Politechnika Wrocławska

Wydział Informatyki i Zarządzania

Kierunek studiów: Informatyka

Praca dyplomowa – inżynierska

GRA TYPU „SURVIVAL” Z PROCEDURALNIE GENEROWANYM ŚWIATEM

Dawid Antczak

słowa kluczowe:

gra, survival, proceduralne generowanie

krótkie streszczenie:
W pracy przedstawiono projekt i implementację gry komputerowej z gatunku „survival” z proceduralnie generowanym światem. Opisano założenia rozgrywki, fragmenty implementacji, przeprowadzone testy oraz plany rozwojowe.

Opiekun pracy dyplomowej	dr inż. Marek Kopel	.....	.....
	Tytuł/stopień naukowy/imię i nazwisko	ocena	podpis
Ostateczna ocena za pracę dyplomową			
Przewodniczący Komisji egzaminu dyplomowego		.....	.....
	Tytuł/stopień naukowy/imię i nazwisko	ocena	podpis

*Do celów archiwalnych pracę dyplomową zakwalifikowano do:**

a) kategorii A (akta wieczyste)

b) kategorii BE 50 (po 50 latach podlegające ekspertyzie)

** niepotrzebne skreślić*

pieczątka wydziałowa

Wrocław

2019

STRESZCZENIE

Celem niniejszej pracy jest zaimplementowanie gry komputerowej zawierającej typowe mechaniki gatunku *survival* oraz wykorzystującej techniki proceduralnego generowania w tworzeniu unikalnego świata na potrzeby rozgrywki. Pracę rozpoczęto od wprowadzenia w tematykę problemu, a następnie dokonano przeglądu istniejących rozwiązań, sformułowano wymagania funkcjonalne i niefunkcjonalne oraz założenia rozgrywki. Przedstawione zostały najciekawsze aspekty implementacji i uzyskane efekty. Praca zawiera również opis i wyniki przeprowadzonych testów wydajności. Przygotowane w ramach projektu inżynierskiego rozwiązanie jest grywalną wersją alfa z nakreślonymi planami rozwojowymi.

ABSTRACT

The main goal of this thesis was to implement a computer game containing typical survival mechanics and using procedural generation techniques to create unique worlds. Thesis began with an introduction to the problem followed by a review of existing solutions. Afterwards, functional and non-functional requirements along with gameplay assumptions were formulated. The most interesting aspects of implementation as well as the results obtained were presented. The thesis also contains a description and outcomes of performed performance tests. The solution prepared as part of this thesis is a playable alpha version with outlined development plans.

SPIS TREŚCI

WSTĘP

Światowy rynek gier komputerowych rozwija się prężnie. Zgodnie z danymi z roku 2018, wart jest ponad pół biliona złotych i rośnie w tempie przekraczającym 9% rocznie. Tendencja ta ma się za się utrzymać przynajmniej do 2021 roku[31]. Gry wideo stały się drugą najchętniej wybieraną formą spędzania czasu.

Specjalnym rodzajem gier są gry niezależne (ang. *indie*), które powstają w małych zespołach (niekiedy jednoosobowych). Ta gałąź gier zawdzięcza swój rozwój w głównej mierze powstaniu i rozbudowie kanałów cyfrowej dystrybucji, poprzez które wydawanie gier jest znacznie tańsze[9]. Nie bez znaczenia są także możliwości pozyskania funduszy na rozwój, dzięki instytucjom zajmującym się inwestowaniem w nowe produkcje oraz finansowaniu społecznościowym (ang. *crowdfunding*).

Celem pracy jest wytworzenie gry komputerowej z gatunku *survival* z proceduralnie generowanym światem. W tym celu należy zapoznać się z cechami gatunku, zasadami proceduralnego generowania oraz przeanalizować istniejące rozwiązania rynkowe. Na tej podstawie zostanie zaprojektowany i zaimplementowany produkt, z wykorzystaniem odpowiednich do tego celu narzędzi.

Zakres pracy obejmuje przygotowanie projektu, określającego wymagania funkcjonalne i niefunkcjonalne oraz założenia rozgrywki, a także realizację w postaci grywalnego produktu w wersji alfa. Wytworzona gra ma posiadać mechaniki typowe dla gatunku *survival*, które będą stanowić istotny element zabawy. Na potrzeby każdej rozgrywki będzie generowany unikalny świat, z wykorzystaniem odpowiednich algorytmów. Zostaną przygotowane lub pozyskane konieczne grafiki i dźwięki oraz przeprowadzone niezbędne testy sprawdzające wydajność gry.

1. TEMATYKA PROJEKTU

1.1. GATUNEK SURVIVAL

Gatunek survival stał się bardzo popularny w ostatnich latach za sprawą, między innymi takich gier jak *Minecraft* oraz *DayZ*. W głównej mierze to niezależne studia odpowiadają za sukces gatunku, silnie wpływając na największych producentów oraz wydawców. Obecnie wiele mechanik znanych z wysokobudżetowych gier ma swój rodowód w grach survivalowych[16].

Typowe mechaniki dla gatunku survival to zbieranie zasobów, wytwarzanie przedmiotów i dbanie o potrzeby postaci. Rozgrywka zazwyczaj toczy się w dużym otwartym świecie, często proceduralnie generowanym, a graczowi nie zostają postawione żadne konkretne cele do zrealizowania. Fabuła rzadko stanowi istotny element gry.

1.2. PROCEDURALNE GENEROWANIE

Proceduralne generowanie (ang. *procedural content generation*) jest techniką programistyczną szeroko stosowaną w grach komputerowych, która może być definiowana na wiele sposobów[21]. Proceduralne generowanie treści jest zautomatyzowanym tworzeniem zawartości z wykorzystaniem algorytmów. Najczęściej w ramach generowania proceduralnego wykorzystuje się niewielką liczbę parametrów wejściowych oraz pseudolosowe procesy w celu wygenerowania treści, która będzie spełniała postawione wymagania. Algorytmy działające całkowicie losowo, mogą stworzyć zawartość, która nie będzie grywalna, dlatego warunkiem im postawionym powinno być sprawdzanie rezultatu pod kątem zasad gry. Wyniki wygenerowane dla różnych parametrów wejściowych powinny zauważalnie różnić się od siebie oraz nie powinny umożliwiać łatwego opisania przy pomocy ogólnego wzoru.

Proceduralne generowanie na potrzeby gier może posłużyć do stworzenia dowolnych treści. Najczęściej należą do nich mapa świata, układ poziomów gry, tekstury, siatki modeli trójwymiarowych, dźwięki, muzyka i elementy fabuły.

Proceduralne generowanie cieszy się popularnością szczególnie wśród twórców niezależnych. Dysponują oni znacznie mniejszymi zasobami finansowymi oraz ludzkimi, a technika ta pozwala zaoszczędzić czas przy tworzeniu zawartości.

2. PROJEKT

2.1. PRZEGLĄD ISTNIEJĄCYCH ROZWIĄZAŃ

2.1.1. Minecraft

Gra komputerowa stworzona przez Markusa Perssona i rozwijana przez studio *Mojang AB*. Pierwsza publiczna wersja testowa została wydana w 2009 i jeszcze przed swoją oficjalną premierą w roku 2011 stała się fenomenem za sprawą mediów społecznościowych [19]. Powszechnie chwalony był brak ograniczeń w modyfikacji terenu oraz generator świata, którego efekty można zaobserwować na rysunku 2.1. Odpowiada on za zróżnicowane biomy, wioski niezależnych postaci i podziemia. Prostota, swoboda, brak narzuconego celu zabawy sprawiły, że gracze z całego świata docenili tę produkcję. Pewne mechaniki, jak zbieranie zasobów, budowanie, wytwarzanie przedmiotów, stały się za sprawą *Minecrafta* powszechne także w innych gatunkach gier[14]. Gra po upływie lat doczekała się portów na wiele innych urządzeń[11]. Wszystkie edycje sprzedały się łącznie w postaci ponad 176 milionów kopii[29].

Rys. 2.1. Fragment świata z gry *Minecraft*, dostęp 26.10.2019
https://primewikis.com/guides/how-to-transfer-minecraft-worlds-from-pc-to-xbox-one/

2.1.2. Terraria

Gra niezależnego studia *Re-Logic* została wydana w roku 2011 na komputery osobiste, a z czasem doczekała się również wersji na inne urządzenia[17]. Terraria czerpie inspiracje z *Minecrafta*, takie jak zróżnicowany generowany świat, wytwarzanie przedmiotów oraz dowolną ingerencję w otoczenie. Tym, co wyróżnia tę grę jest postawienie na dwuwymiarową grafikę (rys. 2.2), postaci niezależne, fabułę oraz walkę. To właśnie te aspekty, jak i fakt wieloletniego rozwijania jej przez twórców, sprawiły, że wybija się ona na tle innych podobnych gier[33] i sprzedała się w nakładzie ponad 27 milionów kopii[18]. Gra z upływem lat stała się bardzo bogata w zawartość, ale i wyjątkowo skomplikowana w opinii nowych użytkowników.

Rys. 2.2. Fragment rozgrywki z gry *Terraria*
przeznaczony na urządzenia z systemem *Android*, dostęp 26.10.2019
https://play.google.com/store/apps/details?id=com.and.games505.TerrariaPaid

2.1.3. Don't Starve

Wydana w roku 2013 na komputery osobiste gra niezależnego studia *Klei Entertainment*. Jeszcze w roku wydania sprzedała się w ponad milionowym nakładzie[30]. Tak jak i poprzednie produkcje, z czasem doczekała się również portów na inne urządzenia[17]. *Don't Starve* w mniejszym stopniu stawia na eksplorację świata, budowanie i kształtowanie otoczenia, a w zamian kładzie nacisk na przetrwanie w niesprzyjającym otoczeniu. Aspektom surwiwalowym sprzyja ciekawy, podzielony na biomy, proceduralnie generowany świat. Charakterystyczną, istotną dla rozgrywki oraz najczęściej krytykowaną mechaniką jest permanentna śmierć (ang. *permadeath*)[8]. *Don't Starve* operuje również dwuwymiarową

grafiką, a świat obserwujemy z perspektywy rzutu izometrycznego (rys. 2.3). Wieloletnie wsparcie twórców gry, owocowało w postaci wielu aktualizacji i kilku oficjalnych dodatków[7].

Rys. 2.3. Fragment rozgrywki z gry *Don't Starve*, dostęp 26.10.2019
https://www.pastemagazine.com/articles/2013/05/dont-starve-review-pcmaclinux.html

2.2. WYMAGANIA FUNKCJONALNE

Po analizie konkurencyjnych rozwiązań, kluczowe elementy, które powinny znaleźć się w grze, zostały sformułowane w postaci wymagań funkcjonalnych:

— Proceduralnie generowany świat
— Możliwość niszczenia środowiska
— Możliwość budowania
— Zbieranie zasobów
— Wytwarzanie przedmiotów
— Możliwość śmierci postaci wskutek zaniedbań gracza
— Możliwość zatrzymania rozgrywki w dowolnym momencie

2.3. WYMAGANIA NIEFUNKCJONALNE

Najważniejsze reguły techniczne, które powinna spełnić gra, zostały sformułowane w postaci wymagań niefunkcjonalnych:

— Płynna rozgrywka na średniej klasy komputerze osobistym z systemem Windows

— Sterowanie przystosowane do klawiatury i myszy
— Działanie bez połączenia z Internetem

2.4. ZAŁOŻENIA ROZGRYWKI

Typowe metodyki, stosowane przy rozwoju oprogramowania, nie sprawdzają się zazwyczaj przy tworzeniu gier [2]. Zamiast definiować poszczególne przypadki użycia, zdecydowano się krótko opisać założenia rozgrywki.

Gracz przed rozpoczęciem rozgrywki dostaje możliwość skonfigurowania parametrów świata, w skład których wchodzą szerokość, wysokość, oraz ziarno generatora. Po wybraniu odpowiedniej opcji, graczowi zostaje oddany wygenerowany dwuwymiarowy świat złożony z bloków o różnym wyglądzie i charakterystyce. Gracz, obserwując rozgrywkę z widoku z boku (ang. *side-scroller*), przypominającym rysunek 2.5, przemierza świat niszcząc tworzące go bloki oraz zbierając zasoby.

Część zebranych przedmiotów może posłużyć w celu budowania dowolnych struktur, a inne można dalej przetwarzać. Gracz wytwarza przedmioty umieszczając składniki w odpowiednim panelu, w którym następnie jest wyświetlane co może uzyskać. Pozwala to na produkcję narzędzi oraz pozostałych obiektów ułatwiających dalszą eksplorację i umożliwiających przetrwanie.

Koniec gry następuje w momencie śmierci bohatera, wskutek utraty wszystkich punktów zdrowia. Punkty zdrowia można stracić poprzez upadek z wysokości, wygłodzenie, odwodnienie lub utopienie postaci.

2.5. STYL GRAFICZNY

Spójny styl graficzny jest bardzo istotną cechą dla gier. Mając na uwadze wymagania oraz założenia rozgrywki, postanowiono obrać styl graficzny znany jako *pixel art*. Grafiki *pixel art* cechują się się przede wszystkim małą liczbą pikseli, co sprawia, że bardzo szybkie jest ich tworzenie, a w Internecie można znaleźć wiele darmowych obrazów do wykorzystania. Fakt ten jest istotny przy ograniczonych środkach.

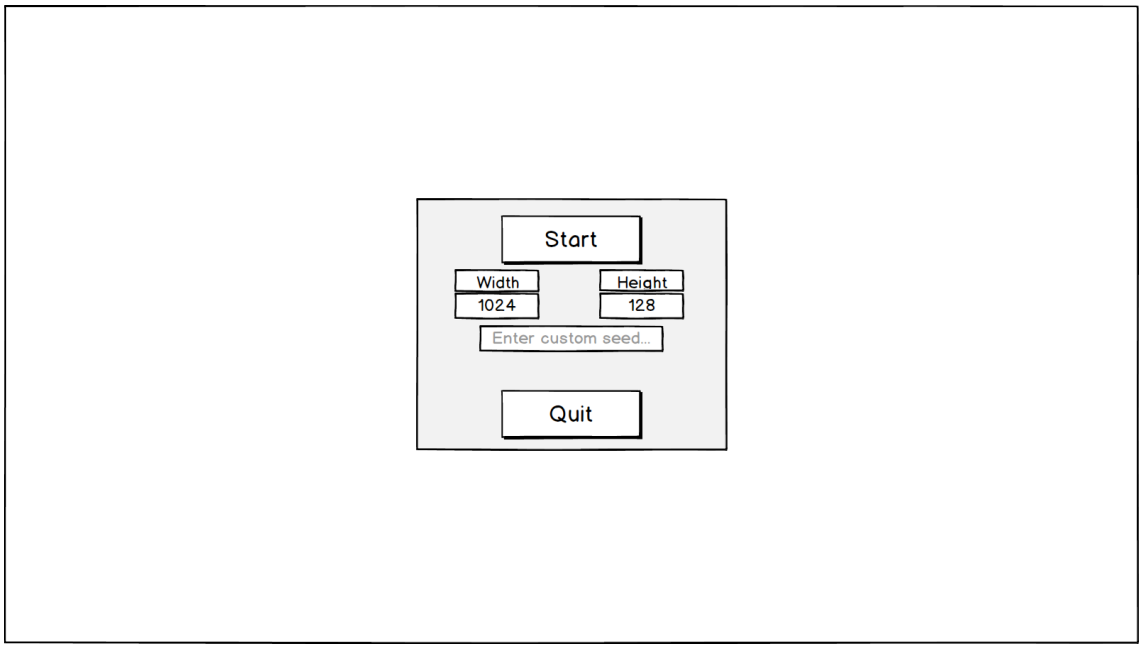
Pixel art sprawdza się w przypadku dwuwymiarowych światów złożonych z bloków, co udowodnił wcześniej zaprezentowany przykład *Terrarii*.

2.6. INTERFEJS UŻYTKOWNIKA

Na podstawie sformułowanych wymagań funkcjonalnych i niefunkcjonalnych oraz założeń rozgrywki, zostały sporządzone projekty interfejsów dla czterech widoków: menu główne, ekran gry, ekran pauzy, ekran porażki.

2.6.1. Menu główne

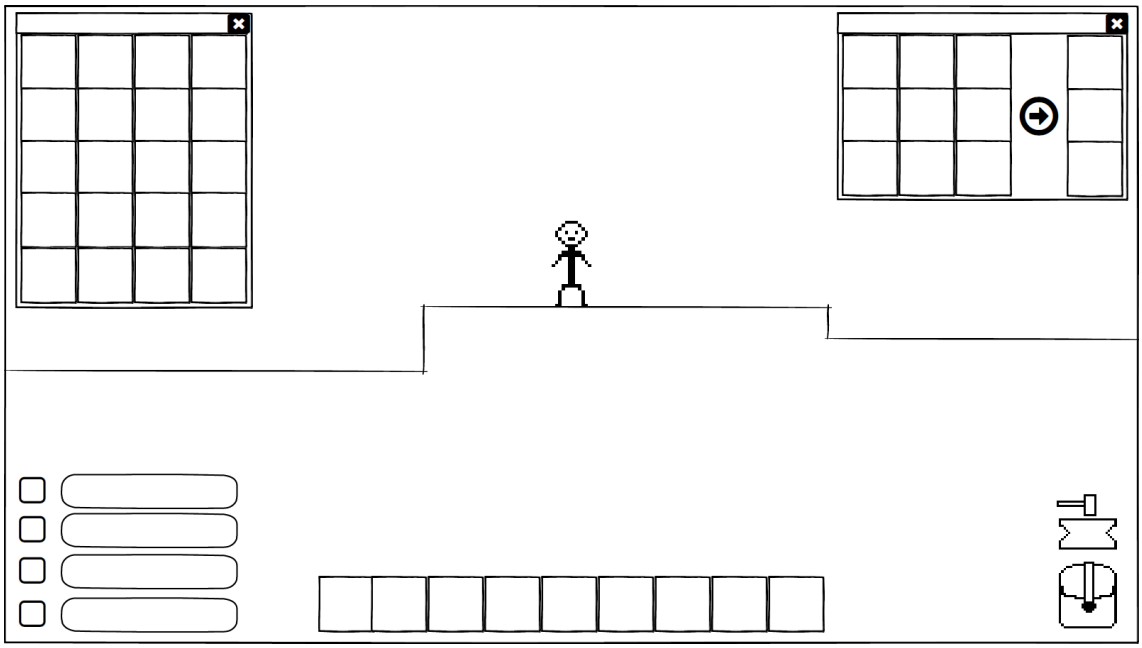
Pierwszy z ekranów po uruchomieniu, zaprezentowany na rysunku 2.4, powinien pozwalać na skonfigurowanie oraz rozpoczęcie rozgrywki jak i opuszczenie gry. Znajdują się na nim przyciski *Start*, który pozwala rozpocząć grę oraz *Quit* umożliwiający wyłączyć program, jak i pola tekstowe pozwalające wprowadzić parametry generowanego świata, czyli szerokość, wysokość oraz opcjonalne ziarno generatora. Pozostała część ekranu wypełniona jest statyczną grafiką tła.



Rys. 2.4. Ekran głównego menu. (źródło własne)

2.6.2. Ekran gry

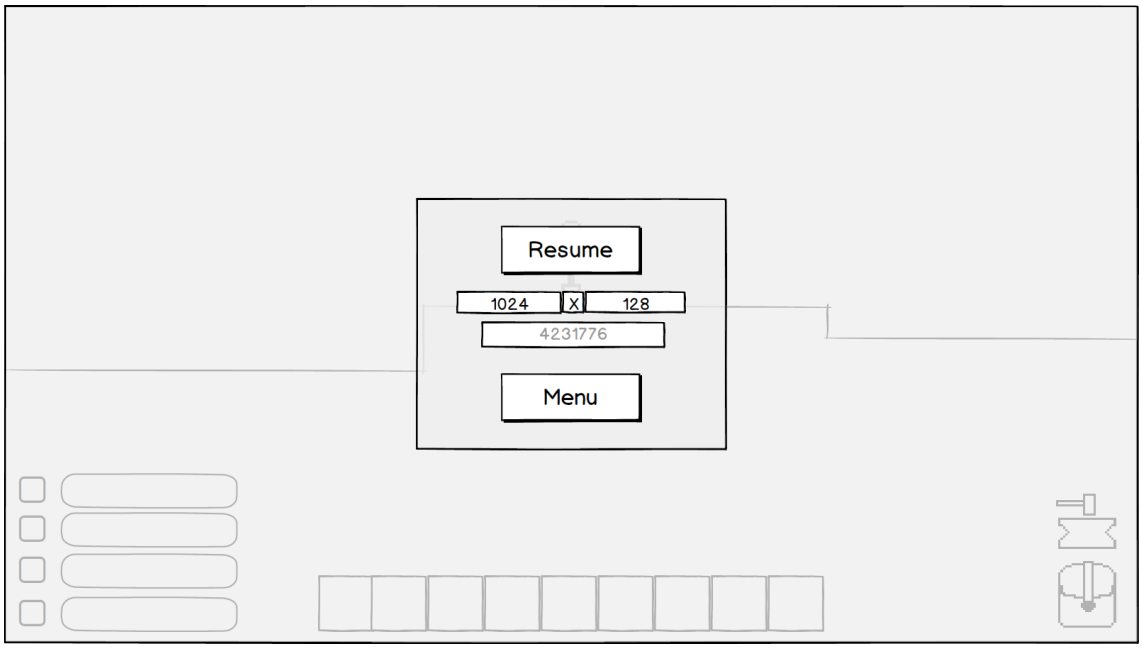
Właściwa rozgrywka odbywa się na ekranie gry (rys. 2.5). Gracz powinien widzieć świat wraz z bohaterem w centralnym punkcie, mieć dostęp do zgromadzonego ekwipunku oraz opcji wytwarzania przedmiotów, jak również powinien mieć wgląd do atrybutów postaci. W lewym dolnym rogu ekranu znajdują się opatrzone ikonami wskaźniki w formie pasków, przedstawiające wartości atrybutów bohatera. Z nich gracz dowie się o stanie jego zdrowia. W dolnej centralnej części umieszczone zostały pola szybkiego dostępu do części przedmiotów. Po prawej stronie odnaleźć można przyciski pozwalające otworzyć okna ekwipunku oraz wytwarzania przedmiotów. Oba okna można w dowolny sposób przemieścić, przeciągając ich górną belkę, oraz zamknąć charakterystycznym przyciskiem w kształcie litery *X*.



Rys. 2.5. Ekran gry wraz z otwartymi oknami ekwipunku oraz wytwarzania przedmiotów. (źródło własne)

2.6.3. Ekran pauzy

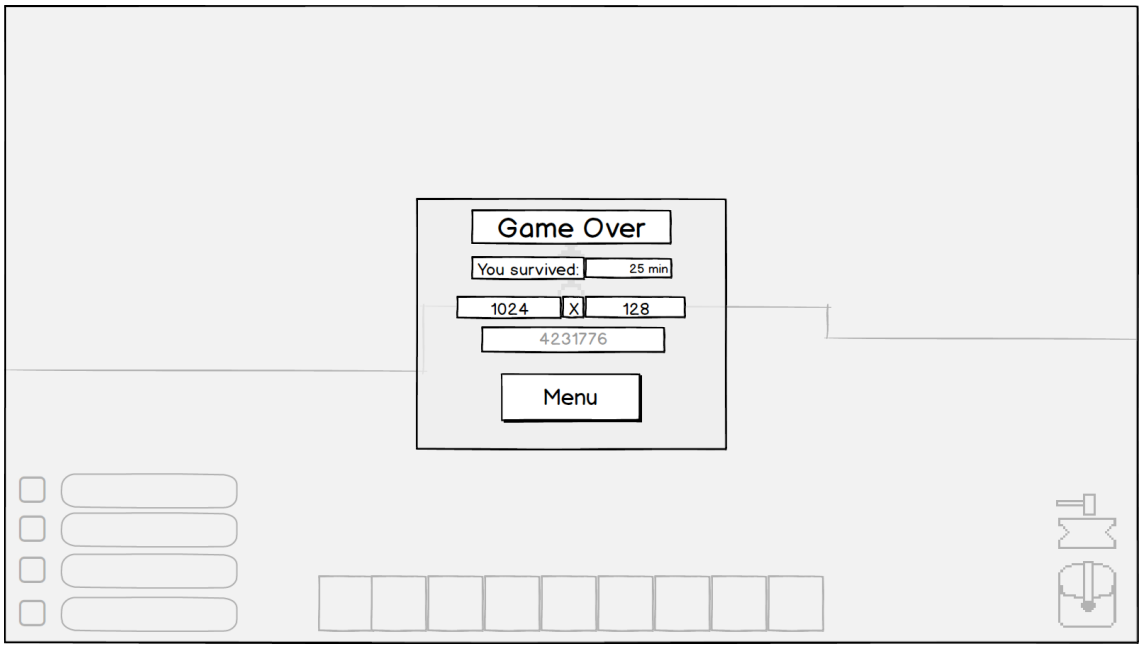
Gracz powinien mieć możliwość zatrzymania gry, by później ją wznowić lub opuścić (rys. 2.6). Po naciśnięciu przycisku *Escape* na klawiaturze, gra zostaje wstrzymana oraz pojawia się panel z przyciskami pozwalającymi wznowić grę (*Resume*) lub powrócić do menu głównego (*Menu*). Dodatkowo wyświetlone zostają tam informacje o parametrach świata, by móc je wykorzystać w celu wygenerowania tożsamego terenu w przyszłości.



Rys. 2.6. Ekran pauzy. (źródło własne)

2.6.4. Ekran porażki

Po spełnieniu warunków porażki, gracz powinien zostać poinformowany o końcu gry, przy pomocy panelu widocznego na rysunku 2.7. Przedstawia on czas trwania rozgrywki oraz posiada przycisk pozwalający powrócić do menu głównego (*Menu*). Dodatkowo, tak jak w przypadku ekranu pauzy, wyświetlone zostają tam informacje o parametrach świata, by móc je wykorzystać w celu wygenerowania identycznego terenu w kolejnej rozgrywce.



Rys. 2.7. Ekran porażki. (źródło własne)

3. IMPLEMENTACJA

W tym rozdziale opisano aspekty implementacyjne projektu, wyjaśniając użyte technologie oraz sposób realizacji opisanych w poprzednich rozdziałach założeń.

3.1. WYBRANE ŚRODOWISKO I NARZĘDZIA

Do implementacji rdzenia rozgrywki wykorzystano zintegrowane środowisko do tworzenia trójwymiarowych oraz dwuwymiarowych gier *Unity*[1] w wersji *2019.1.8f1 personal*. Unity jest najpopularniejszym środowiskiem do tworzenia gier wśród twórców niezależnych[15]. Jego głównymi zaletami jest darmowość (przy przychodzie do 100.000 USD rocznie), możliwość tworzenia gier wieloplatformowych, przyjazny interfejs oraz rozbudowana dokumentacja i liczne poradniki dostarczane przez twórców[28]. Wokół *Unity* została zbudowana liczna społeczność i obecnie możliwości silnika można dowolnie rozszerzać, dzięki darmowym jak i płatnym wtyczkom oraz innym dodatkom, udostępnionym w ramach wbudowanego sklepu *Asset Store*[2]. W tymże sklepie znaleźć można również gotowe skrypty realizujące pewne mechaniki rozgrywki, grafiki, modele trójwymiarowe, dźwięki oraz wiele innych treści pomocnych przy tworzeniu gier w pojedynkę lub w małym zespole.

Unity udostępnia ogromną liczbę gotowych rozwiązań, między innymi w postaci komponentów graficznych, fizycznych i kontrolerów animacji, które realizują główne wymagania większości gier. Pozwala to twórcom zredukować ilość pisanego kodu. Własne komponenty na potrzeby gier w silniku *Unity* są pisane w języku programowania *C#*. Jako edytor, pozwalający na pracę z kodem w języku *C#*, został wykorzystany *Rider* w wersji 2019.1.2 rozwijany przez firmę *JetBrains*. Alternatywnym rozwiązaniem jest *Visual Studio* należący do przedsiębiorstwa *Microsoft*. Oba narzędzia posiadają bardzo dobrą integrację z *Unity* oraz są oficjalnie wspierane. *Rider* oferuje jednak znacznie więcej udogodnień[5], a poza tym okazuje się działać szybciej i w mniejszym stopniu obciążać zasoby komputera.

Dodatkowo wykorzystane zasoby (ang. *assets*) na potrzeby gry:

— *Terrain Engine 2D*[3]

Kompleksowe narzędzie pozwalające na generowanie i zarządzanie dwuwymiarowym terenem złożonym z bloków. Umożliwia definiowanie własnych algorytmów gene-

[1] `https://unity.com/` (dost. 20 listopada 2019)

[2] `https://assetstore.unity.com/` (dost. 20 listopada 2019)

[3] `https://assetstore.unity.com/packages/tools/terrain/terrain-engine-2d-115381` (dost. 20 listopada 2019)

rujących oraz konfigurowanie bloków składowych. Dostarcza API pozwalające na ingerowanie w świat, zajmując się jednocześnie niskopoziomowymi aspektami optymalizacyjnymi oraz symulacjami oświetlenia i fizyki (w tym fizyki cieczy).

— *Serialized Dictionary Lite* [4]
Rozszerzenie dostarcza klas ułatwiających serializację słowników oraz ich edycję z poziomu edytora *Unity*.

— *THE BOY - FREE SPRITE*[5]
Zbiór grafik na potrzeby animowanej poklatkowo postaci gracza.

— *Shikashi's Fantasy Icons Pack*[6]
Zbiór ikon przedmiotów.

— *Farland Skies - Cloudy Crown* [7]
Zbiór grafik tła.

— Wiele dźwięków z witryny *Freesound*[8] dostępnych na licencji *CC0. 1.0*[9]

Część grafik została przygotowana w darmowym programie *Gimp*[10], a niektóre z wykorzystanych dźwięków zostały poddane obróbce w darmowym programie *Audacity*[11].

3.2. BUDOWA ŚWIATA

Dwuwymiarowy teren jest generowany w ramach silnika gry *Unity* za pomocą narzędzia *Terrain Engine 2D*, który dzieli świat na kwadraty o równej wielkości w ramach osi *XY* oraz na warstwy w ramach osi *Z*. Teren następnie jest budowany z różnych bloków. Na każdym kwadracie w ramach każdej ze zdefiniowanych warstw może znajdować się jeden blok lub nie być tam żadnego. Dodatkowo w ramach jednej z warstw może występować blok specjalny, jakim jest woda. Typowy fragment świata prezentuje się zazwyczaj jak na grafice 3.1.

[4] `https://assetstore.unity.com/packages/tools/utilities/serialized-dictionary-lite-110992` (dost. 20 listopada 2019)

[5] `https://www.gameart2d.com/the-boy---free-sprites.html` (dost. 20 listopada 2019)

[6] `https://shikashiassets.itch.io/shikashis-fantasy-icons-pack` (dost. 20 listopada 2019)

[7] `https://assetstore.unity.com/packages/2d/textures-materials/sky/farland-skies-cloudy-crown-60004` (dost. 20 listopada 2019)

[8] `https://freesound.org/` (dost. 7 grudnia 2019)

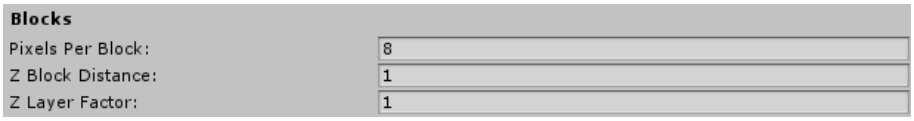
[9] `https://creativecommons.org/publicdomain/zero/1.0/deed.pl` (dost. 20 listopada 2019)

[10] `https://www.gimp.org/` (dost. 20 listopada 2019)

[11] `https://www.audacityteam.org/` (dost. 20 listopada 2019)

Rys. 3.1. Budowa świata. (źródło własne)

Większość istotnych ustawień, dotyczących budowy świata, znajduje się w komponencie *World*. Z poziomu edytora, w zakładce *Block Setup* znajduje się sekcja *Blocks* opisująca między innymi liczbę pikseli przypadających na jeden blok (rysunek 3.2). Własność ta skonfigurowana została na wartość *8*, gdyż na potrzeby gry zostały przygotowane grafiki bloków o rozmiarze *8x8* pikseli.

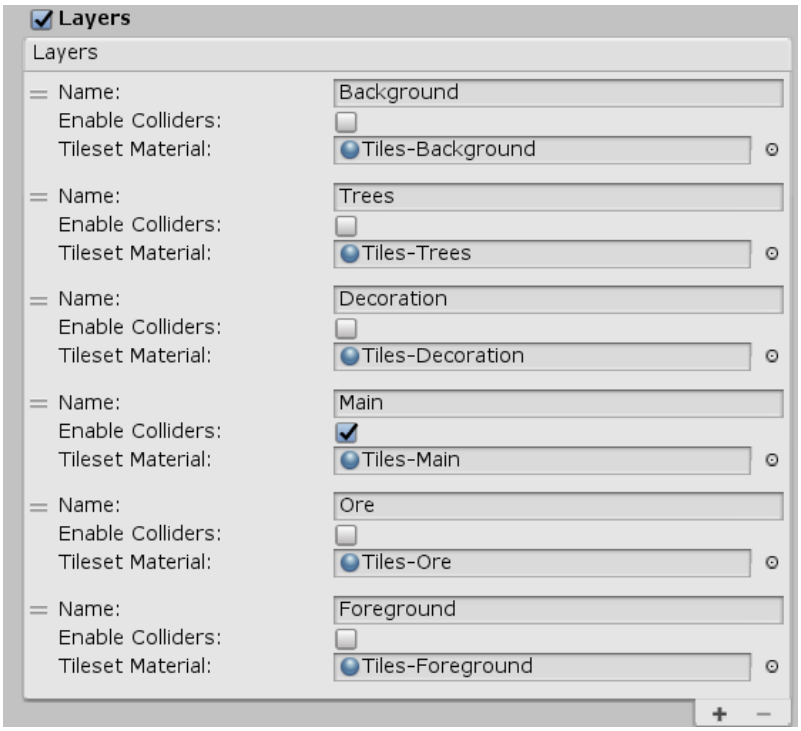


Rys. 3.2. Ustawienia sekcji *Blocks* komponentu *World*. (źródło własne)

Dalej zdefiniowane zostały warstwy oraz przypisane do nich materiały z przygotowaną teksturą, jak na rysunku 3.3. Na potrzeby gry zostało zdefiniowane sześć warstw:

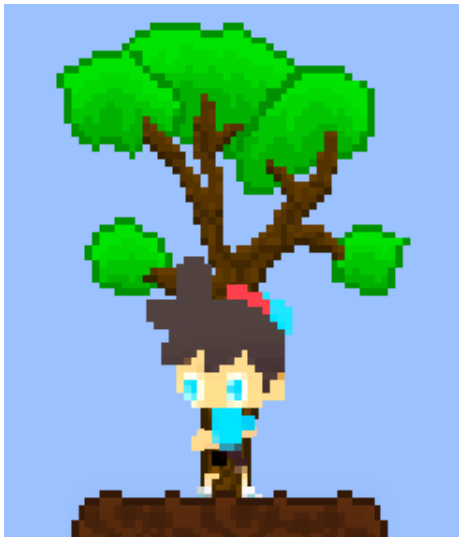
1. *Background* (tło)
2. *Trees* (drzewa)
3. *Decoration* (dekoracje)
4. *Main* (warstwa główna)
5. *Ore* (ruda)
6. *Foreground* (pierwszy plan)

Kolejność warstw jest istotna, dlatego że w takiej kolejności, w jakiej są zdefiniowane, są następnie rysowane na ekranie. Powinny być zapisane od najdalszej (z perspektywy kamery) do najbliższej. Dodatkowo istotne jest ustalenie, z którymi warstwami postaci wchodzą w kolizję, a które mogą być przez nie przenikane. Na potrzeby gry zostało postanowione, że w kontakt można wejść tylko z warstwą główną (*Main*).



Rys. 3.3. Ustawienia sekcji *Layers* komponentu *World*. (źródło własne)

Zgodnie z wcześniej zdefiniowaną hierarchią warstw, bloki poprzedzające warstwę główną gracz będzie mijał przesłaniając je (rysunek 3.4), natomiast bloki na warstwach następujących po warstwie głównej będzie mijał, dając się przesłonić (rysunek 3.5).



Rys. 3.4. Zasłanianie bloków tła i drzew. (źródło własne)

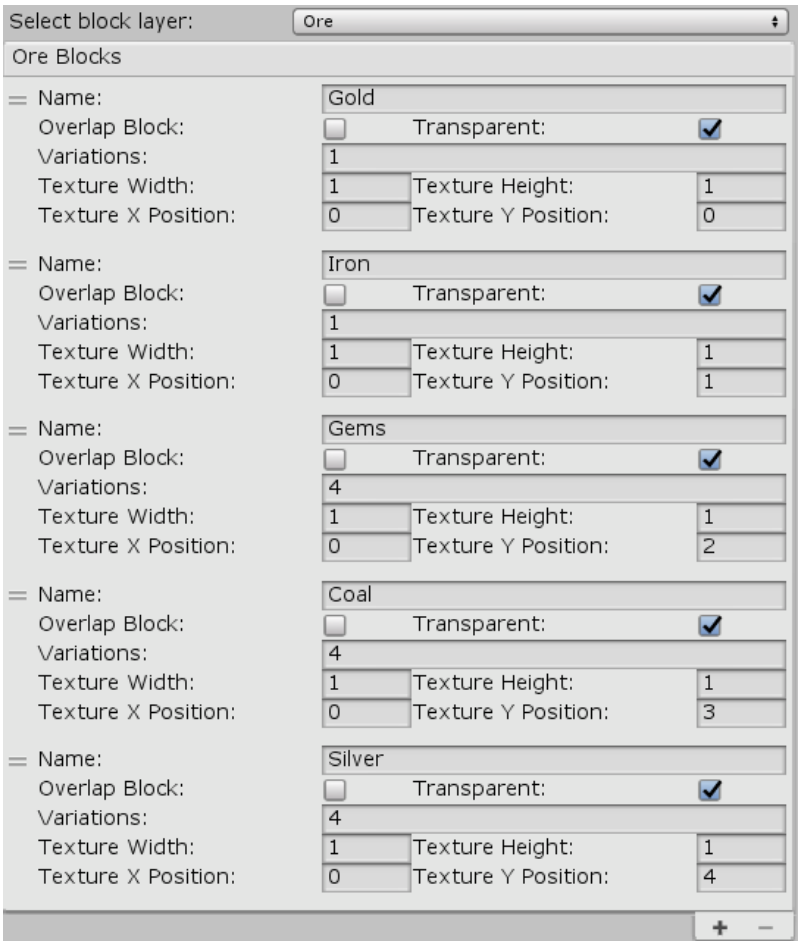
Rys. 3.5. Częściowe zasłonięcie postaci gracza przez kwiaty. (źródło własne)

Każda warstwa posiada pewną pulę bloków, jakie mogą się na niej znajdować. Dla każdej z warstw zdefiniowano jednak tylko jeden plik, który zawiera przygotowane grafiki

dla wszystkich bloków, które mogą na niej wystąpić (przykład na rysunku 3.6). Łączenie wielu tekstur w ramach jednego pliku jest techniką optymalizacyjną zwaną atlasem tekstur (ang. *texture atlas*)[4].

W kolejnej sekcji komponentu *World* skonfigurowane zostały szczegółowe dane dotyczące korzystania z wcześniej zdefiniowanych tekstur dla konkretnej warstwy (rysunek 3.7). Określone zostaje tam w jaki sposób dzielić pojedynczą grafikę na fragmenty. Zmienna *Variations* określa ile wersji graficznych danego bloku jest przygotowane. W zaprezentowanym przykładzie wynosi ona *4*, gdyż dla każdego bloku przygotowano 4 warianty, które przylegają do siebie w rzędzie. W trakcie generowania świata wybierana jest losowa z nich. Poza urozmaiceniem graficznym, rózne wersje tego samego bloku nie wpływają w żaden sposób na rogrywkę.

Rys. 3.6. Przygotowane tekstury dla bloków rudy. (źródło własne)



Rys. 3.7. Szczegółowe ustawienia warstwy rudy (*Ore*) w ramach komponentu *World*. (źródło własne)

3.3. GENEROWANIE TERENU

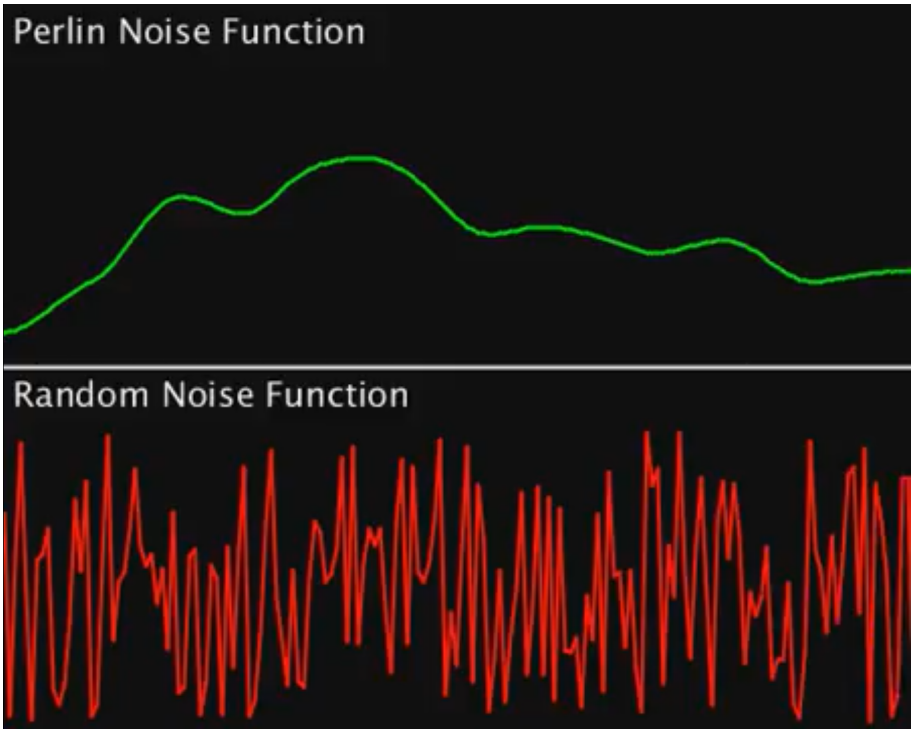
3.3.1. Wykorzystanie proceduralnego generowania

Głównym celem przyświecającym wykorzystaniu proceduralnego generowania w projekcie było urozmaicenie gry, dzięki zapewnieniu, że każda rozgrywka będzie inna. Ograniczenie się do zaprojektowania jednego lub kilku poziomów sprawiłoby, że kolejne rozgrywki nudziłyby gracza. Zróżnicowanie generowanych światów i związany z tym element nieprzewidywalności sprawia, że gra zachowuje dłużej grywalność (ang. *replayability*).

W celu wygenerowania świata gry zostanie wykorzystany powszechnie stosowany szum Perlina, jak i własne algorytmy, pozwalające przystosować świat na potrzeby projektowanej rozgrywki. Wykorzystywanym przez część algorytmów generatororem liczb pseudolosowych jest *System.Random*, który jest w pełni deterministyczny, dla konkretnego ziarna, użytego przy jego utworzenie i określającego stan początkowy[20]. Pozwoli to na wygenerowanie identycznego terenu, jeśli gracze zechcą skorzystać wielokrotnie z tego samego ziarna.

3.3.2. Szum Perlina

Szum Perlina jest algorytm generującym szum gradientowy opracowanym przez Kena Perlina. Wygląda on bardziej naturalnie niż całkowicie losowo generowane wartości, gdyż wytwarza naturalnie sortowane sekwencje liczb pseudo-losowych[6]. Na rysunku 3.8 przedstawiono porównanie szumu Perlina oraz losowo wygenerowanych wartości.



Rys. 3.8. Wykresy porównujące jednowymiarowy szum Perlina oraz losowo wygenerowane wartości, dostęp 29.10.2019
https://www.youtube.com/watch?v=mv0JlJwk72w

Szum Perlina w grach najczęściej używany jest do generowania tekstur oraz map wysokości. Twórca gry *Minecraft*, Markus Persson, na swoim blogu opisuje proces generowania mapy z wykorzystaniem szumu Perlina oraz problemy z tym związane[13]. Typowe zastosowanie w postaci generowania mapy wysokości nie pozwala na uformowanie jaskiń. Markus Persson wykorzystał szumu Perlina zarówno w osi horyzontalnej jak i wertykalnej co pozwoliło mu na osiągnięcie niepowtarzalnych wyników.

W tym projekcie wykorzystano domyślą implementację szumu Perlina dla silnika *Unity*, znajdującą się w klasie *UnityEngine.Mathf* [25]. Wykorzystany jest on do generowania poziomu wysokości podłoża oraz wody. Dodatkowo służy on tworzeniu złóż surowców podziemnych oraz drążeniu jaskiń.

3.3.3. Algorytm generowania świata

Świat przy każdej nowej rozgrywce jest w całości generowany przy starcie. Tym samym jest on ograniczonej wielkości, którą można skonfigurować, zgodnie z potrzebami oraz możliwościami sprzętowymi. Gracz obok parametrów wielkości świata, może podać ziarno dla generatora. Dla konkretnej wielkości świata, tożsame ziarno gwarantuje wygenerowanie identycznego świata przy każdym uruchomieniu rozgrywki, dzięki właściwościom wcześniej opisanego generatora *System.Random*.

Proces generowania można rozłożyć na dwa główne etapy. W pierwszej iteracji, dla każdej jednostki szerokości świata, korzystając z szumu Perlina wygenerowana zostaje wysokość terenu, a następnie uzupełniane są bloki podziemne i generowane jaskinie. Na koniec umieszczane są kwiaty i drzewa na powierzchni. W drugiej iteracji, dla każdej jednostki szerokości świata, w przestrzeniach, gdzie nie ma bloków na warstwie głównej, generowane są baseny wodne oraz pnącza.

Narzędzie *Terrain Engine 2D* wymaga, by algorytm generujący świat został zaimplementowany w ramach metody *GenerateData* w klasie dziedziczącej po *TerrainEngine2D.TerrainData*, tak jak ma to miejsce na listingu 3.1.

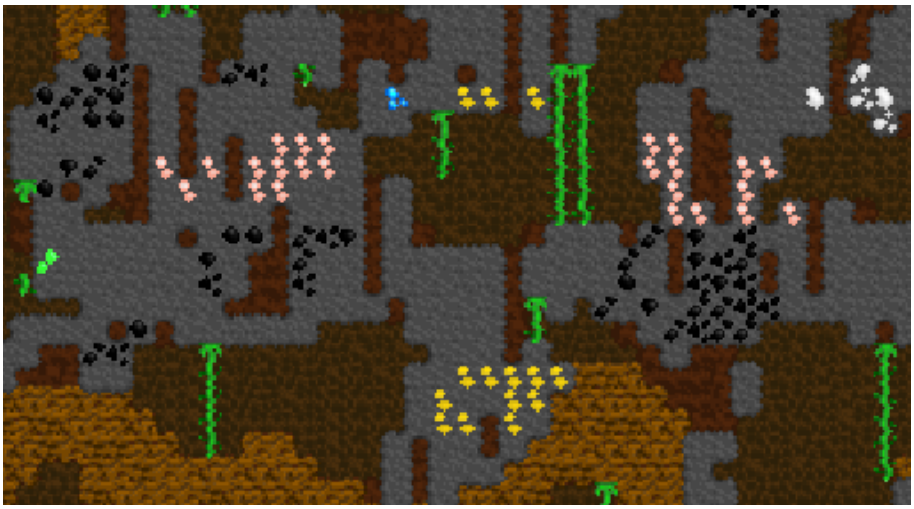
```
1  public override void GenerateData()
2  {
3     base.GenerateData();
4     for (int x = 0; x < world.WorldWidth; x++)
5     {
6        int groundLevel = CalculateGroundLevel(x);
7        PlaceUnderGroundLevel(x, groundLevel);
8        PlaceForegroundElementsAndTrees(x, groundLevel);
9     }
10    for (int x = 0; x < world.WorldWidth; x++)
11    {
12       int waterHeight = CalculateWaterHeight(x);
13       for (int y = 0; y < world.WorldHeight; y++)
14       {
15          PlaceWaterPools(x, y, waterHeight);
16          PlaceVines(x, y);
17       }
18    }
19 }
```

Listing 3.1: Algorytm generowania świata (opr. wł.)

Metody z listingu 3.1 *CalculateGroundLevel* (linia 6) oraz *CalculateWaterHeight* (linia 12) wyliczają wartość wysokości poziomów podłoża oraz wody, korzystając z szumu Perlina. W celu urozmaicenia wyników, w ramach owych metod, nałożone zostały rezultaty szumu

dla kilku wartości parametrów. Podejście takie określane jest syntezą spektralną (ang. *spectral synthesis*)[12].

W generowniu fragmentów podziemnych wykorzystany jest fakt, że wartość dwuwymiarowego szumu Perlina dla dwóch punktów znajdujących się blisko siebie na płaszczyźnie jest zbliżona. Dzięki temu rozkład surowców nie jest całkowicie losowy, a tworzone są naturalnie wyglądające skondensowane obszary ze złożami. Przykład tego zjawiska można zaobserwować na rysunku 3.9. Używana implementacja funkcji generującej szum Perlina zwraca wartości z zakresu [0,1], więc wymagane jest jedynie określenie jaki zakres wyników ma skutkować wystąpienieniem konkretnego złoża. Dla różnych surowców wykorzystane są niezależne wartości szumów, gdzie parametry wejściowe zostają jeszcze przemnożone przez pewne czynniki. Sprawia to, że niektóre z surowców występują częściej, ale ich złoża są małe, natomiast inne są rzadziej spotykane, ale w większych ilościach. Ta sama metoda została zastosowana przy generowaniu jaskiń oraz określaniu wystąpień wielu innych bloków.



Rys. 3.9. Skupiska surowców podziemnych oraz jaskinie. (źródło własne)

W celu generowania pnączy, drzew, traw i innych elementów pierwszego planu posłużono się prostymi algorytmami sprawdzającymi charakterystyczne miejsca, gdzie mogą one wystąpić i dodającymi je z pewnym prawdopodobieństwem. Dla pnączy punktem startu jest pierwszy pusty blok pod blokiem warstwy głównej, a dla pozostałych elementów roślinnych jest to pierwsze puste miejsce nad blokiem warstwy głównej. W obu przypadkach sprawdzane jest również czy znajduje się wystarczająco miejsca na całą roślinę. Minimalne i maksymalne wysokości pnączy i drzew zostały określone w kodzie.

Baseny wodne są generowane z wykorzystaniem prostego algorytmu dostarczonego wraz z narzędziem *Terrain Engine 2D*. Gdy miejsce na warstwie głównej jest puste i znajduje się poniżej wcześniej wyznaczonego poziomu, dodany jest z pewnym niewielkim

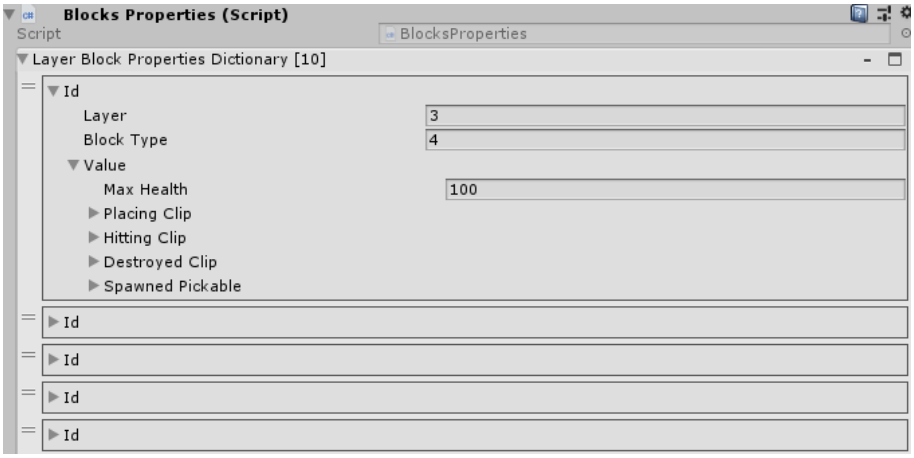
prawdopodobieństwem blok wody, a następnie rekurencyjnie uzupełniane są wszystkie sąsiadujące puste bloki znajdujące się pod nim oraz na prawo i na lewo od niego.

3.4. PARAMETRY BLOKÓW

Każdy blok posiada oprócz tekstury, z którą jest rysowany, szereg własności wykorzystywanych na potrzeby mechanik gry. Należą do nich:

— Maksymalna wytrzymałość bloku - wykorzystywana przy niszczeniu (wydobywaniu) bloku odpowiednimi narzędziami.
— Dźwięk umieszczania - dźwięk odtwarzany przy budowaniu z wykorzystaniem bloku.
— Dźwięk uderzania - dźwięk odtwarzany podczas uderzania w blok.
— Dźwięk zniszczenia - dźwięk odtwarzany po zniszczeniu (wydobyciu) bloku.
— Tworzony przedmiot - przedmiot pojawiający się po zniszczeniu (wydobyciu) bloku, możliwy do podniesienia przez gracza.

Owe własności są przechowywane w ramach singletonowej klasy *BlocksProperties* i są możliwe do konfiguracji z poziomu edytora silnika *Unity*.



Rys. 3.10. Własności bloków konfigurowalne z poziomu edytora *Unity* (źródło własne)

Unity nie wspiera serializacji struktury danych *Dictionary*, która najlepiej sprawdza się w celu mapowania bloków na dodatkowe atrybuty. Aby więc osiągnąć efekt z rysunku 3.10 i móc ustawiać atrybuty bloków, posłużono się rozwiązaniem opisanym na oficjalnym forum *Unity*[12]. Napisano niezbędne klasy i struktury, które pozwalają przechowywać cechy bloków oraz ustawiać je z poziomu edytora silnika (listing 3.2).

[12] `https://forum.unity.com/threads/finally-a-serializable-dictionary-for\-unity-extracted-from-system-collections-generic.335797/` (dost. 25 listopada 2019)

```
[Serializable]
public struct LayerBlockTypePair
{
    [SerializeField] private byte layer;
    [SerializeField] private byte blockType;
    ...
}
[Serializable]
public class BlockProperties
{
    [SerializeField] private float maxHealth = 100f;
    [SerializeField] private CustomAudioClip placingClip;
    [SerializeField] private CustomAudioClip hittingClip;
    [SerializeField] private CustomAudioClip destroyedClip;
    [SerializeField] private StorableItem spawnedPickable;
    ...
}
[Serializable]
public class LayerBlockTypePropertiesDictionary :
↪  SerializableDictionaryBase<LayerBlockTypePair, BlockProperties> { }
```

Listing 3.2: Deklaracje wymaganych klas oraz struktur na potrzeby serializacji własności bloków. (opr. wł.)

3.5. PRZEDMIOTY

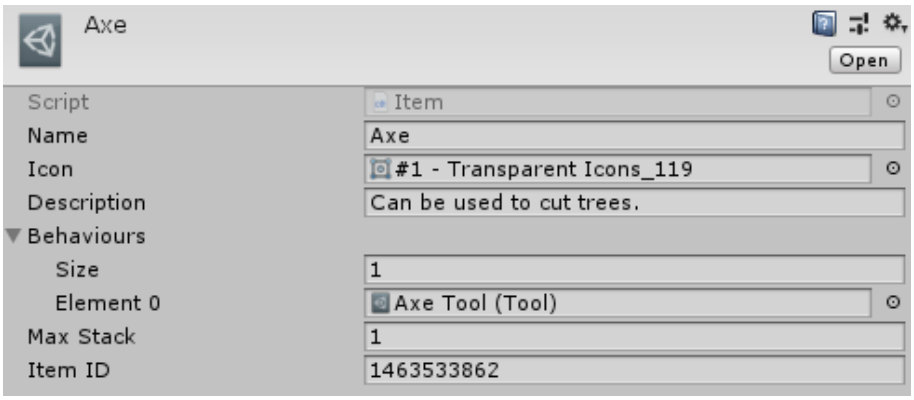
Mówiąc o przedmiotach, należy rozróżnić definicje przedmiotów (klasa *Item* z listingu 3.3) od instancji przedmiotów (klasa *StorableItem* z listingu 3.6).

3.5.1. Definicje przedmiotów

Jako definicje przedmiotów rozumiane są obiekty skonfigurowane, z poziomu edytora *Unity* (rys. 3.11) i nie zmieniające swoich atrybutów w trakcie gry. Służą one jako schemat dla poszczególnych instancji przedmiotów. Dziedziczą one po klasie *ScriptableObject*, co pozwala tworzyć je niczym zwyczajne zasoby gry (assety)[27]. Zawierają one nazwę, ikonę, opis, listę zachowań, informację o maksymalnej liczbie sztuk w stosie oraz unikalny identyfikator.

```
public class Item : ScriptableObject
{
    [SerializeField] private new string name;
    [SerializeField] private Sprite icon;
    [SerializeField] private string description;
    [SerializeField] private List<ItemBehaviour> behaviours;
    [SerializeField] private int maxStack = 1;
    [SerializeField] private int _itemID =
    ↪ Guid.NewGuid().ToString().GetHashCode();
    ...
}
```

Listing 3.3: Cześć definicji klasy *Item*. (opr. wł.)



Rys. 3.11. Własności przedmiotu *Axe* konfigurowalne z poziomu edytora *Unity*. (źródło własne)

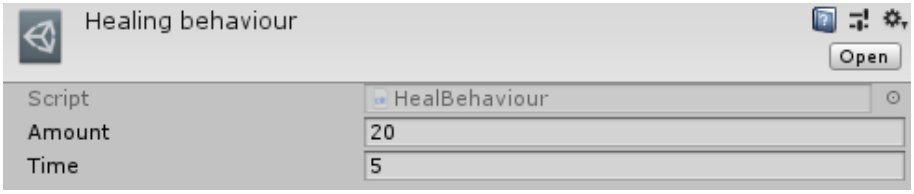
Ciekawym elementem definicji przedmiotu jest lista zachowań, w której skład wchodzą obiekty, zawierające logikę akcji uruchamianych przy użyciu przedmiotu.

ItemBehaviour jest abstrakcyjną klasą, która również dziedziczy po *ScriptableObject*. W przypadku, w którym chcemy, by użycie przez gracza konkretnego przedmiotu wiązało się z jakąś akcją, zdefiniować należy klasę, która dziedziczy po *ItemBehaviour* oraz implementuje interfejs *IUsable*. Przykładem jest zachowanie leczące *HealBehaviour* z listingu 3.4. W ramach metody *Use(SelectableIcon selectableIcon)* interfejsu *IUsable* została zaimplementowana logika związana z uleczeniem gracza. W edytorze możemy skonfigurować dodatkowe parametry leczenia (rys. 3.12) i dodać obiekt do listy zachowań przedmiotów, które chcemy, by leczyły w dany sposób.

```
public class HealBehaviour : ItemBehaviour, IUsable
{
    [SerializeField] private int amount;
    [SerializeField] private float time;

    public void Use(SelectableIcon selectableIcon)
    {
        Player.Instance.StartCoroutine(Heal());
    }
    ...
}
```

Listing 3.4: Cześć definicji klasy *HealBehaviour*. (opr. wł.)



Rys. 3.12. Parametry leczenia zachowania *HealingBehaviour*. (źródło własne)

Klasa *Item* udostępnia metodę, która umożliwia wykonanie akcji na zachowaniach konkretnego typu. Jej implementacja oraz przykładowe użycia widoczne są na listingu 3.5. W ten sposób, wszystkie zachowania typu *IUsable* są uruchamiane w momencie kliknięcia prawym przyciskiem myszy na przedmiot, natomiast zachowania *IAlternateUsable*, gdy kliknięciu towarzyszy przytrzymanie lewego klawisza *Shift*. W przyszłości można określić dowolne inne interakcje z przedmiotem i dodać kolejne interfejsy i zachowania je implementujące.

Podejście budowania przedmiotów poprzez kompozycję komponentów jest znacznie lepsze niż klasyczne hierarchie dziedziczenia, gdyż pozwala dowolnie dodawać i mieszać zachowania. W podejściu z dziedziczeniem, każdy przedmiot implementowałby odpowiednią metodę wywoływaną wraz z użyciem przedmiotu. Gdybyśmy chcieli utworzyć przedmiot leczący, należałoby utworzyć klasę dziedziczącą po *Item* z odpowiednią logiką. Podobnie należałoby postąpić, gdybyśmy potrzebowali przedmiotu zmniejszającego głód lub pragnienie bohatera. Problem pojawiłby się w przypadku, gdy jeden przedmiot miałby zarówno leczyć, zmniejszać głód, jak i zmniejszać pragnienie. Wymagałoby to utworzenia kolejnej klasy i powodowałoby redundancję kodu, a sama hierarchia dziedziczenia byłaby nieoczywista.

Zastosowana tutaj zasada kompozycji zamiast dziedziczenia (ang. *composition over inheritance*) została szeroko opisana w literaturze[3]. Reguła ta okazuje się szczególnie

```
public void ExecuteOnBehaviours<T>(Action<T> action) where T : class
{
   foreach (var beh in behaviours)
   {
      if (beh is T behAsT)
      {
         action(behAsT);
      }
   }
}

public void OnUse(SelectableIcon selectableIcon)
{
   ExecuteOnBehaviours((IUsable x) => x.Use(selectableIcon));
}

public void OnAlternateUse(SelectableIcon selectableIcon)
{
   ExecuteOnBehaviours((IAlternateUsable x) =>
   ↪  x.AlternateUse(selectableIcon));
}
```

Listing 3.5: Implementacja metody wykonującej akcję na zachowaniach oraz przykładowe użycia. (opr. wł.)

sprawdzać w przypadku tworzenia gier komputerowych, a sam silnik *Unity* wspiera to podejście[1].

3.5.2. Instancje przedmiotów

Instancje przedmiotów są konkretnymi egzemplarzami, które można znaleźć w świecie gry, magazynować w ekwipunku oraz przetwarzać. Obiekty te, oprócz referencji do swojej definicji (dane niezmieniające się), przechowują informacje o liczbie sztuk oraz pozostałej wytrzymałości przedmiotu.

```
1 public class StorableItem
2 {
3     public Item Item;
4     [SerializeField] private int count;
5     [SerializeField] private int durability;
6     ...
7 }
```

Listing 3.6: Cześć definicji klasy *StorableItem*. (opr. wł.)

3.6. EKWIPUNEK

Gracz posiada nieskończony plecak (rys. 3.14) dostępny pod przyciskiem widocznym na rysunku 3.13 oraz pasek z przedmiotami tak zwanego *szybkiego dostępu* na dole ekranu (rysunek 3.15). W trakcie gry można gromadzić przedmioty wchodząc z nimi w kontakt, po którym zostaną automatycznie dodane na pierwsze wolne miejsce w ekwipunku. Przedmioty te można następnie dowolnie układać oraz dzielić na stosy.

Rys. 3.13. Przycisk otwierający lub zamykający okno ekwipunku. (źródło własne)

Rys. 3.14. Widok okna plecaka wraz z kilkoma przedmiotami. (źródło własne)

Rys. 3.15. Pasek szybkiego dostępu wraz z kilkoma przedmiotami. (źródło własne)

Informacja o wszystkich magazynowanych przedmiotach, wykorzystywana przez odpowiednie widoki w celu ich wizualizacji, jest przechowywana w klasie *Storage*. Klasa ta udostępnia iterator, pozwalający zainicjować widoki, oraz wydarzenie, na które widoki mogą się zarejestrować w celu aktualizowania wyświetlanych informacji. Niezbędne metody pozwalające edytować magazynowane przedmioty, wywołują owe wydarzenie.

Wykorzystanie pomocniczej klasy *PositionStorableItemDictionary* pozwala obejść problem silnika z serializacją słowników i skonfigurować początkowe przedmioty gracza z poziomu edytora silnika.

```
public class Storage : IEnumerable
{
   public event Action<int, StorableItem> ChangedItem;
   [SerializeField] private PositionStorableItemDictionary storedItems;

   public void Add(int place, StorableItem itemToStore)
   {
      ...
      ChangedItem?.Invoke(place, stored);
   }
   public void Remove(int place, int amount)
   {
      ...
      ChangedItem?.Invoke(place, stored);
   }
   public IEnumerator GetEnumerator()
   {
      return storedItems.GetEnumerator();
   }
   ...
}
[Serializable]
public class PositionStorableItemDictionary :
↪  SerializableDictionaryBase<int, StorableItem> { }
```

Listing 3.7: Cześć definicji klasy *Storage* oraz niezbędnej klasy pomocniczej. (opr. wł.)

3.7. MODYFIKOWANIE ŚRODOWISKA

Gracz, korzystając z narzędzi, może niszczyć bloki na poszczególnych warstwach terenu. Każdy blok oraz każde narzędzie posiadają określoną liczbę punktów wytrzymałości. Każde uderzenie narzędziem potrafiącym niszczyć wybrany blok, powoduje osłabienie wytrzymałości bloku oraz narzędzia. Gracz uderza blok poprzez przytrzymanie lewego przycisku myszy. Każdemu uderzeniu towarzyszą odpowiednie efekty graficzne oraz dźwiękowe. Gdy wartość wytrzymałości bloku spadnie do zera, zostaje on zniszczony. W konsekwencji zniszczenia bloku, znika jego dotychczasowa reprezentacja graficzna, a pojawia się przedmiot, któremu na starcie nadana jest pewna niewielka siła w losowym kierunku i który gracz może podnieść (przykład na rysunku 3.16).

Rys. 3.16. Przedmiot pojawiający się po zniszczeniu bloku ziemi na warstwie głównej. (źródło własne)

Maksymalna wartość wytrzymałości narzędzia wynosi 100, a gdy spadnie do zera, narzędzie zostanie usunięte z ekwipunku gracza. Jest ona przedstawiana graczowi w postaci paska, którego długość oraz kolor zmieniają się wraz z zużyciem narzędzia jak na rysunku 3.17.

Rys. 3.17. Łopaty o pozostałej wytrzymałości 100, 50 oraz 10. (źródło własne)

3.8. BUDOWANIE

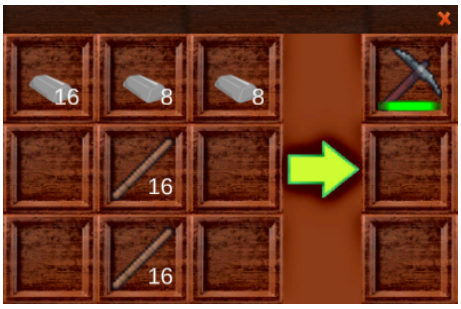
Korzystając z zebranych pozostałości po zniszczonych blokach lub wytwarzając odpowiednie przedmioty, gracz może budować. Wyposażając się w przedmiot bloku oraz wybierając pozycję na mapie i potwierdzając ją prawym przyciskiem myszy, gracz może ustawić w danym miejscu trzymany element, o ile na docelowej warstwie nie ma żadnego innego bloku. Przykład takiej sytuacji przedstawiono na rysunku 3.18.

W niektórych przypadkach, oprócz wymogu braku bloku na warstwie docelowej, pojawiają się dodatkowe reguły. Pochodnia, która należy do warstwy dekoracji, nie może zostać postawiona w miejscu, gdzie znajduje się blok warstwy głównej. Zasada ta działa w dwie strony.

Rys. 3.18. Umieszczanie bloku w pustym miejscu. (źródło własne)

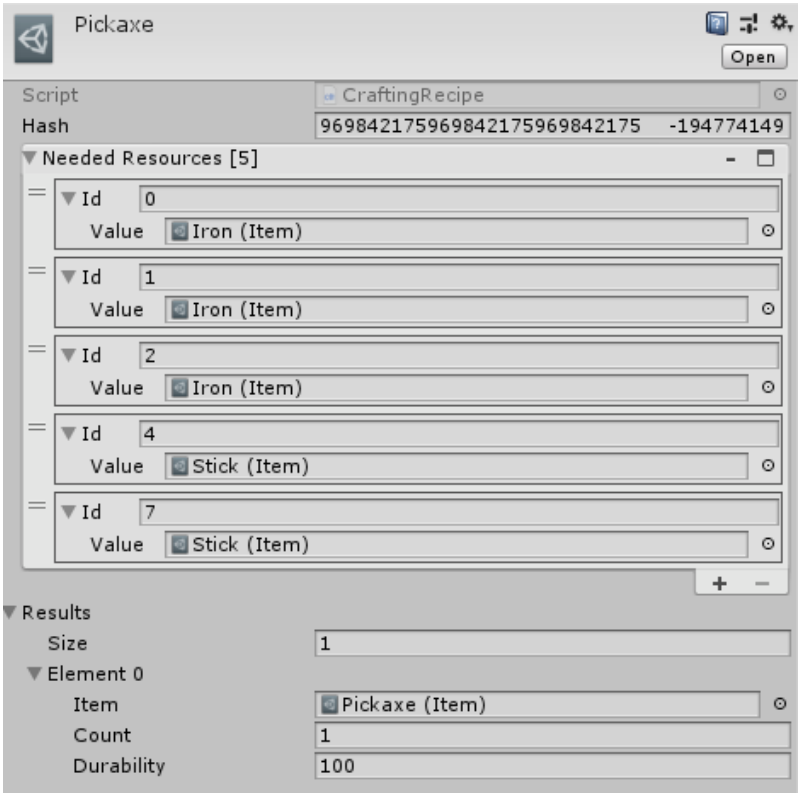
3.9. WYTWARZANIE PRZEDMIOTÓW

Gracz umieszcza przedmioty na kwadratowej siatce o wymiarach 3x3. W efekcie zostaje przedstawione, co może z danej kombinacji przedmiotów wytworzyć, jak ma to miejsce na rysunku 3.19. Użytkownik może przestawić składniki, wycofać je lub zdecydować się je przetworzyć (tracąc je bezpowrotnie) i w efekcie otrzymać nowy przedmiot.



Rys. 3.19. Odpowiednie ułożenie 2 patyków oraz 3 sztabek żelaza pozwalające na wytworzenie kilofu. (źródło własne)

Receptury (schematy) wytwatrzania przedmiotów zostały zdefiniowane z poziomu edytora *Unity* (rys. 3.20), dzięki zastosowaniu struktur jak na listingu 3.8.



Rys. 3.20. Zdefiniowana receptura, pozwalająca wytworzyć kilof. (źródło własne)

Aby usprawnić proces sprawdzania czy z ułożonych w oknie przedmiotów da się wytworzyć jakiś produkt, zastosowano funkcję skrótu (ang. *hash function*). W przypadku każdej receptury policzono wartość funkcji skrótu dla układu jej składników w metodzie *OnValidate*, wykorzystując unikalne identyfikatory przedmiotów. Metoda *OnValidate* w przypadku obiektów dziedziczących po klasach *MonoBehaviour* lub *ScriptableObject*, jest wywoływana przy każdej zmianie wartości którejś ze zmiennej z poziomu edytora, co zaktualizuje również skrót[26].

```
public class CraftingRecipe : ScriptableObject
{
    [SerializeField] private string hash;
    [SerializeField] private PositionItemDictionary neededResources;
    [SerializeField] private StorableItem [] results;
    ...
    private void OnValidate()
    {
        ...
        hash = BuildHash(neededResources);
    }
    private string BuildHash(PositionItemDictionary resources)
    { ... }
}
[Serializable]
public class PositionItemDictionary : SerializableDictionaryBase<int, Item>
↪ { }
```

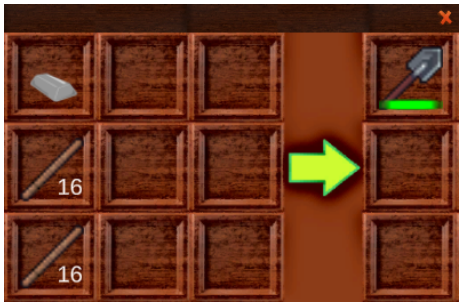
Listing 3.8: Cześć definicji klasy *CraftingRecipe* oraz klasy pomocniczej. (opr. wł.)

Przy każdej zmianie któregoś ze składników w oknie wytwarzania przedmiotów, przeliczana jest wartość skrótu dla aktualnego układu przedmiotów i następuje sprawdzenie czy w słowniku przechowującym wszystkie receptury, istnieje klucz o tej samej wartości (listing 3.9). Następnie taka informacja służy w celu wyświetlenia potencjalnych produktów.

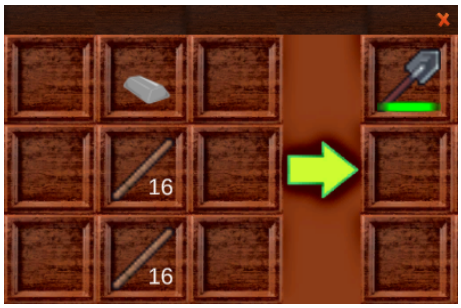
```
public StorableItem [] GetWhatCanBeCrafted(Item [] resources)
{
    var hash = CraftingRecipe.BuildHash(resources);
    return m_recipes.TryGetValue(hash, out var recipe) ? recipe.Results :
    ↪ null;
}
```

Listing 3.9: Metoda szukająca zgodnej receptury. (opr. wł.)

Bezwzględna pozycja przedmiotów na siatce nie jest ważna. Istotne jest względne ułożenie składników wobec siebie. Różne rozmieszczenia tych samych przedmiotów mogą prowadzić do tego samego produktu końcowego, jak w przypadkach z rysunków 3.21 oraz 3.22. Kluczowe w celu osiągnięcia tego efektu, było zaimplementowanie funkcji skrótu, która będzie generowała tożsame wartości w takich sytuacjach (listing 3.10).



Rys. 3.21. Jedno z możliwych pionowych ustawień 2 patyków oraz jednej sztabki żelaza pozwalające na wytworzenie łopaty. (źródło własne)



Rys. 3.22. Inne możliwe pionowe ustawienie 2 patyków oraz jednej sztabki żelaza pozwalające na wytworzenie łopaty. (źródło własne)

```
1  public static string BuildHash(Item [] resources)
2  {
3      var hash = new StringBuilder();
4      for (int y = 0; y < Rows; y++)
5      {
6          for (int x = 0; x < Columns; x++)
7          {
8              var item = resources[y * Columns + x];
9              hash.Append(item != null ? item.UniqueItemId.ToString() : " ");
10         }
11         hash.Append("   ");
12     }
13     return hash.ToString().Trim();
14 }
```

Listing 3.10: Implementacja funkcji skrótu dla schematów składników. (opr. wł.)

Na potrzeby obecnej wersji gry, przygotowano ponad 20 unikalnych receptur wytwarzania przedmiotów. Niektóre z nich układają się w naturalne hierarchie, gdzie produkt jednego schematu służy jako składnik innego. Przykładem, obrazującym tę sytuację, jest żelazny kilof. Aby go wytworzyć wymagane są patyki oraz sztabki żelaza. Żadnego z tych

składników nie znajdziemy w grze podczas eksploracji i wydobywania bloków. Aby pozyskać patyki, wymagane jest przetworzenie desek, aby znowuż wejść w posiadanie desek należy przetworzyć kłody. Dopiero te ostatnie jesteśmy w stanie wydobyć ze świata gry, wycinając drzewa. Podobnie sytuacja ma się ze sztabkami żelaza, do uzyskania których wymagana jest ruda żelaza oraz węgiel.

3.10. MECHANIKI POTRZEB

Stan gracza jest opisywany za pomocą czterech atrybutów. Każdy z nich może przyjmować dowolną liczbę zmiennoprzecinkową z zakresu [0, 100], a aktualna ich wartość jest przedstawiana za pomocą pasków, jak na rysunku 3.23.

— Zdrowie
  Najważniejszy atrybut. W przypadku spadku zdrowia do zera, następuje śmierć gracza. Wartość zdrowia może maleć, w następstwie wyczerpania, któregoś z innych atrybutów lub upadku z wysokości. Można ją podnieść między innymi miksturami przyrządzonymi z odpowiednich roślin.
— Zaspokojenie głodu
  Atrybut zaspokojenia głodu spada regularnie o pewną wartość. W przypadku spadku do zera, kolejne wielkości zostaną odejmowane od atrybutu zdrowia. Można podnieść wartość zaspokojenia głodu między innymi poprzez spożycie przyrządzonego jedzenia.
— Zaspokojenie pragnienia
  Atrybut zaspokojenia pragnienia spada regularnie o pewną wartość. W przypadku spadku do zera, kolejne wielkości zostaną odejmowane od atrybutu zdrowia. Można podnieść wartość zaspokojenia pragnienia między innymi poprzez wypicie zgromadzonej wody.
— Tlen
  Atrybut tlenu spada regularnie o pewną wartość, gdy gracz znajduje się pod wodą, a regeneruje się, gdy gracz znajduje się na powierzchni. W przypadku spadku do zera, kolejne wielkości zostaną odejmowane od atrybutu zdrowia.

Rys. 3.23. Graficzna reprezentacja atrybutów. (źródło własne)

Singletonowa klasa *StatsManager* przechowuje aktualne wartości atrybutów oraz udostępnia metody służące do manipulacji nimi. Dodatkowo można tam znaleźć wydarzenia, wywoływane w momencie zmiany wartości któregoś z atrybutów, używane w celu informowania widoków.

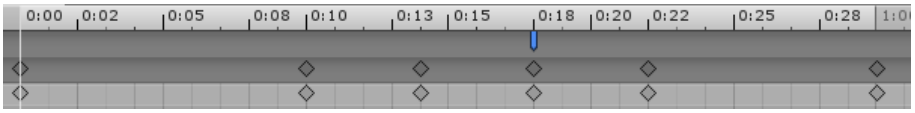
3.11. PROBLEMY IMPLEMENTACYJNE

Zasady działania oraz ograniczenia silnika gry sprawiają, że w trakcie implementacji pojawiły się problemy. Niektóre z nich zostały opisane wraz z wykorzystanymi rozwiązaniami.

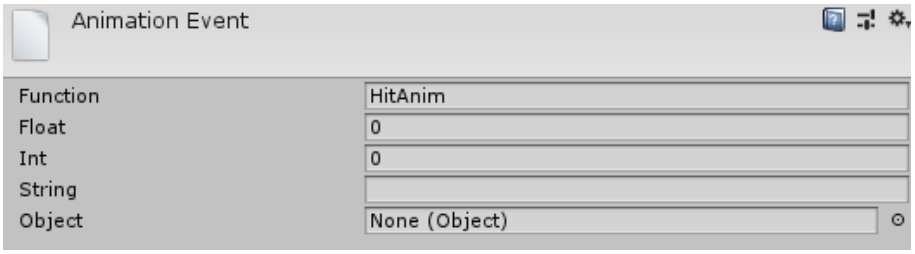
3.11.1. Synchronizacja animacji narzędzi z efektami dźwiękowymi

Gracz używając narzędzi widzi animację, w której narzędzie zmienia swoją rotację, symulując uderzanie. Dźwięk uderzenia powinien być zsynchronizowany z chwilą maksymalnego wychylenia używanego przedmiotu.

Aby rozwiązać problem posłużono się wydarzeniami, które można dołączyć do klipów animacji - *Animation Event*[22]. Obok kluczy animacji, określających rotację przedmiotu, dodano wywołanie wydarzenia w chwili największego wychylenia, jak na rysunku 3.24. Następnie określono nazwę metody, która ma zostać wywołana, na *HitAnim*, pomijając dodatkowe parametry (rys. 3.25). Narzędzie posiada publiczną metodę o takiej nazwie, która obsługuję uderzenie bloku, odtwarzając skonfigurowane nagranie.



Rys. 3.24. Klucze animacji używania narzędzia. (źródło własne)



Rys. 3.25. Konfiguracja wydarzenia animacji. (źródło własne)

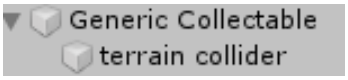
3.11.2. Kolidowanie z przedmiotami

Możliwe do zebrania przedmioty są obiektami, które podlegają fizyce. Muszą posiadać zderzacz (ang. *collider*), niepozwalający im przenikać przez teren. Jednocześnie nie powin-

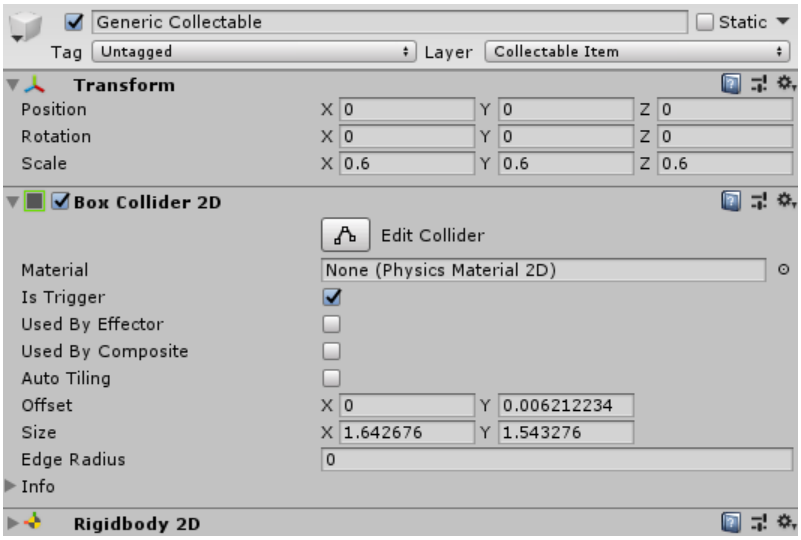
ny one blokować gracza, w szczególności zderzenie z przedmiotem nie powinno wpływać na prędkość ruchu postaci.

Silnik *Unity* pozwala przypisywać obiektom gry warstwy fizyczne i edytować macierz, określającą które warstwy wchodzą ze sobą w kontakt [24]. Każdy zderzacz określa na jakiej zasadzie wchodzą z nim w kontakt inne obiekty i może to być zwykła kolizja lub wyzwalacz (ang. *trigger*), który nie informuje ciała sztywnego (ang. *rigidbody*) o kontakcie[23]. W przypadku przedmiotów, potrzebna jest zwykła kolizja z terenem, a jednocześnie działanie na zasadzie wyzwalacza z postacią gracza.

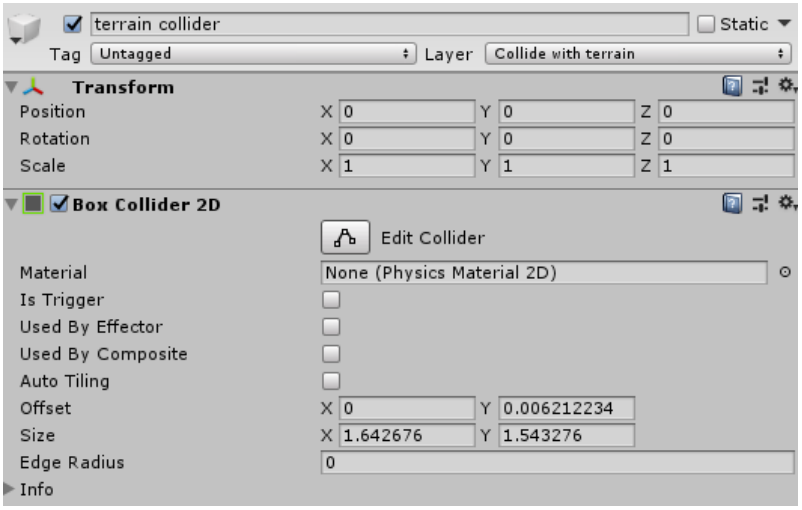
Nie jest możliwe, by jeden zderzacz działał na odmiennej zasadzie w przypadku kolizji z różnymi obiektami. Obchodząc ograniczenia silnika, użyto hierarchii obiektów z rysunku 3.26. Obiekt rodzica (rys. 3.27) znajduje się na warstwie *Collectable Item*, a jego zderzacz działa na zasadzie wyzwalacza, dzięki zaznaczonemu polu *Is Trigger*. Obiekt dziecka (rys. 3.28) znajduje się na warstwie *Collide with terrain* oraz posiada zderzacz o identycznym rozmiarze jak rodzic, ale reagujący zwykłą kolizją. Następnie zostało skonfigurowane, które warstwy wchodzą ze sobą w interakcję, zgodnie z rysunkiem 3.28.



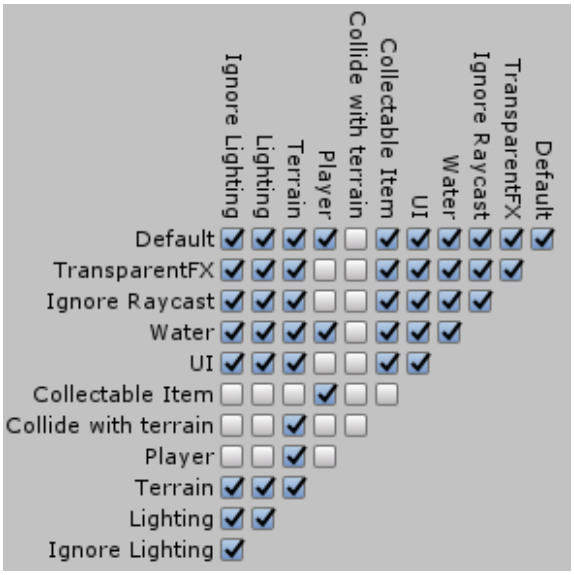
Rys. 3.26. Hierarchia obiektu przedmiotu. (źródło własne)



Rys. 3.27. Podstawowy zderzacz przedmiotów. (źródło własne)



Rys. 3.28. Dodatkowy zderzacz przedmiotów. (źródło własne)



Rys. 3.29. Macierz kolizji. (źródło własne)

3.11.3. Wyrzucanie przedmiotów

W chwili wyrzucenia przedmiotu z ekwipunku, na pozycji gracza pojawia się obiekt, który można ponownie podnieść. Niestety w takiej sytuacji momentalnie zostaje on ponownie umieszczony w ekwipunku, gdyż wykryty zostaje kontakt z postacią gracza.

Aby wyrzucone obiekty nie były od razu podnoszone, dodano dodatkową zmienną *_collectOnContact* typu *bool*. Zmienna ta, w przypadku wyrzucenia przedmiotu z ekwipunku, przyjmuje wartość *false*, natomiast, by przy dotknięciu przez postać gracza przedmiot został podniesiony, musi ona przyjmować wartość *true*. Obsługa tej sytuacji została zaimplementowana, jak na listingu 3.11 w ramach metody *OnTriggerEnter2D*, wywoływanej przez silnik *Unity* przy rozpoczęciu kontaktu z wyzwalaczem.

Przestawienie zmiennej *_collectOnContact* na wartość pozwalającą podnieść przedmiot następuję przy pierwszej utracie kontaktu z postacią gracza w metodzie *OnTriggerExit2D* (listing 3.12), również wywoływanej przez silnik gry.

```
1  private void OnTriggerEnter2D(Collider2D other)
2  {
3     if (!IsPlayer(other.gameObject))
4        return;
5
6     if (_collectOnContact)
7     {
8        Collect();
9     }
10 }
```

Listing 3.11: Obsługa wejścia w kontakt przedmiotów z graczem. (opr. wł.)

```
1  private void OnTriggerExit2D(Collider2D other)
2  {
3     if (!IsPlayer(other.gameObject))
4        return;
5
6     _collectOnContact = true;
7  }
```

Listing 3.12: Obsługa zakończenia kontaktu przedmiotów z graczem. (opr. wł.)

3.11.4. Wypadnięcie poza granice świata

Świat jest ograniczonej wielkości. Gdy gracz dosięgnie granic świata, powinien istnieć mechanizm uniemożliwiający mu bezpowrotne wypadnięcie poza teren gry.

Zdecydowano się, że po przekroczeniu prawej lub lewej granicy świata, postać gracza zostanie przeniesiona na drugi kraniec terenu. W skrypcie powiązanym z graczem, w metodzie *Update*, wykonywanej przez silnik *Unity* w każdej klatce, następuje sprawdzenie czy gracz przekroczył lewą bądź prawą granicę świata. Właściwość *transform.position* zwraca wektor opisujący położenie obiektu, do którego został dołączony skrypt. Teren rozciąga się w osi *OX* od *0* do wartości szerokości świata równej *World.Instance.WorldWidth*.

Zgodnie z implementacją z listingu 3.13 w przypadku przekroczenia granic terenu przez obiekt, nastąpi przeniesienie go, na przeciwległy kraniec. Użyta w liniach 6 oraz 12 metoda *GetGroundLevel* zwraca dla zadanej szerokości wysokość terenu. Określenie odpowiedniej

wysokości, na którą przemieszczony zostanie gracz, jest istotne, by nie przesunąć go w miejsce, gdzie znajdują się bloki, co spowodowałoby jego utknięcie. Z drugiej strony zbyt duża wartość wysokości mogłaby spowodować szybką śmierć przez upadek.

```
private void Update()
{
   if (transform.position.x > World.Instance.WorldWidth)
   {
      var x = 0;
      var y = GetGroundLevel(x) + PLAYER_HEIGHT_OFFSET;
      transform.position = new Vector3(x + 0.5f, y, transform.position.z);;
   }
   else if(transform.position.x < 0)
   {
      var x = World.Instance.WorldWidth - 1;
      var y = GetGroundLevel(x) + PLAYER_HEIGHT_OFFSET;
      transform.position = new Vector3(x + 0.5f, y, transform.position.z);;
   }
}
```

Listing 3.13: Obsługa przekroczenia granic świata. (opr. wł.)

Permanentne wypadnięcie poza górną granicę świata jest nierealne z racji istnienia grawitacji. Wyjście poza dolny kraniec świata jest niemożliwe, dzięki niezniszczalnym blokom, znajdującym się w najniższych partiach terenu. Niewrażliwe na uszkodzenia bloki, noszące nazwę skały macierzystej (ang. *bedrock*), są stosowane także w takich grach jak *Minecraft* oraz *Terraria*.

3.12. EFEKT KOŃCOWY

3.12.1. Ekrany gry

Na rysunku 3.30 widoczne jest menu główne z polami do wprowadzenia wielkości świata oraz opcjonalnego ziarna generatora i przyciski do uruchomienia oraz opuszczenia gry.

Rys. 3.30. Skończone menu główne gry. (źródło własne)

Na rysunkach 3.31 oraz 3.32 widoczny jest ekran gry wraz z ekwipunkiem, panelem wytwarzania przedmiotów i innymi elementami interfejsu. Pierwszy ze zrzutów ekranu został zrobiony w dzień, a drugi w nocy.

Rys. 3.31. Ekran gry w dzień. (źródło własne)

Rys. 3.32. Ekran gry w nocy. (źródło własne)

Na rysunkach 3.33 oraz 3.34 widoczne są kolejno ekran pauzy oraz ekran porażki z wszystkimi niezbędnymi informacjami oraz przyciskami nawigacyjnymi.

Rys. 3.33. Skończony ekran pauzy. (źródło własne)

Rys. 3.34. Skończony ekran porażki. (źródło własne)

3.12.2. Dostęp do gry

Gra została umieszczona w serwisie skupiającym niezależnych twórców gier *itch.io* i jest dostępna pod łączem: `https://dawidantczak.itch.io/blockbox`.

Na stronie znajdują się zrzuty ekranu, instrukcja oraz plik z grą do pobrania. Poza tym opisane zostały tam plany rozwojowe.

4. TESTY

W ramach projektu zostały przeprowadzone testy sprawdzające szybkość procesu generowania świata dla różnych parametrów wielkości oraz płynność rozgrywki. Odbywały się one na komputerze osobistym o poniższej specyfikacji:

— System operacyjny: *Windows 10*
— Procesor: *Intel Core i7-8750H*
— Karta graficzna: *GeForce GTX 1060*
— Pamięć RAM: *16GB DDR3*

4.1. CZAS GENEROWANIA ŚWIATA

Do pomiaru czasu generowania świata użyto klasy *System.Diagnostics.Stopwatch*[10]. Dodano logikę startującą stoper na początku procesu generowania oraz wyświetlającą czas, gdy świat jest gotowy. Pomiary przeprowadzono trzykrotnie dla różnych wielkości świata i ziarna równego *0*. Wyniki zostały przedstawione w tabeli 4.1.

Zgodnie z oczekiwaniami, czas generowania świata wzrasta w przybliżeniu liniowo w stosunku do liczby bloków składających się na niego i nie wymaga dalszych optymalizacji na tym etapie projektu.

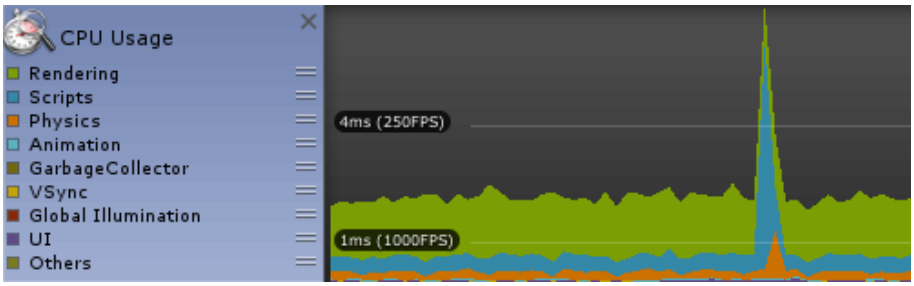
Tabela 4.1. Czas generowania świata dla różnych wielkości.

szerokość świata	wysokość świata	czas 1 [ms]	czas 2 [ms]	czas 3 [ms]
256	128	184	166	171
512	128	363	379	344
1024	128	710	682	649
2048	128	1429	1380	1334
2048	256	2565	2745	2649
4096	256	5851	5718	5676
4096	512	11874	12122	12119

Testy zostały przeprowadzone w wersji *Development Build* pozwalające na użycie narzędzia profilującego *Profiler*, dostępnego w *Unity*. Tym samym, wydajność w końcowym produkcie powinna być jeszcze wyższa, aniżeli w przedstawionych wynikach.

Na potrzeby testu wygenerowany został świat o rozmiarze 4096 na 512 bloków. W ramach rozgrywki wykonywane były podstawowe czynności, które może robić gracz.

Standardowa liczba generowanych klatek na sekundę w trakcie gry wynosiła około 500. Znaczne spadki pojawiały się w momencie edytowania środowiska (budowanie, niszczenie) oraz podczas doładowywania terenu, przemierzając go, co widoczne jest na rysunku 4.1. Obie sytuacje są spowodowane tymi samymi czynnikami, którymi są przeliczanie oświetlenia oraz siatek zderzaczy. Przy każdej zmianie w terenie te aspekty muszą być wyliczone na nowo. Nie wpływa to jednak w żaden sposób na komfort grania i nie jest zależne od wielkości świata, gdyż narzędzie *Terrain Engine 2D* optymalizuje te procesy, dzieląc teren na kawałki (ang. *chunks*)[32].



Rys. 4.1. Chwilowy spadek wydajności przy przeliczaniu terenu. (źródło własne)

Przeprowadzono również test, który miał za zadanie sprawdzić czy dojdzie do obniżenia wydajności w przypadku rozstawienia dużej liczby źródeł światła w postaci pochodni. Zostały one rozmieszczone blisko siebie, jak na rysunku 4.2.

Mimo wielu obiektów znajdujących na ekranie, które generują światło oraz posiadają efekty cząsteczkowe (ogień), nie zaobserwowano żadnego spadku wydajności.

Rys. 4.2. Rozmieszczenie pochodni na potrzeby testu wydajności. (źródło własne)

ZAKOŃCZENIE

W pracy zostały zrealizowane wszystkie postawione cele. Stworzona gra, posiada mechaniki typowe dla gatunku *survival*, które stanowią trzon zabawy. Świat jest generowany proceduralnie i w znaczący sposób różni się na potrzeby każdego podejścia. Gra została przystosowana do komputerów osobistych i zgodnie z wynikami przeprowadzonych testów, działa w sposób płynny.

Starano się zachować spójną oprawę graficzną, co jest trudne z racji ograniczonych środków. Nie zawsze możliwe było znalezienie idealnych ikon lub tekstur, które byłyby darmowe.

Zakres poruszonych w ramach pracy zagadnień jest bardzo szeroki, ponieważ wymagane było stworzenie grywalnego produktu. Na szczęście, użycie narzędzi takich, jak *Unity* oraz *Terrain Engine 2D* znacznie przyspieszyło realizację, pozwalając skupić się na wysokopoziomowych aspektach implementacji.

Produkt jest grywalny, ale znajduje się we wczesnej fazie rozwoju ze stosunkowo niewielką ilością zawartości. Użyte rozwiązania, sprawiają jednak, że bardzo łatwo jest go rozwijać. Dodawanie kolejnej zawartości w postaci przedmiotów, schematów ich wytwarzania oraz bloków jest bardzo proste i nie wymaga pisania żadnego kodu.

Kolejnymi krokami w rozwoju powinno być dodanie zapisywania stanu rozgrywki, stworzeń niezależnych, upraw roślin, rozbudowanych podziemi z pułapkami i skarbami do odnalezienia, efektów pogodowych, pór roku oraz zróżnicowanych biomów z unikalną fauną i florą. Planowana powinna być także zmiana grafik na takie, które mogłyby nadać grze niepowtarzalnego klimatu. W dalszej przyszłości rozważone powinno zostać dodanie rozgrywki wieloosobowej, która idealnie sprawdziłaby się w oferowanej formie zabawy.

Dzięki użyciu silnika *Unity* możliwe jest stworzenie portów na inne platformy do gier. Obrany styl graficzny jak i dotychczasowa wysoka płynność gry, sprawiają, że urządzenia mobilne powinny być naturalną formą ekspansji. Jedynym elementem wymagającym dostosowania jest interfejs.

Gra może z czasem pójść w ślady znanych dzieł, przytoczonych w ramach analizy konkurencyjnych rozwiązań. Startowały one skromnie i były rozwijane latami, w dużej mierze z czynnym udziałem społeczności.

BIBLIOGRAFIA

[1] Barbosa, M.B., Rêgo, A.B., de Medeiros, I., *Developing games with object composition: A case study using the Unity3D platform*. 2015.

[2] Bates, B., *Game Design*, wyd. 2 wyd. (Thomson Course Technology, 2004).

[3] Gamma, E., Helm, R., Johnson, R., Vlissides, J.M., *Design Patterns: Elements of Reusable Object-Oriented Software.*, wyd. 1 wyd. (Reading, Mass., Addison-Wesley, 1994).

[4] Ivan-Assen Ivanov, *Practical Texture Atlases*, https://www.gamasutra.com/view/feature/130940/practical_texture_atlases.php. Ost. dost. 20 listopada 2019.

[5] JetBrains, *Unity support: Rider vs. Visual Studio vs. Visual Studio for Mac*, https://www.jetbrains.com/rider/compare/unity-rider-vs-visual-studio/. Ost. dost. 20 listopada 2019.

[6] Khan Academy, *Szum Perlina*, https://pl.khanacademy.org/computing/computer-programming/programming-natural-simulations/programming-noise/a/perlin-noise. Ost. dost. 29 października 2019.

[7] Klei Entertainment, *Don't Starve DLC*, https://dontstarve.fandom.com/wiki/DLC. Ost. dost. 26 października 2019.

[8] Marchiafava, J., *Don't Starve: A Life-And-Death Struggle In Need Of A Point*, https://www.gameinformer.com/games/dont_starve/b/pc/archive/2013/04/26/dont-starve-review-a-life-and-death-struggle-without-a-point.aspx. Ost. dost. 26 października 2019.

[9] Maria B. Garda, B.L., *Indie games: fenomen niezależnych gier komputerowych*. 2011.

[10] Microsoft, *Stopwatch Class*, https://docs.microsoft.com/en-gb/dotnet/api/system.diagnostics.stopwatch. Ost. dost. 30 listopada 2019.

[11] Mojang AB, *Different Minecraft Editions*, https://help.mojang.com/customer/en/portal/articles/1274139-different-minecraft-editions/. Ost. dost. 26 października 2019.

[12] Olsen, J., *Realtime procedural terrain generation - realtime synthesis of eroded fractal terrain for use in computer games*. 2004.

[13] Persson, M., *Terrain generation, Part 1*, https://notch.tumblr.com/post/3746989361/terrain-generation-part-1. Ost. dost. 29 października 2019.

[14] Players Headquarters, TJ Vore, *How Minecraft has changed gaming entirely*, https://bvnwnews.com/opinion/blogs/2018/01/30/players-headquarters-how-minecraft-has-changed-gaming-entirely/. Ost. dost. 26 października 2019.

[15] Polsinelli, P., *Why is Unity so popular for videogame development?*, https://designagame.eu/2013/12/unity-popular-videogame-development/. Ost. dost. 20 listopada 2019.

[16] Radzewicz, S., *Gry z gatunku survival podbijają rynek PC. Jestem w (komputerowym) niebie*, https://www.spidersweb.pl/2014/07/gry-gatunku-survival-dominuja-rynek-pc-niebie.html. Ost. dost. 30 listopada 2019.

[17] Re-Logic, *Terraria: Game platform*, https://terraria.gamepedia.com/Game_platform. Ost. dost. 26 października 2019.

[18] Re-Logic, *Terraria State of the Game - May 2019*, https://forums.terraria.org/index.php?threads/terraria-state-of-the-game-may-2019.79706/. Ost. dost. 26 października 2019.

[19] Silverman, M., *Minecraft: How Social Media Spawned a Gaming Sensation*, https://mashable.com/2010/10/01/minecraft-social-media/. Ost. dost. 26 października 2019.

[20] Skeet, J., *Random numbers*, https://csharpindepth.com/articles/Random. Ost. dost. 25 listopada 2019.

[21] Togelius, J., Kastbjerg, E., Schedl, D., Yannakakis, G., *What is Procedural Content Generation? Mario on the borderline*. 2011.

[22] Unity Technologies, *AnimationEvent*, https://docs.unity3d.com/ScriptReference/AnimationEvent.html. Ost. dost. 30 listopada 2019.

[23] Unity Technologies, *Collider.isTrigger*, https://docs.unity3d.com/ScriptReference/Collider-isTrigger.html. Ost. dost. 30 listopada 2019.

[24] Unity Technologies, *Layer-based collision detection*, https://docs.unity3d.com/Manual/LayerBasedCollision.html. Ost. dost. 30 listopada 2019.

[25] Unity Technologies, *Mathf.PerlinNoise*, https://docs.unity3d.com/ScriptReference/Mathf.PerlinNoise.html. Ost. dost. 30 listopada 2019.

[26] Unity Technologies, *MonoBehaviour.OnValidate()*, https://docs.unity3d.com/ScriptReference/MonoBehaviour.OnValidate.html. Ost. dost. 29 listopada 2019.

[27] Unity Technologies, *ScriptableObject*, https://docs.unity3d.com/Manual/class-ScriptableObject.html. Ost. dost. 26 listopada 2019.

[28] Unity Technologies, *Unity User Manual*, https://docs.unity3d.com/Manual/UnityManual.html. Ost. dost. 20 listopada 2019.

[29] Valentine, R., *Minecraft has sold 176 million copies worldwide*, https://www.gamesindustry.biz/articles/2019-05-17-minecraft-has-sold-176-million-copies-worldwide. Ost. dost. 26 października 2019.

[30] Wawro, A., *Klei sold more than a million copies of Don't Starve in 2013*, https://www.gamasutra.com/view/news/208997/Klei_sold_more_than_a_million_copies_of_Dont_Starve_in_2013.php. Ost. dost. 26 października 2019.

[31] Wijman, T., *The Global Games Market Will Generate $152.1 Billion in 2019 as the U.S. Overtakes China as the Biggest Market*, https://newzoo.com/insights/articles/the-global-games-market-will-generate-152-1-billion-in-2019-as-the-u-s\-overtakes-china-as-the-biggest-market/. Ost. dost. 4 grudnia 2019.

[32] Wilson, M., *TerrainEngine2D.Chunk Class Reference*, https://activegamedev.com/API/html/class_terrain_engine2_d_1_1_chunk.html. Ost. dost. 30 listopada 2019.

[33] Winkie, L., *Terraria review*, https://www.pcgamer.com/terraria-review/. Ost. dost. 26 października 2019.

SPIS RYSUNKÓW

SPIS LISTINGÓW

SPIS TABEL