Politechnika Wrocławska

Wydział Informatyki i Zarządzania
kierunek studiów: Informatyka

Praca dyplomowa - inżynierska

Gra sterowana głosem

Krzysztof Heldt

słowa kluczowe:
Unreal Engine 4
Voice control
Game

krótkie streszczenie:
Celem pracy jest zaprojektowanie i zaimplementowanie gry sterowanej głosem. Przedstawiono krótko inne produkcje bazujące na podobnym systemie sterowania, projekt gry oraz szczegóły implementacyjne głównych mechanik opisywanej gry wideo.

opiekun pracy dyplomowej	.....	.....	.....
	Tytuł/stopień naukowy/imię i nazwisko	*ocena*	*podpis*

*Do celów archiwalnych pracę dyplomową zakwalifikowano do:**
a) kategorii A (akta wieczyste)
b) kategorii BE 50 (po 50 latach podlegające ekspertyzie)
* *niepotrzebne skreślić*

pieczątka wydziałowa

Wrocław 2017

TBD.

Streszczenie

Rynek gier wideo na dzień dzisiejszy jest już gałęzią rozrywki, której nie należy lekceważyć, a będzie on w najbliższych latach stale rosnąć. Jednak jego rozmiar niesie ze sobą nie tylko korzyści. Na rynku pojawia się aktualnie niezwykle duża liczba gier co sprawia, że trudno jest zwrócić uwagę konsumenta na swój produkt. Z tego powodu poza wysoką jakością wykonania gry potrzebna jest także zazwyczaj unikalna mechanika wyróżniająca produkcję od innych.

W pracy przedstawiona zostanie wizja gry wideo sterowanej w sposób odbiegający od standardowego schematu wydawania poleceń w grach na komputery osobiste. Zaprezentowany zostanie projekt i implementacja gry sterowanej za pomocą głosu. W pracy przedstawione zostanie kilka gier wideo implementujących podobne sposoby sterowania, dokument opisujący część koncepcyjną gry – GDD, a także opisy sposobu implementacji głównych mechanik gry.

Wynikiem pracy jest działająca gra zbudowana na komputery osobiste, pozwalająca na przeprowadzenie pełnej rozgrywki z komputerowym przeciwnikiem, za pomocą poleceń głosowych lub konwencjonalnego sposobu sterowania – klawiatury i myszki, w przypadku braku mikrofonu.

Abstract

Today's video game market is a branch of entertainment that should not be underestimated, and will continue to grow in the coming years. But its size does not only bring benefits. The market is currently experiencing an enormous number of games, which makes it difficult to attract consumers to your product. For this reason, apart from the high quality of the game, there is also needed a unique mechanic that distinguishes the production from the others.

In this paper, a vision of a video game that is different from the standard control in personal computer games will be presented. The design and implementation of a video game controlled by voice commands will be presented. The paper will present several video games that implement similar controls, a document describing game concept- GDD, as well as descriptions of how the main game mechanics are implemented.

The result is a personal computer game that allows you to play a full game with opponent controller by an AI, using voice commands or conventional keyboard and mouse control, in the absence of a microphone.

Słownik pojęć i skrótów

W pracy, ze względu na jej tematykę występuje wiele terminów i skrótów, które nie mają swojego dokładnego odpowiednika w języku polskim. Postaram się więc jak w tej części jak najlepiej opisać te pojęcia czy skróty.

- **2D/3D** – skrót od 2/3 Dimensions, określenie stosowane do sprecyzowania stylu graficznego i specyfiki rozgrywki, gry 2D dzieją się zwykle w świecie przedstawianym dwuwymiarowo (przykłady: Shovel Knight, The Binding of Isaac). Analogicznie gry 3D rozgrywają się w świecie trójwymiarowym, który to aktualnie jest najpopularniejszym typem gier. Należy także wspomnieć o grach typu 2.5 D, których mechanika opiera się na dwóch wymiarach, jednak wizualizacja świata jest trójwymiarowa (Trine, Inside)
- **4X** – eXplore, eXpand, eXploit, eXterminate – gatunek gier, gdzie sterujemy frakcją, która, na przestrzeni czasu odkrywa nowe tereny, kolonizuje je, a także rozwija się technologicznie, czy wchodzi w konflikty z innymi frakcjami. Bardzo często ze względu na mnogość opcji dostępnych dla gracza, gry te są grami turowymi (przykłady: seria Endless Space, seria Sid Meier's Civilization, seria Galactic Civilizations), jednak istnieją także gry rozgrywające się w czasie rzeczywistym (z możliwością aktywnej pauzy, przykłady: Stellaris, Distant Worlds, Sins of a Solar Empire)
- **Asset** – z angielskiego kapitał, określenie na pliki/dane gry niebędące kodem, czyli tekstury, modele, dźwięki, czy inne dane specyficzne dla gry. Pojęcie wzięło się z podobnego określenia w biznesie, gdzie „assets" oznacza zasób posiadany przez firmę - aktywa lub pasywa.
- **Vertical Slice, Alpha, Beta, Release** - Kolejne etapy w produkcji gry. „Vertical Slice" jest pierwszym grywalną wersją gry, często zawierającą wiele błędów, nieposiadającą dużej części mechanik, jednak pokazującą koncepcję gry. Pozwala ona stwierdzić, czy gra ma szansę odnieść sukces. „Alpha" zakłada, że większość mechanik przewidzianych w grze jest zaimplementowana, jednak dalej występują w nich błędy. „Beta", jest to moment, gdy wszystkie „duże" mechaniki gry są zaimplementowane i należy teraz zabrać się za poboczne mechaniki, poprawę balansu gry, i poprawę szczegółów, tzw. „polish". „Release" to etap, gdy gra jest gotowa do wydania.
- **Blueprint** – z angielskiego plan, odbitka. Mechanizm użytego w pracy silnika (Unreal Engine 4) pozwalający dziedziczyć po klasach zaimplementowanych w kodzie, za pomocą tworów istniejących tylko w edytorze. Udostępniają one możliwość dodawania logiki za pomocą wizualnego języka skryptowego, jak i stanowią bardzo wygodny sposób na ustawianie parametrów logiki, czy assetów stanowiących wizualizację obiektu w grze.
- **Design** – pojęcie odnoszące się do strony koncepcyjnej gry (design gry jest odpowiednikiem projektu klasycznej aplikacji), jednak ze względu na specyfikę metodyki wytwarzania gier, nie jest to, tak jak w przypadku zwykłej aplikacji projekt gry. Design obecny jest przez cały okres implementacji gry i stanowi swego rodzaju wytyczną, określając mechanikę gry na relatywnie wysokim poziomie abstrakcji, zostawiając szczegóły implementacyjne programiście.
- **FPS** – First Person Shooter, gatunek gier, w których świat widziany jest "oczami" bohatera. Gra zwykle w dużej części skupia się na walce za pomocą różnego rodzaju broni dystansowych, od renesansowych muszkietów (mod do gry Half-Life 2 - Battlegrounds) do futurystycznych bronii (seria Tribes, Hard Reset, Quake).

Uogólnieniem gatunku FPS są gry typu FPP (First Person Perspective), w których jedynym wymaganiem jest widzenie świata z perspektywy pierwszej osoby.

- **Gameplay** – pojęcie odnoszące się do ogólnie pojętej rozgrywki w grze. Stwierdzenie, że w gra ma „dobry gameplay" oznacza, że jej mechaniki zostały zaprojektowane i wykonane w takie sposób, że dostarczają graczowi satysfakcję z rozgrywki.
- **GDD** – Game Design Document, dokument stanowiący podstawę designu gry. Odzwierciedla on aktualny stan gry, jest wytyczną przy procesie implementacji, przechowuje pomysły na przyszłe mechaniki, a także argumentuje pewne wybory dokonane przez projektantów.
- **Hack and Slash (H&S/HnS)** – gatunek gier, często także łączony z gatunkiem ARPG (Action Role Playing Game). W grach tego typu gracz walczy zwykle z niezwykle dużą liczbą przeciwników, lecz jest od nich zdecydowanie silniejszy. Gry te skupiają się na różnych sposobach eliminacji dużej liczby przeciwników, jaki na bardzo rozbudowanym systemie generacji przedmiotów (bardzo często losowej). Przykłady to seria Diablo, Path of Exile, seria Torchlight, czy Grim Dawn.
- **MOBA** – Multiplayer Online Battle Arena, gatunek gier, który powstał relatywnie niedawno (2003). Zapoczątkowany został przez mod do gry Warcraft III o nazwie Defence of the Ancients (DotA), w której do dwie pięcioosobowe drużyny graczy walczyły między sobą na prawie symetrycznej mapie. BOhaterownie sterowani przez graczy zabijali jednostki sterowane przez SI zdobywając złoto, za które kupowali przedmioty ulepszające statystyki sterowanych prze siebie postaci, aby w końcu zakończyć grę poprzez zniszczenie kluczowego budynku drużyny przeciwnej. Przykłady: DotA 2, League of Legends, Heroes of the Storm, Smite, Paragon.
- **Plugin** – „wtyczka" do silnika. Jest to zwykle zestaw klas które za pomocą mechanizmu rozszerzeń w silniku dokładają do niego dodatkowe funkcjonalności (zwykle dość specyficzne potrzebne do konkretnej gry czy typu gier), takie jak obsługa dodatkowego języka skryptowego, usprawnienia edytora przyspieszające wykonywanie pewnych czynności, czy po prostu zestaw pomocniczych funkcji do wykorzystania we własnym kodzie
- **RPG** – Role Playing Game, gatunek gier, w których gracz wciela się w postać (lub grupę postaci) i za jej pomocą przeżywa przygody. Gry tego typu kładą duży nacisk na kreację świata i historii. Współczesne gry RPG zwykle oferują graczowi otwarty świat (gry typu „open world"), wypełniony zadaniami fabularnymi dotyczącymi głównego wątku fabularnego, jak i zadaniami pobocznymi – zadaniami z historią zwykle niezwiązaną z głównym wątkiem fabularnym, przybliżające bardziej klimat świata, w którym znajduje się postać, frakcji w nim występujących itd. Gry RPG zwykle posiadają także mechanizm rozwoju postaci, często ściśle związany z mechaniką doświadczenia zdobywanego przez wykonywanie określonych akcji w świecie. Przykłady: seria Fallout, seria The Elder Scrolls, Pillars of Etrnity, Tyranny, Prey.
- **RTS** – Real Time Strategy, gatunek gier, w których gracz obejmuje kontrolę, nad pewnym kolektywem, frakcją, rasą, narodem, i za pomocą działań wykonywanych przez posiadane jednostki dąży do wypełnienia postawionych przed nim celów. Gry RTS, jak nazwa wskazuje dzieją się w czasie rzeczywistym (czasem posiadając opcję aktywnej pauzy). Gry RTS, czy w ogólności gry strategiczne, są dość szerokim gatunkiem, przez co niewiele można o nich powiedzieć bez rozdrabniania się na poszczególnymi seriami. Przykłady: seria Starcraft, seria Total War, seria Age of Empires, Seria Warcraft

- **UE4** – Unreal Engine 4, jeden z najpopularniejszych silników przeznaczonych do implementacji gier wideo, rozwijany przez firmę Epic Games. Przykłady gier na silniku UE4: Fortnite, Hellblade: Senua’s Sacrifice, ARK: Survival Evolved, Paragon

Spis treści

1. Wstęp

1.1. Wprowadzenie w tematykę

Historia gier wideo sięga roku 1947, kiedy to została wytworzona pierwsza taka gra o nazwie „Cathode-Ray Tube Amusement Device” [1]. Jednak aż do lat 80-tych gry wideo były dość niszową rozrywką, ze względu na dostępność takich rozwiązań, a także koszt sprzętu potrzebnego, aby się nimi cieszyć. Dopiero początek lat osiemdziesiątych przyniósł gwałtowny rozwój salonów z automatami do gier, tzw. „video arcades”, których to liczba uległa podwojeniu na przestrzeni lat 1980 – 1982 [2]. Jednak późniejsze lata przyniosły niespodziewane załamanie rynku elektronicznej rozrywki. W jego wyniku wartość sprzedaży sprzętu i oprogramowania do gier wideo zanotowała ogromny spadek z 3.2 miliarda dolarów amerykańskich w 1982 r. do 100 milionów dolarów amerykańskich w 1985 r. (dane dotyczą Ameryki Północnej) [3]. Dopiero premiera konsoli NES (Nintendo Entertainment System) zakończyła recesję i od tego momentu rynek gier wideo notuje coroczny, mniejszy lub większy wzrost.

Dzisiejszy rynek elektronicznej rozrywki prezentuje się oczywiście o wiele lepiej niż kiedykolwiek. Jego wartość w 2016r. osiągnęła pułap 104 miliardów dolarów amerykańskich, zaś aktualne prognozy przewidują, że w 2020 wartość ta wyniesie 143.5 miliarda dolarów amerykańskich (Rys. 1).

Rys. 1: Wartość rynku elektronicznej rozrywki w 2016r. i prognozowane wartości na lata 2017-2020

Źródło: Newzoo.com

Zdecydowana większość aktualnie wydawanych gier komputerowych opiera się na standardowym sterowaniu, czy to pad w wypadku konsol, czy to myszka i klawiatura w

przypadku komputerów osobistych. Pośród setek gier wydawanych w poprzednich latach tylko kilka z nich próbuje zintegrować sterowanie głosem jako sposób na sterowanie w grze, jednak w większości przypadków jest to dodatek pozwalający kontrolować głosem tylko część rozgrywki.

1.2. Przegląd istniejących rozwiązań

Spośród gier wydanych na przestrzeni 10 lat kilka z nich podchodzi do tematu sterowania głosem w sposób kompleksowy. Jedną z nich jest gra pod tytułem (Rys. 2) produkcji Iriduim Studios. Jest to gra taktyczna rozgrywana w czasie rzeczywistym w rzucie izometrycznym, w której dowodzimy grupą postaci i rozgrywamy taktyczne potyczki. Gra osadzona jest w klimacie science fiction, zaś sam system sterowania głosem pozawala na poruszanie się do wyznaczonych miejsc, atakowanie wyznaczonych przeciwników, zmianę broni poszczególnych członków drużyny, użycie specjalnych zdolności postaci, łączenie zestawów komend w skróty czy kolejkowanie wcześniej wspomnianych rozkazów.

Rys. 2: Zrzut ekrany z gry "There Came an Echo"

Źródło: Sklep Steam

Kolejnym przykładem może być gra zatytułowana „Tom Clancy's EndWar" (Rys. 3) studia Ubisoft Shanghai, wydana przez firmę Ubisoft. Jest to gra strategiczna w której gracz wciela się w dowódcę armii i wykonuje przeróżne zadania dowodząc jednostkami na polu bitwy, ataki na wyznaczone pozycje, obrony punktów czy eskorta innych, sprzymierzonych jednostek to tylko część z dostępnych do zrealizowania celów w misjach. Za pomocą głosu możemy wydawać polecenia jednostkom, rozkazując im poruszyć się do poszczególnych miejsc czy skupić się na ataku wyznaczonej jednostki.

Rys. 3: Zrzut ekranu z gry "Tom Clancy's EndWar"

Źródło: Sklep Steam

Ostatnią z prezentowanych gier, jest tytuł odbiegający trochę zastosowaniem sterowania głosem od pozostałych. Produkcja nosi tytuł „In Verbis Virtus” (*Rys. 4*) i została wydana przez studio Indomitus Games. Jest to gra zaliczająca się do gatunku RPG, gdzie wcielamy się w rolę czarodzieja eksplorującego podziemia. Tu sterowanie jest połączeniem tradycyjnego sterowania w grze (pad/klawiatura i mysz) i sterowania głosem. Samą postacią sterujemy tradycyjnie, zaś czarujemy wypowiadając odpowiednie frazy symbolizujące inkantacje zaklęć.

Rys. 4 : Jedna z grafik promocyjnych z gry "In Verbis Virtus"
Źródło: Sklep Steam

1.3. Cel pracy

Celem pracy jest zaprojektowanie i zaimplementowanie gry sterowanej głosem. Za pomocą poleceń głosowych będzie możliwe rozegranie pełniej rozgrywki bez potrzeby używania innych sposobów sterowania.

Zakres pracy obejmuje zbudowanie gry działającej na platformie PC. Wybór platformy, na której będzie działała aplikacja nie ogranicza jednak możliwości zbudowania jej na inne systemy, ponieważ do implementacji gry został wykorzystany silnik Unreal Engine 4, który jest w stanie budować aplikacje na wszystkie popularne platformy poczynając właśnie od komputerów osobistych, poprzez konsole i urządzenia mobilne, na platformach webowych kończąc.

Do implementacji samej gry użyty został komputer osobisty z systemem Windows, a użyte środowisko programistyczne z pomocą którego została napisana aplikacja to Visual Studio 2015.

Praca składa się z czterech części. Wstęp przybliża pokrótce rozwój rynku gier wideo i jego aktualny stan oraz opisuje klika wybranych gier wykorzystujących podobny sposób sterowania. W rozdziale drugim przedstawiony zostanie projekt gry, jej mechaniki i możliwości sterowania za pomocą głosu czy też konwencjonalnych środków. Ta część projektowa zostanie zaprezentowana w formie GDD (Game Design Document), który to jest standardem opisu gry komputerowej na etapie koncepcyjnym w branży. Rozdział trzeci będzie poświęcony implementacji gry. Zostanie w nim opisana użyta technologia, czyli silnik, za pomocą którego gra została zaimplementowana czy biblioteka użyta do rozpoznawania poleceń głosowych. Następnie opisany zostanie sposób implementacji sterowania głosem, a także innych mechanik opisanych w części projektowej. Na koniec tego rozdziału przedstawione zostaną perspektywy rozwoju gry. Czwarty, ostatni rozdział poświęcony zostanie podsumowaniu realizacji pracy. Pracę zakańcza wykaz literatury, na której opiera się praca, i która została w niej zacytowana.

2. Projekt

Faza projektowania gry wideo (tzw. preprodukcja), różni się diametralnie od standardowego projektowania aplikacji użytkowej. Są dwa główne powody takiego stanu rzeczy. Po pierwsze gra nie posiada wymagań funkcjonalnych, ponieważ nie jest to aplikacja wytwarzana w celu rozwiązania problemu czy usprawnienia procesu biznesowego. Zamiast tego gra stara się dostarczyć rozrywki potencjalnemu klientowi poprzez cztery główne sposoby wywoływania satysfakcji z gry (klasyfikacja według Hirokazu Yasuhary [4]):

- Competition – rywalizacja w grze, zrealizowana czy to za pomocą rankingu wyników innych graczy, czy też za pomocą bezpośredniej rywalizacji. Przykładami gier skupiających się na rywalizacji są gry sportowe, FPS (First Person Shooter) w trybie wieloosobowym, gry strategiczne w trybie wieloosobowym, a także gry typu MOBA (Multiplayer Online Battle Arena)

- Chance – losowość/hazard, gry tego typu starają się dostarczyć przyjemność płynącą z rozgrywki, poprzez podekscytowanie szansą na wygraną lub poprzez okiełznanie losowości przez gracza i użycie jej dla własnego zysku. Oczywiście te dwa sposoby nie wykluczają się wzajemnie i mogą się wspierać w dostarczeniu rozgrywki na wysokim poziomie. Przykładami takich gier są głównie gry typu Hack&Slash (Diablo, Torchlight, Path of Exile). Koncetrują się one na mnogości losowo generowanych przedmiotów, których wartość w grze może wahać się od całkowicie bezwartościowych, po unikatowe przedmioty warte wiele dni, czy nawet tygodni zbierania waluty wykorzystywane w grze.

- Role Playing – odgrywanie roli, wcielanie się w postać. Gry tego typu w dużej mierze skupiają się na kreacji świata i klimatu w grze. Zazwyczaj mają one bogatą historię, rozbudowany, otwarty świat i wiele postaci z bogatym charakterem. Głównym typem gier korzystających z tego sposobu dostarczania rozrywki są gry RPG (Role Playing Game), takie jak seria Fallout, Dragon Age, czy Divinity: Original Sin, ale także gry skupiające się tylko i wyłącznie na historii, takie jak Life is Strange, Heavy Rain

- Vertigo – uczucie prędkości, płynności ruchu, wykorzystywane głównie w grach zręcznościowych i wyścigowych, flagową serią jest tutaj seria Sonic, ale także gry z serii Need for Speed czy Burnout.

Oczywiście gra nie musi skupić się tylko na jednej z wyżej opisanych dróg. Mogą one być ze sobą łączone i przeplatane, co daje w wyniku gry, które zwykle są hybrydami gatunkowymi, jak gry typu MOBA, które wyszły z połączenia gier strategicznych i RPG.

Drugą podstawową różnicą pomiędzy projektowaniem gier a projektowaniem standardowej aplikacji użytkowej jest niezwykle wysoka płynność fazy projektowania. Oznacza to, że z jednej strony zaprojektowane mechaniki gry mogą okazać się nietrafione tzn. nie sprawiają, że gra się przyjemnie i będą wymagały całkowitego przeprojektowania już w trakcie implementacji, a dodatkowo zwykle w fazie projektowania mechaniki są wymyślane na abstrakcyjnym poziomie, dopiero w fazie implementacji ustalane są ich szczegóły, a także pojawiają się nowe mechaniki zainspirowane efektami implementacji mechanik już zawartych w dokumencie.

Z powodu tych różnic w projektowaniu gier nie stosuje się podejścia wykorzystywanego przy projektowaniu aplikacji użytkowych, czyli zdefiniowania wymagań, diagramów przypadków użycia, sekwencji i innych. Zamiast tego wykorzystuje się GDD (Game Design Document), który w przeciwieństwie do projektu aplikacji nie jest dokumentem sztywno definiującym zakres implementacji aplikacji, a raczej odzwierciedleniem aktualnej koncepcji gry i jej mechanik, propozycji implementacji nowych, a także formalnym ich ugruntowaniem. Płynność GDD sprawia, że często jest on określany mianem „żywego dokumentu" (ang. living document). Dobrze napisane GDD pozwala na późniejszym etapie uniknąć niedomówień związanych z implementacją mechanik.

W poniższym GDD będą pojawiały się opisy mechanik po których następuje dopisek „[NIY]" (Not Implemented Yet). Oznacza on, że dana mechanika, czy jej część nie została zaimplementowana w opisywanej aplikacji i jest bardziej pomysłem na rozwój aplikacji. Jest to ważne, ponieważ dzisiejszy rynek gier wideo coraz bardziej skłania się w stronę produktów, które po wydaniu są wspierane i rozwijane jeszcze przez długi czas, w przeciwieństwie do wcześniejszego standardu – wydać i zapomnieć.

2.1. Koncepcja gry

Gra jest strategią czasu rzeczywistego dla jednego gracza, w której to dwóch graczy (z których jeden jest sterowany przez SI) stara się nawzajem przejąć wszystkie planety należące do przeciwnika. Planety pod kontrolą graczy będą produkować statki zależnie od populacji planety. Populacja planety będzie wzrastać z czasem, prowadząc do coraz szybszej produkcji statków. Statki będą mogły przemieszczać się tylko pomiędzy wybranymi planetami, co pozwala zaprojektować miejsca dogodne do obrony i uczynić kolejność ekspansji bardziej strategiczną. Dodatkowo gra może być od początku do końca rozegrana tylko za pomocą poleceń głosowych.

2.2. Ocena biznesowa

Gry sterowane głosem są produktem rzadko spotykanym na rynku gier i z tego powodu są raczej produktem niszowym. Jednak od przeszło dwóch lat nie wyszła na rynku żadna większa produkcja, w której można by przeprowadzić pełną rozgrywkę tylko i wyłącznie za pomocą sterowania głosem. W połączeniu z faktem, iż poziom skomplikowania rozgrywki jest relatywnie niski, powoduje to, że gra powinna poradzić sobie dość dobrze na pułapie cenowym małych gier (5 – 10 €), jako ciekawa gra na kilka godzin.

2.3. Mechanika planet

Planety są podstawowym elementów rozgrywki. Każdy z graczy zaczyna grę posiadając jedną planetę, zaś utrata przez gracza jego wszystkich planet oznacza jego porażkę. Każda z planet, która nie jest neutralna z upływem czasu zwiększa swój poziom populacji, aż do maksymalnego poziomu. Poziom populacji wpływa na szybkość produkcji statków na planecie, im większa populacja, tym szybciej powstają nowe statki. Na orbicie planety może przebywać dowolna ilość statków dowolnej ilości graczy.

Planeta może zostać zajęta pod kilkoma warunkami:

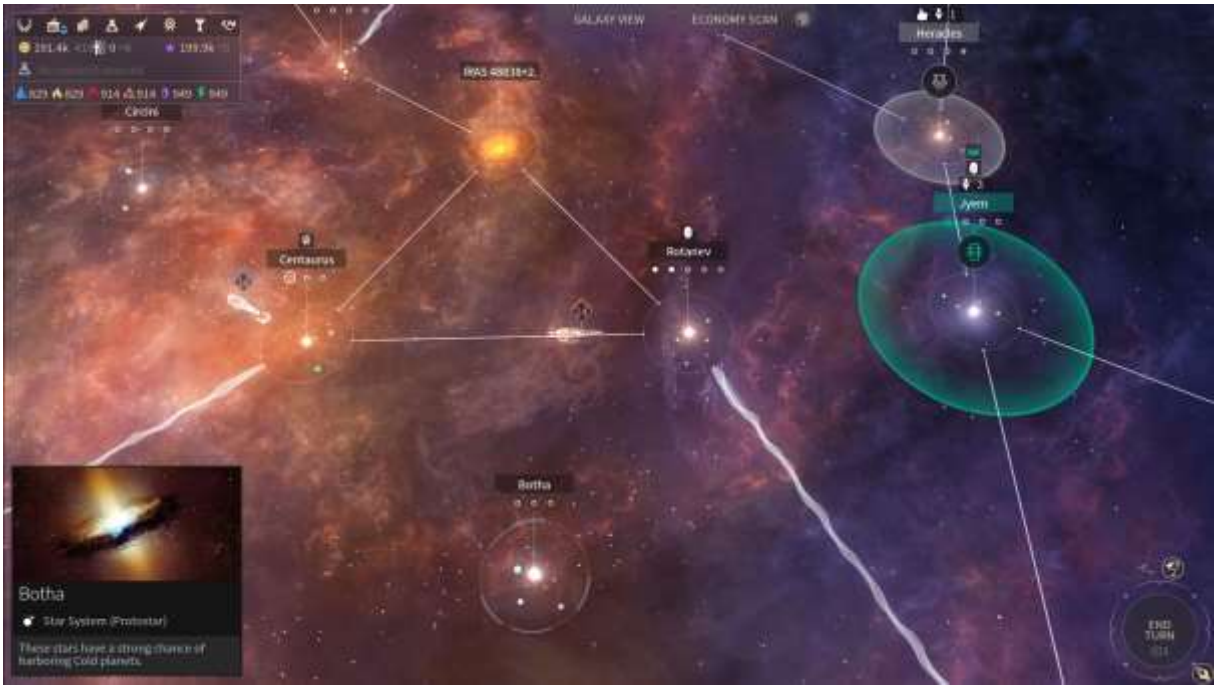
- Na orbicie planety nie ma żadnych wrogich statków

- Planeta nie jest już pod kontrolą gracza
- Gracz posiada na orbicie planety odpowiednią ilość siły militarnej w statkach

Po spełnieniu tych warunków liczba statków gracza na orbicie zajmowanej planety zostanie pomniejszona o wartość siły militarnej potrzebnej do zajęcia planety, populacja planety zostanie zmniejszona o połowę i właściciel planety zostanie zmieniony na nowego gracza.

Planety nie produkują statków ani nie rozwijają swojej populacji, jeżeli na orbicie znajdują się statki przeciwnika.

Każda planeta posiada pewną liczbę połączeń z innymi planetami i tylko za pomocą tych połączeń możliwe jest przemieszczanie statków pomiędzy planetami, jak to ma miejsce w większości gier typu 4X (skrót od eXplore, eXpand, eXploit, eXterminate) mających miejsce w kosmosie, przykładem może być gra Endless Space 2 (Rys. 5).



Rys. 5: Zrzut ekranu z gry Endless Space 2

Źródło: Sklep Steam

Planety mogą być różnych typów (kontynentalna, pustynna, arktyczna itd.) wpływając na prędkość rozwoju populacji, czas produkcji statku czy stopień przyspieszenia produkcji spowodowany poziomem populacji. [NIY]

Różne typy planet będą w stanie produkować różne typy statków, w zależności od poziomu populacji. Ma to na celu promowanie obrony własnych planet i możliwość obierania różnych strategii rozwoju, agresywnej zakładającej pokonanie przeciwnika zanim uda mu się osiągnąć odpowiednie rozmiary populacji, aby rozpocząć produkcję bardziej zaawansowanych statków, czy właśnie ukierunkowanej na rozwój, z założeniem, że uda się graczowi obronić kluczowe planety na tyle długo, aby później pokonać przeciwnika statkami o większej sile militarnej. [NIY]

2.4. Mechanika statków

Statki mogą istnieć tylko w dwóch miejscach:

- Na orbitach planet
- Jako flota statków przemieszczająca się pomiędzy planetami

Gracz może sterować posiadanymi przez siebie statkami za pomocą rozkazów ruchu, wskazując planetę docelową na które statki mają się udać. Statkom można wydawać rozkazy jedynie, gdy znajdują się na orbicie planety. Ruch statków odbywa się za pomocą następujących zasad:

- Przy określaniu ścieżki składającej się z kolejnych planet do których muszą udać się statki, aby osiągnąć swój cel planety zawierające statki przeciwnika są, w miarę możliwości omijane.
- Gdy flota osiągnie jedną z planet nie będzie mogła kontynuować ruchu, jeżeli na orbicie planety znajdują się wrogie siły, dopiero gdy zostaną one zniszczone flota będzie mogła kontynuować swój ruch

Statki posiadają zestaw statystyk określających ich właściwości:

- Punkty wytrzymałości – określają, ile obrażeń statek może przyjąć zanim zostanie zniszczony
- Punkty obrażeń – określają, ile obrażeń na sekundę podczas walki statek może zadać
- Punkty siły militarnej – używane do przejmowania planet, a także używane w interfejsie użytkownika do kalkulacji wyświetlanej siły floty podróżującej pomiędzy planetami czy stacjonującej na orbicie

Gdy na orbicie planety znajdą się statki należące do wrogich sobie graczy następuje między nimi walka, obrażenia zadawane przez każdą z flot są przydzielane jednemu ze statków floty otrzymującej obrażenia aż do jego zniszczenia, po czym wybierany jest kolejny statek, jeżeli po zniszczeniu poprzedniego statku zostały jakieś obrażenia do przydzielenia.

2.5. Sterowanie

W grze rozkazy można wydawać w sposób klasyczny, czyli za pomocą myszki i klawiatury lub za pomocą poleceń głosowych. Grę można w rozegrać w pełni, używając tylko i wyłączanie jednego ze sposobów, jednak nie ma też problemów, gdy podczas korzystania z obu naraz.

W przypadku myszki i klawiatury planety mogą być zaznaczone za pomocą LPM, zaś rozkazy wydawane są za pomocą klikania na docelowe planety za pomocą PPM.

Sterowanie głosem odbywa się za pomocą z góry ustalonych komend, w języku angielskim (przewidywane późniejsze wsparcie dla najpopularniejszych języków), za ich pomocą można zaznaczyć planetę wydać rozkazy ruchu na planetę lub za pomocą jednej komendy wydać połączenie obu tych rozkazów, czyli ruch z planety na planetę.

Dodatkowo dla sterowania głosem możliwe jest przypisanie często używanych rozkazów do poleceń-skrótów. Jednocześnie dostępna jest lista ostatnio wydanych rozkazów, które można wywołać odpowiednią komendą. [NIY]

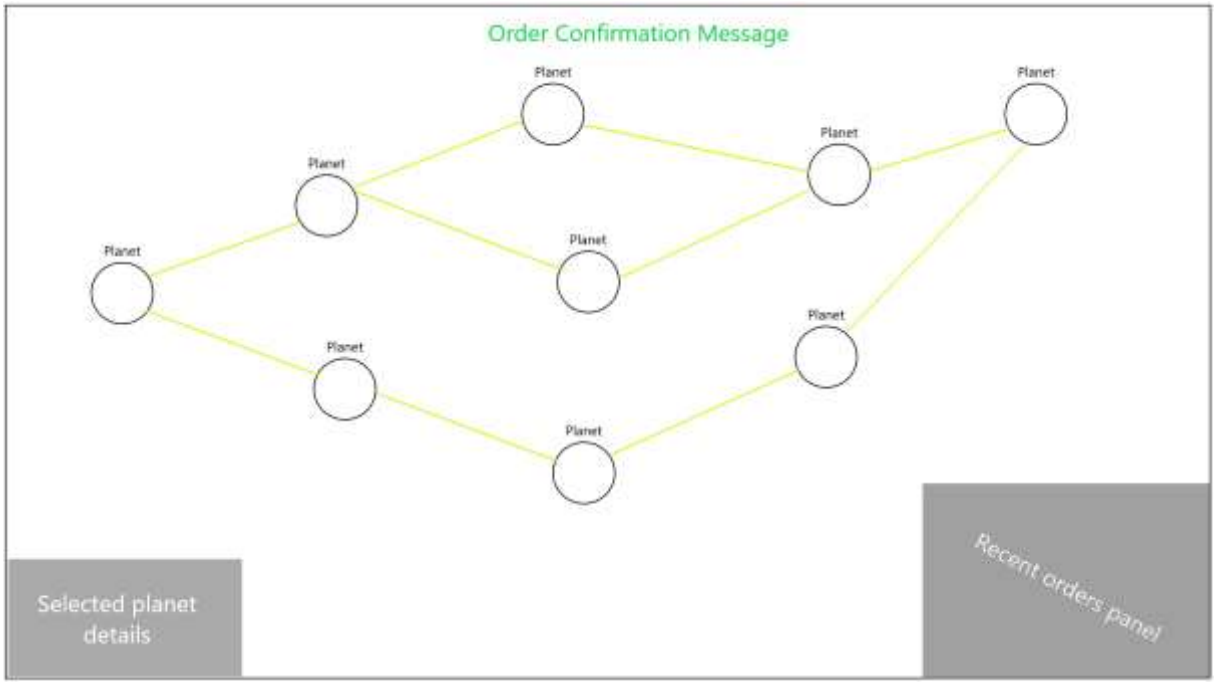
2.6. Konstrukcja map

Z powodu przyjętego założenia, że całą rozgrywkę, od początku do końca da się przeprowadzić za pomocą komend głosowych, a także z powodu przyjętej dynamiki rozgrywki, operowanie kamerą za pomocą komend głosowych nie wchodzi w grę. Z tego powodu mapy na których toczyć będzie się rozgrywka, muszą zmieścić się na pojedynczym ekranie i być czytelne dla użytkownika, co znacznie ogranicza ich rozmiar.

Rozwiązaniem na większe mapy może buc koncepcja „widoków" na różne części mapy między którymi gracz będzie mógł się przełączać za pomocą komend. Pociąga to jednak za sobą konieczność dodania minimapy informującej gracza o stanie całej mapy, zmniejszając efektywny widok na świat. [NIY]

2.7. Interfejs użytkownika

Brak możliwości poruszania kamerą sprawia, że każdy kawałek ekranu jest niezwykle cenny, z tego powodu interfejs użytkownika powinien być jam najbardziej minimalistyczny, aby zmaksymalizować przestrzeń użytkową (Rys. 6).



Rys. 6: Mockup pokazujący koncepcję interfejsu użytkownika w grze

Źródło: Opracowanie własne

2.8. Styl graficzny

Z powodu braku możliwości narysowania wysokiej jakości grafiki 2D do gry (brak grafika, brak zdolności artystycznych, krótki okres czasu na powstanie) zastosowano styl graficzny zwany „Pixel art" (Rys. 7), który sprawdził się w wielu grach niezależnych studiów (Shovel Knight, Crawl, Papers Please, Terraria, Nuclear Throne). Pixel art opiera się na rysowaniu grafiki piksel po pikselu, jednak na potrzeby produkcji gry stosuje się różnorakie programy

przyspieszające jej powstawanie, poprzez automatyczne rysowanie kształtów, wypełnianie kolorami itp. Pozwala to na przygotowanie szaty graficznej na akceptowalnym poziomie, nawet bez większych zdolności artystycznych, co przy jednoosobowym zespole jest dość kluczowe.

Rys. 7: Przykład grafiki w stylu pixel art. Autor: Lucas V. Barbosa

Źródło: Wikimedia Commons

2.9. Sztuczna inteligencja

Jako że gra jest przeznaczona dla jednego gracza, ważnym jest, aby SI w grze było w stanie nawiązać walkę z żywym graczem. Podstawowe SI powinno być w stanie co najmniej wygrać rozgrywkę z graczem, który nie jest specjalnie zaznajomiony z mechanikami gry. To założenie wymusza na SI umiejętność kolonizacji planet, reakcji na inwazje przeciwnika, a także identyfikacji celów do przeprowadzenia inwazji, przygotowania się do niej i jej przeprowadzenia.

Bardziej zaawansowana wersja SI, powinna być w stanie reagować na ruchy sił przeciwnika i przewidywać jego ataki, wiedzieć, kiedy może atakować, a kiedy powinna się bronić, a także identyfikować kluczowe planety i skupiać się na ich obronie lub inwazji. [NIY]

3. Implementacja

Ponieważ proces wytwarzania gry komputerowej poza użyciem środowiska programistycznego i silnika wymaga użycia wielu innych narzędzi (często zaimplementowanych w edytorze dołączonym do silnika), w poszczególnych podrozdziałach dotyczących mechanik opisane będą elementy silnika i edytora wykorzystane w procesie implementacji gry. Ponadto użyty przy implementacji silnik (Unreal Engine 4) wprowadza wiele nabudowań na standard języka C++, co zostanie pokrótce opisane w poniższym rozdziale.

3.1. Unreal Engine 4

Unreal Engine 4 jest jednym z największych silników do implementacji gier wideo. Jego historia sięga roku 1998, kiedy to Epic Games (wtedy Epic MegaGames) wypuściło grę o nazwie „Unreal". Od tego momentu jest on nieprzerwanie rozwijany i na moment pisania niniejszej pracy jego obecna, stabilna wersja ma oznaczenie 4.18. Aktualnie poza wykorzystaniem silnika w produkcji gier wideo, coraz częściej stosuje się go także w przemyśle filmowym (render droida K-2SO w filmie „Łotr 1. Gwiezdne wojny - historie" czy w kooperacji z firmą Chevrolet i Mill krótkometrażowa produkcja „The Human Race" [5] [6]).

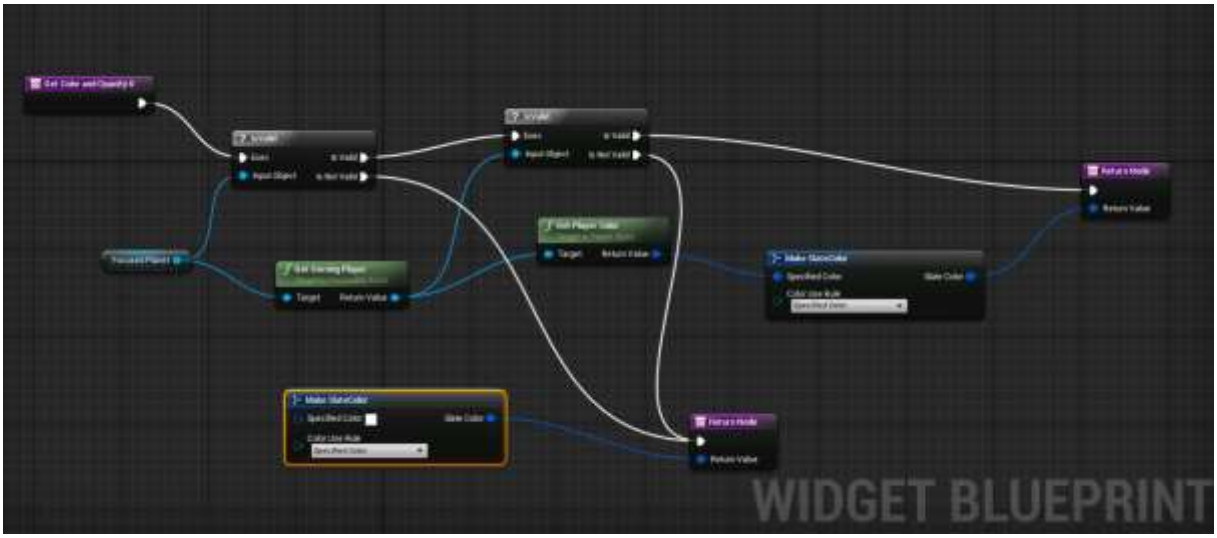
Unreal Engine 4 posiada wiele narzędzi wspomagających powstawanie elementów gry niewymagających pisania kodu (rozkład interfejsu użytkownika, obróbka animacji), czy też pozwalających na implementację logiki w sposób wizualny (drzewa behawioralne, edytor materiałów, blueprinty). Dlatego też w dużej części podpunktów opisujących poszczególne elementy gry zawarty zostanie krótki opis możliwości edytora, który został użyty podczas implementacji danej części aplikacji.

Logika gry w silniku jest implementowana na dwa sposoby:

- Standardowo, za pomocą klas w języku C++
- Wizualnie, za pomocą klas w edytorze silnika przy użyciu języka skryptowego – blueprintów [7] (Rys. 8)

Należy jednak zaznaczyć, że klasy zdefiniowane za pomocą edytora mogą dziedziczyć po klasach napisanych w kodzie. Dlatego też powszechnie stosowaną praktyką przy implementacji logiki w grach za pomocą Unreala jest utworzenie klas obiektów w grze posiadających ogólną logikę, bez dokładnego ustalenia parametrów takich jak zdrowie czy obrażenia za pomocą kodu, a następnie dziedziczenie po tych klasach za pomocą blueprintów (które mogą być edytowane tylko w edytorze) i to w nich dopiero ustalenie używanych tekstur, parametrów dla logiki itp.

Aby edytor był świadomy klas i ich atrybutów zdefiniowanych w kodzie, przed rozpoczęciem kompilacji uruchamiany jest preprocesor parsujący kod w poszukiwaniu specyficznych słów kluczowych (UPROPETY, UFUNCTION, UCLASS itd.) i na ich podstawie generuje dane dla edytora. Równocześnie dane te służą za mechanizm refleksji dla kodu napisanego z wykorzystaniem powyższych słów kluczowych. Unreal Engine 4 posiada także mechanizm automatycznego zarządzania pamięcią (Garbage Collection), jednak tylko dla obiektów i atrybutów rozpoznanych przez preprocesor [8].

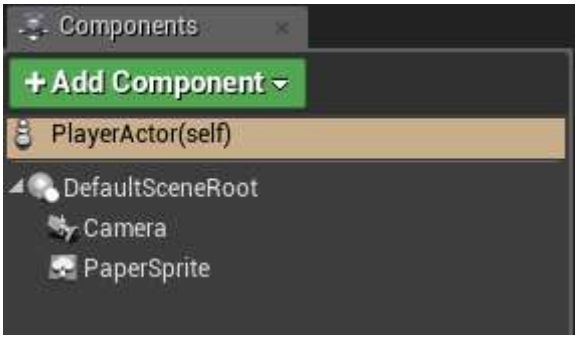


Rys. 8: Implementacja logiki za pomocą w/w wizualnego języka skryptowego

Źródło: Opracowanie własne

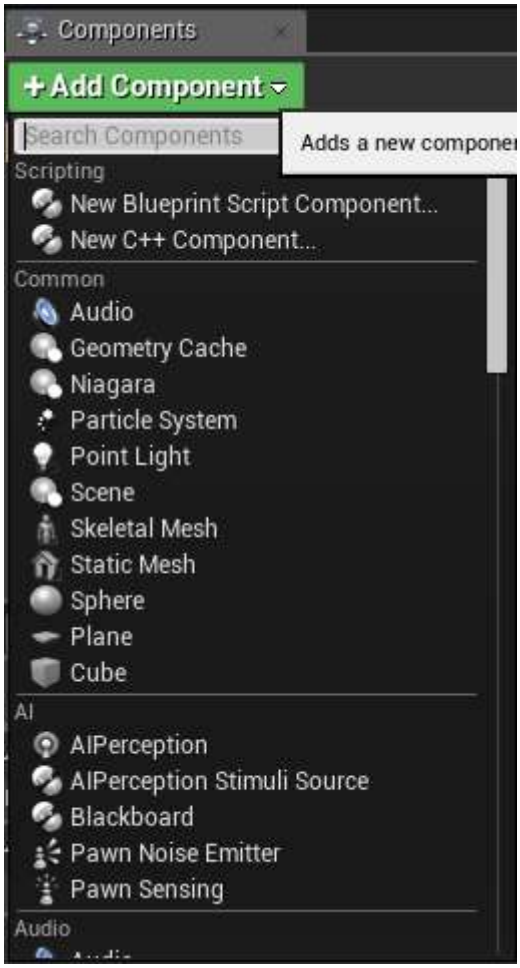
Unreal Engine 4 jest silnikiem opartym na architekturze komponentowej. Oznacza to, że wszystkie klasy dziedziczące po podstawowej klasie aktora na scenie - *AActor* mogą zawierać w sobie komponenty, które są uniwersalnymi kawałkami logiki odpowiedzialnymi na wizualizację, interakcję w świecie czy poruszanie się (Rys. **9**)(Rys. **10**). Oczywiście możliwe jest napisanie własnego komponentu poprzez dziedziczenie po klasie *UActorComponent* lub jednego z już napisanych komponentów w celu rozszerzenia jego funkcjonalności [9]. Klasa *AActor* dziedziczy po klasie *UObject*, która jest najbardziej bazową klasą w silniku i jest odpowiednikiem pustej klasy w normalnej aplikacji. Klasa ta dostarcza szereg funkcjonalności, które silnik wspiera, z automatycznym zarządzaniem pamięcią na czele, możliwość dziedziczenia po nich przez blueprinty w edytorze i wiele innych [10].

UE4 posiada także kilka klas przeznaczonych do zarządzania rozgrywką na poziomach, a także przechowywania danych dla pojedynczego poziomu, czy na przestrzeni całej gry. Każdy poziom posiada swoją klasę dziedziczącą po *AGameModeBase*. Służy ona do implementacji logiki mającej zachodzić na początku takiego poziomu, czy jako kontener dla klas które powinny być ogólnodostępne na poziomie [11]. Dodatkowo na przestrzeni całego czasu działania aplikacji dostępna jest klasa dziedzicząca po *UMainGameInstance* (konkretna klasa ustawiana jest w edytorze dla każdego projektu), używana zazwyczaj do przechowywania opcji ustawianych przez użytkownika, takich jak poziomy dźwięku, jakość grafiki czy schemat sterowania [12].



Rys. 9: Drzewo komponentów jednego z aktorów w grze

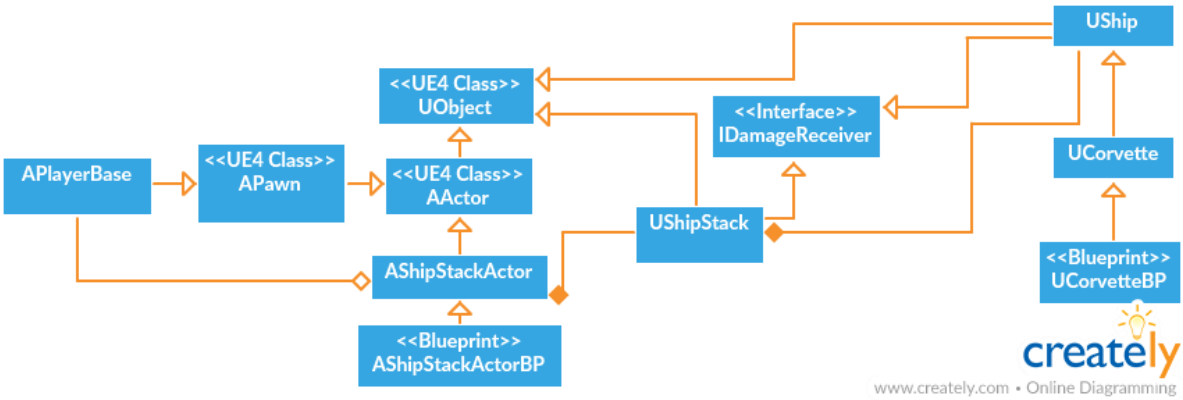
Źródło: Opracowanie własne



Rys. 10: Część listy komponentów możliwych do dodania do aktora

Źródło: Opracowanie własne

3.2. Statki



Rys. 11: Diagram klas przedstawiający część gry dotyczącą statków

Źródło: Opracowanie własne

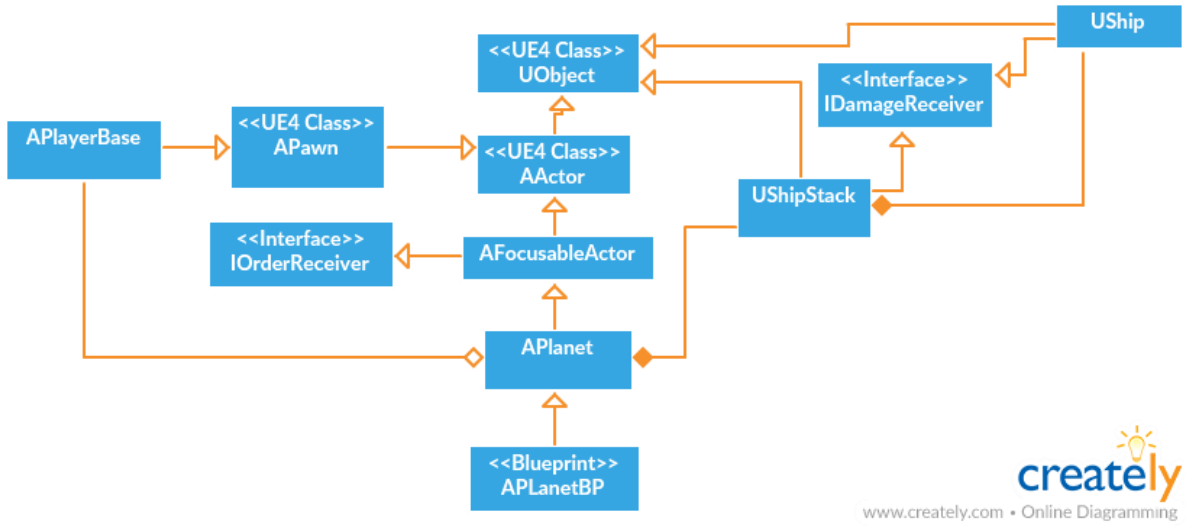
Statki stanowią jeden z podstawowych elementów rozgrywki, jako że to właśnie poprzez nie gracz będzie wpływał na stan świata. Fakt, że każda planeta pod kontrolą gracza produkuje statki, im większa populacja planety, tym szybciej, powoduje, że gracz ma pod swoją kontrolą bardzo duże ich ilości. To, jak i sposób w jaki statki poruszają się po świecie sprawia, że nie mogą one być zwykłymi aktorami, gdyż było by to kłopotliwe ze względów wydajnościowych, jak i gameplay'owych. W związku z tym pojedynczy statek jest reprezentowany poprzez klasę dziedziczącą po najbardziej bazowej klasie silnika (*UObject*) zapewniającej minimalne funkcjonalności opisane w poprzednim rozdziale. Klasa ta - *UShip* posiada wszystkie parametry statku, takie jak zdrowie, zadawane obrażenia czy prędkość przemieszczania się. Na potrzeby niniejszej pracy zaimplementowany został tylko jeden typ statku – korweta, jednak wszystkie elementy ich dotyczące zostały zaimplementowane w sposób, pozwalający dodawać kolejne typy jak najmniejszym nakładem pracy. Ponadto, statki mogą istnieć tylko w pewnych miejscach (planety, floty podróżujące pomiędzy planetami), z tego powodu została utworzona klasa grupy statków *UShipStack*, która gromadzi statki jednego gracza w tym samym miejscu i poza którą statki nie mogą istnieć (Rys. 11).

Jako że statki muszą się przemieszczać, a także potrzebują wizualizacji podczas przemieszczania się, powstała klasa *AShipStackActor*, która opakowuje klasę *UShipStack* dostarczając wizualizację i obsługę logiki podróży między planetami (Rys. 12). Aktor po otrzymaniu rozkazu ruchu na planetę zwraca się do klasy *UPlanetManager* umiejscowionej w klasie dziedziczącej po *AGameModeBase* odpowiedzialną za zarządzanie planetami i połączeniami między nimi, z żądaniem wyznaczenia ścieżki pomiędzy dwoma planetami. Ścieżka wyznaczana jest za pomocą algorytmu przeszukiwania grafu BFS (Breadth First Search). Jego wynikiem jest kolejka planet do odwiedzenia w kolejności wiodących do celu najkrótszą ścieżką z pominięciem planet z wrogimi siłami, jeżeli to możliwe. Flota porusza się od planety do planety zgodnie z dostarczoną kolejką. Dodatkowo każda grupa statków ma przypisanego do siebie gracza, który jest właścicielem danej grupy i tylko rozkazy pochodzące od niego są akceptowane przez grupę.

Rys. 12 : Wizualizacja aktora floty poruszającej się pomiędzy planetami

Źródło: Opracowanie własne

3.3. Planety



Rys. 13: Diagram klas przedstawiający część gry związaną z planetami

Źródło: Opracowanie własne

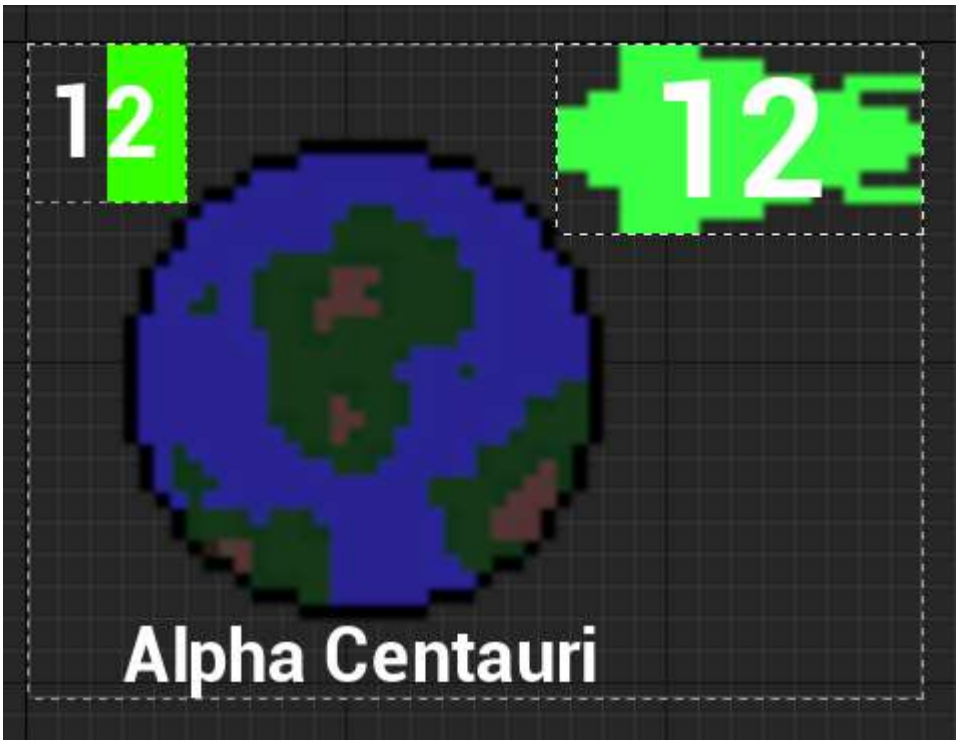
Planety odpowiedzialne są za wiele rzeczy w grze i są jej centralnym punktem, gdyż utrata wszystkich prowadzi do przegranej. Po pierwsze przyrost populacji i produkcja statków. Obie z tych rzeczy zrealizowane są za pomocą akumulacji czasu na funkcji *Tick*, która wywoływana jest przez silnik co klatkę z argumentem oznaczającym czas, jaki upłynął od ostatniego wywołania. Dzięki temu jakiekolwiek mechaniki związane z czasem są łatwe w implementacji. Tu co ilość czasu określoną przez zmienną ustawianą w edytorze następuje zwiększenie się populacji o jeden poziom, jeżeli populacja nie osiągnęła jeszcze maksymalnego pułapu. Jednocześnie budowane są statki, a wielkość populacji na planecie przyspiesza ten proces według następującej formuły:
$BazowyCzasBudowy * (1 + \sqrt{WielkośćPopulacji})$,
gdzie bazowy czas planety także jest wartością ustalaną w edytorze.

Każda planeta na początku gry wytwarza dla siebie po jednej instancji klasy *UShipStack* dla każdego z graczy biorących udział w grze. To tam znajdować się będą statki przebywające na orbicie danej planety. Poza instancjami, które posiada planeta każda z nich ma także kontener na instancje nie należące do planety, a do flot które weszły na orbitę z powodu rozkazu ruchu, lecz nie mogły go kontynuować, gdyż wdały się w walkę. Aby poprawnie wyświetlić siły flot każdego z graczy na danej planecie należy wtedy zsumować siły statków przebywających na orbicie i statków należących do flot podróżujących przez nią.

Jednocześnie planety odpowiedzialne są za rozstrzyganie bitew mających miejsce na orbicie planety. Jeżeli na orbicie znajdą się floty dwóch wrogich sobie graczy planeta w każdej ramce gry wylicza zsumowaną wartość obrażeń floty gracza, dzieli ją przez deltę czasu i dystrybuuje obrażenia pomiędzy wrogich graczy biorących udział w bitwie. Obrażenia w obrębie floty jednego gracza przydzielane są jednemu ze statków gracza, dopóki nie zostanie on zniszczony, a następnie wybierany jest kolejny ze statków. Bitwa trwa tak długo, aż na orbicie znajdą się statki tylko jednego z graczy.

Planety dziedziczą po klasie *AFocusableActor*, co sprawia, że gracz może wybrać ją jako obiekt, z którym chce wejść w interakcję, czy to poprzez standardowe sterowanie, czy

sterowanie głosem. Wybranie planety skutkuje pojawieniem się ekranu ze szczegółami dotyczącymi wybranej planety. Dodatkowo każdy standardowy rozkaz ruchu będzie przyjmował aktualnie wybraną planetę, jako planetę źródłową, co oznacza, że to statki przebywające na jej orbicie zostaną uformowane w flotę i zostanie im przydzielony wydany rozkaz ruchu.



Rys. 14: Reprezentacja planety widziana przez użytkownika (widok w edytorze interfejsu użytkownika)

Źródło: Opracowanie własne

3.4. Konstrukcja poziomów

W aktualnej wersji planety, ich parametry i połączenia między nimi rozmieszczane są na poziomie ręcznie. Planety posiadają zestaw parametrów, które mogą być ustawiane per instancja planety i są widoczne na Rys. 15. „Planet size” oznacza maksymalny możliwy rozmiar populacji na planecie, „Population size” to aktualna populacja planety, należy dodać tu, że nawet neutralne planety mogą być zamieszkane przez populację pewnego rozmiaru, nie będą one produkowały jednak statków, dopóki nie zostaną przejęte przez jednego z graczy. Flaga „Is Starting Planet” używana jest tylko na samym początku gry. Wtedy to każdy z graczy po kolei iteruje po wszystkich planetach na mapie w poszukiwaniu niezajętych jeszcze planet posiadających tę flagę i wybiera pierwszą taką napotkaną planetę, jako swoją planetę startową, po czym ustawia siebie, jako właściciela taj planety. Pozawala to na ustalenie pozycji startowych graczy na mapie na etapie projektowania mapy.

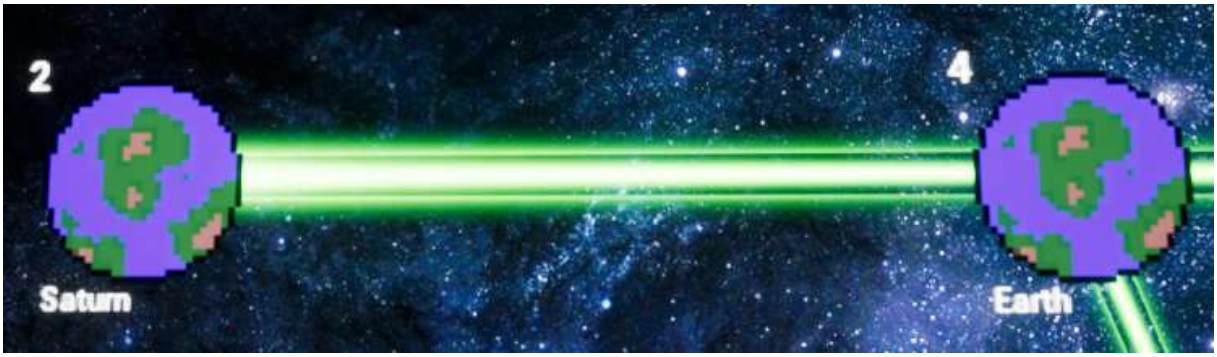
Rys. 15: Zrzut części ekranu edytora ukazujący paramerty planety możliwe do ustawienia

Źródło: Opracowanie własne

Na poziomie istnieje także pojedyncza instancja klasy *UPlanetManager*, która zbiera na początku gry informacje o wszystkich planetach na poziomie, dostarcza planetom zestaw graczy biorących udział w grze, aby mogły wytworzyć sobie odpowiednie struktury, wylicza odległości pomiędzy połączonymi planetami na potrzeby wyszukiwania ścieżek, którą to funkcjonalność także oferuje. Ponadto, jako że planety muszą być jednoznacznie identyfikowalne, gdy sterowanie odbywa się za pomocą głosu, klasa ta parsuje plik definiujący gramatykę używaną do sterowania, który to plik i samo sterowanie zostaną opisane w późniejszej części pracy, i na podstawie tam zdefiniowanych nazw planet przydziela je planetom na poziomie i wytwarza odpowiednią strukturę danych, aby w późniejszym czasie dało się szybko zidentyfikować planetę na podstawie jej nazwy.

Jako że planety muszą być ze sobą połączone, aby umożliwić przemieszczanie się statków, a same połączenia muszą zostać zwizualizowane, została utworzona klasa *APlanetLink*. W niej został dodany komponent odpowiedzialny za renderowanie efektów cząsteczkowych, za pomocą których zostały zrealizowane wizualnie połączenia. Wykorzystany został dostarczany przez silnik system efektów cząsteczkowych Cascade. Oferuje on możliwość wizualizacji efektu cząsteczkowego promienia, który to został tu zastosowany. Promień potrzebuje dwóch parametrów – koordynatów początku i końca efektu, którymi w tym wypadku są aktorzy wizualizujący planety. Aby uprościć i przyspieszyć proces rozmieszczania takich połączeń, w klasie *APlanetLink* został wykorzystany mechanizm oferowany przez silnik - reakcji na zmianę wartości zmiennej w edytorze. W klasie zostały utworzone pola na dwie planety, które mają zostać ze sobą połączone i natychmiast po ustawieniu wartości tych zmiennych, zostają one przepisane do parametrów emitera efektu cząsteczkowego co sprawia, że po ustawieniu tych wartości w edytorze połączenie zmienia swoją wizualizację, aby dostasować się do nowych położeń początku i końca promienia. Pozwala to na łatwe zaplanowanie rozkładu poziomu, gdyż i planety, i połączenia między nimi są wizualizowane już na etapie jego kreacji.

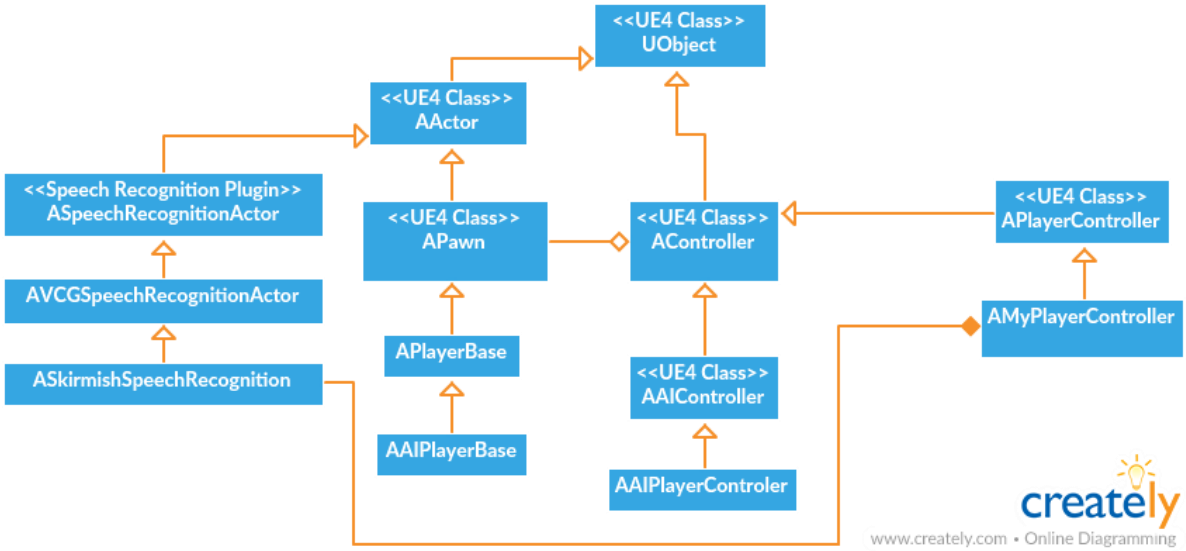
Dodatkowo podczas startu poziomu *UPlanetManager* iteruje po wszystkich aktorach wizualizujących połączenia, odczytując pary planet i budując na ich podstawie wewnętrzną reprezentację grafu planet i połączeń między nimi, co oznacza, że aby utworzyć połączenie między planetami, które w grze jest wizualizowane, a także istnieje i wpływa na logikę gry, jedyne co należy zrobić to umieścić aktora *APlanetLink* na poziomie i wybrać dwóch aktorów planet które mają zostać połączone. Cała reszta odbywa się już automatycznie.



Rys. 16: Połączenie pomiędzy planetami widziane w grze

Źródło: Opracowanie własne

3.5. Gracze



Rys. 17: Diagram klas przedstawiający część gry dotyczącą graczy

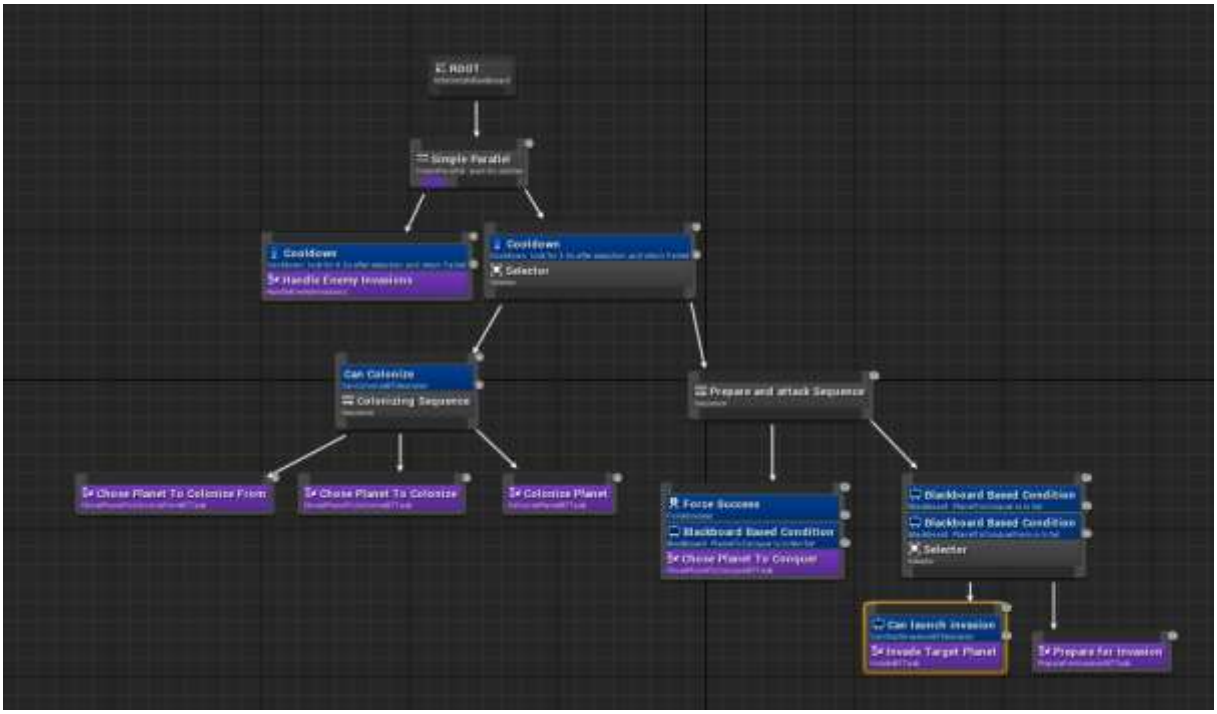
Źródło: Opracowanie własne

Unreal Engine 4 implementuje koncepcję gracza w następujący sposób. Sam gracz reprezentowany jest przez kontroler (klasę dziedziczącą z klasy *AController*), jednak on sam nie ma swojej reprezentacji w świecie. Kontroler gracza może przejąć kontrolę nad szczególnym typem aktorów – pionków, które mogą wpływać na świat (klasy dziedziczące po klasie *APawn*). W przypadku gracza ludzkiego jego działania (wciskanie przycisków na klawiaturze/myszce/kontrolerze) są mapowane na wywołania odpowiednich funkcji, czy to w kontrolerze, jeżeli są niezmienne, czy w pionku, jeżeli gracz może przejmować kontrolę nad wieloma różnymi pionkami, których schematy sterowania czy możliwość wykonywanych akcji różnią się. W przypadku gracza sterowanego przez SI jego logika wywołuje już odpowiednie metody kontrolowanego pionka, nie trzeba więc żadnego mapowania. Dla przeciwników sterowanych przez komputer potrzebujących złożonej logiki działania Unreal Engine zaimplementował mechanizm drzew behawioralnych. Drzewo behawioralne to, jak nazwa wskazuje struktura drzewiasta, której poszczególne węzły są elementami logiki. Węzły dzielą się na trzy podstawowe typy: węzły – selektory, węzły sekwencji i węzły – zadania. Pierwsze dwa typy nie mogą być liśćmi, a ostatni typ może występować tylko jako liść. Zadania są jedynym elementem odpowiedzialnym za wykonywanie konkretnych akcji przez

SI, tam znajduje się implementacja logiki działań komputerowego gracza. Selektory i sekwencje grupują pod sobą kolejne węzły i ich zadaniem jest odpowiednie sterowane przepływem w drzewie. Drzewo behawioralne może wykonywać w danej chwili tylko jeden węzeł. Węzły drzewa podczas wykonywania mogą zwrócić jeden z poniższych stanów:

- Succeeded – oznacza, że węzeł zakończył swoje działanie i wykonanie jego logiki przebiegło pomyślnie
- Failed – także sygnalizuje zakończeniu zadania, ale informuje o porażce w wykonywaniu logiki lub niemożności jej wykonania
- Aborted – syganlizuje, że zadnie zostało zakończone przedwcześnie
- InProgress – wskazuje, że zadanie wciąż jest wykonywane i nie jest jeszcze znany wynik jego wykonania

Drzewo zawsze zaczyna się korzeniem, od którego rozpoczyna się wykonywanie logiki. Korzeń rozpoczyna wykonywanie węzła pod sobą, który z zasady jest selektorem lub sekwencją. Selektor ma za zadnie wykonywać węzły zgrupowane pod nim w kolejności tak długo, dopóki jeden z węzłów nie zwróci w wyniku swojego działania stanu *Succeeded,* w takim wypadku selektor kończy swoje wykonywanie i także zwraca stan *Succeeded,* w prezciwnym wypadku zwraca *Failed*. Sekwencja także wykonuje węzły pod sobą jednak wykonuje je w kolejności tak długo, jak długo wykonanie każdego węzła zwraca stan *Succeeded.* Jeżeli wszystkie węzły wykonają się pomyślnie sekwencja zwraca *Succedeed* w przeciwnym wypadku *Failed*. Stan *InProgress* w drzewie behawioralnym pozwala na implementację zadań wymagających upłynięcia pewnego czasu. Gdy zadanie zwraca ten stan drzewo wstrzymuje swoje wykonanie, dopóki z węzła nie zostanie zwrócony jeden z trzech pozostałych stanów. Dodatkowo do drzewa dołączana jest struktura danych współdzielona pomiędzy nim a kontrolerem, który jest właścicielem logiki reprezentowanej przez drzewo behawioralne. Struktura ta jest nazywana tablicą (ang. blackboard) i dostęp do niej realizowany jest poprzez klucze odnoszące się do poszczególnych zmiennych w niej [13] [14].

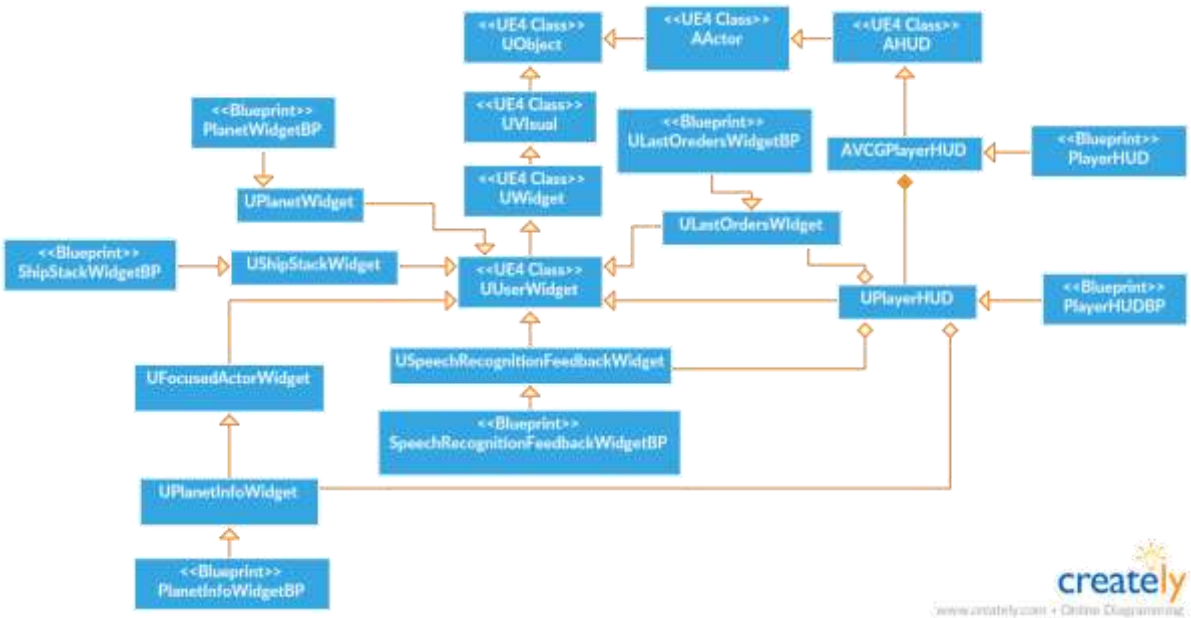


Rys. 18: Zrzut ekranu edytora przedstawiający drzewo behawioralne implementujące logikę przeciwnika komputerowego w grze (fioletowe węzły to zadania, szare to selektory i sekwencje, niebieskie obszary nie są węzłami, to dekoratory - dodatkowa logika związana z węzłem)
Źródło: Opracowanie własne

Gracze dzielą się na dwa typy – sterowani przez człowieka i sztuczną inteligencję. Gracz ludzki w swoim kontrolerze zaimplementowaną ma konfigurację standardowego sterowania (włączenie obsługi myszki, konfiguracja dostępnych klawiszy), a także inicjalizację zewnętrznej biblioteki odpowiedzialnej za sterowanie głosem, które zostanie opisane w późniejszej części tej pracy i obsługa zdarzeń przychodzących z niej za pomocą klasy *ASkirmishSpeechRecognition*, do której kontroler rejestruje odpowiednie metody pionka odpowiedzialne, za zaznaczanie planet czy poruszanie jednostkami. Klasa ta parsuje tekst przychodzący z systemu rozpoznawania mowy i na jego podstawie, jeżeli tekst rozkazu jest poprawny wywołuje konkretne zarejestrowane metody z odpowiednimi parametrami poprzez mechanizm delegatów (w skrócie jest to implementacja wskaźników na funkcje znajdujące się w klasach (ang. member functions) z dodatkowymi funkcjonalnościami) [15].

Gracze sterowani przez SI są reprezentowani, przez klasę *AAIPlayerComponent*, która to posiada zmienną przechowującą wskaźnik na klasę drzewa behawioralnego które ma stanowić logikę gracz. Powoduje to, że zaimplementowanie SI które zachowuje się w inny sposób, jest agresywniejsze, bardziej inteligentne, jest jedynie kwestią utworzenia kolejnego drzewa behawioralnego i ustawienia go w kontrolerze. Dodatkowo została utworzona klasa *AAIPlayerBase* dziedzicząca po klasie pionka gracza *APlayerBase*, ponieważ SI potrzebuje dodatkowych informacji o świecie gry, które gracz sam zbiera np. lista neutralnych planet możliwych do przejęcia itp.

3.6. Interfejs użytkownika



Rys. 19: Diagram klas przedstawiający część gry dotyczący interfejsu użytkownika

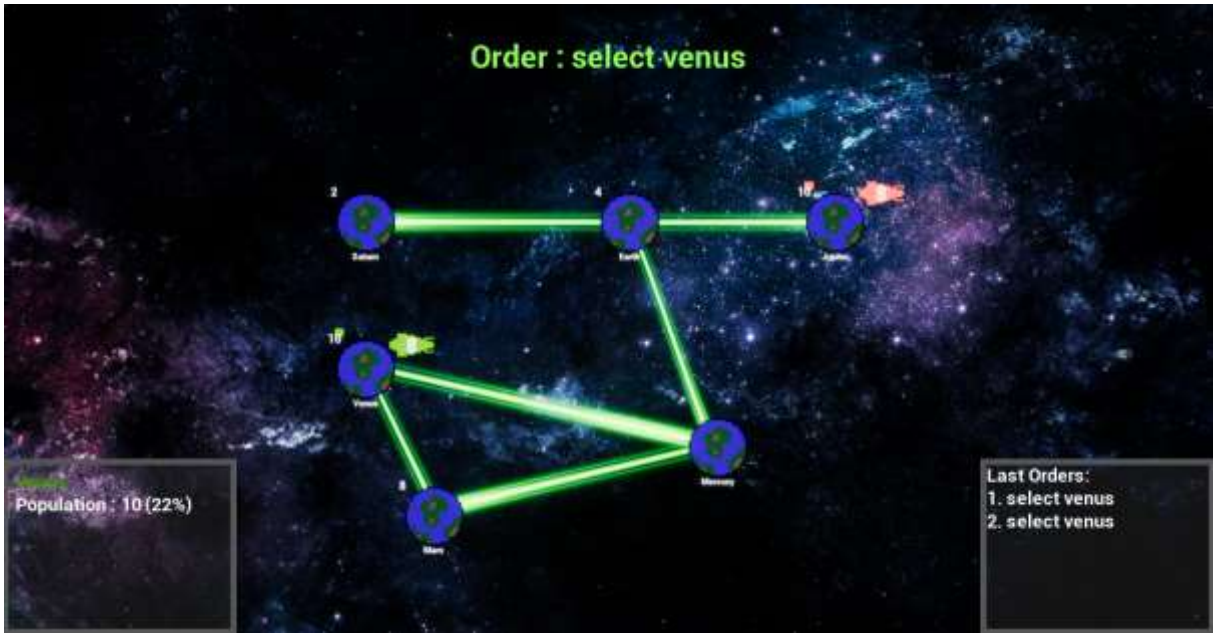
Źródło: Opracowanie własne

Jak zostało opisane w części projektowej, interfejs użytkownika powinien być jak najbardziej minimalistyczny. Początkowo jedynym planowanym elementem interfejsu użytkownika był panel zawierający szczegółowe informacje dotyczące wybranej planety. Jednak wprowadzenie sterowania głosem pokazało, że pewne rozkazy powtarzają się bardzo często. Logicznym usprawnieniem było wprowadzenie mechaniki zapamiętywania ostatnio wydanych rozkazów i możliwości wywołania ich za pomocą zdecydowanie krótszych fraz. Aby użytkownik wiedział w danej chwili jaki rozkaz możliwy jest do wywołania poprzez frazy dotyczące ostatnich rozkazów potrzebny był panel wyświetlający ostatnio wydane rozkazy. Analogicznie podczas użytkowania sterowania za pomocą głosu wyszła na jaw potrzeba komunikowania graczowi poprawnego sparsowania i wykonania rozkazu. Pozwala to w łatwy sposób zakomunikować użytkownikowi czy fraza, którą wypowiedział została poprawnie zrozumiana i czy w wyniku jej wypowiedzenia został wykonany rozkaz. Panel wyświetlający potwierdzenia przyjęcia rozkazu pozwala także identyfikować przypadki, gdy biblioteka rozpoznawania głosu niepoprawnie identyfikuje wejście mikrofonu, które zwykle jest szumem, jako rozkaz (tzw „false positives").

Każdy z elementów interfejsu wykonany został za pomocą widgetów UMG (Unreal Motion Graphics). Widgety takie implementuje się w edytorze z klasy *UUserWidget* i klas po niej dziedziczących. Pojedynczy panel interfejsu składany jest z dostraczonych przez silnik elementów napisanych w frameworku Slate [16] lub z innych wcześniej przygotowanych paneli interfejsu. Aby zredukować do minimum ilość logiki, która musi zostać zaimplementowana w edytorze, aby każdy z elementów interfejsu pobrał potrzebne mu wartości, każdy element interfejsu utworzony jest w oparciu o własną klasę, która dostarcza zestaw metod ułatwiających pobieranie potrzebnych danych. W blueprintach widgetów użyty został mechanizm wiązań (ang. „bindings"). Polega on na tym, że gdy dany element interfejsu potrzebuje informacji np. tekstu czy tekstury, wywołuje on odpowiednią metodę, która jest odpowiedzialna za pobranie tych danych. Dzięki takiemu rozwiązaniu duża część logiki wcześniej odpowiedzialna, za sprawdzanie czy wartość uległa zmianie i przepisywania jej do

elementu docelowego przestaje być potrzebna, a wpływ tego mechanizmu na wydajność jest praktycznie niezauważalny.

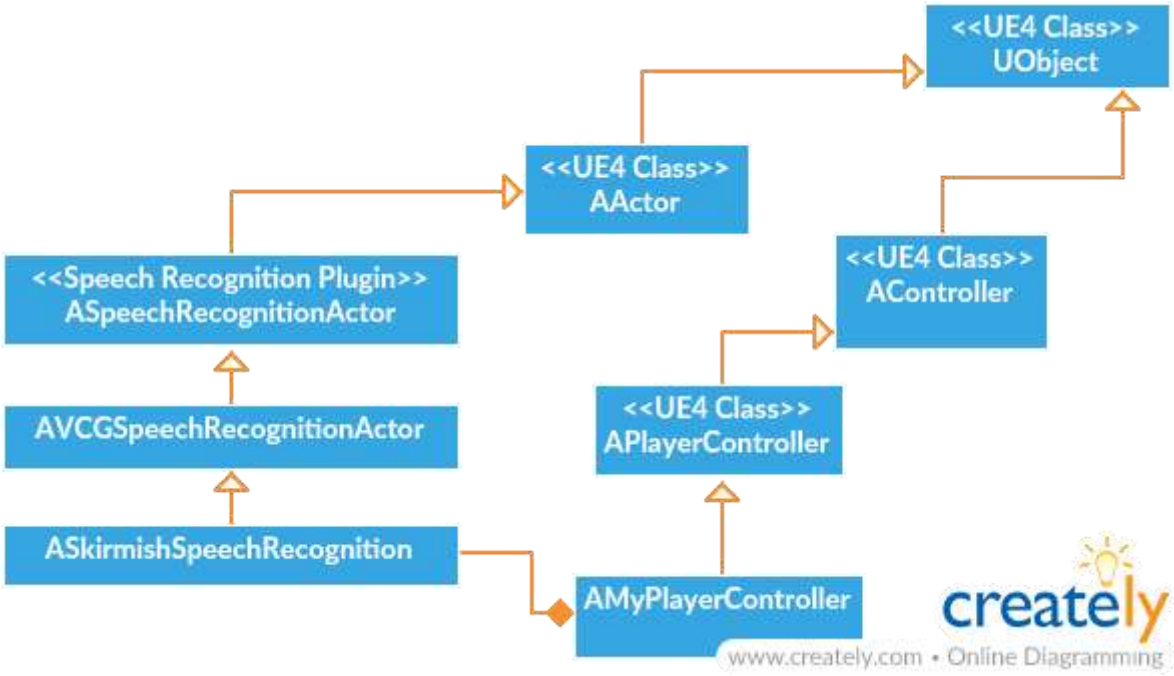
Poniżej (Rys. 20) przedstawiony jest efekt implementacji interfejsu użytkownika. W prawym dolnym rogu widać panel przekazujący graczowi szczegóły dotyczące wybranej planety (został powiększony względem projektu, aby pomieścić dodatkowe atrybuty planety, które jednak nie zostały zaimplementowane do momentu pisania tej pracy, z powodu zbyt małej ilości czasu), na środku u góry widnieje tekst potwierdzający przyjęcie polecenia głosowego, zaś w prawym dolnym rogu ekranu jest panel pokazujący użytkownikowi ostatnio wydane rozkazy, które może ponownie wykonać wydając krótszy rozkaz, co zostanie szerzej opisane w następnym rozdziale.



Rys. 20: Zrzut ekranu z opisywanej gry, przedstawiający wszystkie opisywane elementy interfejsu

Źródło: opracowanie własne

3.7. Sterowanie głosem



Rys. 21: Diagram klas przedstawiający część gry odpowiedzialnej za sterowanie głosem

Źródło: Opracowanie własne

Sterowanie głosem w grze odbywa się za pomocą zewnętrznej biblioteki PocketSphinx przystosowanej do użycia w silniku za pomocą wtyczki Speech Recognition Plugin. Dostarcza ona klasę A*SpeechRecognitionActor*, po której dziedziczą klasy zajmujące się przetwarzaniem komend głosowych na potrzeby opisywanej gry. Używana biblioteka potrzebuje ustalenia pewnych parametrów zanim będzie w stanie działać. Jest to lokalizacja danych modelu potrzebnych do rozpoznawania mowy, a także tryb działania biblioteki, gdyż udostępnia ona trzy sposoby wydawania komend. Pierwszy z nich to sztywne zdefiniowanie komend uznawanych za poprawne (tzw. „phrase mode"), wtedy tylko i wyłącznie komendy w pełni zgadzające się z jedną z wcześniej zdefiniowanych zostaną rozpoznane. Drugi to tryb gramatyki (tzw. „grammar mode"), definiowana jest ona za pomocą języka JSGF (JSpeech Grammar Format [17]), osobno dla każdego języka naturalnego. W tym trybie każda komenda, która jest podzbiorem zdefiniowanej gramatyki zostaje rozpoznana jako poprawna. Oznacza to że jeżeli pełną poprawną komendą jest „choose Venus", to poprawnie rozpoznane zostaną także same słowa „choose" i „Venus". Ostatnim trybem działania biblioteki jest tryb językowy (tzw. „language mode"), który uznaje wszelkie słowa które występują w dostarczonym modelu za poprawne.

Sterowanie głosem w opisywanej grze zrealizowane zostało za pomocą opisywanego wyżej trybu gramatyki. Klasa A*SpeechRecognitionActor*, która jest klasą dostarczoną przez plugin jako klasa przeznaczona do komunikacji z biblioteką, posiada zdefiniowanych delegatów wykonywanych w momencie rozpoznania lub nierozpoznania komendy, jak i w momencie rozpoczęcia czy zakończenia jej wydawania, jednak aktualnie w grze jedynie pierwszy z nich jest wykorzystywany. Z powodu uznawania rozkazów częściowo kompletnych za poprawne, w klasie odpowiedzialnej za przetwarzanie komend następuje sprawdzenie, czy podana fraza jest jednym z rozkazów i zawiera wszystkie potrzebne informacje. Proces ten jest wykonywany za pomocą wyrażeń regularnych. Pozwalają one nie tylko zweryfikować

poprawność frazy jako rozkazu, a także za pomocą tzw. „capture groups” wyekstrahowanie zmiennych części rozkazu takich jak nazwa planety czy liczba porządkowa wcześniej wydanego rozkazu. Po rozpoznaniu poprawnego rozkazu i ustaleniu jego parametrów wykonywany jest odpowiedni delegat, do którego kontroler gracza rejestruje właściwe metody odpowiedzialne za faktyczne wykonanie rozkazów. I tak na przykład klasa *ASkirmishSpeechRecognition* posiada zestaw delegatów wywoływanych w momencie przyjęcia rozkazu wyboru planety czy ruchu.

Jak już wcześniej zostało wspomniane, gramatyka wykorzystywana w procesie rozpoznawania mowy jest definiowana w osobnym pliku za pomocą języka JSGF. Nie wdając się w szczegóły języka, gramatyka budowana jest za pomocą zasad (ang. rules). Zasady mogą składać się z innych zasad, czy też stałych słów. Finalna gramatyka używana w procesie rozpoznawania mowy, jest jedną zasadą, w której zwykle znajdują się złożenia pozostałych, wcześniej zdefiniowanych zasad. I tak nazwy planet zostały określone w jednej zasadzie (która to zasada jest potem parsowana, aby wydobyć wszystkie dostępne nazwy, co zostało opisane w podrozdziale dotyczącym planet), a każdy dotyczący planet rozkaz stanowi jedną zasadę, które na samym końcu składane są w finalną zasadę stanowiącą gramatykę używaną podczas gry.

Na moment pisania pracy dostępne są cztery rozkazy głosowe w języku angielskim:

- **select <NazwaPlanety>** - Wybiera planetę o nazwie <NazwaPlanety>, o ile taka istnieje w grze
- **move to <NazwaPlanety>** - Nakazuje statkom kontrolowanym przez gracza na orbicie wybranej planety ruch na planetę <NazwaPlanety>, o ile jakakolwiek planeta jest wybrana i wskazana w rozkazie planeta istnieje w grze
- **move from <NazwaPlanety> to <NazwaPlanety2>** - Nakazuje statkom kontrolowanym przez gracza stacjonującym na orbicie planety <NazwaPlanety> (o ile taka istnieje) ruch na planetę <NazwaPlanety2>, o ile taka istnieje
- **execute <NumerRozkazu>** - Nakazuje powtórzenie n-tego ostatnio wykonanego rozkazu, gdzie <NumerRozkazu> jest liczbą pomiędzy 1 a 6

Po zaimplementowaniu sterowania głosem potrzebne było dostosowanie pewnych parametrów w bibliotece, ponieważ okazało się, że biblioteka przestaje nasłuchiwać po zbyt krótkim okresie ciszy, a także ma zbyt wielką tolerancję dla wypowiadanych komend, tzn. potrafiła ona wykryć wydanie rozkazu z dźwięku szumu w mikrofonie. Tak więc należało podczas inicjalizacji biblioteki nadpisać domyślne parametry, aby rozpoznawanie głosu było bardziej restrykcyjne, ale także tolerancyjne dla komend wypowiadanych wolniej, czy szybciej niż normalnie. Na potrzeby pisania niniejszej pracy zaimplementowana została obsługa komend wydawanych w języku angielskim, i to dla tego języka zostały dobrane odpowiednie parametry. Języki, które diametralnie się od niego różnią, np. język polski, który jest językiem silnie fleksyjnym będą wymagały ponownego dobrania poszczególnych ustawień biblioteki.

3.8. Perspektywy rozwoju

Opisywana aplikacja w momencie pisania pracy jest aplikacją w pełni funkcjonalną, jednak widać w niej wiele miejsc, które mogłyby zostać rozbudowane czy zmienione, aby jakość rozgrywki była wyższa, czy same mechaniki bardziej rozbudowane. Poniżej znajduje się lista pomysłów, które zaimplementowane, rozwinęłyby grę:

- W grze powinny zostać dodatkowe typy planet, na potrzeby pisania pracy został utworzony jeden, bazowy typ, jednak na więcej zabrakło czasu. Samo dodanie nowego typu planety nie zajmuje wiele czasy, gdyż planety zostały zaimplementowane z myślą o ich różnorodnych typach, jednak dodatkowe typy planet wymagałyby czasu poświęconego na testowanie wybranych parametrów typu planety, aby nie burzyły aktualnego stanu gry, a jedynie go rozbudowywały
- Dodatkowe typy statków znacznie pogłębiłyby rozgrywkę wprowadzając problem kompozycji floty i reakcji, na kompozycję którą stosuje przeciwnik. Tak jak wyżej, logika planet została napisana z myślą o obsłudze produkcji różnych typów statków, jednak sam interfejs, sterowanie i SI nie są do tego przystosowane, a to wymagałoby już dość dużych nakładów pracy
- Najczęściej używane komendy głosowe mogłyby być podpisywane pod skróty, które byłyby o wiele szybsze w wydawaniu dodatkowo, aby użytkownik wiedział jaka komenda została przypisana pod jaki skrót należy zaprojektować i zaimplementować odpowiedni panel interfejsu.
- Dodatkowym elementem gry mógłby być system automatycznego generowania map, z uwzględnieniem ilości graczy, rozkładający planety zapewniając relatywnie symetryczną mapę dla każdego z graczy
- Ilość w pełni obsługiwanych graczy powinna zostać zwiększona z dwóch do dowolnej liczby, jednak to poniosłoby za sobą potrzebę zaimplementowania w/w systemu generacji map, aby konstrukcja mapy pasowała do ilości graczy, nie faworyzując żadnego z nich
- SI w grze powinno zostać wzbogacone o kilka mechanizmów pozwalających lepiej reagować na poczynania gracza, np. identyfikacja i gromadzenie sił w strategicznych miejscach mapy, takich jak planety będące jedyną drogą do innych części mapy.
- Po zaimplementowaniu chociaż części mechanik wspomnianych wcześniej ciekawym kierunkiem rozwoju gry byłaby kampania dla pojedynczego gracza, składająca się z szeregu scenariuszy, o rosnącym poziomie trudności
- Ostatnim kierunkiem rozwoju dla gry z racji, że aktualnie gra opiera się na modelu symetrycznej potyczki jest oczywiście wsparcie dla wielu graczy ludzkich za pośrednictwem sieci lokalnej, czy Internetu

4. Podsumowanie

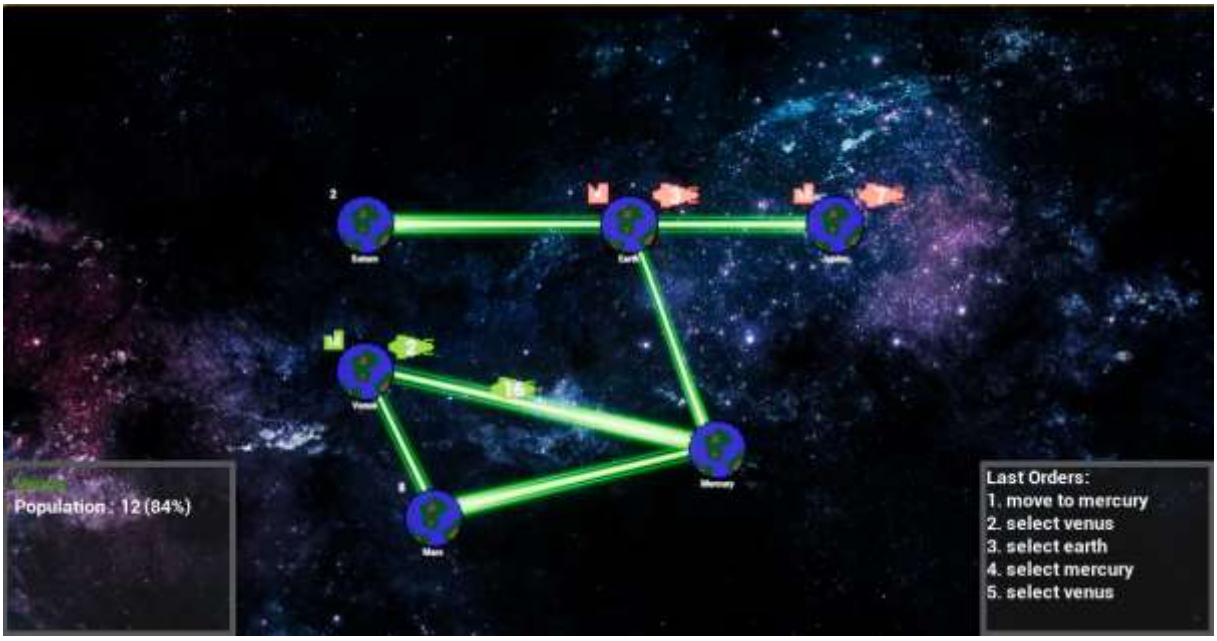
Przestawiona powyżej implementacja i projekt gry nie powinny być pod żadnym pozorem widziane jako skończony produkt. O ile po opisanej implementacji gra jest w pełni grywalna, to jednak jest to dopiero punkt wyjściowy do wytworzenia produktu atrakcyjnego dla konsumenta. Jest to swoista wersja alfa gry, gdzie bazowe mechaniki są na miejscu, jednak wymagają one rozbudowania, atrybuty poszczególnych jednostek nie są odpowiednio dobrane, brakuje dodatkowych typów jednostek itd.

Opisywana gra jest tzw. „proof of concept” (z ang. “dowód słuszności koncepcji”) pokazuje ona, że gra w całości sterowana głosem na komputery osobiste jest możliwa. Ten sposób sterowania rozgrywką ma swoje problemy i jest słabo zbadany przez twórców, co dodatkowo utrudnia projektowanie takiej aplikacji. Powyższa gra była bardzo ciekawym problemem, szczególnie na etapie projektowania ze względu właśnie na nietypowy jak na grę sposób sterowania, który wymuszał wiele niepopularnych rozwiązań.

W pracy dokonano przeglądu komercyjnych gier korzystających ze sterowania głosem pod kątem możliwości które daje graczowi sterowanie głosem, zaprezentowano aktualny, na

moment pisania pracy stan GDD, a także opisano sposób implementacji każdej z głównych mechanik gry, w szczególności sterowanie głosem.

Rynek gier jest niezwykle prężną gałęzią rozrywki, a należy pamiętać, że już osiągnął on imponujące rozmiary. Aktualnie w świadomości społecznej gry powoli przestają być postrzegane jako rozrywka „dla dzieci", a coraz częściej są stawiane na równi z pozostałymi gałęziami rozrywki, takimi jak telewizja czy filmy. To sprawia, że coraz więcej ludzi gra w gry wideo, tak więc gra z niekonwencjonalnym sposobem sterowania, ma bardzo dużą szansę znaleźć swoją niszę przy aktualnym stanie rynku.



Rys. 22: Zrzut ekranu z opisywanej gry

Źródło: opracowanie własne

Bibliografia

[1] T. T. G. Jr. i E. R. Mann, „Patent USA nr 2,455,992”. USA Patent 2455992, 14 12 1948.

[2] M. J. P. Wolf, „The Video Game Explosion: A History from PONG to Playstation and Beyond,” w *The Video Game Explosion: A History from PONG to Playstation and Beyond*, Westport, Connecticut, Greenwood Press, 2008, pp. 104-106.

[3] R. Dvorchak, "NEC out to dazzle Nintendo fans," *The Times News*, pp. 1D, 8D, 30 Lipiec 1989.

[4] H. Yasuhara, „How to make a game "fun",” Kraków, 2017.

[5] J. Alexander, 1 Marzec 2017. [Online]. Available: http://www.develop-online.net/news/some-rogue-one-scenes-shot-using-unreal-engine-4/0230147. [Data uzyskania dostępu: 07 Listopad 2017].

[6] „YouTube,” 1 Marzec 2017. [Online]. Available: https://www.youtube.com/watch?v=MQ3QuZtrxl0. [Data uzyskania dostępu: 7 Listopad 2017].

[7] N. Valcasara, Unreal Engine Game Development Blueprints, Packt Publishing, 2015.

[8] Epic Games, „Unreal Architecture | Unreal Engine,” [Online]. Available: https://docs.unrealengine.com/latest/INT/Programming/UnrealArchitecture/Reference/index.html. [Data uzyskania dostępu: 7 Listopad 2017].

[9] Epic Games, „Actors | Unreal Engine,” [Online]. Available: https://docs.unrealengine.com/latest/INT/Programming/UnrealArchitecture/Actors/. [Data uzyskania dostępu: 12 Listopad 2017].

[10] Epic Games, „Objects | Unreal Engine,” [Online]. Available: https://docs.unrealengine.com/latest/INT/Programming/UnrealArchitecture/Objects/. [Data uzyskania dostępu: 12 Listopad 2017].

[11] Epic Games, „Game Mode and Game State | Unreal Engine,” [Online]. Available: https://docs.unrealengine.com/latest/INT/Gameplay/Framework/GameMode/. [Data uzyskania dostępu: 2017 Listopad 12].

[12] Epic Games, „Game Instance, Custom Game Instance For Inter-Level Persistent Data Storage - Epic Wiki,” [Online]. Available: https://wiki.unrealengine.com/Game_Instance,_Custom_Game_Instance_For_Inter-Level_Persistent_Data_Storage. [Data uzyskania dostępu: 2017 Listopad 12].

[13] Epic Games, „How Unreal Engine 4 Behavior Trees Differ | Unreal Engine,” [Online]. Available: https://docs.unrealengine.com/latest/INT/Engine/AI/BehaviorTrees/HowUE4BehaviorTreesDiffer/index.html. [Data uzyskania dostępu: 13 Listopad 2017].

[14] P. L. Newton i J. Feng, Unreal Engine 4 AI Programming Essentials, Packt Publishing, 2016.

[15] Epic Games, „Delegates | Unreal Engine,” [Online]. Available: https://docs.unrealengine.com/latest/INT/Programming/UnrealArchitecture/Delegates/. [Data uzyskania dostępu: 13 Listopad 2017].

[16] Epic Games, „Slate Overview | Unreal Engine,” [Online]. Available: https://docs.unrealengine.com/latest/INT/Programming/Slate/Overview/index.html. [Data uzyskania dostępu: 2017 Listopad 15].

[17] Sun Microsystems, „JSpeech Grammar Format,” 2000. [Online]. Available: https://www.w3.org/TR/2000/NOTE-jsgf-20000605/. [Data uzyskania dostępu: 25 Listopad 2017].

Spis ilustracji