



# PAŃSTWOWA WYŻSZA SZKOŁA ZAWODOWA W NYSIE

Kierunek: Informatyka

Specjalność: Systemy internetowe

Praca dyplomowa inżynierska

**Last Adventure - Gra komputerowa 2D na silniku Unity 3D**

Promotor:

**dr inż. Adam Sudół**

Autor:

**Aleksander Sinkowski**

Nysa 2019

## **Spis treści**

<b>1. Wstęp .....</b>	3
1.1 Geneza projektu .....	3
1.2 Zakres pracy oraz założenia .....	3
1.3 Przegląd materiałów źródłowych.....	5
1.4 O licencji. ....	6
<b>2. Narzędzia.....</b>	7
2.1 Unity 3D.....	7
2.2 Piskel, Gimp, Pixel Art i animacje poklatkowe. ....	14
2.3 BoscaCeoil. ....	19
2.4 AudaCity. ....	22
<b>3. Plan działania.....</b>	23
3.1 Krok 1: Nowa scena oraz sterowanie.....	23
3.2 Krok 2: Animacje.....	26
3.3 Krok 3: Zapis, odczyt i obiekt „Config”.....	27
3.4 Krok 4: Menu główne & UI.....	28
3.5 Krok 5: Design, światło, cienie, kamera i portale.....	29
3.6 Krok 6: Fabuła, wizja gry.....	32
3.7 Krok 7: Non-playable character (NPC).....	34
3.8 Krok 8: System dialogów i quest’ów.....	36
3.9 Krok 9: Ekwipunek oraz przedmioty. ....	36
3.10 Krok 10: System walki, przeciwnicy i bossowie. ....	38
3.11 Krok 11: Napisy końcowe.....	39
<b>4. Z teorii do praktyki .....</b>	40
4.1 Pierwsza scena, pierwsza postać, pierwszy skrypt (krok 1).....	40
4.2 Poklatkowa animacja w animatorze (krok 2).....	48
4.3 Zapis, odczyt i obiekt „Config” (krok 3). ....	57
4.4 Podstawy UI, menu główne i konfiguracja początkowa (krok 4).....	61
4.5 Projekt świata, iluzja światła, ruch kamery i portale (krok 5). ....	71
4.6 NPC, interakcja, dialogi i zadania (krok 7 - 8). ....	76
4.7 Ekwipunek, przedmioty i sterowanie 2.0 (krok 9). ....	81
4.8 System walki, przeciwnicy i bossowie (krok 10).....	92
4.9 Fin, czyli scena kończąca (krok 11).....	99
<b>5. Podsumowanie .....</b>	100
5.1 Produkt końcowy i porzucone pomysły.....	100
5.2 Test wydajności oraz perspektywa rozwoju. ....	102
5.3 Instrukcja oraz streszczenie.....	103
<b>Lista grafik .....</b>	107
<b>Bibliografia .....</b>	108

## **1. Wstęp**

### **1.1 Geneza projektu**

Maszyny będące komputerami, służące przede wszystkim do obliczeń, z biegiem czasu stały się mniejsze oraz bardziej przystępne zarówno dla organizacji jak i dla zwykłych ludzi. Zaczęto rozszerzać funkcjonalności sprzętów tworząc m.in. maszyny takie jak konsole czy automaty, które bardzo pozytywnie przyjęły się jako nowa forma rozrywki. Rozwój komputerów osobistych przeniósł ową zabawę na wyższy poziom zalewając rynek niezliczoną ilością gier tekstowych od przeróżnych nowopowstałych firm developerskich. Wprowadzanie interfejsów graficznych pozwoliło na kolejną ewolucję branży gier oraz stworzyło wyścig graficzny, który wciąż trwa. Z biegiem lat założono wiele nowych organizacji, wiele również upadło, a nieliczni weterani stali się ogromnymi firmami i mimo potknieć trzymają się twardo po dziś dzień. Cała ta ewolucja jaką przeszedł rynek gier komputerowych sprawiła, iż powstało mnóstwo sposobów oraz opisów dających osobą niedoświadczonym szansę na stworzenie własnego produktu i to właśnie tego dotyczy ta praca. Obecny projekt to podjęcie wyzwania, którym jest przejście drogi twórców gier, by finalnie powstał produkt dający omawianą formę rozrywki.

Z powodu braku doświadczenia, projekt nie zakłada stworzenia ogromnego, zróżnicowanego środowiska mającego niezliczoną ilość aktywności, gdyż byłoby to zbyt dużym wyzwaniem. Podjęto jednak stworzenie gry przygodowej o zróżnicowanych, choć niewielkich lokacjach, z elementami zręcznościowymi, walką dystansową oraz linią fabularną. Kolejne strony zawierają opis działań, doświadczeń, przemyśleń oraz rozwiązań, które nie zawsze okażą się najlepsze mimo, iż działają bez problemu. Zaś z uwagi na obecny pęd technologiczny czy wyścig graficzny, uwagę skupiono na wydajności gry, a nie na zasobozernych efektach specjalnych. Te osiągnięte zostaną z pomocą grafik oraz iluzji w dwuwymiarowym, pikselowym świecie poklatkowych animacji.

### **1.2 Zakres pracy oraz założenia**

Jak wcześniej wspomniano, praca będzie skupiała przede wszystkim uwagę na wydajności gry. Takie założenie absolutnie nie stoi na równi z kiepską grafiką, choć prawdę jest, iż wszelkie elementy będą w niskiej rozdzielcości. Często 32x32 piksele z użyciem grafiki rastrowej. Konkretnie ujmując, projekt utrzymany jest w stylu graficznym

zwany „Pixel Art”. Standardowo rozcięgnięty obraz rastrowy przypominać będzie plamę z pikseli, jednak środowisko Unity 3D w wersji 5.x świetnie wspiera skalowanie grafiki utrzymanej w wyżej wymienionym stylu, dzięki czemu jakość pozostanie na odpowiednim poziomie.

Komponowanie utworów muzycznych jest dość czasochłonnym zajęciem wymagającym doświadczenia, jak i wyczucia, dlatego też przedstawiony zostanie jeden przykład kompozycji z użyciem otwartego (open source) oprogramowania „*BoscaCeoil*”. Pozostałe ścieżki dźwiękowe pobrane zostały z serwisu „*Soundcloud*” z profilu użytkownika Myuu (Nicolas Gasparini) oraz kanału w serwisie „*YouTube*” o nazwie „*Free Music for Commercial Use*”. W obu przypadkach utwory zostały udostępnione na licencji „*Common Creative*”.

Do tworzenia grafiki oraz animacji elementów wirtualnego świata posłuży otwarte oprogramowanie (Open Source) dostępne zarówno online jak i offline o nazwie „*Piskel*”. Prosty, przejrzysty interfejs pozwala intuicyjnie zorientowanie się w narzędziach, zaś szeroki wachlarz opcji oraz podgląd finalnej kompozycji pomaga przy tworzeniu wieloklatkowych, wielowarstwowych animacji. Zdecydowanym plusem są możliwości eksportu projektu do jednego, równo podzielonego pliku zawierającego wszystkie warstwowo połączone klatki animacji, które później trafią do Unity.

Część graficzna oraz udźwiękowienie, samotnie nie mają większego znaczenia w interaktywnej produkcji. Gracz w grze jednoosobowej oczekuje przede wszystkim akcji oraz fabuły i tak jak w drugim przypadku wystarczy upust fantazji, tak akcja wymaga przemyślanych mechanik, skryptów, rozwiązań. Ten problem był główną przyczyną wybrania Unity 3D. Społeczność zebrana wokół środowiska udostępnia masę poradników wideo w serwisie „*YouTube*”, dostępne są książki opisujące podstawy, jak i bardziej zaawansowane elementy, a w dodatku samo Unity oferuje rozbudowaną dokumentację.

Punktem odniesienia kształtującym wizję końcową projektu jest bardzo sympatyczna dla oka choć wymagająca od gracza nie małych umiejętności produkcja pod tytułem „*Hollow Knight*” studia Team Cherry. Gra ta również powstała na silniku Unity 3D w perspektywie dwuwymiarowej gry platformowej. Bohater, którym przyjdzie nam stworzyć podobnie jak postaci spotkane po drodze to przyjemnie, ręcznie rysowane robaczki/żuczki. Większość z nich okaże się przeciwnikami, jednak znajdą się od czasu do czasu i tacy co chętnie nam pomogą (NPC, ang. non-playable character) oferując zakup

przedmioty, bądź ciekawe dialogi. Gra zaliczana jest do gatunku zręcznościówek oraz metroidvanii, co samo w sobie sugeruje wyzwania koordynacyjne oko-ręka rozszerzane w czasie gry po przez zyskiwanie nowych umiejętności lub ich rozwój. Kolejnym założeniem są większe przeciwnicy pilnujący przejścia do następnego fragmentu świata zwani „*Boss’ami*” lokacji. Posiadają oni znacznie większą wytrzymałość oraz zakres ruchów, jednak występują tylko raz na całą grę. Świat w „*Hollow Knight*” podzielony jest na strefy, a każda ma swój unikalny wygląd. Graczowi przyjdzie zwiedzić jałowe ruiny, porośnięte bujną zielenią tereny, a nawet ciąg jaskiń wypełnionych grzybami o przeróżnych kolorach.

Wszelkie powyższe cechy zawarte zostaną w większym, bądź mniejszym stopniu w opisywanym projekcie nie pominiawszy zamieszczenia „*easter-egg’ów*” (sekretów, często odniesień do rzeczy z poza gry).

### **1.3 Przegląd materiałów źródłowych**

Podstawowymi źródłami wiedzy niezbędnymi do stworzenia projektu będą materiały pisemne takie jak dokumentacja do silnika Unity czy też książki o tematyce tworzenia gier oraz kursy wideo najczęściej z serwisu „*YouTube*”, tworzone nieraz hobbystycznie przez doświadczonych deweloperów. Szczególną uwagę warto zwrócić na kanał „*Blackthornprod*”, w którym w bardzo przystępny sposób przedstawione zostały takie elementy jak animacje, animator, interfejs graficzny dla gry, proste mechaniki poruszania, audio oraz system particle (cząsteczek). Drugim wartym polecenia jest „*Brackeys*” przedstawiający materiały bardziej po stronie kodu, jak zarządzanie menu głównym, system zapisu i odczytu stanu gry, zatrzymania rozgrywki, czy wykrywanie kolizji z pomocą raycast’u. Materiały udostępnione w sieci choć przeważnie dostarczą nienajgorszą wiedzę na start, to klasyczne źródła opisowe przeważnie zawierają głębsze wyjaśnienia, dzięki czemu stanowią idealne dopełnienie zdobytych podstaw. Z pomocą m.in. takich pozycji jak „*Getting Started with Unity 2015*” dr. Edwarda Laviera, „*Unity 2017 2D Game Development Projects*” autorstwa Lauren S. Ferro & Francesco Sapiro czy „*Swift 3 Game Development*” od Stephen'a Haney'a mamy możliwość na dopracowanie początkowych założeń oraz podejścia do tworzenia gier.

#### **1.4 O licencji.**

Wszelkie oprogramowanie oraz utwory muzyczne stosowane do stworzenia projektu bazują na licencji darmowej:

- **Unity 3D** – silnik oraz środowisko stworzone z myślą o produkcji gier wieloplatformowych, dostępny w wersji darmowej o ile zarobki nie przekraczają 100.000USD rocznie.
- **MS Visual Studio Enterprise 2017** – środowisko programistyczne wspierające Unity 3D, umożliwiające pisanie oraz komplikację skryptów. Posiadana przeze mnie wersja wymaga licencji jednak spokojnie wystarczyłaby darmowa alternatywa Visual Studio 2017 Community domyślnie instalowana razem ze środowiskiem Unity.
- **Piskel** – oprogramowanie dostępne na zasadzie „*Open Source*”. Posłuży do stworzenia większości elementów graficznych w stylu „*Pixel Art*”, które trafią do gry. Sam program zaprojektowany został z myślą o tworzeniu animowanych „*Pixel Art’ów*”. Występuje w wersji online jak i offline.
- **Gimp** – darmowe oprogramowanie będące edytorem grafiki. Posłuży do wykonania gradientów wykorzystanych później jako efekty wizualne.
- **BoscaCeoil** – środowiska z wieloma instrumentami pozwalające na dość proste komponowanie całkiem złożonych utworów muzycznych. Oprogramowanie jest rozwiązaniami z kodem otwartym, czyli darmowy „*Open Source*”.
- **AudaCity** – rozbudowany, darmowy edytor dźwięków, umożliwiający tworzenie oraz modyfikację nagrani. Posłuży do przygotowania efektów dźwiękowych.
- **Muzyka** – zawarte w projekcie utwory muzyczne posiadają licencję Creative Commons, w związku z czym autorzy zawarci zostali z napisach końcowych.

## **2. Narzędzia**

### **2.1 Unity 3D.**

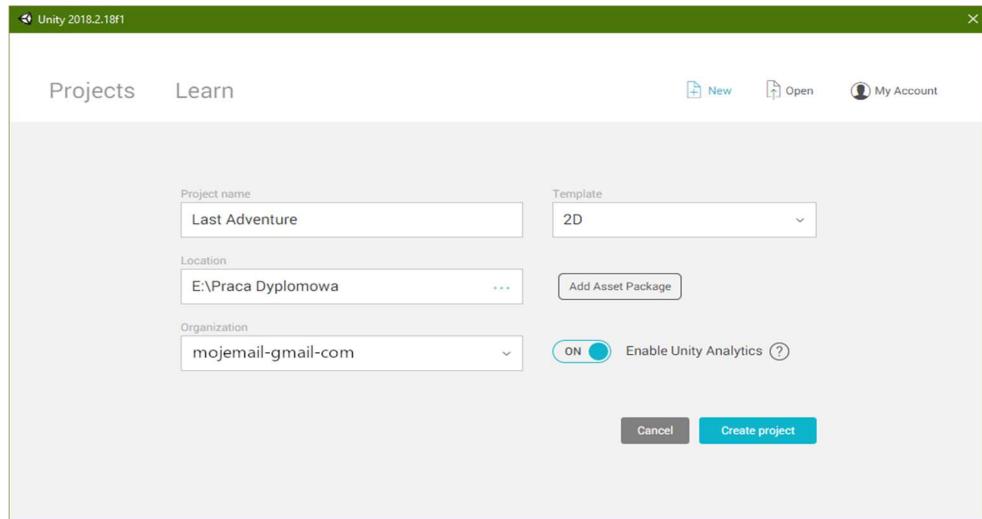
We wstępnie kilkukrotnie wspomniano o powodach mających wpływ na wybór Unity 3D jako środowiska pracy. Podsumowując jest to oprogramowanie w pełni darmowe, również dla celów komercyjnych o ile zarobki nie przekraczają 100.000USD, posiada ogólną społeczność tworzącą wszelkiego rodzaju bezpłatne poradniki wideo, sami twórcy silnika oferują rozległą dokumentację, a nawet własne kursy dla początkujących i co najbardziej oczywiste, dostępne są w księgarniach klasycznych bądź internetowych książki. Zaplecze wydaje się solidne, a także bardzo przystępne dla niewtajemniczonych. Wraz z kolejnymi aktualizacjami projekt rozwijał się, aktualnie oferując multum narzędzi dających ogromne możliwości.

Do rozpoczęcia pracy wymagane jest posiadanie owego oprogramowania na sprzęcie przeznaczonym do pracy to też plik instalacyjny znajdziemy na oficjalnej stronie Unity, po wybraniu zakładki „*Products*”, bądź „*Get Started*”. Następnie pojawią się dostępne oferty. W obecnym projekcie interesuje nas opcja „*Personal*”, będąca wcześniej opisywaną wersją darmową. Po jej wybraniu nastąpi przeładowanie do opisu, gdzie znajduje się przycisk „*Try personal*”. Odeśle on do podstrony z możliwością pobrania instalatora dla systemów z rodziny „*Windows*”. Warto zauważyć, że dobrym posunięciem będzie wcześniejsze, bądź późniejsze założenie konta lub zalogowanie się przy pomocy Google. Użytkownicy zarejestrowani w systemie mają możliwość przechowywania projektów w chmurze oraz kooperację z innymi. Całość działa na zasadzie kontroli wersji i łączenia rozgałęzień projektu niczym w systemie „*GIT*”. Przypomina to nic innego jak dobrego, starego „*Github'a*”.

Proces instalacji nie jest skomplikowany, jednak należy pamiętać, że domyślnie instaluje również Visual Studio Community, w związku z czym, posiadając inną jego wersję bądź decydując się na inne środowisko IDE, czy język programowania, warto wybrać niestandardową instalację by odłączyć niechciane elementy. Proces może być dość czasochłonny.

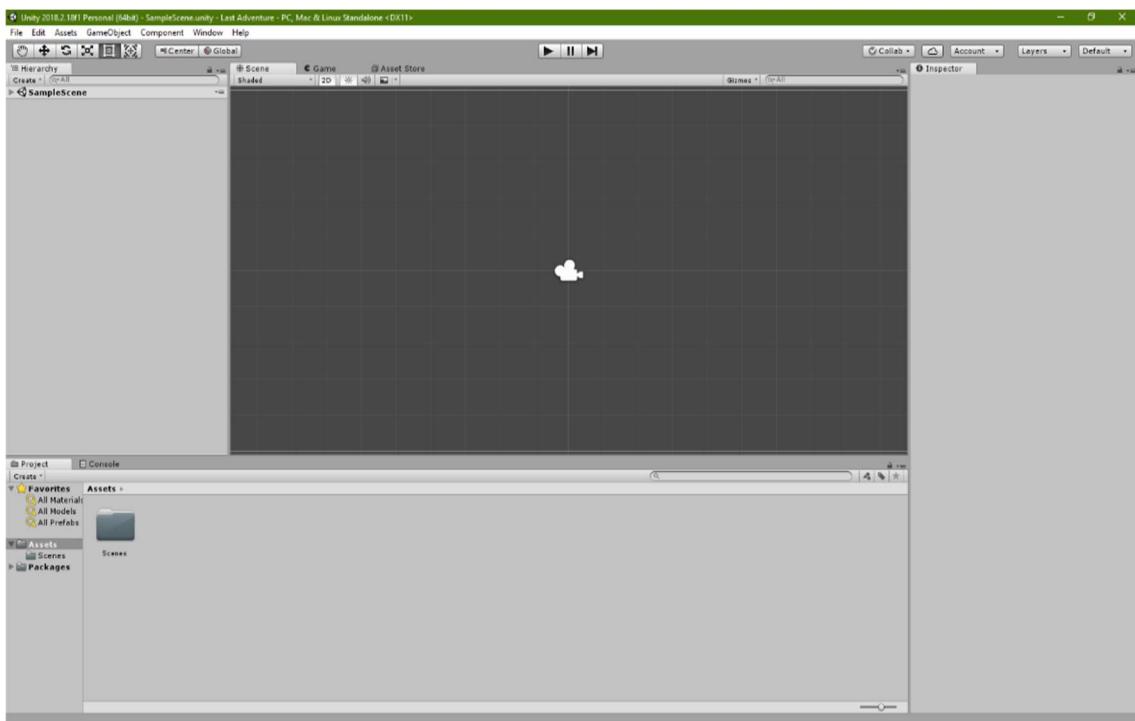
Po uruchomieniu programu, naszym oczom ukaże się okienko będące menadżerem projektów. W lewym górnym rogu znajdziemy kolejno przyciski/zakładki „*Projects*” oraz „*Learn*” przełączające widok między projektami, a kursami od twórców. Prawy

górny róg zawiera „New”, czyli nowy projekt, „Open” umożliwiający otworzenie istniejącego oraz „My Account” z informacjami i opcjami odnośnie wcześniej wspomiananego konta. Uwagę należy poświęcić zakładce „Projects”. Zawiera ona projekty przechowywane nie tylko lokalnie, ale również w chmurze.



Rysunek 1 - Początkowe dane dla nowego projektu.

Stworzenie nowego projektu nie powinno stanowić wyzwania. Wystarczy wybrać przycisk „New”, by pojawiły się pola na wpisanie podstawowych informacji, czyli nazwa projektu, lokalizacja na dysku, nazwa organizacji (najczęściej wypełniona automatycznie na podstawie maila), szablon, możliwość dodania paczki z elementami do tworzenia gry



Rysunek 2 - Nowy projekt z pustą sceną.

(Add Asset Package) oraz przełącznik dla Unity Analytics. Po wciśnięciu przycisku „Create project” nastąpi zimportowanie podstawowych modułów, by następnie ukazał się interfejs Unity z czystą sceną gotową do zapełnienia.

Początkowo należy zapoznać się z interfejsem oprogramowania. Klasycznie górny pasek zawiera menu główne z dostępem do funkcji środowiska. Opisanie każdego przycisku po kolei nie ma większego sensu, dlatego też więcej uwagi zostanie poświęconej jedynie elementom ważniejszym.



Rysunek 3 - Pasek narzędzi oraz warstw.

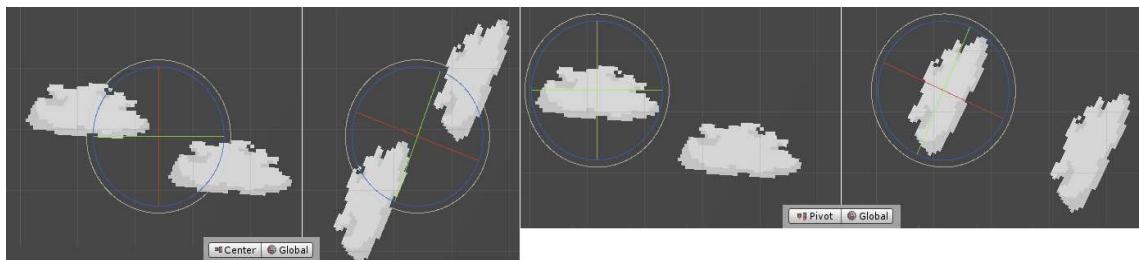
Widoczny na rysunku 3 pasek narzędzi oraz warstw zawiera kolejno [1] [2]:

- „*Hand Tool*” – łapkę znaną z podobnych rozwiązań, umożliwiającą przesuwanie sceny, klawisz skrót do tego narzędzia to „Q”.
- „*Move Tool*” – pozwalający na przesuwanie wybranych obiektów na scenie, klawisz skrót to „W”
- „*Rotate Tool*” – pozwalający na obracanie obiektów, do którego dostać się można również dzięki klawiszowi „E”.
- „*Scale Tool*” – służący do skalowania obiektów, ze skrótem „R”.
- „*Rect Tool*” – dający możliwość swobodnego rozciągania oraz przemieszczania obiektów, skrót prowadzący do narzędzia to klawisz „T”.
- „*Move, Rotate or Scale selected objects*” – łączący w sobie wszystkie wcześniej wymienione cechy poza swobodnym rozciąganiem, szybki dostęp zapewnia „Y”.

Powyższe narzędzia, po opanowaniu ich zmiany po przez skróty klawiszowe, będą służyć praktycznie cały czas przy budowie sceny przyśpieszając pracę. Równie przydatne są opisane kolejne dwa poniżej, gdyż ich przeoczenie może prowadzić do niechcianych sytuacji.

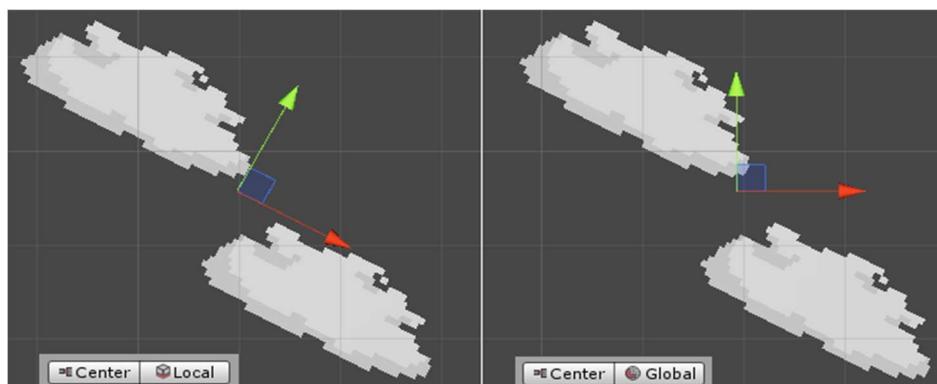
- Przycisk „*Toggle Tool Handle Position*” z domyślnym ustawieniem „Center”, ma możliwość zmiany swojego stanu na „*Pivot*”, co w praktyce oznacza modyfikację wielu obiektów w różny sposób. Dzięki opcja „Center” zaznaczone elementy postrzegane są przez edytor sceny jako jeden obiekt, dzięki czemu przy rotacji, bądź

skalowaniu zmieniają swoje cechy oraz położenie jako całość. „Pivot” traktuje każdy obiekt osobno według jego własnych cech.



Rysunek 4 - Edycja wielu obiektów w trybie Center i Pivot.

- Sąsiedni przycisk „*Toggle Tool Handle Rotation*” z ustawieniem domyślnym „*Global*”, jako wersję alternatywną przyjmuje „*Local*”, co w praktyce oznacza modyfikację obiektów z podoglądem na parametry w odniesieniu do całej sceny w przypadku ustawienia „*Global*” i w odniesieniu do edytowanego obiektu w przypadku „*Local*”. Dla przykładu, obracając przedmiot w trybie lokalnym, widoczna jest owa rotacja, a co z tym idzie, dla przykładu narzędzie „*Move Tool*” przemieszcza obiekt względem nadanego obrotu. Wracając do opcji globalnej, przedmiot obrócony przesunięty będzie względem oryginalnego kąta.



Rysunek 5 - Edycja obiektu z ustawieniem lokalnym i globalnym.

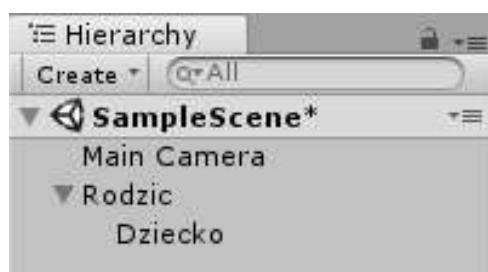
- „*Play*” uruchomienie gry startując na aktualnie edytowanej scenie.
- „*Pause*” pozawala na zatrzymanie gry.
- „*Step*” pozwala po-krokowo odtwarzać projekt, dzięki czemu można obserwować zmiany zachodzące w czasie gry i przykładowo wyłapywać niepożądane akcje.

Należy wspomnieć, iż przy uruchomieniu gry przez edytor wciąż mamy dostęp do obiektów na scenie, dzięki czemu można wprowadzać modyfikacje w czasie rzeczywistym. Uwaga! Zmiany wprowadzone w ten sposób zostaną wycofane w momencie zatrzymania gry.

- „*Collab*” to wspomniana wcześniej opcja zapisu projektu w chmurze na zasadzie kontroli wersji Git, podobnej do serwisu „*Github*”. Ułatwia w ten sposób pracę nad projektem więcej niż jednej osobie.
- Przycisk z grafiką chmury włącza zakładkę „*Services*” oferującą wiele funkcjonalności online w chmurze Unity.
- „*Account*” jak nazwa wskazuje zawiera opcje związane z kontem użytkownika.
- „*Layers*”, przycisk przedostatni, umożliwia wybranie warstw widocznych na scenie. Przy włączonych wszystkich warstwach, często zaznaczane są elementy nie te, które wskazano, a te które są bliżej kamery. Wyłączenie niepotrzebnych aktualnie warstw oszczędzi frustracji przy odnajdowaniu obiektów na scenie.
- Ostatnim przyciskiem, czyli „*Layouts*” oferuje zmianę szablonu okienek w Unity, który domyślnie ustawiony powinien być na „*Default*”.

Warto dobrze zapoznać się z powyżej wymienionymi, gdyż sprane operowanie tymi funkcjami znacznie poprawi wydajność pracy z edytorem oraz ułatwi budowanie scen.

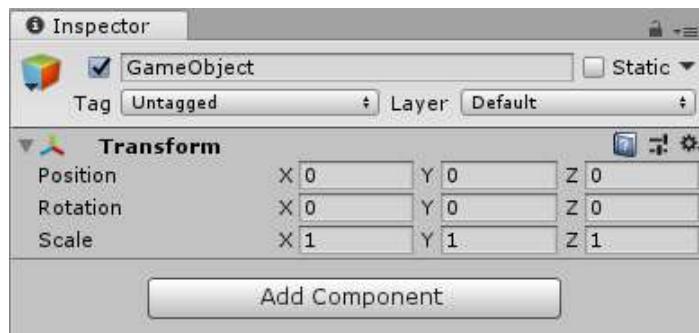
Pozostała część Unity złożona jest z okienek. Po lewej znajduje się zakładka z hierarchią, wyświetlająca w formie listy obiekty na scenie. Przy pustym projekcie, domyślnie dodana zostanie kamera. Obiekty mogą być ze sobą łączone na zasadzie „*Parent – Child*”. Obiekt mający w sobie inny obiekt staje się rodzicem i otrzymuje ikonkę trójkąta przy nazwie, oznaczającą możliwość rozwinięcia listy wewnętrznej z jego dziećmi.



Rysunek 6 - Hierarchia obiektów.

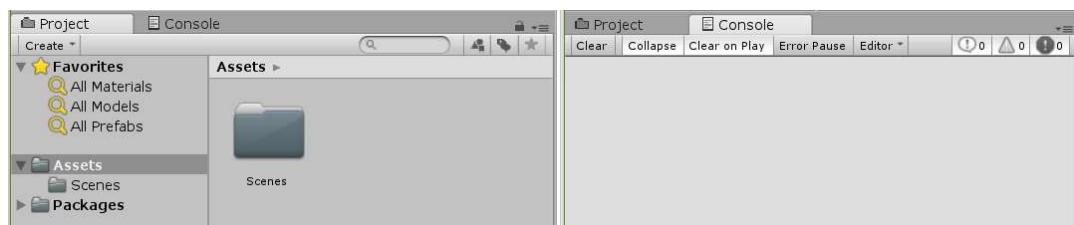
Niezbędnym do sprawnej pracy ze edytorem jest zrozumienie pojęcia „*GameObject*”, czyli obiekt występujący na scenie. Jest to pojęcie nie ścisłe, a raczej ogólne odnoszące się do wszystkiego co trafi na scenę. Najlepiej można to scharakteryzować słowem kontener, gdyż on sam to jedynie pusty twór o standardowej nazwie, znaczniku (Tag) i warstwie (Layer), mogący przyjąć dowolne cechy (komponenty), przybierając finalnie

formę jaką nada mu twórca. Domyślnie przy tworzeniu nowego, pustego obiektu, dołączany zostanie komponent „*Transform*” odpowiadający za jego istnienie na scenie (nadalaje pozycję w trzech wymiarach, rotację oraz skalę; Rysunek 7). Inspektor obiektów domyślnie znajdujący się po prawej stronie okna. Umożliwia on edycję komponentów definiujących wybrany obiekt. Bez obecności tej funkcjonalności, tak naprawdę twórcy niewiele jest w stanie zrobić, gdyż dzięki temu okienku wyświetlane są wszystkie cechy edytowanego elementu.



Rysunek 7 - Nowy, pusty obiekt - cechy w inspektorze.

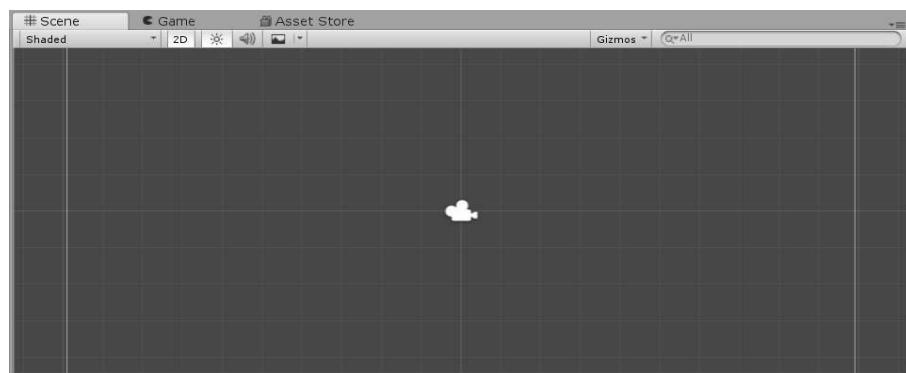
Część zajmują zakładki „*Project*” oraz „*Console*” (Rysunek 8). Pierwsza wymieniona to nic innego jak lista wszelkich elementów mogących posłużyć do budowy sceny. Unity umożliwia import gotowych paczek zarówno własnych, jak i pobranych z „*Asset Store*”. Takie elementy trafiają do folderu o nazwie „*Packages*” w opisywanej zakładce „*Project*”. Bardziej interesujący jest katalog wyżej z nazwą „*Assets*”, w którym znajdują się wszystkie elementy stworzone własnoręcznie na potrzeby projektu. Naturalnie we wnętrzu tego folderu można utworzyć podkatalogi by pogrupować dodane pliki. Druga zakładka o nazwie „*Console*” służy do podglądu wszelkich błędów oraz informacji od środowiska Unity, a także komunikatów wywołanych po stronie skryptów, a umieszczonych przez programistę. Konsola jest bardzo przydatna w wyłapywaniu błędów, przeważnie wskazuje miejsca ich wystąpienia, co w szybki sposób umożliwia zrozumienie przyczyny niepożądanej akcji. Wyjątkami są niemożliwe do zlokalizowania niekrytyczne błędy (nie blokujące programu) bazujące na prefabrykowanych obiektach usuwanych



Rysunek 8 - Zakładki: Projekt i Konsola.

podczas rozgrywki, w takim wypadku zostaje jedynie domyślnie się co jest powodem błędu.

Okno zajmujące największą część edytora, umieszczone w samym centrum to scena (Rysunek 9). Na niej przyjdzie twórcę umieszczać obiekty mające dać w finale produkt interaktywny w postaci wirtualnego środowiska gry, bądź programów. Warto wspomnieć, że Unity 3D może posłużyć nie tylko do produkcji gier, ale również przykładowo do tworzenia aplikacji urządzania wnętrz. Okno sceny posiada zakładki sąsiadujące, czyli „Game” oraz „Asset Store”. Pierwsza pozwala na wyświetlenie widoku z perspektywy kamery oraz służy za ekran przy odtwarzaniu edytowanego projektu. Druga zaś to sklep, na którym przeróżni twórcy oferują mnóstwo zróżnicowanych paczek z gotowymi do zaimportowania obiektaami (asset’ami). Większość ofert jest możliwa do pobrania za opłatą, jednak występują również darmowe.



Rysunek 9 - Scena z domyślną kamerą.

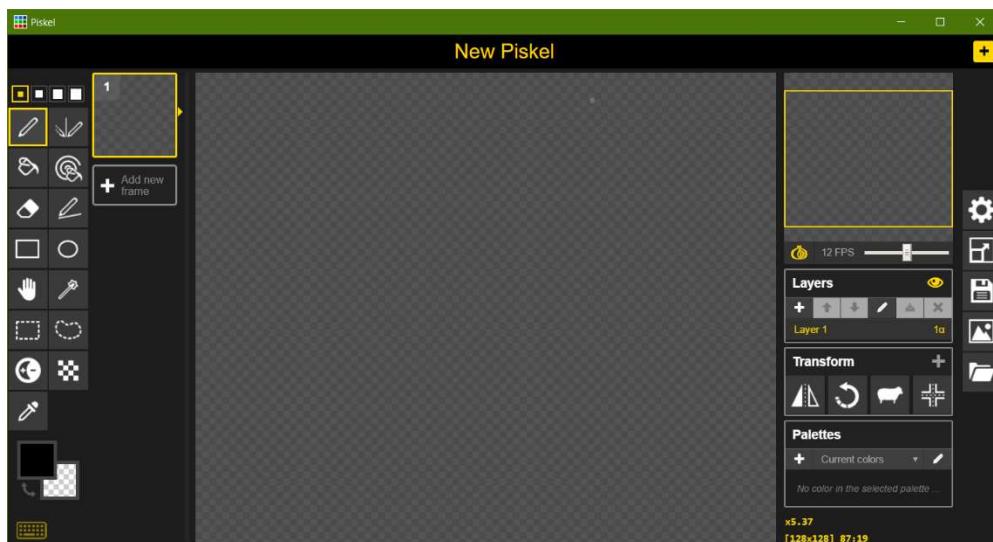
Wydedukować można z powyższego opisu interfejsu, że Unity złożony jest na zasadzie okienkowej reprezentacji funkcjonalności, podobnie jak, np. narzędzia Adobe. Oznacza to, że wybrane okno można odłączyć od całości przenosząc je poza strefę aplikacji głównej, dla przykładu zakładkę „Game” można umieścić na drugim monitorze by mieć lepszy podgląd na budowaną scenę z punktu widzenia kamery. Okienka można ze sobą łączyć dzięki czemu powstają zakładki będące szybkim dostępem do wybranej funkcjonalności. W razie potrzeby powrotu do domyślnego rozmieszczenia, wystarczy wejść w zakładkę menu głównego „Window”, najechać na „Layouts” i wybrać „Default”, bądź we wcześniej wspomniany przycisk na pasku narzędzi „Layouts” i wybrać „Default”. Wygodnym dla użytkowania jest zmodyfikowany szablon domyślny. Zakładkę „Project” warto przenieść obok „Hierarchy”, zaś „Animation” oraz „Animator” obok „Console”. Domyślnie „Animator” i „Animation” nie są włączone, można jednak to zmienić wybierając z menu głównego zakładkę „Window”, następnie „Animation” i klikając w

szukane funkcjonalności. Nie każdemu może przypaść do gustu zaproponowany szablon, dlatego zaleca się sprawdzenie różnych kombinacji i wybranie najwygodniejszej.

## 2.2 Piskel, Gimp, Pixel Art i animacje poklatkowe.

Grafika, choć nie zawsze najistotniejsza to na czasy obecne niezbędna przy produkcji gier. Najczęściej też to ten fragment zajmuje największą część zasobów maszyny, co jest głównym powodem wybrania stylu graficznego zwanego „Pixel Art”. Wszelkie elementy wykonane zostanie w niskiej rozdzielcości nie tracą przy tym swego uroku, co mocno ograniczy pobór mocy. Z pomocą silnikowi Unity 3D, całość zostanie bezstratnie przeskalowana.

Program umożliwiający komfortowe przygotowanie tej części projektu nosi nazwę „Piskel” i występuje zarówno w wersji online (Web), jak i offline (Desktop) na licencji Apache 2.0 (Open Source). Dla wygody pobrano przesykowaną przez twórcę paczkę dla systemu Windows z najnowszą wersją (obecnie: 0.14.0). Oprogramowania nie trzeba instalować, działa całkowicie na zasadzie przenośnej aplikacji, wystarczy rozpakować archiwum i uruchomić program (Piskel-0.14.0.exe).



Rysunek 10 - Nowy projekt w aplikacji Piskel.

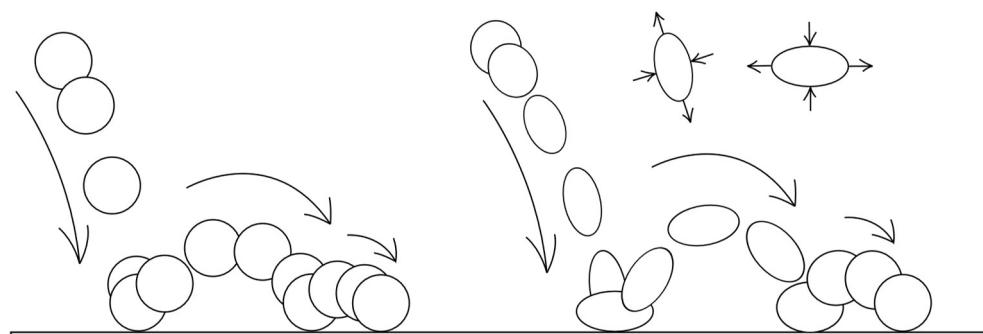
Każdorazowe uruchomienie, powita twórcę okienkiem z pustym projektem co przedstawiono na rysunku wyżej. Góra część będąca wąskim paskiem wyświetla nazwę aktualnie otwartego pliku oraz zawiera przycisk z symbolem plus otwierający nowe okno dla nowego projektu. Podobnie jak w Photoshopie, po lewej umieszczono pasek narzędzi do edycji zawartości płótna, tuż obok znajduje się lista klatek, z których złożona zostanie

animacja, środek zajmuje płótno będące obszarem zawierającym tworzoną grafikę, po prawej, kolejno od góry zamieszczono podgląd animacji, listę warstw na klatkach, narzędzi szybkiej transformacji obrazu oraz paletę kolorów użytych w projekcie. Pionowy pasek po prawej stronie zawiera opcje dla programu, umożliwiające zmianę rozmiaru płotna, czy też jego kolory, zapis stanu projektu, eksport do finalnej formy i funkcję importu plików obsługiwanych.

Do stworzenia dwóch gradientów wykorzystanych w późniejszej fazie użyto darmowego edytora grafiki „*Gimp*”, ale z powodu nieznacznej roli w odniesieniu do całości pozwolę sobie ominąć jego opis.

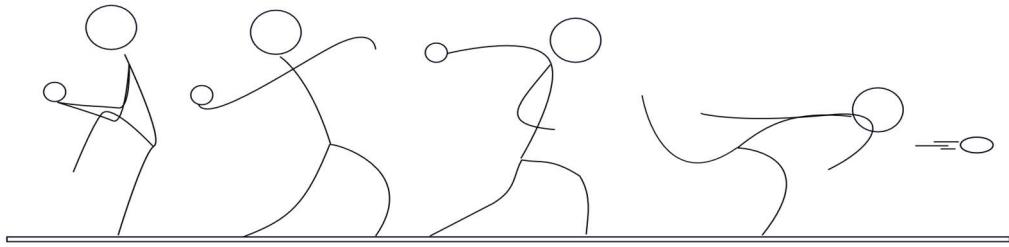
Obecny podrozdział poza przybliżeniem pracy z aplikacją Piskel zawiera również dwanaście zasad animacji, będące nieocenioną pomocą przy ich tworzeniu. Nie będzie to jednak opis szczegółowy, gdyż uwaga zostanie skupiona na krótkim, w miarę przejrzystym przedstawieniu sensu każdej zasady. Dla zyskania szerszej wiedzy warto sprawdzić podane w bibliografii źródła, tj. „*12 zasad animacji Disneya, czyli jak nakręcić idealny film animowany*” autorstwa Marcina Tomaszewskiego, „*12 zasad animacji*” od Rafała Schuberta oraz „*12 Principles of Animation (Official Full Series)*” z kanału YouTube o nazwie „*AlanBeckerTutorials*” [3] [4] [5].

- „*Squash and stretch*”, czyli zgniatanie i rozciąganie – zasada pierwsza i najważniejsza, gdyż jest to zabieg mający nadać animowanemu obiektem nieco wagi, twardości oraz właściwości fizycznych przy poruszaniu. Najczęściej przywoływany przykład piłki świetnie odzwierciedla tą regułę. Piłka nieelastyczna poleci w dół niewiele zmieniając swoją formę przy kontakcie z podłożem, gdy miękka wersja rozplaszczy się kumulując siłę odrzutu. W ten sposób oszukuje oko przedstawiając obiekt nie jako szkic, a coś fizycznego.



Rysunek 11 - Animacja obiektu twardego i miękkiego.

- „*Anticipation*”, czyli wyprzedzanie, mające na celu przygotowanie animowanego obiektu do jakiejś akcji. Kolejnym często przedstawianym przykładem jest baseballowy miotacz przyjmujący przed rzutem pozę świadczącą o jego zamiarze. W dalszej części pracy, przy opisie tworzenia przeciwnika, przedstawiono poklatkową animację, gdzie dobrze widać, iż wróg zamierza zaatakować.



Rysunek 12 - Wyprzedzenie w animacji na przykładzie miotającego.

- „*Staging*”, czyli inscenizacja, ma na celu zwrócenie uwagi, a raczej przedstawienie animacji w sposób oddający ducha sceny, po przez podkreślenie ważnych dla akcji fragmentów pokazanych w odpowiednim czasie. Ta reguła lepiej pasje do filmu animowanego niż pojedynczego elementu, to też w projekcie odnosi się bardziej do efektu końcowego całej zaprojektowanej sceny. W jednym z fragmentów mapy, idąc przez most w mrocznym lesie kamera jest blisko sterowanej postaci wywołując lekki niepokój po przez otaczający mrok, przez co odbiorca skupia się na samej postaci i najbliższym jej otoczeniu. Następnie w odpowiednim momencie widok znacznie odala się by ukazać grę światel i cieni w wąwozie zalanym fluorescencyjnymi chemicziami, tym samym przyciągając wzrok osoby grającej i tworząc efekt „WOW”.
- „*Straight Ahead Action and Pose to Pose*”, czyli rysowanie progresywne i dorysowywanie klatek pomiędzy kluczowymi. Ważnym jest to, iż są to dwa podejścia do tworzenia animacji. „*Straight Ahead Action*” wykorzystane zostanie przede wszystkim do animowania trudnego do przewidzenia w swoim zachowaniu obiektu. Polega to na tworzeniu klatki po klatce dając do uzyskania wystarczająco dobrego efektu końcowego. W przypadku „*Pose to Pose*” wiadomo jaki ruch w jaki sposób zostanie wykonany dlatego początkowo umieszczone zostaną klatki kluczowe, a następnie dorysowane zostaną klatki pośrednie. Przykładem obrazującym dla pierwszego podejścia może być powiewająca na wietrze flaga, gdzie klatka po klatce narysowane zostaną zagięcia materiału, zaś dla drugiego podskakujący człowiek. Tu można

ustalić przykładowo trzy klatki kluczowe: moment przybrania pozycji do wybicia, następnie postać będąca w powietrzu na najwyższej odległości i wylądowanie.

- „*Overlapping action and follow through*”, czyli nakładanie/zazębianie i podążanie za akcją. Ta część w zbiorze zasad polega na nadaniu różnym częścią animacji własnych ruchów w oparciu o główną część oraz zasymulowaniu pewnej fizyki animowanego ciała. Przykładowo postać zaczyna się rozpędzać. Elementy ciała wykonują swój ruch w oparciu o część główną, czyli tors, tworząc ruch całej postaci. Nagle droga się kończy, a zaczyna urwisko, więc postać wyhamowuje. Włosy podążające za postacią, polecą do przodu jako ostatnie wytracając siłę nadaną przez pęd by zaraz opaść wzduż ciała przyciągnięte przez grawitację.
- „*Ease in Ease out*”, rozpędzenie i zwolnienie, to nic innego jak nadanie naturalnego ruch obiekowi poddanemu animacji po przez przyśpieszenie na startie i wyhamowanie przy końcu. Biegącą postać nie przejdzie ze stanu spoczynku w nagły bieg, najpierw przyjmie pozycję nachylając się nieco do przodu, następnie nastąpi odechnięcie przez stopę i stopniowe przyśpieszenie. Tak samo w drugą stronę, rozpędzona osoba zacznie stopniowo wyhamowywać wytracając prędkość.
- „*Arc*”, czyli ruch po łuku. Większość otaczającego nas świata porusza się w pewnym stopniu po łuku. Elementy przemieszczające się liniowo sprawiają wrażenie ruchu robotycznego, ostrego i mało naturalnego. Prosty przykład: rzucona piłka po skosie w górę, nie poleci do najwyższego punktu w linii prostej, a po łuku i tak samo opadnie.
- „*Secondary actions*”, akcja drugoplanowa, to wszelkie dodatkowe elementy zawarte w animacji mające na celu lepsze oddanie sytuacji. Przykładowo osoba jedząca kanapkę, może ją ugryźć i zjeść, jednak jak kanapka jest smaczna dodatkowo przed pierwszym kęsem może oblizać usta, wskazując na to, iż ma na nią ochotę. Dodatkowo podczas przeżuwania owej kanapki, może okazać dzięki mimice twarzy, czy faktycznie mu smakuje, czy może jednak szkoda mu wypluć dlatego jakoś ją je.
- „*Timing*” lub synchronizacja, ważna kwestia, od której zależy naturalne odtworzenie animacji oraz przygotowanie dynamicznych scen. Ruch obiektów nie powinien

łamać praw fizyki, wręcz przeciwnie, powinien wyglądać jak najbardziej naturalnie. Nie jest jednak to reguła ścisła, a dopasowanie odtwarzania zależy od efektu końcowego. Nadając scenie bardziej dramatyczny wydźwięk, animacja będzie wolniejsza, o mniejszej ilości klatek pośrednich. Przy scenie dynamicznej, np. podczas walki, lepiej odebrana zostanie płynna animacja o większej ilości klatek.

- „*Exaggeration*”, wyolbrzymienie, reguła w pewnym stopniu ratująca twórców animacji przed pułapką „*Doliny Niesamowitości*” polegającej na tym, iż coś jest bardzo ludzkie, a jednocześnie kompletnie obce. Wywołuje to strach oraz odrazę. O tym zjawisku dobrze wiedział Walt Disney, dlatego też w swych produkcjach często wyolbrzymiał pewne zachowania. Prawdziwym celem zastosowania owej reguły jest jednak przełamanie nudy i nadanie kreskówkowego charakteru animowanemu obiekowi. Przede wszystkim tego typu zabiegi mają uwydatnić, podkreślić akcje obecnie mającą miejsce.
- „*Solid drawing*”, czyli zasada przedostatnia, nie będąca niczym innym jak doświadczeniem. Rysownik powinien znać anatomię ciała i wiedzieć jaki ruch w danej sytuacji może wykonać, dotyczy to również zwierząt oraz obiektów. Nadmierne wyginanie doprowadzi do nieprzyjemnych, nienaturalnych efektów.
- „*Appeal*”, ostatnia zasada będąca odpowiednikiem charyzmy, polega przede wszystkim nie na uroczym wyglądzie postaci, a raczej na jej czytelności i osobowości wyrażanej podczas ruchów. Na tą ostatnią regułę przeważnie wpływają wszystkie powyższe.

Powyższy etap powinien w pewnym stopniu przygotować każdego twórcę strony wizualnej do wyzwań jakie czekają przy projektowaniu scen. Sam Piskel nie jest trudny do opanowania. Choć posiada wiele narzędzi to są one jak najbardziej przydatne, dobrze opisane i intuicyjne, w związku z czym wystarczy chwila praktyki by nabrać wprawy.

## 2.3 BoscaCeoil.

Dźwięki wydobywające się z głośników to przetworzona energia oraz drgania w odpowiedniej częstotliwości. Tworząc nagrania, w rzeczywistości powstaje lista próbek z odpowiednim odstępem w czasie, a im więcej próbek tym lepsza jakość odtwarzania finalnej wersji. Ta wiedza prowadzi do logicznego wniosku, który brzmi: nie potrzebny jest instrument by uzyskać jego dźwięk. Wystarczy odpowiednia modyfikacja. Powstało wiele narzędzi pozwalających komponować z poziomu oprogramowania i „BoscaCeoil” jest właśnie takim programem, a na dodatek z licencją open source. Podobnie jak w przypadku Piskel'a, występuje w wersji online (web) oraz offline (desktop) na zasadzie aplikacji przenośnej. Po pobraniu paczki wystarczy rozpakować, by następnie uruchomić graficzny edytor.

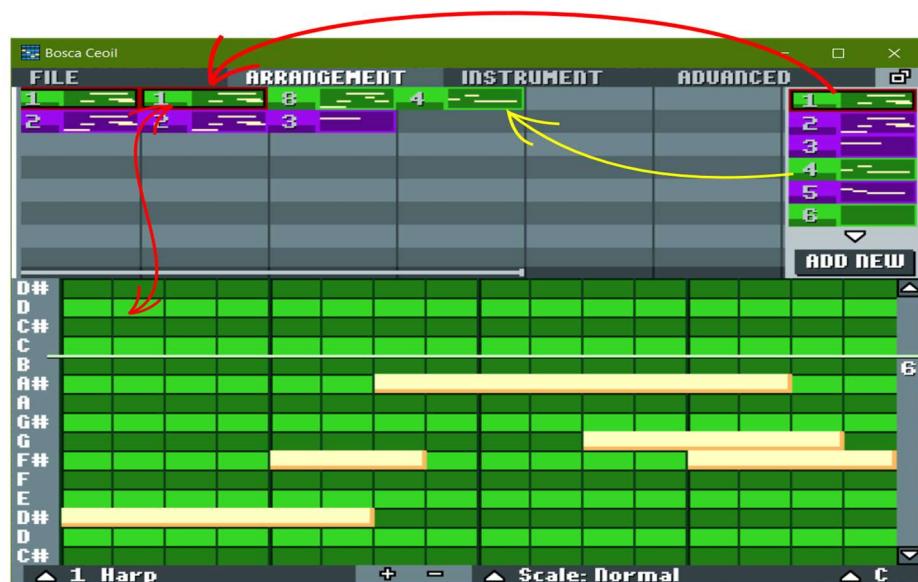


Rysunek 13 - Nowy projekt w Bosca Ceoil.

Aplikacje podzielono na dwie części, górną oraz dolną, gdzie dolna zawsze zostaje niezmienna, tj. daje podgląd na projekt. Powyżej w zależności od wybranej zakładki: „File”, „Arrangement”, „Instrument”, „Advanced”, wyświetcone zostaną odpowiednie opcje/narzędzia. Poniżej krótko opisano przyciski oraz funkcje dla zakładek:

- „File” – odnoszący się głównie do plików, będący również częścią powitalno-informacyjną dla programu, gdyż po lewej zawarto nazwę oprogramowania, jego wersję, autora głównego oraz przyciski z twórcami, pomocą i odtwarzaniem powstałego z projektu dźwięku. Prawa strona zawiera podstawowe funkcje związane z zapisem, odczytem, eksportem i importem projektu oraz ustawienie „Pattern” - długości ścieżki (maksymalnie 32 kratki, domyślnie 16), podgląd podziału (domyślnie 4) i „BPM” - prędkość odtwarzania (domyślnie 120).

- „Arrangement” – jest czymś w rodzaju linii czasu dla fragmentów kompozycji, dzięki czemu można łączyć wiele instrumentów oraz zróżnicowanych ścieżek, tworząc utwór końcowy. Edytor w dolnej części wyświetla nie cały projekt jak to sugeruje wcześniejszy opis, a wybrane fragmenty/kompozycje, które później zostaną połączone. Kafelki reprezentujące owe fragmenty kompozycji znajdują się w liście po prawej. Na rysunku 14, liniami czerwonymi przedstawiono kafelek z listy umieszczony na osi odtwarzania (część górna) i wyświetlono jego zawartość w edytorze (część dolna).



Rysunek 14 - Edytor fragmentów kompozycji w Bosca Ceoil.

- „Instrument”, na co nazwa wskazuje jest to zbiór instrumentów dostępnych w programie oraz modyfikator dla każdego z nich, zmieniający wydźwięk końcowy. Dzięki niemu można przykładowo uzyskać bardziej „kosmiczne” dźwięki jak strzał „blastera” z filmów „Star Wars”. Instrument należy najpierw stworzyć przez jego wybór i modyfikację, zaś finalna wersja trafi do listy po lewej, po czym będzie możliwy do wykorzystania w projekcie.

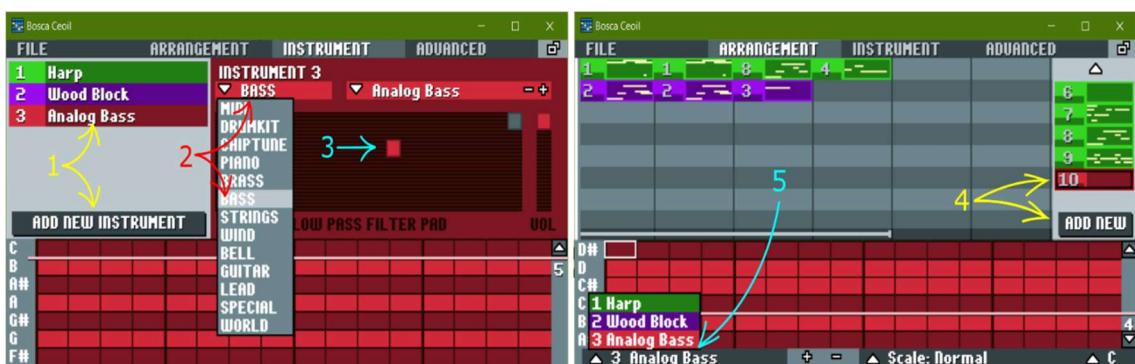


Rysunek 15 - Instrument "Wood Block" z wartościami domyślnymi.

- „Advanced”, opcje dodatkowe, których zmiana nie jest niezbędna, choć próby wprowadzenia zmian mogą przynieść całkiem ciekawy efekt końcowy.

Tak z ogólnie prezentuje się interfejs w „*Bosca Ceoil*”. Cała „magia” powstaje w części edytora fragmentów, jednak by przejść do tworzenia, warto pewne podstawy. Po pierwsze im wyżej w edytorze, im wyższa cyfra po prawej tym wyższy dźwięk i na odwrót. Przedstawiony na rysunku 14 projekt korzystał z szóstego poziomu, zaś scena zawiera 16 kratek z wydzieleniem stref przez nieco grubszą linię co 4 oraz prędkością otwierania 120 BPM. Po kliknięciu w konkretny prostokąt zostanie nałożony krótki dźwięk wybranego instrumentu. Używając kółka przewijania na myszce, obszar zaznaczenia rozciągnie się bądź zwęgi, a im dłuższy tym dłużej dźwięk zostaje wyciszany. Efekt można porównać do przytrzymania klawisza pianina. Usunięcie wstawionego dźwięku nastąpi po kliknięciu prawym przyciskiem myszy wskazanej kratki. Jeden fragment/scena może używać wyłącznie jednego instrumentu, dlatego też w zakładce „Arrangement” skomponowane elementy da się umieścić jeden pod drugim uzyskując nałożenie dźwięków. Widać ten zabieg na rysunku 14, fioletowy kafelek z kompozycją znajduje się pod zielonym.

Dodanie nowego instrumentu wiąże się z dodaniem nowej kompozycji, bądź wskazanie pustej. Najpierw należy przejść do zakładki „Instrument”, kliknąć „Add new instrument” w liście po lewej (krok 1, rysunek 16), by następnie nadać mu brzmienie przez wybór gotowych próbek z listy (krok 2) i modyfikację (krok 3). Kolejno w zakładce „Arrangement” dodano nową kompozycję (krok 4), by następnie wybrać dla niej wcześniej utworzony instrument (krok 5).

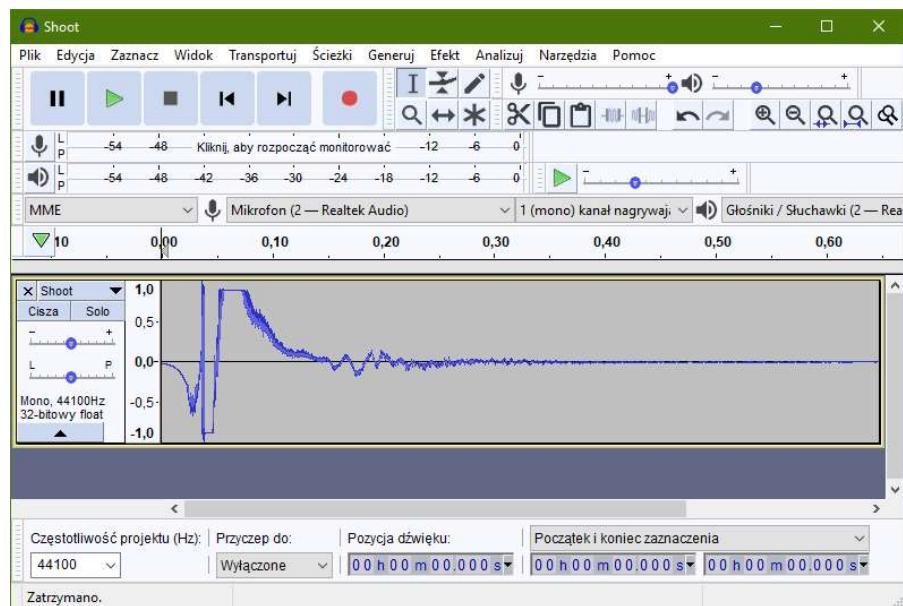


Rysunek 16 - Nowy instrument w *Bosca Ceoil*.

Dalsze działania to przemyślane ułożenie dźwięków lub metoda prób i błędów. Pewnym jest, że trzeba nabrać doświadczenia by komponować długie, przyjemne dla ucha ścieżki dźwiękowe.

## 2.4 Audacity.

Prawdopodobnie najpopularniejszy darmowy edytor audio, oferujący wiele przydatnych narzędzi manipulacji dźwiękiem, dostępny do pobrania rzecz jasna na stronie producenta. Posłuży do wykonania efektów dźwiękowych takich jak strzał wykonany uderzeniem w pustą butelkę, poddany odszumieniu oraz zniekształceniom typu ton, prędkość, czy odwrócenie. Podobnie jak w przypadku Gimp'a, nie ma sensu zawierać szczegółowego opisu używania Audacity, gdyż w sieci istnieje wiele dobrze przedstawionych poradników.



Rysunek 17 - Okno edycji dźwięku w Audacity.

### 3. Plan działania

Rozdział drugi powinien wystarczająco przygotować do procesu realizacji zadań, jednak pierw należy owe zadania ustalić. W związku z tym ta część pracy poświęcona jest planowaniu kolejnych etapów.

#### 3.1 Krok 1: Nowa scena oraz sterowanie.

Pierwszy kamień milowy to stworzenie sceny testowej, by przygotować podstawowy ruch postaci oraz umieścić pierwsze obiekty. Dla czytelności wyodrębniono kilka podpunktów, na które należy skierować uwagę [6]:

- **Warstwy** – dla filtrowania obiektów na scenie i interakcji. Domyślnie ustawiona warstwa to „*Default*”, co w praktyce oznacza, iż każdy element może wejść z innymi w interakcję (widzą swoją obecność) oraz są widoczne na scenie i przez kamerę. Oczywiście jest opcja zmodyfikowania właściwości warstwy, dzięki czemu twórca ustali co w rzeczywistości powinno ze sobą wchodzić w reakcję. Podążając tym tokiem myślenia, stworzono dodatkowe warstwy: „*Player*” oraz „*Ground*”, na siebie oddziaływujące [1] [7].
- **Kamera** – przed umieszczeniem pierwszych obiektów, ważne jest podjęcie decyzji jaki widok kamery odpowiada najbardziej do projektu. „*Perspective*” z efektem głębi, czy „*Orthographic*”, w którym nie ma większego znaczenia oś „*z*”. Pierwszy przypadek polega na umieszczeniu elementów z różną odległością od kamery, w efekcie czego uzyskana zostanie głębia widoczna przy ruchu. Choć na scenie nie widać zmian to w obrazie końcowym obiekty umieszczone dalej będą przemieszczały się nieco wolniej, niż te umieszczone bliżej kamery. Druga opcja wyklucza głębię, co znacznie upraszcza proces produkcji. Efekt widoczny na scenie zawsze będzie tak samo przedstawiony przez kamerę. W ramach tej pracy użyty zostanie widok „*Orthographic*”, który mimo wyrzeczenia fajnego efektu, ostatecznie jest czytelniejszy po stronie edytora oraz widoku. Obiekt kamery umieszczony zostanie w obiekcie rodzica, co ma na celu nadanie większej swobody zarządzania jej położeniem. Skrypt śledzący bohatera płynnym ruchem przemieści całość względem pozycji globalne całego projektu, zaś dziecko użyje animatora z animacjami typu potrząsanie bazując na położeniu lokalnych. W praktyce oznacza to, że kamera nie będzie blokowała ruchu za graczem z powodu animacji. [8]

- **Bohater** – postać przyjmie formę obiektu głównego posiadającą w sobie kilka innych. Pozwoli to wyodrębnić funkcje dla różnych elementów [9] [10]:
  - punkt kontaktu z podłożem, który posłuży do oprogramowania podskoku,
  - odtwarzacz dźwięku podczas poruszania,
  - punk śledzenia dla kamery, by ta podążała za bohaterem, jednocześnie nie stawiając go idealnie w centrum,
  - cień postaci, za który posłuży rozciągnięta grafika z przezroczystością,
  - ekwipunek zawierający w sobie pola/miejsca na przedmioty przechowywane oraz wyekwipowanie.
  - broń w miejscu rąk, by możliwa była podmiana fragmentu grafiki zamiast całej postaci w momencie wyekwipowania innej broni niż domyślna (przy braku wyposażenia załączona zostanie grafika z rękoma),
  - głowa, jednak nie z podmienieniem grafiki, a nałożeniem obiektu, przysłaniając oryginalną,
  - pośmiertny duch, bardziej w ramach urozmaicenia.

Do obiektu rodzica (obiekt główny) trafią komponenty odpowiedzialne za fizykę, kolizje, wyświetlanie grafiki 2D, stan animacji oraz skrypt kontroli, zaś jako warstwę ustawiono wcześniej stworzoną o nazwie „*Player*”.

- **Podłoż** – scena jest w pewny sensie przestrzenią nieograniczoną przypominającą pusty kosmos z punktem centralnym. Tu po raz kolejny przyjdzie twórcy podjąć decyzję, w jaki sposób zostanie przedstawiona gra: od góry, od boku czy w rzucie izometrycznym. Dlaczego to jest ważne? Widok zdefiniuje czym są w świecie gry osie „*x*”, „*y*” i „*z*”, gdzie jest dół, a gdzie góra i czy w ogóle można spaść w otchłań nieskończoności. Przy widokach, w których kamera znajduje się nad projektowaną sceną, nie trzeba myśleć o tworzeniu ograniczeń „spadania”, jak ma to miejsce w widoku z boku, a wystarczy wyznaczyć granice mapy. Dodatkowo mniejsze znaczenie mają krajobrazy w tle, gdyż nie pojawiają się wcale, bądź są całkiem sprytnie umieszczone jak np. w „*Hyper Light Drifter*”. Obecny projekt ma charakter gry platformowo-przygodowej z elementami metroidvanii w związku z czym najlepszym wyborem jest klasyczny widok z boku, będący m.in. w dobrze znanym większości

osobom „*Mario*”. Punkt wcześniej wspomniano o nadanej bohaterowi fizyce, co oznacza, że postać może spadać w nieskończoność. Zapobiegnie temu obiekt z komponentem kolizji o odpowiedniej warstwie „*Ground*”.

- **Sterowanie** – skrypt sterujący postacią oparty będzie o standardowy dla Unity język programowania C#, z wykorzystaniem standardowych funkcji [11]:
  - „*Start*” – dla ustawienia parametrów początkowych oraz określenia prefabrykatów z poziomu kodu.
  - „*Update*” – działając na zasadzie pętli, odświeża stany według wydajności sprzętu. Posłuży przede wszystkim do odczytu wciskanych klawiszy sterujących wywołujących odpowiednie działania.
  - „*FixedUpdate*” – tak samo jak przy „*Update*” jednak ze stałym czasem odświeżania, tj. około 0.02ms. Przykładowo przemieszczenie obiektu z pomocą funkcji „*Update*” sprawi, że na słabszej konfiguracji odbywać się to będzie wolniej niż na mocnym sprzęcie, zaś niejednakowy czas wpłynie na płynność ruchu, wywołując wrażenie szarpania.
  - „*OnTriggerEnter2D*” – głównie dla wyłapywania kolizji mającej na celu uruchomić jakiś ciąg działań. Przykładowo po wejściu w wyznaczoną strefę nastąpi oddalenie kamery.
  - Inne funkcje prywatne oraz publiczne wpływające na sterowanie, takie jak blokowanie ruchu podczas ważnego dialogu, bądź wywołanie animacji śmierci i uruchomienie procesu ładowania ostatniego zapisu.

Przypisanie przycisków klawiatury z poziomu kodu nie jest opcją najlepszą, o wiele bardziej zaleca się użycie przystosowanej do tego funkcji w Unity, a konkretnie „*Input Manager*”. Zwiększając parametr „*Size*” nastąpi dodanie nowego przycisku przez powielenie ostatniego. Zakładając, iż wymagany jest skok, zmodyfikowano nowo dodany element nadając mu nazwę „*Jump*” oraz przypisując do „*Positive Button*” klawisz spacji, czyli „*space*”. Wracając do kodu, od teraz pobranie klawisza skoku nastąpi z wykorzystaniem nazwy w tabeli wejść zamiast sztywno przypisanego przycisku z klawiatury.

### 3.2 Krok 2: Animacje.

Przemieszczający się obrazek po ekranie to jedno, zaś drugie to odpowiednia jego zmiana w odpowiednim czasie. W zależności od wykonywanej czynności bohater powinien przyjąć inny ruch, co oznacza nic innego jak zmianę animacji.

- Animacje bohatera:

- „Idle” – zapętlona animacja odtwarzana w czasie bezczynności (animacja nie dotyczy jedynie rąk, ponieważ będą one osobnym obiektem z osobnym animatorem),
- „Run” – czyli nic innego jak bieg podczas przemieszczania postaci, jednak w tym przypadku z bronią w rękach,
- „Jump” – przy obecnym widoku postać może pójść w lewo, prawo, dół i oczywiście górę. Spaść w niżej można za sprawą grawitacji, jednak latanie nie jest naturalnym zachowaniem, dlatego też skok będzie najlepszym wyjściem na dostranie się na elementy znajdujące się wyżej,
- „Crouch” – kucanie, odpowiedzialne przede wszystkim za uniki.
- „Equip” – animacja przy szperaniu w ekwipunku,
- „Shoot” – odrzut przy wystrzale, nie animuje broni, gdyż ta podobnie jak ręce będzie posiadała osobną animację oraz animator,
- „Dead” – animacja podczas utraty wszystkich punktów życia, czyli śmierci,

Ręce jak i strzelba, będą osobnymi obiektami. Ma to na celu wyeliminowanie potrzeby tworzenia całej grafiki postaci od początku dla każdej nowej broni bądź przedmiotów możliwych do wyekwipowania.

- Animacje strzelby:

- „Gun\_Idle” – sama broń się nie porusza jednak posiada animację przemieszczającego się światelka oznaczającego gotowość do strzału,
- „Gun\_Shoot” – poderwanie broni przy strzale,
- „Gun\_Run” – podskakiwanie broni przy biegu.

- Animacje rak:

- „Hand\_Idle” – zapętlona animacja rąk w czasie bezczynności.

- „*Hand\_Run*” – zapętlona animacja rąk przy biegu,
- „*Hand\_Jump*” – jedna klatka z pozycją rąk podczas skoku,
- „*Hand\_Crouch*” – jedna klatka animacji biegu, wykorzystana przy kucnięciu.

Zmiana między rękoma, a bronią zadziała dopiero przy sprawnym zarządzaniu ekwi-punkiem, dlatego też początkowo domyślnie wstawiony zostanie obiekt ze strzelbą. Ani-mator sterujący animacjami omówiony zostanie w następnym rozdziale podczas tworze-nia, z przedstawieniem adekwatnych grafik.

### **3.3 Krok 3: Zapis, odczyt i obiekt „Config”.**

Ten etap skupi uwagę na mechanizmie zapisu oraz odczytu stanu gry, by zapobiec sytuacji, gdzie osoba grająca musiałby przechodzić całość od początku w wyniku utraty postępu. Jest to również dobry punkt na omówienie obiektu przechodzącego między sce-nami. Domyślnie przy lądowaniu kolejnej lokacji, obiekty zostaną zniszczone, by za-la-dowane mogły być nowe. Sprawia to pewne problemy w grze złożonej z wielu obszarów. Przykładowo portal przenoszący do innej sceny to nie problem, jeśli nie są potrzebne dane takie jak ilość aktualnie posiadanych punktów życia, czy miejsca docelowe. Gra jest jednak spora, a sposób z każdorazowym wczytywaniem informacji z zapisu nieopty-malny, dlatego też posiadając obiekt istniejący od momentu włączenia gry do jej za-mknięcia, bardzo usprawnia przechowywanie i modyfikacje kluczowych danych [12].

- „*Zapis gry*” – powstanie klasa „*DataSave*” posiadająca zmienne z informacją o: indeksie aktualnej sceny oraz pozycji gracza w momencie zapisu. To na początek. W późniejszej fazie dojdzie zapis litych przedmiotów niewyekwipowanych, listy przed-miotów wyekwipowanych, przedmiotów do zniszczenia w grze (przedmioty możli-wych do zebrania tylko raz) oraz listy aktywowanych zdarzeń (co zdecyduje o linii dialogowej). Obiekt powstały z owej klasy zostanie zapisany do pliku w postaci bi-narnej co ma na celu ochronę danych przed modyfikacją. Miejsce zapisu to katalog jawny (dostępny nawet po zbudowaniu projektu), czyli „*StreamingAssets*” w pod-katalogu „*Data*”. Sam zapis realizowany będzie przy wyznaczonych punktach w świecie gry. Dla komfortu testowania projektu dodano szybki zapis/odczyt pod kła-wiszami funkcyjnymi F5 i F8.
- „*Wczytanie gry*” – po wczytaniu pliku z zapisem, zostanie on przeformatowany do obiektu typu „*DataSave*”, po czym zwrocony do ogólnodostępnej zmiennej w

przenoszonym między scenami obiekcie „*Config*”. W przypadku błędu przy odczytywaniu, zwrócone zostaną dane domyślne, utworzywszy przy okazji nowy plik zapisu.

- „*Config*” – czyli, obiekt wędrujący między scenami, którego główną rolą będzie pamiętanie kluczowych zmiennych oraz posiadanie funkcji ogólnych jak właśnie zapis czy odczyt save'a. Podsumowując: początkowo w menu głównym trafią do niego dane konfiguracji dla gry (poziom dźwięku w tle oraz efektów, a także wybrany język), następnie przy kontynuacji przygody wczytany zostanie stan gry i dopiero po tem elementy gry właściwej w funkcji „*Start*” pobiorą z niego potrzebne dane.

### 3.4 Krok 4: Menu główne & UI.

Każda gra wita osoby zainteresowane swoim ekranem z menu głównym. Ta swego rodzaju brama do świata gry to nie tylko pierwszy kontakt z graczem ale również szereg opcji jakie mu oferuje, by dostosować rozgrywkę do indywidualnych potrzeb. W przypadku tego projektu nie będzie inaczej, dlatego też poniżej zawarto wizję omawianego tematu [10].

- „*Last Adventure*” – tytuł gry widniejący dumnie w górnej części ekranu, oświetlany przez białą smugę światła, a zaraz pod nim podtytuł: „*Epizode One: Story About Locked Door*”,
- Niewielka grafika wojownika w tle za podtytułem,
- „*Kontynuuj*” – przycisk wczytujący ostatni zapis gry,
- „*Nowa Gra*” – przycisk rozpoczynający nową grę,
- „*Ustawienia*” – przejście do ustawień, w których wystąpi możliwość zmiany głośności ścieżki dźwiękowej w tle oraz efektów, a także zmienić język gry,
- „*Sterowanie*” – wyświetlenie grafiki z instrukcją przycisków dla klawiatury oraz pada z rodziną xbox,
- „*Wyjście*” – zamknięcie okna gry,
- Wersja gry w prawym dolnym rogu,
- System cząsteczek po prawej i lewej stronie, wypuszczający w górę kolorowe rozmyte plamki,

- Na koniec dla lepszego efektu przyciemnione krawędzie stworzone z pomocą gradientu przechodzącego z pełnej czerni do przezroczystości.

Przechodzenie po przyciskach zrealizowane zostanie z pomocą skryptu oraz animacji, a same przyciski wykorzystają komponenty z zakładki „UI”, które przeznaczone są właśnie do tego zadania. Całość zamieszczona zostanie w obiekcie „Canvas” posiadającym bardzo przydatny komponent „Canvas Scaler”, który z ustawieniem „Scale With Screen Size” w „UI Scale Mode” automatycznie dostosuje rozmiar gry do ekranu monitora [13] [14].

Co do interfejsu podczas rozgrywki, stworzony zostanie w identyczny sposób, ale ograniczony do grafiki przedstawiającej ilość obecnie posiadanych życia w postaci listków na gałzce. Z każdym obrażeniem uruchomiona zostanie animacja rozpadania listka, zaś z przywróceniem zdrowia, animacja odrośnięcia.

### **3.5 Krok 5: Design, światło, cienie, kamera i portale.**

Osoby oglądające film widzą przede wszystkim obraz im przedstawiony, czyli finalne nagrania często z domieszką efektów specjalnych, dopiero następnie zwrócią większość uwagi na fabułę, udźwiękowienie i grę aktorską. Podobnie jest z grą, gdzie zazwyczaj odbiorca ogląda akcję, na którą ma wpływ i dzięki czemu reaguje na to co widzi. Ta interakcja odróżnia film od gry, tym samym nieco modyfikując zasady odbieranych doznań. Mapa posiadająca wiele efektów wizualnych oraz obiektów na scenie, w szybkiej rozgrywce będzie mało czytelna i odwrotnie, w grze przygodowej z wolnym tempem, ograniczona, powtarzalna ilość obiektów sprawi, że całość wyda się po prostu dość brzydka. „Last Adventure” znajdzie się gdzieś pomiędzy, mieszając elementy gry przygodowej z domieszką akcji, to też nie można doprowadzić do sytuacji, gdzie otoczenie w jakim gracz się znajduje jest bardzo powtarzalne. Zniechęci to osobę grającą do dalszej przygody oraz eksploracji. Druga strona medalu to świadomość, iż im więcej różnych obiektów znajdzie się na mapie tym więcej zasobów sprzętowych zostanie wykorzystane i więcej czasu pochłonie tworzenie grafik. Powstaje więc pytanie: jak to zoptymalizować? Sprytny twórca posłuży się możliwościami komponentu „Transform”. Zacznie go obracając, wyginając, zmieniając jasność i kolor (komponent „Sprite Renderer”), uzyskując w ten sposób nie raz zaskakujące efekty, co dobrze widoczne będzie w następnym rozdziale. Różne lokacje oddają różny klimat, a w osiągnięciu dobrego efektu kluczową rolę zagra światło oraz cień. Czym jednak są owe zjawiska? W nieoświetlonym

pomieszczeniu człowiek zobaczy jedynie mrok. Nie znaczy to oczywiście, że przedmioty znajdujące się tam znikną, wciąż można potknąć się o chociażby śpiącego kota. Świat gry podobnie jak film w efekcie końcowym stanie się obrazem dwuwymiarowym, czyli wyzbięcie się przestrzeni i to pod kątem tej perspektywy należy zastanowić się czym jest światło, a czym cień. Malarz tworząc swoje dzieło miesza kolory farb dla uzyskania tych, które go interesują, to też skupiając się na atmosferze radosnej, przykładowo malując słoneczną plażę, wybierze odcienie jasne, zaś oddając klimat mrocznego zamku te ciemniejsze. Z takiej obserwacji wniosek nasuwa się sam. Światło to po prostu kolory jasne, a cień wręcz przeciwnie. Projekt utrzymany jest w perspektywie 2D, co oznacza, że zarówno na scenie jak i w kamerze, tworząc lokacje z użyciem świateł, obiekty nie zostaną oświetlone, lecz rozjaśnione, bądź przyciemnione dzięki odpowiedniej grafice nałożonej warstwę wyżej (bliżej kamery) z ustaniem przezroczystości. Po ustaleniu tych wstępnych założeń, można wypunktować zasady, na których bazować będzie projekt.

- **Obiekty na scenie** – cały świat wirtualny złożony zostanie z grafik dwuwymiarowych odpowiednio ułożonych wedle parametru „*Order in Layer*”, dzięki któremu twórca decyduje co się znajdzie za czym. Po zimportowaniu grafiki, należy pamiętać o zmianie jej parametru „*Filter Mode*” na rozwijany z listy „*Point (no filter)*”, co zapobiegnie rozmazaniu.
- **Grafika terenu** – elementy podłożu stanowić będzie kilka zapętlonych oraz wykańczających grafik, stworzonych w sposób dający wrażenie glebi. Dodatkowo powstaną elementy stanowiące całe pojedyncze obiekty.
- **Elementy świata** – powstaną grafiki statyczne jak i animowane wykorzystywane wielokrotnie na mapie. Możliwość modyfikacji obiektów pozwoli na uzyskanie zróżnicowanych efektów. Gdy elementy reagują na obecność gracza, świat staje się bardziej naturalny [15].
- **Światło i cień** – czyli przygotowany gradient w postaci koła przechodzący z białego wypełnienia środka do całkowitego zaniknięcia na krawędzi. „*Sprite Renderer*” umożliwi manipulację kolorem (biała grafika może przybrać każdy dowolny kolor z palety) oraz przezroczystością. Twórca ma możliwość rozciągnięcia obiektu do formy długiej smugi, zaś po nałożeniu nad inny obiekt rozjaśni grafikę pod nim. Sterowanie rozjaśnieniem następuje przez modyfikację kanału „*Alpha*”

(przezroczystość). Podobnie uzyskany zostanie cień. Wystarczy jedynie zmienić kolor sprite'a na czerń [16].

- **Przejście do innej sceny** – sprytnie ukryty portal w drzwiach, wrotach, szczeliny i tak dalej. Po wejściu postaci w wyznaczony obszar, nastąpi przeładowanie sceny do wskazanego innego portalu i tu należy pamiętać o dodatkowym obiekcie dziecku, odsuniętym od owego obszaru. Stanowi ono miejsce pojawienia się bohatera, by ten nie utknął w nieskończonej pętli.
- **Skrypty i animacje** – elementy sceny nie muszą być statyczne, nadając nieco animacji ożywiamy świat wirtualny, polepszając odbiór końcowy. Idąc krok dalej przy wstawianiu większej ilości obiektów warto stworzyć małe skrypty wprowadzające nieco losowości dla każdego. Przykładowo losowany zostanie rozmiar chmurki oraz jej prędkość, dzięki czemu nie trzeba każdej z osobna konfigurować.
- **Muzyka i efekty dźwiękowe** – gra, czy też film bez dźwięków wydają się niedokończoną produkcją. Jakby nie patrzeć, atmosfera miejsca to nie tylko gra świateł, ale również audio w tle nasuwające na myśl sielankową atmosferę, nadające nutkę niepokoju, czy też wartką akcję. Podobnie z efektami przy podjętych akcjach. Wystrzałowi z broni powinien towarzyszyć huk, a biegu po jesiennych liściach szelest [17].

Dalsze działania, czyli projektowanie mapy to kwestia puszczenia wodzy wyobraźni przez twórcę. Projekt zakłada zróżnicowane lokacje, dzięki czemu gracz zwiedzi takie miejsca jak jaskinię, górę z drzewem życia, katakumby, mroczny las z radioaktywnym bagnem, podniebne miasto.

Sama sceneria nie zawsze przedstawia ogólne swe piękno. Dobrze nakręcony film to ujęcia wywołujące w człowieku zachwyt i nie tylko. W grze jest to nie mniej ważne. Odpowiednie ruchy, podążanie kamery za bohaterem oraz zoom na miejsca warte większej uwagi, bądź punkty widokowe nie raz potrafią przyprawić o pozytywną gęsią skórkę. Przykładowo w mrocznym lesie obraz zostanie przybliżony ograniczając widoczność, a niepokój nieco wzrośnie, bo nie widać wroga choć go np. słyszać. Odwrotny efekt będzie w momencie oddalenia kamery, ukazując tym samym niesamowity krajobraz, magię tego konkretnego miejsca. Przygotowany zatem zostanie skrypt kontroli nad kamerą, dzięki któremu będzie podążała za bohaterem z ruchem płynnym oraz wykona odpowiedni, również płynny zoom.

Ostatnią kwestią będą portale między scenami. Ich działanie polega na wskazaniu sceny oraz portalu docelowego, co przechowane zostanie w „*Config'u*”, a użyte po stronie przeładowania sceny z pomocą animowanego „*Plane'a*” w „*Canvas*”. Nie jest to nic skomplikowanego, jednak należy pamiętać by odpowiednio przyozdobić miejsce przejścia, przykładowo chowając je za skałą w wejściu do jaskini co da wrażenie, że faktycznie bohater tam wszedł i coś tam dalej jest.

### 3.6 Krok 6: Fabuła, wizja gry.

Wszelkie utwory, czy to książki, filmy, obrazy, a nawet muzyk zawierają w sobie pewną historię, którą pragną przekazać. Jest to ważna część, a dla produkcji z serii przygodowej wręcz niezbędną, ponieważ definiuje przebieg akcji. Nie mając w zanadrzu niesamowitej historii, można zacząć improwizację przez umieszczenie bohatera w nieznanym mu miejscu i kompletną amnezją. Mimo wszystko warto trzymać się założenia od ogólna do szczegółu, mieć pewną wizję całości, kierującą do kolejnych interakcji [10] [18]. Z powodu, iż jest to czysta teoria, w części praktycznej ten punkt zostanie pominięty.

- **Uogólniona wizja** – bohater budzi się w jaskini, gdzie spotyka starca, który przedstawi mu pierwsze mechaniki i przybliży sytuację, taki tutorial na wstępie. Następnie gracz trafi na masywne wrota, gdzie musi zmierzyć się ze strażnikiem, jednak jest zbyt słaby, dlatego w poszukiwaniu mocy zwraca się o pomoc do starca. Ten zleca zebranie przedmiotów wymaganych do ulepszenia broni. Podczas wyprawy graczu przyjdzie zmierzyć się z dwoma innymi bossami lokacji (najmocniejsi jednorazowi przeciwnicy) w celu zdobycia potrzebnych rzeczy, by otrzymać wzmacnienie i ruszyć na ostatniego przeciwnika. Wygrana otworzy przejście, które przeładuje scenę do napisów końcowych.
- **Wizja końcowa** – bohater budzi się w jaskini, jest完全nie zagubiony. Na stołku przy ognisku widzi starca, z którym postanawia porozmawiać. Dialog przedstawi pierwsze mechaniki i przybliży sytuację gracza, taki tutorial na wstępie. Dodatkowo do ekwipunku trafi strzelba jako broń główna, tajemnicza skrzynka, którą należy dostarczyć do miasta w ramach odwdzięczenia się staruszkowi i jabłko regenerujące jedno z trzech punktów zdrowia. Gracz dowie, że nie ten jegomość go znalazł, a ktoś z imieniem Sally. Po wyjściu z jaskini, idąc w lewo bohater natknie się na zamknięte

drzwi do katakumb, zaś kierując się w stronę góρ (tj. w prawo), gracz spotka wnuczkę starca o wcześniejszej wspomnianym imieniu. Z przeprowadzonego dialog wyniknie, że to ona go znalazła i zaciągnęła do jaskini, więc w ramach wdzięczności graczowi przyjdzie wykonać kolejną przysługę, a mianowicie odnalezienie kilku przedmiotów do naprawy maszyny pozwalającej dostać się do podniebnego miasta. Idąc dalej gracz trafi na masywne. Okażą się zamknięte, a próba otworzenia ich z użyciem pradawnych inskrypcji jedynie zbudzi śpiącego strażnika. Bez walki się nie obejdzie. Bohater jest jednak zbyt słaby, dlatego w poszukiwaniu mocy zwraca się o pomoc do starca. Ten zleca zebranie przedmiotów wymaganych do ulepszenia broni, gdzie jednym z nich będzie katalizator, który posiada jego wnuczka. Podczas wyprawy graczowi przyjdzie zmierzyć się z dwoma innymi bossami (najmocniejsi, jednorazowi przeciwnicy w danej lokacji) w celu zdobycia potrzebnych rzeczy. Zadanie od Sally jest niezbędne do wykonania, gdyż część maszyny stanowi agregat prądotwórczy, zasilający również drzwi do katakumb. Tam czeka pierwszy boss „*Giga Slime*”. Nim jednak przejście stanie otworem, w drodze między zleceniodawczynią, a wrotami ze strażnikiem znajdzie się most. Gracz zdobędzie przy nim maskę przeciwigazową. Ta część ekwipunku jest niezbędna, gdyż na dnie wąwozu, w radioaktywnych chemikaliach, stoi rozbita cysterna, to też bez wyekwipowanej maski, bohater szybko zginie od toksycznych oparów. Na pojeździe znajdą się narzędzia potrzebne do naprawy generatora oraz karta dostępu do katakumb. Drugim elementem niezbędnym do napraw jest układ scalony, będący gdzieś we wspomnianej przed chwilą lokacji, tam zostaną odnalezione rakietowe buty pozwalające na drugi, wyższy skok oraz szarżę. Będą one niezbędne do pokonania bossa, wyjścia z pułapki katakumb oraz zdobycia płytka do urządzienia Sally. Po destrukcji przeciwnika lokacji, do ekwipunku trafi toksyna, czyli kolejny ze składników. Dostarczenie układu jest równoznaczne z naprawą wyrzutni, to też gracz uzyska dostęp do nowej lokacji. Podniebne miasto stworzone zostało z myślą o ukochanej autora, dlatego przeciwnik końcowy będzie nietypowym bossem. Całość utrzymana jest w wizji zrujnowanego miasta w chmurach, posiadającego liczne jeziora, strzeżone przez wodniki, czyli kule wody lecące w kierunku bohatera. Po trafieniu odbierają punkt zdrowia. Sam boss to stworzony z magii duszy strażnik dla ukochanej budowniczego, to też posiada on, a raczej ona część cech swej pani. Zwrotem akcji okaże się brak walki, ponieważ przeciwnik skojarzy postać gracza z dawnym mistrzem i się zauroczy. Przeprowadzony w pacyfistyczny sposób dialog doprowadzi do zyskania ostatniego składnika

przeznaczonego do stworzenia wzmacnienia strzelby. Zostaje teraz wrócić do staruszka, wrzucić wszystkie składniki do kotła na stole alchemicznym i wyekwipować nowo powstały przedmiot. Pociski zmienią kolor na zielony, zaczną zadawać czterokrotnie wyższe obrażenia, a ostatni przeciwnik stanie się wrażliwy na ataki. Wygrane starcie odblokuje wrota, które doprowadzą do napisów końcowych. Urozmaiceniem gry będzie możliwość jej przejścia już po kilku minutach w ramach ukrytego zakończenia. Po sprawdzeniu wrót, a przed obudzeniem strażnika, gracz może udać się do staruszka, który podpowie jak je otworzyć. Wtedy też zadanie z pomocą w naprawie wyrzutni stanie się poboczne.

### 3.7 Krok 7: Non-playable character (NPC).

We wcześniejszym punkcie przedstawiono zarys fabularny, z którego wynika, że podczas przygody bohater gry spotka trzech npc (non-playable character). Postać będąca w świecie gry, nie posiadająca żadnych funkcji, mija się z celem, dlatego też spotkane osoby zaoferują dialogi oraz zadania [19].

- „*Old Man*”:
  - **Projekt postaci** – starszy mężczyzna (nawiązanie do gry „*The Legend of Zelda*” oraz „*Deponia*”) z bujnym wąsem oraz brwiami, siedzący na stołku przy ognisku i palący fajkę. Dawniej biolog w pobliskim mieście, teraz pustelnik badający mutacje fauny i flory po eksplozji osobliwej materii. Z natury dobra dusza, ale walczy o swoje. Pomaga mu jego wnuczka Sally.
  - **Dialogi** – staruszek stanowiący pierwszy kontakt ze światem gry. Podczas dialogu, okaże się, że bohater nie jest w stanie mówić. Nie stanowi to większego problemu więc brnąc dalej, gracz pozna podstawowe sterowanie oraz mechaniki, otrzyma główne zadanie, jakim będzie dostarczenie paczki do karczmy „*Miś kudłacz*” oraz broń towarzyszącą przez resztę gry. Przez rozmowę podjętą po pewnym czasie, gracz zyska jabłko przywracające jeden punkt zdrowia. Kolejna opcja dialogowa dotyczy nadania zadania prowadzącego do stworzenia modyfikacji do broni, w celu wyrównania szans ze strażnikiem bramy. Dodatkowo umieszczono dialog sekretnego zakończenia, wystarczy sprawdzić wrota i nie budzić strażnika.
- „*Sally*”:

- **Projekt postaci** – rudowłosa dziewczyna, z zamiłowania mechanik (nawiązanie do gry „*Deponia*”), która podobnie jak jej dziadek (postać wyżej) bada okolicę, jednak w poszukiwaniu starej technologii z przed eksplozji. Aktualnie prowadzi prace nad starą wyrzutnią do miasta w chmurach „*Sky Town*”.
  - **Dialogi** – podczas rozmowy okazuje się, że dzięki niej bohater wciąż żyje, gdyż spadł z nieba i leżał nieprzytomny. Zaciągnęła go do dziadka, który udzielił odpowiedniej opieki medycznej. Kolejno w ramach wdzięczności zleci graczowi odnalezienie brakujących rzeczy pozwalających na naprawę wyrzutni prowadzącej do lokacji z ruinami podniebnego miasta. Zdobycie narzędzi doprowadzi do naprawy generatora, a przyniesienie układu scalonego posłuży do ukończenia remontu całej maszyny.
- „*Yamiko*”:
- **Projekt postaci** – początkowo miał być to boss w „*Sky Town*” stworzony na podstawie mej ukochanej, jednak dla nadania zwrotu akcji przeciwnik nie będzie atakował, tylko rozpoczęcie dialog. Sama postać to strażnik, którego ciało tworzy woda, powstały z magii duszy, dzięki czemu przejął część osobowości swej pani. Zapewni to jego wierność po wsze czasy. Eksplozja osobliwej materii zrujnowała miasto i zabiła wszystkich mieszkańców, zaś w miejscu spoczynku założycieli wyrósł potężny kwiat zbierający esencję słońca. Strażnik po stracie swych właścicieli, zamiast zginąć z nieznanego powodu został przy życiu, chroniąc pozostałości fortecy.
  - **Dialogi** – w trakcie rozmowy wynika, że postać strażnika widzi w bohaterze gry znajomą osobę, co prowadzi do zauroczenia i rezygnacji z walki. Tu po raz pierwszy bohater ma możliwość wypowiadania własnych słów, gdyż z niewyjasnionych przyczyn odzyskał tę umiejętność. W trakcie kolejnych linii nastąpi podsumowanie dotychczasowej przygody, po czym padnie pytanie o ostatni składnik jakim jest płatek z kwiatu przepełnionej esencją. Początkowo npc zareaguje agresywnie jednak z ostatecznie podaruje graczowi potrzebny przedmiot.

### **3.8 Krok 8: System dialogów i quest'ów.**

Podczas tworzenia należy zastanowić się nad możliwością zmiany języka. W tym projekcie jest taka opcja, co oznacza, że dialogi powinny zostać wczytane z odpowiednich plików. Taki sposób umożliwi łatwe dodawanie tłumaczeń. Ułatwia to również pracę przy braku szczegółowego scenariusza, gdyż pozwoli na bieżąco wprowadzać zmiany w dialogach dla uzyskania finalnej wersji. Problem zacznie się w momencie, gdzie zmiana tekstów przez np. dodanie linii dialogowej wpłynie na sztywno ustawione reakcje po stronie kodu. Na to jednak też jest sposób choć dość ryzykowny. Stworzony zostanie mechanizm wykonujący działania na podstawie pierwszych trzech parametrów zawartych w każdej linii dialogowej, oddzielonych znakiem pionowej kreski, zaś ostatni, czwarty będzie tekstem do wyświetlenia. Dzięki temu podejściu wyeliminowana zostanie konieczność stworzenia wielu indywidualnych skryptów dla różnych sytuacji z dialogiem.

Zadania nadane przez spotkanych npc w rzeczywistości nie muszą trafić do specjalnego dziennika zawierającego wszelkie informacje. Wykorzystując wyżej wspomnianą mechanikę odczytywania dialogów, możliwym stanie się skakanie po liniach tekstu. Przykładowo jedna z postaci posiada zadanie, więc odczyta listę zdarzeń zawartą w obiekcie „*Config*”, by sprawdzić czy zostało wykonane. Zakładając, że nie, poprosi o przyniesienie jakiegoś przedmiotu, np. jabłka. Po powrocie posiadając wymagany przedmiot w ekwipunku, npc w pierwszej kolejności sprawdzi to samo co poprzednio, a następnie czy gracz nie posiada czegoś co spełni warunki zadania. Widząc, że posiada przejdzie do kwestii związanej ze zleceniem, zaś w innym wypadku poruszy kolejną kwestię, bądź poinformuje grającego o lokalizacji owego owocu. Dodawanie oraz usuwanie przedmiotów (najczęściej fabularnych) również wskazane będzie w dialogu. By nieco bardziej rozjaśnić możliwości takiego mechanizmu, realizowane będzie wszystko to co zaprogramuje twórca i sterowane z użyciem jedynie kilku parametrów z poziomu tekstu. Wracając do ryzyka: użytkownik, który rozgryzie ten system może swobodnie modyfikować sporą część linii fabularnej gry oraz dodawać nowe zadania.

### **3.9 Krok 9: Ekwipunek oraz przedmioty.**

Gra przygodowa bez zbierania przedmiotów w celu późniejszego zastosowania, jest niczym komedia bez dobrego humoru. Posiadanie wielu ciekawych rzeczy, zmienia charakter rozgrywki, chociażby przez takie banały jak odzyskanie zdrowia po wypiciu

eliksiru, czy wzmacnienie dla bronи. „*Last Adventure*” zakłada nabycie nowych umiejętności oraz elementów ekwipunku w celu odblokowania niedostępnej wcześniej dla bohatera gry drogi.

- **Forma wizualna** – za ekwipunek posłuży obiekt, będący częścią gracza zawierający w sobie równo rozłożone podobiekty, z których szesnaście posłuży za pola ekwipunku, cztery za miejsca do wyekwipowania i jedno za wskaźnik na obecnie zaznaczone pole. W momencie najechania wskaźnikiem na przedmiot, okno dialogowe wyświetli nazwę, obraz oraz opis.
- **Zarządzanie** – skrypt zarządzający ekwipunkiem z poziomu kodu, wyłączy możliwość poruszania postacią i używa przeznaczonych do tego przycisków.
- **Skrypt** – forma wizualna jedynie wyświetla przedmioty, zaś faktyczna informacja o stanie ekwipunku znajdzie się we wcześniej wspomnianym obiekcie Config. Posłużą do tego dwie listy z nazwami przedmiotów, jedna dla wyekwipowanych, druga dla niewyekwipowanych. Taka forma pozwoli wykonać zapis w postaci łańcucha znaków, czyli tekstu dzielonego z pomocą znaku klucza, przykładowo pionowej kreski.
- **Przedmioty** – na jeden przedmiot przypadną dwa elementy, obiekt właściwy w świecie gry oraz ten wyświetlany w ekwipunku. Oba będą zawierały takie informacje jak nazwa pliku z grafiką (dla wyświetlania w oknie dialogowym), nazwę przedmiotu, opis, typ (np. przedmiot fabularny) oraz niepowtarzalny identyfikator. Elementy w świecie gry, trafią do obiektu rodzica, od którego przyjmą lokalne przesunięcie w przestrzeni. Prawie wszystkie możliwe do zebrania rzeczy, będą jednorazowe, co oznacza, iż po ich podniesieniu, do listy w Config'u dopisane zostanie id przedmiotu. Po przeładowaniu sceny (np. zmieniając lokację) obiekt, którego identyfikator pojawił się na liście ulegnie autodestrukcji.

Sama gra nie będzie należała do długich, ponieważ stanowi na chwilę obecną jedynie projekt laboratoryjny, to też pominięty zostanie element wyrzucania przedmiotu z ekwipunku. Oszczędzi to dodatkowej pracy w postaci zapisywania informacji o wyrzuconych przedmiotach na konkretnych mapach, z konkretnymi współrzędnymi. Alternatywą może być skrzynia rodem ze starszych horrorów takich jak „*Residen Evil*”, która przechowywała wszelkie przedmioty w momencie przepelenienia ekwipunku.

### **3.10 Krok 10: System walki, przeciwnicy i bossowie.**

Jedno z założeń projektu to zmieszanie tematyki fantastycznej z fantastyczno-naukową, tak więc broń którą dzierży gracz jest strzelbą. Wymusza to zaprojektowanie walki dystansowej bazującej na uniku i ataku podkręcając w ten sposób powolną część przygodową do tempa akcji. Naturalnie wykonanie strzału odbędzie się wyłącznie po wyekwipowaniu odpowiedniego przedmiotu, natomiast sama strzelba nie będzie ograniczana ilością posiadanych pocisków. Wystrzelenie kuli rzeczywistości oznacza utworzenie obiektu z gotowego prefabrykatu posiadającego własny skrypt na przemieszczenie w czasie i przestrzeni. Kontakt z przeciwnikiem doprowadza do małej eksplozji zadając obrażenia [10] [20].

Przeciwnik – w przypadku utworów opartych o wyobraźnię autora, dosłownie wszystko może stać się czymś wrogim, zagrożeniem czyhającym na gracza w najmniej oczekiwany momencie, jak pozornie niewinne, ale jednak „*Krwiożercze donaty*” (o dziwo istnieje taki film). Nie wybiegając jednak tak daleko w przedziwny świat fantazji, do gry trafią [18]:

- „*Slime'y*” – popularne galaretki występujące w niejednej grze z gatunku RPG jako pierwsze, zazwyczaj niezbyt wymagające. Tu jednak nieco zmieni się ich natura, gdyż niesprowokowane zachowają neutralność wobec gracza, jednak zaatakowany wypuści z siebie chmurę gazu, po czym zacznie tworzyć mniejsze wersje służąca za pełzające do bohatera bomby.
- „*Duchy*” – małe stworki zamieszkujące mroczny las swobodnie latające sobie między drzewami. Przeważnie znikają z pola widzenia gracza, by zaraz pojawić się za jego plecami i eksplodować zadając obrażenia. Na nic zda się wszelka broń, gdyż przeniknie przez ich niematerialne ciała, jest jednak na nie sposób. Panicznie boją wzroku osób żywych to też wystarczy z nimi stanąć oko w oko, by zaraz uciekły.
- „*Wodniki*” – stworzenie będące żywiołem wody. W momencie gdy gracz się zbliży, materializuje swą formę, by zaraz potem wystrzelić w jego stronę. Bezpośredni kontakt z bohaterem oznacza autodestrukcję i zadanie odebranie punktu zdrowia.

Każdy z przeciwników posiada niewielki skrypt monitorujący otoczenie i reagujący w odpowiedni sposób na pozycje gracza z wykorzystaniem warunków oraz przełączników.

- „*Giga Slime*” – przeciwnik końcowy będący modyfikacją slime'a, mający w sobie toksynę, będącą elementem niezbędnym do stworzenia modyfikacji broni. Skrypt sterujący bossem, podobnie jak wcześniejsi wrogowie oparty jest na śledzeniu otoczenia oraz zachowania gracza w celu dostosowania własnych akcji. Atakami będzie tworzenie bomb pełzających do gracza oraz toksycznych pocisków i gazów.
- „*Yamiko*” – ze względu na odmienny charakter tego przeciwnika, trafił do podpunktu o NPC.
- „*Golem*” – strażnik wielkich wrót, a zarazem drogi prze góry prowadzącej do miasta. Ostatni boss w grze do pokonania, którego niezbędny będzie ekwipunek zdobyty podczas przygody. Jego ruchy w przeciwieństwie do pozostałych przeciwników wykonane zostaną z pomocą nie animacji pokaltkowej, a przez modyfikacje parametrów w komponentach. Skrypt z zachowaniem nie odbiega od wcześniejszych założeń.

### **3.11 Krok 11: Napisy końcowe.**

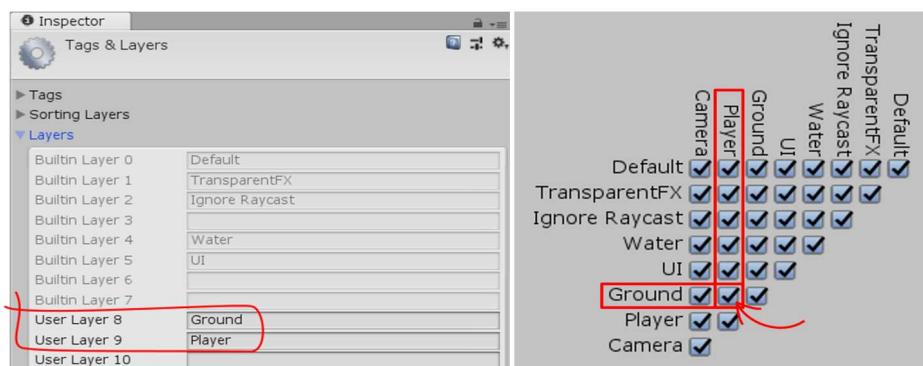
Ostatnia scena bazować będzie na menu głównym z adekwatną zmianą zawartości. Ogólna wizja wygląda następująco: Pojawi się tytuł gry wraz z wyeksponowanym przez biel podtytułem, po czym zacznie sunąć w górę. Wraz z nim przemieszczą się napisy z podziękowaniem, twórcą gry, promotorem pracy oraz twórcami ścieżek dźwiękowych. Całość będzie możliwa do przyśpieszenia przez przytrzymanie odpowiedniego przycisku, a dotarcie do końca wywoła menu główne.

## 4. Z teorii do praktyki

Faktyczny proces tworzenia nie zawsze jest tak prosty jak można to ująć w teorii. Nie raz niezbędne stanie się kombinowanie z realizacją pomysłu, by wybrać najlepszą/najwydajniejszą metodę. Część opisowa z rozdziału poprzedniego jest swego rodzaju mapą, dzięki której łatwiej będzie odnaleźć się w projekcie.

### 4.1 Pierwsza scena, pierwsza postać, pierwszy skrypt (krok 1).

Na początek wymagane będzie wprowadzenie kilku podstawowych rzeczy. Wraz z nowym projektem Unity tworzy nową scenę z dodając przy okazji kamerę o domyślnych ustawieniach, jednak nie dodaje podkatalogów niezbędnych w dalszej części projektu. Z czasem przyjdzie potrzeba na uzyskiwanie zasobów gry z poziomu kodu przez wskazanie miejsca oraz nazwy, dlatego też w katalogu głównym „Assets” umieścił folder o narzuconej nazwie „Resources”. Kolejny taki katalog to „StreamingAssets”, dzięki któremu po zbudowaniu gry zawarte w nim pliki nie zostaną spakowane do paczki rozpoznawanej przez silnik, ale widoczne zostaną w swej oryginalnej formie. Każda scena powinna być gdzieś zapisana, w związku z tym, w „Resources” dodano katalog „Scenes”, do którego zapisano obecnie otworzoną nadając nazwę „TestMap”.

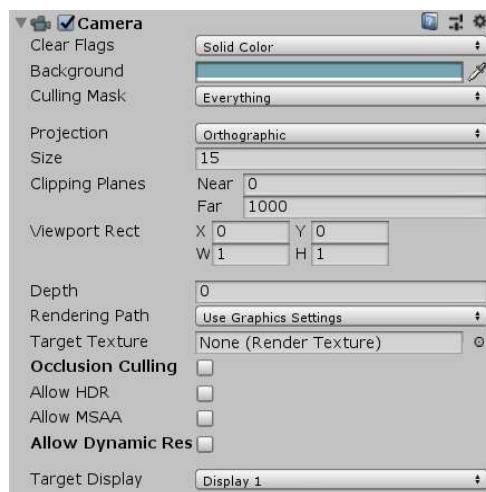


Rysunek 18 - Dodawanie nowych warstw oraz ustawienie relacji w Physic 2D.

Dalej, bazując na kolejności działań z rozdziału trzeciego, pierwszą podjętą czynnością jest utworzenie dwóch dodatkowych warstw o nazwach „Player” oraz „Ground”, a dokonano tego przez wybranie z menu głównego „Edit” >> „Project Settings” >> „Tags & Layers” oraz rozwinięcia listy o nazwie „Layers”. W wolnych polach wpisano wybrane słowa, co sprawiło, iż są teraz widoczne w menu rozwijanym z poziomu inspektora obiektów. Oddziaływanie między warstwami ustalono po wejściu w „Edit” >>

„Project Settings” >> „Physics 2D” po rozwinięciu zakładki „Layer Collision Matrix” co widać załączonym rysunku 18.

Kolejny krok wymaga dostosowania kamery. W tym celu stworzono nowy obiekt o nazwie „MainCamera”, któremu przypisano pozycję globalną na osiach równą zero, po czym umieszczono w nim istniejącą kamerę ze zmienioną nazwą na „Camera” i przypisano jak wcześniej pozycję zerową z tą różnicą, iż jest ona teraz lokalna. Jak ustalono w części teoretycznej, obiekt definiując zawarte w nim komponenty, tak więc w rzeczywistości modyfikacja ustawień kamery nastąpi przez zmiany wprowadzone z pomocą inspektora do odpowiedniego komponentu. Kolejno jak widać na rysunku 19, odłączono zbędne efekty, rozmiar widoku ustawiono na 15, a projekcję na „Orthographic” [8] [11]. Parametr „Culling Mask” odpowiada za wyświetlane warstwy, dlatego ustawienie „Everything” pozwoli renderować wszystko co znajdzie się na scenie.

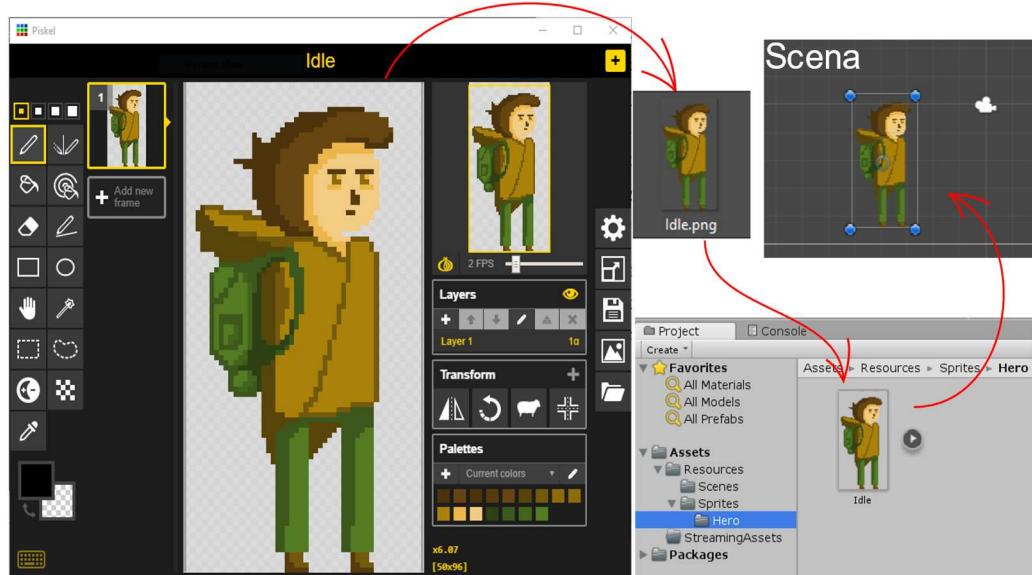


Rysunek 19 - Ustawienia kamery.

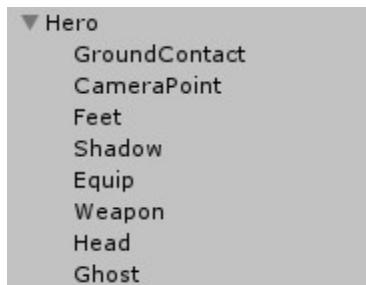
#### ❖ Bohater gry:

Bohater gry dla ludzkiego oka to jedynie ruszająca się grafika, jednak w świecie gry jest on obiektem złożonym z kilku innych reagującym na otoczenie. Z pomocą Piskel'a przygotowano i wyeksportowano do formatu png prosty obraz, mającą posłużyć za obiekt główny (rodzica). Dla utrzymania porządku w zasobach projektu (z czasem zimportowana zostanie masa elementów) utorzono podkatalog „Sprites” w „Resources”, do którego trafią wszelkie grafiki oraz animacje z nimi związane. Dodatkowo dodano kolejny podkatalog, tym razem wewnętrz nowo utworzonego z nazwą „Hero”, po czym dopiero tam zimportowano grafikę bohatera, a następnie przeniesiono ją na scenę. Ten sposób umieszczenia zasobu utworzy w hierarchii projektu

nowy obiekt o nazwie grafiki, to też zmienić ją należy z pomocą inspektora lub przez kliknięcie przycisku F2. Domyślne parametry zaimportowanego obrazka sprawią, że przy skalowaniu będzie on rozmazany. Zapobiegnie temu zmiana opcji „Filter Mode” z „Bilinear” na „Point (no filter)” oraz zmiana kompresji na „None”. Powstała postać nadal jest niekompletna, tak więc podążając wyznacznikami z poprzedniego rozdziału, dodano nowe obiekty otrzymując efekt widoczny na rysunku 21 [15] [19].



Rysunek 21 - Dodanie nowego elementu przez wstawienie grafiki na scenę.



Rysunek 20 - Budowa bohatera w zakładce hierarchii.

- „Hero” – obiekt rodzic zawierający w sobie komponenty odpowiedzialne za fizykę, animacje, kolizje oraz sterowanie, czyli: „Rigidbody 2D”, „Box Collider 2D”, „Sprite Renderer”, „Animator” oraz skrypt kontroli. Ustawiona zostanie również warstwa na „Player”.
- „GroundContact” – pusty element umieszczony w dolnej części postaci na podstwiaie, którego sprawdzony zostanie kontakt warstwy „Player” z „Ground”.
- „CameraPoint” – pusty obiekt służący kamarze jako punkt śledzenia postaci.
- „Feet” – z komponentem „Audio Source”, do którego trafi nagranie imitujące bieg.

- „Shadow” – z dodaniem wcześniej przygotowanej grafiki gradientu w formie koła, z ustawionym kolorem czarnym, przezroczystością oraz odpowiednią skalą.
- „Equip” – na chwilę obecną pusty.
- „Weapon” – do którego trafi obiekt z rękoma lub bronią.
- „Head” – domyślnie pusty, do którego trafi maska przeciwgazowa przysłaniająca część twarzy.
- „Ghost” – z komponentami „Sprite Renderer” oraz „Animator”. Niewidoczna grafika ducha pojawi się w momencie śmierci bohatera, unosząc się nad nim.

Komponent odpowiedzialny za fizykę zapewni bohaterowi wieczne spadanie za sprawą parametru grawitacji. Oczywiście rozwiązaniem jest dodanie podłoża o warstwie „Ground” oraz kolizji z użyciem odpowiednio rozciągniętego „Box Collider 2D”. Komponent kolizji zawarty w bohaterze również należy dopasować rozmiarem, by uzyskać w miarę naturalny efekt. Aktualnie postać zatrzyma się na podłożu i zacznie się chwiać na boki, co zdecydowanie nie jest pożądanym efektem. Rozwiązaniem jest zaznaczenie parametru „Freeze Rotation Z” w „Rigidbody 2D”.

Mając podstawową budowę bohatera gry oraz podłożę, po którym będzie stąpał, można przejść do pisania skryptu dającego graczowi kontrolę nad postacią. Stworzono nowy folder w „Resources” o nazwie „Scripts”, w nim zaś podkatalog „Controllers”. Do niego trafią wszelkie skrypty służące do zarządzania bohaterem, przeciwnikami i npc. Dodano zatem nowy skrypt C# nadając mu nazwę „ControllerPlayer”, co ułatwi późniejsze odnajdywanie innych kontrolerów. Uwaga! Przy zmianie nazwy skryptów, należy pamiętać o zmianie nazwy klasy wewnętrznie niego na taką samą, w przeciwnym razie Unity będzie wyświetlał błęd. Przypisanie skryptu do bohatera to nic innego jak dodanie go jako komponent obiektu. Domyślnie nowy skrypt zawiera w sobie metody „Start” oraz „Update”, w związku z czym należy dodać jeszcze dwie „ FixedUpdate ” i „ OnTriggerEnter2D ”. Są to podstawy do pracy z kodem, gdyż w starcie ustawione zostaną wszelkie odwołania do obiektów na scenie, update zapewni odświeżanie skryptu z prędkością na jaką pozwoli konfiguracja sprzętowa, „ FixedUpdate ” działa jak zwykły update z tą różnicą, iż ma stały czas między cyklami, dzięki czemu umieszczone w nim elementy będą odświeżane z identyczną prędkością na każdej maszynie. Ostatnia metoda wyłapie kolizję, którą następnie można oprogramować. Oczywiście poza tą czwórką jest opcja tworzenia własnych funkcji niewymaganych przez Unity.

❖ Movement – sterowanie postacią:

Poruszanie postacią zrealizowane zostało przez modyfikację wektorów obiektu posiadającego komponent „*Rigidbody 2D*”. Jak najbardziej można wykorzystać „*Transform*”, jednak dzięki nadanej fizyce bohater napotkawszy przeszkodę przestanie się przemieszczać (ruch będzie przyblokowany), gdzie przy zmianie pozycji na osiach, dodatkowo wymaganym byłoby sprawdzenie ewentualnej kolizji gracza z otoczeniem (inaczej przeniknie przez przeszkodę). Dla ułatwienia operowania kodem utworzono prywatną zmienną o typie „*Rigidbody2D*” z nazwą „*body*”, po czym w metodzie „*Start*” przypisano do niej adekwatny komponent rodzica. Dodatkowo umieszczone zostaną „*moveSpeed*” oraz „*move*” typu zmiennoprzecinkowego „*float*”, gdzie pierwszy stanowi o prędkości poruszania z przypisaną wartością 14, zaś drugi w zależności od kierunku ruchu przyjmie oryginalną wartość z „*moveSpeed*”, bądź przemnożoną przez -1. Kod do tej części zamieszczony zostanie w „ *FixedUpdate*” i wykorzysta menedżera wejść. Sprawdzane będzie horyzontalne (poziome) odchylenie gałki pada, gdzie wartości mniejsze od -0.4 (40% wychylenia w lewo) nadadzą zmiennej „*move*” wartość zmiennej „*moveSpeed \* -1*”. Podobnie przy wartości przeciwej, czyli 0.4 (40% wychylenia w prawo), lecz „*move*” przyjmie oryginalny „*moveSpeed*”. W przypadku użycia klawiatury, przycisk „*A*” lub „*Left*” symuluje wartość -1, zaś „*D*” lub „*Right*” +1, czyli maksymalne wychylenia. Ostatecznie magia przemieszczenia wykonuje się wraz z linią przypisującą do „*Rigidbody 2D*” nowy wektor stworzony na podstawie zmiennej „*move*” podanej jako x oraz z zachowaniem oryginalnego y. Ze strony praktycznej wygląda to następująco [9] [21] [22]:

[ ControllerPlayer.cs ]

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ControllerPlayer : MonoBehaviour {
    private Rigidbody2D body;
    private float moveSpeed = 14;
    private float move;

    void Start () {
        /* Dzięki zmiennej „body” nie trzeba pisać za każdym razem
        GetComponent<Rigidbody2D>() by dostać się do parametrów komponentu,
        wystarczy nazwa zmiennej */
        body = GetComponent<Rigidbody2D>();
    }

    void Update(){ // Chwilowo niepotrzebny }

    private void FixedUpdate(){
        if (Input.GetAxis("Horizontal") < 0)
            move = moveSpeed * -1;
        else if (Input.GetAxis("Horizontal") > 0)
            move = moveSpeed;
        else
            move = 0;
        body.velocity = new Vector2(move, body.velocity.y);
    }
}
```

```

/* Ustalenie wykonania oraz kierunku ruchu z użyciem menadżera wejść tj.
Input.GetAxisRaw("Horizontal") */
    if (Input.GetAxisRaw("Horizontal") > 0.4) move = moveSpeed;
    else if (Input.GetAxisRaw("Horizontal") < -0.4) move = moveSpeed*-1;
    else move = 0;

    // Wykonanie przesunięcia przez nadanie pędu
    body.velocity = new Vector2(move, body.velocity.y);
}
}

```

Teraz postać jest w stanie poruszać się po wcześniej przygotowanej powierzchni zarówno w prawo jak i w lewo. Niech zatem jeszcze skacze tym wyżej im dłużej przycisk zostanie przytrzymyany. Na początek należy zrealizować wykrywanie kontaktu warstwy gracza z podłożem, w związku z czym dopisane zostały kolejne zmienne prywatne określające położenie punktu kontaktu z ziemią, przechowującą warstwę „Ground”, ustalające promień w obrębie którego sprawdzany będzie kontakt z podłożem oraz finalna, do której trafi true lub false. Wygląda to następująco:

[ ControllerPlayer.cs ]

```

private Transform groundContact;
private LayerMask groundLayer;
private float    groundRadius = 0.2f;
private bool     grounded    = false;

void Start () {
    ...
    /* Podobna sytuacja jak z "body", jednak nie jest to nawiązanie do całego
    obiektu, a do jego komponentu */
    groundContact = GameObject.Find("Hero/GroundContact").gameObject.transform;
    groundLayer   = LayerMask.GetMask("Ground");
}

void Update(){
    /* funkcja „OverlapCircle” sprawdzi czy obszar wokół punktu kontaktowego
    gracz-podłoże nachodzi na warstwę obiekt o warstwie „Ground”, w wyniku czego
    zwróci true jeśli jest nachodzi lub false jeśli nie */
    grounded = Physics2D.OverlapCircle(groundContact.position, groundRadius,
                                     groundLayer);
}

```

Od teraz zmienna „grounded” będzie jednym ze źródeł informacji o możliwości skoku bohatera. Kolejnym jest czas, a konkretnie długość przebywania w powietrzu, tj. okres bez kontaktu z podłożem. W tym celu niezbędne będą trzy zmienne prywatne typu float: „jumpForce” przechowujący siłę skoku, „jumpTime” określający maksymalny czas przytrzymania przycisku oraz „jumpClock” odliczający pozostały czas z użyciem wartości „Time.deltaTime”. W momencie wylądowania, wykryty zostanie kontakt z podłożem, co doprowadzi do zresetowania czasu skoku. Odbędzie się to po stronie „Update”, zaś odliczanie wzbijania się bohatera umieszczone w „ FixedUpdate ”, aby zachować jednakowe odstępy czasu. Samo wcisnięcie przycisku będzie reagowało jedynie w

momencie, gdy czas wzbijania jest większy od zera. Niewielkim problemem stanie się wielokrotne wciskanie klawisza, które wywoła drgającą lewitację postaci. Sposobem na to jest wyzerowanie czasu skoku w momencie zwolnienia [23]. W praktyce wygląda to następująco:

[ ControllerPlayer.cs ]

```
...
private float jumpForce = 15;
private float jumpTime = 0.3f;
private float jumpClock = 0.3f;
void Update(){
    ...
    // Reset czasu w momencie kontaktu z podłożem
    if (grounded) jumpClock = jumpTime;

    /* Zwiększenie wektora w osi y przez przytrzymanie przycisku do momentu
    końca czasu... */
    if (Input.GetKeyDown(KeyCode.Space) && jumpClock > 0)
    {
        body.velocity = Vector2.up * jumpForce;
    }

    // ... bądź zwolnienia przycisku, zerując pozostały czas
    if(Input.GetKeyUp(KeyCode.Space)){
        jumpClock = 0;
    }
}
private void FixedUpdate()
{
    // Jeśli nie występuje kontakt z podłożem, odejmuj stały czas co odświeżenie
    if (!grounded) jumpClock -= Time.deltaTime;
    ...
}
```

W ramach zakończenia kroku pierwszego, zostanie wykonany prosty zabieg odwracający grafikę postaci do kierunku marszu. Jest to dość prosta sztuczka polegająca na odwróceniu lokalnego skalowania parametru x w komponencie „Transform”. Obraz umieszczony na scenie to sylwetka bohatera gry stojącego z twarzą zwróconą w prawo, czyli domyślnie skala będzie dodatnia. Po sprawdzeniu zmiennej „move” oraz obecnej skali można podjąć decyzję o jej zmianie, to też ruch większy od zera wraz ze skalą x ujemną powinien wywołać funkcję, która wykona przekształcenia. Tak samo w przypadku, gdy ruch jest mniejszy od zera, a skala większa. Z poziomu kodu przedstawia się to następująco:

[ ControllerPlayer.cs ]

```
private void FixedUpdate()
{
    ...
    // Ruch większy od zera, a skala mniejsza ...
    if (move > 0 && transform.localScale.x < 0)
        Direction();
    // ... lub ruch mniejszy od zera, a skala większa
```

```

    else if (move < 0 && transform.localScale.x > 0)
        Direction();
    }

    public void Direction()
    {
        /* Z braku możliwości bezpośredniej modyfikacji pojedynczego parametru
        nastąpi pobranie aktualnej skali do nowo utworzonej zmiennej, a następnie
        odwrócenie wartości za pomocą przemnożenia przez -1. Otrzymana w ten sposób
        nowa skala, zostanie przypisana do obiektu. */
        Vector3 newVector = transform.localScale;
        newVector.x *= -1;
        transform.localScale = newVector;
    }
}

```

❖ Menadżer wejść:

Tworząc skrypt poruszania wykorzystano wskazanie konkretnego przycisku dla skoku, co nie jest najlepszą metodą. Unity posiada narzędzie pozwalające ustalić jakie klawisze (nie tylko klawiatury) przypisane będą do wykonania danej akcji. Wykorzystano już w kodzie wychylenie gałki analogowej wraz z alternatywnymi przyciskami. Jest to jedno z domyślnie ustawionych wejść w Unity. By otworzyć „*Input Manager*” wystarczy wejść w „*Edit*” >> „*Project Settings*” >> „*Inputs*”, tam ukaże się lista gotowych, standardowych wejść, m.in. „*Jump*” z przyciskiem spacji. Zwiększając parametr „*Size*” nastąpi powielenie ostatniego wejścia, które swobodnie da się edytować. Tymczasem wprowadzono zmiany podmieniając sztywno przypisane klawisze na te z tabeli wejść.

[ ControllerPlayer.cs ]

```

void Update() {
    grounded = Physics2D.OverlapCircle(groundContact.position, groundRadius,
    groundLayer);
    if (grounded) jumpClock = jumpTime;
    if (Input.GetButton("Jump") && jumpClock > 0) {
        body.velocity = Vector2.up * jumpForce;
    }
    if (Input.GetButtonUp("Jump")) jumpClock = 0;
}

```

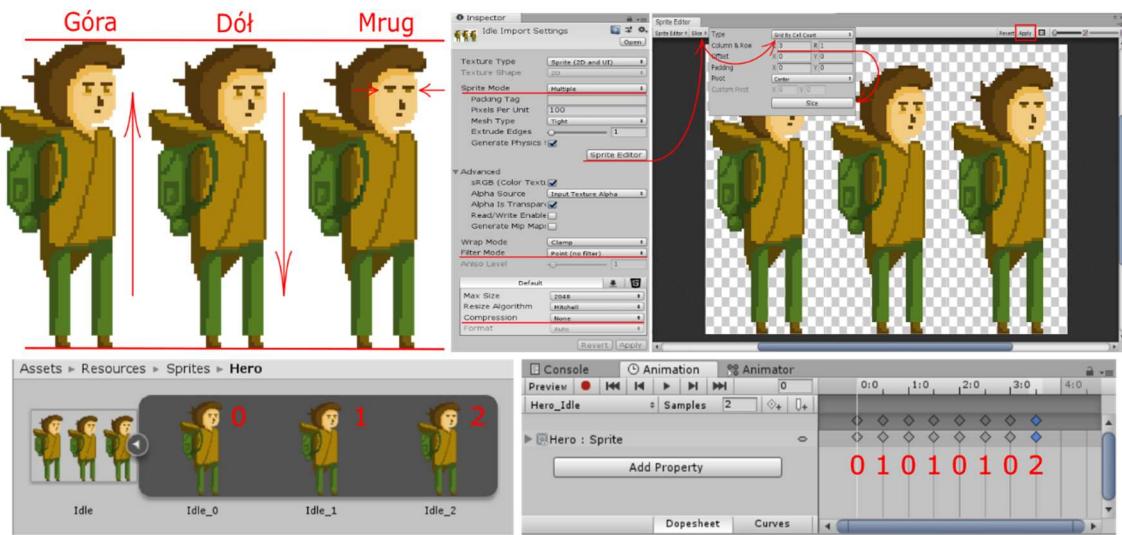
## 4.2 Poklatkowa animacja w animatorze (krok 2).

Kierując się planem z rozdziału trzeciego, utworzono zestaw grafik z animacją poklatkową. Z uwagi, iż jest to proces bardzo czasochłonny, ograniczono ilość klatek z zachowaniem dobrego efektu.

### ❖ Idle:

Ruch podczas bezczynności. Animator Unity pozwala na wielokrotne wstawianie klatek oraz dostosowywanie czasu animacji, to też z pomocą Piskel'a oraz wcześniej stworzonej pojedynczej klatki, dorysowano dwie dodatkowe, tak by zasymulować ruch torsu przy oddychaniu oraz mruganie oczu. Wyeksportowano je do jednego pliku png, by następnie zaimportować do folderu bohatera. Podobnie jak wcześniej ustalono w opcjach filtrowanie „*Point (no filter)*” oraz kompresje na „*None*”, by nie powstało rozmycie podczas skalowania. Rozdzielenie klatek będzie możliwe po ustawieniu parametru „*Sprite Mode*” na „*Multiple*” w opcjach importowanej grafiki i użyciu przycisku „*Sprite Editor*”. Jak widać na poniższym rysunku, przycisk „*Slice*” wyświetli okienko z parametrami dzielenia. Obraz posiada trzy klatki o jednakowym rozmiarze dlatego z listy rozwijanej wybrano „*Grid By Cell Count*”, po czym wskazano trzy kolumny u jeden wiersz. Pozostałe parametry pozostawiono domyślnie, wcisnięto przycisk dzielenia „*Slice*” i zapisano ustawienie przyciskiem „*Apply*” w górnej części okna. Plik w zasobach posiada teraz podgląd na rozzielone fragmenty. Dla uzyskania animacji postaci w świecie gry, dodano elementy „*Animation*” z nazwą „*Hero\_Idle*” oraz „*Animator Controller*” z nazwą „*Hero*”. Obiekt bohater wzbogacił się o komponent „*Animator*”, do którego podczepiono przed chwilą dodany kontroler. Zakładka animatora wyświetla aktualny stan zaznaczonego obiektu, dlatego należy upewnić się czy wybrano odpowiedni element z hierarchii. Tak samo działa edytor animacji. Nowo utworzona animacja jest pusta i nie widoczna w animatorze bohatera. W celu zmiany tego stanu rzeczy, zaznaczono obiekt główny bohatera, po czym do animatora przeniesiono plik animacji. Pojawi się kwadracik z nazwą pliku oraz automatycznie przypisana ścieżka od kafelka początkowego uruchamiająca plik. Po tych czynnościach, animacja widoczna jest w edytorze z ustawieniami domyślnymi, które zostały zmienione na potrzeby poklatkowej grafiki. Czas odtwarzania otrzymał wartość dwóch klatek na sekundę, na linię czasu umieszczono naprzemianie pierwszy i drugi fragment grafiki kończąc trzecim

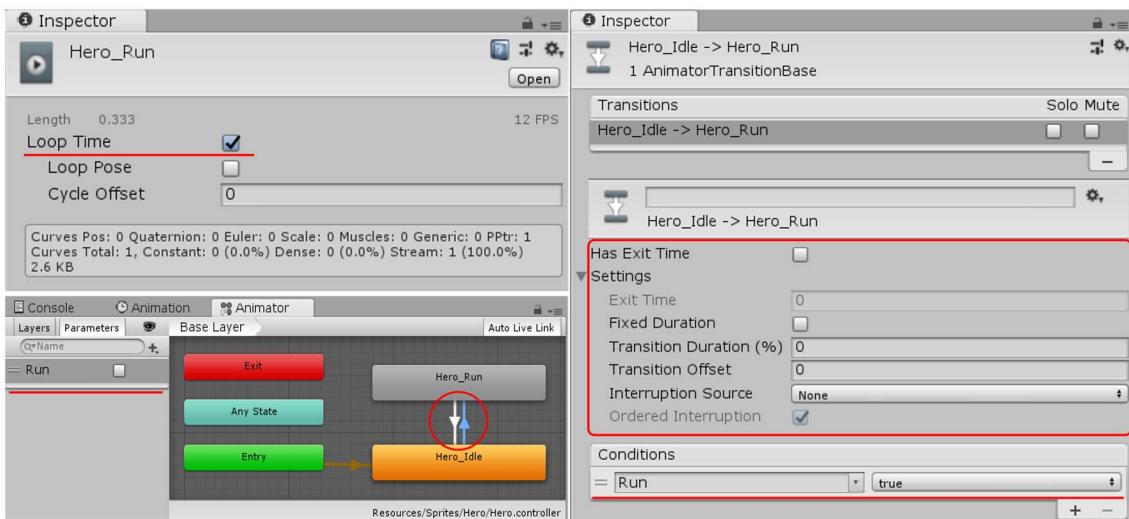
(osiem klatek), zaś samą animację zapętlono parametrem „Loop” ustawionym na true w inspektorze pliku z animacją [19].



Rysunek 22 - Animacja poklatkowa "Idle".

❖ Run:

Dodanie biegu jest niemalże identyczne jak we wcześniejszym opisie. Teraz jednak możliwe jest stworzenie nowej animacji przez wcisnięcie w edytorze animacji nazwy obecnie edytowanego pliku, gdyż rozwinię się lista aktualnie przypisanych animacji do animatora, zakończona opcją „Create New Clip...”. Tak jak wcześniej, przeniesione zostaną klatki ze wskazaniem prędkości odtwarzania (tym razem 12) oraz zapętleniem. Poostaje kwestia przełączania animacji na podstawie obecnie wykonywanych działań bohatera, do czego wykorzystano kontroler, czyli „Animator”. Kliknięcie prawego przycisku myszy na wybrany kafelek wyświetli menu podręczne z opcją „Make Transition”. Pojawi się strzałka prowadząca od miejsca kliknięcia, której należy wskazać animację



Rysunek 23 - Przejście animacji w animatorze Idle <-> Run.

docelową, czyli na chwilę obecną „*Hero\_Run*”. W ten sposób uzyskano nawiązanie prowadzące z bezczynności do biegu. To jednak nie wszystko, gdyż nie ma warunku, który przełączyłby animacje. Lewa strona edytora zawiera zakładkę „*Parameters*”. To w niej ustawiona zostanie zmienność typu bool o nazwie „Run” z domyślnie przypisanym false. Teraz można zmodyfikować ustawienia strzałki dodając parametr „Run” z warunkiem wykonania, gdy zawiera true oraz zerując i odłączając opcje płynnego przejścia z opóźnieniem, gdyż są one całkiem zbędne przy poklatkowym stylu. Identycznie stworzono strzałeczkę powrotną z ustawieniem warunku na false.



Rysunek 24 - Animacja biegu.

Animacje muszą wiedzieć kiedy się uruchomić, dlatego czas przejść do kodu. Startowo wykonywane w pętli będzie „Idle” i dopiero po wpisaniu do parametru „Run” wartości true, odpalony zostanie „Run”. Tworząc w metodzie „Update” warunek bazujący na sprawdzeniu podłoża oraz wartości ruchu, wywołana zostanie funkcja animatora zmieniająca stan jego wewnętrznych zmiennej. Kod prezentuje się następująco:

[ ControllerPlayer.cs ]

```
...
private Animator animator;
void Start () {
    ...
    // Odwołanie do animatora z pomocą krótszej zmiennej
    animator = GetComponent<Animator>();
}
void Update(){
    ...
    /* W momencie gdy gracz wykona ruch mając kontakt z podłożem, zmienność „Run”
    w animatorze przyjmie wartość „true” uruchamiając zapętloną animację
    biegu... */
    if(grounded && move != 0) animator.SetBool("Run", true);
    /* ... inaczej przypisana zostanie wartość „false” i powrót do „Idle”. */
    else animator.SetBool("Run", false);
}
```

❖ Jump & Crouch:



Rysunek 25 - Animacja ekwipunku, skoku i kucania.

Skok i kucnięcie dodano podobnie jak wcześniejsze animacje z dopisaniem odpowiednich linii kodu oraz rozszerzono menadżer wejść o dodatkowy input z klawiszem „S” na klawiaturze, „LB” na padzie xbox'a 360 z wykorzystaniem odpowiedniego nazewnictwa, tj. „*joystick button 4*”. Kod rozszerzono o dodatkowe linie, gdzie w przypadku kucnięcia nastąpi sprawdzenie, czy wciśnięto i przytrzymano odpowiedni klawisz, zaś w przypadku skoku wykozystano już istniejący blok sprawdzający zmienną „*grounded*”. Zmiany odpowiednich parametrów animatora wywołają przejścia. Animacja skoku będzie jednocześnie animacją spadania, wywoływaną przy braku kontaktu z podłożem. Sam kod prezentuje się następująco:

[ ControllerPlayer.cs ]

```
void Update() {
    ...
    /* Jeśli wciśnięto przycisk kucania i ruch jest równy zero to parametr
     animatora o nazwie „Crouch” ustaw na true, wywoła to przejście animacji...
    */
    if (Input.GetButton("Crouch") && move == 0){
        animator.SetBool("Crouch", true);
    }
    /* ... w przeciwnym razie ustaw na false, wracając do „Idle” */
    else if (Input.GetButtonUp("Crouch") || move != 0){
        animator.SetBool("Crouch", false);
    }

    if (grounded)
    {
        ...
        /* Jeśli wystąpił kontakt z podłożem ustaw parametr animatora „Jump” na
         false by wrócić do „Idle”... */
        animator.SetBool("Jump", false);
    }
    /* ... w przeciwnym razie ustaw na true, by postać przyjęła animację podczas
     skoku, będącą również animacją spadania */
    else animator.SetBool("Jump", true);
}
```

❖ Equip:

Stworzenie ekwipunku wymaga zablokowania możliwości ruchu postaci, gdyż przyciski sterujące posłużą do nawigowania po zawartości plecaka. Dodano do kodu zmienną typu bool o nazwie „*keyboardOn*” ustawioną domyśle na true (sterowanie dozwolone), po czym między klamrami warunku umieszczono bloki kodu z reakcją na wciśnięte przyciski, rzecz jasna poza tymi odpowiedzialnymi za włączenie/wyłączenie ekwipunku. Dla rozpoznania aktualnego stanu ekwipunku, tj. włączony/wyłączony, stworzono zmienną typu bool o nazwie „*equipActive*” z przypisaniem fasle (zamknięty). Po tych modyfikacjach kod wygląda następująco:

[ ControllerPlayer.cs ]

```
private bool keyboardOn = true;
private bool equipActive = false;
void Update(){
    grounded = Physics2D.OverlapCircle(groundContact.position, groundRadius,
                                         groundLayer);

    if (keyboardOn){
        if (Input.GetButton("Jump") && jumpClock > 0) {
            body.velocity = Vector2.up * jumpForce;
        }else if (Input.GetButtonUp("Jump")) jumpClock = 0;

        if (grounded && move != 0) animator.SetBool("Run", true);
        else animator.SetBool("Run", false);

        if (Input.GetButton("Crouch") && move == 0){
            animator.SetBool("Crouch", true);
        }else if (Input.GetButtonUp("Crouch") || move != 0){
            animator.SetBool("Crouch", false);
        }
    }
    if (grounded){
        jumpClock = jumpTime;
        animator.SetBool("Jump", false);
    }
    else animator.SetBool("Jump", true);

    if (Input.GetButtonDown("Equip")){
        if (!equipActive && grounded){
            animator.SetBool("Equip", true);
            equipActive = true;
            keyboardOn = false;
        }else{
            animator.SetBool("Equip", false);
            equipActive = false;
            keyboardOn = true;
        }
    }
}

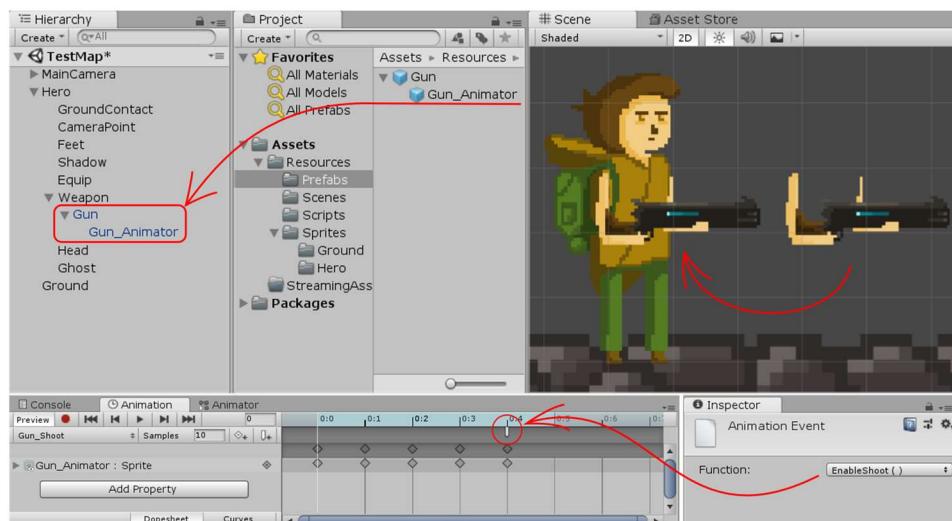
private void FixedUpdate(){
    if (keyboardOn){
        if (Input.GetAxisRaw("Horizontal") > 0.4) move = moveSpeed;
        else if (Input.GetAxisRaw("Horizontal") < -0.4) move = moveSpeed * -1;
        else move = 0;
    }
}
```

❖ Shoot & Dead:

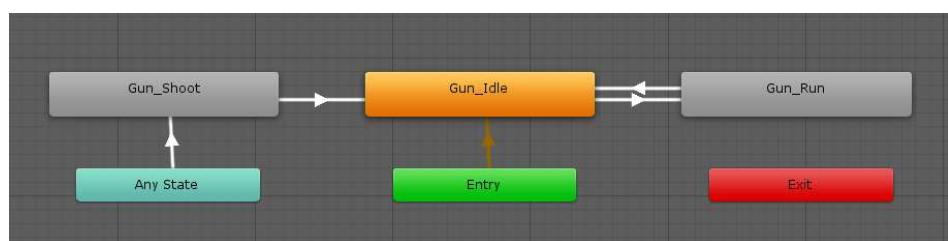
Odrzut oraz śmierć bohatera wywołane zostaną w odpowiednich miejscach nieco później. Przygotowano jednak niezapętlone animacje na tą okazję oraz do animatora wstawiono parametry:

- „Dead” typu bool wywołujący reakcję od strony „Idle” do animacji śmierci,
- „Death” typu „Trigger” będący czymś na wzór impulsu, umieszczony zostanie w przejściu z „Any State” do animacji śmierci (tej samej co w przypadku „Dead”), dzięki czemu po stracie wszystkich życia wykona się natychmiastowo,
- „Shoot” będący „Trigger’em” podczepionym pod „Any State” w celu natychmiastowego wywołania animacji odrzutu. Z tamtej zaś nastąpi przekierowanie do „Idle” po zakończeniu animacji, dlatego też należy zaznaczyć opcję „Has Exit Time” i ustawić ją na zero. Reszta parametrów wyłączona jak w każdym poprzednim przypadku.

❖ Strzelba & ręce:



Rysunek 27 - Broń jako prefab oraz animacja strzału z wywołaniem funkcji.



Rysunek 26 - Animator strzelby z trzema animacjami.

Wedle wcześniejszych ustaleń, bohater gry będzie mógł zmieniać broń, to też rozdzielono element będący rękoma od reszty ciała, by przygotować grafikę trzymanej strzelby oraz animacje dla niej. Powielono plik piskel'a z nazwą „Idle”, by następnie

utworzywszy nową warstwę, dorysować ręce z trzymaną spluwą. Resztę ciała wymazano, a rozmiar grafiki dopasowano do powstałego elementu, ponieważ doczepienie obiektu ze strzelbą jako dziecko bohatera, może zostać odpowiednio przesunięte za sprawą pozycji lokalnej. Grafiki na siebie najdą tworząc całość. Zostaje jeszcze wskazać w obiekcie broni, sortowanie warstw na jedną wyżej, by przykryć tors bohatera. Podobnie jak w opisywanych wcześniej animacjach, i tu uzupełniono brakujące, by współgrały z całością. Otrzymano więc schemat animatora jak na grafice 27. Przedstawione strzałki od razu wskazują na użycie trigger'a przy warunku wykonania strzału oraz bool'a dla ruchu przy biegu. Korzystając z menadżera wejść, zmodyfikowano „*Fire1*” wprowadzając nazwę „*Attack*” i przypisując przyciski „*J*” dla klawiatury oraz „*X*” dla pada, tj. „*joystick button 2*”. Bardzo ważnym jest fakt, że animowanie elementu z użyciem pozycji, blokuje możliwość jej zmiany z poziomu kodu, dlatego też dodano nowy, pusty obiekt „*Gun*”, do którego przeniesiono obecnie omawiany, ze zmianą nazwy na „*Gun\_Animator*”. Dzięki takiemu zabiegowi, skrypt trafi do nadrzędnego elementu sterując całym obiektem, a animacja zacznie odnosić się do lokalnego położenia względem rodzica. Ostatni krok co do formy polega na stworzeniu prefabrykatu, który posłuży w późniejszym etapie za wzór do stworzenia obiektu. Jest to wymagane, gdyż dziecko w „*Weapon*” będzie usuwane i dodawane, na podstawie aktualnego wyekwipowania. Wystarczy więc przeciągnąć gotowy element do odpowiedniego folderu (stworzono katalog „*Prefabs*”) w zasobach. Udana akcja zmieni kolor nazwy obiektu w hierarchii na niebieski. Nastał czas na skrypt numer dwa, każda broń może zawierać inny zestaw akcji, to też niechaj zachowanie doczepianych obiektów będzie oddzielone od kontrolera bohatera. Stworzono nowy skrypt C# „*ControllerGun*”, w którym wskazano obiekt „*Hero*”, dodano reakcje na przyciski oraz odpowiednie przejście animacji na podstawie zachowania rodzica. Najprostsza droga do takiej obserwacji prowadzi przez zmianę parametrów „*move*”, „*keyboardOn*” oraz „*equipActive*” z prywatnych na publiczne w skrypcie sterującym bohaterem, co da wgląd do nich z poziomu innych skryptów, zaś by nie zaśmiecać inspektora dopisano nad zmiennymi „*[HideInInspector]*”. Sama broń musi reagować na wciśnięty przycisk strzału, odpowiednio zachować się w każdym momencie animacji postaci oraz posiadać czas przerwy między wyrzucaniem pocisków. Ostatni problem rozwiązano z użyciem animacji wywołując na końcu strzału funkcję zmieniającą wartość zmiennej „*canShoot*” na true, co pozwoli wykonać kolejny strzał.

[ ControllerGun.cs ]

```

public class ControllerGun : MonoBehaviour
{
    private ControllerPlayer hero;
    private Animator hero_animator;
    private Transform gun;
    private bool canShoot = true;

    void Start(){
        /* Dzięki „hero” odwołanie do skryptu gracza będzie krótsze */
        hero = GameObject.Find("Hero").gameObject.GetComponent<ControllerPlayer>();

        /* Z animatorem bohatera podobnie jak wyżej */
        hero_animator = GameObject.Find("Hero").gameObject.GetComponent<Animator>();

        /* Skrypt znajduje się w obiekcie childe, zaś modyfikacja pozycji wykonana
        zostanie dla rodzica, to też dla łatwiejszego odwoływanego stworzono zmienna
        gun typu „Transform” */
        gun = transform.parent;
    }

    private void Update(){
        /* W momencie otwarcia ekwipunku, animacja postaci jest uniwersalna,
        więc posiada ręce, którymi szpera w plecaku, dlatego zamiast usuwać
        aktualną broń, wyłączono wyświetlanie jej grafiki */
        if (hero.equipActive) GetComponent<SpriteRenderer>().enabled = false;
        else{
            /* W innym przypadku grafika zostanie włączona oraz możliwe będą akcje
            związane z animacjami, jak np. strzał. */
            GetComponent<SpriteRenderer>().enabled = true;

            /* Po wcisnięciu przycisku ataku... */
            if (Input.GetButton("Attack")){
                /* ... sprawdzono czy postać ma możliwość strzału i czy bohater może
                się ruszyć, a jeśli tak to wywołano animację oraz zablokowano
                strzelanie... */
                if (canShoot && hero.keyboardOn){
                    canShoot = false;
                    GetComponent<Animator>().SetTrigger("Shoot");

                    /* W dodatku jeśli bohater stał w miejscu, wywołano odrzut po
                    stronie jego animatora. */
                    if (hero.move == 0 && !Input.GetButton("Crouch")){
                        hero_animator.SetTrigger("Shoot");
                    }
                }
            }
        }

        /* Podczas biegu uruchomiono odpowiednią animację broni... */
        if (hero.move != 0) GetComponent<Animator>().SetBool("Run", true);
        /* ... inaczej wraca do „Idle”. */
        else GetComponent<Animator>().SetBool("Run", false);

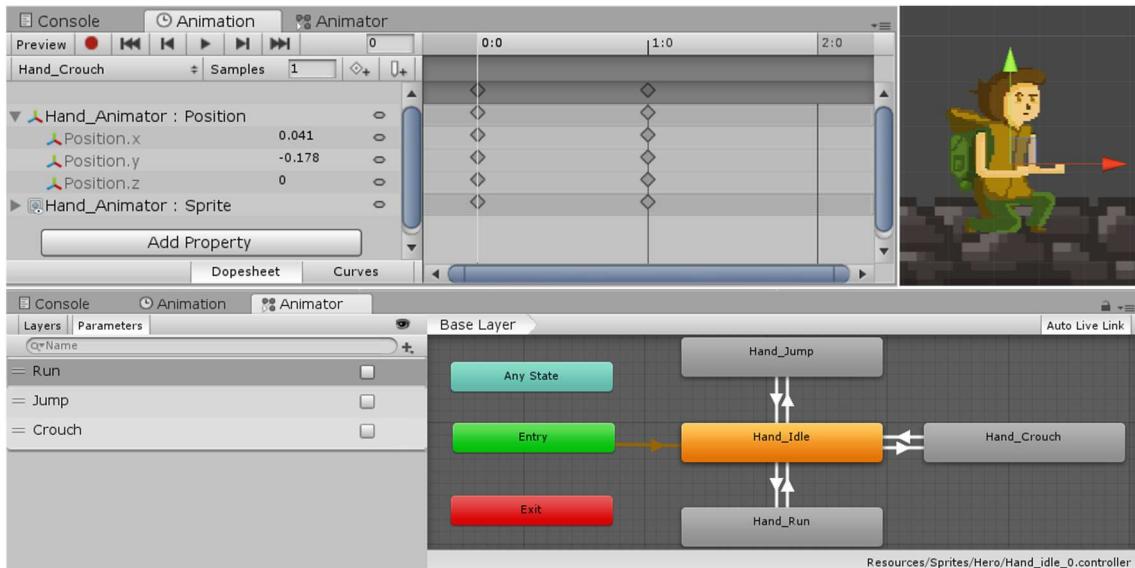
        /* By nie tworzyć specjalnie animacji kucnięcia, przesunięto pozycję
        lokalną rodzica nadając wartości podejrzane w inspektorze, po ręcznym
        przesunięciu całego obiektu w momencie kucania postaci... */
        if (hero.move == 0 && Input.GetButton("Crouch")){
            gun.localPosition = new Vector3(-0.032f, -0.154f, 0);
        }
        /* ... jednak przy nie spełnionym warunku, powrócono do wartości
        oryginalnych. Taki zabieg nie byłby możliwy, gdyby operowano na obiekcie z
        narzuconą animacją modyfikującą położenie, przez co wymagany jest obiekt
        nadzędny, o którym wspomiano w opisie nad kodem. */
        else gun.localPosition = new Vector3(0, 0, 0);
    }
}

```

```

/* Funkcja wywołana po stronie animacji, mająca na celu umożliwić oddanie kolejnego strzału. Taki zabieg jest możliwy o ile animacja i skrypt są przypisane do tego samego obiektu. */
public void EnableShoot(){
    canShoot = true;
}
}

```



Rysunek 28 - Animacja kucnięcia z przesunięciem pozycji.

Efekt końcowy jest całkiem zadowalający, tym bardziej, iż ręce bohatera (gdy nie ma nic wyekwipowanego), stworzono w niemalże identyczny sposób. Tu też użyto pojedynczą klatkę z animacji biegu by wywołać inną pozę przy kucaniu. Przy okazji nadano odpowiednią pozycję ustawiając początkowe i końcowe wartości animacji na takie same. Można to wykonać z pomocą czerwonej kropki w edytorze animacji, ponieważ po jej włączeniu, wszelkie modyfikacje wybranego obiektu trafią na linię czasu w animacji. Wcześniej ten zabieg wykonano z poziomu kodu nadając nowe wartości. Nie założono ataku w ręcz, dlatego pozbyto się również tych bloków kodu, co ostatecznie w skrypcie wygląda następująco:

[ ControllerHand.cs ]

```

public class ControllerHand : MonoBehaviour
{
    private ControllerPlayer hero;
    private Transform hand;

    void Start(){
        hero = GameObject.Find("Hero").gameObject.GetComponent<ControllerPlayer>();
        hand = transform.parent;
    }

    private void Update(){
        // Po raz kolejny ustalono widoczność grafiki w odniesieniu do ekwipunku.
        if (hero.equipActive) GetComponent<SpriteRenderer>().enabled = false;
    }
}

```

```

        else GetComponent<SpriteRenderer>().enabled = true;

        /* Animacja skoku na podstawie kontaktu z podłożem */
        if(!hero.grounded) GetComponent<Animator>().SetBool("Jump", true);
        else GetComponent<Animator>().SetBool("Jump", false);

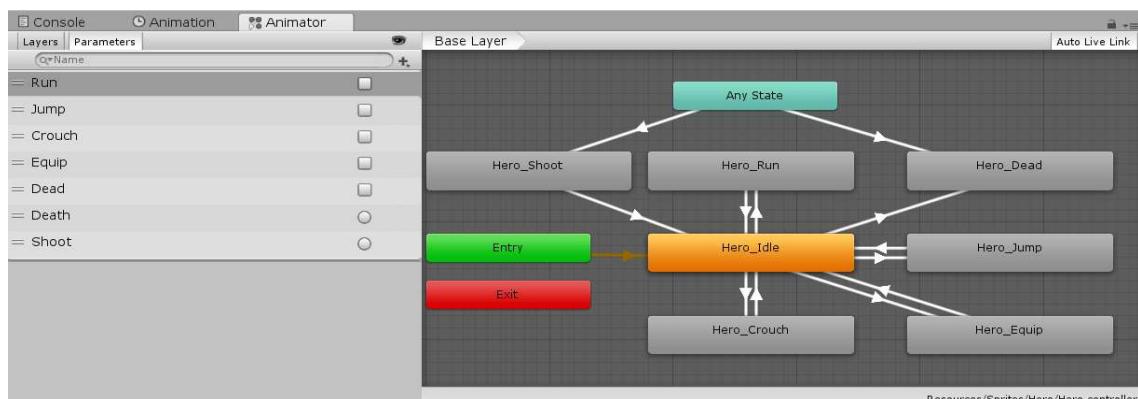
        /* Animacja biegu podobnie jak w przypadku z bronią */
        if (hero.move != 0) GetComponent<Animator>().SetBool("Run", true);
        else GetComponent<Animator>().SetBool("Run", false);

        /* Wywołano animacje kucnięcia, bez zmiany położenia rodzica, gdyż w samej
        animacji zawarto zmianę położenia dziecka */
        if (hero.move == 0 && Input.GetButton("Crouch")){
            GetComponent<Animator>().SetBool("Crouch", true);
        }else GetComponent<Animator>().SetBool("Crouch", false);
    }
}

```

❖ Podsumowanie:

Każda kolejna animacja czy to do bohatera czy dla przedmiotów będzie oparta na zasadach przedstawionych w opisywanym kroku. W ramach zakończenia tego punktu zamieszczono obraz 29, który przedstawia kompletny stan animatora bohatera gry, pozostałe zaś zawarko przy okazji omawiania wcześniejszych podpunktów.



Rysunek 29 - Animator bohatera z ustawionymi przejściami animacji.

#### 4.3 Zapis, odczyt i obiekt „Config” (krok 3).

Na starcie stworzono obiekt z nazwą „Config”, po czym dodano do niego nowy skrypt o takim samym nazewnictwie. Cel jego istnienia wyjaśniono w rozdziale poprzednim to też w skrócie ujmując, będzie to element „zarządcą”, przechodzący między scenami. Aby nie uległ destrukcji podczas ładowania nowej sceny, po pierwsze zamieszczono w kodzie zmienną typu „Config” (czyli obecna klasa) z nazwą „instance” oraz parametrem „static”, dzięki czemu, zmienna stanie się wspólną dla każdego stworzonego obiektu. Następnie dodano metodę Unity „Awake”, która wykona się przed funkcją „Start”, gdzie sprawdzone zostanie czy zmienna „instance” jest pusta. W przypadku spełnienia warunku przypisany zostanie obecny obiekt z zakazem usuwania, inaczej

nastąpi usunięcie, ponieważ będzie to drugi, nowo stworzony element. Wyjaśniając, zmiana sceny powiegi obiekt, co doprowadzi do problemów z odnoszeniem się do właściwego, dlatego też niezbędne jest zachowanie jednego, oryginalnego „Config'a” [17].

Tu też nastąpi wczytanie parametrów startowych dla gry (o czym podczas omawiania menu głównego), a z uwagi, że założono jedynie jeden slot zapisu, wczytany zostanie „save”. Zapis oraz odczyt stanu gry to nic innego jak przechowanie kluczowych informacji wskazujących, w którym miejscu gracz przerwał przygodę. Stworzono zatem plik z klasą „static” (nie dziedziczącą tym razem po „MonoBehaviour”) o nazwie „SaveSystem”. Nie rozrzucając zadania po plikach, dodano na dole drugą klasę publiczną, ale nie statyczną, o nazwie „DataSave”. Do niej trafiły zmienne przechowujące kluczowe parametry. Klasa „SaveSystem” zawiera metody zapisu, odczytu, tworzenia nowego zapisu z początkowymi parametrami (nowa gra) oraz ścieżki wskazujące na folder nie poddany spakowaniu po zbudowaniu gry, czyli „StreamingAssets”. Wspominano o nim m.in. przy okazji punktu pierwszego obecnego rozdziału. Pierwsze co wykonano podczas zapisu i odczytu to sprawdzenie czy ścieżka do katalogu „data” istnieje, a w przypadku braku nastąpi utworzenie. Metoda „SaveGame” oczekuje przekazania obiektu z danymi, które następnie poddane zostaną serializacji i zapisze do pliku binarnego pod nazwą „la\_data\_0.sav” (nazwa może być dowolna). Podobnie w przypadku „LoadGame”, gdzie dodatkowo sprawdzono istnienie pliku z zapisem, wczytano go, zdeserializowano do obiektu o typie „DataSave” i zwrócono. W razie braku save'a, zostanie wywołana funkcja „NewGame”, która przez wywołanie metody „SaveGame” z wartościami początkowymi stworzy nowy zapis. Proces serializacji posłuży do zabezpieczenia danych przed niepożądaną modyfikacją, choć warto zaznaczyć, iż nie jest to metoda nie do złamania [12] [24].

[ SaveSystem.cs ]

```
/* Użycie biblioteki wejścia wyjścia w celu zapisu i odczytu. */
using System.IO;
/* Użycie biblioteki formatującej dane do postaci binarnej. */
using System.Runtime.Serialization.Formatters.Binary;

public static class SaveSystem{
    /* Przypisanie do zmiennej „path_dir” ścieżki z folderem „data” znajdującego się w katalogu niepakowanym podczas budowy projektu. Należy zaznaczyć, iż projekt tworzono pod kątem systemów z rodziny Windows, dlatego dodanie do ścieżki „/data” zadziała jedynie dla owego systemu */
    static string path_dir = Application.streamingAssetsPath + "/data";
    /* Jak wyżej, ale do pliku zapisu. */
    static string path_sav = Application.streamingAssetsPath +
        "/data/la_data_0.sav";
```

```

public static void SaveGame(DataSave data){
    /* Nowa zmienna dla formatu binarnego. */
    BinaryFormatter formatter = new BinaryFormatter();
    /* Stworzenie katalogu dla zapisu jeśli ten nie istnieje. */
    if (!Directory.Exists(path_dir)) Directory.CreateDirectory(path_dir);
    /* Nowy plik strumień na podstawie miejsca zapisu. */
    FileStream strem = new FileStream(path_sav, FileMode.Create);
    /* Serializacja do pliku. */
    formatter.Serialize(strem, data);
    /* Zamknięcie pliku. */
    strem.Close();
}

public static DataSave LoadGame(){
    /* Stworzenie katalogu dla zapisu jeśli ten nie istnieje */
    if (!Directory.Exists(path_dir)) Directory.CreateDirectory(path_dir);
    /* Jeśli plik istnieje... */
    if (File.Exists(path_sav)){
        BinaryFormatter formatter = new BinaryFormatter();
        /* Wczytanie pliku z zapisem gry. */
        FileStream strem = new FileStream(path_sav, FileMode.Open);
        /* Przeformatowanie danych do obiektu typu „DataSave”. */
        DataSave data = formatter.Deserialize(strem) as DataSave;
        strem.Close();
        return data;
    /* W wypadku braku pliku... */
    }else{
        /* ... wywołaj funkcję nowej gry by stworzyć nowy zapis i zwróć dane. */
        DataSave data = NewGame();
        return data;
    }
}

public static DataSave NewGame(){
    /* Stwórz dane początkowe zapisu... */
    DataSave data = new DataSave(1, 0, 0);
    /* ... i wykonaj zapis z użyciem powyższej metody „SaveGame”. */
    SaveGame(data);
    return data;
}
}

/* Poniższa klasa wskaże dane do przechowania. Wraz z rozbudową projektu,
nastąpi jej powiększenie, zaś obecnie przechowa jedynie indeks oraz pozycje
gracza x i y (oś z zawsze będzie zero) */
[System.Serializable]
public class DataSave{
    public int scene;
    public float[] position;
    public DataSave(int sceneIndex = 1, float x = 0, float y = 0){
        scene = sceneIndex;
        position = new float[2];
        position[0] = x;
        position[1] = y;
    }
}

```

System stworzony w ten sposób powinien zachować ważniejsze dane gry nie dając bezpośredniej możliwości ich modyfikacji chociażby przez notatnik. To jednak nie koniec zadania, na scenę wraca obiekt „Config”, gdyż to w nim umieszczone funkcje mające wywołać zapis lub odczyt. Niezbędna będzie zmienna typu „DataSave” o nazwie, np. „LoadedData”, z której inne obiekty odczytają potrzebne parametry. Kolejno dopisano trzy funkcje:

- „SaveGame” tworzącej obiekt z danymi do zapisu,
- „LoadGame” wczytujący dane z pliku, następnie ładujący odpowiednią scenę,
- „NewGame” tworzący nowy zapis z parametrami początkowymi, które trafią również do zmiennej „LoadedData”. Przeładowanie do sceny początkowej.

[ Config.cs ]

```
/* Zainportowano menadżer scen, by możliwe było pobranie indeksu sceny oraz
załadowanie wskazanej. */
using UnityEngine.SceneManagement;

public class Config : MonoBehaviour {
    /* Zmienna wspólna dla wszelkich obiektów klasy. */
    private static Config instance;
    private Transform hero;
    /* Zmienna publiczna przechowująca dane z zapisu gry, do której dostęp mają
    inne obiekty. Przypisano wartość null, na podstawie której w skrypcie gracza
    umieszczono warunek */
    public DataSave LoadedData = null;

    /* Metoda Unity wykonywana przed „Start”... */
    void Awake(){
        /* ... sprawdza czy przypisano już obiekt „Config”... */
        if (instance == null){
            /* ... jeśli nie to przypisany zostanie obecny... */
            instance = this;
            /* ... oraz wskazano, by go nie niszczyć przy przeładowaniu sceny. */
            DontDestroyOnLoad(instance);
            /* Jeśli obiekt jest już przypisany, oznacza to, że powstał drugi, który
            powinien zostać usunięty. */
            else Destroy(gameObject);
        }
    }

    public void SaveGame(){
        /* Stworzenie nowego obiektu z aktualnymi danymi dla zapisu. W tym
        przypadku nie ma sensu ustawać na starcie nawiązania do bohatera, gdyż po
        przeładowaniu sceny zostanie ono zerwane, w związku z czym w momencie
        zapisu odszukany zostanie wymagany obiekt i pobrane zostaną współrzędne.*/
        DataSave data = new DataSave(
            SceneManager.GetActiveScene().buildIndex,
            GameObject.Find("Hero").transform.position.x,
            GameObject.Find("Hero").transform.position.y
        );
        /* Wykonanie zapisu do pliku. */
        SaveSystem.SaveGame(data);
    }

    public void LoadGame() {
        // Załadowanie danych z zapisu do zmiennej dostępnej dla innych obiektów.
        LoadedData = SaveSystem.LoadGame();
        /* Przeładowanie sceny do wskazanej. */
        SceneManager.LoadScene(LoadedData.scene);
    }

    public void NewGame() {
        /* Załadowanie danych do zmiennej dostępnej dla innych obiektów przy
        okazji tworzenia nowego zapisu. */
        LoadedData = SaveSystem.NewGame();
        /* Przeładowanie sceny do wskazanej. */
        SceneManager.LoadScene(LoadedData.scene);
    }
}
```

Skrypty są gotowe do działania, zostaje teraz dodać element wywołujący odpowiednie metody. W tym celu stworzono bardzo przydatne w dalszej części tworzenia skróty klawiszowe „F5” oraz „F8” po wcisnięciu których zastąpi zapis lub odczyt gry w dowolnym momencie. Dopisano zatem po stronie kontroli bohaterem, w metodzie „Update”, sprawdzanie na sztywno tych dwóch przycisków. Trzeba pamiętać również, iż bohater na starcie powinien przyjąć pozycję z save'u. Zmodyfikowano więc skrypt gracza:

[ ControllerPlayer.cs ]

```
private Config config;

void Start () {
    ...
    Config = GameObject.Find("Config").GetComponent<Config>();
    /* Nadanie nowej pozycji na podstawie wczytanych wartości o ile zostały
    wczytane. W przypadku testowania sceny, wczytanie zapisu na początku jest
    pomijane. */
    if(config.LoadedData != null){
        transform.position = new Vector3(
            config.LoadedData.position[0],
            config.LoadedData.position[1],
            0
        );
    }
}

void Update(){
    /* Szybki zapis. */
    if (Input.GetKeyDown(KeyCode.F5)) config.SaveGame();
    /* Szybkie wczytanie. */
    if (Input.GetKeyDown(KeyCode.F8)) config.LoadGame();
    ...
}
```

Wygląda na to, że wszystko działa jak należy. Z czasem skrypty zostaną rozbudowane co opisane zostało w dalszej części pracy. Zamiast szybkich zapisów powstaną punkty do tego przeznaczone, wzorowana na ogniskach z „Dark Souls”, co oznacza, że użycie przywróci całe zdrowie oraz wskrzesi pokonanych przeciwników.

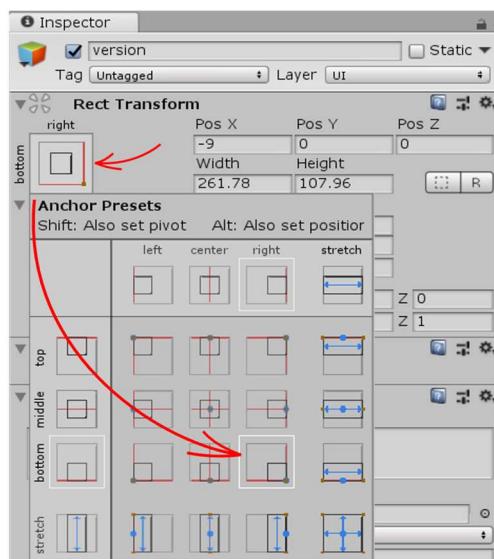
#### 4.4 Podstawy UI, menu główne i konfiguracja początkowa (krok 4).

Menu główne to w rzeczywistości nowa scena z obiektem pozwalającym używać elementów interfejsu. Dodano zatem scenę z nazwą „MainMenu” i kliknawszy prawym przyciskiem myszy na okno hierarchii, najechano na „UI” po czym wskazano „Canvas”. Okno inspektora dla „Canvas” ukazało szereg opcji, z których zmodyfikowano „UI Scale Mode” wybierając „Scale With Screen Size” oraz wprowadzając poniżej w „Reference Resolution” X = 1280, Y = 720. Taki zabieg powinien elegancko dostosować

stworzone menu pod rozdzielcość monitora. Identycznie będzie w przypadku dodania interfejsu podczas rozgrywki [14].

❖ O menu głównym i podmenu:

Dodano do płotna (tj. canvas) z dostępnych elementów UI, obiekt z tekstem o nazwie „*Version*”, w którym zamieszczono ręcznie wpisaną wersję gry. Teraz bardzo ważny zbieg. Napis z wersją musi się zawsze znajdować po prawej na dole, to też w obiektach dla UI zamiast komponentu „*Transform*” domyślnie umieszczono „*Rect Transform*”. Daje on możliwości zachowania pozycji obiektu do płotna (canvas). Kliknięto na grafikę pozycji (zaraz pod nazwą komponentu) by wyświetlić okno pozycyjne, następnie przytrzymując klawisz „*Shift*” + „*Alt*” i wybrano dolny prawy róg. „*Shift*” wyświetla informację o punkcie przylepienia, w tym wypadku obiekt z wersją gry ma się podpiąć prawym dolnym rogiem. „*Alt*” zaś wskazuje położenie całego obiektu względem „*Canvas*”. Użyta kombinacja oznacza nic innego jak podepnij obiekt do prawej oraz do dołu gdzie punktem przylepienia będzie prawy, dolny róg obiektu.



Rysunek 30 - Pozycja UI z Rect Transform.

Następnie tło kamery ustawiono na kolor szaro-niebieski (#2C3241) oraz dodano element „*Image*” przypisując wcześniej stworzoną w Gimp’ie grafikę białej winiety. Zmieniono jej kolor na czarny dla lepszego efektu wizualnego. Tym razem obiekt nie będzie przyjmował położenia na podstawie wskazanego miejsca, a pokryje się z całym płotnem, co jest możliwe dzięki wybraniu ostatniej opcji w menadżerze pozycji. Ikonka przedstawia strzałki góra-dół oraz lewo-prawo, na środku zaś jest kropka. Można to

interpretować jako: ustaw wysokość i szerokość na takie jakie ma rodzinę oraz ustaw punkt przyczepienia w centrum.

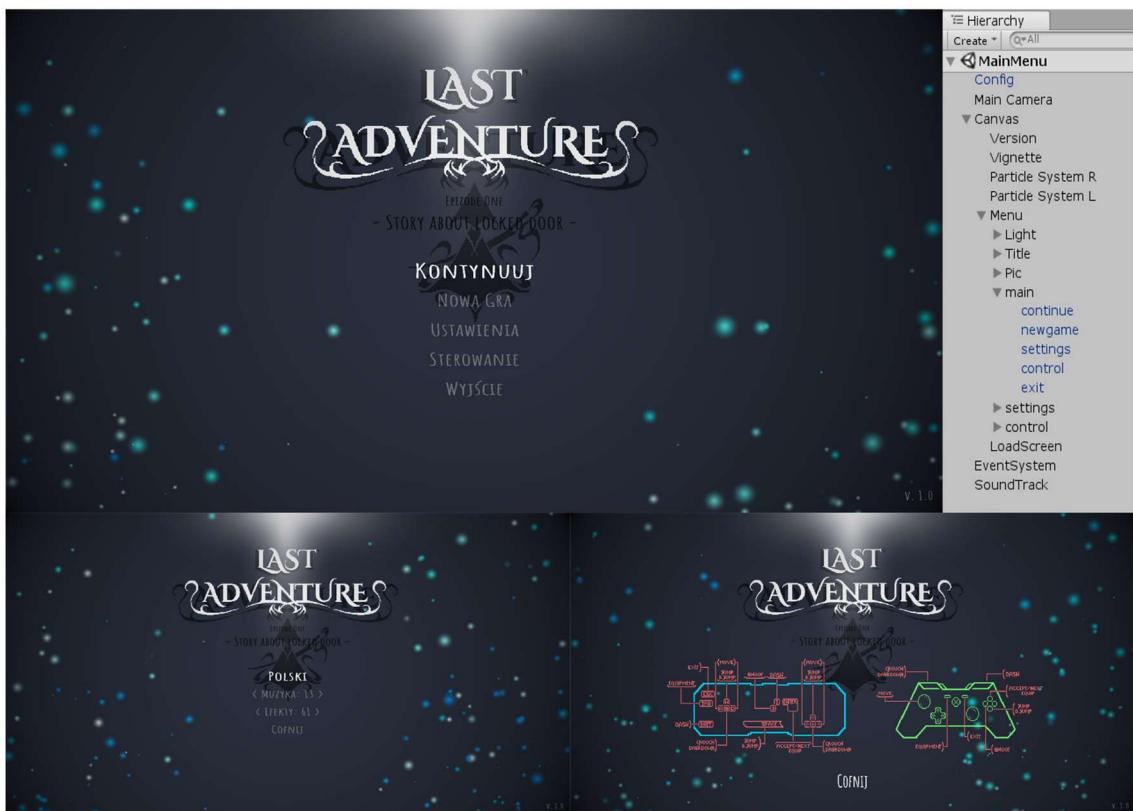
Dorzucono również system cząsteczek (PPM na Canvas > Effects > Particle System), zmieniając takie parametry jak kolor, by przechodził po gradiencie od niebieskiego do białego, rozmiar ze wskazaniem minimalnego i maksymalnego, delikatną, ujemną grawitację (-0.01) by leciały ku górze niczym unoszone pyłki, a całość zapętlono i zaznaczono tak zwany „*prewarm*”, dzięki któremu już na starcie przestrzeni zostanie zapełniona cząsteczkami. Powielono obiekt, po czym oba obrócono tak by leciały od dołu do krawędzi na górze. Tu również ustawiiono odpowiednią pozycję z niewielkim przesunięciem [25].

Kolejno dodano obiekt z nazwą „*Menu*”, do którego trafią: tytuł gry, smuga światła, podtytuł, grafika oraz trzy menu z przyciskami, tj. „*main*”, „*settings*” i „*control*”. Przygotowano białą grafikę z tytułem, po czym umieszczono ją w pustym obiekcie z nazwą „*Title*” i powielono. Klonowi ustawiiono czarny kolor, nadano przezroczystość oraz nieco rozciągnięto. Całość ustawiiono tak by tytuł biały przysłaniał ciemny dzięki czemu powstał efekt cienia. Nad obiektem tytułu dodano nowy z nazwy „*Light*”, w którym wstawiono rozciągnięte gradienty przygotowane w Gimp’ie (o których mowa była na początku projektu) i nadano nieco przezroczystości. Pod „*Title*” umieszczono obiekt z nazwą „*Pic*”, zaś w nim grafikę bliżej niezidentyfikowanej postaci i podtytuł będący tym razem czarnym napisem z lekką przezroczystością. Co do czcionki, użyta została „*AmaticSC-Bold*” dostępna do pobrania ze strony internetowej „*goole fonts*”, następnie zainportowana i wskazana w komponencie. Całość roz mieszczonego odnosząc się do „*Canvas*”, jak w przypadku wyżej opisanego obiektu „*Version*”. Ze względu na tworzenie gry z myślą o padzie, po menu użytkownik poruszać się będzie bez interakcji z myszą. Przyciski zatem mogą być obiektem z tekstem, wizualnie ulepszone animacją, a konkretnie modyfikacja skali oraz przezroczystości. Umieszczone w rodzinie komponent „ *AudioSource*” z krótkim dźwiękiem dla efektu przechodzenia po przyciskach.

Na końcu „*Canvas*” umieszczone „*LoadScene*”, czyli element „*Plane*” z UI. Połuży za wyciemnienie przed przeładowaniem sceny i na początku po załadowaniu. Ustawiono kolor biały bez przezroczystości. Dodano animator z animacjami zanikania oraz przysłaniania, czyli zmiana parametru „*Alpha*” w kolorze. Jako domyślna od startu wykonana się ta z przejściem przezroczystości od 100% do 0%. Animacje nie są zapętlone, to też użyto parametru typu bool do przełączania nimi. Przy zmianie sceny odpalona

zostanie ta z zabieleniem ekranu, to też na jej końcu umieszczono wywołanie funkcji „*ReloadScene*” z niewielkiego skryptu dołączonego jako komponent o nazwie „*Reload*”. Znaczy to tyle, że nie trzeba wywoływać przeładowania, a jedynie wywołać animację. Numer sceny przechowany zostanie w parametrze „*LoadedData.scene*”. Biorąc to pod uwagę, przeładowanie zawarte w obiekcie „*Config*” przy „*LoadGame()*” oraz „*NewGame()*” można podmienić na wywołanie kurtyny.

Ostatni element w hierarchii to obiekt ze ścieżką dźwiękową w tle o nazwie „*SoundTrack*”. Umieszczony w nim komponent „ *AudioSource*” odtworzy wraz z załadowa-



Rysunek 31 - Menu główne oraz podmenu gry.

niem sceny zapętlony utwór „*Flights of Fantasy*”. Dołączono również skrypt zarządzający poziomem głośności, na podstawie wskazanej w konfiguracji gry. Całość ostatecznie wygląda naprawdę dobrze, a to dzięki minimalistycznemu stylowi oraz kolorystyce, co razem nie wywołuje natłoku informacji (nie ma chaosu).

#### ❖ O przyciskach w menu:

Przechodzenie po przyciskach zrealizowano na zasadzie sprawdzenia ilości podobiektów w rodzicu dzięki czemu, ustawiając zmienną liczącą nie wyjdziemy poza zakres. Użytkownik wciskając przycisk w góre, bądź w dół zmienia aktualnie wskazywany

numer dziecka. W momencie gdy nowa wartość nie wychodzi poza zakres nastąpi animacja wygaszająca obecny przycisk, a uruchomiona zostanie włączająca dla naastego. Po wciśnięciu przycisku akceptacji, sprawdzone zostanie z pomocą switch'a oraz nazw obiektów, który przycisk aktualnie został wciśnięty i wywołany zostanie odpowiedni ciąg akcji. Od tej pory zaczyna się większa ilość kodu, to też wklejanie całości zajmowałoby po kilka stron. Zostaną zatem umieszczone jedynie co ciekawsze rozwiązań, jak np. funkcja przechodzenia po przyciskach.

[ MainMenu.cs ]

```
/* Funkcja odpowiada za zmianę przycisków. Przyjmuje string'a na podstawie
którego obsługuje kierunek */
public void Move(string go){
    /* Jeśli przekazano „up” lub „down” wykonaj... */
    if (go == "up" || go == "down"){
        int add = 0;
        /* ...add ustaw na zero, sprawdź czy możliwy jest ruch, a jeśli tak to
ustaw -1 dla skoku w górę lub 1 dla skoku w dół... */
        if (go == "up" && child > 0) add = -1;
        else if (go == "down" && child < childCount) add = 1;
        /*...Jeśli ustalono podejmij próbę skoku... */
        if (add != 0){
            child += add;
            /*...child wskazuje obecny przycisk z pomocą cyfry dlatego dodano add w
którym jest 1 lub -1. Następnie nastąpi sprawdzenie czy wskazany obiekt
posiada komponent animator. Jeśli tak to nastąpi animacja... */
            if (transform.GetChild(child).GetComponent<Animator>() != null){
                /* Animacja wygaszenia dla elementu aktywnego. */
                transform.GetChild(child - add).GetComponent<Animator>().
                    SetBool("Select", false);

                /* Animacja włączenia dla elementu wskazanego. */
                transform.GetChild(child).GetComponent<Animator>().
                    SetBool("Select", true);

                /* Odtworzenie krótkiego dźwięku. */
                GetComponent<

```

❖ O konfiguracji i językach:

Jak wyżej wspomniano, wciśnięcie przycisku wywoła ciąg akcji wskazany w odpowiednim case'ie, a to za sprawą porównania nazwy obiektu z dostępnymi warunkami.

Wracając do „*Config'a*”, wspominano kilkakrotnie o wczytaniu ustawień początkowych gry. Nadszedł czas na zgłębienie owego zagadnienia. Chodzi o stworzenie pliku w formacie json, z którego odczytane zostaną parametry takie jak poziom dźwięku, wybrany język i języki dostępne. Sam plik będzie w folderze „*StreamingAssets*”, by dostęp był łatwy. Z poziomu kodu jest to dość proste, gdyż istnieją gotowe funkcje na przetwarzanie danych z obiektu i do obiektu. Stworzono zatem dwie klasy na podstawie których będą przeprowadzane operacje:

- „*configJson*” – posiadającą w sobie kluczowe informacje startowe, czyli: obecnie wybrany język, języki dostępne, poziom muzyki w tle oraz efektów. Powstały plik nosi nazwę „*config.json*” i wygląda to następująco:

```
{ "Language": 0, "LangList": ["pl", "en"], "SoundtrackVolume": 0.7, "EffectVolume": 1}
```

- „*menuJson*” – tu zaś będą tłumaczenia przycisków. Plik nosi nazwę „*pl-menu.json*” i wygląda jak poniżej:

```
{
    "Conti" : "Kontynuuj",
    "NewGame": "Nowa Gra",
    "Sett" : "Ustawienia",
    "Lang" : "Polski",
    "Exit" : "Wyjście",
    "Volume" : "Dźwięki",
    "Sound" : "Muzyka",
    "Effect" : "Efekty",
    "Control": "Sterowanie",
    "Back" : "Cofnij"
}
```

Na tej podstawie możliwa jest automatyczna konwersja. Ścieżka do pliku jest składana na podstawie konfiguracji początkowej, gdzie numer języka wskazuje na ten w liście dostępnych. Wykorzystując parametr „*Application.streamingAssetsPath*”, uzyskano ścieżkę do folderu wymiany, teraz zostaje dokleić nazwę katalogu i pliku, co ostatecznie będzie wyglądało mniej więcej tak (na zielono zaznaczono dodaną informację odnośnie ścieżki z pliku konfiguracji, na pomarańczowo dopisane elementy w kodzie):

```
„D:\Last Adventure\Last Adventure_Data\StreamingAssets\language\pl\pl-
menu.json”
```

Wracając do skryptu „*Config*”. Dodano tam potrzebne funkcje plus jedną sterującą poziomem dźwięku. Polega ona na zmianie aktualnej wartości głośności o +/- 0.01, w zależności od tego co odebrała funkcja wywołana podczas sterowania w menu. Reszta zaś wygląda on następująco:

## [ Config.cs ]

```
/* Zadeklarowano zmienne o typie klasy ustalonej w dolnej części oraz
string'owe dla wczytania konfiguracji i ustalenia ścieżek. */
private menuJSON menu;
public configJSON config = new configJSON();
private string jsonString;
private string languaPath;
private string configPath;
/* Funkcja odpalona zostanie w metodzie „Awake” z momentem ustawienia
instancji obiektu „Config”, czyli na samym początku uruchomienia gry. */
public void ReadConfig(){
    /* Wskazano ścieżkę do pliku z konfiguracją dźwięków, języka, itd., na
    podstawie folderu wymiany „StreamingAssets”. Po raz kolejny ta metoda
    zadziała
    jedynie dla Windows'a, ze względu na zapis ścieżki. */
    configPath = Application.streamingAssetsPath + "/config.json";
    if (!File.Exists(configPath)){
        /* Jeśli plik nie istnieje to do zmiennej wprowadzone zostaną wartości
        domyślne po czym plik zostanie zapisany. */
        config.Language = 0;
        config.LangList = new List<string>(new string[] { "pl", "en" });
        config.SoundtrackVolume = 0.5f;
        config.EffectVolume = 0.5f;
        saveConfig();
    }else{
        /* Jeśli plik istnieje nastąpi jego wczytanie... */
        jsonString = File.ReadAllText(configPath);
        /* ... a następnie przekonwertowanie do obiektu. */
        config = JsonUtility.FromJson<configJSON>(jsonString);
    }
    /* Na podstawie konfiguracji początkowej nastąpi uzupełnienie ścieżki do
    plików z tekstami o różnych tłumaczeniach. */
    languaPath = Application.streamingAssetsPath + "/language/" +
        config.LangList[config.Language] + "/" + config.LangList[config.Language]
        + "-menu.json";
}
/* Funkcja wywołana zostanie po stronie skryptu „MainMenu” i ma na celu
wczytać konfigurację językową oraz uzupełnić przyciski odpowiednimi tekstami.
*/
public void MainMenuConfig(){
    // Odczytano plik z wcześniej ustalonej ścieżki do zmiennej string'owej...
    jsonString = File.ReadAllText(languaPath);
    /* ... następnie z wykorzystaniem gotowej funkcji przetworzono dane do
    obiektu. */
    menu = JsonUtility.FromJson<menuJSON>(jsonString);
    ...
    // Dalej zawarto switch by przydzielić odpowiednie nazwy do przycisków menu.
}

public void saveConfig(){
    /* Z użyciem gotowej funkcji przetworzono obiekt do postaci string'owej na
    wzór pliku json... */
    string json = JsonUtility.ToJson(config);
    /* ... następnie odpowiednio zapisano plik. */
    File.WriteAllText(configPath, json);
}

/* Klasa z danymi zapisu dla konfiguracji. */
[System.Serializable]
public class configJSON
{
    public int Language;
    public List<string> LangList;
    public float SoundtrackVolume;
    public float EffectVolume;
}
```

```
[System.Serializable]
public class menuJSON
{
    public string NewGame;
    public string Conti;
    public string Sett;
    public string Lang;
    public string Exit;
    public string Volume;
    public string Sound;
    public string Effect;
    public string Control;
    public string Back;
}
```

❖ O interfejsie gry:

Wygląda na to, że po raz kolejny wszystko działa bez większego problemu, w związku z czym pora przejść do kwestii interfejsu w grze. Po pierwsze cały czas widoczny będzie stan zdrowia. Po drugie poddasz otrzymania obrażeń ekran na bokach zrobi się czerwony. Po trzecie w momencie śmierci nastąpi przyciemnienie ekranu i wyświetli się napis „*Nie czas na spanko!*”. Dodano zatem do zapisu nową zmienną typu całkowitego, która przechowuje ilość życia bohatera. Kolejno utworzono „*Canvas*” na scenie z bohaterem, do którego dodano:

- grafikę z gałązką życia, do niej tąflą listki odzwierciedlającą ilość zdrowia,
- winietę dla lepszego efektu,
- info będące oknem dialogowym,
- „*Plane*”, który poddany animacji będzie wyświetlany podczas śmierci bohatera,
- obiekt z czterema elementami przyczepionymi na boki ekranu, posłuży to do zatrzymania kamery, gdy bohater zbliży się do krawędzi mapy,
- animacja wyświetlana w momencie zapisu,
- informacje o bossie takie jak ilość zdrowia i nazwa.

❖ O życiu i śmierci:

Dialogami zajęto się nieco później, a tymczasem grafikę gałązki przyczepiono z pomocą „*Rect Transform*” do lewego górnego rogu. Same listki będą animowanymi prefabami umieszczanymi na podstawie przemieszczenia od rodzica to też w configu dopisano potrzebne funkcje zarządzające: „*AddLife*”, „*DropLife*”, „*RespawnLife*” oraz „*OnePunchDeath()*”. Lepiej opisywać kod w kodzie więc:

[ Config.cs ]

```

/* Funkcja wywoływana przez stracie wcześniejszej wspomnianej „Reload”, z tej
uwagi, iż config wykona start tylko raz, przy włączeniu gry. */
public void RespawnLifes(){
    /* Ilość obecnie przechowywanego zdrowia trafi do zmiennej target by
    wyzerować życia. Następna funkcja, która zostanie stąd wywołana buduje
    listki i inkrementuje zdrowie, dlatego ten zabieg jest potrzebny. */
    int target = LoadedData.life;
    LoadedData.life = 0;
    for (int i = 0; i < target; i++) AddLife();
}

/* Funkcja tworzy jeden listek życia i inkrementuje wartość w
„LoadedData.Life”. */
public void AddLife(){
    /* Jako pierwsza zostanie ustalona pozycja listka na podstawie obecnych
    punktów zdrowia oraz podanego przesunięcia. Oczywiście jeden listek to jeden
    punkt HP. */
    Vector3 newVec3 = new Vector3(0.47f + (LoadedData.life * 0.34f), -0.21f,
    LifeTree.transform.position.z);
    /* W sposób pokazany poniżej powstanie nowy obiekt będący dzieckiem gałązki
    życia. */
    GameObject tmp = Instantiate(leafPrefab, LifeTree.transform, false);
    /* Jako że nie można nadać pozycji przy tworzeniu obiekt trafił do zmiennej
    tymczasowej w celu odwołania do jego komponentów. Teraz można wsazać mu
    odpowiednie miejsce ustalone w pierwszej linijce funkcji. */
    tmp.transform.localPosition = newVec3;
    /* Inkrementacja punktów zdrowia. */
    LoadedData.life++;
}

/* Metoda usuwa ostatni listek i zmniejsza ilość punktów zdrowia. */
public void DropLife(string mode = ""){
    /* Najpierw nastąpi sprawdzenie czy w ogóle są jakieś życia. */
    if (LoadedData.life > 0){
        /* Jeśli tak to zlokalizowany zostanie ostatni element gałązki życia, po
        czym wywołana zostanie animacja destrukcji. Jak wcześniej wspomniano,
        listek ma w sobie własne komponenty więc na końcu animacji wywołuje skrypt
        samozniszczenia. */
        LifeTree.transform.GetChild(LoadedData.life - 1).gameObject.
            GetComponent<Animator>().SetTrigger("Destroy");

        LoadedData.life--;
        if (LoadedData.life <= 0){
            /* Teraz jeśli życie jest równe 0 bądź mniej nastąpi sprawdzenie czy
            bohater nie posiada specjalnego przedmiotu przywracającego max zdrowia.
            Jeśli posiada to zostanie przywrócone całe zdrowie, ale jeśli ustawiono
            wartość force to ominie fakt posiadania przedmiotu. Chodzi o krytyczne
            sytuacje jak np. wypadnięcie poza mapę. Bohater spadałby w
            nieskończoność, a tak to zginie i się odrodzi przy zapisie.Więcej o tym
            przy okazji omawiania ekwipunku. */
            if (CheckActiveItem("LifeCrystal") && mode != "force"){
                DropActiveItem("LifeCrystal");
                LoadedData.life = MaxLife;
                RespawnLifes();
                /* W wypadku przeciwnym wywołana zostanie funkcja podejmująca akcję
                uśmiercenia. */
            }else GameObject.Find("Hero").GetComponent<ControllerPlayer>().Death();
        }
    }
}

/* Funkcja natychmiastowego uśmiercenia x _ x */
public void OnePunchDeath(){
    for (int i = 0; i < LoadedData.life; i++) DropLife("force");
}

```

❖ O skrypcie Reload:

Pół strony kodu, pół opisu do niego, a zapowiada się jeszcze więcej. Cóż praca będzie naprawdę obszerna. Wywołanie śmierci postaci to kwestia kolejnych animacji, z końcem, bądź w trakcie których wywołane zostaną inne, aż do tej z przeładowaniem sceny. Zostaje jeszcze dopełnienie kwestii powrotu do menu. Do tego celu przypisano przycisk powrotu w inspektorze, czyli „Esc” i „Start” (pad xbox’a, joystick button 7). Zmodyfikowano nieco skrypt obiektu Config dodając zmienną typu bool „Esc” z początkową wartością false. W „Reload” zaś nastąpi sprawdzenie czy aby właśnie ta zmienna nie jest na true ustawniona, a jeśli tak to nastąpi przeładowanie bez zapisu do sceny z menu. Jest to skrypt krótki to i widoczny poniżej:

[ Reload.cs ]

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class Reload : MonoBehaviour {
    private Config config;

    private void Start(){
        config = GameObject.Find("Config").GetComponent<Config>();
        /* Podczas startu wywołuje po stronie Config'a odświeżenie powiązań z
        obiektami. Jest to wymagane, gdyż podczas zmiany sceny wszelkie obiekty
        (poza Config) zostają usunięte i wczytane nowe. */
        config.LinkRefresh();
    }

    public void ReloadScene(){
        /* Jeśli wskazano zakończenie to przejdź do sceny z nim. */
        if (config.Fin){
            /* Profilaktycznie ustaw portal na false, czyli nie użyty. Jest to
            potrzebne przy przechodzeniu postaci między scenami. O tym nieco
            później. */
            config.portal = false;
            SceneManager.LoadScene(config.FinScene);
        }
        /* Jeśli wciśnięto esc, przeładuj scenę do menu, ustawiając zmienne portal
        oraz esc na false by widoczne były jako nie użyte. */
        else if (config.Esc){
            config.portal = false;
            config.Esc = false;
            SceneManager.LoadScene(0);
        }
        /* Tu zaś nastąpi wczytanie sceny, do której prowadzi przejście. O tym
        więcej nieco później */
        else if (config.portal){
            if (config.portalBuild == 0) config.portal = false;
            SceneManager.LoadScene(config.portalBuild);
        }
        /* Ostatnia opcja to wczytanie sceny z zapisu gry. */
        else SceneManager.LoadScene(config.LoadedData.scene);
    }
}
```

Na koniec dodano punk „*Respawn*” będący obiektem, mającym posłużyć jedynie do pobrania pozycji. Wykorzystany zostanie w momencie startu w skrypcie bohatera. Sprawdzone zostaną x oraz y z zapisu i jeśli wskazują zera to bohater przyjmie pozycję owego punktu. Będzie to bardzo wygodne przy debugowaniu scen oraz posłuży za strat po wybraniu nowej gry. Poniżej zamieszczono wygląd finalny interfejsu w grze.



Rysunek 32 - Interfejs w grze.

#### 4.5 Projekt świata, iluzja światła, ruch kamery i portale (krok 5).

Podczas projektowania świata, twórcę ogranicza jedynie wyobraźnia. W grach komputerowych można stworzyć dosłownie wszystko niczym w opowieści. Warto jednak pamiętać by trzymać spójność świata, choć w nieco bardziej zwariowanych przygodaach, łamanie tej zasady może być całkiem interesującym zabiegiem. Projektując „*Last Adventure*” chciałem połączyć tematyki post-apokalipsy z fantastyką, to też nie wykluczono ani magii, ani technologii tworząc przede wszystkim całkiem ciekawe krajobrazy.

##### ❖ Elementy świata, czyli przygotowanie grafiki:

Wszelkie elementu powstaną z rozdzielcością niską ale odpowiednią do potrzeb. W obecnym projekcie nie jest wymagane utrzymanie jakiegoś standardu, gdyż można

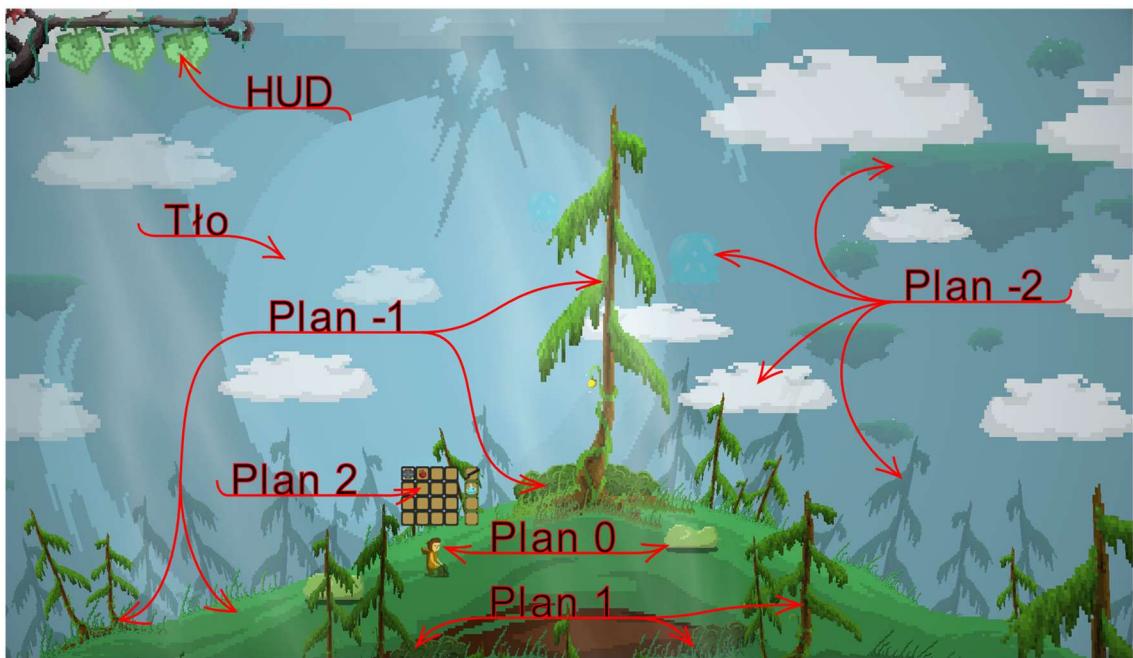
swobodnie wszelkie obiekty skalować. Nie znaczy to, że różnice mogą być kłujące w oczy. Co do kwestii skalowania obiektu na scenie, warto unikać sytuacji gdzie x oraz y będą różne. Elementu świata, w większości będą grafiką bez animacji, choć nic nie stoi na przeszkodzie by np. takie drzewko wprawić w delikatny ruch. Nadając światu życia, nadano ruch trawie tworząc kilka klatek, po czym ją powielono i rozmieszczono na całej scenie. Efekt wyszedł całkiem przyjemny, a rozbudować go można o dodatkowy wariant animacji, gdzie w przypadku styczności w obiektem imitującym wiatr poruszy się w nieco innym stylu niż domyślnym, przyjmując przez chwilę lekko rozjaśniony kolor. Efektem takiego zabiegu będzie fala.

❖ Sprite Renderer i sortowanie warstwami:

Podczas ustawiania obiektów na scenie bardzo ważnym jest by zwrócić uwagę na sortowanie warstw. Czasem, gdy dwa elementy mają ten sam numer warstwy, a jeden zakrywa drugiego, może dojść do sytuacji, że nagle drugi zacznie zakrywać pierwszego choć nic nie zostało zmienione. Dobrze jest w pełni to kontrolować, dlatego jeśli są elementu przykrywające się, nadano odpowiednie numery warstw. Podążając tym tropem można wyłonić sześć planów, na których rozmieszczone będą elementy świata.

- **Tło, warstwy mniejsze niż -21:** czyli wszystko to co będzie w tle, m.in. cienie, efekty, grafiki urozmaicające.
- **Plan -2, warstwy od -20 do -11:** głównie zajęty przez grafiki w tle takie jak roślinność, budynki, latające meduzy oraz urozmaicenia.
- **Plan -1, warstwy od -10 do -1:** tu zaś dodane zostaną głównie elementu podłoża, roślinność, obiekty, itp.
- **Plan 0, warstwy od 0 do 10:** zarezerwowany dla bohatera oraz przeciwników.
- **Plan 1, warstwy od 11 do 20:** Elementy przysłaniające wszelkie dotychczasowe dla uzyskania lepszego efektu głębi. Zawarto tu zatem grafiki podłoża, roślinność, światła, cienia, itp.
- **Plan 2, warstwy większe niż 21:** przede wszystkim ekwipunek wyświetlany nad graczem, by zawsze był dobrze widoczny oraz elementy pierwszoplanowe, głównie światła, cienie, czasem flora.
- **HUD (ang. Head-Up Display):** nie jako warstwa ale interfejs w grze, o którym mowa była w podrozdziale wcześniejszym.

Trzymanie się tych założeń ułatwia pracę przy tworzeniu sceny oraz ogranicza ewentualne błędy wizualne. Na grafice 33 przedstawiono przykładowy fragment świata „Last Adventure” z rozmieszczonymi obiektyami.



Rysunek 33 - Rozmieszczenie obiektów według warstw.

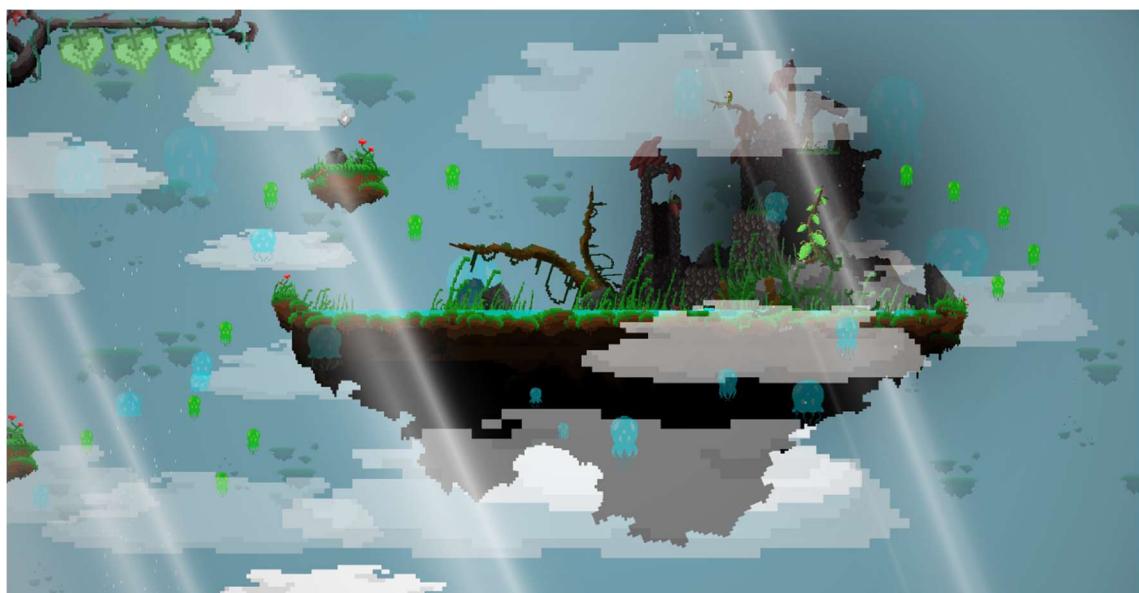
❖ Iluzja światła i cienia:

Słowo iluzja użyto nie bez powodu. To co wygląda na promienie/smugi światła w grafice 33 oraz 34 to w rzeczywistości biały gradient, nieco rozciągnięta, z nadaną przezroczystością. Przykrywając elementy w rzeczywistości rozjaśnia kolory pod nią. Na takiej samej zasadzie stworzono cienie z tą różnicą, iż nadano tej samej grafice czerń z pomocą edytora kolorów w „Sprite Renderer”. Dobrą praktyką jest tworzenie obrazów jasnych, gdyż można je przyciemnić zmieniając kolorystykę (jak wyżej) w zakresie skali szarości. Niestety rozjaśnić się nie da w ten sposób, ponieważ domyślnie ustaliona jest biel. Obraz 33 prezentuje drzewa na wzgórzu. Do stworzenia tego fragmentu użyto jedynie jednej grafiki drzewka. Wszystko co tam widać to jeden powielony sprite, a każdy zmieniony nieco na potrzeby sceny. Identycznie zrobiono w przypadku chmur, trawy, podłożą i wielu innych obiektach, a wszystko to, by zaoszczędzić jak najwięcej zasobów. Innym bardzo fajnym zabiegiem jest przyciemnianie oraz zmiana przezroczystości. Drzewa na tyle sceny, lewitujące skały, czy też meduzy wykonano z pomocą edycji kolorystyki oraz zmniejszenia skali, co dało fajny efekt nie uderzający w oczy i wypełniający tło. Z kolei elementy bliżej kamery powiększono, również przyciemniono dla efektu iluzji perspektywy.

Latające meduzy widoczne w tle na rysunku wyżej, dla odmiany poza animacją, poruszają się z użyciem skryptu. W nim losowana jest wartość wysokości na jaką ma żyjętko podlecieć, jej wielkość oraz zmiana animacji w momencie opadania.

❖ Kamera:

Dobrze wykonany ruch kamery jest niezbędny dla dobrego odbioru całej produkcji to też przygotowano skrypt mający na celu zautomatyzować po części jej pracę. Domyślnie będzie gładkim ruchem podążała za bohaterem, jednak, w niektórych miejscach jak np. w SkyTown widocznym w powyższej grafice, będą obiekty z kolizją wywołujące oddalenie i zmianę punktu śledzenia do momentu wyjścia ze strefy. Dodatkowo przy otworzeniu ekwipunku, obraz zostanie przybliżony zwiększąc komfort zarządzania przedmiotami, w każdym miejscu. Skrypt jest dość pokaźny dlatego omówione zostaną części ważniejsze.



Rysunek 34 - Ruiny w SkyTown.

[ ControllerCamera.cs ]

```
private GameObject hero;
private GameObject target;
private Camera cam;
private float defaultSize;
private float zoomSpeed = 0.1f;

void Start(){
    hero    = GameObject.Find("Hero/CameraPoint");
    target  = hero;
    cam     = transform.GetChild(0).GetComponent<Camera>();
    defaultSize = 30;
}

void FixedUpdate(){
    /* Dzięki Mathf.SmoothDamp transformacja w czasie nastąpi płynnie pod koniec
```

```
zwalniając co da przyjemny efekt płynącej kamery za bohaterem. */
if (followX){
    X = Mathf.SmoothDamp(transform.position.x, target.transform.position.x,
                          ref velocity.x, SmoothX);
}
if (followY){
    Y = Mathf.SmoothDamp(transform.position.y, target.transform.position.y,
                          ref velocity.y, SmoothY);
}
transform.position = new Vector3(X, Y + addToY, transform.position.z);

/* Ta część odpowiada za zoom. */
if (cameraResize){
    /* Oddalenie. */
    if (defaultSize > cam.orthographicSize){
        cam.orthographicSize += zoomSpeed;
        if (cam.orthographicSize >= defaultSize) zoomChanged();
    }
    /* Zbliżenie. */
    else if (cam.orthographicSize){
        cam.orthographicSize -= zoomSpeed;
        if (cam.orthographicSize <= defaultSize) zoomChanged();
    }
}
```

Najważniejsze zrobione. Kamera podąża płynnym ruchem za bohaterem, więc teraz wskażmy punkty, w których nastąpi zmiana zbliżenia. Jest to obiekt posiadający „*Box Collider 2D*” oraz nazwę „*Zoom*” plus wartość zbliżenia, co wyeliminuje pisanie dodatkowego skryptu. Kolizja wykryta zostanie po stronie gracza i z pomocą funkcji stringowej „*Substring*”, nastąpi wydobyci cyfry wskazujące zoom kamery. Te zaś przetworzono na liczby całkowite, po czym wysłano do funkcji w skrypcie kamery. Podobnie stworzono Focus na konkretne miejsce, z zachowaniem wcześniej wskazanego celu. Na mapie umieszczono zaś obiekty z kolizją, których wywołanie uruchomi odpowiednią zmianę ustawień kamery. Dodatkowymi efektami są wstrząsy wywoływane przy strzale, upadku, czy też wybiciu bohatera na odpowiednim elemencie, stworzone z pomocą prostych animacji [26]. Nada to nieco więcej dynamiki rozgrywce jaki ruchowi.

#### ❖ Portal:

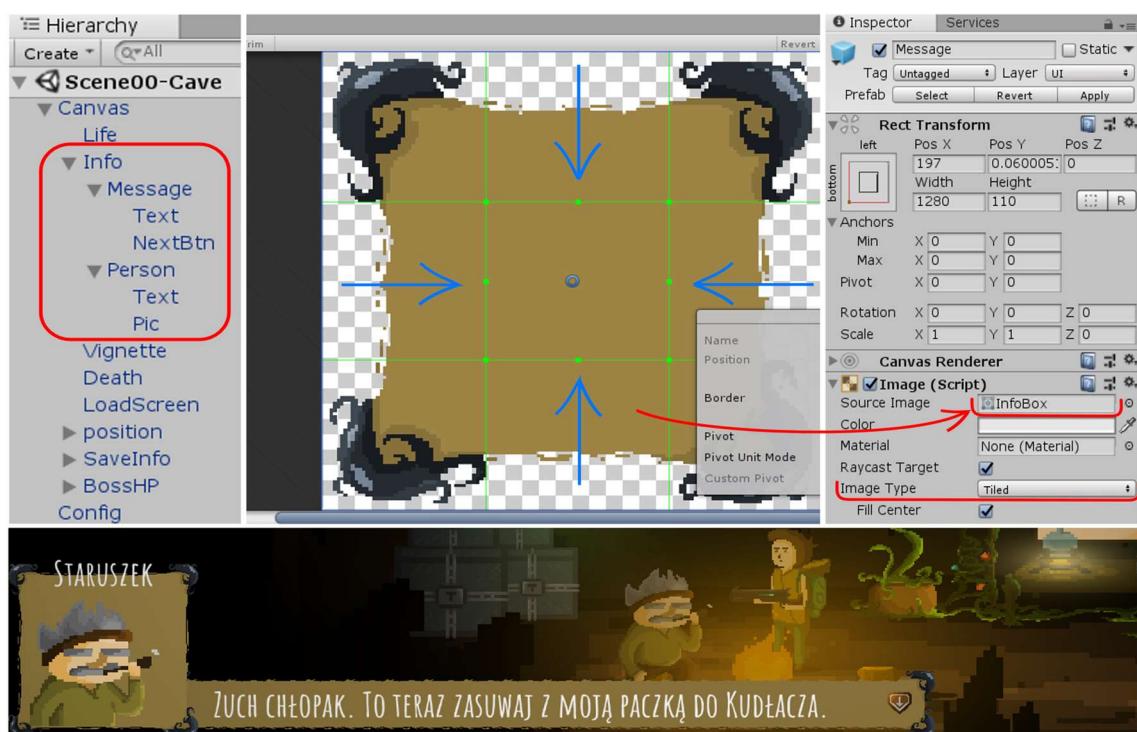
Nic skomplikowanego. Dodano obiekt, do niego pod-objekt oraz skrypt. Rodzic posłuży za punkt przeładowania sceny, zaś dziecko za miejsce, w którym pojawi się bohater gry. Za pomocą inspektora wskazano scenę oraz portal docelowy, ponieważ nic nie stopi na przeszkodzie by było ich więcej na jednej mapie. Następnie wywołano animację wybielania ekranu, która po skończeniu uruchomi ładowanie nowej sceny.

#### 4.6 NPC, interakcja, dialogi i zadania (krok 7 - 8).

Świat bez celu jest pusty i nudny, to też w prawie każdej grze znajdzie się ktoś lub coś co wskaże graczowi drogę ku przygodzie. W projekcie umieszczono dwie takie postaci, starszego naukowca oraz jego wnuczkę. Dzidziuś zleci graczowi zadanie oraz wyjaśni zasady gry, zaś dziewczyna poprosi o pomoc, tym samym zlecając kolejnego quest'a. Zaczęto od strony wizualnej przygotowując odpowiednie grafiki z animacją. Poza rozmowami nie przewidziano większych interakcji dlatego cała magia tkwi w dialogach.

- ❖ Okno dialogowe:

Okno dialogowe przedstawi portret postaci, nazwę oraz linię dialogową. Dodano zatem do „Canvas” obiekt z nazwą „Info”, w nim zaś pod-obiekty „Message” oraz „Person”, a w nich zaś „Text”, „NextBtn” oraz „Text”, „Pic”. Przygotowano niewielką grafikę, którą z pomocą edytora spritów podzielono na 9 elementów. Do obiektu wstawiono komponent „Image” z przygotowanym obrazem, który posłuży jako tło pod tekst wiadomości, zaś dzięki wybraniu opcji „Tiled” z listy rozwijanej „Image Type”, obraz będzie się zapętlał i dostosowywał rozmiarem do wielkości okna. Jako czcionkę dla tekstu użyto „AmaticsSC-Bold” pobraną ze strony Google Fonts. Poniższy wycinek z gry przedstawia efekt końcowy [27].



Rysunek 35 - Przygotowanie okna dialogowego.

❖ Menadżer dialogów:

System dialogów oparty został na interpretacji tekstu linia po linii oraz podejmowaniu odpowiednich działań. W celu zapobiegnięcia powielonego kodu, stworzono klasę „*ControllerDialogManager*”, której instancja stworzona zostanie w niewielkim skrypcie sterującym „*DialogMaker.cs*”. Większość informacji zawarta została w tekście z użyciem słów kluczowych, które oddzielają separatory pionowej kreski, w celu oddzielenia części głównych i dwukropka w celu oddzielenia części wewnętrznych. Sam tekst wczytany zostanie po stronie „*DialogMaker'a*” i przekazany do kontrolera w postaci listy wraz z dodatkowymi parametrami takimi jak odległość aktywacji oraz obiekt rodzic. Przykładowa linia: „*GOTO|ItemInEquip:BlueCard|Open|*” co interpretowane jest jako: „*Idź do | Jeśli w ekwipunku jest przedmiot : nazwa obiektu/przedmiotu | Szukane miejsce | Tekst w dialogu (dla ułatwienia wstawiony jedynie gdy wcześniejsze wartości są puste)*”. Tak przygotowany fragment poddana zostanie rozbiciu na podstawie pionowej kreski, po czym z pomocą switch'ów oraz pierwszej wartości, podjęta zostanie odpowiednia akcja. W przypadku gdzie wystąpi tekst w czwartym parametrze, wywołane zostanie omawiane wyżej okno ukazując wypowiedź. Po podaniu parametrów takich jak nazwa grafiki oraz imię postaci, dodatkowo wyświetcone zostaną odpowiednie elementy informujące kto aktualnie zabiera głos. Ostatecznie tekst wyświetlony będzie do momentu wciśnięcia odpowiedniego przycisku, po czym przejdzie dalej bądź wyjście poza strefę aktywacji, wtedy się wyłączy. Takie rozwiązanie nie ogranicza się jedynie do dialogów z postaciami, może posłużyć również za informację na tablicy ogłoszeń czy wyświetlać opisy przedmiotów.

[ DialogMaker.cs ]

```
/* Informacje pobrane zostaną z pliku więc niezbędne będzie dołączenie biblioteki. */
using System.IO;
public class DialogMaker : MonoBehaviour {
    /* Stworzono pusty obiekt menadżera dialogów, decydującego o akcji, na podstawie aktualnie sprawdzanej linii dialogowej. */
    private ControllerDialogManager CDM;
    /* Ustalenie odległości aktywacji dialogu. */
    public int HorizontalScan = 5;
    public int VerticalScan = 10;
    /* Prefabrykowany obiekt z dźwiękiem. */
    private GameObject audioNext;
    /* Tablica na rozbitą string. */
    private string[] dialog;
    /* Nazwa pliku podana z poziomu inspektora. */
    public string file;
    /* Wczytany plik tekstowy. */
    private string txt;
    /* Ścieżka do pliku. */
    private string path;
    /* Lista kwestii, która trafi do menadżera dialogów. */
}
```

```

public List<string> DialogList = new List<string>();
void Start(){
    /* Ustalenie ścieżki do odpowiedniego pliku. */
    path = GameObject.Find("Config").GetComponent<Config>().
        getLangPath()+file+".txt";

    /* Otwarcie strumienia. */
    StreamReader reader = new StreamReader(path);
    /* Odczytanie pliku tekstowego. */
    txt = reader.ReadToEnd();
    /* Zamknięcie strumienia. */
    reader.Close();
    /* Rozbiście tekstu na linie. */
    dialog = txt.Split( new[] {"\r\n","\r","\n"},StringSplitOptions.None);

    /* Stworzenie listy z kwestiami. */
    foreach (var dialogLine in dialog) DialogList.Add(dialogLine);

    /* Dźwięk przy przejściu do następnej kwestii. */
    audioNext = (GameObject)Resources.
        Load("Audio/Effects/Prefab/Prefab_AudioNext", typeof(GameObject));

    /* Nowy menadżer dialogów. Przekazano do niego wymagane parametry. */
    CDM = new ControllerDialogManager(
        gameObject,
        DialogList,
        HorizontalScan,
        VerticalScan
    );
}
private void Update(){
    /* Odświeżanie okna dialogowego. Menadżer zwraca słowa klucze jako akcje
    dodatkowe, niewykonalne po jego stronie. */
    switch (CDM.DialogController()){
        /* Opcja autodestrukcji. */
        case "Destroy":
            Destroy(gameObject);
            break;
        /* Odtworzenie dźwięku przy przejściu do następnej kwestii. */
        case "Play":
            Destroy(Instantiate(audioNext, transform, false), 0.3f);
            break;
    }
}
}

```

Tak przygotowany plik to jedynie wstęp do części właściwej. Cała magia dzieje się po stronie menadżera dialogów, który sieka całe linie tekstu, wydobywając informacje potrzebne do ciągu akcji. Ogólna koncepcja to nic innego jak skakanie po tekście w celu wyświetlania właściwych kwestii i podejmowaniu adekwatnych działań.

[ ControllerDialogManager.cs ]

```

public class ControllerDialogManager{
    private Config config;
    private GameObject hero;
    private ControllerPlayer c_hero;
    private GameObject host;
    private int DialogLine;
    /* Dystans aktywujący dialogi. */
    private int distanceX      = 5;
    private int distanceXCopy = 5;
    private int distanceY      = 2;
    /* Obiekt cały czas sprawdza odległość od gracza to też by wyłączenie

```

```

dialogu odbyło się jedynie raz, ustawiono dodatkowy parametr
ograniczający.*/
private int onTriggerExit = 0;
/* Przechowuje aktualną linię dialogową.*/
private string[] DialogAssistant;
/* Ustawienia do wyświetlenia grafiki oraz nazwy osoby bądź przedmiotu.*/
private string[] DialogPicture;
/* Zwracane słowo klucz. */
private string callback = "";
/* Ścieżka do folderu z portretami.*/
private string picPath = "Sprites/DialogPic/";
/* Zatrzymanie dialogów.*/
private bool stop = false;
private bool makeSave = false;
private bool reload = false;
/* Lista linii dialogowych.*/
private List<string> DialogList = new List<string>();

/* Konstruktor wykorzystany również jako start, by nadać odpowiednie
wartości początkowe.*/
public ControllerDialogManager(GameObject _Host,List<string> Dialogs,
                                  int x,int y){
    config = GameObject.Find("Config").GetComponent<Config>();
    hero = GameObject.Find("Hero").gameObject;
    c_hero = hero.GetComponent<ControllerPlayer>();
    host = _Host;
    DialogList = Dialogs;
    distanceX = x;
    distanceY = y;
    DialogLine = 0;
    distanceXCpy = x + 1;
    /* Rozbicie zerowej linii. Na tej zasadzie działa „NextLine()”*/
    DialogAssistant = DialogList[DialogLine].Split('|');
    DialogPicture = null;
}

public string DialogController(){
    // Wyczyszczenie zwracanej wartości, będącej wywołaniem akcji dodatkowej.
    callback = "";
    // Mierzenie odległości od gracza. Jeśli jest w zasięgu, uruchom dialog.
    if (Mathf.Abs(host.transform.position.x - hero.transform.position.x) <
        distanceX && Mathf.Abs(host.transform.position.y -
        hero.transform.position.y) < distanceY){
        /* Zwiększenie dystansu dialogu, by ten się nie wyłączył w razie
        minimalnego przesunięcia bohatera.*/
        if (DialogLine == 0 && distanceXCpy < distanceX) distanceX += 1;
        /* Reset wyjścia z dialogu.*/
        onTriggerExit = 0;
        if (!c_hero.equipActive && stop == false){
            /* Wybierz działanie na podstawie parametru pierwszego.*/
            switch (DialogAssistant[0]){
                case "Person":
                    /* Jeśli w trzeciej wartości słowo kluczowe to item, ścieżka do
                    grafiki zostanie zmieniona.*/
                    if (DialogAssistant[2] == "item") picPath = "Sprites/Items/Items/";
                    else picPath = "Sprites/DialogPic/";
                    /* Grafika : Nazwa bądź imię*/
                    DialogPicture = DialogAssistant[1].Split(':');
                    /* To nie dialog więc przejdź do następnej linii.*/
                    NextLine();
                    break;
                    /* i wiele, wiele więcej. Nie zawarto wszystkiego, albowiem zajęłoby
                    to zbyt dużo stron. Koncepcja się nie zmienia, jedynie rozbudowuje o
                    wewnętrzne sprawdzanie kolejnych parametrów, tj. drugiego i
                    trzeciego. Czwarty jest kwestią, więc brany pod uwagę poza
                    switch'em.*/
            }
        }
    }
}

```

```

/* Proste sprawdzenie czy na końcu (czwarty parametr) jest tekst. */
if (DialogAssistant.Length == 4){
    /* Dodatkowe sprawdzenie czy tekst nie jest pusty. */
    if (DialogAssistant[3] != ""){
        /* Włączenie okna dialogowego z przekazaniem wiadomości,
        imienia/nazy oraz ścieżki do grafiki. */
        config.MessageBox(true, DialogAssistant[3], DialogPicture[1],
                          picPath + DialogPicture[0]);
        /* Blokada niektórych ruchów bohatera. */
        c_hero.RunDialog(true);
        /* Akcja po wciśnięciu przycisku kontynuacji kwestii wywołana jeśli
        dialog nie jest zatrzymany. */
        if (Input.GetButtonDown("Submit") && stop == false){
            callback = "Play";
            NextLine();
        }
    }
}
/* W razie braku dialogu odblokuj ruch gracza. */
else KeyboardDisable(false);
}
} else{
    /* Ograniczenie. */
    onTriggerExit++;
    if (onTriggerExit == 1) ExitDialog();
}
// Zwraca ewentualną odpowiedź jak np. Play, która ma wywołać krótki dźwięk.
return callback;
}

```

Oczywiście to nie cały skrypt. Zawarto również funkcje ustawiające wartości początkowe przy wyjściu ze strefy aktywacji, zmianę linii dialogu, szukanie wskazanej, przeskakiwanie do następnych, itp. W warunkach zaś jest masa odwołań do innych skryptów w celu uruchomienia różnych funkcji, jak np. zapis gry, dodanie przedmiotu, czy zdarzenia w świecie. Nie ma większego sensu by opisywać każdy element, ponieważ są one bardzo powtarzalne, a zajęłyby kilka kolejnych stron.

#### ❖ Zdarzenia i quest'y:

W większości gier zakłada się stworzenie pewnego rodzaju dziennika z zadaniami. W przypadku „*Last Adventure*” nie jest to zbyt konieczne, gdyż sama gra nie jest dłuża, a wszelkie informacje zostaną zawarte w powtarzanych kwestiach dialogowych. Jedyne co jest niezbędne to lista zapisywana wraz z save'em, przechowująca zdarzenia jakie miały miejsce. Dla ułatwienia wybrano listę string'ów, a zdarzeniami będą słowa klucze. Pozwoli to na łatwe zarządzanie również z poziomu menadżera dialogów. Przygotowano zatem w config'u funkcje dodającą do listy event, o ile jeszcze nie wystąpił oraz przeszukanie i zwrócenie wartości true jeśli istnieje, false jeśli nie.

[ ControllerDialogManager.cs ]

```
public List<string> global_ActivetedEventList = new List<string>();  
  
/* Sprawdzenie czy w liście zdarzeń występuje przekazane. */  
public bool CheckExistActivetedEvent(string e_event) {  
    for(int i = 0; i < global_ActivetedEventList.Count; i++){  
        /* Zwróć true jeśli wystąpi. */  
        if (global_ActivetedEventList[i] == e_event) return true;  
    }  
    /* Zwróć false jeśli nie wystąpi. */  
    return false;  
}  
  
/* Wstawienie zdarzenia przez wysłanie nazwy eventu. */  
public void SetActivetedEvent(string action)  
{  
    /* Jeśli nie ma wystąpienia to dodaj na koniec nowe zdarzenia. */  
    if(CheckExistActivetedEvent(action) == false)  
        global_ActivetedEventList.Add(action);  
}
```

Przygotowano zapis zdarzeń, ale co z zadaniami? Skakanie po liniach dialogowych! Wstawiając w pierwszej wartości „*GOTO*”, menadżer będzie szukał pierwszej pasującej wartości do wskazanej jako cel, o ile warunek zostanie spełniony, a warunkiem będzie zdarzenie, brak zdarzenia, posiadanie przedmiotu, czy też jego wyekwipowanie. Wszystkie tego typu akcje zawarto w switch'ach menadżera dialogów.

#### 4.7 Ekwipunek, przedmioty i sterowanie 2.0 (krok 9).

Dodanie ekwipunku to w rzeczywistości nowe możliwości dla gry. Od lat twórcy umieszczały w większym bądź mniejszy stopniu jakiś bagaż, którym gracz może zarządzać. Klasyczny „*Diablo*” już w pierwszej odsłonie oferował pokaźnych rozmiarów arsenał przedmiotów, których parametry częściowo były losowane. Idea ta nie umarła, wręcz przeciwnie. Gra „*Borderlands 2*” mocno postawiła na pomysł przedmiotów generowanych na podstawie pewnych reguł, dzięki czemu gracze mają multum możliwości w wyborze broni. „*Last Adventure*” nie jest jednak na tyle dużym projektem to też skupiło się na stworzeniu paru przedmiotów wykorzystanych do leczenia, walki oraz fabuły. Istnieje oczywiście możliwość rozbudowy owego modułu jednak to temat na inny czas. Przede wszystkim należy ustalić w jaki sposób przechować przedmioty w zapisie gry. Oczywiście z pomocą listy. Drugi problem jak zapisać informację o tych przedmiotach, które posiadamy, które są wyekwipowane oraz jakie mają zniknąć z mapy? Więcej list! Nie trzeba wskazać jakie konkretnie id przedmiotu posiada bohater, bo wystarczy, że znamy jego nazwę, a reszta zawarta zostanie w samym przedmiocie.

- ❖ Część wizualna:



Rysunek 36 - Ekwipunek na scenie oraz w hierarchii projektu.

Na początek skupiono jednak uwagę na stronę wizualną ekwipunku. Będzie to obiekt posiadający w sobie inne obiekty, a konkretnie 21 elementów z czego 16 posłuży za ekwipunek, 4 za miejsca wyekwipowania oraz jeden animowany wskaźnik na aktualnie wybrany przedmiot. Przygotowano zatem odpowiednie grafiki, następnie wstawiono je do obiektu „Equip” w bohaterze. By móc się łatwo odwołać do konkretnej komórki oraz pobrać numer wskazujący na kolejność wystąpienia, nadano wartości będące współrzędnymi według rozmieszczenia na scenie, co widać na dołączonej grafice. Przechodzenie po ekwipunku zrealizowano bardzo podobnie jak w przypadku menu głównego, z dodaniem wyszukiwarki zwracającej numer aktualnie zaznaczonego dziecka w obiekcie ekwipunku.

[ ControllerEquip.cs ]

```
...
private GameObject hero;
private Config config;
private GameObject audioNext;
private GameObject audioSelect;
private bool eqMove = true;
private int x = 0, y = 0;
private int i_pos = 1;
private string s_pos = "00";
private string[] childsTab;

void Start (){
    config = GameObject.Find("Config").GetComponent<Config>();
    hero = GameObject.Find("Hero").gameObject;
    // Wywołanie tej funkcji spowoduje załadowanie ekwipunku, ale o tym później.
    config.ActiveEquip();
    /* Ekwipunek domyślnie będzie aktywny przy stracie, lecz zaraz potem zmieni
    ten stan, dlatego też do zmiennej w skrypcie gracza, wysłano nawiązanie do
```

```

aktualnego kodu. */
hero.GetComponent<ControllerPlayer>().equip = this.gameObject;
/* Stworzono tablicę nazw elementów ekwipunku, z której zwracany będzie
numer miejsca w rzędzie. */
childTab = new string[transform.childCount];
for (int i = 0; i < transform.childCount; i++){
    childTab[i] = transform.GetChild(i).transform.name;
}
/* Ustawiono obiekt ekwipunku na nieaktywny, co oznacza, że zarówno
elementy, jak i skrypt przestaną działać. Jego aktywacja zawarta została po
stronie bohatera. */
gameObject.SetActive(false);
}

/* Przy aktywnym ekwipunku, wywołanie reakcji na przyciski. */
private void Update(){ EquipController(); }

public void EquipController(){
/* Zezwolenie na reakcję przy ruchu. */
if (Input.GetAxisRaw("Vertical") ==0 && Input.GetAxisRaw("Horizontal") ==0){
    eqMove = true;
}
/* Jeśli eqMove zawiera true to możliwy jest odczyt ruchu po ekwipunku. Ma
to zapobiec niezwykle szybkim przeskokom po kratkach ograniczając ruch do
pojedynczych zmian. */
if (eqMove){
    if (Input.GetAxisRaw("Horizontal") > 0.4) Move("right");
    else if (Input.GetAxisRaw("Horizontal") < -0.4) Move("left");
    else if (Input.GetAxisRaw("Vertical") < -0.4) Move("down");
    else if (Input.GetAxisRaw("Vertical") > 0.4) Move("up");
    else if (Input.GetButtonDown("Accept")){
        if (transform.GetChild(i_pos).transform.childCount > 0){
            /* Użyj przedmiotu jeśli istnieje w ekwipunku. Ta funkcja zostanie
            omówiona nieco później. */
            config.UseItem(transform.GetChild(i_pos).transform.GetChild(0).
                gameObject, i_pos);
        }
        /* Stworzenie obiektu odtwarzającego krótki dźwięk. Dzięki funkcji
        Destroy element zostanie zniszczony po 0.5 sekundy. */
        Destroy(Instantiate(audioSelect, transform, false), 0.5f);
    }
}
}

/* Funkcja przeszuka tablicę pod kątem przekazanej nazwy będącej współrzędnymi
miejscem w ekwipunku i zwróci numer w szeregu hierarchii. */
public int FindMe(string pos = ""){
    for (int i = 0; i < childTab.Length; i++) if (pos == childTab[i]) return i;
    return -1;
}

/* Ta funkcja odpowiada za ustalenie położenia obiektu „select” oraz
lokalizację wybranej kratki w hierarchii. Działa bardzo podobnie do tego co
było w skrypcie poruszania po menu. */
public void Move(string go = ""){
    /* Ruch w górę lub dół. Zmienia wartość y wskazującą obecne miejsce w
    pionie. */
    if (go == "up" || go == "down"){
        /* Zmiana położenia ustawiona na zero. */
        int add = 0;
        /* Przypisanie odpowiedniej wartości do kierunku ruchu z ograniczeniem
        wyjścia poza ustalony z góry zakres. */
        if (go == "up" && y > 0) add = -1;
        else if (go == "down" && y < 3) add = 1;

        if (add != 0){
            // Położenie w pionie ze strony sceny zwiększa/zmniejsza o wartość w add.
            y += add;
        }
    }
}

```

```

        /* Stwórz współrzędne zgodne z nazwami obiektów, np. '21'. */
        s_pos = y.ToString() + x.ToString();
        /* Zlokalizuj numer w hierarchii, np. 21 to child(10) licząc od 0. */
        i_pos = FindMe(s_pos);
        /* Zmień położenie „select” przypisując położenie wskazanego dziecka. */
        transform.GetChild(0).gameObject.transform.position =
            transform.GetChild(i_pos).gameObject.transform.position;
        /* Krótki dźwięk informujący o zmianie. */
        Destroy(Instantiate(audioNext, transform, false), 0.3f);
    }
}

/* Ruch w lewo lub prawo. Działa identycznie jak pierwszy warunek, z ta-
różnicą, że zmienia wartość x, a nie y. */
else if (go == "left" || go == "right"){
    int add = 0;
    if (go == "left" && x > 0) add = -1;
    else if (go == "right" && x < 4) add = 1;

    if (add != 0){
        x += add;
        ... // jak w pierwszym warunku
    }
}
/* Blokada ruchu do momentu puszczenia przycisku bądź gałki na padzie. */
eqMove = false;
}

```

❖ Przedmioty:

Część wizualna gotowa stworzono więc teraz pierwszy przedmiot, na podstawie którego powstaną kolejne. Na początek przygotowano grafikę jabłka, którego celem będzie odnowienie punktu życia. Przedmiot wystąpi w dwóch postaciach, ten w świecie wirtualnym oraz ten w ekwipunku. Dodano zatem do okna hierarchii pusty obiekt o nazwie „*ItemsList*”, do którego trafią wszelkie występujące na scenie przedmioty. Przeciągnięto na scenę grafikę jabłka by powstał kolejny obiekt, któremu nadano sortowanie warstw na 5 i dopasowano rozmiarem do świata gry. Dla lepszego efektu dodano również do obiektu zaanimowane poklatkowe miganie, by wyróżnić przedmiot z otoczenia i zwrócić uwagę gracza. Wcześniej wspomniano o tym, iż to przedmioty posiadać będą właściwości, dla tego stworzono krótki skrypt z klasą mającą przechować informacje o obiekcie. Grę projektowano z myślą o wielu językach dlatego dane przedmiotu zostaną wczytane z pliku w formacie json do obiektu „*itemClass*”. Może się wydawać, że taki zabieg wprowadza konieczność identyfikowania przedmiotów, ale poza nazwą wyświetlaną w grze można użyć innej. Tu posłużyono się nazewnictwem całego obiektu oraz stworzono prefabrykat. Przykładowo, wstawiono przedmiot z nazwą obiektu i prefabrykatu „*ToolKit*”, co zostało wyświetlone z nazwą „*Narzędzia*”, ale „*ToolKit*” zostaje uniwersalny dla każdego języka, będąc swego rodzaju identyfikatorem. By jeszcze bardziej wykorzystać taki zabieg, rozszerzono perspektywę myślenia i nazwano grafikę przedmiotu na „*ToolKit*”. Teraz jest jedna wartość po której wczytane zostaną odpowiednie elementy. Istnienie

dwóch obiektów da dwa różne skrypty, tak więc przedmioty do podniesienia posiadają plik z nazwą „*GlobalItem.cs*”, a te w ekwipunku „*ItemInfo.cs*”.

### [ *ItemInfo.cs* ]

```
/* Będzie odczytany plik, więc niezbedna biblioteka wejścia/wyjścia. */
using System.IO;
public class ItemInfo : MonoBehaviour {
    public string file;
    private Config config;
    private string txt;
    private string path;
    private string itemString;
    public itemClass item;

    private void Start(){
        config = GameObject.Find("Config").GetComponent<Config>();
        /* Ustalenie ścieżki do pliku na podstawie nazwy podanej w inspektorze. */
        path = config.getLangPath() + file + ".json";
        /* Jeśli plik istnieje to zostanie odczytany i przetworzony na obiekt. */
        if (File.Exists(path)){
            itemString = File.ReadAllText(path);
            item = JsonUtility.FromJson<itemClass>(itemString);
        }else Debug.Log("Nie znaleziono pliku.");
    }

    public void ItemUse(int eqIndex){
        /* Ta część zakłada wywołanie odpowiednich akcji w zależności od typu przedmiotu. Przykładowe jabłko to jedzenie, czyli przywraca energię. */
        switch (item.Type){
            case "Food":
                for(int i = 0; i < item.LifeResp; i++) {
                    if(config.LoadedData.life < config.MaxLife) config.AddLife();
                }
                /* Po użyciu zostaje skasowany z ekwipunku. */
                config.DropItem(eqIndex);
                Destroy(gameObject);
                break;
        }
    }
}

/* Klasa dla przedmiotu. */
[System.Serializable]
public class itemClass
{
    public string ObjectName;
    public string ItemName;
    public string Type;
    public string Description;
    public int LifeResp;
    public string Action;

    /* Konstruktor dla przedmiotu. */
    public itemClass(string obj = "", string name = "", string typ = "", string des = "", int life = 0, string act = ""){
        ObjectName = obj;
        ItemName = name;
        Type = typ;
        Description = des;
        LifeResp = life;
        Action = act;
    }
}
```

Posiadane przedmioty to jedno, jeszcze jest kwestią tych do zbioru, co znacznie rozszerza kod. Trzeba nadać linie dialogowe by wyświetlona została możliwość podniesienia oraz wskazać id do usunięcia wprowadzone z poziomu inspektora, co ma na celu wykorzystywanie wielu kopii przedmiotu, ale do podniesienia jednorazowego.

[ GlobalItem.cs ]

```
using System.IO;
public class GlobalItem : MonoBehaviour {
    private ControllerDialogManager CDM;
    public string ItemDestroyID;
    public string file;
    private Config config;
    private string path;
    private string itemString;
    public itemClass item;
    public int HorizontalScan = 4;
    public int VerticalScan = 8;
    private GameObject audioNext;
    private List<string> DialogList = new List<string>();
    private void Start(){
        config = GameObject.Find("Config").GetComponent<Config>();
        /* Ustalenie ścieżki do pliku na podstawie nazwy podanej w inspektorze. */
        path = config.getLangPath() + file + ".json";
        /* Odczyt pliku jeśli ten istnieje. */
        if(File.Exists(path)){
            itemString = File.ReadAllText(path);
            item = JsonUtility.FromJson<itemClass>(itemString);
        }else Debug.Log("Nie znaleziono pliku.");
        /* Jeśli id usuwania wystąpiło w liście przedmiotów do usunięcia, to
        dokonaj autodestrukcji. Funkcja sprawdzająca to nic innego jak
        porównywanie przesłanej wartości do elementów w liście. */
        if (config.checkDestroyItemOnMap(ItemDestroyID)) Destroy(gameObject);
        /* Stworzono linie dialogowe z uzupełnieniem kluczowych wartości. */
        DialogList.Add("Person|" + item.ObjectName + ":" + item.ItemName + "|item|");
        DialogList.Add("|||>> " + item.Action + ": " + item.ItemName + " <<");
        DialogList.Add("Add|" + item.ObjectName + ":" + item.Type + "|Jump|");
        DialogList.Add("Destroy|||");
        /* Pozostały kod jak w przypadku DialogMaker'a z tą różnicą, że dodano
        przedmiot do listy niszczonych na starcie. */
        CDM = new ControllerDialogManager(
            gameObject,
            DialogList,
            HorizontalScan,
            VerticalScan
        );
        audioNext = (GameObject)Resources.
            Load("Audio/Effects/Prefab/Prefab_AudioNext", typeof(GameObject));
    }
    private void Update(){
        switch (CDM.DialogController()){
            case "Destroy":
                /* Dodanie do listy do usunięcia przy starcie. */
                config.AddItemToDestroyOnMap(ItemDestroyID);
                Destroy(gameObject);
                break;
            case "Play":
                Destroy(Instantiate(audioNext, transform, false), 0.3f);
                break;
        }
    }
}
```

❖ Ekwipunek w listach:

Wszystko działa! Nachodząc bohaterem na przedmiot wyświetla się komunikat, grafika oraz nazwa przedmiotu. Teraz została jeszcze część zarządzania ekipunkiem. Przygotowano zatem odpowiednie funkcje po stronie Config'a odpowiedzialne za dodanie, usunięcie, wyekwipowanie, zdjęcie aktywnego przedmiotu oraz przetwarzanie listy na jeden długi string pod zapis do save'u i odwrotnie. Warto zaznaczyć, że lista przedmiotów posiadanych posiada maksymalnie 16 elementów podobnie jak ekipunek oraz lista wyekwipowanych maksymalnie 4, dzięki czemu łatwiej odwołać się do poszczególnych przedmiotów z pomocą wcześniej ustalonego numeru w hierarchii. Pierwszą i najważniejszą metodą jest aktualizacja ekipunku, polegająca na wstawianiu do slotów prefabrykatów w oparciu o ich pozycję lokalną [22].

[ Config.cs ]

```
public List<string> global_ItemList = new List<string>();
public List<string> global_ItemActiveList = new List<string>();
//-----Czyszczenie slotów
/* Czyszczenie ekipunku z obiektów. Pętla przejdzie po slotach sprawdzając
czy posiadają jakieś pod-objekty i jeśli tak to je niszczy. Bez tego dodanie
przedmiotów na nowo sprawi nakładanie się grafik na istniejące. */
public void ResetEquip(){
    GameObject equip = GameObject.Find("Hero/Equip").gameObject;
    for (int i = 1; i < equip.transform.childCount; i++){
        GameObject equ = equip.transform.GetChild(i).gameObject;
        if (equ.transform.childCount > 0)
            Destroy(equ.transform.GetChild(0).gameObject);
    }
}
//-----Wyświetlenie przedmiotów
public void UpdateEquip(){
    if (GameObject.Find("Hero").GetComponent<ControllerPlayer>().equipActive){
        /* Odświeżono jednocześnie z ekipunkiem. */
        GameObject equip = GameObject.Find("Hero/Equip").gameObject;
        /* Oczyszczenie slotów z obiektów. */
        ResetEquip();
        /* Wstawienie wyekwipowanych przedmiotów. */
        for (int i = 0; i < global_ItemActiveList.Count; i++){
            /* Złapanie błędu w razie braku prefabrykatu. */
            try{
                string itemName = global_ItemActiveList[i] + "_Item";
                /* Przygotowanie prefabrykatu do użycia. By go odnaleźć wykorzystano
                nawę jako identyfikator co omówiono wcześniej. */
                GameObject addItem = (GameObject)Resources.
                    Load("prefabs/Items/" + itemName, typeof(GameObject));
                /* Wstawienie obiektu do slotu. Kolejność wstawiania odbywa się w
                kolejności przedmiotów na liście. */
                Instantiate(addItem, equip.transform.
                    GetChild(17 + i).gameObject.transform, false);
            } catch { Debug.Log("Can't load active item."); }
        }
        /* Ta część działa niemalże identycznie, ale dla niewyekwipowanych
        przedmiotów. */
        for (int i = 0; i < global_ItemList.Count; i++){
            try{
                string itemName = global_ItemList[i] + "_Item";

```

```

        GameObject addItem = (GameObject)Resources.Load("prefabs/Items/" +
            + itemName, typeof(GameObject));
        Instantiate(addItem, equip.transform.GetChild(i + 1)
            .gameObject.transform, false);
    } catch { Debug.Log("Cant load item"); }
}
}

//-----Dodanie przedmiotu
/* Po prostu dodanie nazwy przedmiotu do listy. */
public void AddItem(string name){
    globalItemList.Add(name);
}

//-----Usunięcie przedmiotu
/* Usunięcie po wskazaniu miejsca w hierarchii. */
public void DropItem(int eqIndex){
    globalItemList.RemoveAt(eqIndex - 1);
    UpdateEquip();
}

//-----Użycie przedmiotu
public void UseItem(GameObject item, int eqIndex){
    bool search = true;
    string name = item.GetComponent<ItemInfo>().item.ObjectName;
    /* Jeśli typ przedmiotu to aktywny lub broń. */
    if (item.GetComponent<ItemInfo>().item.Type == "Weapon" ||
        item.GetComponent<ItemInfo>().item.Type == "Active"){
        /* Jeśli jest w slotach wyekwipowanych. */
        if (eqIndex >= 17){
            for (int i = 0; i < globalItemActiveList.Count; i++){
                /* Jeśli nazwa pasuje to zakończ szukanie, a przedmiot z aktywnego
                wstaw w nieaktywne. */
                if (globalItemActiveList[i] == name && search){
                    ItemStatus(false, globalItemActiveList[i], eqIndex);
                    search = false;
                }
            }
            /* Jeśli jest w slotach niewyekwipowanych. */
        } else{
            for (int i = 0; i < globalItemList.Count; i++){
                /* Jeśli nazwa pasuje to zakończ szukanie, a przedmiot z nieaktywnego
                wstaw w status aktywnego. */
                if (globalItemList[i] == name && search){
                    ItemStatus(true, globalItemList[i], eqIndex);
                    search = false;
                }
            }
        }
    }
    /* Jeśli typ przedmiotu to jedzenie wykonaj użycie po stronie skryptu owego
    przedmiotu. */
    else if (item.GetComponent<ItemInfo>().item.Type == "Food"){
        if (LoadedData.life < MaxLife){
            item.GetComponent<ItemInfo>().ItemUse(eqIndex);
        }
    }
}
}

//-----Wyekwipowanie/Zdjęcie
/* Wyekwipuj bądź zdejmij przedmiot. */
public void ItemStatus(bool active, string name, int eqIndex){
    GameObject equip = GameObject.Find("Hero/Equip").gameObject;
    if (active){
        if (globalItemActiveList.Count < 4) {
            /* Przerzucenie z listy nieaktywnych do aktywnych. */
            globalItemActiveList.Add(name);
            globalItemList.RemoveAt(eqIndex - 1);
            /* Wywołanie funkcji po stronie ekwipunku w celu zmiany wyposażenia ze
            strony wizualnej, tj. na postaci. */
            equip.GetComponent<ControllerEquip>().Active(active, name);
        }
    }
}

```

```

        }
    }else{
        /* Przerzucenie z listy aktywnych do nieaktywnych. */
        global_ItemList.Add(name);
        global_ItemActiveList.RemoveAt((17 - eqIndex) * -1);
        /* Wywołanie funkcji po stronie ekwipunku w celu zmiany wyposażenia ze
        strony wizualnej, tj. na postaci. */
        equip.GetComponent<ControllerEquip>().Active(active, name);
    }
    /* Odświerzenie stanu ekwipunku. */
    UpdateEquip();
}

```

Tak prezentuje się podstawowy, sprawnie działający ekwipunek. Dodano kilka pomocniczych funkcji typu usuwanie przedmiotu po nazwie, których nie ma sensu dodatkowo opisywać, gdyż nie różnią się wiele od tego co wyżej. Do zapisu gry dodano również pętle montujące z listy przedmiotów wyekwipowanych, niewyekwipowanych oraz tych do usunięcia, trzy długie stringi, które zostaną zapisane razem z save'em. Natomiast przy ładowaniu nastąpi proces odwrotny.

#### ❖ Sterowanie 2.0:

„*Last Adventure*” jako gra mająca cechy metroidvanii musi posiadać coś co rozszerza możliwości gracza. Jednym z takich przedmiotów są rakietowe buty pozwalające na szarżę w przód zarówno na ziemi jak i w powietrzu oraz dodatkowy wyższy skok po wykonaniu podstawowego. Z takimi założeniami wiąże się modyfikacja sterowania, dla tego teraz przedstawione zostaną potrzebne zmiany. Po pierwsze skok wykonano z pomocą switch'a, wyodrębniając cztery stany: „*stay*”, „*jumping*”, „*double*” i „*landing*”. Pierwszy to stan gotowości do skoku, drugi wykonanie skoku, trzeci ewentualne wykonanie skoku drugiego o ile założone są specjalne buty oraz czwarty, który w momencie zetknięcia bohatera z podłożem informuje o resecie ustawień i przygotowaniu stanu „*stay*”. Ostatnie działanie wywoła funkcję „*Landing()*”, czyli odtworzy się dźwięk, powstanie kurzu, stan przełączy się na pierwszy, itp. Samo skakanie wykonano również w sposób kontrolowania wysokości, co oznacza, że im dłużej przycisk będzie wciśnięty tym wyżej skoczy bohater gry.

Wykonanie tak zwanego „*Dash'a*”, czyli szarży, możliwe będzie po stronie kontrolera bohaterem w „*FixedUpdate()*”. Zabieg podobny jest do nadania siły skoku, jednak w przypadku osi x, siła musi zostać nadawana jakiś czas. Stworzono zatem zegar, który odmierza od wartości dodatniej do ujemnej, gdy dash jest aktywny, zaś po osiągnięciu wartości -2, wystąpi możliwość ponownego użycia. W grach taki czas odnowienia umiejętności nazwano „*Cooldown*” i występuje w większości produkcji. Gracz wtedy jest

zmuszyły wyczuć możliwości ruchowe oraz przemyśleć strategiczne wykorzystanie zdolności. Co do skoku, odnowi się w momencie kontaktu z podłożem bądź wybranym obiektem. Zaimplementowano możliwość podskakiwania niczym na trampolinie lądując na szczycie slim'ów oraz latających zielonych meduz, co będzie wyzwaniem w Sky-Town, gdyż drogi tam od dawna nie istnieją. Mówiąc wprost, trzeba po nich skakać by przejść dalej, a w przypadku zlecenia w dół, gracz wróci do mapy wcześniejszej obok wyrzutni przy Sally. Teraz nieco więcej kodu:

[ ControllerPlayer.cs ]

```
void Update(){
    /* Wywołanie funkcji dla lądowania po skoku. */
    if (grounded && jumpMode != "stay")
        if (jumpMode == "landing" || jumpMode == "double") Landing();

    /* Przełącznik trybu skoku. */
    switch (jumpMode){
        /* Tryb stay oznacza gotowość do skoku. Jeśli gracz nie podjął dialogu i
        wcisnął przycisk skoku, ustawia tryb na skakanie... */
        case "stay":
            if (Input.GetButton("Jump") && !inDialog){
                if (move == 0){
                    /* Efekt kurzu przy skoku w pionie z nadaną autodestrukcją. */
                    Destroy(Instantiate(DustUpPrefab, groundContact.position,
                        Quaternion.identity), DestroyTime);
                }else{
                    /* Efekt kierunkowego kurzu przy skoku. */
                    GameObject tmp = Instantiate(DustDirectionPrefab,
                        groundContact.position, Quaternion.identity);
                    /* Kierunek kurzu. */
                    tmp.GetComponent<SpriteRenderer>().flipX = !goRight;
                    /* Nadanie autodestrukcji po czasie. */
                    Destroy(tmp, DestroyTime);
                }
                /* Efekt audio. */
                Destroy(Instantiate(audioJump, transform, false), 0.3f);
                /* Reset zegara skoku oraz nadanie trybu. */
                jumpClock    = jumpTime;
                jumpMode     = "jumping";
            }
            break;

        /* ... w nim zaś wznosi się tak długo na ile pozwala jumpClock, bądź do
        momentu puszczenia przycisku. Potem jeśli gracz posiada buty rakietowe,
        aktywowany zostanie tryb double dla dodatkowego skoku, inaczej przejdzie
        w lądowanie. */
        case "jumping":
            if (Input.GetButton("Jump") && jumpClock > 0){
                /* Nadanie siły skoku. */
                body.velocity = Vector2.up * jumpForce;
            }else{
                /* Sprawdzenie czy gracz posiada rakietowe buty. */
                if (config.CheckActiveItem("RacketShoes")) jumpMode = "double";
                else jumpMode = "landing";
            }
            break;

        /* Tu zaś wykonany zostanie dodatkowy skok, po czym tryb osiągnie
        ostatnią fazę, tj. landing. Teraz po wylądowaniu nastąpi reset do
        stay. */
    }
}
```

```

        case "double":
            if (Input.GetButtonDown("Jump") && !doubleJump){
                /* Efekt dźwiękowy przy skoku. */
                Destroy(Instantiate(audioDoubleJump, transform, false), 1f);
                /* Nadanie siły skoku. */
                body.velocity = Vector2.up * jumpForce * 3;
                /* Animacja. */
                animator.SetBool("Jump", true);
                /* Efekt kurzu przy skoku. */
                Destroy(Instantiate(DustDownPrefab, groundContact.position,
                    Quaternion.identity), DestroyTime);
                /* Przełączenie wartości. */
                doubleJump = true;
                jumpMode = "landing";
            }
            break;
        }

void FixedUpdate(){
    /* Odmierzanie czasów skoku oraz szarzy. */
    if (!grounded) jumpClock -= Time.deltaTime;
    if (timerDash >= -5) timerDash -= Time.deltaTime;

    if (grounded){
        /* Przy kontakcie z podłożem wykonaj szarzę lub ruch według sterowania. */
        if (timerDash > 0) DashAhead();
        else body.velocity = new Vector2(move, body.velocity.y);
    }else{
        /* Wyłącz grafikę cienia. */
        PlayerShadow.GetComponent<SpriteRenderer>().enabled = false;
        /* Przy braku kontaktu z podłożem wykonaj szarzę w powietrzu lub ruch
        według sterowania. */
        if (timerDash > 0) DashAhead();
        else{
            /* Podczas przebywania w powietrzu rośnie grawitacja dając efekt
            zwiększenia prędkości spadania, co jest bardziej naturalne. */
            if (body.gravityScale < 5f) body.gravityScale += 0.05f;
            body.velocity = new Vector2(move, body.velocity.y);
        }
        /* Jeśli bohater spadnie poza mapę to umiera w celu odrodzenia.
        Zabezpieczenie na wypadek ewentualnych błędów z kolizjami podłożą. */
        if (transform.position.y < -50) config.OnePunchDeath();
    }
}

/* Dodanie efektów lądowania oraz reset ustawień skoku. */
public void Landing(){
    jumpMode      = "stay";
    dashDown     = false;
    doubleJump   = false;
    body.gravityScale = 3;
    animator.SetBool("Jump", false);
    PlayerShadow.GetComponent<SpriteRenderer>().enabled = true;
    Destroy(Instantiate(audioGrounded, transform, false), 0.1f);
    Destroy(Instantiate(DustDownPrefab, groundContact.position,
        Quaternion.identity), DestroyTime);
}

public void DashAhead(){
    /* Ustalenie kierunku. */
    int dir = (transform.localScale.x < 0) ? -1 : 1;
    /* Nadanie pędu. */
    body.velocity = new Vector2(dir * 100, body.velocity.y);
    /* Efekt eksplozji przy starcie. */
    if (DashMaker == 0){
        Destroy(Instantiate(ExplodePrefab, transform.position,
            Quaternion.identity), 1f);
    }
}

```

```

        }
        /* Dodanie efektu kurzu co 3 pętle. */
        DashMaker++;
        if (DashMaker > 3f){
            DashMaker = 1;
            Destroy(Instantiate(DustDownPrefab, transform.position,
                Quaternion.identity), 1f);
        }
    }
}

```

Po raz kolejny całość działa znakomicie, a samo sterownie z użyciem podwójnego skoku oraz szarży jest o wiele szybsze i bardziej komfortowe. Gracz jest w stanie dostać się do miejsc wcześniej nie dostępnych, a nowo powstałe ruchy specjalne stają się nieodłączną częścią mechaniki zręcznościowej. Podobnie jak przygotowano buty rakietowe, tj. przez modyfikacje skryptu tak też dodano ulepszenie do strzelby, co po stronie kodu oznacza zmianę parametrów pocisku na podstawie których wróg otrzymuje obrażenia.

#### **4.8 System walki, przeciwnicy i bossowie (krok 10).**

System walki, jak ustalono w części teoretycznej, będzie dystansowy jeśli chodzi o ataki bohatera, zaś u przeciwników głównie kontakt bezpośredni wywoła obrażenia co wymusi podejście zręcznościowe. Ujmując wprost, gracz musi atakować i nie dać się dotknąć. Sam zasięg broni zostanie ograniczony by nie dać znacznej przewagi osobie grającej. Przeciwnicy poruszać się będą z pewnym zestawem ruchów, zaprojektowanym po stronie skryptów oraz animacji tak by z zachowania, widoczne były momenty podjęcia ataku. Takie podejście da szansę na reakcję ze strony gracza w postaci uniku bądź kontratak.

❖ Atak:

Bohater gry dzierży broń w postaci strzelby, tak więc atak wyprowadzany jest na dystans. Każdy strzał tworzy nowy obiekt mający rolę wystrzelonej kuli. Jej celem rzecz jasna jest zadanie obrażeń trafionemu przeciwnikowi. Stworzono zatem obiekt z nazwą „*Bullet*” po czym dodano prostą grafikę pocisku. Sam przemieszczający się element nie robi najlepszego wrażenia, więc dodano zaanimowany obiekt zmniejszającego się prostokąta, co daje bardzo satysfakcjonujący efekt, powielany jest podczas lotu co niewielką odległość. Obiekt po przelecieniu odpowiedniego dystansu ulegnie autodestrukcji, tworząc przy tym nowy obiekt na podstawie prefabrykatu z animacją eksplozji, co zadziała identycznie podczas kontaktu z wrogą jednostką. Na koniec dodano krótkie audio wystrzału odtwarzane raz przy stracie, dzięki czemu nie wystąpiła konieczność dopisywania

kodu zarządzającego dźwiękiem w skrypcie strzelby. Wracając zaś do broni, dodano prefabrykat animowanego dymu przy strzale by uzyskać lepszy efekt wizualny. Całość prezentuje się zacne co widać na obrazie 37.



Rysunek 37 - Atak dystansowy.

Pocisk składa się z komponentów takich jak „*Sprite Renderer*” by wyświetlał grafikę, domyślnie stworzoną w odcieniach szarości co później posłuży do zmiany koloru z poziomu kodu, z „*Box Collider'a*”, by wyłapać zderzenie z wrogiem oraz skrypt. Kod zająłby sporo stron dlatego zostanie on pominięty, ale opisana zostanie logika działania kuli:

- Na starcie sprawdzono, w którą stronę zwrócona jest broń, by nadać odpowiedni kierunek przemieszczania pocisku. Wykorzystano do tego skalę lokalną x obiektu ze strzelbą, która w wartości ujemnej wskazuje zwrot w lewą stronę, a przy dodatniej w prawą. Samo przemieszczanie przycisku umieszczono w „*Update*”, co wykonano przez zmianę w komponencie „*Transform*” pozycji x dodając dwie jednostki co odświeżenie.
- Na starcie ustalono również początkową pozycję pocisku, zaś w „*Update*” dopisano sprawdzanie odległości. Jeśli ta przekroczy 100 jednostek, wywołana zostanie funkcja autodestrukcji polegająca na stworzeniu z prefabrykatu eksplozji, dźwięku oraz usunięciu samej kuli.
- Na starcie zostało też sprawdzone czy gracz posiada przedmiot specjalny, a jeśli tak to kolor kuli zostanie zmieniony przez podany w inspektorze do publicznej zmiennej typu „*Color*”. W tym przypadku jest to jaskrawa zieleń przy założonej modyfikacji

na broń oraz czerń dla klasycznych pocisków. Z racji, iż grafika smugi posiada biały kolor, ją również odpowiednio przefarbowano, a w „Update” wstawiono pętlę, która tworzący obiekty z prefabrykatu co pewien dystans. Smuga zostaje zniszczona przez wywołanie funkcji autodestrukcji na końcu animacji.

- Ostatni krok to wskazanie zadawanych obrażeń, co nastąpi przy kontakcie z wrogiem. Domyślnie kula odbiera jeden punkt zdrowia, zaś z modyfikacją, aż cztery. By wywołać odpowiednią funkcję u przeciwnika, należy wskazać nazwę skryptu jako komponent, co oznacza, że trzeba rozróżnić kolizje po stronie kuli. Stworzono zatem „switch” sprawdzający „Tag” napotkanego obiektu, który dobiera odpowiednie działanie oraz skrypt przeciwnika.

❖ Przeciwnik Slime:

Najpopularniejszy, podstawowy przeciwnik w grach fantastycznych, przeważnie nie zadający wielkich obrażeń. W tym projekcie nieco zmieniono zachowanie kultowej galaretki na całkiem pasywną o ile nie jest atakowany i ekstremalnie groźną w momencie zadania obrażeń. W pierwszym wypadku skrypt wygeneruje losowe przemieszczanie się od lewej do prawej, na podstawie ustalonych granic, zaś w drugim przypadku nie dość, że slime zaatakuje trującym gazem, to w dodatku wygeneruje małą armię galaretek podróżującą za bohaterem, z ograniczeniem życia do około 10 sekund. Na pierwszy rzut oka nie wydaje się to być groźne jednak spoglądając na fakt posiadania trzech punktów zdrowia oraz generowanej co trzy sekundy małej, pięcio-glutkowej, skaczącej armii bomb, gracz szybko zrozumie, że ma kłopoty. W przypadku pokonania przeciwnika, ten po czasie się zregeneruje jakby nigdy nic [19]. Owo podejście zakładało zdobywanie punktów, które można wydać na przedmioty oraz rozwój postaci, jednak zabrakło czasu na pełne prowadzenie tego systemu. Graczowi musi wystarczyć satysfakcja z wygranej.

Sam skrypt nie jest skomplikowany. Przeciwnik podejmuje akcje na podstawie prostych zmiennych, jak wcześniej wspomniana pasywność przechodząca w agresję, gdy zadane zostaną obrażenia. Podczas ataku działają liczniki pełniące funkcję odnowienia umiejętności galaretki, dzięki czemu przeciwnik nie atakuje non stop jak szalony. Slime przejdzie w stan pasywny jeśli odległość między nim, a graczem odpowiednio się zwiększy. Ta proste podejmowanie decyzji wykonano z wykorzystaniem instrukcji warunkowych „if”, zaś w kolejnym przeciwniku „switch”, co miało na celu wybranie skuteczniejszej oraz bardziej komfortowej metody [22].

❖ Przeciwnik Duch:

Tu nieco zmieniono podejście do zachowania przeciwnika. Po pierwsze będzie to przeciwnik nieśmiertelny, chyba że dokona ataku, wtem cały eksploduje zadając bohaterowi obrażenia. Taki kamikadze. Na dodatek będzie się pojawiał zawsze za plecami gracza i starał się dolecieć do niego w celu zaatakowania. Pojawi się jedynie w mrocznym lesie, co oznacza, że kamera będzie zbliżona, a to z kolei ograniczy czas na reakcję. Taki zabieg może czasem nieźle przestraszyć, gdy w tle słuchać złowieszczy śmiech, a przeciwnika jeszcze nie widać. Jedyną opcją uniknięcia obrażeń jest spojrzenie na ducha, co go spłoszy (lecz nie na długo). Pociski ze strzelby nie zrobią wrażenia na wrogu, przeznikną przez niego jak, no właśnie, przez ducha. Sam skrypt jest nie krótszy niż w przypadku slime'ów, to też opisano jedynie logikę.

Jak w przypadku galaretek wystąpił stan nieagresywny oraz agresywny, gdzie w pierwszym przypadku duchy latają w obrębie punktu startowego, który jest środkiem całego obiektu. Dzięki komponentowi kolizji ustalono strefę reakcji na bohatera. Oznacza to, że w momencie wejścia w strefę, duch przyjmuje stan agresji, co w praktyce zmienia wartość zmiennej string'owej o nazwie „*switcher*” na „*wait*”. Przeciwnik stanie się niewidoczny i będzie czekał, aż gracz przejdzie dalej by znaleźć się za jego plecami. To wykonano z pomocą porównania pozycji x bohatera i ducha z pewnym marginesem by przeciwnik nie pojawił się natychmiast. Spełnienie warunku sprawi, że zjawa się ujawni, by zacząć podążać za bohaterem. Samo podążanie działa na zasadzie podobnej do ruchu kamery. Wykorzystano „*SmoothDamp*” by uzyskać efekt zmniejszania prędkości oraz płynność w momencie zbliżenia się do gracza, takie wyhamowanie. Ma to dać osobie grającej czas na reakcję. Całość użycia „*switch'a*” działa podobnie jak w przypadku skoku w Sterowaniu 2.0 omawianym w rozdziale poprzednim. Słowo klucz definiuje aktualną akcję. W momencie gdy gracz wyjdzie poza strefę aktywności ducha, przestaje go śledzić i wraca do swobodnego lotu, który polega na losowym wybraniu współrzędnych docelowych w obrębie punktu startowego za każdym razem, gdy dolegi do aktualnie wskazanych.

Wracając do walki „*switch*” kontra „*if*”, dużo wygodniejsze i przejrzyste jest użycie switch'ów. Kod jest uporządkowany, a akcja podejmowana na podstawie wartości jednej zamiast kilku zmiennych.

❖ Bossowie:



Rysunek 38 - Przykład ruchu na podstawie drugiej zasady animacji.

Zasadniczo choć mają zwiększoną odporność, pojawiają się jedynie raz i posiadają zestaw unikalnych ruchów, nie różnią się wiele od mniejszych przeciwników. Zasada działania polega na tym samym, skrypt przeciwnika reaguje na zachowanie gracza na podstawie prostych sygnałów. Jednak dla bossów dodatkowo jest parę mechanik typu: coś co wyzwała walkę, obszar na którym dzieje się pojedynek oraz pasek życia przeciwnika. Wzorując się m.in. na serii „Dark Souls”, „Hollow Knight”, czy choćby „UnEpic”, można zauważać, że ważną częścią w zachowaniu bossów są zestawy ataków możliwe do zapamiętania, wyrażające swój zamiar chwilę przed wykonaniem. W rozdziale teoretycznym o animacjach przedstawiono to jako „Anticipation”, dzięki czemu gracz jest w stanie odpowiednio szybko zareagować unikiem, bądź kontratakiem [3]. W momencie gdy slime atakuje trucizną, wywołana jest charakterystyczna animacja, w której podnosi się, naprężając galaretowe ciało i zmieniając kolor, by zaraz opaść wypuszczając trujący gaz. Strażnik zaś przyjmuje pozę świadczącą o chęci wykonania szarży. Wracając do paska zdrowia, jest to nic innego jak część interfejsu graficznego złożonego z ramki, napisu oraz paska, a całość ulepszona o animację wejścia/wyjścia ze strefy kamery. W momencie rozpoczęcia walki z bossem, do tekstu wprowadzona zostanie jego nazwa, a pasek życia wypełni się wykorzystując skalowanie lokalne od 0 do 1 (0% - 100%). Zaś równo stan życia jak i całe zachowanie bossa odświeża metoda „FixedUpdate”. Każdy po pokonaniu doda zdarzenie do listy eventów, dzięki którym będą w stanie ulec autodestrukcji na starcie o ile takie zostali już pokonani. W ten sposób uzyskają swą unikatowość występując jedynie raz.

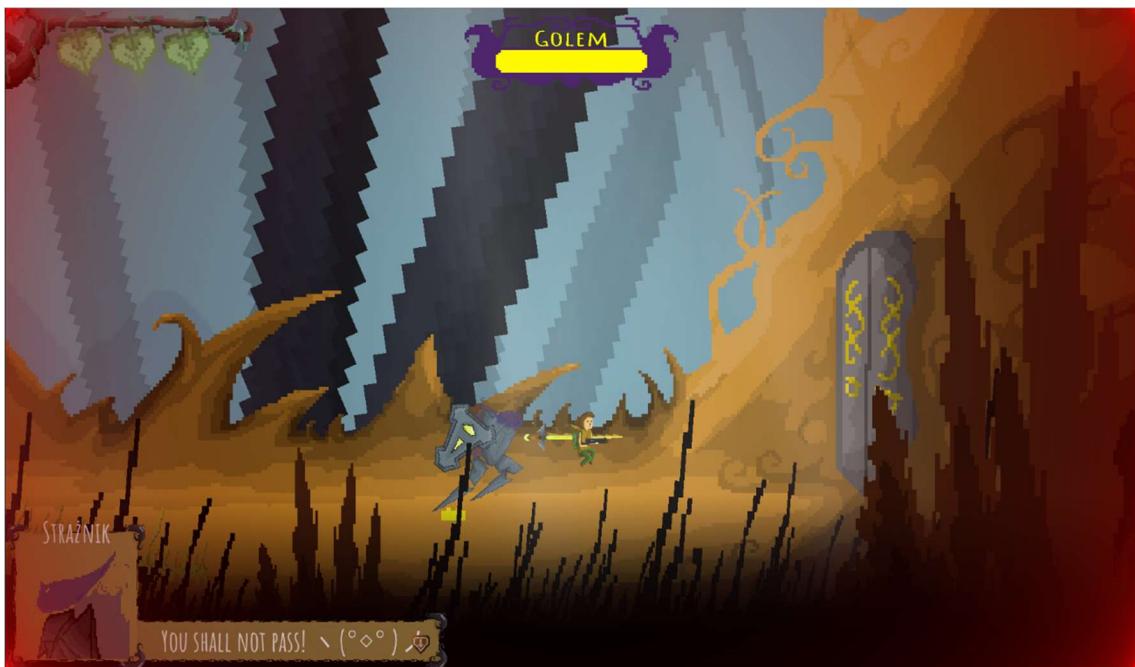
**Giga Slime** to w rzeczywistości powiększona wersja klasycznego slime'a z dodanymi cechami dodatkowymi jak tworzenie lecących pocisków, podgląd stanu zdrowia, odłączenie regeneracji po śmierci, pojawia się tylko raz (za sprawą autodestrukcji obiektu, jeśli wykryty zostanie event w liście zdarzeń), po pokonaniu doda zdarzenie do listy oraz ma zwiększone zdrowie. Fabularnie umieszczono w nim przedmiot, a konkretnie za nim, będący pojemnikiem wchłaniającym toksynę, który posłuży do stworzenia modyfikacji broni. Pojemnik zostanie uwolniony w momencie zniszczenia bossa, który do tej pory blokował do niego dostęp.



Rysunek 39 - Boss: Giga Slime.

**Golem**, inaczej strażnik wrót, pierwszy, a zarazem ostatni boss, a to za sprawą odporności na pociski bez modyfikacji. W zasadzie celem gry jest zdobycie przedmiotów umożliwiających stworzenie ulepszenia, które pozwoli na stoczenie równej walki. Przy okazji w trakcie przygody gracz zdobędzie rakietowe buty dające znaczną przewagę w starciu. Zatem walka w początkowej fazie gry jest jak najbardziej możliwa, jednak z góry przegrana. Zestaw ruchów dla odmiany przygotowany został z użyciem animatora, ale nie animacji poklatkowych, co polegało na stworzeniu kluczowych pozycji każdego elementu bossa. Jest to metoda szybsza oraz lepsza w modyfikacji, choć w wizualnym efekcie końcowym, wygląda gorzej, traci charakter oraz klimat. Zachowanie bossa nie odstaje od slime'a, ponieważ i tu użyto prostych przełączników oraz „Cooldown'ów” umiejętności. Golem do momentu otrzymania obrażeń lub przekroczenia miejsca, w którym stoi jest całkiem niegroźny i dopiero prowokacja wywoła w nim chęć mordu. W momencie, gdy bohater zbliży się za bardzo, strażnik zasłania się tarczą będąc niewrażliwym na ataki.

Dopiero po oddaleniu się, bądź przeskoczeniu wroga, ten zacznie wykonywać szarże. W momencie gdy szarży wykonać nie może, ponownie zasłania się tarczą by przeszekać okres odnowienia umiejętności.



Rysunek 40 - Boss: Starżnik Golem.

**Yamiko**, w zasadzie nie jest bossem, a NPC, jednak całość zaprojektowano tak by w pierwszej chwili przypominała rozpoczęcie walki. Początkowo faktycznie miało dojść do starcia, jednak z braku wystarczającej ilości czasu, zdecydowano o stworzeniu pewnego zwrotu akcji. W wyniku dialogu gracz otrzyma ostatni przedmiot niezbędny do stworzenia wzmocnienia dla broni.



Rysunek 41 - Boss, który został NPC: Yamiko.

#### 4.9 Fin, czyli scena kończąca (krok 11).

Scena kończąca, czyli nic innego jak podziękowania oraz napisy z imionami twórców. Do jej stworzenia wykorzystano menu główne, wycinając część odpowiedzialną za przyciski, a dodając animację wejścia tytułu oraz blok napisów ze skryptem przewijania umieszczonym w Canvas. Dodano także kod reagujący na przytrzymanie przycisku w celu przyśpieszenia przewijania tekstu. Samo przewijanie wywołane zostanie przez animację wejścia tytułu, a konkretne na jej końcu. Skrypt będzie sprawdzał czy tekst osiągnął odpowiednią wysokość, by ostatecznie przeładować scenę do pierwszej. Nastąpi to ustawiając wcześniej w Config'u zmienna Fin na false, a Esc na true, co zadziała jak wcisnięcie klawisza powrotu do menu głównego.



Rysunek 42 - Napisy końcowe.

## 5. Podsumowanie

### 5.1 Produkt końcowy i porzucone pomysły.

„Last Adventure”, czyli produkt końcowy będący grą stworzoną z myślą o zdobyciu wiedzy oraz doświadczenia ze świata developerów. Muszę przyznać, że podejmując taki, a nie inny temat nie sądziłem, iż zakres prac będzie aż tak ogromny. Mimo nieprzespanych nocy, dla mnie jako fana gier, była to niezwykła przygoda, począwszy od zapoznania ze środowiskiem Unity 3D, przemyślenia mechanik oraz fabuły, prze tworzenie animacji, udźwiękowienia i dialogów po pisanie skryptów nadających nieco życia wirtualnemu światu. Do tej pory choć wiedziałem, że stworzenie tego typu produkcji wymaga wiele wysiłku, to dopiero teraz poznalem jak wiele jest to pracy. Mój projekt nie wyszedł idealny, a i rozgrywka nie trwa zbyt długo (około godzinki), jednak spełnia mnie duma, gdyż podobałem zadaniu tworząc grę w pełni grywalną, wydajną oraz naprawdę niebrzydką. Podsumowując. W ciągu czterech miesięcy (włączając czas nauki), powstała gra przygodowo-zręcznościowa z elementami walki, prostą fabułą oraz zróżnicowanymi lokacjami dająca około godziny frajdy.

- Wykonane elementu:
  - Zróżnicowany teren oraz roślinność z użyciem niewielkiej ilości grafik, co osiągnięto przez ich modyfikację (rozciąganie, obracanie, itp.).
  - Sterowanie podstawowe, tj. regulowany skok (im dłużej zostanie przytrzymany tym wyżej bohater skoczy, oczywiście nałożono ograniczenie), ruch w lewo oraz prawo, atak, kucnięcie.
  - Zapis oraz odczyt stanu gry przechowujący jedynie kluczowe informacje. Punkty zapisu gry w postaci lewitującego głazu.
  - Menu główne dla gry z ustawieniami początkowymi, możliwością zmiany języka oraz poziomu głośności efektów dźwiękowych i muzyki.
  - Przeładowanie scen oraz przejście między mapami.
  - Obiekt będący magazynem danych oraz funkcji, nie poddawany usunięciu przy zmianie sceny oraz występujący jako jedyny na każdej scenie.
  - Zarządzanie ekwipunkiem, czyli podnoszenie przedmiotów, używanie ich, wyekwipowanie oraz zastosowanie przedmiotów fabularnych.
  - Przedmioty zdatne do użycia, wyekwipowania oraz fabularne. Buty rakietowy zmieniają podejście do poruszania postacią, nadając możliwość dodatkowego

skoku oraz szybkiej szarży w przód. Kryształ esencji zapewni pełną regenerację zdrowia w momencie śmierci, a modyfikacja do broni zwiększy moc pocisków oraz wywoła wrażliwość na atak u „Golema”.

- System walki, otrzymywanie obrażeń oraz ich zadawanie. Użycie wyekwipowanego przedmiotu do ataku, w tym przypadku strzelba generuje pocisk zadający obrażenia.
- Postacie w świecie gry (NPC) z dialogami oraz zadaniami. Tutorial zawarty w pierwszym dialogu.
- System dialogów oparty na tekście i skoku po liniach dialogowych, wykorzystujący uniwersalne funkcje wywoływanie dzięki słowom kluczowym.
- Mniejsi przeciwnicy spotykani na mapach.
- Bossowie lokacji, stanowiący wyzwanie oraz część zadania.
- Muzyka w tle, dźwięki odtwarzane podczas czynności oraz zmiana muzyki podczas walki z bossami.
- Zmiana elementów otoczenia z pomocą systemu zapisującego zdarzenia oraz listy przedmiotów podniesionych w celu odtworzenia stanu obiektów na mapie oraz momentów w dialogowych.

Niestety nie udało się zrealizować wszystkich pomysłów. Jednym z fajniejszych był system tworzenia przedmiotów z innych przedmiotów w celu późniejszego wykorzystania, jak np. fiolki z leczącą substancją. Podobnie miał wystąpić pasek życia oraz stany zatrucia i wzmacniania. Pominięto również mechaniki rozwoju postaci, w których gracz mógłby inwestować zdobyte z pokonanych przeciwników punkty w zwiększenie zdrowia, szybkości ruchów, itp.

- Pominięte elementu:
  - System tworzenia przedmiotów z posiadanych (Crafting).
  - Umiejętności specjalne zahaczające o magię.
  - Dziennik zadań z mapą świata.
  - Pasek życia, energii oraz ikony zatrucia i wzmacniania.
  - Ulepszanie postaci z zebranych punktów oraz odblokowywanie umiejętności.
  - Sklep z przedmiotami oraz skrzynia na przechowanie przedmiotów.

## 5.2 Test wydajności oraz perspektywa rozwoju.

Ostatecznie udało się wykonać około ¾ zakładanych rzeczy, co mimo wszystko jest bardzo satysfakcjonujące. Spisano również kilka pomiarów na kilku maszynach w celu sprawdzenia stabilności gry oraz osiągniętej wydajności. Do uzyskania ilości klatek wykorzystano skrypt udostępniony w sieci [28], który odświeża element tekstowy w Canvas. Pomiary procesora i pamięci pobrano z windowsowego menadżera zadań. Oto wyniki:

- CPU: Core Intel i7-7700HQ, 4 x 2 x 2.80 – 3.80 GHz
  - Intel HD Graphic 630: 950 – 1150 MHz
  - Rozdzielcość: 1920 x 1080
  - Zużycie RAM: 180 – 330 MB
  - Zużycie CPU: 50 – 70%
  - Menu główne: 2300 – 2500 FPS
  - Lokacja „*Big Tree*”: 314 – 657 FPS
  - Lokacja „*Catacumbs*”: 589 – 972 FPS
  - Lokacja „*Sky Town*”: 332 – 373 FPS
  - Lokacja „*Cave*”: 1273 – 1418 FPS
- CPU: Intel Pentium 2020M, 2 x 2.40 GHz
  - Intel HD Graphic: 650 – 1100 MHz
  - Rozdzielcość: 1600 x 900
  - Zużycie RAM: 325 MB
  - Zużycie CPU: 50 – 75%
  - Menu główne: 135 – 142 FPS
  - Lokacja „*Big Tree*”: 26 – 80 FPS
  - Lokacja „*Catacumbs*”: 43 – 76 FPS
  - Lokacja „*Sky Town*”: 62 – 80 FPS
  - Lokacja „*Cave*”: 60 – 72 FPS
- CPU: Intel Pentium 2020M, 2 x 2.40 GHz
  - nVidia GT720M: 625 – 938 MHz
  - Rozdzielcość: 1600 x 900
  - Zużycie RAM: 240 - 300 MB

- Zużycie CPU: 30 – 60%
- Menu główne: 238 – 246 FPS
- Lokacja „*Big Tree*”: 103 – 193 FPS
- Lokacja „*Catacumbs*”: 104 – 138 FPS
- Lokacja „*Sky Town*”: 130 – 135 FPS
- Lokacja „*Cave*”: 182 – 213 FPS

Wyżej zapisane wyniki wskazują na sporą niestabilność w generowaniu klatek. By zapobiec skokom, dodano do skryptu „*Config.cs*” linię ustawiającą limit [29]:

```
Application.targetFrameRate = 60;
```

Z obserwacji wartości fps’ów podczas gry wywnioskowano, iż spadki wystąpiły w momencie aktywacji przeciwników, działających ze skryptem odświerzającym ich stan. Ogólnie całość działa wystarczająco dobrze by z grą dały radę słabe jednostki, a ustawienie blokady klatek ogranicza nadużycia mocniejszych zestawów.

❖ Przyszłość projektu:

Jeśli chodzi o rozwój tego konkretnego projektu, nie zostanie on kontynuowany. W trakcie tworzenia narodził się pomysł o wiele ciekawszy ulepszający wszelkie mechaniki tu przedstawione oraz zakładający dodanie tych pominiętych. Znaczy to tyle, iż użyte rozwiązania jak najbardziej zostaną jeszcze wykorzystane oraz ulepszone w celu stworzenia projektu o wiele większego, ten zaś zostanie w pełni udostępniony na platformie GitHub użytkownika „*Morfeu5z*” pod nazwą „*LastAdventure*”.

### 5.3 Instrukcja oraz streszczenie.

Na początek warto opisać sterowanie. W projekcie zawarto możliwość podłączenia zarówno klawiatury jak i pada od Xbox'a 360/One:

- Klawiatura
  - Ruch prawo/lewo: strzałki L / R lub klawisze A / D
  - Skok: strzałka w górę lub klawisz W/Spacja
  - Kucnięcie: strzałka w dół lub klawisz S
  - Strzał: klawisz J / Ctrl

- Szarża: klawisz I
  - Unik w dół: będąc w powietrzu przycisk kucnięcia
  - Dodatkowy skok: będąc w powietrzu przycisk skoku
  - Włącz ekwipunek: Tab
  - Wyłącz ekwipunek: Tab / Esc
  - Menu Główne: Esc
  - Zatwierdź / Dalej: Enter
  - Poruszanie po menu: strzałki lub WSAD
  - Wyekwipuj / Zdejmij / Użyj: Enter
- 
- Pad Xbox 360 / One
    - Ruch prawo/lewo: lewa gałka
    - Skok: przycisk A
    - Kucnięcie: przycisk LB
    - Strzał: przycisk X
    - Szarża: przycisk RB
    - Unik w dół: będąc w powietrzu przycisk kucnięcia
    - Dodatkowy skok: będąc w powietrzu przycisk skoku
    - Włącz ekwipunek: przycisk SELECT
    - Wyłącz ekwipunek: przycisk SELECT / START
    - Menu Główne: przycisk START
    - Zatwierdź / Dalej: przycisk Y / A
    - Poruszanie po menu: lewa gałka
    - Wyekwipuj / Zdejmij / Użyj: przycisk Y / A

❖ Streszczenie / Poradnik do gry:

Gra zaczyna się w jaskini, gdzie bohater budzi się z brakiem wspomnień i możliwością swobodnej rozmowy, co jest wynikiem pewnej kłatywy. Wyjście z jaskini możliwe będzie po dialogu ze staruszkiem oraz zapisie gry siadając na stołku. Podczas rozmowy bohater dowie się jak trafił do dziadka oraz przedstawiona zostanie prosta instrukcja do gry. Do ekwipunku trafią: tajemnicza paczka, strzelba oraz jabłko. Pierwsze należy

dostarczyć do miasta co jest głównym celem gry. Strzelbę należy wyekwipować, zaś jabłko zjeść. Po odpoczynku, tj. zapisie gry przy stołku, możliwe jest wyjście z jaskini, jednak pierw warto podnieść przedmiot na stole alchemicznym „Kryształ wypełniony esencją”. Wyekwipowany przywraca całe zdrowie w momencie śmierci. W nowej lokacji idąc w lewo, gracz natrafi na zamknięte drzwi do katakumb, dlatego należy udać się w prawo gdzie przy starej wyrzutni stoi Sally. Po drodze napotkać można Slime'y, niegroźne stworzonka o ile nie zostaną sprowokowane. Npc opowie bohaterowi o tym, że próbuje naprawić urządzenie i pomóc byłaby nieoceniona. Aktualnie jest to misja pomocnicza, gdyż gra ma aż trzy ścieżki fabularne.

### **Uwaga!**

Od tego momentu opisane zostaną wszelkie sposoby na przejście gry dlatego też w celu uniknięcia spoilerów należy pomiń ten fragment.

**Obudzony strażnik:** Kierując się w prawo gracz napotka lewitującą skałę, jest to punkt zapisu, idąc dalej dotrze do mrocznego lasu. Straszą tam duchy chcące z zaskoczenia zaatakować przybysza, co czynią pojawiając się za plecami. Sposobem na nie jest spojrzenie im w oczy. Wychodząc z lasu gracz dotrze do inskrypcji na szczytce klifu. Nie jest w stanie rozszyfrować tekstu, ale może zeskoczyć w dół, gdzie znajduje się ogromna brama prowadząca do miasta. Niestety jest zamknięta. Obok leży księga szyfrów za pomocą której możliwe staje się odczytanie tajemniczego zapisu. Budzi to strażnika bramy „*Golema*”. Ten ostrzega, że zacznie walkę jeśli bohater przekroczy miejsce, w którym stoi i tak też się dzieje. Na chwilę obecną przeciwnik jest nie do pokonania w związku z czym, pomocy należy szukać u staruszka z początku gry. Ten zleca zdobycie kilku składników do stworzenia modyfikacji strzelby: katalizator, toksyna oraz płatek z esencji. Katalizator posiada Sally i odda go bez większych problemów. Niezbędne będzie wykonanie wcześniejszego zlecenia, tj. zdobycie narzędzi oraz płytka. Wracając do mrocznego lasu, trafi na most z dziurą. Wskakując do niej wyląduje na gałęzi na końcu której spoczywa maska przeciwgazowa, niezbędna do wyekwipowania przed skokiem w dół, tam unoszą się trujące gazy. Na dnie wąwozu, przy nieboszczyku znajduje się list oraz karta dostępu do katakumb, zaś na ciężarówce poszukiwane narzędzia. By dostać się do katakumb w celu odnalezienia płytka, należy pierw oddać narzędzia, by npc mógł naprawić generator. Teraz, wraz z kartą dostępu drzwi staną otworem. Należy założyć ponownie maskę gazową, gdyż dookoła unoszą się toksyczne opary. Na końcu pomieszczenia jest dziura, a

za nią skrzynie wraz z układem scalonym. Skacząc w odpowiednim momencie można tam się dostać, jednak w większości prób raczej spadnie się w dół, a to po to, by spotkać kolejne zwłoki i zyskać odrzutowe buty. Od teraz bohater jest w stanie wykonać podwójny skok, szarżę oraz unik w dół będąc w powietrzu. Aby wydostać się z jaskini i zdobyć toksynę, niezbędne będzie pokonanie drugiego bossa, czyli Giga Slime'a. Sposobem na niego jest skakanie na ostatnim przeciwniku, nie wskakując na półkę z bossem i strzelanie z dystansu. Po chwili można przejść dalej do wyrzutni, którą gracz trafi do wąwozu w mrocznym lesie. Oddając Sally płytę, ta naprawi wyrzutnię, dzięki czemu odblokowana zostanie nowa lokacja Sky Town. Jest to mapa głównie zręcznościowa, gdzie należy skakać po zielonych latających meduzach by dostać się z wysepki na wysepkę. Krocąc po podniebnych jeziorach, gracz naraża się na atak wodników wystrzelujących w jego stronę. Sporą szansę na unik daje kucnięcie lecz nie zawsze jest to skuteczne. Są tu również dwa punkty widokowe, pierwszy na ogromnej gałęzi przy pierwszym jeziorze, drugi na szczycie ruin fortecy, przy której swoje miejsce ma ostatni boss, ale nie boss. Jest to Yamiko (dla mojej ukochanej), NPC, który na pierwsze wrażenie ma przypominać przeciwnika. W trakcie rozmowy uzyskać można potrzebny kwiat, ostatni element do stworzenia modyfikacji. Faza końcowa to połączenie składników u staruszka przy stole alchemicznym i walka z „Golemem”. Po otwarciu bramy nie dane będzie do trzeć do miasta, ale pojawią się napisy końcowe z podziękowaniami oraz twórcami.

**Śpiący strażnik:** Tak naprawdę można wykonać wszystko powyżej nie budząc Golema, ani nie walcząc ze Slimem. Wystarczy nie odszyfrowywać inskrypcji, przeskoczyć Slima (buty są potrzebne w Sky Town) oraz skorzystać z sekretu drzwi by je otworzyć.

**Sekret zamkniętych drzwi:** Drzwi się zacięły, gracz dowie się o tym po powrocie od staruszka o ile nie obudzi golema. Po tym może je otworzyć i skończyć całą grę nim jeszcze ją dobrze zacznie.

## **Lista grafik**

Rysunek 1 - Początkowe dane dla nowego projektu.	8
Rysunek 2 - Nowy projekt z pustą sceną.	8
Rysunek 3 - Pasek narzędzi oraz warstw.	9
Rysunek 5 - Edycja obiektu z ustawieniem lokalnym i globalnym.	10
Rysunek 4 - Edycja wielu obiektów w trybie Center i Pivot.	10
Rysunek 6 - Hierarchia obiektów.	11
Rysunek 7 - Nowy, pusty obiekt - cechy w inspektorze.	12
Rysunek 8 - Zakładki: Projekt i Konsola.	12
Rysunek 9 - Scena z domyślną kamerą.	13
Rysunek 10 - Nowy projekt w aplikacji Piskel.	14
Rysunek 11 - Animacja obiektu twardego i miękkiego.	15
Rysunek 12 - Wyprzedzenie w animacji na przykładzie miotającego.	16
Rysunek 13 - Nowy projekt w Bosca Ceoil.	19
Rysunek 14 - Edytor fragmentów kompozycji w Bosca Ceoil.	20
Rysunek 15 - Instrument "Wood Block" z wartościami domyślnymi.	20
Rysunek 16 - Nowy instrument w Bosca Ceoil.	21
Rysunek 17 - Okno edycji dźwięku w Audacity.	22
Rysunek 18 - Dodawanie nowych warstw oraz ustawienie relac w Physic 2D.	40
Rysunek 19 - Ustawienia kamery.	41
Rysunek 20 - Dodanie nowego elementu przez wstawienie grafiki na scenę.	42
Rysunek 21 - Budowa bohatera w zakładce hierarchii.	42
Rysunek 22 - Animacja poklatkowa "Idle".	49
Rysunek 23 - Przejście animacji w animatorze Idle <-> Run.	49
Rysunek 24 - Animacja biegu.	50
Rysunek 25 - Animacja ekwipunku, skoku i kucania.	51
Rysunek 26 - Broń jako prefab oraz animacja strzału z wywołaniem funkcji.	53
Rysunek 27 - Animator strzelby z trzema animacjami.	53
Rysunek 28 - Animacja kucnięcia z przesunięciem pozycji.	56
Rysunek 29 - Animator bohatera z ustawionymi przejściami animacji.	57
Rysunek 30 - Pozycja UI z Rect Transform.	62
Rysunek 31 - Menu główne oraz podmenu gry.	64
Rysunek 32 - Interfejs w grze.	71
Rysunek 33 - Rozmieszczenie obiektów według warstw.	73
Rysunek 34 - Ruiny w SkyTown.	74
Rysunek 35 - Przygotowanie okna dialogowego.	76
Rysunek 36 - Ekwipunek na scenie oraz w hierarchii projektu.	82
Rysunek 37 - Atak dystansowy.	93
Rysunek 38 - Przykład ruchu na podstawie drugiej zasady animacji.	96
Rysunek 39 - Boss: Giga Slime.	97
Rysunek 40 - Boss: Starżnik Golem.	98
Rysunek 41 - Boss, który został NPC: Yamiko.	98
Rysunek 42 - Napisy końcowe.	99

## Bibliografia

- [1] D. E. Lavieri, *Getting Started with Unity 2018*, III red., Birmingham: Published by Packt Publishing Ltd., 2018.
- [2] M. Winiarski, „Podstawy Unity 3D,” grudzień 2018. [Online]. Available: <https://mwin.pl/unity3d-tutorial-0-podstawy-podstawa-niezbednik-uzytkowania-unity/>.
- [3] R. Schubert, „12 zasad animacji,” Grudzień 2018. [Online]. Available: <http://pananimator.pl/animacja/12-zasad-animacji/>.
- [4] A. Becker, „12 Principles of Animation (Official Full Series),” Grudzień 2018. [Online]. Available: <https://youtu.be/uDqjldI4bF4>.
- [5] M. Tomaszewski, „12 zasad animacji Disneya, czyli jak nakręcić idealny film animowany.,” Styczeń 2019. [Online]. Available: <https://lekturaobowiazkowa.pl/na-ekranie/12-zasad-animacji-disneya/>.
- [6] S. Haney, *Swift 3 Game Development*, II red., Birmingham: Published by Packt Publishing Ltd., 2017.
- [7] Unity Technologies, „2D Game Creation,” grudzień 2018. [Online]. Available: <https://unity3d.com/learn/tutorials/s/2d-game-creation>.
- [8] Resistance Studio, „Making your Pixel Art Game look Pixel Perfect in Unity3D,” grudzień 2018. [Online]. Available: <https://hackernoon.com/making-your-pixel-art-game-look-pixel-perfect-in-unity3d-3534963cad1d>.
- [9] N. Calice, „2D double/triple jump platformer controller - Easy Unity Tutorial,” grudzień 2018. [Online]. Available: <https://youtu.be/QGDeafTx5ug>.
- [10] A. Kramarzewski i E. D. Nucci, *Practical Game Design*, I red., Birmingham: Published by Packt Publishing Ltd., 2018.
- [11] C. Dickinson, *Unity 2017 Game Optimization*, II red., Birmingham: Published by Packt Publishing Ltd., 2017.
- [12] T. Asbjørn, „Save & Load system in Unity,” styczeń 2019. [Online]. Available: [https://youtu.be/XOjd\\_qU2Ido](https://youtu.be/XOjd_qU2Ido).
- [13] T. Asbjørn, „Start Menu in Unity,” styczeń 2019. [Online]. Available: [https://youtu.be/zc8ac\\_qUXQY](https://youtu.be/zc8ac_qUXQY).
- [14] N. Calice, „How to make UI in Unity - Easy tutorial,” styczeń 2019. [Online]. Available: [https://youtu.be/\\_RlsfVOqTaE](https://youtu.be/_RlsfVOqTaE).
- [15] N. Calice, „6 details to make a great player character - Unity Tutorial,” grudzień 2018. [Online]. Available: <https://youtu.be/C2wwxd8SnKY>.
- [16] N. Calice, „Creating 2D lights - Unity Tutorial,” luty 2019. [Online]. Available: [https://youtu.be/GLS5\\_V7kN-8](https://youtu.be/GLS5_V7kN-8).
- [17] N. Calice, „How to make music transitions - Easy Unity Tutorial,” styczeń 2019. [Online]. Available: [https://youtu.be/ToVL\\_f9G9Yk](https://youtu.be/ToVL_f9G9Yk).
- [18] A. Millard, „6 Lessons from Hollow Knight's immersive tutorial,” grudzień 2018. [Online]. Available: <https://youtu.be/vWiDS8SUvds>.
- [19] N. Calice, „How to draw pixel art game characters in PS - Tutorial,” grudzień 2018. [Online]. Available: <https://youtu.be/qzvYu48kw5Q>.

- [20] N. Calice, „Fire & Explosions pixel art game effects - PS Tutorial,” luty 2019. [Online]. Available: <https://youtu.be/4qqmHtYGMc0>.
- [21] T. Asbjørn, „2D Movement in Unity (Tutorial),” grudzień 2018. [Online]. Available: <https://youtu.be/dwCT-Dch0bA?list=PLPV2Kylb3jR6TFcFuzl2bB7TMNlBpKMQ>.
- [22] M. DaGraça i G. Lukosek, Learning C# 7 By Developing Games with Unity 2017, III red., Birmingham: Published by Packt Publishing Ltd., 2017.
- [23] N. Calice, „Hold jump key to jump higher - 2D platformer controller - Unity Tutorial,” styczeń 2018. [Online]. Available: <https://youtu.be/j111eKN8sJw>.
- [24] F. S. Lauren S. Ferro, Unity 2017 2D Game Development Projects, I red., Birmingham: Published by Packt Publishing Ltd., 2018.
- [25] N. Calice, „How to make 2D particle effects - Unity Tutorial,” styczeń 2019. [Online]. Available: [https://youtu.be/\\_z68\\_OoC\\_0o](https://youtu.be/_z68_OoC_0o).
- [26] N. Calice, „Screen shake with no code - Easy Unity tutorial,” styczeń 2019. [Online]. Available: <https://youtu.be/N24MhfeoUpE>.
- [27] N. Calice, „Cool dialog system - Easy Unity and C# Tutorial,” luty 2019. [Online]. Available: [https://youtu.be/f-oSXg6\\_AMQ](https://youtu.be/f-oSXg6_AMQ).
- [28] A. Pranckevicius, „FramesPerSecond,” kwiecień 2019. [Online]. Available: <https://wiki.unity3d.com/index.php/FramesPerSecond>.
- [29] Unity Technologies, „Application.targetFrameRate,” marzec 2019. [Online]. Available: <https://docs.unity3d.com/ScriptReference/Application-targetFrameRate.html>.