Dokumentacja Projektu Gry

Nazwa projektu:	GoQuest!
Nazwa zespołu:	GoDev!
Członkowie zespołu:	Daniel Kowalewski - programista, projektant, główna i jedyna głowa zespołu prowadząca cały projekt

---

Spis Treści

Opis gry

GoQuest! to 2D slash-action gra na komputery stacjonarne z systemem Windows. Miejsce gry to odległa kraina BlooLand, w której toczymy walkę między dobrem, a złem.

Cele projektu

Celem gry było:

- Zapoznanie gracza ze światem BlooLand i wydarzeniami z rozgrywki
- Rozwinięcie przyjemnego trybu gry poprzez:
    - Odkrycie - gracz odkrywa historię, nowe poziomy, a także zdobywa osiągnięcia i nagrody
    - Doświadczenie - gracz bardziej zagłębiający się w rozgrywkę, będzie posiadał więcej doświadczenia i możliwości rozwoju postaci
    - Walka - gra opiera się na nieustannej walce ze złem
- Styl graficzny “Pixelart” - oparta na wykorzystaniu prostoty grafiki, aby stworzyć unikalny styl

Produkcja

Gra głównie jest skierowana dla ludzi o następujących cechach:

- Od 13 do 16 lat
- Mężczyzna
- Posiada komputer PC
- Lubi gry
- Kupuje aplikacje online

Mimo, że gra jest bardziej dedykowana młodzieży to jest uniwersalna pod względem rozrywki dla każdego z graczy

Specyfikacja funkcjonalna

Mechanika gry

Gra główna

Gracz odkrywa grę poruszając się w 2D (dwóch wymiarach). Przeciwnicy pojawiają się w sytuacjach, gdy postać stara się wykonać zadania, bądź historię główną. Użytkownik musi pokonać zagrożenie jeśli chce przeżyć, oraz zdobyć doświadczenie. Gracz może spotkać na swojej drodze liczne utrudnienia, ale także NPC (non-player character), którzy będą wprowadzać fabułę, oraz dodatkowe misje.

Styl gry nawiązuje do prowadzenia dialogów, oraz wynoszeniu z nich informacji, które będą decydować o pomyślności naszych kolejnych poczynań.

Akcje jakie gracz może przeprowadzić:

1. Poruszać się w lewo, prawo, góra, dół
2. Zaatakować
3. Przeprowadzić rozmowę
4. Przenosić się między mapami
5. Wykonywać zadania

Postacie

1. *Tian*: wyrwany ze snu budzi się w lesie. Posiada miecz jako uzbrojenie i musi się dowiedzieć co się stało, bo nic nie pamięta. Gracz nim steruje.
2. *Nait*: twój brat bliźniak, którego musisz odnaleźć. Ponoć zna rozwiązanie zagadki BlooLand
3. *Bloo*: gluto podobne potwory. Nie wiadomo czym są, skąd pochodzą, ale wiadomo jest jedno, nie mają dobrych zamiarów
4. Mutant Bloo: niczym się nie różni ten glut od pozostałych oprócz tego, że zadaje potwornie wręcz obrażenia

Elementy rozgrywki

Elementy informujące o stanie postaci:

- *Pasek życia* - pokazuje aktualny i maksymalny stan życia. Posiada też pole tekstowe z precyzyjną ilością życia
- *Stan poziomu* - informuje nas o ilości zdobytego poziomu
- *Siła* - możliwość zadawania obrażeń postaci. Ustawiona jest domyślnie dla przeciwników i dla poszczególnego ekwipunku zadającego obrażenia

Fizyka i kolizja

Fizyka w grze:

- Gracz porusza się w świecie 2D (góra, dół, lewo, prawo)
- Gracz obserwuje grę z lotu ptaka
- Gracz koliduje z przeciwnikiem, w konsekwencji gracz dostaje obrażenia
- Postacie w grze mogą przesuwać gracza
- Grawitacja istnieje w grze
- Gracz napotykający blokadę, nie może nic zrobić aby przez nią przejść

Sztuczna inteligencja

Przeciwnicy:

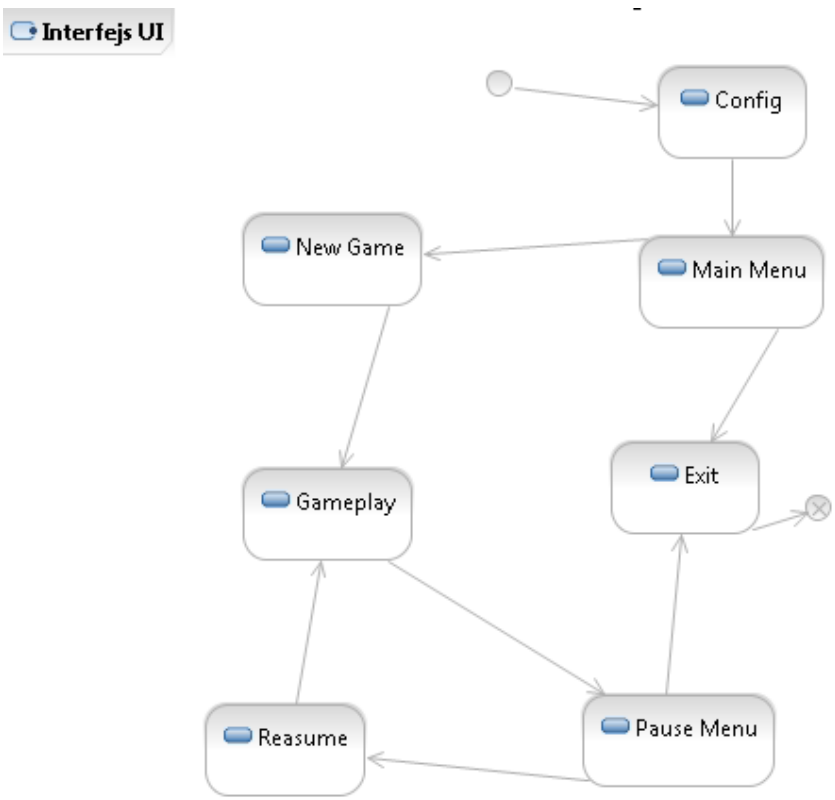
- Poruszają się po siatce XY(mogą się poruszać na ukos)
- Pojawiają się na mapie podczas odkrywania świata
- Nie gonią przeciwnika tylko losowo się poruszają w losowym kierunku
- Mogą zaatakować gracza poprzez kolizje z nim

NPC(non-player character):

- Poruszają się po siatce XY(mogą się poruszać na ukos)
- Pojawiają się na mapie podczas odkrywania świata
- Istnieje możliwość przeprowadzenia rozmowy z nimi
- Podczas kolizji odpychają gracza

Interfejs

Wykres stanów



- *Config* - okno nawigacyjne przeznaczone do konfiguracji programu
- *Main Menu* - prezentuje wszystkie możliwe wybory
- *Start* - utworzenie nowej gry w świecie
- *Gameplay* - miejsce gdzie rozgrywa się akcja, posiada HUD
- *Pause Menu* - obiekt interfejsu, który wywołuje pauze w grze
- *Resume* - kontynuacja danej gry
- Exit - wyjście, oraz zamknięcie aplikacji

Obiekty interfejsu

Start aplikacji

- START - nowa gra
- INFO - informacje dotyczace rozgrywki i tworcy
- EXIT - wyjście z aplikacji

Informacje

- MOVE – ruch postaci za pomoca strzalek
- ATTACK – atak za pomoca przycisniecia przycisku „J”
- PAUSE – pauza za pomocą wciśnięcia przycisku „P”
- BACK – powrót do głownego menu

Nowa gra

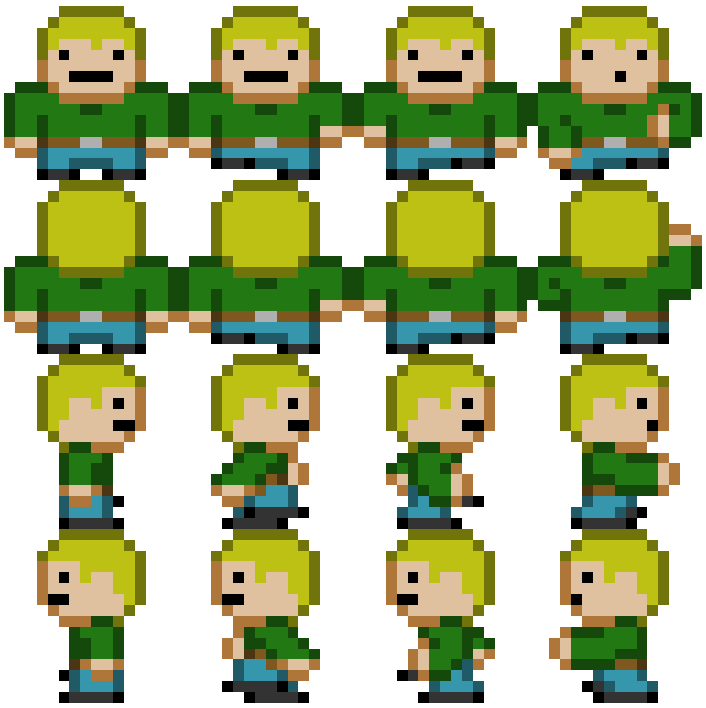
Po utworzeniu nowej gry na ekranie gracza ukażą się elementy interfejsu, które będą informować gracza o aktualnym stanie postaci. Obiekty te nazywane są HUD(Heads Up Display) i są to:

- *Pasek stanu życia gracza* - pokazuje aktualną ilość żywotności
- *Pole liczbowe ze stanem życia* - pokazuje liczbowo ile mamy życia
- *Pole tekstowe ze poziomem* - ukazuje aktualny stan poziomu postaci
- *Okno dialogowe* - pojawia się, wówczas interakcji z NPC

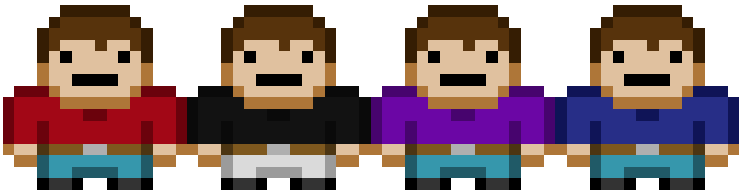
Modele i animacje

W projekcie zostały użyte darmowe modele postaci, tła, oraz obiektów. Na ich podstawie zostały utworzone animacje, lokacje, a także dodatkowe postacie.

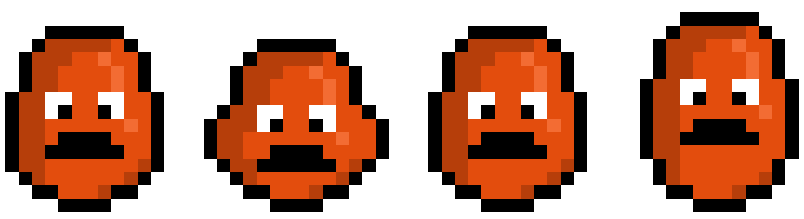
Animacje i model gracza



Model NPC



Model i animacja Bloo



Efekty audio i muzyka

Efekty dźwiękowe

Udźwiękowienie pochodzi z darmowej biblioteki dźwięków i wtyczek od Unity3D

Efekty SFX użyte w grze to:

- Explosion.mp3 - dźwięk oznaczający koniec gry i śmierć gracza
- Swoosh.mp3 - użyty jako dźwięk animacji ataku miecza
- Hurt.mp3 - odtwarzany w chwili utraty części życia przez gracza

Muzyka

Muzyki możemy używać zgodnie z licencją, która pozwala na dowolne używanie materiału. Została pobrana z tej samej strony co efekty dźwiękowe

Muzyka użyta w grze:

- Overworld.mp3 - użyte jako tło podczas przeglądania głównego menu
- Big Stream.mp3 - główny motyw towarzyszący podczas rozgrywki

Specyfikacja techniczna

Mechanika gry

Silnik gry

Do rozwoju gry został użyty silnik Unity3D, który został stworzony przez Unity Technologies. Silnik gry to system zaprojektowany, aby tworzyć gry na przeróżnych platformach takich jak konsole, komputery, czy urządzenia mobilne. Unity3D jest darmowym narzędziem.

Platforma i system operacyjny

Unity daje nam możliwość eksportowania gry na wiele różnych platform w tym: iOS, Mac Standalone, Windows Standalone, Sieć, Nintendo Wii, Xbox 360, PlayStation, Android.

Gra jest dedykowana pod platformę Windows, przez co można zagrać w grę komfortowo jedyna na komputerach z tym systemem, lecz w przyszłości planowane jest rozszerzenie możliwości grania na innych platformach.

Kod

Jak wcześniej zostało wspomniane gra korzysta z silnika gry Unity3D. Oprócz silnika zostały zaimplementowane dodatkowe funkcje. Użytym językiem programowania ze względu na Unity został wybrany obiektowy język C#. Do edycji kodu został użyty program Visual Studio 2017 od firmy Microsoft. Jest to środowisko programistyczny do kompilacji i pisania kodu. Jest ono darmowe.

Edytor map

Do utworzenia map gry został wykorzystany specjalny program Tiled, który współpracuje z Unity3D. Wybrany program ułatwiał dostęp do modeli terenów, a także w szybki sposób eksportował format pliku w edytorze na obsługiwalny przez sam silnik gry. Oprogramowanie to również jest bezpłatne.

Obiekty

Player

Jest to podstawowy obiekt w aplikacji. Służy on do kontroli postaci gracza. Jest otagowany jako „Player”. Korzysta ze skryptów PlayerController.cs, oraz PlayerHealthManager.cs. Pierwszy jest odpowiedzialny za kontrolę obiektu gracza, natomiast drugi to manager życia gracza.

Walk Area

Skrzynka kolizyjna, która definiuje gdzie NPC może się poruszać na mapie. Tworzy taką jakby zagrodę dla obiektu, który się porusza po planszy.

NPC

Obiekt odpowiedzialny za interakcję z graczem. Wprowadza możliwość pociągnięcia wątku fabularnego, a także wykonywania zadań zlecanych od tych postaci. Korzysta ze skryptu VillagerMovement.cs, który określa w jaki sposób może się pruszać obiekt.

Slime

Obiekt odgrywający rolę przeciwnika w grze. Korzysta ze skryptów HurtPlayer.cs, EnemyHealthManager.cs, SlimeController.cs. Kolejno skrypty odpowiedzialne są za system raniena bohatera poprzez nadanie ataku dla obiektu Slime, manager stanu zycia obiektu, oraz skrypt odpowiedzialny za kontrole obiektu Slime

CurrentMap

Obiekt mapy przekonwertowany na potrzeby i możliwości użytkowe Unity3D. Obiekt ten posiada swoje podobiekty, który składają się na cały schemat danej mapy. Posiada Skrzynki kolizyjne, które nie pozwalają graczu na przejście przez niektore obiekty.

Main Camera

Obiekt odpowiedzialny za funkcje kamery w grze. Jest to obiekt podążający za obiektem określonym, jako „Player”. Korzysta ze skryptu CameraController.cs, który określa żywotność kamery w aplikacji, oraz jej atrybuty.

StartPoint/VillageOut/HouseOut

Punkt, który opisuje wyjściowe miejsce gracza. Domyślnie jest ono tam gdzie znajduje się obiekt Player. Pełni funckję przeniesienia z wyjścia. Korzysta ze skryptu PlayeStartPoint. Nazwa wpisana w „Point Name” prowadzi nas do danego obiektu określonego nazwą wpisaną w pole.

HouseEntry/VillageIn

Punkt odpowiedzialny za wejście do sceny, bądź określonego w grze. Jest lustrzanym odbiciem działania obiektu StartPoint. Korzysta ze skryptu LoadNewArea.cs, który wczytuje daną scenę a następnie prowadzi nas do określonegu punktu utworzonego.

QuestTrigger:

Pole, który po kontakcje z graczem powoduje wyświetlenie konkretnego stanu zadania. Korzysta ze skryptu QuestTrigger. Za jego pomocą możemy określić, które zadanie ma być „zasugerowane", a następnie czy to ma być start tego zadania lub koniec.

QuestItem

Obiekt, który działa jako przedmiot do wykonania zadania w grze. Trzeba wejść w jego pole kolizji, aby dokonać zmiany stanu zadania. Korzysta ze skryptu QuestItem.cs, który nadaję nazwę przedmiotowi, oraz numer zadania z którym ta rzecz jest powiązana.

Bounds

Strefa kolizyjna, która warunkuje wędrówkę kamery. Korzysta ze skryptu Bounds.cs. Sprawia, że kamera nie wychodzi za zaznaczone pola na mapie, przez co rozgrywka wydaje się być bardziej realistyczna.

Slider

Informuje nas o stanie życia gracza i jego poziomie. Sklada sie na trzy elementy. Pasek zycia, Ilosc życia gracza zapisana liczbowo, oraz poziom gracza.

Quest

Zadanie do wykonania przez gracza. Mowi co gracz musi zrobić, aby kontynuować rozgrywkę. Używa QuestObject.cs, który określa numer zadania, manager do ktore powinien sie odwolywac, tekst towarzyszacy przy stacie zadania (tak samo przy zakonczeniu), oraz konkretne specyfikacje czy to jest zadanie na zabijanie ilosci potworow, na spotkanie sie z inna postacia, czy odnalezieniu konkretnego obiektu w grze.

Quest Manager

Jest odpowiedzialny za spis zadan dostepnych w grze. Korzysta ze skryptu QuestManager.cs, który opisuje ilość zadań w tabeli. Są to zadania możliwe do wykonania, oraz takie które już zostały wykonane, aby je odznaczyć z tablicy.

Pause Menu

Służy jako interfejs użytkownika. Po wciśnięciu odpowiedniego przycisków na ekranie za pomocą myszki, gracz jest w stanie wyjść z aplikacji albo powrócić do stanu sprzed wciśnięcia klawisza „P”.

Audio

Obiekt przechowywujący inne obiekty odpowiedzialne za zarządzaniem dźwięku w grze.

SFX Manager

Odtwarza konkretne dźwięki przy określonych warunkach. SFXManager.cs wskazuje na pliki, które mają być odtwarzane kolejno podczas obrażeń odniesionych przez gracza, przy śmierci gracza, przy ataku gracza.

Music Manager

Określa w jakiej kolejności powinna być odtwarzana muzyka. Dzięki skryptowi MusicController.cs z które korzysta ten obiekt, nie zdarzy się nam usłyszeć urwaną muzykę.

Dialogue Manager

Wyświetla okno dialogowe po spełnieniu określonych warunków. Dzięki skrypotwi DialogueManager.cs możemy wybrać pudło dialogowe, tekst który tam ma się znajdować, a także wybrać ilość linni w dialogu.

Skrypty – atrybuty i krótki opis

Korzystanie z wbudowanych bibliotek *Unity3D* zaoszczędziła wiele cennego czasu pracy nad projektem. Dzięki nim można korzystać z gotowych metod, oraz klas. Biblioteka *UnityEngine.SceneManagment* pozwala nam na manipulowanie scenami, poprzez funckje i metody dołączone do tej biblioteki. Cykl życia skryptu w *Unity3D* opiera się na funkcjach wykonywanych w kodzie. Z tej okazji mamy dwie funckje odpowiedzialne za działanie kodów. Przed akutalizacją pierwszej klatki wywołuje się funckję *void Start(),* w której określamy co ma się ładować na początku inicjacji. Funkcja *void Update()* wykonuje się cyklicznie raz na klatkę. W niej zapisujemy większość kodu. Poniżej jest zaprezentowany spis, oraz krótki opis wraz z atrybutami każdego pliku z kodem.

PlayerController

Ogólnie odpowiada za kontrolę obiektu „Player”. Wpływ fizyki, sposób poruszania się, animacje, oraz wpływ innych obiektów na „Player”. Wywołuje animacje, ciało fizyczne, oraz udźwiękowienie. Sprawdza też na samym początku czy dany obiekt istnieje, ustawia możliwość ruchu, oraz położenie ostatniego ruchu, jako dół. Co klatke sprawdza czy gracz się rusza i ustawia domyślnie wartość false dla poruszania się. Jeśli nie może się rusza przyspieszenie działające na ciało fizyczne jest ustawione jako 0. Pobiera siłę nacisku przycisku następnie konwertuje tą wartość na działanie przyspieszenia o określonym wektorze. Wproadzona zostala normalizacja ruchu, aby ruch był płynny i bardziej realistyczny. Jesli zostanie wcisniety przycisk „J” zmienia wartości ataku (czas ataku, czas miedzy interwalami, stan ataku tak/nie), oraz odtwarza dźwięk odpowiedzialny za atak. Zostają także wywołane animacje chodzenia, stania w określonym kierunku, ataku.

- **public** float *moveSpeed*;
  - prędkość poruszania się
- **private** Animator *anim*;
  - komponent Animator, który jest odpowiedzialny za animacje
- **private** Rigidbody2D *myRigidbody*;
  - komponent RigidBody2D nadaje obiektowi mozliwosc bycia kontrolowanym przez silnik fizyczny
- **private** bool *playerMoving*;
  - wartosc prawda/fałsz w przypadku, gdy gracz się porusza
- **public** Vector2 *lastMove*;
  - Vector2 reprezentuje wekotory i punkty w dwóch wymiarach. Nazwiązuje do ostatniego ruchu postaci
- **private static** bool *playerExists*;
  - wartość statyczna sprawdzająca czy gracz istnieje w grze
- **private** bool *attacking*;
  - wartość sprawdzająca stan ataku (tak/nie)
- **public** float *attackTime*;
  - długość w czasie ataku
- **private** float *attackTimeCounter*;
  - licznik pomiędzy atakami
- **public** string *startPoint*;

  - określa położenie punktu startowego gracza
- **private** float *currentMoveSpeed*;
  - bieżąca warość szybkości poruszania się
- **public** bool *canMove*;
  - sprawdza, czy gracz może się ruszyć w danym momencie

SlimeController

Nadaje kontrolę nad obiektami typu „Slime”. Określa działanie fizyki na ciało, poruszanie się, animacje, oraz interakcje z „Player”. Przy uruchomieniu wywołuje ciałą fizyczne i określa randomizację ruchu obiektu. Przy kojenych klatkach sprawdza czy obiekt się porusza.

- **public** float *moveSpeed*;
  - stała, prędkość poruszania się
- **public** float *moveInterval*;
  - czas między ruchami
- **public** float *moveTime*;
  - czas ruchu
- **private** Rigidbody2D *enemyRigidBody*;
  - wywołanie ciała na które działają prawa fizyki z silnika
- **private** bool *isMoving*;
  - sprawdza czy się rusza
- **private** Vector3 moveDirection;
  - określa kierunek *ruchu*
- **private** float *moveIntervalCounter*;
  - odlicza ilosc czasu miedzy ruchem
- **private** float *moveTimeCounter*;
  - ile ma trwac ruch
- **public** float *waitToReload*;
  - określa czas potrzebny to załadowania poziomu od nowa
- **private** bool *reloading*;
  - załadować od początku poziom
- **private** GameObject *thePlayer*;
  - obiekt gracza

CameraController

Opisuje działanie kamery w programie. Wywołanie rozpoczyna się od sprawdzenia czy kamera istnieje, oraz tego jak powinna się zachowywać w przypadku kontaktu z krawędziami mapy. Co klatkę jest aktualizowane położenie kamery. Kamera podąża za obiektem „Player”.

- **public** GameObject *followTarget*;
  - obiekt do podążania
- **private** Vector3 *targetPos*;
  - koordynaty celu do podążania
- **public** float *moveSpeed*;
  - prędkość ruchu kamery
- **private** static bool *cameraExists*;
  - sprawdzenie czy kamera istnieje

- **public** BoxCollider2D *boundBox*
  - wywołanie komponentu BoxColider2D
- **private** Vector3 *minBounds*;
  - ustawienie granicy minimalnej przestrzeni Bounds w wymiarze 3D
- **private** Vector3 *maxBounds;*
  - ustawienie granicy maksymalnej przestrzeni Bounds w wymiarze 3D

DestoryOverTime

Niszczy obiekt w czasie. Specjalny skrypt do pozbywania się niepotrzebnych obiektów. Jeśli czas spadnie do zera to automatycznie obiekt jest usuwany ze stanu gry.

- **public** float *timeToDestroy*
  - czas potrzebny do zniszczenia

dialogHolder

Przechowuje linie dialogu, gdyż są one wyświetlane linijka po linijce. Do przechowywania dialogów została użyta tablica. Można określić ile linii tekstu ma zawierać konwersacja. Aktywacji dialogu z postacami dokonujemy poprzez wciśnięcie klawisza „Spacja”. Posiada funkcję interakcji z obiektem „Player”. W przypadku użycia przycisku „Space” przez „Player”, przy obiekcie któremu jest przypisany skryp, odtwarza dialog przypisany danej postaci.

- **public** string *dialogue*;
  - co ma byc wyswietlone na poczatku
- **private** DialogueManager *dMan*;
  - wykorzystanie obiektu DialogueManager
- **public** string[] *dialogueLines*;
  - ilość linii tekstu przeznaczona na dialog

DialogueManager

Służy do zarządzania oknami z dialogami, oraz dialogami i ich liniami, które powinny zostać wyświetlone zgodnie z określoną kolejnością. Na statcie wywołuje obiekt związany ze skryptem PlayerController. Kolejno sprawdza czy dialog został aktywowany, oraz czy została odciśnięta spacja. Jeśli aktywna linii jest większa bądź równa linii ostatniej w tablicy to zostaje zakończona interakcja.

- **public** GameObject *dBox*;
  - wywołanie obiektu okno dialogowe
- **public** Text *dText*;
  - wywołanie obiektu tekst w oknie dialogowym
- **public** bool *dialogActive*;
  - sprawdzenie czy obiekt jest aktywny
- **public** string[] *dialogueLines*;
  - tablica przechowywująca linie dialogowe
- **public** int *currentLine*;
  - określa bieżącą linie dialogu
- **private** PlayerController *thePlayer*;
  - wywołanie obiektu PlayerController

EnemyHealthManager

Manager do obsługi stanu życia przeciwnika. Na starcie wyszukuje statystyki gracza, zadanie bierzące do wykonania oraz nadaje wartosci biżacemu i maksymalnemu stanu życia przeciwnika. Podczas odświeżania sprawdza czy przeciwnik został pokonany, czy zostały zadane mu obrażenia, a także czy dany osobnik należał do zadania.

- **public** int *enemyMaxHealth*;
  - maksymalna wartosc zycia przeciwnika
- **public** int *enemyCurrentHealth*;
  - wartość bieżącego stanu życia przeciwnika
- **private** PlayerStats *thePlayerStats*;
  - wywołanie statystyk gracza
- **public** int *expToGive*;
  - ile doświadczenia ma oddać po pokonaniu go

FloatingNumbers

Skrypt pokazujący ilość obrażeń zadanych przeciwnikowi. Napis wyświetla się w chwili uderzenia przeciwnika przez gracza. Znika on w czasie i unosi się do góry.

- **public** float *moveSpeed*;
  - szybkość unoszącego się tekstu
- **public** int *damageNumber*;
  - ilość obrażeń do wyświetlenia
- **public** Text *displayNumber*;
  - ilość obrażeń pobrana.

HurtEnemy

Określa co się dzieje podczas ataku gracza na przeciwnika. Na początku wywołuje statystyki bohatera, a następnie zadaje rany odpowiednie do bieżacych obrażeń gracza. Po zadaniu obrażeń odtwarzana jest animacja rozlewu krwi, i w tym samym czasie jest uruchomiany skrypt FloatingNumbers do ukazania ilości tych obrażeń.

- **public** int *damageToGive*;
  - ile zadać obrażeń
- **private** int *currentDamage*;
  - bieżąca wartość obrażeń
- **public** GameObject *damageBurst*;
  - wywołanie tak zwanego “rozlewu krwi”
- **public** Transform *hitPoint*;
  - punkt w którym dochodzi do zadania obrażeń
- **public** GameObject *damageNumber*;
  - wywołuje obiekt damageNumber

HurtPlayer

Identyczna zasada działania co w skrypcie HurtEnemy, lecz wszystko wiąże się teraz z graczem. Co nie jest wyświetlane podczas nadania obrażeń dla gracza to animacja rozlewu krwi.

- **public** Transform *hitPoint*;

  - punkt w którym dochodzi do obrażeń
- **public** GameObject *damageNumber*;
  - obiekt z liczbą obrażeń
- **public** int *damageToGive*;
  - ile oddać obrażeń
- **private** int *currentDamage*;
  - aktualna ilość obrażeń
- **private** PlayerStats *thePS*;
  - odwołanie do statystyk bohatera

LoadNewArea

Skrypt służacy do ładowania nowych środowisk w grze. Jeśli dojdzie do interakcji bohatera z tym obiektem jest on przenoszony na plansze o określonej nazwie.

- **public** string *levelToLoad*;
  - jaką scene załadować
- **public** string *exitPoint*;
  - punkt wyjścia
- **private** PlayerController *thePlayer*;
  - wczytanie obiektu Player

MenuManager

Menedżer do obsługi głownego menu. Posiada opis poszczególnych przycisków, oraz funkcje które działają po wciśnięciu przycisków. MenuManager jako jedyny nie posiada funkcji Update ze względu na to że nie ma co odświeżać w skrypcie co klatkę.

- **public** void ToGame(string sceneName)
  - przenosi nas do sceny o określonej nazwie
- **public** void *Exit*()
  - wychodzi z gry, wyłącza aplikację

PauseMenu

Menedżer do obsługi menu pauzy. Działa podobnie jak MenuManager z wyjątkiem, że nie posiada możliwości przenoszenia między scenami, a może wstrzymać stan gry. Ten skrypt w przeciwieństwie do MenuManager posiada void Update. Po wciśnięciu przycisku „P” gracz może zatrzymać grę.

- **public** bool *isPaused*;
  - czy gra aktualnie jest zapauzowana
- **public** GameObject *pauseMenuCanvas*;
  - odniesienie się do obiektu MenuCanvas który posiada obiekty w postaci przycisków, tekstów, oraz obrazków
- **public** bool *PauseExists*;
  - sprawdza czy pauza istnieje

PlayerHelathManager

Definiuje obsługę życia gracza, a także sprawia, że gracz podczas otrzymywania obrażeń zaczyna migać. Odtwarza dzwięk odpowiedzialny za zadanie obrażeń graczowi. W fazie startowej skryptu inicjujemy kolejno obiekty potrzebne czyli menadżer dźwięku, model

postaci, kontroller gracza, zakończenie gry. Co klatkę sprawdzamy czy gracz przypadkiem nie ma życia poniżej 0, jeśli tak to odtwarzany jest dźwięk. Jeśli miganie jest aktywne to gracz kolejno znika, pojawia się, znika i jeszcze raz pojawia. Obrażenia zadane graczowi także odtwarzają dźwięk. Wykorzystano liczniki pomiędzy interwalami migotania.

- **public** int *playerMaxHealth*;
  - maksymalna ilość życia gracza
- **public** int *playerCurrentHealth*;
  - aktualny stan życia gracza
- **private** bool *flashActive*;
  - czy migotanie jest aktywne
- **public** float *flashLength*;
  - jak długo powinien się świecić
- **private** float *flashCounter*;
  - licznik migotań
- **private** SpriteRenderer *playerSprite*;
  - wczytanie modelu postaci potrzebnego do migotania
- **private** SFXManager *sfxMan*;
  - wywołuje menadżer dźwięków
- **public** GameOver *GameOverScreen*
  - pobiera okno informujące nas o zakończeniu gry
- **public** PlayerController *thePlayer*
  - pobiera kontroler gracza

PlayerStartPoint

Ustawia startowe położenie gracza. Może też być użyte jako punkt do przechodzenia między planszami (wyjściowy). Skrypt się tylko wywołuje. Sprawdza czy istnieje kontroler gracza (oraz kamerę sprawdza)i jeśli on wejdzie w interakcję z punktem to przenosi gracza w oznaczone miejsce.

- **private** PlayerController *thePlayer*;
  - wywołanie obiektu kontrolera gracza
- **private** CameraController *theCamera*;
  - wywołanie obiektu kontrolera kamery
- **public** Vector2 *startDirection*;
  - w jakim kierunku powinna zostać zwrócona postać
- **public** string *pointName*;
  - pobiera nazwe punktu

PlayerStats

Definiuje ilość możliwego doświadczenia do zdobycia, ilość poziomów do zdobycia przez gracza, kolejno przyrost życia, ataku oraz obrony na poziom. Na starcie wywołuje tablice z wartościami pierwszych elementów z tablic zadeklarowanych jako życie,atak i obrona.

- **public** int *currentLevel*;
  - obecny poziom gracza
- **public** int *currentExp*;
  - obecna ilość doświadczenia zdobyta przez gracza

- **public** int[] *toLevelUp*;
  - tablica z warunkami co do zdobytego poziomu za punkty doświadczenia
- **public** int[] *HPLevels*;
  - tablica z poziomami życia gracza
- **public int[]** attackLevels;
  - tablica z poziomami wartości ataku gracza
- **public int[]** *defenceLevels*;
  - tablica z poziomami wartości obrony gracza
- **public** int *currentHP*;
  - aktualna ilość życia
- **public** int *currentAttack*;
  - aktuala ilość ataku
- **public** int *currentDefence*;
  - aktualna ilość obrony
- **private** PlayerHelathManager *thePlayerHealth*;
  - odwołanie do obiektu

QuestItem

Określa czy dany obiekt jest przedmiotem do zadania. Obiekt posiadający ten skrypt, oraz po uzupełnieniu spejalnych danych staje się przedmiotem zadaniowym. Wywołuje menadżer zadań z którego pobiera potrzebne zadanie. Przy interakcji „Player" z „QuestItem", gracz podnosi przedmiot, przedmiot znika, a stan zadania jest aktualizowany.

- **public** int *questNumber*;
  - numer zadania
- **private** QuestManager *theQM*;
  - odwołanie do menadżera zadań
- **public** string *itemName*;
  - nazwa przedmiotu

QuestManager

Manadżer do zadań. Służy do obsługi zadań. Kiedy mają być wyświetlane dialogi, oraz potwierdzenie wykonania zadania.

- **public** QuestObject[] *quests*;
  - wywołuje tablice typu QuestObject
- **public** bool[] *questCompleted*;
  - nadaje wartość w tablicy czy dane zadanie zostało wykonane
- **public** DialogueManager *theDM*;
  - wywołanie obiektu typu DialogueManager
- **public** string *itemCollected*;
  - nazwa przedmiotu zebranego
- **public** string *enemyKilled*;
  - ilość pokonanych wrogów

QuestObject

Sprawdza pokolei czy dany obiekt jest obiektem zadaniowym, a następnie sprawdza, czy to rzecz do podniesienia, czy to wróg do pokonania lub czy to postać/miejsce w które trzeba się udać. Określa kiedy zadanie ma się zakończyć.

- **public** int *QuestNumber*;
  - numer zadania
- **public** QuestManager *theQM*;
  - odwołanie do obiektu typu QuestManager
- **public** string *startText*;
  - wiadomość wyświetlana przy starcie zadania
- **public** string *endText*;
  - wiadomość wyświetlana przy zakończeniu zadania
- **public** bool *isItemQuest*;
  - sprawdzenie czy zdanie dotyczy podnoszenia przedmiotow
- **public** string *targetItem*;
  - nadaje konkretna nazwe przedmiotowi zadaniowemu
- **private** bool *isEnemyQuest*;
  - sprawdza czy zadanie jest zwiazane z zabijaniem
- **public** string *targetEnemy*;
  - nazwa celu do zabicia
- **public** int *enemiesToKill*;
  - ilość zabójstw potrzebna do wykonania zadania
- **private** int *enemyKillCount*;
  - licznik zabitych stworzeń

QuestTrigger

Sprawdza czy dane zadanie zostało aktywowane, bądź zdezaktywowane.

- **private** QuestManager *theQM*;
  - wywołanie obiektu typu QuestManager
- **public** int *questNumber*;
  - numer zadania
- **public** bool *startQuest*;
  - czy to jest poczatek zadania
- **public** bool *endQuest*;
  - czy to jest koniec zadania

UIManager

Skrypt odpowiedzialny za położenie interfejsu widocznego dla gracza podczas rozgrywki. Sprawdza czy istnieje dany obiekt w rozgrywce, jesli nie to go tworzy.

- **public** Slider *healthBar*;
  - wywołanie obiektu suwaka jako pasek życia
- **public** Text *HPText*;
  - wywołanie obiektu tekstowego jako stan życia bohatera
- **public** PlayerHelathManager *playerHealth*;
  - wywołanie obiektu typu PlayerHealthManager

- **private static** bool *UIExists*;
  - sprawdzenie czy interfejs istnieje
- **public** Text *LevelText*;
  - wywołanie obiektu typu tekst, jako poziom doświadczenia gracza
- **private** PlayerStats *thePS*;
  - wywołanie obiektu typu PlayerStats

VillagerMovement

Określa w jaki sposób powinien się poruszać NPC(non-playercharacter>postać, która nie jest graczem). Przy wywołaniu funkcji tworzy ciała na kóre bedą działa prawa fizyki. Następnie w sposób losowy jest generowany kierunek drogi NPC. Sprawdza, czy posiada przestrzen do chodzenia ustawioną z góry wcześniej i sprawdza czy się może ruszać. Przy co klatkowym wywołaniu funkcji działa w następujący sposób. Jesli nie ma interakcji NPC z graczem to może iść dalej, jeśli się nie może ruszyć to jego przyspieszenie zmienia wartość na zero. Później jest warunek dotycząc, czy postać się rusza. Nadaje możliwość poruszania się NPC po krawędziach pola, które definiuje granice poruszania się obiektu.

- **public** float moveSpeed;
  - warość prędkość poruszania się
- **private** Vector2 *minWalkPoint*;
  - granica punktu do chodzenia minimalna
- **private** Vector2 *maxWalkPoint*;
  - granica maksymalna punktu chodzenia
- **private** Rigidbody2D *myRigidBody*;
  - wywołanie ciała na która będzie działać fizyka
- **public** float *walkTime*;
  - jak długo ma się poruszać postać
- **public** bool *isWalking*;
  - czy postac jest w ruchu
- **public** float *waitTime*;
  - ile odczekac
- **private** float *walkCounter*;
  - licznik marszu
- **private** float *waitCounter*;
  - licznik oczekiwania
- **private** int *WalkDirection*;
  - kierunek zapisany liczbowo
- **public** Collider2D *walkZone*;
  - wywołanie obiektu Colider2D jako strefa przeznaczona do marszu
- **private** bool *hasWalkZone*;
  - wartość, czy posiada obszar do chodzenia
- **public** bool *CanMove*;
  - sprawdza czy mozna sie poruszac
- **private** DialogueManager *theDM*;
  - wywołanie obiektu typu DialogueManager