Politechnika Wrocławska

Wydział Informatyki i Zarządzania

Kierunek studiów: Informatyka

Praca dyplomowa – inżynierska

GRA MOBILNA Z PROCEDURALNYM GENEROWANIEM POZIOMÓW W SILNIKU UNITY

Konrad Jakubowski

słowa kluczowe:

generowanie proceduralne, Unity, gra mobilna

krótkie streszczenie:
W pracy przedstawiono projekt i prototypową implementację gry zrealizowanej w silniku Unity przeznaczonej na urządzenia mobilne z system Android, która wykorzystuje proceduralne generowanie w celu kreowania poziomów.

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, rok 2019

SPIS TREŚCI

STRESZCZENIE

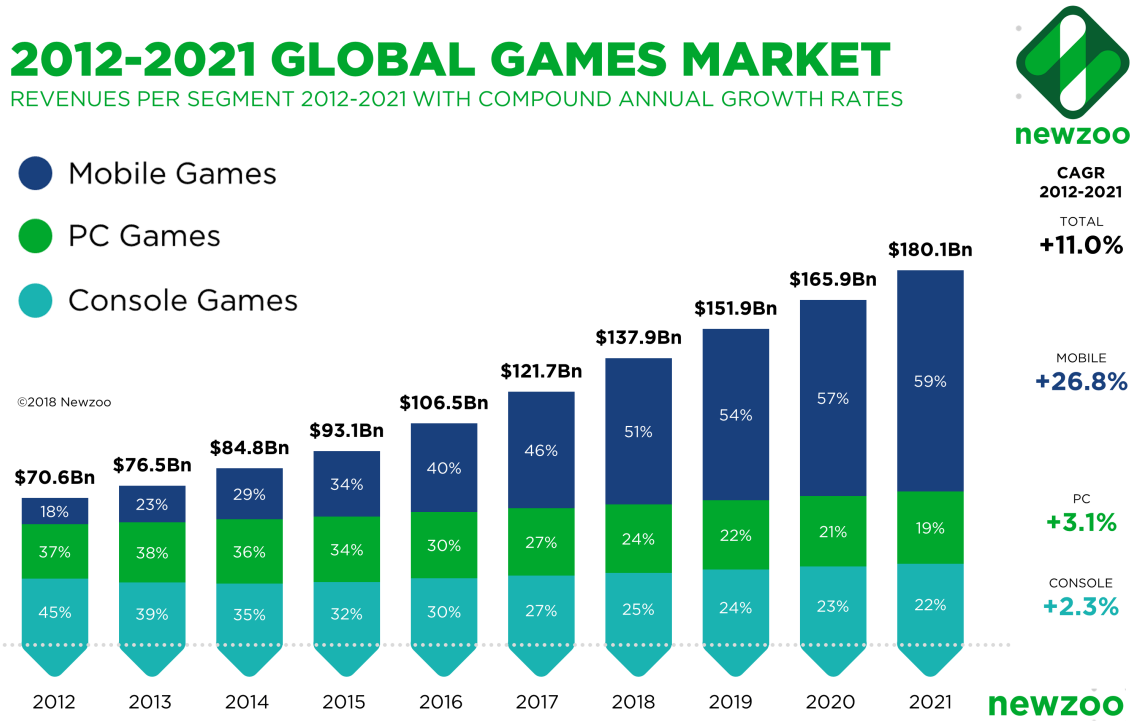
Celem pracy był projekt i prototypowa implementacja gry przeznaczonej na platformę mobilną, która będzie wykorzystywać algorytmy do proceduralnego generowania poziomów. Przygotowana implementacja w silniku Unity, wykorzystuje algorytm Prima do generowania labiryntów, gdzie każde jego pole jest większym obszarem planszy, do wygenerowania której użyto m.in. automatu komórkowego, algorytmu Prima, algorytmu przeszukiwania wszerz, algorytmu Bresenhama. Dla spełnienia części grywalnej projektu, przygotowano m.in.: postać gracza, z obsługą poruszania na urządzeniach mobilnych, możliwość strzelania, postacie przeciwników, przeszkody na planszy.

ABSTRACT

The goal of the work was to design and prototype implementation of a mobile game that will use algorithms for procedural level generation. Prepared implementation in the Unity engine, uses Prim's algorithm to generate maze, where each of its chunks is a larger area generated using, inter alia, cellular automata, Prims' algorithm, breadth-first search algorithm, Bresenham's line algorithm. For the fulfillment of the playable part of the project, there were prepared, inter alia: a player character, support for controls on mobile devices, the shooting functionality, enemy characters, obstacles on the level.

WSTĘP

Wybór realizacji gry jako tematu pracy dyplomowej wynika z zainteresowań autora w zakresie projektowania i implementacji gier komputerowych oraz chęcią rozwoju w tej dziedzinie. Ograniczenie tematu do gry mobilnej kierowane było tym, że od wielu lat zauważa się rosnący trend udziału urządzeń mobilnych w zyskach całej branży gier ([34], rys. 1). Udział ten około dwa lata temu przekroczył zyski z innych głównych typów urządzeń dla graczy – konsol i komputerów osobistych. Natomiast na rok 2018 szacuje się, że będzie stanowił większość całego rynku. Urządzenia mobilne stają się coraz wydajniejsze i powoli, ale stale zaczynają zastępować pozostałe urządzenia. Według prognoz taka tendencja się nie zmieni.



Rys. 1: Udział w zyskach branży gier w zależności od typu urządzenia, wraz z prognozą. Lata 2012-2021. (źródło: [34])

Gry na urządzenia mobilne początkowo były bardzo proste, co wynikało z wydajności tych urządzeń. Jednak wraz z ich rozwojem, gry zaczęły być coraz większe, bardziej rozbudowane, lepiej dopracowane. Co w połączeniu ze wzrostem popularności urządzeń mobilnych skutkowało tym, że producenci gier zaczęli poświęcać na gry mobilne coraz

większe budżety. Wciąż nie są to budżety porównywalne z produkcjami przeznaczonymi na urządzenia stacjonarne. Jednak można porównać obecny stan do historii początków gier komputerowych, kiedy to komputery o słabych parametrach wymuszały tworzenie mniej rozbudowanych gier.

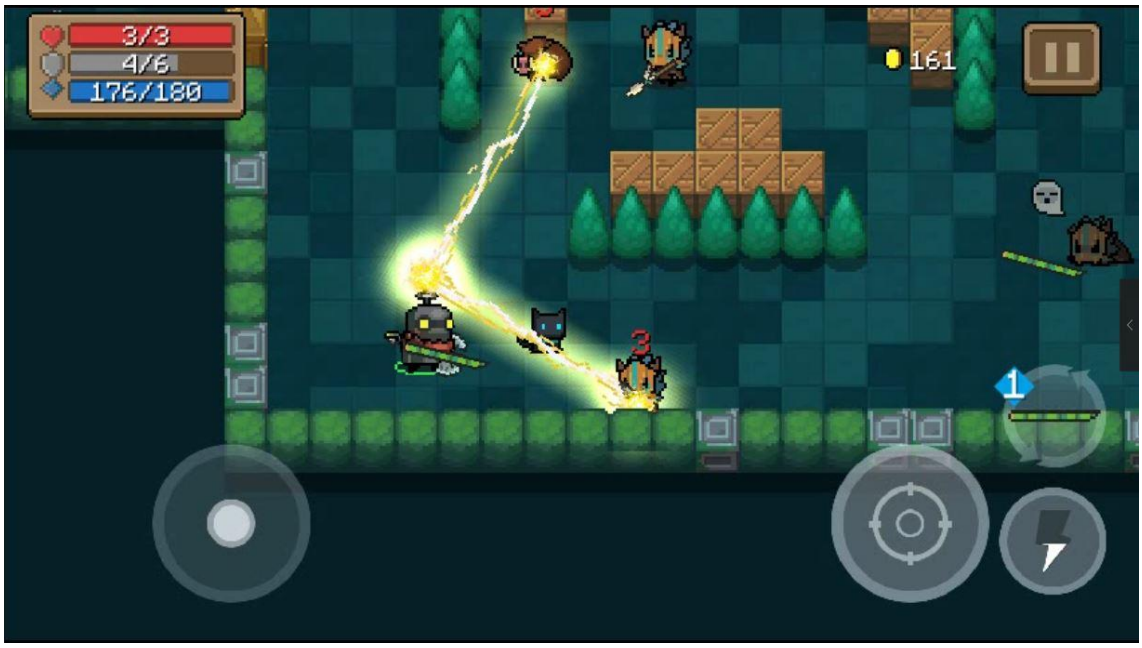
Przy coraz wyższych kosztach produkcji, istotne jest, aby szukać sposobów, które pozwolą na ich zmniejszenie. Jednym z kluczowych kosztów ponoszonych przy produkcji jest projektowanie i implementacja zawartości gry. Sposobem na ich zmniejszenie jest wykorzystanie proceduralnego generowania zawartości. Najczęściej oznacza ono losowe generowanie treści. Proceduralne generowanie może dotyczyć dowolnego obszaru gry: poziomów, postaci, grafiki, mechanik, zasad rozgrywki, fabuły. Wszystkie zostały szeroko opisane w książce Shakera, Togeliusa i Nelsona: *Procedural Content Generation in Games* [18]. Wadą proceduralnego generowania jest słabsza kontrola nad tym czego będzie doświadczał gracz. Jednak da się temu zapobiec ograniczając losowość, można to zrobić narzucając zasady, według których elementy są generowane. Możliwe jest też ograniczenie proceduralnego generowania do minimum poprzez generowanie elementów z wcześniej przygotowanych specjalnych szablonów opisujących te elementów. Przykładowo do generowania poziomu może zostać wykorzystana wcześniej zaprojektowana pula części składowych tego poziomu. Zapewnia to lepszą kontrolę nad doświadczeniami gracza, a zarówno zapewnia świeżość rozgrywki dzięki zastosowaniu losowości.

Przegląd istniejących rozwiązań

Przykładem gry wykorzystującej proceduralne generowanie jest trójwymiarowa gra *Minecraft* studia *Mojang* wspierająca wiele platform. Pozycja ta zdobyła ogromną popularność, głównie wśród dzieci i młodzieży, która mimo lat jakie upłynęły od pierwszej publikacji tytułu, wcale nie słabnie. Sama gra, dzięki łagodnej formie artystycznej, jest obecnie często wykorzystywana w celach edukacyjnych, m.in. w nauce programowania dla dzieci. W grze *Minecraft* proceduralne generowanie jest wykorzystane do tworzenia trójwymiarowych poziomów - ogromnych interaktywnych obszarów terenu z ogromną liczbą elementów składowych. Każdy wygenerowany poziom jest w pełni unikalny. Co w połączeniu z otwartymi zasadami rozgrywki, pozostawia graczowi dużą dowolność tego co będzie w świecie gry robił.

Innym przykładem jest gra *Soul Knight* studia *ChillyRoom*. Gra jest przeznaczona na urządzenia mobilne. Rozgrywka jest przedstawiona w postaci dwuwymiarowej, z widokiem z góry. Ekran rozgrywki został przedstawiony na rysunku 2. Gra polega na przechodzeniu kolejnych niewielkich komnat, które razem tworzą większy labirynt. W każdej komnacie gracz ma za zadanie pokonać pojawiających się w niej przeciwników, w tym celu może korzystać z dostępnych w grze broni. Gra umożliwia również grę z innymi użytkowni-

kami. *Soul Knight* wykorzystuje proceduralne generowanie do generowania poziomów o strukturach labiryntu w postaci niewielkich połączonych ze sobą komnat, które z kolei są generowane na podstawie gotowych szablonów. Dodatkowo w grze występują losowe mechaniki, na których wybór gracz ma wpływ - gra co jakiś czas oferuje wybór jednej z kilku mechanik. Gra ta będzie dobrą inspiracją dla realizowanego projektu.



Rys. 2: Ekran rozgrywki gry mobilnej *Soul Knight* (źródło: sklep *Google Play* [6])

Cel i zakres pracy

Celem pracy jest wykonanie projektu i implementacji prototypu gry komputerowej, która nie będzie posiadać ręcznie zaprojektowanych poziomów, zamiast tego będzie je generować proceduralnie. Gra powinna zapewnić wsparcie urządzeniom mobilnym. Implementacja powinna zostać przeprowadzona w silniku gier Unity.

Zakres pracy obejmuje analizę i wybór algorytmów do proceduralnego generowania poziomów. Przygotowanie projektu rozgrywki odbywającej się na wygenerowanych poziomach, gdzie projekt uwzględnia wsparcie dla urządzeń mobilnych. Następnie implementację projektu. Realizacja zostanie przeprowadzona w silniku gier Unity, który umożliwi późniejszą instalację przygotowanego prototypu na urządzeniach z systemem Android. Wykorzystanym środowiskiem programistycznym będzie *Microsoft Visual Studio*. Do testów zostanie wykorzystany telefon z systemem *Android 6.0 Marshmallow*.

Układ pracy

Praca składa się z pięciu rozdziałów, wstępu, zakończenia i wykazu literatury. W rozdziale pierwszym omówione zostały wymagania i zasady, które tworzona gra powinna spełnić, przedstawiono w nim wymagania funkcjonalne i niefunkcjonalne. W rozdziale drugim została przeprowadzona analiza wymagań, w której dobrano algorytmy umożliwiające proceduralne generowanie poziomów oraz została przeprowadzona analiza w celu określenia widoków, jakie powinny w grze występować. W rozdziale trzecim zostały przedstawione najważniejsze wykorzystane technologie i narzędzia, z których korzystano przy tworzeniu pracy. Najważniejszą częścią pracy są rozdziały czwarty i piąty. W czwartym przeprowadzono pełny projekt realizacji prototypu. W piątym dokonano opisu najistotniejszych części implementacji, jak również opisano napotkane podczas niej problemy. W zakończeniu podsumowano efekty realizacji zamierzonych celów i spojrzano na projekt pod kątem potencjalnego rozwoju.

1. SPECYFIKACJA WYMAGAŃ I ZASAD ROZGRYWKI

W celu zaprojektowania gry i zaimplementowania jej prototypu, konieczne jest wcześniejsze określenie zasad rozgrywki, aby było możliwe wyspecyfikowanie bardziej precyzyjnych wymagań. Na początek zostanie przedstawiona ogólna identyfikacja wymagań wynikająca z tematu i celu pracy. Następnie zostanie przedstawiony koncept zasad rozgrywki dla projektowanej gry. Całość pozwoli na identyfikację precyzyjnych wymagań aniżeliby to wynikało jedynie z wymagań bezpośrednio powiązanych z tematem pracy.

1.1. IDENTYFIKACJA WYMAGAŃ PODSTAWOWYCH

Nad projektowaniem i implementacją gier komputerowych najczęściej pracują zespoły osób, gdzie wielkość zespołu wynika bezpośrednio z planowanego rozmiaru i złożoności powstającej gry. Mniejsze gry, tworzone przez pojedyncze osoby są często albo bardzo proste, albo ich realizacja jest bardzo rozłożona w czasie. W przypadku produkcji gry dotyczącej tej pracy, jej projekt (wynikający z celu pracy) nie będzie prosty, natomiast czas realizacji jest mocno ograniczony. Stąd należy przyjąć pewne ograniczenia dotyczące gry, której w tej pracy zostanie przygotowany jedynie prototyp aniżeli kompletny produkt. Podstawowymi wymaganiami są przede wszystkim te zawarte w nazwie tematu pracy, czyli: gra ma być przeznaczona na platformy mobilne, ma wykorzystywać generowanie proceduralne w celu konstruowania poziomów, głównym narzędziem do jej przygotowania ma być silnik Unity.

Z tak ogólnie zarysowanymi wymaganiami, można przejść do ich dalszego ograniczania. Jako że platform mobilnych jest wiele, a uwzględnianie wszystkich byłoby zbyt czasochłonne w procesie produkcji i testowania, to projekt ograniczy się do przygotowania wersji gry na platformę Android. Decyzja ta wynika z ogromnej popularności i wsparcia tego systemu, jak również tego iż autor jest w posiadaniu urządzenia z tym systemem, co znacznie ułatwi proces testowania.

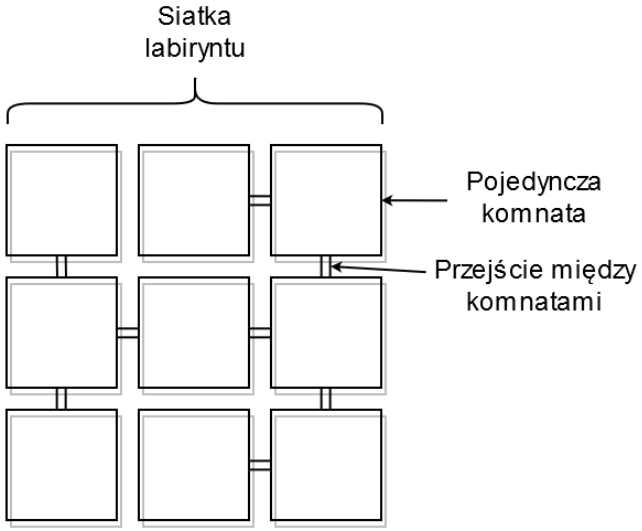
Proceduralne generowanie może obejmować wiele elementów gry: grafikę, projekt poziomu, mechaniki (zasady) rozgrywki, itp. Wymaganiem podstawowym, wynikającym z tematu pracy jest generowanie poziomów w sposób proceduralny, pozostałe elementy nie będą generowane proceduralnie. Jednak nie jest wykluczone generowanie innych elementów w przypadku dalszego rozwoju produktu.

Jedynym warunkiem dotyczącym generowania poziomów jest to, że każdy z poziomów musi być możliwy do przejścia. Wymaganiem powiązanym jest możliwość gry w nieskończoność, choć nie dotyczy ono bezpośrednio sposobu generowania.

Każda gra powinna być reprezentowana przez warstwę graficzną i dźwiękową. Dźwięki zostaną w całości pominięte w tym prototypie. Natomiast podstawowymi założeniami realizacji warstwy graficznej będzie inspiracja stylem gier retro, jednak uwaga poświęcana warstwie graficznej będzie marginalna i zrealizowana jedynie na potrzeby subiektywnie estetycznej wizualizacji prototypu. Wynika to też z faktu, aby na chwilę obecną nie przywiązywać się zbyt mocno do konkretnego stylu graficznego, co w przypadku chęci rozwoju produktu może umożliwić jego łatwą zmianę lub określenie dokładnego kierunku warstwy audiowizualnej.

1.2. OPIS ROZGRYWKI

Poziom składa się z siatki większych obszarów terenu, tzw. komnat, które są generowane proceduralnie. Sama siatka stanowi labirynt z określonymi połączeniami między komnatami. Szkic takiej struktury został przedstawiony na rysunku 1.1. Komnata jako obszar terenu składa się z nienaruszalnych ścian i podłoża. W procesie generacji komnaty czasami pojawiają się na niej skrzynie możliwe do zniszczenia. Nie ma możliwości wyjścia poza mapę gry - każdy poziom musi być otoczony ścianami. W momencie wygenerowania poziomu, dwie losowe komnaty są określane jako komnata startowa i finałowa.

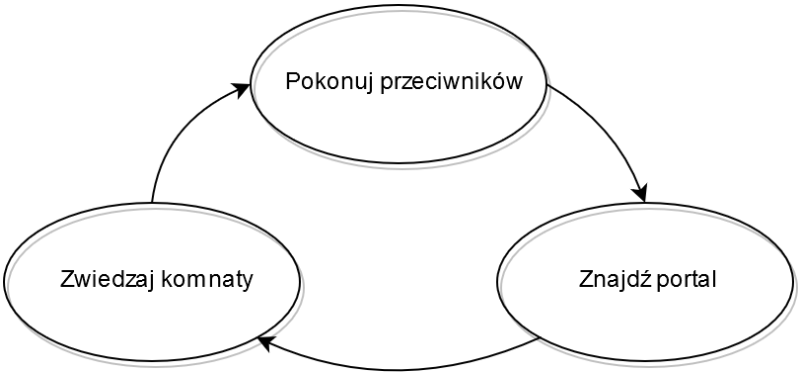


Rys. 1.1: Szkic struktury poziomu (opr. wł.)

Postać gracza pojawia się w komnacie startowej i jego celem jest odnalezienie komnaty finałowej, w której znajduje się specjalny obiekt - *portal* do kolejnego poziomu. Oznacza to

tym samym, że gracz ma możliwość poruszania się własną postacią. Przeszkodą dla gracza na drodze ku zwycięstwu, będą stanowić przeciwnicy komputerowi. W momencie gdy gracz wejdzie do danej komnaty po raz pierwszy, na kilku jej wolnych polach pojawią się postacie przeciwników. Wyjście z komnaty (przejście do innych komnat) zostanie zablokowane dopóty, dopóki wszyscy przeciwnicy nie zostaną unieszkodliwieni.

Gracz może unieszkodliwić przeciwników dzięki możliwości strzelania pociskami. Trafienie przeciwników określoną liczbę razy spowoduje ich zniknięcie z planszy rozgrywki. Jednak to zadanie również zostanie utrudnione. Przeciwnicy poruszają się i strzelają w kierunku gracza. Gracz powinien jednocześnie unikać pocisków przeciwników i samemu strzelać w ich kierunku. Pocisk rozbija się o pierwszy trafiony obiekt (w tym ścianę).



Rys. 1.2: Główna pętla celów gry, tzw. *core loop* [14] (opr. wł.)

Całość rozgrywki można by opisać specjalnym schematem, tzw. *core loop'em* [14], który opisuje cele gry w sposób maksymalnie uproszczony, a zarazem pozwalający dać użytkownikowi wyobrażenie na temat występujących w grze najważniejszych mechanik. Celem definiowania takiego schematu jest znalezienie i zrozumienie podstawowych mechanik produkowanej gry i zadbanie o to, aby były one jak najciekawsze, gdyż to z nimi gracz spędzi najwięcej czasu. *Core loop* dla tego projektu został przedstawiony na rysunku 1.2. Można z niego wyszczególnić między innymi dwie aspekty rządzące tą grą, są to: walka i zwiedzanie. Należy zadbać, aby oba te elementy były maksymalnie doszlifowane. Oczywiście nie ma możliwości wystarczającego dopracowania tych elementów w czasie realizacji tej pracy, gdyż wymaga to znacznie większych pokładów czasowych i zbieraniu opinii od potencjalnych klientów. Mimo to, w jakimś stopniu można zadbać o poprawienie tych elementów.

Należy się zastanowić czy walka będzie wystarczająco atrakcyjna i jak można temu zaradzić już na etapie projektowania. Można przypuszczać, że walka będzie frustrująca, bo gracz trafiony pociskiem będzie często ginął. Jednym z rozwiązań, oprócz balansu szybkości pocisków, może być dodanie graczowi punktów zdrowia. Zapewni to graczowi nieco bezpieczeństwa na wypadek zostania trafionym pociskiem.

Gry na urządzeniach mobilnych, przez brak fizycznych przycisków często mają problemy z intuicyjnym sterowaniem. Jako że od gracza będzie wymagane precyzyjne poruszanie się, aby manewrować pomiędzy ścianami i pociskami przeciwników, to jeszcze trudniejsze będzie odpowiednie celowanie, zakładając, że gracz strzela w tym samym kierunku, w którym się porusza. Należy w takim razie dokonać pewnych usprawnień, aby ułatwić rozgrywkę graczom, którzy będą mieli z tym problemy. W związku z czym, w grze powinna się znaleźć opcja pozwalająca na automatyczne celowanie w przeciwników, niezależnie od kierunku ruchu gracza.

Atrakcyjność zwiedzania powinna być zapewniona już na etapie generowania różnorodnych poziomów. Poczucie nowości z każdą kolejną próbą na pewno poprawia atrakcyjność podróżowania między komnatami. Jednak nic nie stoi na przeszkodzie, aby jeszcze bardziej ją uatrakcyjnić. Dlatego co każdy poziom będzie pojawiać się poziom z *silnym przeciwnikiem*, który będzie na swój sposób unikalny od zwykłych przeciwników - znacznie trudniejszy do pokonania, z wyjątkowymi mechanikami go wyróżniającymi. Na potrzeby prototypu silny przeciwnik będzie się cechował bardzo dużą liczbą punktów zdrowia oraz strzelaniem pociskami w kilku kierunkach jednocześnie.

Nie można również zapomnieć o konieczności zapewnienia graczowi możliwości pauzowania rozgrywki w dowolnym momencie.

1.3. SPECYFIKACJA WYMAGAŃ

Wycinki opisane w poprzednich podrozdziałach (1.2, 1.1) podsumowano i opisano w postaci dwóch tabel. W tabeli 1.1 sprecyzowano wymagania niefunkcjonalne na cały projekt. W tabeli 1.2 sprecyzowano wymagania funkcjonalne z podziałem na kategorie, wyróżniono m.in.: poziom, gracza i przeciwników.

Tabela 1.1: Wymagania niefunkcjonalne.

KATEGORIA	WYMAGANIA
Projekt	• Gra przeznaczona na system Android. • Styl graficzny: 2D, widok z góry. • Realizacja w silniku Unity.

Tabela 1.2: Wymagania funkcjonalne pogrupowane wg kategorii.

KATEGORIA	WYMAGANIA
Poziom	• Generowany proceduralnie. • Możliwość przejścia każdego labiryntu. • Możliwość przejścia każdej komnaty. • Możliwość gry w nieskończoność.
Interfejs	• Rozpoczęcie gry. • Pauzowanie rozgrywki w dowolnym momencie. • Zmiana ustawień celowania.
Gracz	• Poruszanie się na urządzeniach mobilnych. • Strzelanie w kierunku obrotu (zależnie od opcji). • Strzelanie w kierunku przeciwnika (zależnie od opcji). • Interakcja z otoczeniem (kolizje). • Posiada punkty zdrowia.
Przeciwnicy	• Pojawiają się na wolnych polach. • Poruszają się. • Strzelają w kierunku gracza. • Interakcja z otoczeniem (kolizje). • Posiadają punkty zdrowia.
Pocisk	• Interakcja z przeciwnikiem. • Interakcja z graczem. • Interakcja z przeszkodami terenu.
Skrzynia	• Interakcja z pociskami.

2. ANALIZA WYMAGAŃ

W tym rozdziale zostanie przeprowadzona analiza wymagań dotycząca sposobu wykorzystania proceduralnego generowania i wyboru algorytmów go dotyczących. Następnie zostanie przeprowadzona analiza pozwalająca wyszczególnić widoki w realizowanej grze na podstawie zasad rozgrywki opisanych w rozdziale 1.

2.1. GENEROWANIE POZIOMU

W rozdziale 1 określono strukturę poziomu. Ma się on składać z labiryntu, gdzie każde jego pole będzie tzw. komnatą, czyli większym obszarem terenu. Całe generowanie poziomu należy w takim razie podzielić na osobno: generowanie labiryntu i generowanie komnaty. Warto w tym momencie podkreślić, że gra ma być zrealizowana w dwóch wymiarach, gdzie widok jest z lotu ptaka. Oznacza to, że generowanie należy rozpatrywać pod kątem generowania elementów na siatce dwuwymiarowej.

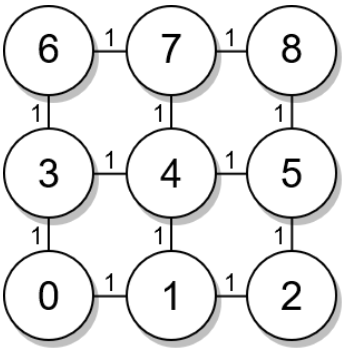
2.1.1. Generowanie labiryntu

Labirynty można sklasyfikować rozróżniając ich cechy na podstawie kilku kategorii [15], m.in.: wymiaru, hiper-wymiaru, topologii, tzw. *teselacji* (ang. *tesselation*), routing, tekstura. Najważniejszymi cechami dla tego projektu są:

- Wymiar, który został wcześniej zdefiniowany jako *dwuwymiarowy*.
- Teselacja, która powinna być *ortogonalna*, co oznacza, że siatka będzie prostokątem złożonym z pól.
- Routing. Najistotniejsza cecha ze wszystkich. Powinna być *doskonała*. Oznacza to, że labirynt nie będzie posiadał żadnych zapętlonych ścieżek i żadnych niedostępnych obszarów, czyli między dwoma dowolnie wybranymi punktami zawsze będzie istniała jedna ścieżka.

Na labirynt można spojrzeć jak na graf ze zbiorem wierzchołków i połączeń. W takim przypadku, wcześniej przytoczone cechy są możliwe do spełniania dzięki wykorzystaniu algorytmu do znajdowania minimalnego drzewa rozpinającego. Taki algorytm można wtedy zastosować na grafie opisanym na podstawie siatki dwuwymiarowej (przykład na rys. 2.1, gdzie okręgi to wierzchołki, a linie między nimi to połączenia z wagą odległości). Jednym z takich algorytmów jest algorytm Prima. Uproszczony algorytm Prima [5] zakłada tą samą odległość między wszystkimi połączonymi wierzchołkami. W zasadzie całkowicie ją pomija

i punkty wybiera losowo. Pozwala to na losowe wygenerowanie labiryntu, czyli losowe określenie połączeń między wszystkimi punktami na siatce. Spełnienie cech doskonałego labiryntu, czyni algorytm Prima właściwym wyborem dla tego etapu generowania poziomu.



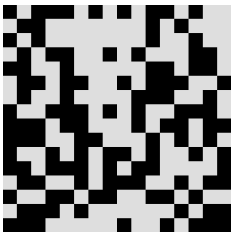
Rys. 2.1: Siatka dwuwymiarowa opisana w postaci grafu (opr. wł.)

2.1.2. Generowanie komnaty

Mając wygenerowany labirynt, można przejść do generowania poszczególnych obszarów terenu stanowiących pojedyncze pola labiryntu. Jest wiele referencji do generowania trójwymiarowych map terenu [16], które zawsze opierają się na wykorzystaniu szumu, najczęściej *szumu Perlina*, który został dobrze opisany w książce Daniela Shiffmana *The Nature of Code* [19]. Generowanie mapy terenu w dwóch wymiarach również jest możliwe wykorzystanie podobnych mechanizmów. Jednak należy dokonać pewnych uproszczeń wynikających z mniejszej liczby wymiarów. Przede wszystkim, w postaci dwuwymiarowej nie ma wysokości, więc teren należy rozróżnić w inny sposób. W tym przypadku będzie to po prostu określenie: teren (ściana) lub podłoże, czyli rozróżnienie binarne: 0 lub 1.

Mapa szumu

Tak jak w trójwymiarowym generowaniu terenu, na początek należy wygenerować mapę szumu. Jednak nie ma potrzeby jej komplikowania dedykowanymi algorytmami, jak np. szum Perlina. Wystarczy, że mapa szumu będzie maksymalnie uproszczona, tj. każde pole otrzyma losową, niezależną od innych pól wartość, która później będzie identyfikowana jako rodzaj terenu. Tak jak wcześniej wspomniano, teren rozróżniono jedynie dwoma wartościami. Stąd wygenerowane mapy szumu powinny wyglądać podobnie do przykładowej z rysunku 2.2, gdzie czarne i szare pola oznaczają dwa różne elementy terenu. Tak przygotowana mapa jest najważniejszym elementem pseudolosowym całego generowania komnat, dzięki niej będzie można dokonać kolejnych przekształceń, które już nie muszą być losowe (choć nie jest to wykluczone), a i tak pozwoli to na uzyskanie unikalnych komnat.



Rys. 2.2: Przykładowa mapa szumu dla siatki 16x16 (opr. wł.)

Automat komórkowy

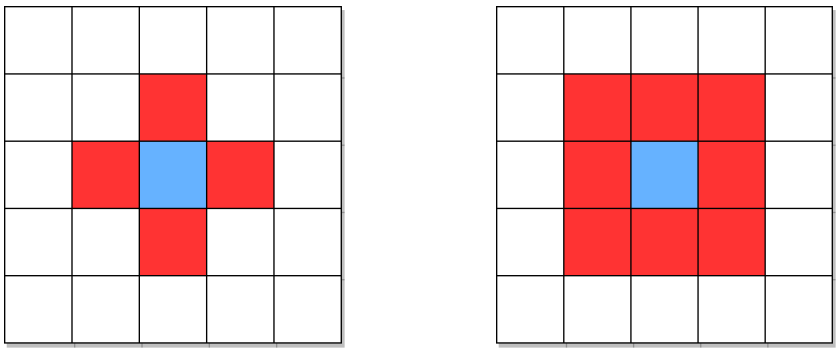
Zakładając, że wcześniej przedstawiona mapa szumu stanowi teren, nie będzie możliwości, aby gracz mógł komfortowo się po nim poruszać. Przede wszystkim sama mapa nie zapewnia przechodniości między dowolnymi punktami tego samego typu. Ziarnistość takiej mapy jest zbyt wysoka, można w niej wyróżnić bardzo dużo skupisk punktów tego samego typu.

W celu pozbycia się tak dużej ziarnistości i uzyskania efektu *wygładzenia*, które spowoduje uzyskanie mniejszej liczby większych skupisk obszarów tego samego typu, należy dokonać pewnych przekształceń. Idealnym rozwiązaniem wydaje się zastosowanie *automatu komórkowego*, któremu została poświęcona cała książka: Joela Schiffa *Cellular Automata: A Discrete View of the World* [17]. Automat komórkowy został też bardzo dobrze wyjaśniony w *The Nature of Code* Daniela Shiffmana [19].

W automacie komórkowym każda komórka (pole) na siatce posiada swój stan. Jednocześnie każda komórka jest automatem, który analizuje stan swój i stany swoich kilku najbliższych sąsiadów. Po takiej analizie następuje przekształcenie swojego stanu na inny z puli możliwych stanów, zgodnie z zasadami wspólnymi dla wszystkich automatów (tu: komórek, pól). Wszystkie przekształcenia dla danej ewolucji są realizowane jednocześnie - synchronicznie dla danej ewolucji. Oznacza to, że już zmienione komórki pokazują się swoim sąsiadom jakby jeszcze nie były zmienione dopóty, dopóki nie nastąpi zakończenie danej ewolucji. Po tym mogą nastąpić kolejne ewolucje na nowych stanach komórek, ale na tych samych zasadach przekształceń.

Podejście synchroniczne jest najczęstszym podejściem w implementacji tego algorytmu, jednak istnieją wyjątki. Wyróżnia się też podejścia stochastyczne i asynchroniczne, gdzie asynchroniczne polega na faktycznej zmianie stanu komórek (i pokazaniu nowego stanu sąsiadom) dla każdej komórki po kolei, w określonej kolejności. Stosowanie takiego podejścia będzie produkować mapy, które będą posiadać wizualnie podobne do siebie cechy w zależności od kolejności przechodzenia między komórkami. Podejście stochastyczne oznacza po prostu wykorzystanie losowości w swoich przekształceniach dotyczących zmiany stanów, co może być stosowane zarówno z podejściem synchronicznym i asynchronicznym.

Stanem początkowym dla całego automatu komórkowego będzie wcześniej wygenerowana mapa szumu. Wspomniano, że przekształcenia są dokonywane na podstawie stanów sąsiedztwa danych komórek. Istotne jest, w jaki sposób to sąsiedztwo zostanie dobrane. Tutaj wyróżnia się dwa najczęstsze sposoby: *sąsiedztwo von Neumanna* oraz *sąsiedztwo Moore'a*. Sąsiedztwem von Neumanna są cztery najbliższe komórki otaczające daną komórkę ze stron: z góry, z prawej, z dołu, z lewej. W takim przypadku analizuje się stany tych komórek. Sąsiedztwem Moore'a jest osiem najbliższych komórek otaczających daną komórkę, są to te same pola co w przypadku sąsiedztwa von Neumanna, ale z dodaniem komórek sąsiadujących na skos. Czyli są to komórki ze stron: góra-lewo, góra, góra-prawo, prawo, dół-prawo, dół, dól-lewo, lewo. Oba sposoby zostały przedstawione na rysunku 2.3, gdzie komórka niebieska to automat, dla którego analizowane jest sąsiedztwo, a komórki czerwone to wykryte sąsiedztwo. W celu uzyskania określonych efektów można oczywiście modyfikować sposoby doboru sąsiedztwa.



(a) Sąsiedztwo von Neumanna (b) Sąsiedztwo Moore'a

Rys. 2.3: Porównanie sąsiedztw w automacie komórkowym (opr. wł.)

Dla sąsiedztwa również istotnym elementem są wartości brzegowe. Czyli to jak należy analizować sąsiedztwo w sytuacji, gdy wychodzi ono poza siatkę mapy. W takim przypadku można dokonać pewnych założeń. Przykładowo, że takie komórki mają zawsze określony stan i automaty będą zakładać, że właśnie z takimi komórkami sąsiadują. W zależności od implementacji można też takich komórek po prostu nie uwzględniać przy przekształceniach.

Przy dobraniu odpowiednich przekształceń i wykonaniu kilku ewolucji automatu komórkowego, z mapy szumu, powinna powstać wygładzona mapa terenu, która będzie się cechować występowaniem większych skupisk terenu tego samego typu. Przykład przedstawiono na rysunku 2.4, gdzie kolory czarny i szary oznaczają różne elementy terenu.

Rys. 2.4: Przykładowa mapa po zastosowaniu automatu komórkowego dla siatki 32x32 (opr. wł.)

Algorytm przeszukiwania wszerz

Na wcześniej przedstawionej mapie (rys. 2.4) możliwa byłaby komfortowa rozgrywka. Gracz mógłby się poruszać po większych obszarach. Jednak nie zostałoby spełnione wymaganie o przechodniości każdej komnaty. W celu zapobiegnięcia temu problemowi, należy dobrać algorytm lub algorytmy, które pozwolą na połączenie takich komnat. Właściwym rozwiązaniem będzie podzielenie tego zadania na mniejsze części: wykrycie niepołączonych obszarów. Następnie ich połączenie.

Wykrycie niepołączonych obszarów można uzyskać stosując algorytm przeszukiwania wszerz [3] (ang. *breadth-first search*, BFS). Algorytm ten polega na przeszukaniu grafu i znalezieniu wszystkich punktów możliwych do odwiedzenia z zadanego punktu - korzenia. BFS działa w ten sposób, że rozpoczyna przeszukiwanie od swojego korzenia i odwiedza kolejne połączone wierzchołki tym samym powodując ich wykrycie przynależności do danego obszaru. Całość jest oparta o kolejkę, a nie o stos, jak w przypadku podobnych algorytmów, które mogą się opierać na stosie: *Flood Fill*[11], *Przeszukiwanie wgłąb*[8]. Czyni to go wystarczająco odpornym na przepełnienie, co znajdzie swoje zastosowanie w przypadku generowania większych terenów map. Najważniejsze, że pozwoli on na wykrycie połączonych obszarów dowolnego typu terenu.

Algorytm Prima

W momencie kiedy będą już wykryte niepołączone obszary terenu, należy doprowadzić do ich połączenia. Jednak zanim to nastąpi należy takie połączenia znaleźć. Tutaj z pomocą ponownie przychodzi opisywany przy generowaniu labiryntu (podrozdział: 2.1.1) algorytm Prima. Wykryte obszary mogą zostać opisane w postaci grafu, więc ponowne wykorzystanie tego algorytmu będzie właściwe. Jednak to wykorzystanie algorytmu Prima będzie się nieco różniło od poprzedniego, warto przypomnieć, że wcześniej został przedstawiony *uproszczony* algorytm Prima. W tym przypadku nie można założyć, że odległości między wierzchołkami grafu będą zawsze takie same, bądź pominięte. Na szczęście sam algorytm Prima jako algorytm do znajdowania minimalnego drzewa rozpinającego uwzględnia istnienie różnych odległości między wierzchołkami.

Po wykorzystaniu tego algorytmu powinny powstać definicje połączeń między obszarami, w taki sposób, aby możliwe było przejście z każdego obszaru, do innego dowolnie wybranego.

Algorytm Bresenhama

Z już sprecyzowanymi definicjami połączeń, możliwe będzie wykorzystanie kolejnego algorytmu, aby te połączenia faktycznie wykonać. Tutaj pomocny okaże się *algorytm Bresenhama* [36], który w swoich założeniach pozwala na tworzenie linii między dwoma punktami na siatce dwuwymiarowej. Jest on głównie wykorzystywany w grafice komputerowej, właśnie do rysowania linii. Jednak nic nie stoi na przeszkodzie, aby wykorzystać go również jako element procesu generowania mapy.

Wykorzystanie wszystkich dotychczas opisanych algorytmów, powinno pozwolić na wygenerowanie komnaty, która będzie spełniać wymagania przechodniości - z każdego punktu będzie możliwe przejście do dowolnego innego. Przykład jaki powinien być możliwy do uzyskania został przedstawiony na rysunku 2.5, gdzie czarne pola oznaczają teren nieprzechodni, a szare pola to podłoże, możliwe do przejścia przez gracza.

Rys. 2.5: Przykładowa mapa po zastosowaniu wszystkich algorytmów na siatce 32x32 (opr. wł.)

Zamknięcie labiryntu i połączenia komnat

Po wygenerowaniu pojedynczej komnaty należy ją dopasować do istniejącego labiryntu. Komnata jako pole labiryntu musi umożliwiać przejście do innych sąsiadujących komnat. Dlatego we wcześniej opisanych algorytmach należy uwzględnić, aby zabezpieczyć możliwość dojścia do granic komnaty tak, aby umożliwiało to przejście do sąsiadującej komnaty.

Przy rozgrywce, naturalne wydaje się, aby nie było możliwości wyjścia poza mapę. Może to spowodować powstanie dziwnych, nieoczekiwanych błędów w rozgrywce, a w niektórych przypadkach nawet kompletne zawieszenie aplikacji. Należy więc zabezpieczyć taką ewentualność. W tym przypadku najprostszym wydaje się zamknięcie całego poziomu,

czyli otoczenie go nieprzechodnim terenem. Otoczenie jednak całego poziomu pojedynczym prostokątem stanowiącym ścianę nie będzie jednak właściwe, bo możliwe będzie wystąpienie sytuacji, że powstanie dziura między komnatami. Najwłaściwsze wydaje się zapewnienie, aby każda komnata była osobno otoczona przez teren. Trzeba o to zadbać w procesie jej generowania.

Skrzynie

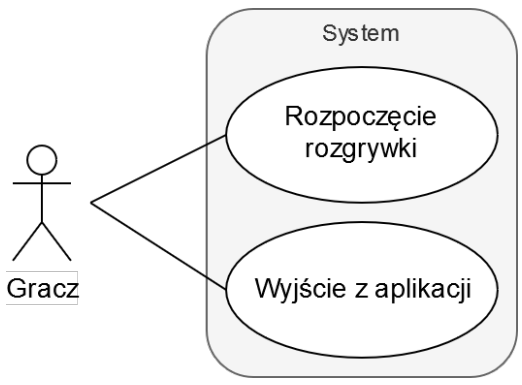
W wymaganiach wspomniane zostało występowanie na poziomie skrzyń, jako obiektów możliwych do zniszczenia. Jako, że są to obiekty możliwe do zniszczenia, to pod nimi musi się znajdować teren. W takim razie nie ma sensu uwzględniać skrzyń w samym procesie generowania terenu, a można je dodać już po wygenerowaniu poziomu.

2.2. INTERFEJS UŻYTKOWNIKA

Informacje zawarte w wymaganiach pozwalają na wywnioskowanie jakie widoki będą użytkownikowi (graczowi) potrzebne i jakie minimalne funkcje powinny one oferować. Stąd wysnuto, że właściwy będzie podział na następujące widoki: widok menu głównego, widok rozgrywki, widok pauzy, widok przegranej. Dodatkowo zdecydowano się przedstawić diagram przypadków użycia (zgodny z zasadami UML [9]) dla każdego widoku osobno, co wydaje się czytelniejsze i łatwiejsze w dalszej analizie oraz realizacji tego projektu.

2.2.1. Widok menu głównego

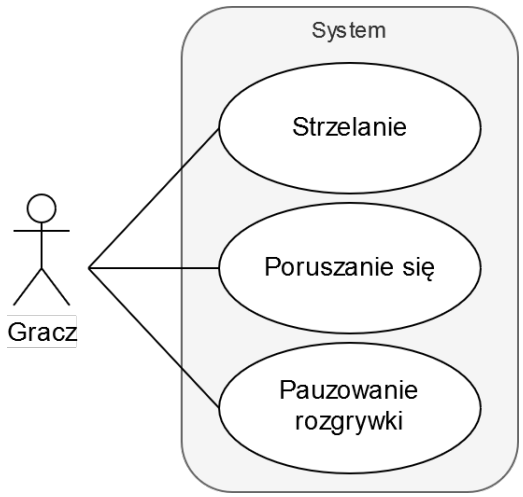
Przygotowanie menu głównego wydaje się być oczywistą decyzją w każdej grze komputerowej. W tym przypadku nie będzie inaczej. W menu głównym z pewnością musi się znaleźć funkcjonalność rozpoczęcia rozgrywki. Jako, że gra jest przeznaczona na system Android, gdzie nie ma aplikacji okienkowych, to gra musi oferować również możliwość jej wyłączenia. Zawarcie takiej opcji w menu głównym wydaje się właściwe w przypadku choćby nieumyślnego uruchomienia aplikacji. Diagram przypadków użycia dotyczący jedynie widoku menu głównego przedstawiono na rysunku 2.6.



Rys. 2.6: Diagram przypadków użycia dotyczący widoku menu głównego (opr. wł.)

2.2.2. Widok rozgrywki

W widoku rozgrywki muszą graczowi zostać wszystkie funkcjonalności wynikające z zasad rozgrywki. Decyzją gracza będzie skorzystanie z ekranu swojego urządzenia mobilnego, aby strzelać pociskami i się poruszać. Gracz w tym widoku spędzi większość czasu, a bloki czasowe w jakich będzie na tym ekranie przebywał będą znacznie dłuższe niż te dotyczące pozostałych widoków. Konieczne jest więc zapewnienie graczowi możliwości pauzowania rozgrywki, co zostało już określone w wymaganiach (rozdział: 1). Diagram przypadków użycia dotyczący jedynie widoku rozgrywki przedstawiono na rysunku 2.7.

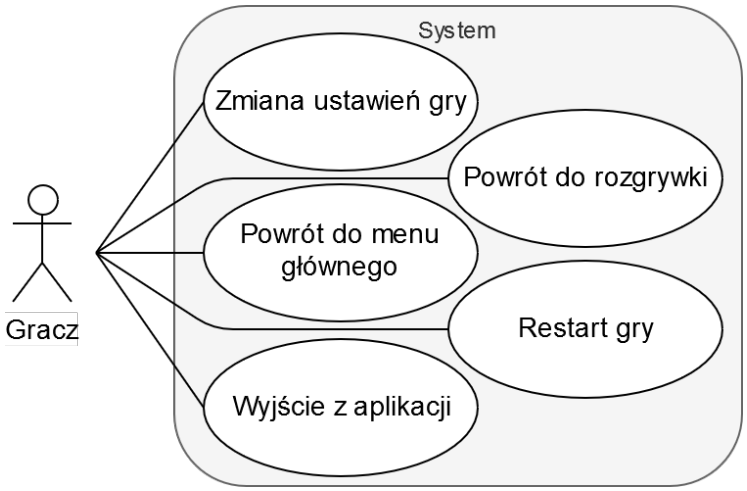


Rys. 2.7: Diagram przypadków użycia dotyczący widoku rozgrywki (opr. wł.)

2.2.3. Widok pauzy

Najważniejszą opcją, która musi być oferowana graczowi w widoku pauzy jest powrót do rozgrywki. W wymaganiach (rozdział: 1) wspomniano o możliwości zmiany ustawień gry dotyczących celowania. Właściwym wydaje się, aby ta opcja znalazła się w widoku pauzy, aby gracz podczas dynamicznej rozgrywki przypadkowo tych ustawień nie zmienił. Występowanie ustawień w menu pauzy w przyszłości pozwoli na łatwiejsze dodawanie kolejnych opcji ustawień, gdyż na ekranie rozgrywki trzeba być ostrożnym przy projektowaniu ekranów, aby nie zasłonić samej rozgrywki. Natomiast opcje w menu głównym mogą być krzywdzące dla gracza, gdy ten zapomni ich zmienić przed rozgrywką.

Oprócz pauzowania i zmiany ustawień gry. Gra powinna pozwolić na powrót do menu głównego. Dodatkowo, aby oszczędzić graczowi czas, zostaną również dodane opcje typowe dla menu głównego: rozpoczęcie rozgrywki w postaci restartu gry oraz wyjście z aplikacji. Diagram przypadków użycia dotyczący widoku pauzy przedstawiono na rysunku 2.8.

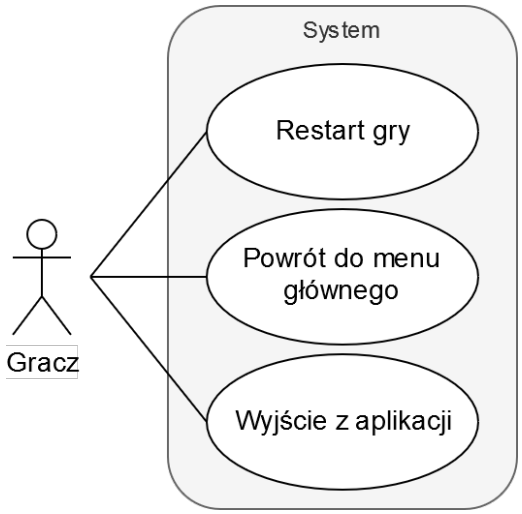


Rys. 2.8: Diagram przypadków użycia dotyczący widoku pauzy (opr. wł.)

2.2.4. Widok przegranej

Zamiast istnienia widoku przegranej, można by zrobić po prostu powrót do menu głównego. Jednak gracz może przeoczyć powód swojej porażki (utrata wszystkich punktów zdrowia), dlatego właściwym wydaje się przygotowanie również widoku przegranej, który będzie się różnił od menu głównego i pozwoli na uniknięcie frustracji gracza.

Widok przegranej powinno umożliwić ponowne rozpoczęcie rozgrywki, powrót do menu głównego oraz wyjście z aplikacji. Diagram przypadków użycia dotyczący jedynie widoku przegranej przedstawiono na rysunku 2.9.



Rys. 2.9: Diagram przypadków użycia dotyczący widoku porażki (opr. wł.)

3. WYKORZYSTANE TECHNOLOGIE

W tym rozdziale zostaną przedstawione najważniejsze technologie i narzędzia, z których korzystano przy realizacji pracy.

3.1. SILNIK GIER UNITY

Jako silnik gry został wykorzystany, zgodnie z tematem pracy, silnik *Unity* wyprodukowany przez firmę *Unity Technologies*. Jest on najpopularniejszym środowiskiem do tworzenia gier, które jest wykorzystywane zarówno przez wielkie studia, jak i pojedyncze osoby. Silnik ten jest bardzo rozbudowany i pozwala na tworzenie wieloplatformowych gier, zarówno trójwymiarowych, jak i dwuwymiarowych. Oczywiście umożliwienie tworzenia gier to tylko główne założenie tego silnika. Jednak można w nim tworzyć również aplikacje o innym przeznaczeniu, np. symulacje czy aplikacje multimedialne. Wspiera on także wykorzystanie rozszerzonej i wirtualnej rzeczywistości.

Silnik *Unity* jest przede wszystkim darmowy (do 100 000 USD przychodu rocznie), co wraz z jego wszechstronnością i przyjaznym interfejsem użytkownika pozwoliło na zbudowanie ogromnej społeczności. W związku z tym bardzo łatwo jest znaleźć liczne poradniki na dowolny temat związany z silnikiem. A w przypadku pojawienia się problemów, na odpowiedź na forum społeczności nie trzeba długo czekać. Oprócz tego, sama firma *Unity Technologies* posiada swoje znaczące zaplecze poradników zarówno dla początkujących, jak i doświadczonych twórców. Nieocenioną zaletą jest również kompletna i rozbudowana dokumentacja silnika i jego API [23].

Konstrukcja projektów realizowanych w tym silniku opiera się przede wszystkim na scenach, które składają się z obiektów, a te z kolei składają się z komponentów. Komponenty są to specjalne skrypty rozszerzające klasę *MonoBehaviour* [26]. Takie skrypty mogą być tworzone za pomocą języka programowania *C Sharp*. *Unity* zapewnia wiele podstawowych, standardowych komponentów [28], które mogą zostać wykorzystane bez konieczności znajomości języka programowania.

Każdy umieszczony na dowolnej scenie obiekt, posiadający swój określony zestaw komponentów wraz z ich parametrami, może zostać zapisany jako zasób projektu. Tworzy on wtedy tzw. prefabrykat [29], dzięki czemu można go dynamicznie instancjonować na scenie w czasie jej trwania. Umożliwia on również edycję wszystkich istniejących na dowolnych scenach obiektów pochodzących z tego prefabrykatu, poprzez edycję tego

prefabrykatu w panelu zasobów.

Edytor *Unity* można dodatkowo rozszerzać poprzez liczne dodatki, udostępnione poprzez specjalny, wbudowany sklep, tzw. *Asset Store*, w którym wiele pozycji jest darmowych. Oprócz rozszerzeń funkcjonalności edytora można tam również nabyć gotowe komponenty i inne rodzaje zasobów pomocne przy realizacji własnych produkcji.

Wszechstronność silnika jest zarazem jego największą wadą. Istnienie tak wielu elementów składowych silnika, m.in. wbudowany silnik graficzny czy silnik fizyki, powoduje, że projekty tworzone za pomocą *Unity* są dużych rozmiarów. Jest to szczególnie kluczowe w przypadku urządzeń mobilnych, gdzie pamięć masowa jest wciąż mocno ograniczona. Pusty projekt, przeznaczony na system Android, z zapewnieniem wsparcia dla wszystkich dostępnych w edytorze architektur procesorów dla tego systemu, po instalacji na urządzeniu mobilnym posiada rozmiar około 100 MB.

Na szczęście *Unity Technologies* już pracuje nad rozwiązaniem tego problemu. Firma zamierza drastycznie zmniejszyć rozmiary podstawowych bibliotek, które po kompresji pozwolą na zmniejszenie ich rozmiaru do nawet 73 KB w przypadku projektów realizowanych jako gry webowe [33]. Można więc oczekiwać, że rozmiary projektów przeznaczonych na urządzenia mobilne również znacznie zmaleją.

Oprócz silnika *Unity* dostępne są także konkurencyjne silniki gier, np. *Godot* [13] czy *Cocos2d-x* [21], które również umożliwiają tworzenie gier 2D na platformy mobilne, gdzie ich projekty po instalacji są znacznie mniejszych rozmiarów niż te przygotowane w *Unity*. Ich kluczową wadą jest gorsze wsparcie wynikające z mniejszej popularności, co mogłoby znacznie utrudnić realizację projektu w przypadku napotkania większych problemów.

Przy realizacji pracy korzystano z wersji silnika *Unity: 2018.2.11f1*.

3.2. JĘZYK PROGRAMOWANIA C SHARP

Przygotowywanie własnych komponentów w silniku *Unity* wymaga wykorzystania języka *C Sharp*. Niegdyś możliwe było programowanie w specjalnym języku *UnityScript* (bazującym na *JavaScript*), jednak oficjalnie wraz z wersją *Unity 2017.2* zakończyło się jego wsparcie [7], dlatego odradza się korzystania z tego języka programowania.

C Sharp jest wieloparadygmatowym językiem programowania wysokiego poziomu, który jest kompilowany do języka CIL [20], który z kolei jest wykonywany w środowisku uruchomieniowym obejmowanym przez platformę *.NET*. Zarówno *C Sharp*, jak i *.NET*

należą do firmy *Microsoft*.

W wersji silnika *Unity: 2018.2.11f1* możliwe jest korzystanie z języka *C Sharp* w wersji *6.0*. Wraz z wersją *Unity 2018.3* pojawiło się wsparcie dla wersji języka *C Sharp 7.3* [32].

3.3. ŚRODOWISKO PROGRAMISTYCZNE MICROSOFT VISUAL STUDIO

Jest to zintegrowane środowisko programistyczne, którego autorem jest firma *Microsoft*. Środowisko to umożliwia, m.in. tworzenie oprogramowania w języku *C Sharp*, co pozwoli na przygotowanie skryptów możliwych do wykorzystania w silniku *Unity*.

Microsoft Visual Studio bezpośrednio łączy się z edytorem Unity i pozwala na dołączenie do niego debuggera, co pozwala na pełne śledzenie wykonywanego kodu wraz z monitorowaniem pamięci. Jest to kluczowa funkcjonalność pozwalająca na znaczne przyspieszenie procesu szukania i naprawy błędów.

Alternatywami są środowiska: udostępniany wraz z *Unity*, mało rozbudowany - *MonoDevelop* i poważny konkurent dla środowiska *Visual Studio* w zakresie pracy z edytorem *Unity* - *Rider* od JetBrains.

Przy realizacji pracy korzystano z wersji *Microsoft Visual Studio 2017 Enterprise*.

3.4. SYSTEM KONTROLI WERSJI GIT I NARZĘDZIA POWIĄZANE

System kontroli wersji pozwala śledzić zmiany w projekcie na przestrzeni czasu, jak i łatwo tworzyć kopie zapasowe. Głównym celem stosowania systemów kontroli wersji jest oczywiście kontrolowanie i śledzenie zmian przeprowadzanych w projekcie przez wiele osób jednocześnie. Jednak w przypadku tej pracy, gdzie projekt wykonuje tylko jedna osoba, pozwoli on przede wszystkim na wygodne tworzenie kopii zapasowych.

Jako system kontroli wersji zostanie wykorzystany najpopularniejszy z nich - *Git*. Jako serwis hostingowy dla repozytorium zostanie wykorzystany *Bitbucket* firmy *Atlassian*. Wszystko dodatkowo będzie zarządzane poprzez przez aplikację kliencką *Sourcetree* firmy *Atlassian*.

3.5. APLIKACJA UNITY REMOTE

Aplikacja ta jest dostępna w sklepie *Google Play*, dostępnym na urządzeniach z systemem Android. Aplikacja umożliwia połączenie telefonu z edytorem Unity, co pozwala na testowanie gry na urządzeniu mobilnym bez konieczności budowania projektu przed

każdą próbą testowania pojedynczych funkcjonalności. Znacznie przyspiesza i ułatwia to testy aplikacji, w szczególności we wczesnej fazie tworzenia gry. Jednak w późniejszych fazach, gdzie pracuje się nad wykończeniem całej aplikacji i testy przeprowadza się bardziej kompleksowo, ta aplikacja nie powinna być wykorzystywana. Jest to spowodowane tym, że sama aplikacja współpracuje z edytorem Unity na zasadzie przesyłania obrazu z edytora do telefonu oraz przesyłania akcji wejścia z urządzenia mobilnego do edytora. Oznacza to, że wszystkie obliczenia w dalszym ciągu realizowane są po stronie edytora, a nie na urządzeniu mobilnym. Z tego powodu, mimo wszystko, zaleca się przeprowadzanie analizy i testów wydajności poprzez standardowe zbudowanie projektu i zainstalowanie aplikacji na urządzeniu mobilnym. Oprócz wydajności może się okazać, że w dopiero zainstalowanej na takim urządzeniu aplikacji, niektóre funkcje przestaną działać. Dlatego tak ważne jest przeprowadzanie testów końcowych w klasyczny sposób. Nie zmienia to faktu, że aplikacja Unity Remote drastycznie usprawnia cały proces produkcji gier mobilnych realizowanych w silniku Unity.

4. KONCEPT REALIZACJI

Zdefiniowane wymagania (rozdział: 1) i opis rozgrywki, oprócz określenia widoków (jak w Analizie wymagań 2), pozwalają również na opisanie części tzw. Dokumentu Projektu Gry (ang. *Game Design Document*, GDD) [2]. GDD w swoich założeniach ma umożliwić pracę nad projektem dla wszystkich członków zespołu mających różne zadania, od programistów, przez artystów, aż po osoby zajmujące się marketingiem. Taki dokument powinien stanowić kompletną dokumentację gry i powinien być równolegle rozwijany od samego początku aż do samego końca etapu produkcji gry. Nie ma ustalonego szablonu GDD, który sprawdziłby się dla każdego tytułu. Każdy projekt jest unikalny i każde GDD również jest unikalne.

Ten rozdział będzie pełnił funkcję swoistego GDD, oczywiście nie będzie to dokument kompletny, bo sama dokumentacja pochłonęłaby zbyt dużo uwagi. Mimo to wszystkie przedstawione w tym rozdziale spojrzenia na projekt gry będą mogły stanowić element GDD.

4.1. OPIS ELEMENTÓW GRY

Podrozdział opisuje elementy gry, które będą występować w świecie gry. Opis ograniczy się do elementów rozgrywki, które będą posiadać swoją reprezentację graficzną w świecie gry, czyli takie które będą mogły zostać dostrzeżona przez gracza. Przy czym podkreśla się, że nie dotyczące one elementów interfejsu graficznego, który zostanie omówiony w innym podrozdziale.

4.1.1. Rodzaje elementów

W tabelach: 4.1, 4.2, 4.3 zostały opisane elementy rozgrywki zauważalne z perspektywy gracza. W tabeli 4.1 są to ogólne rodzaje elementów, natomiast w tabelach 4.2, 4.3 został dokonany kolejny podział dla elementów: *Przeciwnik* i *Pocisk* na mniejsze kategorie.

Tabela 4.1: Rodzaje elementów rozgrywki.

NAZWA	OPIS
Gracz	Postać sterowana przez gracza. Może się poruszać ze średnią szybkością i strzelać szybkimi pociskami. Posiada zdrowie.
Przeciwnik	Postać sterowana przez komputer. Porusza się w losowym kierunku. Strzela pociskami w kierunku gracza, gdy tylko go zobaczy. Może zostać opisany cechami takimi jak: szybkość, zdrowie, rodzaj pocisku.
Ściana	Element terenu. Koliduje ze wszystkimi postaciami oraz z pociskami. Nie można jej zniszczyć w żaden sposób.
Podłoże	Element terenu. Podłoże pełni głównie funkcję estetyczną. Nie posiada ono kolizji i nie można z nim wejść w żadną interakcję, dzięki czemu postacie mogą się po nim poruszać.
Skrzynia	Nieruchomy obiekt, który zostaje umieszczony na podłożu w losowych miejscach w procesie generowania poziomu. Gracz może się za nią chować, aby uniknąć pocisków przeciwników. Może zostać zniszczona przez pociski gracza.
Pocisk	Ruchomy obiekt. Może zostać wystrzelony przez gracza i przeciwników. W momencie wystrzelenia zostaje mu nadany stały kierunek lotu. Znika po natrafieniu na dowolną przeszkodę jednocześnie zadając jej obrażenia. Może zostać opisany cechami takimi jak: szybkość, obrażenia, czas przeładowania.
Portal	Nieruchomy obiekt, który znajduje się w komnacie finałowej. Gdy gracz w niego wejdzie, nastąpi przejście do kolejnego poziomu. Na każdym poziomie istnieje tylko jeden portal.

Tabela 4.2: Rodzaje przeciwników.

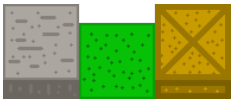
NAZWA	OPIS
Niebieski przeciwnik	Standardowy przeciwnik. Cechuje go duża szybkość i mała liczba punktów zdrowia. Strzela wolnymi pociskami.
Różowy przeciwnik	Standardowy przeciwnik. Cechuje go mała szybkość i duża liczba punktów zdrowia. Strzela wolnymi pociskami.
Silny przeciwnik	Unikalny przeciwnik. Cechuje go średnia szybkość, ogromna liczba punktów zdrowia i unikalna umiejętność - strzelanie w kilku kierunkach jednocześnie. Strzela wolnymi pociskami. Jako jedyny przeciwnik posiada reprezentację graficzną aktualnego zdrowia w postaci paska na interfejsie graficznym.

Tabela 4.3: Rodzaje pocisków.

NAZWA	OPIS
Wolny pocisk	Cechuje go mała szybkość lotu, małe obrażenia i średni czas przeładowania.
Szybki pocisk	Cechuje go duża szybkość lotu, małe obrażenia i średni czas przeładowania.

4.1.2. Reprezentacja graficzna elementów

Każdy z elementów przedstawionych w tabeli 4.1 musi posiadać swoją reprezentację graficzną. W wymaganiach (rozdział 1.1) wspomniano o inspiracji stylem *retro*. W dużym uproszczeniu może oznaczać on po prostu grafiki o niskiej rozdzielczości z prostymi kształtami. W programie *Gimp* autor pracy przygotował bardzo proste grafiki ściany, podłoża i skrzyni (rys. 4.1). W celu pozyskania kolejnych grafik, zostanie wykorzystany sklep z zasobami oferowany przez Unity (tzw. *Asset Store*). Do projektu zaimportowane zostaną niektóre grafiki oferowane w tym sklepie pod hasłem *Sunnyland*. Wszystkie przedstawiono na rys. 4.2, gdzie: (a, b) to animowana postać lisa w celu reprezentacji postaci gracza, (b) wiśnie w celu reprezentacji pocisku. Postać gracza będzie animowana.



Rys. 4.1: Grafiki ściany, podłoża i skrzyni (opr. wł.)

(a) Animacja postoju gracza

(b) Animacja ruchu gracza

(c) Pocisk

Rys. 4.2: Grafiki zaimportowane z darmowego sklepu Unity. (źródło: *Unity Asset Store - Sunny Land* [1])

Dla wszystkich pozostałych elementów rozgrywki zostaną wykorzystane podstawowe możliwości samego silnika Unity pozwalające na kreowanie warstwy graficznej. Rozumie się przez to prymitywne, dwuwymiarowe obiekty takie jak wielokąty czy koła o jednolitym kolorze. Dla przeciwników będą to wielokąty - każdy rodzaj przeciwnika będzie innym wielokątem z jednolitym kolorem. Niebieski przeciwnik - niebieski czworokąt, różowy przeciwnik - różowy sześciokąt, silny przeciwnik - pomarańczowy ośmiokąt. Portal będzie reprezentować prymitywna grafika jasnoniebieskiego koła.

4.1.3. Efekty wizualne

Oprócz tego, że postać gracza będzie animowana, to przydałyby się jeszcze inne efekty wizualne nadające warstwie graficznej trochę dynamizmu. Silnik Unity posiada wbudowany system cząsteczek [31]. Zostanie on wykorzystany przy następujących elementach:

- Pocisk, który w trakcie lotu będzie zostawiał za sobą cząsteczki.
- Portal, na którym będzie zapętlona, nieskończona animacja cząsteczek.
- Skrzynia, która po zniszczeniu jednorazowo *wyrzuci* wiele cząsteczek.

4.2. HISTORYJKI UŻYTKOWNIKA

Posiadając wyszczególnione elementy gry i widoki, można z wysoką dokładnością opisać historyjki użytkownika. Nie będzie to klasyczny opis w postaci: *Jako _ chcę _, aby _* [22]. Forma zostanie zmieniona, na subiektywnie czytelniejszą i łatwiejszą w analizie, aby mogła ona być jednym z kluczowych elementów tworzonego GDD. Takie historyjki użytkownika będą pomocne choćby dla programistów, bo pozwolą one na rozbicie zasad gry na drobne elementy. Jako że tych elementów jest bardzo dużo, zostaną one rozbite i podzielone według wcześniej określonych widoków (podrozdział: 2.2).

Dodatkowo do historyjek zostały dodane nowe opcje w menu pauzy: Nietykalność, która będzie blokować utratę punktów zdrowia gracza. Przydatne do testowania. Oraz opcja:

Automatyczne celowanie przez skrzynie, gdyż nie zostało sprecyzowane czy skrzynie, które można zniszczyć powinny być uwzględniane jako teren przy celowaniu, czy też nie.

Tabela 4.4: Historyjki użytkownika dla widoku menu głównego.

WARUNEK	WEJŚCIE	EFEKT
-	Kliknięto przycisk *Rozpocznij*	Następuje przejście do ekranu rozgrywki.
-	Kliknięto przycisk *Wyjście*	Następuje wyjście z aplikacji.

Tabela 4.5: Historyjki użytkownika dla widoku pauzy.

WARUNEK	WEJŚCIE	EFEKT
-	Kliknięto przycisk *Pauzy* **LUB** kliknięto przycisk *Kontynuuj*	Panel pauzy znika z ekranu, a gra jest kontynuowana.
-	Kliknięto przycisk *Powrót do menu*	Następuje przejście do ekranu menu głównego.
-	Kliknięto przycisk *Restart*	Następuje ponowne załadowanie sceny rozgrywki, wraz z wygenerowaniem nowego poziomu i zresetowaniem wartości statystyk.
-	Kliknięto przycisk *Wyjście*	Następuje wyjście z aplikacji.
-	Aktywowano / dezaktywowano opcję *Nietykalność*	Aktywacja / dezaktywacja trybu: Nietykalność.
-	Aktywowano / dezaktywowano opcję *Automatyczne celowanie*	Aktywacja / dezaktywacja trybu: Automatyczne celowanie.
-	Aktywowano / dezaktywowano opcję *Celowanie przez skrzynie*	Aktywacja / dezaktywacja trybu: Celowanie przez skrzynie.

Tabela 4.6: Historyjki użytkownika dla widoku przegranej.

WARUNEK	WEJŚCIE	EFEKT
-	Kliknięto przycisk *Powrót do menu*	Następuje przejście do ekranu menu głównego.
-	Kliknięto przycisk *Restart*	Następuje ponowne załadowanie sceny rozgrywki, wraz z wygenerowaniem nowego poziomu i zresetowaniem wartości statystyk.
-	Kliknięto przycisk *Wyjście*	Następuje wyjście z aplikacji.

Tabela 4.7: Historyjki użytkownika dla widoku rozgrywki część 1 z 2.

WARUNEK	WEJŚCIE	EFEKT
Poprawne załadowanie sceny	-	Następuje zainicjowanie rozgrywki. Poziom jest automatycznie generowany, a gracz pojawia się na wolnym polu w komnacie startowej. Statystyki są inicjowane na wartości bazowe.
-	Kliknięto przycisk *Pauzy*	Gra się zatrzymuje i pojawia się panel pauzy.
-	Wychylono drążek od pozycji początkowej	Postać gracza porusza się w kierunku, w którym drążek jest wychylony względem pozycji początkowej drążka.
Strzał jest odnowiony **ORAZ** istnieje przeciwnik, który względem gracza nie jest za przeszkodą **ORAZ** gracz ma tryb automatycznego celowania	Przycisk strzału jest dotykany	Postać gracza wykonuje strzał w kierunku najbliższego widocznego przeciwnika.
Strzał jest odnowiony **ORAZ** (**NIE** istnieje przeciwnik, który względem gracza nie jest za przeszkodą **LUB** gracz **NIE** ma trybu automatycznego celowania)	Przycisk strzału jest dotykany	Postać gracza wykonuje strzał w kierunku ruchu postaci gracza.
Co kilka sekund **LUB** przeciwnik natrafił na przeszkodę terenu	-	Przeciwnik zmienia swój kierunek ruchu.
Przeciwnik został trafiony pociskiem pochodzącym od gracza	-	Przeciwnik traci punkty zdrowia. Pocisk znika.

Tabela 4.8: Historyjki użytkownika dla widoku rozgrywki część 2 z 2.

WARUNEK	WEJŚCIE	EFEKT
Gracz został trafiony pociskiem pochodzącym od przeciwnika	-	Gracz traci punkty zdrowia. Pocisk znika.
Pocisk trafił na przeszkodę terenu	-	Pocisk znika.
Pocisk pochodzący od gracza trafił na skrzynię	-	Skrzynia traci punkty zdrowia. Pocisk znika.
Punkty zdrowia przeciwnika spadły do 0	-	Zostaje dodany punkt do statystyk. Przeciwnik znika.
Punkty zdrowia skrzyni spadły do 0	-	Zostaje dodany punkt do statystyk. Skrzynia znika.
Punkty zdrowia gracza spadły do 0	-	Przegrana. Pojawia się ekran przegranej.
Na poziomie pojawia się *silny przeciwnik*	-	Pojawia się pasek zdrowia *silnego przeciwnika*.
Punkty zdrowia *silnego przeciwnika* spadły do 0	-	Znika pasek zdrowia *silnego przeciwnika*. Na poziomie pojawia się *portal*.
Postać gracza weszła w *portal*	-	Następuje przejście do kolejnego poziomu – ponowne wygenerowanie mapy i przeniesienie gracza na pole startowe.
Postać gracza weszła do komnaty po raz pierwszy	-	Przejścia do innych komnat zostają zablokowane. W komnacie pojawiają się przeciwnicy.
W komnacie zniknął ostatni przeciwnik	-	Przejścia do innych komnat zostają odblokowane.
Postać gracza trafiła na przeszkodę	-	Następuje kolizja.
Przeciwnik trafił na przeszkodę	-	Następuje kolizja.
Zawsze	-	Kamera jest wycentrowana na postaci gracza.

4.3. INTERFEJS UŻYTKOWNIKA

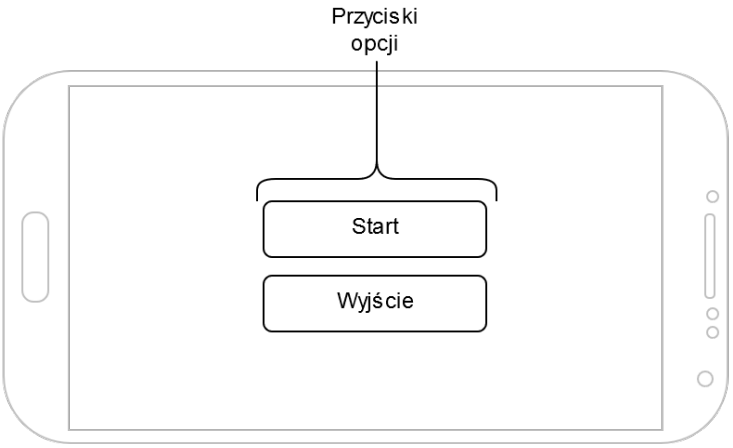
4.3.1. Projekty ekranów

Na podstawie analizy interfejsu użytkownika (rozdział 2.2) i historyjek użytkownika (rozdział 4.2) możliwe jest zaprojektowanie projektów ekranów, dla wcześniej wyróż-

nionych widoków: menu, rozgrywki, pauzy, przegranej. Oprócz spełnienia wymagań, do niektórych ekranów zostaną dodane nowe elementy uatrakcyjniające rozgrywkę.

Ekran menu

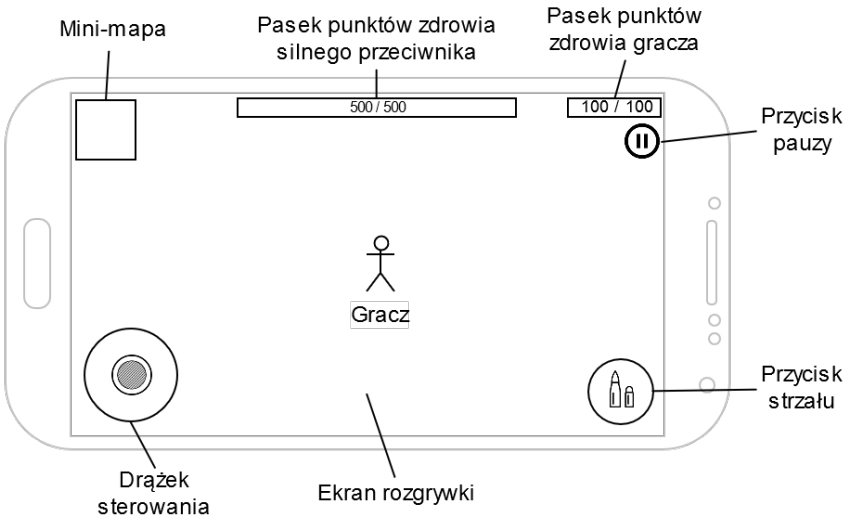
Na ekranie menu wyróżnia się dwa główne przyciski: Start, Wyjście. Nie zauważa się żadnych nowych elementów względem tego co zostało opisane w rozdziałach 2.2, 4.2. Projekt tego ekranu przedstawia rysunek 4.3.



Rys. 4.3: Projekt ekranu menu (opr. wł.)

Ekran rozgrywki

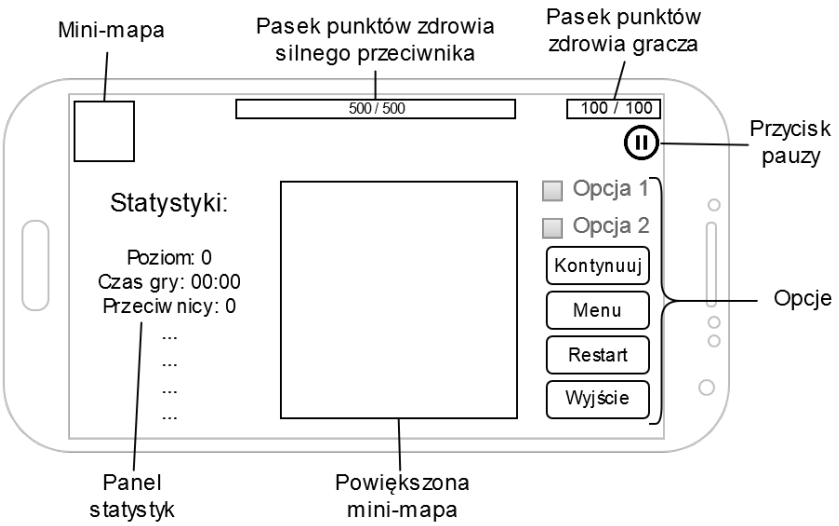
Ekran rozgrywki musi spełnić wymogi odnośnie sterowania postacią. Są to: drążek sterowania i przycisk strzału. Przycisk pauzy pozwoli na pauzowanie rozgrywki. Pojawiły się też paski punktów zdrowia, które były już wspomniane w historyjkach użytkownika (4.2). Oprócz tego pojawiła się również mini-mapa, która będzie stanowić graficzną reprezentację struktury labiryntu. Projekt tego ekranu przedstawia rysunek 4.4.



Rys. 4.4: Projekt ekranu rozgrywki (opr. wł.)

Ekran pauzy

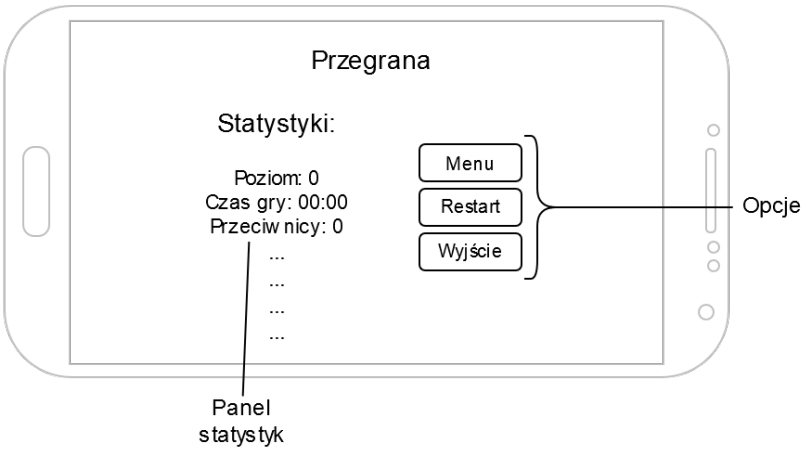
Na ekranie pauzy pojawią się elementy spełniające to co zostało opisane w rozdziałach 2.2 i 4.2, czyli przyciski: Kontynuuj, Powrót do menu, Restart, Wyjście oraz przycisk Pauzy. Oprócz tego, na ekranie pauzy zostaną zachowane niektóre elementy interfejsu ekranu rozgrywki, są to: mini-mapa i paski zdrowia. Dodatkowo na ekranie pojawi się panel statystyk, na którym przedstawione będą niektóre wartości mierzone podczas gry: poziom, czas gry, pokonani przeciwnicy, zniszczone skrzynie. Nowością jest też pojawienie się powiększonej mini-mapy, która względem zwykłej mini-mapy będzie różnić się tym, że oprócz pokazania struktury całego labiryntu, pokaże również teren wszystkich komnat. Projekt tego ekranu przedstawia rysunek 4.5.



Rys. 4.5: Projekt ekranu pauzy (opr. wł.)

Ekran przegranej

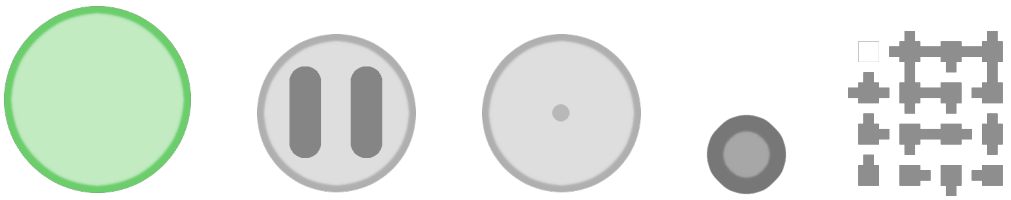
Na ekranie przegranej, oprócz przycisków zgodnych z rozdziałami 2.2 i 4.2 pojawi się również panel statystyk. Taki sam jak na ekranie pauzy. Projekt tego ekranu przedstawia rysunek 4.6.



Rys. 4.6: Projekt ekranu przegranej (opr. wł.)

4.3.2. Reprezentacja graficzna interfejsu użytkownika

W celu reprezentacji graficznej kontrolek na interfejsie użytkownika, w programie *Gimp* zostało przygotowanych kilka grafik. Przedstawiono je na rysunku 4.7, gdzie: (a) to przycisk strzału, na który zostanie nałożona grafika pocisku, (b) to przycisk pauzy, (c) punkt startowy drążka, z którego będzie analizowane wychylenie, (d) drążek, (e) siatka grafik, które będą się składać na mini-mapę.



(a) (b) (c) (d) (e)

Rys. 4.7: Grafiki kontrolek interfejsu (opr. wł.)

Pozostałe elementy interfejsu zostaną przedstawione przy wykorzystaniu podstawowych, wbudowanych grafik silnika Unity. Dotyczy to wszystkich przycisków, tekstów, pól wyboru i pasków zdrowia.

5. OPIS IMPLEMENTACJI

W rozdziale zostaną przedstawione opisy najważniejszych części implementacji. Są to: generowanie poziomów, elementy rozgrywki i interfejs użytkownika. Następnie zostaną przedstawione problemy i ich rozwiązania, które zlokalizowano w procesie testowania.

5.1. IMPLEMENTACJA GENEROWANIA POZIOMU

W celu generowania poziomu przygotowano skrypt zarządzający całym generowaniem: *LevelGenerator.cs*. Skrypt ten dziedziczy po klasie: *MonoBehaviour*, dzięki czemu może zostać wykorzystany jako komponent jednego z obiektów na scenie gry. Tak też zrobiono, dołączono *LevelGenerator* do pustego obiektu sceny (takiego bez żadnych komponentów z wyjątkiem wymaganego: *Transform*). Dodanie skryptu do obiektu pozwoli na automatyczne inicjowanie jego działania przy starcie sceny gry, odbywa się to w metodach pochodzących od *MonoBehaviour*, które są wykonywane kolejno [27]: *Awake()*, *OnEnable() Start()*.

5.1.1. Generowanie labiryntu

W celu generowania labiryntu przygotowano prosty interfejs: *IMazeGenerator* (listing 5.1), który musi być implementowany przez każdy generator labiryntu. Istnienie takiego interfejsu pozwoli w przyszłości na tworzenie wielu skryptów z implementacją różnych algorytmów generowania poziomu. Dzięki temu będzie możliwe przygotowanie puli takich obiektów i generowanie labiryntu będzie następować losowym algorytmem.

```
public interface IMazeGenerator
{
        Maze GenerateMaze();
}
```

Listing 5.1: Kod interfejsu *IMazeGenerator* do generowania labiryntu

Obiekt zwracany przez metodę *GenerateMaze()* jest strukturą opisującą sam labirynt. Posiada przede wszystkim zbiór połączeń między poszczególnymi polami. Co zostało przedstawione na listingu 5.2.

Przygotowano również pierwszą implementację dla interfejsu *IMazeGenerator*. Jest nią algorytm Prima, który został opisany w *Analizie Wymagań* (roz. 2). Przy implementacji tego algorytmu korzystano z opracowania *Maze Generation: Prim's Algorithm* [4]. Uproszczony

algorytm Prima, czyli taki, który pomija wagi połączeń, a zamiast tego wybiera punkty losowo z dostępnych połączeń. Struktura działania takiego algorytmu została przedstawiona na listingu 5.3. Adekwatnie do niej zaimplementowano cały algorytm.

```
public enum Direction
{
        Up = 0,
        Right = 1,
        Down = 2,
        Left = 3
}

public struct Maze
{
        public int[,] Grid { get; private set; }
        public Direction[,][] Directions { get; private set; }

        public Maze(int[,] grid, Direction[,][] directions)
        {
                Grid = grid;
                Directions = directions;
        }
}
```

Listing 5.2: Kod struktury *Maze* i enumeratora *Direction*

```
1. Wybierz losowy punkt P na siatce mapy.
2. Dodaj punkt P do zbioru punktów odwiedzonych.
3. Dodaj nieodwiedzonych sąsiadów punktu P do zbioru punktów
 ↪  przylegających.
4. Dopóki( zbiór punktów przylegających nie jest pusty ):
    5. Wybierz losowy punkt X ze zbioru punktów przylegających.
    6. Wybierz losowy punkt Y ze zbioru punktów odwiedzonych.
    7. Dodaj połączenie między punktami X i Y do zbioru połączeń.
    8. Usuń punkt X ze zbioru punktów przylegających.
    9. Dodaj punkt X do zbioru punktów odwiedzonych.
    10. Dodaj nieodwiedzonych sąsiadów punktu X do zbioru punktów
     ↪  przylegających.
11. Zwróć zbiór połączeń.
```

Listing 5.3: Struktura działania uproszczonego algorytmu Prima

5.1.2. Generowanie komnaty

W celu implementacji generowania komnat (w skryptach nazwanych: *Chunk*) została przygotowana klasa abstrakcyjna *ChunkMapGenerator*, przedstawiona na listingu 5.4. Przy wyborze klasy abstrakcyjnej względem możliwości wyboru interfejsu, kierowano się tym, że obiekt generatora będzie utworzony tylko raz i z tego samego obiektu korzystać będzie *LevelGenerator* do generowania kolejnych komnat. Natomiast każda komnata w generatorze musi być opisana tą samą szerokością przejść, które muszą umożliwiać przejścia do innych komnat. Innym rozwiązaniem mogłoby być jednak zastosowanie interfejsu i za każdym razem przekazywanie szerokości przejść przy wywoływaniu metody *GenerateChunkMap()*. Pozostałe parametry jak szerokość, wysokość i kierunki połączeń będą unikalne dla każdej wygenerowanej komnaty dlatego są przekazywane metodą.

Przygotowano pierwszy skrypt generatora komnaty dziedziczący po *ChunkMapGenerator*, skrypt nazwano *CellularAutomata* z tego względu, że będzie to najistotniejszy algorytm w tym skrypcie. Działanie skryptu będzie polegać na przygotowaniu mapy szumu, dokonaniu na niej przekształceń kolejnymi algorytmami i zwróceniu tak wygenerowanej komnaty. Klasa *CellularAutomata* charakteryzuje się tym, że może zostać jej przekazana wartość ziarna (ang. *seed*), która nadpisze ziarno statycznej klasy *Random* dostępnej w pakiecie *UnityEngine* i tym samym spowoduje, że dany poziom będzie zawsze wygenerowany w ten sam sposób.

```
public abstract class ChunkMapGenerator
{
        public int WidthOfCorridor { get; }
        public abstract CellType[,] GenerateChunkMap(int width, int height,
        ↪  Direction[] directions);

        public ChunkMapGenerator(int widthOfCorridor)
        {
                WidthOfCorridor = widthOfCorridor;
        }
}
```

Listing 5.4: Kod klasy abstrakcyjnej *ChunkMapGenerator*

Szum

Na początku, w skrypcie *CellularAutomata* należy wygenerować mapę szumu. Implementacja szumu ograniczyła się do wygenerowania bardzo prostej mapy szumu, która może przyjmować wartość 0 lub 1, które oznaczają kolejno ścianę (ang. *Wall*) lub podłoże (ang. *Ground*). Dodatkowo dodany został parametr wypełnienia (nazwaną: *FillPercentage*), który będzie rozkładać szanse wylosowania poszczególnych wartości. Kod metody generowania takiego szumu został przedstawiony w listingu 5.5.

```
public enum CellType
{
    Wall = 0,
    Ground = 1
}

private CellType[,] GenerateNoise(int width, int height)
{
    CellType[,] map = new CellType[width, height];
    for (int x = 0; x < map.GetLength(0); x++)
    {
        for (int y = 0; y < map.GetLength(1); y++)
        {
            map[x, y] = Random.value < FillPercentage ? CellType.Wall :
            ↪  CellType.Ground; //Przypisanie do pola losowego typu
            ↪  terenu.
        }
    }
        return map;
}
```

Listing 5.5: Kod metody *GenerateNoise()* do generowania szumu

Automat komórkowy

Zastosowanie automatu komórkowego to najważniejszy etap w procesie generowania komnaty. W *Analizie wymagań* (rozdział 2) wspomniano, że w najczęściej stosuje się sąsiedztwo *von Neumanna* lub *Moore'a*. W tej implementacji zdecydowano się na ten drugi. Oznacza to, że sąsiadami dla każdej komórki automatu będzie osiem pól dookoła niej i to one będą analizowane w przypadku chęci zmiany stanu automatu komórkowego. Zdecydowano się na wybór podejścia asynchronicznego (rozdział 2.1.2 co oznacza, że przekształcenia stanów komórek będą wykonywane po kolei i nowe stany od razu będą widoczne dla sąsiednich komórek bez czekania na ukończenie pełnej ewolucji. Przy implementacji inspirowano się rozwiązaniem przedstawionym w oficjalnym poradniku od *Unity Technologies* [12]. Implementację przedstawiono na listingu 5.6.

Próbowano również podejścia synchronicznego, jednak ten niezależnie od liczby ewolucji, pozostawia chropowatość na obrzeżach terenu danego typu. W porównaniu do tego, podejście asynchroniczne zapewnia dużo lepsze wygładzenie terenu mapy. Ciekawym sposobem mogłoby być również pomieszanie obu podejść i przykładowo dokonaniu kilku ewolucji podejściem synchronicznym, a potem kilku podejściem asynchronicznym. Jednak to nadaje się na całkowicie nową implementację generowania komnat, która nie zostanie poruszona w tej pracy.

```
private void SmoothMap(ref CellType[,] map)
{
    for (int x = 0; x < map.GetLength(0); x++)
    {
        for (int y = 0; y < map.GetLength(1); y++)
        {
            int neighbourWallCount = GetNeighbourWallCount(map, x, y);
            ↪ //Pobranie liczby sąsiadów ze stanem CellType.Wall
            if (neighbourWallCount > 4)
                map[x, y] = CellType.Wall;
            else if (neighbourWallCount < 4)
                map[x, y] = CellType.Ground;
        }
    }
}
```

Listing 5.6: Kod metody *SmoothMap()* będącej implementacją automatu komórkowego

Przeszukiwanie wszerz

Kolejnym algorytmem jaki należy zastosować, zgodnie z *Analizą wymagań* (rozdział 2). Jest algorytm przeszukiwania wszerz (BFS). Polega on na znalezieniu wszystkich pól dostępnych z określonego pola. Całość algorytmu BFS opiera się o kolejkę FIFO - czyli pobieranie elementów z kolejki będzie zgodne z chronologią ich wstawiania do kolejki. Implementację tego algorytmu przedstawiono na listingu 5.8. W celu wykrycia na mapie wszystkich pól o tym samym typie terenu, należy algorytm BFS wywołać jednorazowo na każdym odosobnionym obszarze. Aby tego dokonać należy przejść przez wszystkie pola mapy i po kolei sprawdzać czy posiada ono poszukiwany typ terenu. Jeśli tak to wywołać algorytm BFS, a znalezione pola oznaczyć jako odwiedzone w celu uniknięcia wywołania przeszukiwania wszerz ponownie dla tego samego obszaru. Implementację takiej metody przedstawiono na listingu 5.7.

```
private List<Zone> GetAllZones(CellType[,] map, CellType targetCellType)
{
    List<Zone> zones = new List<Zone>();
    bool[,] visitedMap = new bool[map.GetLength(0), map.GetLength(1)];

    //Przejście po wszystkich polach na mapie
    for (int x = 0; x < map.GetLength(0); x++)
    {
        for (int y = 0; y < map.GetLength(1); y++)
        {
            //Jeśli pole nie było odwiedzone i jest zgodne z poszukiwanym
            ↪  typem terenu, to zastosuj na nim algorytm BFS
            if (!visitedMap[x, y] && map[x, y] == targetCellType)
            {
                Zone newZone = GetZone(map, new Cell(x, y)); //Wywołanie
                ↪  algorytmu przeszukiwania wszerz (BFS)
                zones.Add(newZone);

                //Oznaczenie wszystkich ostatnio znalezionych pól jako
                ↪  odwiedzone
                foreach (Cell c in newZone.CellsIn)
                    visitedMap[c.X, c.Y] = true;
            }
        }
    }
    return zones; //Zwrócenie listy wykrytych obszarów
}
```

Listing 5.7: Kod metody *GetAllZones()* znajdującej wszystkie obszary danego typu terenu

```
private Zone GetZone(CellType[,] map, Cell cell)
{
    Zone zone = new Zone(); //'Zone' stanowi zbiór połączonych pól tego
    ↪ samego typu terenu.
    zone.CellsIn.Add(cell); //Dodanie pierwsze elementu do zbioru pól.
    CellType targetCellType = map[cell.X, cell.Y]; //Pobranie typu terenu,
    ↪ który należy przeszukać
    Queue<Cell> queue = new Queue<Cell>();
    Cell actualCell = cell;
    bool[,] visitedMap = new bool[map.GetLength(0), map.GetLength(1)];
    ↪ //Zbiór odwiedzonych pól.
    visitedMap[actualCell.X, actualCell.Y] = true;

    queue.Enqueue(actualCell); //Dodanie pierwszego pola, z którego nastąpi
    ↪ przeszukiwanie wszerz
    while (queue.Count > 0) //Dopóki kolejka nie jest pusta
    {
        actualCell = queue.Dequeue(); //Zabranie z kolejki pierwszego pola
        //Podwójna pętla przechodzi po 8 polach dookoła
        for (int x = actualCell.X - 1; x <= actualCell.X + 1; x++)
        {
            for (int y = actualCell.Y - 1; y <= actualCell.Y + 1; y++)
            {
                Cell loopCell = new Cell(x, y);
                //Ograniczenie sąsiednich pól z 8 do 4 kierunków oraz
                ↪ sprawdzenie czy pole nie wyszło poza mapę oraz czy nie
                ↪ było już odwiedzone
                if (IsInsideMap(map, loopCell) && (x == actualCell.X || y
                ↪ == actualCell.Y) && !visitedMap[x, y])
                {
                    visitedMap[x, y] = true;
                    if (map[x, y] == targetCellType)
                    {
                        queue.Enqueue(loopCell);
                        zone.CellsIn.Add(loopCell);
                    }
                }
            }
        }
    }
    return zone; //Zwrócenie wykrytego obszaru pól
}
```

Listing 5.8: Kod metody *GetZone()* będącej implementacją algorytmu przeszukiwania wszerz

Algorytm Prima

Mając wykryte obszary należy dokonać ich analizy i przygotować połączenia w postaci minimalnego drzewa rozpinającego. Zgodnie z *Analizą wymagań* (rozdział 2), zostanie wykorzystany algorytm Prima. W odróżnieniu od generowania labiryntu, tym razem algorytm nie będzie posiadał uproszczonej implementacji zakładającej te same koszty dla wszystkich połączeń. Koszty między obszarami zostały obliczone metodą, która dla wszystkich kombinacji par obszarów przeszukiwała wszystkie kombinacje par punktów stanowiących obrzeża danego obszaru (sąsiadujących z innym typem terenu) w celu wybrania najbliższej sobie pary. W taki sposób powstały definicje połączeń między obszarami na zasadzie: każdy z każdym. Posiadanie takiego opisu kosztów umożliwiło implementację algorytmu Prima, która ma za zadanie określić, które obszary mają zostać połączone. Działanie implementacji takiego algorytmu opisano w listingu 5.9.

```
1. Wybierz losowy obszar O ze zbioru obszarów.
2. Dodaj obszar O do zbioru obszarów połączonych.
4. Dopóki( liczba obszarów połączonych jest mniejsza niż liczba obszarów ):
    5. Wybierz obszar X o najmniejszej odległości od dowolnego obszaru
    ↪  znajdującego się w zbiorze obszarów połączonych.
    6. Wybierz obszar Y ze zbioru obszarów połączonych, dla którego obszar
    ↪  X miał najmniejszą odległość.
    7. Dodaj połączenie między obszarami X i Y do zbioru połączeń.
    9. Dodaj obszar X do zbioru obszarów połączonych.
11. Zwróć zbiór połączeń.
```

Listing 5.9: Struktura działania zaimplementowanego algorytmu Prima dla połączenia obszarów terenu

Algorytm Bresenhama

Posiadając definicje połączeń między obszarami, które jednocześnie posiadają definicję między którymi dokładnie punktami takie połączenie powinno nastąpić, można przejść do wykorzystania algorytmu Bresenhama. Algorytm ten polega po prostu na rysowaniu linii między dwoma polami na siatce dwuwymiarowej. W tym przypadku można to określić jako rysowanie polami konkretnego typu po siatce terenu. Należy zadbać, aby tak narysowana linia uwzględniała to, że punkty nie mogą być połączone po skosie - tylko takie połączenie wyklucza przechodniość między tymi polami. Przy implementacji inspirowano się fragmentem *Anti-aliased thick line* pochodzącym z opracowania Aloisa Zingla *The Beauty of Bresenham's Algorithm*: [35]. Został tam opisany algorytm do rysowania linii o zadanej szerokości, ale ze zmiennoprzecinkowymi wartościami komórek w celu wygładzenia linii. Mapa poziomu opiera się jednak tylko na dwóch wartościach 0 lub 1 (ściana lub podłoże), więc w implementacji takie wartości zmiennoprzecinkowe zaokrąglano.

Dodanie skrzyń

Po wygenerowaniu wszystkich komnat pozostało dodanie skrzyń. Ich tworzenie nie będzie się jednak zawierać w skrypcie *LevelGenerator*. Każda utworzona komnata posiada swój skrypt zarządzający nazwany *ChunkController*, przez który będzie mogła sama zarządzać tym co się na niej dzieje.

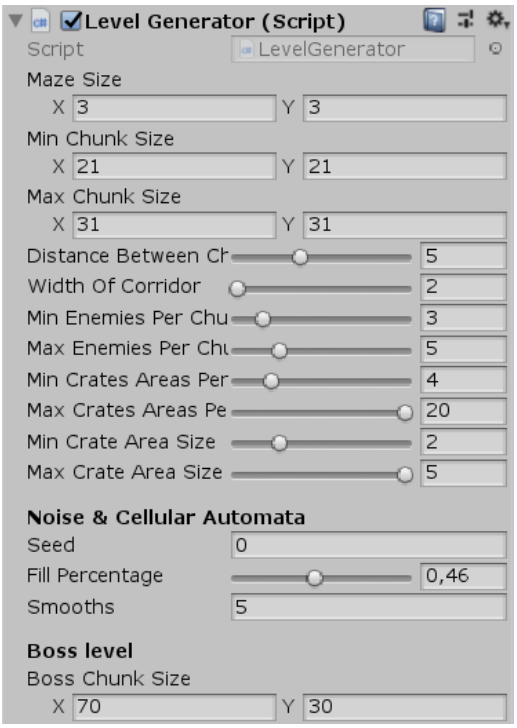
Założono, że generowanie na mapie skrzyń jest właśnie elementem, nad którym powinien mieć kontrolę skrypt zarządzający daną komnatą. Dlatego w momencie tworzenia *ChunkControllera*, będzie on otrzymywać od *LevelGeneratora* parametry opisujące to ile skrzyń powinien wygenerować.

Tworzenie skrzyń w *ChunkControllera* opiera się o to, że przechowuje on zbiór wolnych pól. W celu wygenerowania obszaru ze skrzyniami, wybiera losowy wolny punkt, a następnie wokół niego generuje obszar o zdefiniowanej wielkości, pomijając sąsiadujące ściany. Nie stanowi to zagrożenia dla przechodniości całej komnaty, gdyż gracz może takie skrzynia niszczyć.

5.1.3. Efekty implementacji generowania poziomu

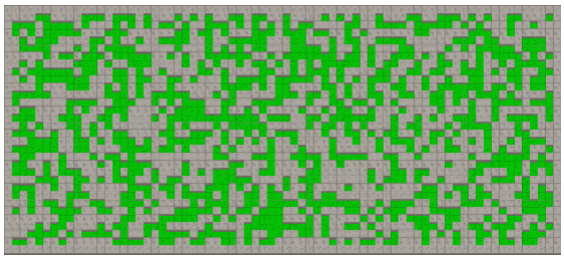
Po implementacji skryptu *LevelGeneratora* i skryptów pochodnych służących do generowania poziomu oraz implementacji *ChunkControllera*, który będzie otrzymywał parametry dotyczące generowania skrzyń i liczby przeciwników na nim przebywających, powstała struktura parametrów ustawianych w inspektorze komponentu *LevelGenerator*. Została ona przedstawiona na rysunku 5.1 i posiada m.in. parametry: rozmiar labiryntu, zakres rozmiaru komnat, rozmiary korytarzy (tuneli) łączących komnaty, zakres liczby przeciwników i skrzyń w każdej komnacie, rozmiar poziomu z silnym przeciwnikiem. Warto przypomnieć, że poziom z silnym przeciwnikiem pojawia się na zmianę ze standardowym poziomem. Gdzie standardowy poziom w przypadku jak na rysunku 5.1 będzie labiryntem

3x3, natomiast poziom z silnym przeciwnikiem labiryntem 1x1 - czyli będzie posiadał jedną, ogromną komnatę.

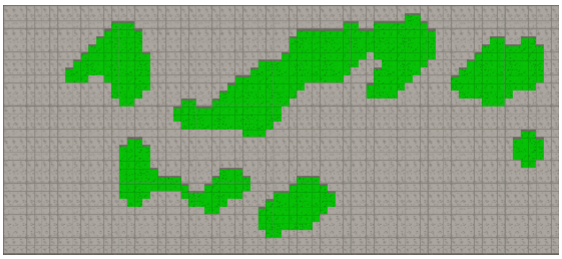


Rys. 5.1: Komponent skryptu *LevelGenerator* w inspektorze Unity (opr. wł.)

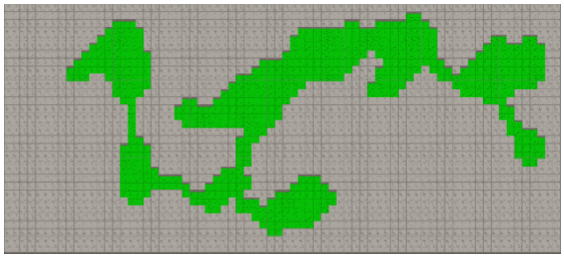
Cały proces generowania pojedynczej komnaty został graficznie przedstawiony na rysunku 5.2.



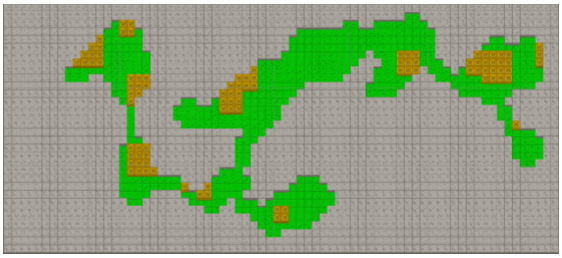
(a) Szum



(b) Automat komórkowy



(c) Połączenie obszarów

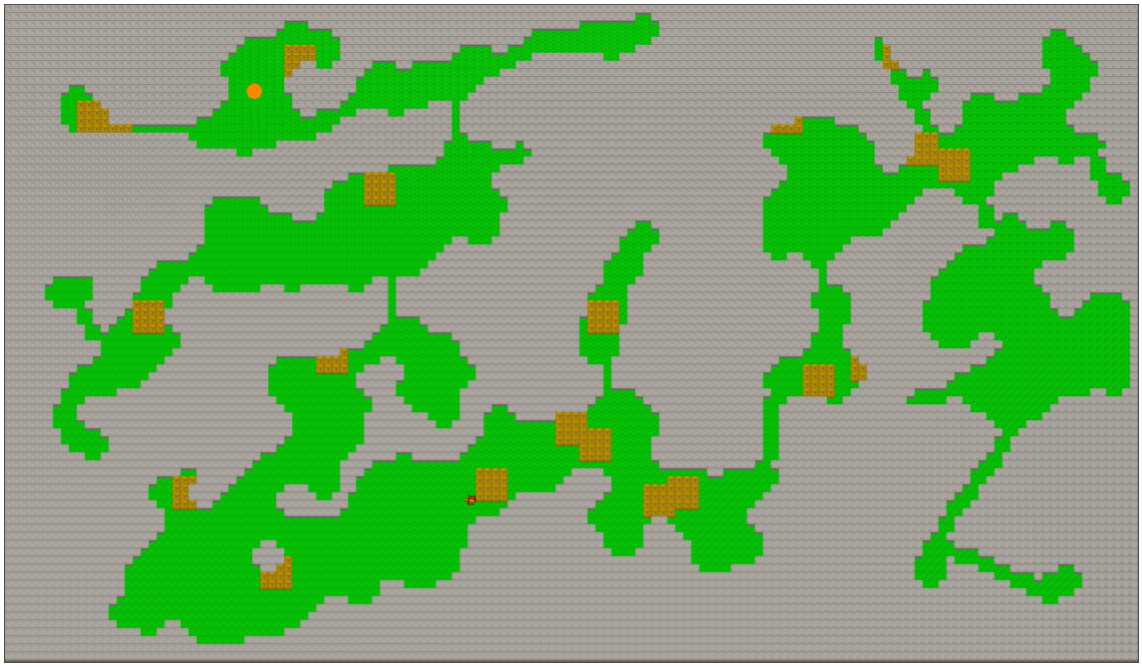


(d) Dodanie skrzyń

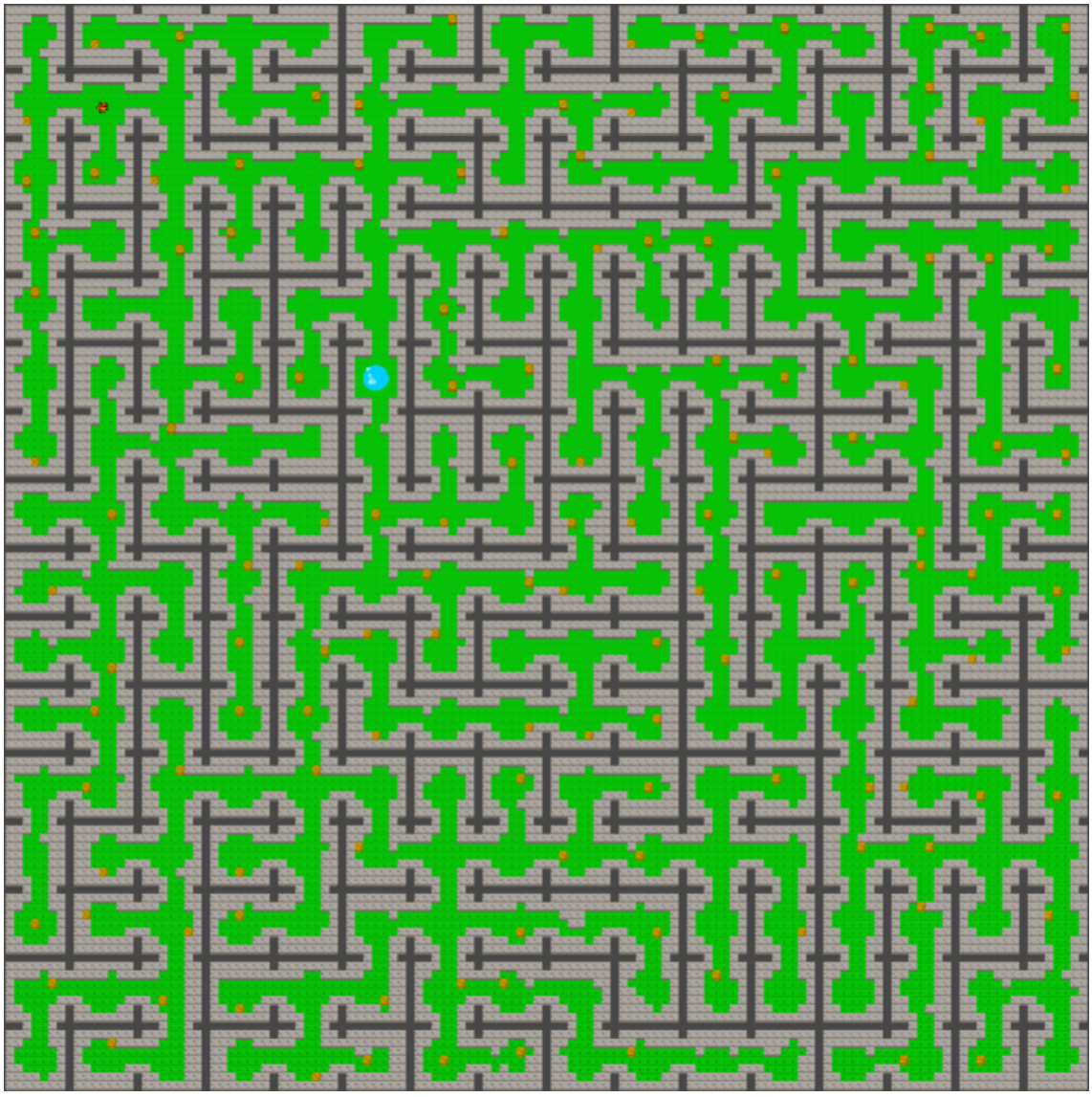
Rys. 5.2: Generowanie komnaty (opr. wł.)

Różnorodność ustawień

Zgodnie z parametrami generowania poziomów (rys. 5.1), możliwa jest bardzo łatwa zmiana charakterystyki gry. Zamiast standardowych poziomów z labiryntem 3x3 i średniej wielkości komnatami, można również generować zupełnie inne poziomy, co udowadnia jak duży jest potencjał tego projektu. Przykłady przedstawiono na rysunkach 5.3, 5.4. Z takimi poziomami można np. spowodować, że gracz będzie musiał skupić się na przejściu dużego labiryntu albo na dynamicznej walce z przeciwnikami na rozległych obszarach



Rys. 5.3: Wygenerowany poziom po zmianie parametrów - labirynt 1x1, komnata 140x80 (opr. wł.)



Rys. 5.4: Wygenerowany poziom po zmianie parametrów - labirynt 16x16, komnata 5x5 (opr. wł.)

5.2. IMPLEMENTACJA ELEMENTÓW ROZGRYWKI

5.2.1. Obiekt zarządzający grą

Bardzo ważnym elementem przy tworzeniu gier w silniku Unity jest zastosowanie jakiegoś sposobu zarządzania głównymi mechanizmami rozgrywki. Najczęściej jest to realizowane w postaci menedżerów. Tak również zostało zrobione w tym projekcie.

Do sceny rozgrywki dołączono pusty obiekt, dla którego przygotowano trzy skrypty menedżerów: *GameManager*, *GameUIManager*, *InputManager*.

GameManager jest najważniejszym menedżerem zapewniającym kontrolę nad większością elementów rozgrywki. Implementuje on klasę *MonoBehaviour* dzięki czemu może być komponentem dołączonym do obiektu. Mimo to znaczna część jego implementacji opiera się na zmiennych i metodach statycznych. Dodatkowo dodano do niego mechanizm typowy dla wzorca: *Singleton*, który ogranicza występowanie takiego menedżera do jednej instancji. Cała komunikacja między obiektami i skryptami w aplikacji opiera się o system zdarzeń, gdzie wszystkie są zdefiniowane w *GameManager*. Jeśli nastąpiło jakieś kluczowe wydarzenie dotyczące dowolnego obiektu, np. portal wykrył kolizję z graczem, to w takim przypadku portal wywołuje metodę zawartą w *GameManager*, aby przejść do kolejnego poziomu. Natomiast *GameManager* rozsyła informację o tym zdarzeniu do wszystkich subskrybentów danego typu zdarzenia. Oferowane przez niego zdarzenia to:

- *OnInitialPositionChanged* - subskrybowane przez *PlayerController*, informuje o tym, że zmienił się punkt startowy dla poziomu.
- *OnMapChanged* - subskrybowane przez *MiniMapController*, informuje o tym, że zmieniła się struktura labiryntu.
- *OnMazePositionChanged* - subskrybowane przez *MiniMapController*, informuje o tym, że gracz przeszedł do innej komnaty bądź korytarza.
- *OnGameLost* - subskrybowane przez *GameUIManager*, informuje o tym, że nastąpiła przegrana gracza.
- *OnPlayerHPChanged* - subskrybowane przez *GameUIManager*, informuje o tym, że zmieniła się liczba punktów zdrowia gracza.
- *OnBossHPChanged* - subskrybowane przez *GameUIManager*, informuje o tym, że zmieniła się liczba punktów zdrowia silnego przeciwnika.
- *OnLoading* - subskrybowane przez *GameUIManager*, informuje o tym, że nastąpiło rozpoczęcie bądź zakończenie ładowania poziomu.

Skorzystanie z systemu zdarzeń umożliwia bardzo łatwą rozbudowę projektu w przyszłości. Powoduje to uniknięcie bezpośredniej komunikacji między różnymi obiektami, dzięki czemu całość odbywa się zgodnie ze wzorcem *Obserwatora*.

GameManager posiada również odwołania do różnych zbiorów danych, m.in. do typów terenu i typów przeciwników. Te zbiory są specjalnymi klasami dziedziczącymi po klasie

ScriptableObject przeznaczonymi do przechowywania danych. Nie mogą zostać dodane do obiektu jako komponenty, ale mogą być tworzone instancje tych obiektów w edytorze Unity. Istnienie takich instancji umożliwia edycję danych każdej z nich i przypisywanie tych instancji do komponentów. Wykorzystanie takiego mechanizmu pozwala na odseparowanie danych istotnych dla rozgrywki. Dzięki temu możliwe jest przykładowo dodawanie kolejnych prefabrykatów [29] będących elementami terenu do zbioru typów terenu. Dzięki czemu przy generowaniu komnaty taki typ terenu może już być uwzględniony przy losowaniu prefabrykatu danego typu terenu. Obecnie wykorzystane zbiory danych oparte o *ScriptableObject*, to:

- *CellTypesData* - zawiera referencje do prefabrykatów elementów terenu z podziałem na typ (ściana, podłoże, obiekty możliwe do zniszczenia).
- *EnemyPrefabsData* - zawiera referencje do prefabrykatów przeciwników z podziałem na ich unikalność.
- *OtherPrefabsData* - zawiera referencje do prefabrykatów unikalnych obiektów, np. obiekt portalu.

5.2.2. Sterowanie postacią gracza

InputManager jest odpowiedzialny za umożliwienie graczowi sterowania podczas rozgrywki. Odpowiada on za zarządzanie elementami: poruszanie drążkiem, przycisk strzału i zmianę trybów w opcjach. Udostępnia informację o stanie tych elementów:

- Drążek - poziom wychylenia od wartości startowej.
- Przycisk strzału - czy przycisk jest aktualnie przytrzymywany.
- Tryby - aktualne stany dla trzech pól wyboru: Nietykalność, Automatycznie celowanie, Automatyczne celowanie przez skrzynie.

Dzięki tym informacjom *PlayerController* może zdecydować, w którą stronę powinien się poruszać i czy powinien wykonać strzał. Implementacja została pokazana na listingu 5.10, polega na nadaniu komponentowi *Rigidbody2D* prędkości w kierunku zgodnym z tym udostępnianym przez *InputManager*. Komponent *Rigidbody2D* jest pod kontrolą silnika fizycznego i pozwala na zarządzanie ruchem obiektów [30].

```
private void Move()
{
    _rb2d.velocity = new Vector2(_input.AxisHorizontal,
    ↪  _input.AxisVertical).normalized * _stats.MovementSpeed;
}
```

Listing 5.10: Metoda ruchu gracza w skrypcie *PlayerController*

Strzelanie pociskami przez gracza odbywa się poprzez wywołanie co klatkę metody *Shot()* w skrypcie *PlayerController()*. Implementację tej metody przedstawiono na listingu 5.11.

```
private void Shot()
{
    if (!_input.IsHoldingShot)  //Sprawdza czy przycisk jest trzymany
        return;

    if (_nextShotTime > Time.time) //Sprawdza czy strzał jest odnowiony
        return;

    if (_input.AutoAim) //Sprawdza poprzez InputManager czy tryb
    ↪  automatycznego celowania jest włączony
        TargetNearestEnemy();

    _nextShotTime = Time.time + _refreshRate; //Ustawia czas do odnowienia
    ↪  kolejnego strzału

    var bullet = Instantiate(_bullet, _collider2d.bounds.center,
    ↪  Quaternion.identity); //Tworzy pocisk
    bullet.GetComponent<Bullet>().Initialize(new Vector2(_shotX, _shotY),
    ↪  DamageSource.Player); //Inicjalizuje parametry pocisku
    bullet.gameObject.layer = LayerMask.NameToLayer("PlayerShots");
    ↪  //Zmienia warstwę pocisku, aby nie kolidował on z graczem
}
```

Listing 5.11: Metoda strzału przez gracza w skrypcie *PlayerController*

5.2.3. Inne elementy

Poniżej przedstawiono opis kilku innych wartych uwagi zaimplementowanych klas stanowiących komponenty możliwe do dołączenia do obiektów na scenie:

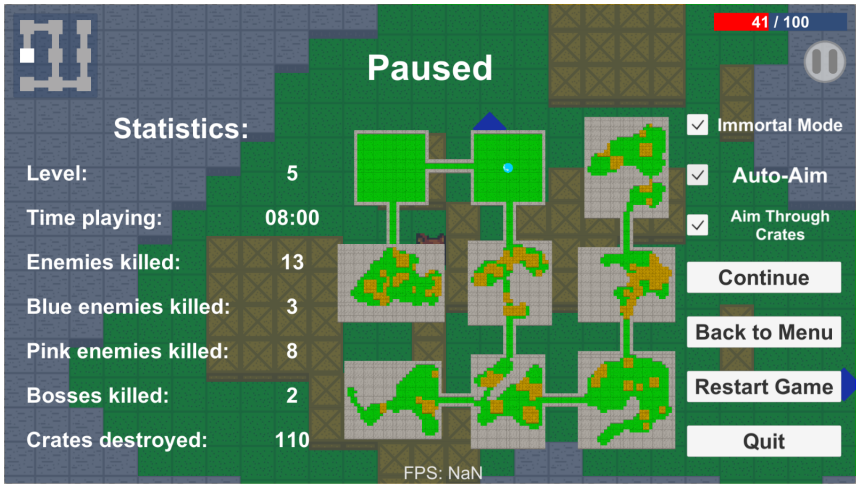
- *PlayerController* - implementuje interfejs *IDestroyable*, odpowiada za sterowanie gracza - obsługuje poruszanie postaci i strzelanie pociskami. Skrypt jest dołączony do postaci gracza.
- *PlayerAnimatorController* - skrypt kontrolujący animatorem postaci gracza, pozwala analizować ruch postaci gracza i wysyłać do komponentu *Animator* informacje jakie animacje powinny w danej chwili zostać wywołane. Dzięki niemu możliwe jest kontrolowanie animacją poruszania się postaci i animacją postoju oraz obrotem grafiki gracza (prawo, lewo). Skrypt jest dołączony do postaci gracza.
- *CameraController* - prosty skrypt zarządzający ruchem kamery. Ustawia on pozycję kamery względem pozycji gracza. Dołączony do obiektu kamery.

- *ChunkController* - skrypt zarządzający komnatą. Reaguje on na większość zdarzeń odbywających się w komnacie. Pozwala na: wykrycie czy postać gracza weszła do lub wyszła z komnaty, zamknięcie lub otwarcie przejść do innych komnat, pojawienie przeciwników, wykrycie pozostających przy życiu przeciwników. Tak jak zostało to opisane w 5.1.2, odpowiada również za tworzenie skrzyń w obrębie komnaty. Przechowuje informacje o tym, które pola w komnacie są wolne, aby możliwe było tworzenie na nich obiektów. Odpowiada również za utworzenie portalu, jeśli komnata została oznaczona jako finałowa i za poinformowanie skryptu zarządzającego *GameManager* o punkcie startowym, jeśli komnata została oznaczona jako startowa.
- *Bullet* - pozwala na ruch pocisku i reaguje na kolizję z innymi obiektami, gdzie reakcją jest zniszczenie własnego obiektu i zadanie obrażeń jeśli trafiony obiekt implementuje interfejs *IDestroyable*. Pozwala również na określenie cech pocisku: szybkość, obrażenia, czas przeładowania.
- *Enemy* - implementuje interfejs *IDestroyable*, obsługuje zachowanie przeciwników: ruch w losowym kierunku gdzie jest wolna przestrzeń i strzelanie w kierunku gracza. W inspektorze pozwala zdefiniować cechy typowe dla przeciwnika (zdrowie, obrażenia, rodzaj pocisku), ale również określić rodzaj przeciwnika i określić liczbę pocisków wystrzeliwanych jednocześnie wokół siebie.
- *DestroyableObject* - implementuje interfejs *IDestroyable*, pozwala na tworzenie nieruchomych obiektów możliwych do zniszczenia. W momencie zniszczenia może dodatkowo wywołać system cząsteczek, jeśli tylko komponent *ParticleSystem* jest dołączony do tego obiektu.
- *EndGate* - reaguje na kolizję z graczem przechodząc do kolejnego poziomu, został dołączony do obiektu portalu.
- *SortingLayerByAxis* - skrypt sortujący obiekt na tzw. warstwach sortujących, może zostać dołączony do dowolnego obiektu, który posiada dołączony standardowy komponent od Unity: *SpriteRenderer*. Pozwala on nadanie grze głębi przez możliwość przykrywania obiektów innymi obiektami, np. przykrycie gracza za ścianą.

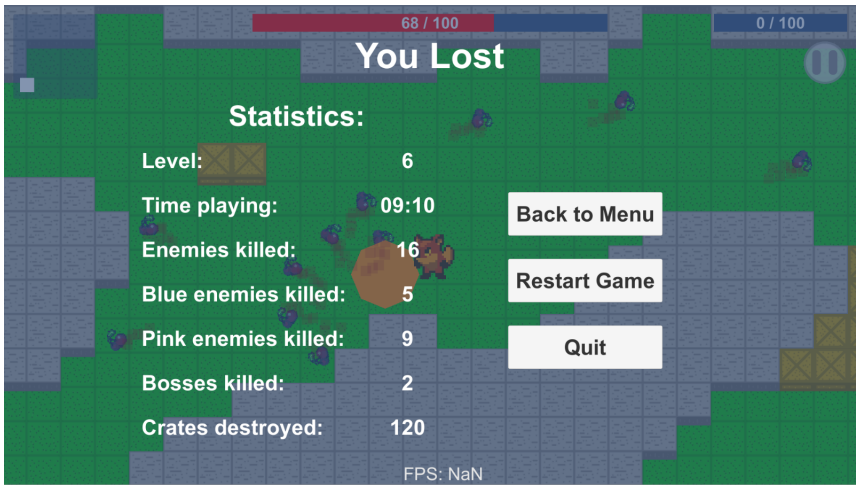
5.3. IMPLEMENTACJA INTERFEJSÓW UŻYTKOWNIKA

Kontrolę nad interfejsami w czasie rozgrywki posiada *GameUIManager*. Gra została podzielona na dwie sceny gry. Pierwszą jest menu główne, drugą jest scena rozgrywki. W scenie rozgrywki zostały zawarte wszystkie widoki interfejsów (z wyjątkiem menu) na zasadzie paneli wyświetlanych nad ekranem rozgrywki. *GameUIManager* posiada referencje do wszystkich elementów interfejsu i zarządza ich wyświetlaniem przez reagowanie na zdarzenia otrzymywane od *GameManager* oraz przez reagowanie na działania użytkownika dotyczące kontrolowanego interfejsu. Efekty zaimplementowanych interfejsów na ekranie rozgrywki przedstawiono na rysunkach: 5.5, 5.6, 5.7.

Rys. 5.5: Ekran gry z interfejsem rozgrywki (opr. wł.)



Rys. 5.6: Ekran gry z interfejsem pauzy (opr. wł.)



Rys. 5.7: Ekran gry z interfejsem przegranej (opr. wł.)

5.4. NAPOTKANE BŁĘDY I ICH ROZWIĄZANIA

5.4.1. Nie uruchamianie się na systemie Android

Opis problemu: Gra po jej poprawnym zbudowaniu w silniku Unity i zainstalowaniu na telefonie z systemem Android nie pozwalała na jej uruchomienie. Zamiast tego widoczny był czarny ekran.

Rozwiązanie: Po przeszukaniu wielu źródeł, udało się znaleźć informację o tym, że budowanie aplikacji na system Android od niedawna wymaga wsparcia dla systemów 64-bitowych [10] i systemów w wersji API: *26 (Android 8.0)*. W takim przypadku wystarczyło zmienić przyjęte ustawienia budowania aplikacji: zmieniono *Target API* z 23 (6.0) na *Automatic* (maksymalne dostępne). W celu dodania wsparcia dla architektury x64, konieczna była zmiana ustawienia *Scripting Backend* z *Mono* na *IL2CPP* co pozwoliło na dodanie wsparcia architektury *ARM64*. W efekcie budowanie takiej aplikacji odbywa się znacznie dłużej niż wcześniej. Jednak dzięki temu jest ona możliwa do uruchomienia i według wpisu na blogu [10] uruchamia się około dwóch razy szybciej.

5.4.2. Mała liczba klatek na sekundę

Opis problemu: Po właściwym uruchomieniu aplikacji okazało się, że gra jest odtwarzana w bardzo małej liczbie klatek na sekundę. Po zmierzeniu było to: 10-12 klatek na sekundę.

Rozwiązanie: Okazało się, że taka wydajność była spowodowana występowaniem dwóch kamer. Jednej głównej, drugiej dla powiększonej mini-mapy na ekranie pauzy. Mimo, że powiększona mini-mapy nie była pokazywana na ekranie telefonu, to była ona mimo wszystko renderowana. Wystarczyło aktywować / dezaktywować obiekt tej kamery w momencie pauzowania / kontynuacji rozgrywki. Pozwoliło to na uzyskanie stabilnych 60 klatek na sekundę na urządzeniu mobilnym.

5.4.3. Zmienna szybkość ruchu postaci

Opis problemu: Postacie poruszały się z różnymi szybkościami, a powinny się poruszać ze stałymi. Wszystko wydawało się zależne od liczby klatek na sekundę.

Rozwiązanie: Okazało się, że błędem było zastosowanie parametru: *Time.deltaTime* jako mnożnik w metodach poruszania postaci *Move()*, gdyż te metody były realizowane poprzez ustawienie stałej prędkości (ang. *velocity*) w każdej klatce. Użycie *Time.deltaTime* byłoby właściwe, gdyby operowano ruchem postaci poprzez nadawanie im wektorów sił,

które byłyby analizowane przez silnik fizyczny. W każdym razie pozbycie się parametru czasu pomiędzy klatkami, pozwoliło na uzyskanie stabilności poruszania się postaci.

5.4.4. Wyrzucanie gracza poza komnatę

Opis problemu: Gracz po powolnym wejściu do komnaty był z niej wyrzucany w momencie pojawienia się blokad przejść do innych komnat.

Rozwiązanie: Problemem był zasięg obszaru wykrywania kolizji (komponent *BoxCollider2D*) w obiekcie komnaty, która była zarządzany skryptem *ChunkController*. W tym skrypcie przy inicjalizacji kontrolera był ustawiany obszar kolizji zgodny z faktyczną wielkością całej komnaty. W momencie, gdy gracz naruszył obszar wykrywania, pojawiały się blokady w postaci ścian, które ze względu na posiadanie kolizji, wypychały gracza na zewnątrz komnaty. Wystarczyło zmniejszyć obszar wykrywania, aby pozostawić granicę błędu dla takich przypadków, został on zmieniony o tyle, aby postać była wpychana do wewnątrz komnaty, co rozwiązało problem.

5.4.5. Krótkie zawieszenie gry przy zmianie poziomu

Opis problemu: Dodana nakładka będąca ekranem ładowania przy zmianie poziomu nie działała poprawnie. Była wyświetlana dopiero po wygenerowaniu nowego poziomu, co z punktu widzenia gracza wyglądało jakby aplikacja się zawiesiła.

Rozwiązanie: Problem tkwi w tym, że zmiana poziomu w metodzie *NextLevel()* klasy *GameManagera* następuje poprzez rozesłanie sygnału o ładowaniu poziomu, a potem utworzenie obiektu *LevelGenerator* (co zostało pokazane na listingu 5.12). *LevelGenerator* natychmiast rozpoczyna generowanie poziomu, a po zakończeniu tej operacji sam się niszczy. Sygnał o ładowaniu jest obserwowany przez *GameUIManager*, który powinien wyświetlić ekran ładowania. Po zbadaniu logów konsol i skorzystaniu z debuggera programu *Visual Studio* wywnioskowano, że wszystko jest wykonywane w odpowiedniej kolejności - *GameUIManager* faktycznie wykonuje własne metody w związku z wykryciem sygnału. W tych metodach zmienia wyświetlanie obiektów interfejsu graficznego. Jednak nie są one odświeżane na tzw. płótnie gry (ang. *canvas*) [24]. Po głębszym przeanalizowaniu okazało się, że po prostu nie następuje przejście do kolejnej klatki dopóty, dopóki *LevelGenerator* nie zostanie zniszczony. Udało się poprawić ten błąd poprzez opóźnienie utworzenia obiektu *LevelGeneratora* do kolejnej klatki. Zrealizowano to przy wykorzystaniu korutyny (ang. *coroutine*) [25]. Poprawioną metodę przejścia do kolejnego poziomu przedstawiono na listingu 5.13.

```
public static void NextLevel()
{
    OnLoading?.Invoke(true); //Sygnał obserwowany przez GameUIManager
    Instantiate(Instance.OtherPrefabsData.LevelGenerator); //Utworzenie
    ↪ obiektu LevelGenerator
}
```

Listing 5.12: Kod metody *NextLevel()* klasy *GameManager* powodujący zawieszenie rozgrywki

```
public static void NextLevel()
{
    OnLoading?.Invoke(true); //Sygnał obserwowany przez GameUIManager
    Action action = delegate //Definicja poleceń do zrealizowania w
    ↪ korutynie
    {
        Instantiate(Instance.OtherPrefabsData.LevelGenerator);
    };
    Instance.StartCoroutine("CallAfterFrame", action);
}

public IEnumerator CallAfterFrame(Action action)
{
    yield return new WaitForEndOfFrame(); //Czekanie z akcją do kolejnej
    ↪ klatki
    action.Invoke();
}
```

Listing 5.13: Poprawiony kod metody *NextLevel()* klasy *GameManager* zapobiegający zawieszeniu rozgrywki

5.4.6. Podwójne działanie pocisku

Opis problemu: Czasami pocisk zadawał obrażenia dwukrotnie.

Rozwiązanie: Pocisk w momencie trafienia na kolizję zadawał jej obrażenia, po czym był niszczony (listing 5.14). Problem wynikał z tego, że od wysłania sygnału do zniszczenia pocisku, do czasu jego faktycznego zniszczenia mogły zostać zarejestrowane inne kolizje z tym pociskiem. Rozwiązaniem mogło być wykorzystanie innej metody oferowanej przez klasę *MonoBehaviour*: *DestroyImmediate()* zamiast *Destroy()*. Jednak stosowanie natychmiastowego niszczenia jest odradzane ze względu na znacznie wyższe koszty wy-

dajnościowe. Zdecydowano się pozostawić użycie metody *Destroy()*, ale do samej metody wykrycia kolizji dodana została flaga, która pozwoli na sprawdzenie czy efekt pocisku został już raz nałożony i blokadę ponownego nałożenia. Nowy kod został przedstawiony na listingu 5.15.

```
private void OnCollisionEnter2D(Collision2D collision)
{
    Destroy(gameObject); //Zniszczenie pocisku

    var destroyable = collision.gameObject.GetComponent<IDestroyable>();
    destroyable?.DealDamage(_damage, _source); //Zadanie obrażeń poprzez
     ↪  użycie interfejsu IDestroyable
}
```

Listing 5.14: Początkowy kod reakcji na kolizję obiektu klasy *Bullet*

```
private bool _collisionCalled; //Flaga - czy metoda była wykonana
private void OnCollisionEnter2D(Collision2D collision)
{
    if (!_collisionCalled) //Sprawdzenie flagi
    {
        _collisionCalled = true; //Ustawienie flagi
        Destroy(gameObject); //Zniszczenie pocisku.

        var destroyable =
         ↪  collision.gameObject.GetComponent<IDestroyable>();
        destroyable?.DealDamage(_damage, _source);
    }
}
```

Listing 5.15: Poprawiony kod reakcji na kolizję obiektu klasy *Bullet*

5.4.7. Obciążenie procesora skryptem sortującym

Opis problemu: Przy testowaniu wydajności w narzędziu *Profiler* wbudowanym w edytor Unity napotkano na opóźnienie procesora pochodzące z wywołania metody *Update()* w skrypcie *SortingLayerByAxis()*. Liczba wywołań była zgodna z liczbą występujących na mapie obiektów terenu. Co przy obszarze wielkości 30 tys. pól powodowało obciążenie procesora w wysokości 3% i pochłaniało 3.5 ms czasu procesora co klatkę.

Rozwiązanie: Nie jest to ogromne użycie procesora, ale wystarczające, aby zainteresowało autora pracy. Skrypt *SortingLayerByAxis()* miał za zadanie sortować elementy na

mapie po osi sortowania. Był dołączony do wszystkich obiektów, a w inspektorze edytora można było wybrać czy sortowanie ma być wykonywane co klatkę, czy tylko raz przy utworzeniu obiektu. Implementacja po prostu sprawdzała co klatkę instrukcją warunkową to, czy należy wykonać sortowanie. Kod takiej implementacji przedstawiono na listingu 5.16. Aby rozwiązać problem wywoływania co klatkę przykładowych 30 tys. instrukcji warunkowych, wystarczyło w metodzie *Start()* dodać wyłączanie tego skryptu w przypadku, gdy miał on zostać wykonany tylko raz. Poprawioną metodę przedstawiono na listingu 5.17. Pozwoliło to zmniejszyć liczbę wywołań tego skryptu z 30 tys. do 1 (postać gracza), gdy na mapie nie ma aktualnie przeciwników. A tym samym zmniejszyć czas użycia procesora przez ten skrypt z 3.5 ms to 0 ms.

```
private enum SortMode
{
        OnStart,
        OnUpdate
}
private void Start()
{
        if (_sortMode == SortMode.OnStart)
                UpdateSortingOrder();
}
private void Update()
{
        if (_sortMode == SortMode.OnUpdate)
                UpdateSortingOrder();
}
```

Listing 5.16: Fragment skryptu *SortingLayerByAxis()*

```
private void Start()
{
    if (_sortMode == SortMode.OnStart)
    {
        UpdateSortingOrder();
        this.enabled = false;
    }
}
```

Listing 5.17: Poprawiona metoda *Start()* w skrypcie *SortingLayerByAxis()*

ZAKOŃCZENIE

W pracy zrealizowano wszystkie założone cele, zarówno te podstawowe wynikające z tematu pracy, jak i te dodatkowe wynikające z powstałego projektu gry. Generowanie poziomów jest wykonywane poprawnie, bez błędów. Podzielenie poziomu na labirynt i jego komnaty, w połączeniu z parametrami, które taki poziom opisują, pozostawia duże pole manewru w celu zmiany charakterystyki generowanych obszarów. Mimo, że cały projekt stanowi prototyp, to zaimplementowane zasady rozgrywki są wystarczająco rozbudowane, aby mogły konkurować z innymi, już wydanymi grami. Urządzenia mobilne otrzymały pełne wsparcie, gra jest do nich przystosowana pod względem istnienia przyjaznego interfejsu graficznego, jak i wygodnego sposobu sterowania.

W zrealizowanym prototypie zauważa się ogromny potencjał do rozwoju. Zarówno w części generowania poziomów, jak i części rozgrywki. Obie części wcale nie muszą występować razem. Możliwe jest wykorzystanie generowania poziomów w innym projekcie, który mógłby z takiego generowania skorzystać. Jak również możliwe jest wykorzystanie zaimplementowanej rozgrywki do gry o nieco innym charakterze, która nie musi korzystać z proceduralnego generowania. Mimo to, na pewno warto poświęcić uwagę całej projektowi.

W pierwszej kolejności do rozwoju powinny być elementy uatrakcyjniające użytkownikom (tu: graczom) korzystanie z aplikacji, tudzież poprawiające uciążliwe elementy. Na pewno takimi są: brak informacji o tym ilu pozostało jeszcze przeciwników w aktualnej komnacie oraz brak informacji o tym gdzie tacy przeciwnicy są w przypadku ogromnych komnat. Po poprawieniu uciążliwych elementów, można by dokonać rozwoju zawartości gry. Pomysłami jakie autor rozważa jest dodanie: nowych rodzajów interaktywnych przeszkód na planszach; nowych przeciwników i poszerzenie liczby cech jakimi mogą zostać opisani; nowe rodzaje pocisków z większą liczbą cech i z możliwością ich zmiany w czasie rozgrywki; system rozwoju postaci gracza; większa liczba statystyk; system osiągnięć zapewniający wyzwania dla najbardziej wymagających graczy; możliwość gry razem przez kilku osób na różnych urządzeniach poprzez wykorzystanie sieci LAN.

Innym istotnym elementem w rozwoju jest poprawienie wydajności. W tym celu już wiele zrobiono, ale z pewnością można zrobić jeszcze więcej. Obecnie generowanie ogromnych poziomów może trwać bardzo długo. Kluczową rolę w procesie generowania poziomu posiada jego wizualizacja już po wygenerowaniu danych mapy. Można by to znacznie poprawić przez niewyświetlanie pełnej mapy, a jedynie wizualizowanie obszarów, które gracz widzi. Dodatkowo zamiast masowo tworzyć i niszczyć obiekty na scenie gry, można

by je dezaktywować i przechowywać w pamięci w celu ponownego wykorzystania co mogłoby pozwolić na szybsze przechodzenie między kolejnymi poziomami. Potencjalnym rozwojem mogłoby być zapewnienie możliwości tworzenia poziomów o nieskończonej powierzchni, które byłyby generowane w kawałkach w momencie zbliżania się gracza do granic mapy.

Po powyższym można wnioskować, iż mimo że sam projekt był bardzo rozbudowany, to jego prawdziwy rozwój może się dopiero zacząć. Potencjał jaki tkwi w proceduralnym generowaniu jest ogromny. Wraz z dalszym rozwojem informatyki, takie generowanie będzie wchodziło na coraz wyższe poziomy. A jego potencjał nie ogranicza się tylko do gier komputerowych. Z czasem zapewne pojawią się nowe miejsca wykorzystujące proceduralne generowanie, gdzie obecnie nikt by się tam go nie spodziewał.

BIBLIOGRAFIA

[1] Ansimuz, *Sunny Land* (Unity Asset Store).
https://assetstore.unity.com/packages/2d/characters/sunny-land-103349
[ost. dost. 2 stycznia 2019].

[2] Bethke, E., *Game Development and Production*, Wordware game developer's library (Wordware Pub., 2003).

[3] Bhargava, A., *Algorytmy. Ilustrowany przewodnik* (Helion, 2017).

[4] Buck, J., *Maze Generation: Prim's Algorithm*,
http://weblog.jamisbuck.org/2011/1/10/maze-generation-prim-s-algorithm.
10.01.2011.
[ost. dost. 2 stycznia 2019].

[5] Buck, J., *Mazes for Programmers: Code Your Own Twisty Little Passages* (Pragmatic Bookshelf, 2015).

[6] ChillRoom, *Soul Knight* (Google Play).
https://play.google.com/store/apps/details?id=com.ChillyRoom.DungeonShooter
[ost. dost. 2 stycznia 2019].

[7] Fine, R., *UnityScript's long ride off into the sunset*,
https://blogs.unity3d.com/2017/08/11/unityscripts-long-ride-off-into-the-sunset/. 11.08.2017.
[ost. dost. 2 stycznia 2019].

[8] Garg, P., *Depth First Search*,
https://www.hackerearth.com/practice/algorithms/graphs/depth-first-search/tutorial/. 2019.
[ost. dost. 2 stycznia 2019].

[9] Graessle, P., Baumann, H., Baumann, P., *UML 2.0 w akcji. Przewodnik oparty na projektach* (Helion, 2006).

[10] Habets, Y., *Meeting Google Play requirements in the future*,
https://blogs.unity3d.com/2017/12/20/meeting-google-play-requirements-in-the-future/. 20.12.2017.
[ost. dost. 2 stycznia 2019].

[11] Jaimini, V., *Flood-fill Algorithm*,
https://www.hackerearth.com/practice/algorithms/graphs/flood-fill-algorithm/tutorial/. 2019.
[ost. dost. 2 stycznia 2019].

[12] Lague, S., *Unity Tutorials - Cellular Automata* ,
https://unity3d.com/learn/tutorials/projects/procedural-cave-

generation-tutorial/cellular-automata.
[ost. dost. 2 stycznia 2019].

[13] Linietsky, J., Manzur, A., Godot community, *Dokumentacja Godot - Introduction*, https://docs.godotengine.org/en/3.0/about/introduction.html.
[ost. dost. 2 stycznia 2019].

[14] Lovato, N., *How To Perfect Your Game's Core Loop*, https://gameanalytics.com/blog/how-to-perfect-your-games-core-loop.html. 13.07.2018.
[ost. dost. 2 stycznia 2019].

[15] Pullen, W., *Maze Algorithms*, http://www.astrolog.org/labyrnth/algrithm.htm. 20.11.2015.
[ost. dost. 2 stycznia 2019].

[16] Red Blob Games, *Making maps with noise functions*, https://www.redblobgames.com/maps/terrain-from-noise/. 07.07.2015.
[ost. dost. 2 stycznia 2019].

[17] Schiff, J., *Cellular Automata: A Discrete View of the World*, Wiley Series in Discrete Mathematics & Optimization (Wiley, 2011).

[18] Shaker, N., Togelius, J., Nelson, M.J., *Procedural Content Generation in Games: A Textbook and an Overview of Current Research* (Springer, 2016).

[19] Shiffman, D., *The Nature of Code* (2012).

[20] Singh, R., *Understanding Common Intermediate Language (CIL)*, https://www.codeproject.com/Articles/362076/Understanding-Common-Intermediate-Language-CIL. 23.03.2013.
[ost. dost. 2 stycznia 2019].

[21] SlackMoehrle, Ricardo, Minggo, Walzer, Jare, James, Chengzhe, *Dokumentacja Cocos2d-x - Basic Concepts*, https://docs.cocos2d-x.org/cocos2d-x/en/basic_concepts/.
[ost. dost. 2 stycznia 2019].

[22] Tomaszewski, T., *Jak napisać dobrą historyjkę użytkownika (user stories)?*, https://productvision.pl/2017/napisac-dobra-historyjke-uzytkownika-user-stories/. 17.07.2017.
[ost. dost. 2 stycznia 2019].

[23] Unity Technologies, *Dokumentacja silnika Unity*, https://docs.unity3d.com/Manual/UnityManual.html. 04.12.2018.
[ost. dost. 2 stycznia 2019].

[24] Unity Technologies, *Dokumentacja Unity - Canvas*, https://docs.unity3d.com/Manual/UICanvas.html. 04.12.2018.
[ost. dost. 2 stycznia 2019].

[25] Unity Technologies, *Dokumentacja Unity - Coroutines*, https://docs.unity3d.com/Manual/Coroutines.html. 04.12.2018.
[ost. dost. 2 stycznia 2019].

[26] Unity Technologies, *Dokumentacja Unity - Creating and Using Scripts*, https://docs.unity3d.com/Manual/CreatingAndUsingScripts.html. 04.12.2018. [ost. dost. 2 stycznia 2019].

[27] Unity Technologies, *Dokumentacja Unity - Execution Order of Event Functions*, https://docs.unity3d.com/Manual/ExecutionOrder.html. 04.12.2018. [ost. dost. 2 stycznia 2019].

[28] Unity Technologies, *Dokumentacja Unity - Introduction to components*, https://docs.unity3d.com/Manual/Components.html. 04.12.2018. [ost. dost. 2 stycznia 2019].

[29] Unity Technologies, *Dokumentacja Unity - Prefabs*, https://docs.unity3d.com/Manual/Prefabs.html. 04.12.2018. [ost. dost. 2 stycznia 2019].

[30] Unity Technologies, *Dokumentacja Unity - Rigidbody 2D*, https://docs.unity3d.com/Manual/class-Rigidbody2D.html. 04.12.2018. [ost. dost. 2 stycznia 2019].

[31] Unity Technologies, *Dokumentacja Unity - What is a Particle System?*, https://docs.unity3d.com/Manual/PartSysWhatIs.html. 04.12.2018. [ost. dost. 2 stycznia 2019].

[32] Unity Technologies, *Unity 2018.3 - Release notes*, https://unity3d.com/unity/whats-new/unity-2018.3.0. 2018. [ost. dost. 2 stycznia 2019].

[33] Unity Technologies - Community Team, *Unity Unveils 2018 Roadmap at GDC*, https://blogs.unity3d.com/2018/03/20/unity-unveils-2018-roadmap-at-gdc/#small. 20.03.2018. [ost. dost. 2 stycznia 2019].

[34] Wijman, T., *Mobile Revenues Account for More Than 50% of the Global Games Market as It Reaches $137.9 Billion in 2018*, https://newzoo.com/insights/articles/global-games-market-reaches-137-9-billion-in-2018-mobile-games-take-half/. 30.04.2018. [ost. dost. 2 stycznia 2019].

[35] Zingl, A., *The Beauty of Bresenham's Algorithm*, http://members.chello.at/~easyfilter/bresenham.html. 08.2016. [ost. dost. 2 stycznia 2019].

[36] Zingl, A., *A Rasterizing Algorithm for Drawing Curves* (2012). http://members.chello.at/~easyfilter/Bresenham.pdf [ost. dost. 2 stycznia 2019].

SPIS RYSUNKÓW

SPIS LISTINGÓW

SPIS TABEL