzymują dane i odpowiadają za zachowanie magii.



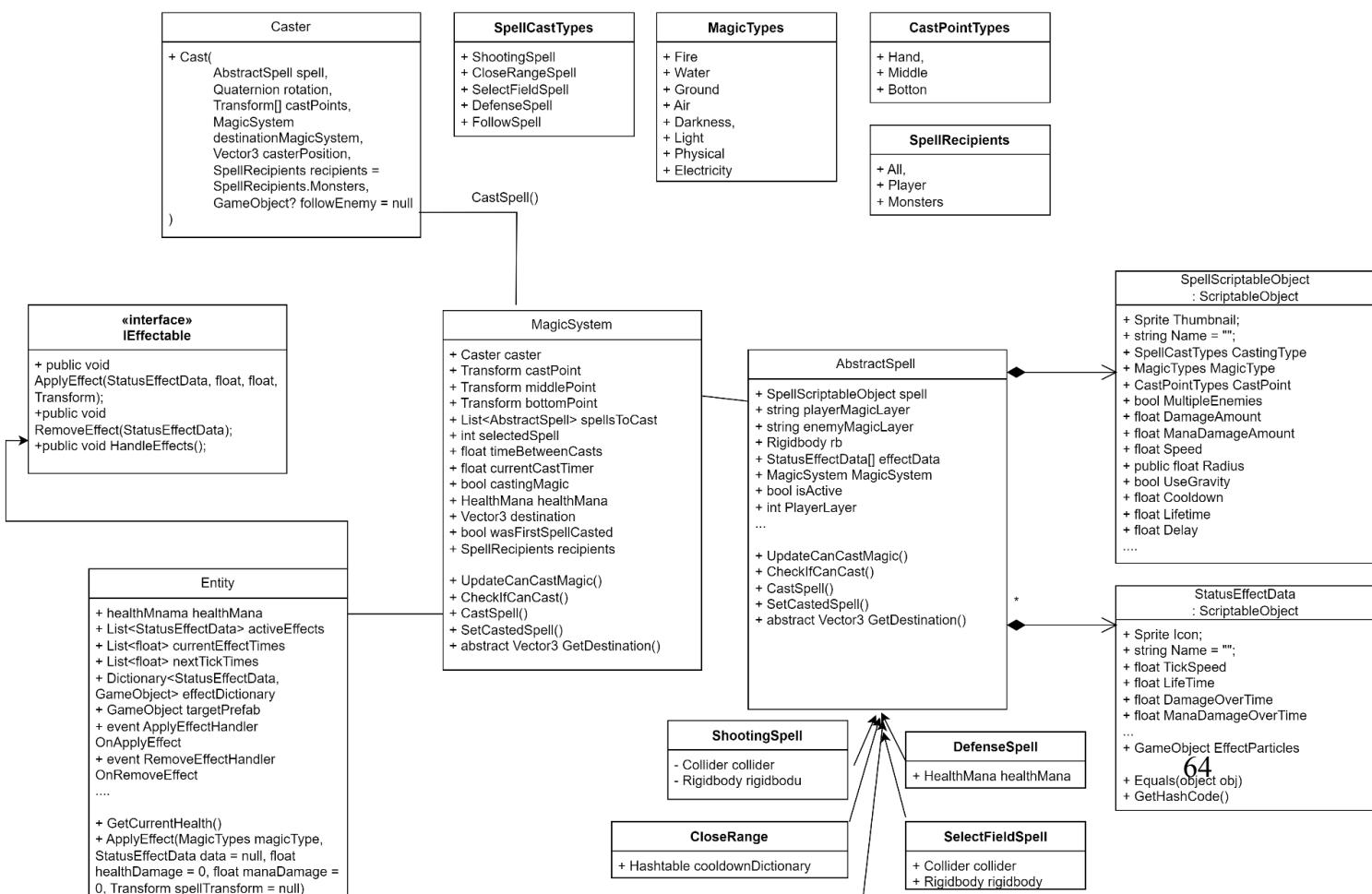
Klasa **SpellData** w Unity służy do przechowywania danych związanych z czarami. Jest to klasa reprezentująca konkretne informacje dotyczące czaru, takie jak jego nazwa, obrazek reprezentujący zaklęcie, typ rzucania, koszt many, obrażenia, czas życia oraz inne ustawienia związane z zachowaniem się czaru. Te wartości pozwalają zdefiniować zachowanie i właściwości różnych rodzajów czarów w grze stworzonej w Unity. Można je wykorzystać do tworzenia różnorodnych efektów magicznych oraz umiejętności w grze.

Klasa **StatusEffectData** w Unity służy do przechowywania danych związanych z efektem magicznym, który będzie nałożony na przeciwnika, którego zaklęcia z efektem trafią. Ta klasa pozwala na definiowanie różnych efektów, jakie mogą być nałożone na postacie w grze. Zawiera informacje dotyczące obrażeń, regeneracji, modyfikacji ruchu oraz innych efektów takich jak niewidzialność czy grawitacja (w aktualnej wersji nie jest zaimplementowane). Może być używana do tworzenia różnorodnych i zróżnicowanych efektów wpływających na rozgrywkę. Dodatkowo, zawiera metody do kopowania **Scriptable Object'ów** efektów, co może być przydatne do manipulacji nimi w grze.

Klasa **AbstractSpell** jest bazą dla różnych rodzajów czarów, gdzie można nadpisywać i dostosowywać jej funkcjonalność dla konkretnych rodzajów. Zawiera funkcje do manipulowania efektami czaru, interakcji z obiektami w grze i obsługi danych związanych z czarem. Może być podstawą do tworzenia różnorodnych umiejętności i efektów magicznych w grze.

Różne spelle oraz efekty, posiadają również skrypty pomocnicze, dzięki którym mamy możliwości manipulacji ich zachowaniem.

Castowanie spelli jest bardzo proste dzięki zaawansowanej implementacji systemu magicznego , który jest w stanie używać każde entity. Dodatkowo nie trzeba praktycznie nic przerabiać ponieważ zarówno gracz jak i przeciwnicy wystarczą, że użyją swojego magic systemu który ma castera. On wybiera jak dokładnie użyć danej magii.



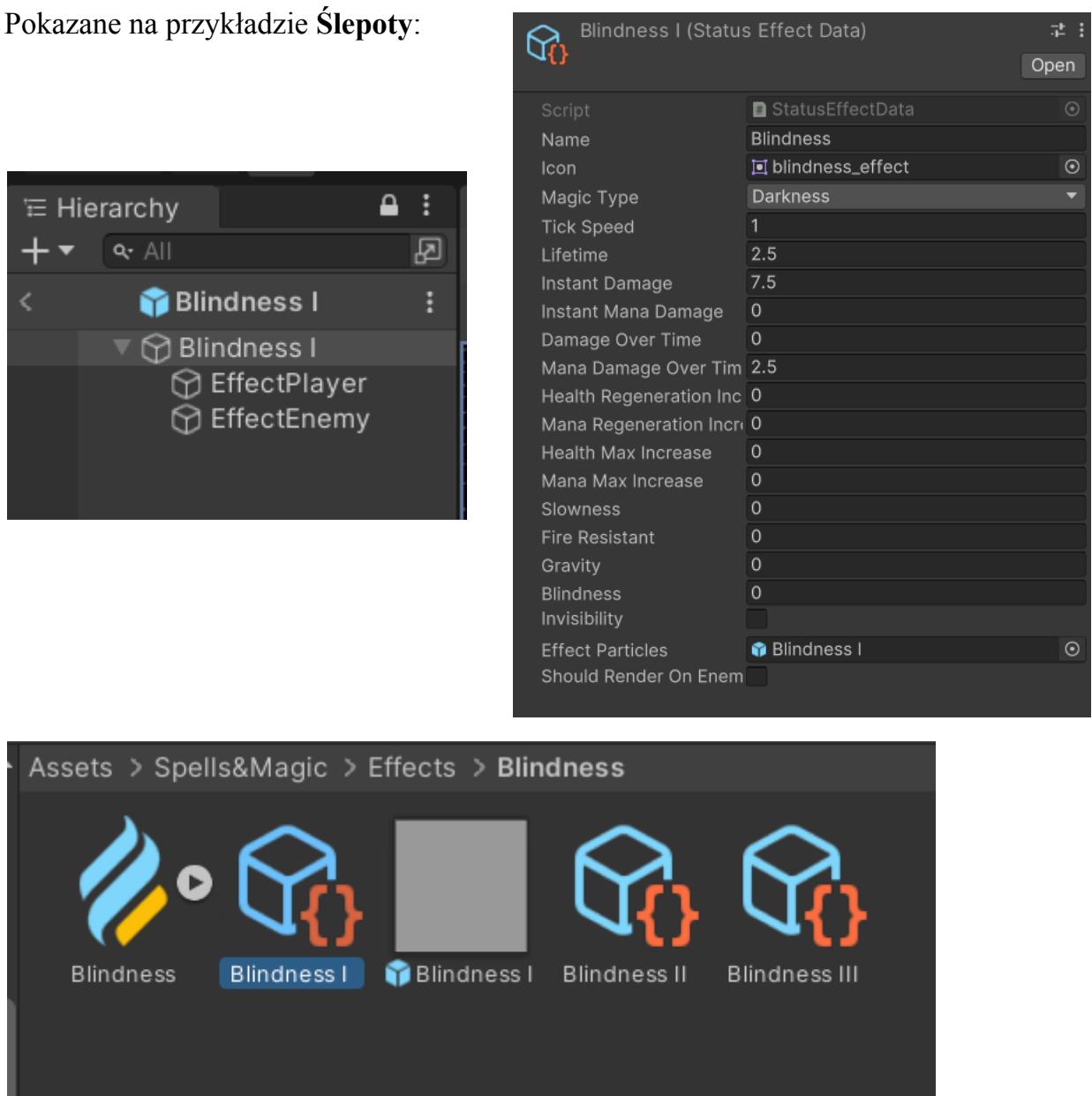
8.10.1 Efekty nakładane na gracza i przeciwników

Podstawowa budowa efektów polega na tym, że obiekt efektu zawiera skrypt **EffectObjectChecker**. Skrypt sprawdza, czy efekt jest nałożony na gracza czy przeciwnika, po sprawdzeniu aktywuje odpowiednią animację ataku oraz wyłącza te niepotrzebną.

Obiekt efektu może zawierać również skrypt **NotRotateWithParent**, dzięki któremu efekt nie obraca się kiedy robi to gracz lub potwór co daje lepszy efekt wizualny.

Prefab takiego efektu jest później umieszczany w StatusEffectData, które są potem przypisane do różnych zaklęć. Obiekt efektu może być wykorzystywany wielokrotnie w wielu efektach.

Pokazane na przykładzie Ślepoty:



Każdy blindness różni się od siebieobrażeniami jakie zadał, oraz czasem trwania efektu.

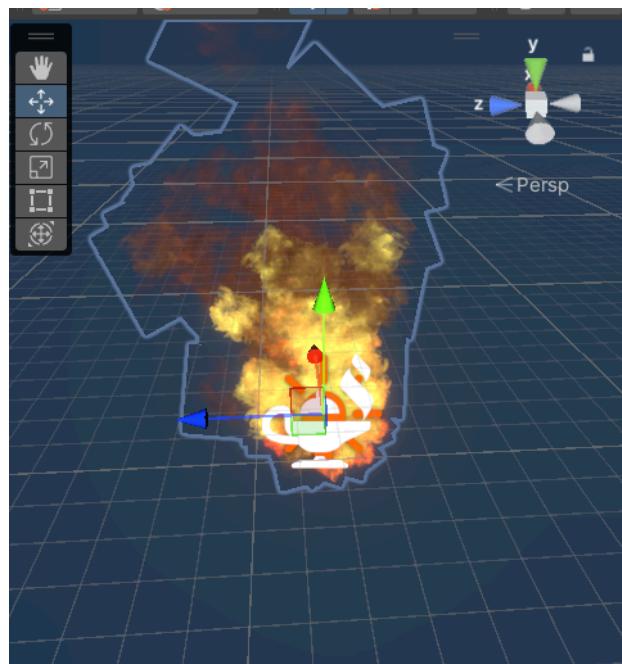
Nałożone efekty się sumują.

Oraz jak wygląda ten efekt w grze:

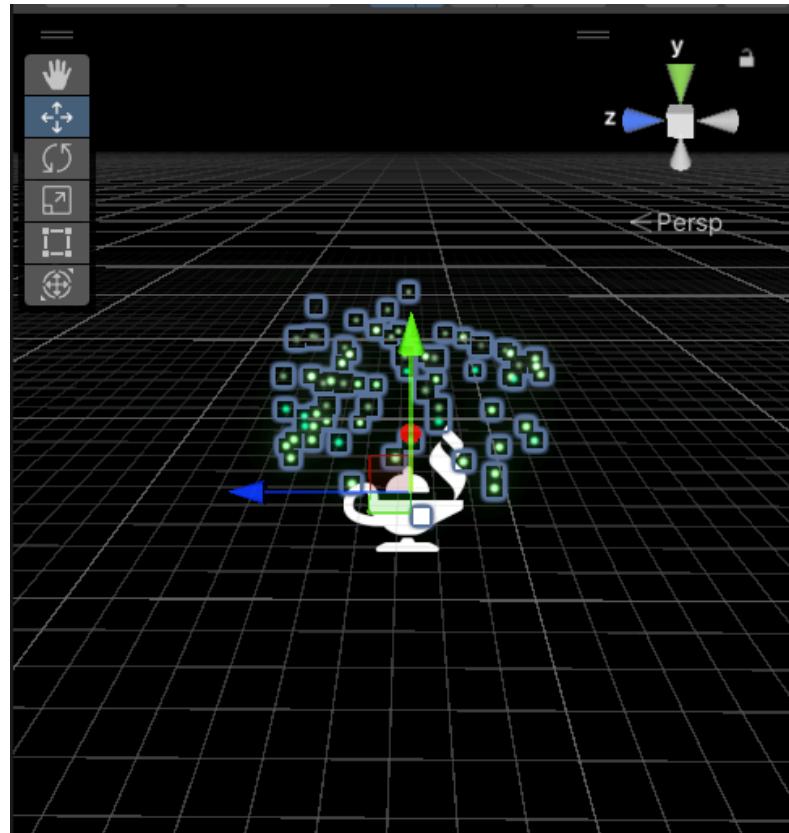


Na obrazku widać również, że efekt jest wyświetlany na ekranie w prawym górnym rogu.

Oprócz efektów ślepoty posiadamy również:



- **Podpalenia** - zadają obrażenia przez pewien czas oraz emituje światło
- **Zatrucie** - zadaje obrażenia przez pewien czas



Dołożenie kolejnych efektów jest bardzo proste dzięki dobrze przygotowanemu systemowi tworzenia obejmujący wiele właściwości nakładanych na gracza lub przeciwnika.

8.10.2. Pomocnicze klasy do zaklęć

8.10.2.1 interfejs ParentCollision

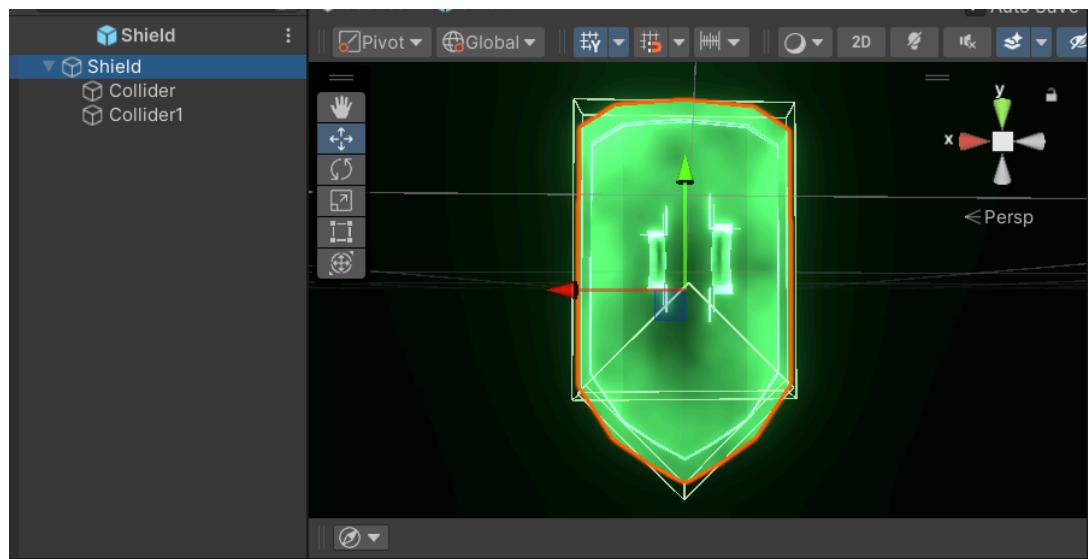
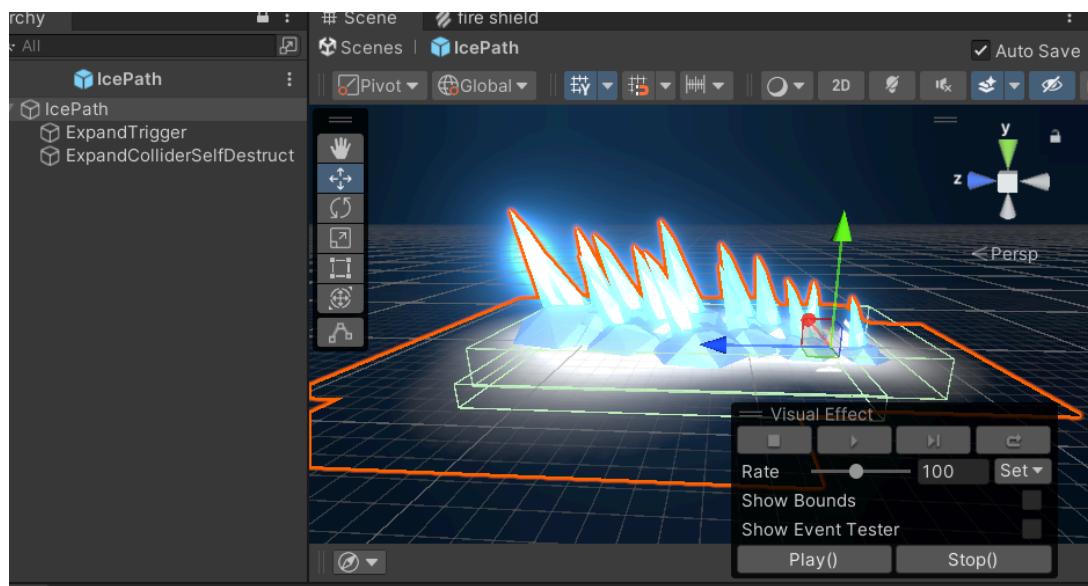
Interfejs ParentCollision służy do przekazywania informacji o kolizji oraz triggerze uruchomionym na dziecku danego obiektu. Jest to interfejs bardzo potrzebny nałożona na wszystkie rodzaje magii ataki. Pozwala to użyć kilku elementów - dzieci, które mają niżej opisany **ChildrenPassCollisionEvent** wraz z colliderami, żeby tworzyć bardziej skomplikowane struktury o wyglądzie

przypominającym dany atak magiczny, który sam w sobie nie ma informacji o kolizji (jedna z wad renderowania na GPU omawiana wcześniej).

8.10.2.2 Skrypt ChildrenPassCollisionEvent

Skrypt jest jednym z ważniejszych w magii, pobiera informacje o eventach związanych z kolizjami oraz triggerami i przekazuje wyżej do rodzica. W jednym obiekcie może być wiele dzieci, które go mają, żeby wiarygodnie odtworzyć prawidłowy wygląd zaklęcia. Często też po prostu główny obiekt musiał być na innej warstwie, albo działać inaczej (np stykać się z podłogą, albo wchodzić w kolizje z innymi obiektami), przez co musiał być na warstwie która nie miałaby kolizji z przeciwnikiem/ graczem. Jednak dzięki temu ataki zawsze przekazują prawidłowe informacje.

Jest używany na przykład w tarczy, żeby odtworzyć kształt, oraz w IcePath, aby jednocześnie mogło stać przyczepione do Ziemi i się rozszerzać



8.10.2.3 Skrypt AttachOnCollision

Skrypt jest jednym z najczęściej wywołujących uśmiech ludzi nim grających, ponieważ sprawia wiele radości grającemu. Obiekt z tym skryptem powoduje, że zaklęcie przyczepia się do innego obiektu po kolizji lub zetknięciu się z nim w miejscu dotknięcia. Aktualnie wykorzystywany jest w ataku przyklejającej się bąby **StickyBomb** omówionym w poniższych sekcjach. Pozwala to na ciekawą mechanikę, szczególnie gdy mamy do czynienia z latającym wrogiem, albo po prostu do rysowania wzorów na ścianach i podłogach podziemi.

8.10.2.4 Skrypt SpawnOnGround

Również jeden z ważniejszych skryptów potrzebnych do ataków. Jak nazwa wskazuje jest używany, aby zespawnowany obiekt na pewno znajdował się na ziemi i nie mógł “odlecieć gdzieś dalej”. Jest używane do zapewnienia, że obiekty spawnują się na odpowiedniej wysokości w grze, ignorując wszelkie nachylenia terenu lub nieregularności w powierzchni. Dodatkowo, flagi `shouldBeZeroX` i `shouldBeZeroZ` pozwalają na ustawienie rotacji obiektu na zero w wybranych osiach, co może być przydatne w niektórych przypadkach, aby kontrolować orientację obiektu.

Jest używany przez sporą liczbę ataków **GroundSlash**, **DarkGroundSlash**, **IceGroundSlash**, **IcePath**, **SummonRhino**, **SummonTiger**, **LavaWave**, **TsunamiWave**.

8.10.2.5 Skrypt TurnColisionAfterTime

Klasa **TurnColisionAfterTime** w Unity jest odpowiedzialna za wyłączenie i włączenie koliderów w określonym czasie po uruchomieniu sceny. W momencie uruchomienia obiektu, w metodzie `Awake()`, wszystkie kolidery zawarte w tablicy `collidersToDisable` są początkowo wyłączone. Następnie uruchamiana jest coroutine `EnableCollidersDelayed()`, która opóźnia włączenie koliderów o określony czas. Po upływie tego czasu coroutine włącza ponownie wszystkie kolidery zawarte w tablicy `collidersToDisable`. Ten skrypt jest przydatny w sytuacjach, gdzie konieczne jest tymczasowe wyłączenie koliderów, na przykład podczas inicjalizacji sceny, a następnie włączenie ich po pewnym czasie. Może to być używane do zabezpieczenia, animacji, efektów specjalnych lub w innych przypadkach, gdzie chcemy kontrolować widoczność kolizji w określonym czasie.

Ten skrypt jest używany w jednym ataku **MagicWeapon** i tam jest również omówione.

Najpierw jest odgrywana animacja dwóch toporów lecących a lewa i prawa a następnie zderzają się powodując eksplozje. Chodziło o to, aby włączyć kolizje w odpowiednim momencie.

8.10.2.6 Skrypt TransformFollowVelocity

Ten prosty skrypt w Unity, nazwany **TransformFollowVelocity**, manipuluje rotacją obiektu w taki sposób, aby jego przód wskazywał w kierunku, w którym porusza się obiekt za pomocą Rigidbody.

```
public class TransformFollowVelocity : MonoBehaviour
{
    void Update()
    {
        transform.LookAt(transform.position - GetComponent<Rigidbody>().velocity);
        transform.Rotate(xAngle: -90, yAngle: 0, zAngle: 0);
    }
}
```

Jeden z bardziej prostych, ale jakże przydatnych funkcjonalności.

Pierwsza linia kodu zmienia orientację obiektu w taki sposób, aby jego "przód" wskazywał w kierunku przeciwnym do kierunku, w którym porusza się obiekt. Dzieje się to dzięki funkcji LookAt, która obraca obiekt tak, aby jego przód był skierowany w stronę punktu docelowego. Tutaj punktem docelowym jest aktualna pozycja obiektu (transform.position) minus jego prędkość (GetComponent<Rigidbody>().velocity). Odejmowanie prędkości od pozycji powoduje, że obiekt patrzy w przeciwnym kierunku do swojego ruchu.

transform.Rotate(-90, 0, 0);: Ta linia kodu dodatkowo obraca obiekt o -90 stopni wzdłuż osi X. Jest to poprawka korekcyjna dla rotacji w przypadku, gdy potrzebna jest specyficzna orientacja obiektu. W tym przypadku, dodatkowa rotacja wynosi -90 stopni wzdłuż osi X, co jest być konieczne ze względu na to jak układłem ataki w przestrzeni.

Ten skrypt jest odpowiedzialny za to, że strzały mają idealny kurs lotu wzdłuż swojej prędkości.

8.10.2.7 Skrypt ExpandTrigger

Ten skrypt pozwala na płynne powiększanie BoxCollidera między wybranymi rozmiarami w czasie określonym przez duration. Jest to przydatne w różnych sytuacjach, takich jak aktywowanie lub wyłączanie obszarów kolizji w sposób animowany w grze.

W naszym przypadku używamy tego skryptu do zmiany wielkości collidera w ataku **IcePath**. Dzięki temu animacja wychodzenia lodowych

kolców ma sens i wybija przeciwników w powietrze lub na bok dodatkowo zadając obrażenia.

8.10.2.8 Skrypt MoveObjectWithCurve

MoveObjectWithCurve wykorzystuje krzywą animacyjną do płynnego przemieszczania obiektu pomiędzy dwoma pozycjami (startPosition i endPosition), jednocześnie kontrolując jego pozycję wzdłuż osi Y w czasie. To może być użyteczne do tworzenia animacji ruchu obiektów, gdzie chcemy kontrolować ich trajektorię ruchu lub zmianę wysokości w czasie w sposób kontrolowany przez krzywą animacyjną.

W naszej grze służy do przesunięcia Collidera EarthWall z taką samą krzywą z jaką jest stworzony efekt Visual Effect. Dzięki czemu mamy zgranā animację z kolizją tarczy przed atakami.

8.10.2.9 Skrypt ExplodeScript

Klasa **ExplodeScript** jest odpowiedzialna za wybuch obiektu w grze, wywoływanym na podstawie kolizji lub z opóźnieniem czasowym. Podczas zderzenia z przeciwnikiem pojawia się eksplozja, która wyświetla jakąś przypisaną do ataku animację. Klasa ExplodeScript definiuje logikę wybuchu w grze, uwzględniając warunki wybuchu, obsługę kolizji i skutki oddziaływania wybuchu na otoczenie. Postacie stojące blisko eksplozji są odrzucane pod odpowiednim kątem. Im bliżej stały tym mocniejsza siła na nich działa. Tak samo jest z zadawanymi obrażeniami. Jeśli istota, w którą strzelimy jest blisko dostanie więcej obrażeń niż ta, która stała daleko od nich.

Przykładowy wybuch świetlistej włóczni.



Występowały w kodzie jeszcze inne pomocnicze skrypty, ale nie były aż tak ciekawe.

8.10.3 Ataki strzelające

Część z tych ataków jest najprostszym rodzajem magii, która jest używana w grze. Często to jeden obiekt, który posiada:

- **VisualEffect** - zawiera efekt zrobiony w **Visual Effect Graph**
- **Rigidbody** - rigidbody jest jednym z kluczowych komponentów w silniku fizycznym w Unity, który symuluje zachowanie fizyczne obiektów w grze. **To komponent, dzięki któremu pomijamy brak zaawansowanej fizyki w VisualEffect!** Nadaje obiektowi właściwości fizyczne, takie jak masa, prędkość, siły, kolizje, grawitacja itp.
- **Collider** - w zależności od kształtu animacji ataku, może przyjmować różne kształty. Dla ataków strzelanych to najczęściej kula. Jest komponentem używanym do definiowania obszarów kolizji obiektów w grze. **Kolejny sposób do pominięcia braku kolizji w VisualEffect.**
- Shooting spell - klasa rozszerzająca abstract spell i implementująca zachowanie sprawdzające kolizję z obiektem, na który można nałożyć atak oraz jego efekty.
- Spell data - klasa kontener na duże ilości danych. Wydzielony z Abstract Spell dla czytelności
- Źródło światła - daje lepszy efekt atakom magicznym i ich otoczeniu.

Przykładem tego typu ataków są np. większość **BasicAttack**, **DarkOrb**, oba **ElectricityBall**, Większość **MagicOrb**, **Magic Arrow**, **Ice Freeze**.

8.10.3.1 Basic Attack II

Pierwszym inaczej zachowującym się atakiem jest BasicAttack II, jest on tym samym co Basic attack z lekko zmienionymi statystykami i kolorem. Jednak najbardziej różniącą go rzeczą jest inna liczba kul jakimi strzela oraz to, że kąt rozstrzału nie jest 0. (**UWAGA**, niestety po wydaniu okazało się, że ustawienia zostały zmienione dlatego opisuje teoretyczne prawidłowe działanie. Można to zmienić dwoma kliknięciami ale aplikacja została już przesłana do oceny)

W prawidłowej wersji liczba kul jest ustawiona na 3 a kąt rozstrzału ma 6. Wyłączona jest również opcja shotgunu. Dzięki temu gracz strzela jednocześnie 3 kulami z odpowiednimi kątami: -3, 0 i 3 stopni podziału w stosunku do normalnego wystrzału.



8.10.3.2 Wybuchające

Kolejnym innym atakiem są wybuchające. Oprócz zwykłego zachowania strzelających ataków, mają również wcześniej omówiony ExplodingScript, który powoduje eksplozje zadając obrażenia otoczeniu.

Przykładowy wybuch fireball'a



8.10.3.3 Basic Attack Shotgun

Od wyżej opisanego ataku BaciAttack II różni się tym, że wysyła 10 kul o kącie 2. Jednak włączony tryb shotgun zmienia trochę działanie. Teraz każda kula ma możliwość wylotu od gracza w kierunku ceownika z dodatkowym losowym kątem -2 do 2 zarówno w osi X jak i Y.



8.10.3.3 Przyklejające się

Atak, który posiada skryp **AttachOnCollision** oraz **Explosion** z pewnym opóźnieniem. Powoduje to ciekawą mechanikę, która przykleja się do przeciwnika, albo otoczenia, żeby po pewnym czasie zadać obrażenia.

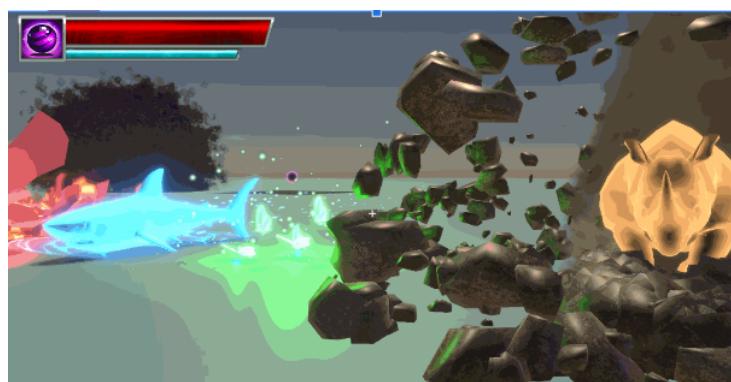


8.10.3.3 Przyzywanie magicznych stworzeń

Następnym niezwykłym atakiem strzelanym są ataki przywołujące stworzenia. Tygrys reprezentujący atak fizyczny. Kruk to przedstawiciel ciemności. Orzeł biały to światłość. Nosorożec to atak ziemisty. Rekin wodny. Motyle to atrybut wiatru. Feniks to oczywiście symbol ognia

Feniks, Kruk, Orzeł oraz motyle można wystrzelić jak normalny pocisk i lecą przez powietrze.

Nosorożec i rekin potrzebują podłoża po którym będą mogły się poruszać ponieważ biegą i płyną po danej powierzchni, dzięki skryptowi **SpawnDataOnGround**.



8.10.4 Defensywne

Następnym Rodzajem magii są zaklęcia defensywne. Charakteryzują się tym, że mają własne **HealthMana** dzięki czemu chronią gracza, przed atakami.

8.10.4.1 Defensywne do położenia

Pierwszym typem magii defensywnej jest typ tarczy do położenia. Ten typ tarczy stawiamy przed siebie. Chroni bardzo dobrze przed silnymi kierunkowymi atakami.



8.10.4.2 Defensywne do nałożenia

Kolejnym typem magii defensywnej jest typ tarczy do nałożenia. Ten typ tarczy stawiamy na siebie. Dzięki temu mamy zapewnioną ochronę z każdej strony. Szczególnie przydatny gdy będzie nas atakować duża liczba przeciwników



8.10.5 Strzelanie naprowadzane

W tym typie mamy specjalny celownik, który używamy do wyboru wroga. Atakiem celujemy na przeciwnika po czym magiczna pocisk (rakieta) sam znajduje określonego wroga i za nim podąża.



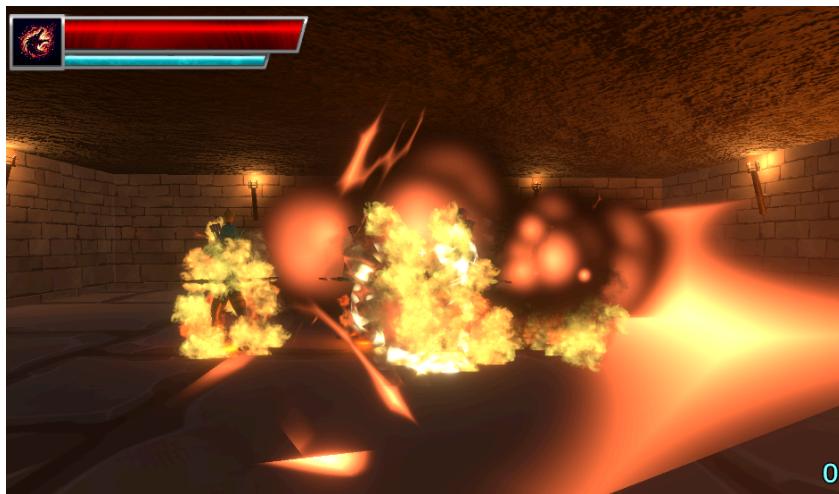
8.10.6 Ataki obszarowe

Atak obszarowy oznacza, że pod wpływem czaru będzie dany obszar. W tym typie mamy specjalny celownik koło. Ataku obszarowego dokonujemy przez wybór pola. Tam aktywowany jest atak na wrogach. Dzięki czemu zadajemy obrażenia sporej liczbie wrogów jednocześnie.



8.10.6 Ataki bliskiego zasięgu

Magia bliskiego zasięgu obejmuje pewien obszar od gracza, działa na wielu przeciwnikach. Zadaje obrażenia przez pewien czas. Przykładem może być ElectricJet albo FireBreath.



8.11 Menu & Score

Waczną częścią gry jest również jego menu. Niestety z braku czasu nie mogliśmy dodać do menu wielu opcji, które sugerowałyby ustawieniami graficznymi, poziomami trudności albo zaawansowanymi poziomami głośności. Jednak uważam, że jak na tak ograniczono czasowo projekt menu wyszło całkiem dobrze.

8.11.1 Pierwsze uruchomienie

Podczas pierwszego uruchomienia przywita nas film promocyjny/ wstęp, który jest opisany w punkcie niżej.

8.12 Fabuła & Film intro (promocyjny)

8.12.1 Początki gry, pomysł i potrzeba na fabułę

Z początku nasza gra nie miała posiadać żadnej fabuły. Był jedynie zarys czasów średniowiecznych. Jednak podczas zajęć seminarium profesor Leszek Borzemski podsunął nam pomysł, abyśmy nasze lochy umieścili w Sudetach. Za co bardzo Profesorowi dziękujemy! Dzięki temu mamy nawiązanie i promocję naszego regionu w grze, która może rozsławiać te tereny, podobnie jak to miało miejsce w przypadku Wiedźmina. Ponadto promocja gry bez jakiegokolwiek sensownej fabuły jest w dzisiejszych czasach ciężka, z powodu ogromnej ilości konkurencji na rynku. Podjąłem się tego zadania i mając zarys gry, gameplayu oraz miejsca ułożyłem fabułę pasującą do charakterystyki naszej gry.

Fabuła ta opowiada o upadku cywilizacji Sudetian i narodzinach bohatera, Wojmira, który musi stawić czoła mrocznym siłom, aby uwolnić Sudety spod klątwy. Świat gry wypełniony jest magią, potworami i tajemniczymi wymiarami. Imię Wojmir wybrałem dlatego, że jest słowiańskie oraz oznacza woj (wojownik) oraz mir (pokój), co pasuje do reszty fabuły.

8.12.2 Fabuła:

“W dawnych czasach, w samym sercu Sudetów, kwitła cywilizacja o wielkich mocach i tajemnicach. Ich kraina mlekiem i miodem płynąca, pełna miłości i dobrobytu.

Ludzie, zwani Sudetianami, opanowali magię, by współistnieć z potężnymi istotami z innych wymiarów.

Jednak ich chciwość i pycha wywołały gniew Przedwiecznych - starożytnych istot, które kiedyś zamieszkiwały te tereny.

Przedwieczni przeklęli tę ziemie, uwalniając potwory i tworząc przepaść między światami. Rozpoczęła się święta wojna między istotami magicznymi a ludźmi.

Cywylizacja Sudetian została zniszczona, a ich miasta zatonęły w mroku i zgubie.

Minęło wiele lat... Wraz z upadkiem Sudetian, ich zapomniane krainy opustoszały. W nieobecność ludzi, otworzyły się terytoria dla nowych władców – potworów.

Potworów, które przybyły z głębin innych wymiarów:

Goblinów, Golemów, Diabłów, Szamanów i Żyjących trupów

Z każdym oddechem wiatru, ich mroczne szeregi rozrastały się, coraz śmielej wychodząc poza granice podziemi, by siać spustoszenie w okolicznych królestwach.

W tym otępieniu los splótł wątki, tworząc bohatera z krwi i kości, zwanego Wojmirem. Jego dusza splatana jest z duchem Przedwiecznego, obdarzając go mocą manipulacji magii jakiej żaden człowiek jeszcze nie posiadał.

Poszukując Artefaktów Zapomnienia, Wojmir odkrywa dawne tajemnice i musi stawić czoła demonom własnej przeszłości. Walka z potworami staje się tylko jednym z wyzwań – prawdziwa bitwa to pogodzenie się ze zdradą, którą popełnili jego przodkowie, oraz odnalezienie siły, by przezwyciężyć klątwę, która spadła na Sudety.

Czy zdoła zmierzyć się z mrocznymi siłami, rozwiązać zagadki starożytnych manuskryptów i stanąć twarzą w twarz z Przedwiecznymi, by uwolnić Sudety spod klątwy i przywrócić równowagę między światami?”

8.12.3 Film promocyjny/ wstęp do gry:

Z chęci promocji naszej gry oraz pokazaniu zarysu fabuły na zajęcia seminarium postanowiłem nagrać film.

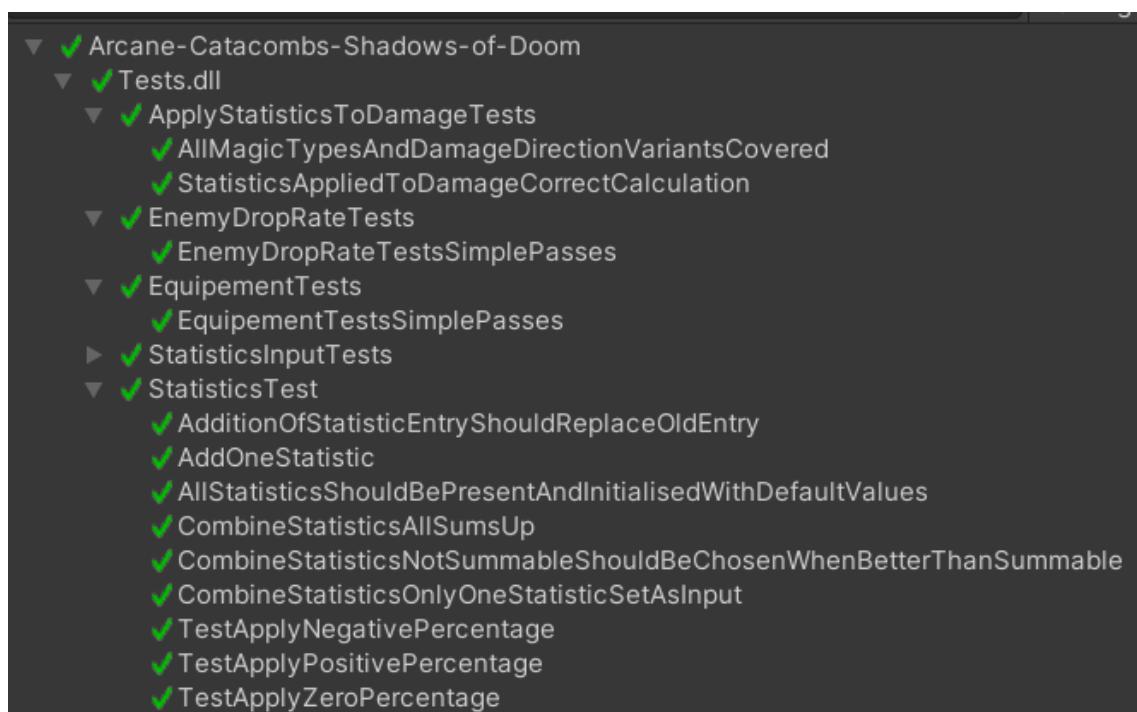
Do stworzenia filmu wykorzystano różnorodne narzędzia, takie jak Audacity do nagrania głosu lektora oraz Da Vinci Resolve do montażu obrazu i dźwięku. Współpraca z darmowym programem AI do generowania obrazów Playground.AI dostarczyła unikalne wizualne elementy (widoczne obrazy), które wzbogaciły prezentację. Film stanowi nie tylko wstęp do gry, ale również pełni rolę budującą atmosferę i przyciągającą uwagę potencjalnych graczy.

Link do filmu:
https://www.youtube.com/watch?v=X5Hb0InEEdk&ab_channel=WojtekBegierski
(o ile do tego czasu adres się nie zmienił powinien być aktualny)

9. Testy produktu programowego/Wyniki i analiza badań

Proces wytwarzania oprogramowania wspierany był przez testy jednostkowe oraz test użytkowników.

9.1 Testy jednostkowe



```
▼ ✓ Arcane-Catacombs-Shadows-of-Doom
  ▼ ✓ Tests.dll
    ▼ ✓ ApplyStatisticsToDamageTests
      ✓ AllMagicTypesAndDamageDirectionVariantsCovered
      ✓ StatisticsAppliedToDamageCorrectCalculation
    ▼ ✓ EnemyDropRateTests
      ✓ EnemyDropRateTestsSimplePasses
    ▼ ✓ EquipementTests
      ✓ EquipementTestsSimplePasses
    ▶ ✓ StatisticsInputTests
    ▼ ✓ StatisticsTest
      ✓ AdditionOfStatisticEntryShouldReplaceOldEntry
      ✓ AddOneStatistic
      ✓ AllStatisticsShouldBePresentAndInitialisedWithDefaultValues
      ✓ CombineStatisticsAllSumsUp
      ✓ CombineStatisticsNotSummableShouldBeChosenWhenBetterThanSummable
      ✓ CombineStatisticsOnlyOneStatisticSetAsInput
      ✓ TestApplyNegativePercentage
      ✓ TestApplyPositivePercentage
      ✓ TestApplyZeroPercentage
```

Testy te miały za zadanie sprawdzić wymagania, których ręczne sprawdzanie byłoby czasochłonne, oraz podatne na błędy algorytmy. Zastosowany został framework JUnit. Przykładem może być test

CombineStatisticsNotSummableShouldBeChosenWhenBetterThanSummable który sprawdza, czy w przypadku gdy statystyki niesumowalne są wybierane zamiast sumowalnych, gdy dają wyższe wartości:

```

[Test]
0 references
public void CombineStatisticsNotSummableShouldBeChosenWhenBetterThanSummable()
{
    Statistics[] statistics = new Statistics[4];
    for (int i = 0; i < statistics.Length; i++)
    {
        statistics[i] = new Statistics();
    }
    statistics[0].Add(new StatisticEntry(StatisticType.Intelligence, 40f, 5f, false, true));
    statistics[1].Add(new StatisticEntry(StatisticType.Intelligence, 23f, 3f, true, true));
    statistics[1].Add(new StatisticEntry(StatisticType.Strength, 5f, 8f, true, false));
    statistics[2].Add(new StatisticEntry(StatisticType.Intelligence, 20f, -2f, false, true));
    statistics[2].Add(new StatisticEntry(StatisticType.Strength, 6f, 3f, true, true));
    statistics[3].Add(new StatisticEntry(StatisticType.Intelligence, 9f, 7f, true, false));
    statistics[3].Add(new StatisticEntry(StatisticType.Strength, 10f, 5f, false, true));

    Statistics combined = Statistics.CombineStatistics(statistics);
    Assert.That(combined.Get(StatisticType.Intelligence).value, Is.EqualTo(40f));
    Assert.That(combined.Get(StatisticType.Intelligence).percentage, Is.EqualTo(7f));

    Assert.That(combined.Get(StatisticType.Strength).value, Is.EqualTo(11f));
    Assert.That(combined.Get(StatisticType.Strength).percentage, Is.EqualTo(8f));
}

```

9.2 Testy użytkowników końcowych

Każdy członek zespołu oraz wybrane osoby spoza zespołu przeprowadzały testy gry. Członkowie sprawdzali poszczególne funkcjonalności oraz swobodną rozgrywkę, zaś osoby postronne proszone były jedynie o swobodną grę, bez scenariuszy.

Zaangażowanie osób niepracujących nad grą pozwoliło na lepsze rozeznanie i zwrócenie uwagi na elementy, które mogły być przeoczone przez członków zespołu, lub podświadomie pomijane jako potencjalnie problematyczne.

Dzięki takiemu podejściu udało się znaleźć wiele błędów. Kilka przykładowych to:

- zbyt duże obciążenie systemu przy intensywnym korzystaniu z ognistej tarczy
- niedziałające collidery
- brak wczytywania zaklęć
- Z-fighting tekstur ścian

10. Podsumowanie

Tworzenie gier komputerowych jest bardzo skomplikowanym i czasochłonnym procesem. Wymagane jest sporo komunikacji między twórcami, aby produkt był spójny i moduły były w stanie się ze sobą poprawnie komunikować. Bardzo pomocnym narzędziem był system kontroli wersji git oraz zdalne repozytorium Github, które pozwoliły równolegle pracować nad różnymi elementami gry bez narażania reszty zespołu na problemy. Niemałą niespodzianką było dla nas to, jak często Unity przegenerowywuje pliki z metadanymi różnych assetów. Używając narzędzia Github należało być czujnym podczas commitowania zmian z assetami, gdyż, w przypadku nieuwagi, można było omyłkowo doprowadzić assety do stanu nieużywalności. Podział pracy na moduły zdecydowanie pomógł w wytworzeniu produktu, gdyż jeżeli z danym

elementem był jakiś problem, to łatwo było ustalić, kto ma największe szanse go rozwiązać. Zdecydowanie wytworzenie pełnej wersji gry tego typu w przeciągu kilku miesięcy graniczy z niemożliwym.

Samo doświadczenie zdecydowanie było pouczające i można było poprawić swoje umiejętności organizacji pracy w czasie i w grupie.

DOKUMENTACJA UŻYTKOWNIKA

1. Wprowadzenie Kontrolowanie postaci:

- W grze poruszamy się przy pomocy WSAD
- Do skakania używamy guzika SPACJA
- W celu wycelowania poruszamy myszką
- W celu użycia zaklęcia klikamy LPM
- Do wykonania DASH'a wykorzystujemy guzik E
- Do zebrania przedmiotu/otwierania drzwi wykorzystujemy guzik R
- Menu podręczne wyboru zaklęć uruchamiamy przytrzymując guzik TAB
- Menu z przedmiotami otwieramy korzystając z guzika I
- Menu statystyk otwieramy korzystając z guzika U
- Menu wyboru zaklęć otwieramy korzystając z guzika O
- Aby kucać przytrzymujemy guzik CTRL
- Aby biec przytrzymujemy guzik SHIFT
- Aby założyć przedmiot klikamy na niego, a następnie klikamy w odpowiedni kwadracik ekranu ekwipunku (np. Boots, Pants)
- Aby założyć zaklęcie klikamy na nie, a następnie klikamy na jeden z ośmiu kwadracików ekranu wyboru zaklęć

2. Instalacja produktu programowego

Wystarczy pobrać plik .zip i go rozpakować.

2.1. Wymagania systemowe

System: Windows 7/8/10/11 64-bitowy

2.2. Opis procesu instalacji

Gra nie wymaga instalacji. Dostępna jest w postaci pliku wykonywalnego .exe

2.3. Opis realizacji typowych zadań z podziałem na ich typy i/lub aktorów

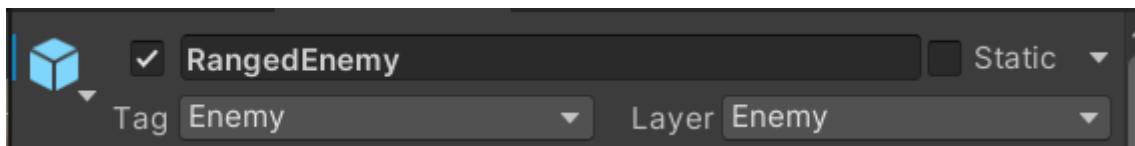
2.3.1 Realizacja kolizji

Kolizje realizowane są przez wbudowany do Unity system warstw kolizji (Collision Layer Matrix):

Physics		?	≡																
▼ Layer Collision Matrix																			
		Default	TransparentFX	Ignore Raycast	Player	Water	UI	Ground	Wall	Magic	Enemy	TriggerMagic	GroundMagic	DroppedItem	LayerMask	SeeThrough	Obstacle	PlayerMagic	EnemyMagic
Default	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
TransparentFX	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Ignore Raycast	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Player	✓	□	✓	✓	✓	□	□	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	□	
Water	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
UI	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Ground	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Wall	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Magic	✓	✓	□	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Enemy	□	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
TriggerMagic	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
GroundMagic	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
DroppedItem	□	□	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
LayerMask	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
SeeThrough	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Obstacle	□	□	□	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
PlayerMagic	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
EnemyMagic	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	

Macierz kolizji pokazuje, które warstwy kolidują ze sobą, np. Enemy nie może kolidować z EnemyMagic,

Każdemu obiektowi gry przypisujemy warstwę kolizji (Layer).



Ten przeciwnik zasięgowy jest na warstwie Enemy

Ważniejsze decyzje w sprawie kolizji:

- Gracz wchodzi w kolizję z przeciwnikami, aby umożliwić szkieletom osaczenie go (mamy szereg zaklęć obszarowych, więc jest to sensowne utrudnienie)
- Magia gracza wchodzi w kolizję z magią przeciwników - daje to możliwość anulowania silnych zaklęć przeciwników, uznaliśmy to za ciekawą mechanikę
- Magia przeciwników nie wchodzi w kolizję z przeciwnikami - sprawia, że przeciwnicy nie przeszkadzają sobie, zatem szkielety są użyteczne w blokowaniu zaklęć gracza, a nie blokują zaklęć zasięgowych przeciwników
- Przeciwnicy wchodzą w kolizję z innymi przeciwnikami, dzięki temu nie ma problemu "stackowania się" przeciwników (czyli zbierania się wielu przeciwników w jednym punkcie)
- Wydropione przedmioty nie wchodzą w kolizję z graczem, aby gracz nie utrudniać graczowi poruszania się po pokoju
- Wydropione przedmiotu wchodzą w kolizję ze sobą nawzajem, aby uzyskać przyjemnie wyglądające efekty przy wypadaniu wielu przedmiotów na raz
- Gracz nie wchodzi w interakcję z magią gracza, aby zapobiec sytuacji rozbicia się własnego zaklęcia o rzucającego

2.3.2 Główne cele

Głównym celem gry jest pokonanie jak największej ilości przeciwników w trakcie trwania rozgrywki. Gra składa się z poziomów, które złożone są z pewnej ilości pomieszczeń. W każdym pomieszczeniu gracz może znaleźć przeciwników do pokonania. Aby po pokonaniu wszystkich przeciwników w danym pokoju wykonać postęp w realizacji głównego celu gracz jest zmuszony przejść do kolejnego pokoju.

Oprócz pokonywania przeciwników gracz jest zachęcany do zapoznawania się z opisem i statystykami przedmiotów oraz statystykami ataków w celu optymalizacji zadawanych obrażeń i minimalizacji straty punktów życia.

Podczas eksperymentowania ze statystykami, gracz bardzo szybko zauważa, że pewni przeciwnicy są bardziej podatni na pewne typy ataków. Eksperymentując z różnymi atakami gracz będzie pozna więcej różnorodnych przedmiotów, dzięki czemu rozgrywka również będzie bardziej różnorodna.

2.3.3 Interakcje gracza

Gracz w trakcie rozgrywki może wchodzić w interakcje z:

- Przeciwnikami - może za pomocą czarów zadawać im obrażenia, a także otrzymywać obrażenia poprzez rzucone przez nich zaklęcia
- EquipableOnScene - gracz może podnosić elementy wyposażenia i czary, a także upuszczać je
- Drzwi - gracz może otwierać drzwi do kolejnego pokoju pod warunkiem, że wszyscy przeciwnicy w obecnym pokoju zostali pokonani
- Portal - przeciwnik może przechodzić przez portal na kolejny poziom lochów
- Elementy UI - opisane wcześniej elementy - menu, ekwipunek itemów, ekwipunek zaklęć, ekran statystyk