Uniwersytet Warmińsko-Mazurski w Olsztynie
Wydział Matematyki i Informatyki

Kierunek: Informatyka

Krystian Potępa

Gra platformowa "Pamięć 404"

Praca inżynierska wykonana
w Katedrze Analizy Zespolonej
pod kierunkiem
dr. Piotra Jastrzębskiego

Olsztyn, 2020 rok

University of Warmia and Mazury in Olsztyn
Faculty of Mathematics and Computer Science

Field of Study: Computer Science

Krystian Potępa

Platform game „Memory 404".

Engineer's Thesis is performed
in the Chair of Complex Analysis
under supervision of
Piotr Jastrzębski, PhD

Olsztyn, 2020

Streszczenie

Zadaniem pracy inżynierskiej jest stworzenie projektu i implementacji gry platformowej z wykorzystaniem środowiska Unity 3D w oparciu o język C#. W pracy przedstawiono opis fabuły, projekt, implementację i opis testowania wytworzonego oprogramowania. Dołączona aplikacja jest dostępna dla systemu operacyjnego Windows. Tytuł gry nawiązuje numerem do jednego z najbardziej rozpoznawalnych błędów aplikacji oraz stron internetowych - 404, zwany również jako `Not Found`. Rozgrywka nie będzie zbyt wymagająca, aby każda nawet niedoświadczona osoba mogła bez problemu ją ukończyć.

Abstract

Platform game „Memory 404”.

This engineering thesis aim is to design and implement a platform game using Unity 3D environment and C# language. In the thesis the author presented the plot of the game, design, implementation and description of testing the developed software. The attached application is available for Windows OS/operating system. The number included in the title of the game references one of the most well-known errors of applications and webpages - 404, also known as `Not Found`. The gameplay is not too challenging so that even an inexperienced person should be able to complete it.

Podziękowanie

Chciałbym złożyć serdeczne podziękowania mojemu promotorowi **dr. Piotrowi Jastrzębskiemu** za pomoc w realizacji mojej pracy, wyrozumiałość oraz przekazanie wiedzy.

Spis treści

Wprowadzenie

Na przestrzeni lat rynek technologiczny stał się jednym z najbardziej opłacanych inwestycji. Na co dzień korzystamy z nowoczesnych urządzeń, których zadaniem jest ułatwić nam pracę, poprawić warunki życia lub po prostu uatrakcyjnić spędzany czas. Pośród niezliczonej ilości aplikacji z olbrzymim wachlarzem funkcjonalności na naszym urządzeniu domowym możemy spotkać się z terminem "gry komputerowej".

Pod tym pojęciem można rozumieć specjalny rodzaj oprogramowania, w którym użytkownik ingeruje w wykreowany świat. Postawione są mu zadania, które musi wypełnić, aby osiągnąć sukces. Warunek ukończenia jest zależny od gatunku aplikacji: logiczne postawią nas przed trudnymi łamigłówkami i wytężą nasz umysł, platformowe opierać się będą na ukończeniu poziomu, unikając pułapek i przeciwników a produkcje wieloosobowe zmuszą nas do kooperacji lub walki z innymi graczami. Pierwsza gra, wydana została w 1972 roku na automaty przez firmę Atari. "Pong" był prostym symulatorem tenisa, mimo tego odniósł ogromny sukces i zapewnił firmie popularność.

Celem mojej pracy inżynierskiej jest stworzenie gry w języku C#. Aplikacja jest tworzona w programie Unity 3D dla komputerów z systemem Windows. Jest to dwuwymiarowa gra platformowa, w której celem gracza jest eksplorowanie pomieszczeń wypełnionych walutą oraz przeszkodami, z którymi będzie musiał się zmierzyć. Aplikacja z założenia ma być prosta i przejście jej nie powinno sprawić użytkownikowi problemu. Czas całej rozgrywki to około 30 minut.

Rozdział 1

Opis fabuły

Fabuła toczy się w roku 2050, gdzie duży nacisk kładzie się na automatyzację oraz sztuczną inteligencję. Wielka korporacja „ZeroStress", zwana również fabryką przyszłości zajmuje się masową wytwórnią sprzętów elektronicznych. Specjalny dział zajmuje się wytwarzaniem nowych rozwiązań, które miałyby pomóc człowiekowi w zwykłych czynnościach domowych. Wysoka ambicja na rozwój wśród pracowników powoduje prace nad robotem domowym, który zastępowałby człowieka w wielu czynnościach a dzięki sztucznej inteligencji uczyłby się informacji o domownikach. Dzięki temu każdy robot byłby indywidualnie spersonalizowany do potrzeb właściciela, co czyni go rewolucyjnym na rynku.

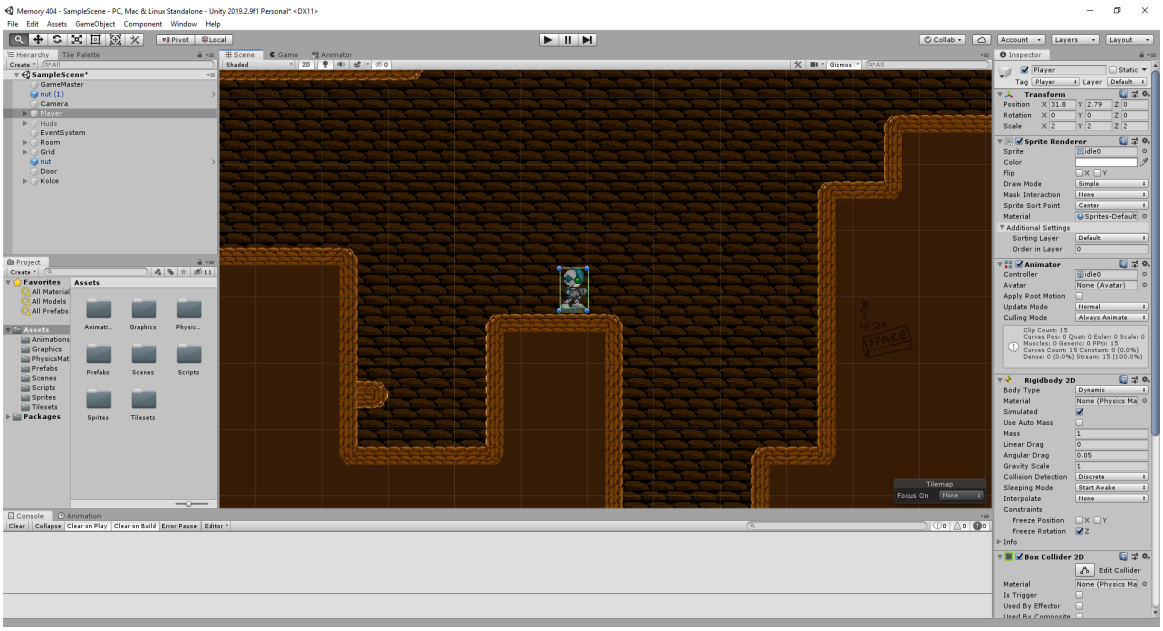
Setki prób i błędów dają w końcu wyniki, gdy po paru latach powstaje prototyp urządzenia. „Mand0m", bo tak robot został nazwany, przy próbie prezentacji nie wykonuje poleceń, zawiesza się oraz wyłącza bezpowrotnie. Zostaje odesłany do zapomnianych czeluści budynku, gdzie czeka na utylizację, gdy trwają prace nad nowym modelem.

Parędziesiąt lat później, w wyniku zwarcia zapomniany wynalazek zostaje uruchomiony i mimo awarii wewnętrznego rdzenia postanawia udać się do stacji naprawczej. Jak się później okazuje, cała fabryka została sparaliżowana przez zaawansowaną sztuczną inteligencję a z centrum naprawczego zostały zgliszcza. Jedyną szansą na przetrwanie zostaje ucieczka z zapuszczonych tuneli starej firmy, korzystając z wbudowanych w niego funkcjonalności. Pokonując liczne przeszkody oraz wrogie roboty, udaje mu się dotrzeć do kryjówki robota, który osiągnął samoświadomość i nie zamierza usługiwać rasie ludzkiej. "Remn4nt" jest nową wersją maszyny, która powstała po awarii głównego bohatera. Zamierza pozbyć się przybysza, wrzucając go do lochów, skąd ucieczka jest bardziej skomplikowana i niebezpieczna.

Rozdział 2

Wykorzystane narzędzia i technologie

Do wykonania mojej gry, wykorzystam program Unity w wersji 2019.2.9f1. Jest to zintegrowane środowisko, pozwalające na stworzenie gry dwu- lub trójwymiarowej z pomocą różnych języków programowania. Środowisko ma wbudowane wiele przydatnych algorytmów, które pomogą mi w wykreowaniu świata oraz fizyki gry. Do pisania skryptów posługuję się językiem C# oraz programem Visual Studio, który posiada oficjalną wtyczkę, ułatwiającą współpracę z Unity.



Rysunek 2.1. Interfejs programu Unity.

Podstawą każdej gry platformowej jest odpowiednie ustawienie fizyki gracza. Pod tym pojęciem rozumiem zbiór wszystkich mechanizmów i algorytmów pozwalających przełożyć pojęcia fizyczne na język "komputerowy". Postać, którą poruszamy się po planszy, musi posiadać zaimplementowane sterowanie, takie jak poruszanie się w lewo oraz w prawo, skok albo atak. Z pomocą przychodzi wbudowany komponent `Rigidbody2D` (odpowiednik pojęcia bryły sztywnej), który dodany do obiektu, nadaje mu właściwości dynamiczne i pozwalające na zastosowanie do niego praw fizyki znanych chociażby ze szkoły.

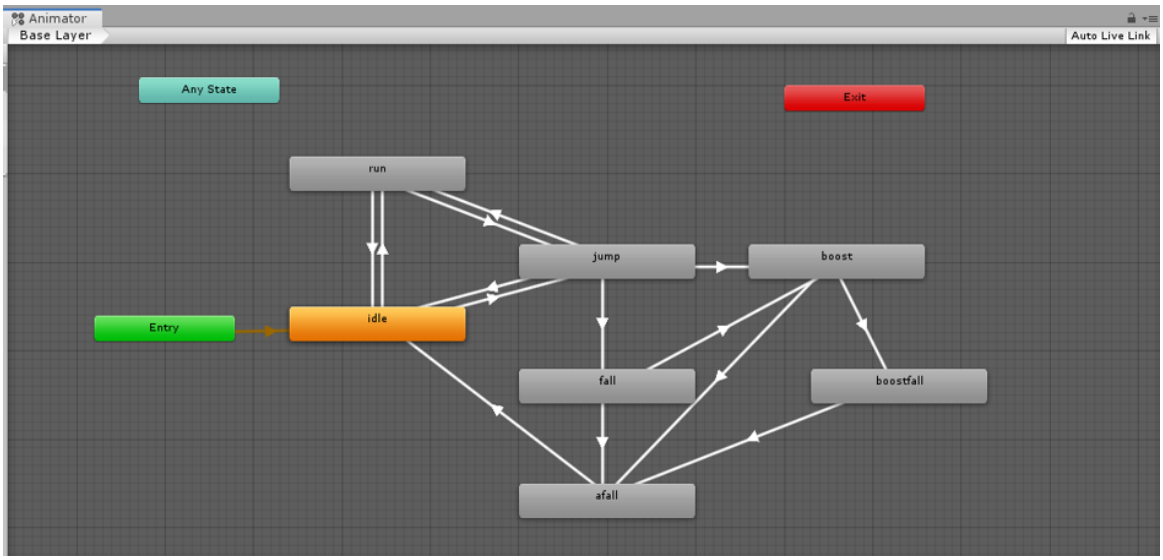
Drugim ważnym elementem mechaniki są kolizje (zderzenia). Najważniejszą z nich dla gry platformowej jest styczność bohatera z podłożem. Kiedy nie uwzględnimy takiego przypadku, postać będzie w nieskończoność spadać w dół planszy. Do wykrywania kolizji użyjemy komponentu `BoxCollider2D`, dzięki któremu będziemy mogli określić granice postaci oraz ziemi, po której ma chodzić. Używając `BoxCollidera` możemy wprowadzić również różne czujniki warunkowe, jak np. sprawdzanie czy postać znajduje się w danym momencie na ziemi albo

Rysunek 2.2. Wygląd BoxCollidera.

uaktywnienie pułapki, gdy postać znajdzie się w danym fragmencie mapy. Dzięki temu uda nam się stworzyć obiekty, które możemy zebrać podczas eksploracji naszej planszy, oraz takie które potrafią zadać nam obrażenia. Obszar kolizji obiektu oznaczony jest domyślnie w Unity 3D jako blok z zielonymi konturami. Podczas działania aplikacji powierzchnia aktywnego komponentu jest niewidoczna dla użytkownika, dlatego z poziomu widoku dostępnego dla użytkownika nie jesteśmy w stanie dostrzec elementów, z którymi mogą kolidować nasze obiekty. Jeżeli jednak nasz obiekt posiada inny kształt niż prostokąt, możemy dowolnie ustawić kontur kolizji według naszych upodobań, nie ograniczając się do czworokątów.

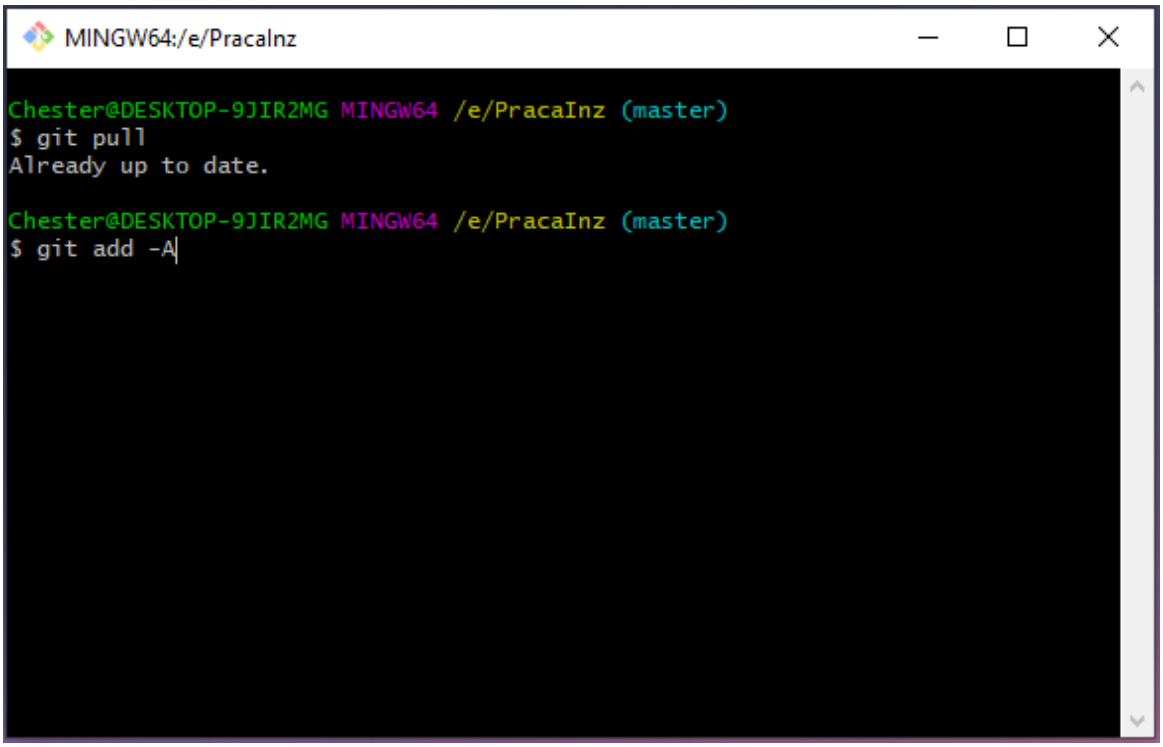
`SpriteRenderer` jest funkcjonalnością prosta, lecz niezbędną. Przypisuje ona grafikę do naszego obiektu dzięki czemu nasz obiekt nie zostaje pusty. Przez to postać, którą sterujemy ma odpowiedni kształt a później możemy stworzyć całe sekwencje animacji korzystając z wytworzonych do obiektu kopii obrazów. Możemy również ustawić filtrowanie, wielkość w pikselach oraz rodzaj powielania dla większych obiektów. Posiada również wbudowany `SpriteEditor`, dzięki któremu możemy wyznaczyć oddzielne klatki dla obrazka składającego się z więcej niż jednego elementu. Wielki obraz przycinany jest w edytorze Unity 3D i rozdzielany na pojedyncze sprite'y, z których możemy korzystać niezależnie od innych.

`Animator` jest komponentem odpowiadającym za włączanie odpowiednich animacji. Po wybraniu animacji domyślnej, możemy dorzucać kolejne bloki. Połączone są one ze sobą wektorem, który posiada w sobie warunek przejścia, czas przejścia oraz inne dodatkowe ustawienia graficzne. Aby postać opuściła animację postoju i przeszła do animacji biegu, muszą zostać spełnione 2 warunki: bohater nie może być w powietrzu (zachodzi aktywna kolizja z podłożem) oraz prędkość bohatera jest większa niż 0,001. Gdy natrafimy na tę kondycję, bieg będzie aktywną animacją postaci, dopóki prędkość postaci będzie równa 0, bądź też spełnią się inne warunki dla innej animacji. Komponent ten wykorzystać można również do nadawania ruchu obiektom,



Rysunek 2.3. Schemat Animatora.

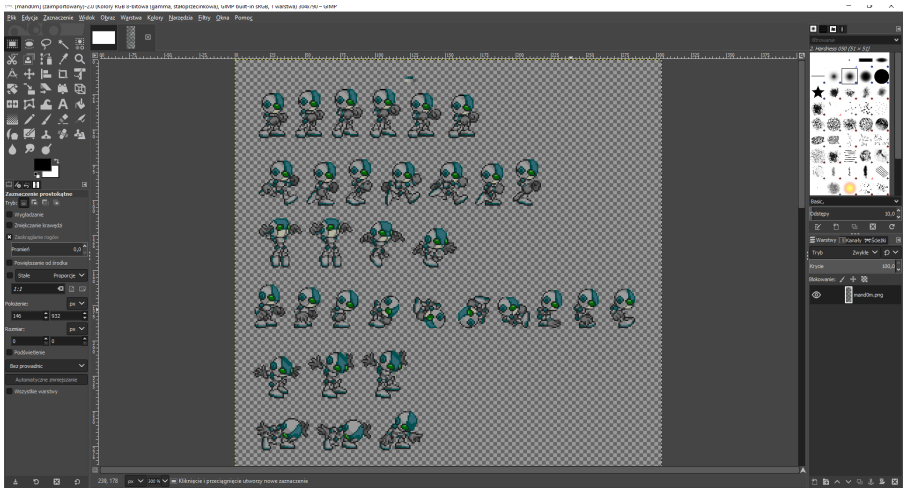
skalowaniu oraz obracaniu ich. Dzięki temu możemy stworzyć na przykład platformy, które odpowiednio ustawione w schemacie animatora potrafią poruszać się w wielu różnych kierunkach.



Rysunek 2.4. Okno komend Git Basha.

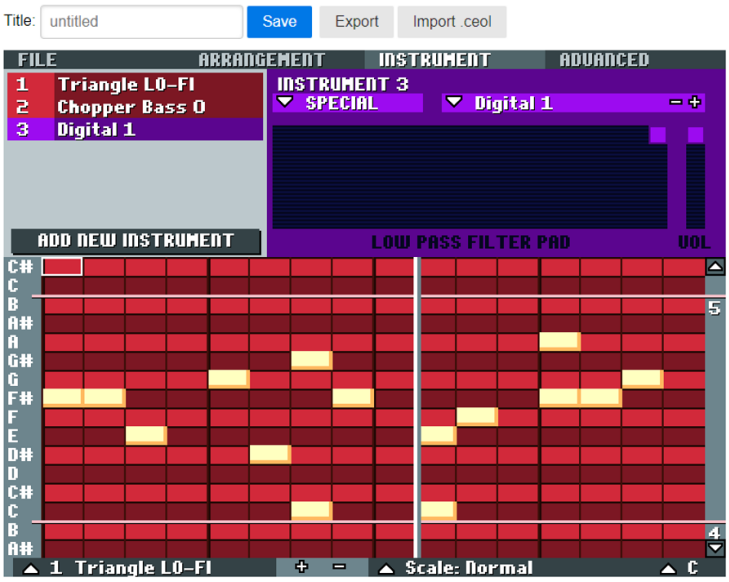
Projekty gier składają się z dużej ilości elementów, często łącznie zajmując po parę gigabajtów. Kopiowanie projektu na zwykły nośnik danych jest problemem zwłaszcza, że przez wypadek losowy bardzo łatwo stracić nasz projekt. Dane trzymane w chmurze mają mniejsze prawdopodobieństwo utraty się oraz są łatwiejsze w prezentacji innym osobom. Korzystając z serwisu `http://www.github.com` (jednej z internetowych wersji rozposzonego systemu kontroli

wersji git) z podłączonym programem GitBash dla systemu Windows mogę szybko przenieść cały swój projekt do repozytorium, do którego wgląd mają upoważnione przeze mnie osoby.



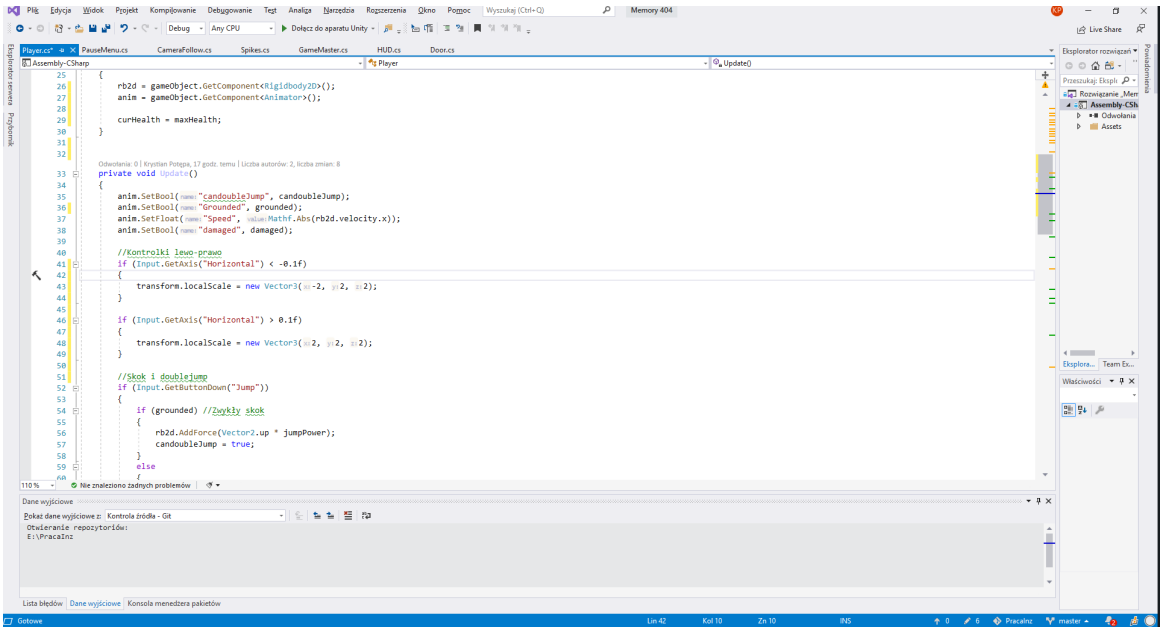
Rysunek 2.5. Interfejs programu GIMP.

GIMP jest darmowym zaawansowanym programem od grafiki rastrowej stworzony na różne systemy operacyjne (Windows, Linux, MacOS). W odróżnieniu od Painta w pełni obsługuje przezroczystość (kanał alpha). Ponadto umożliwia pracę na wielu warstwach. W moim kontekście jest to istotne. Dzięki temu zyskałem możliwość tworzenia obrazków nieposiadających zbędnych teł.



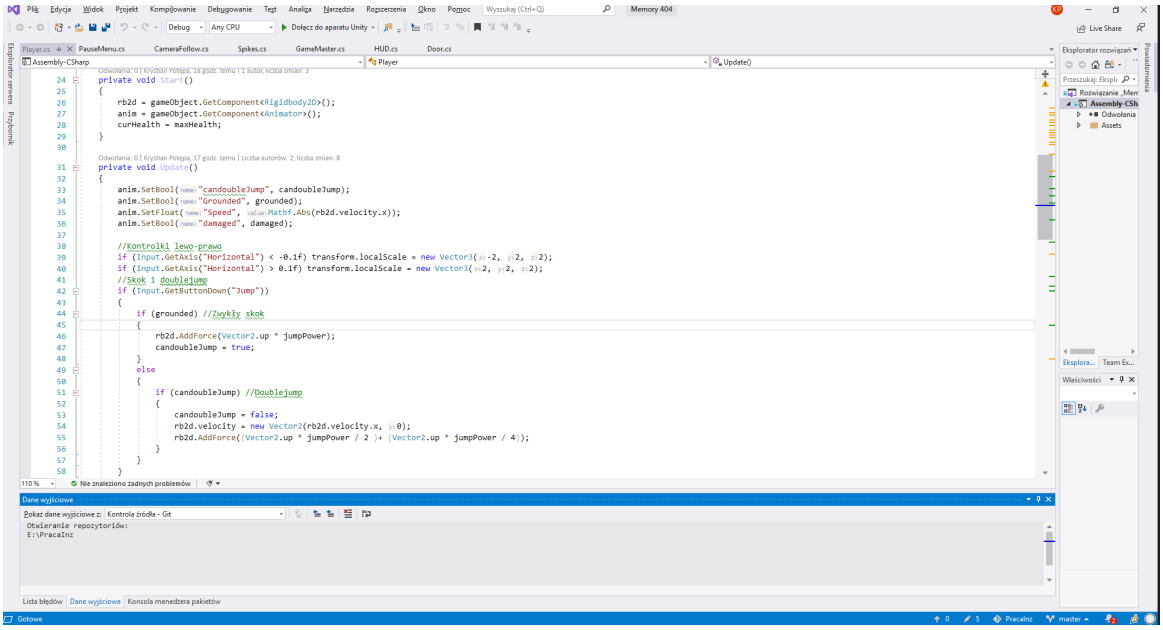
Rysunek 2.6. Wygląd interfejsu Bosca Ceoil.

Bosca Ceoil w wersji 2.0 jest programem na licencji FreeBSD do tworzenia prostej muzyki. Można pobrać go na komputer lub skorzystać z niej online przez przeglądarkę. Wybieramy odpowiedni instrument i ustawiamy melodię w odpowiedniej tonacji na osi czasu. Możemy również zmienić tempo piosenki, dodać więcej intrumentów albo zmienić pattern. Po skończeniu otrzymujemy plik audio z rozszerzeniem .wav, który bez problemu zostanie zaimportowany przez Unity.



Rysunek 2.7. Program Visual Studio z otwartym skryptem.

Visual Studio jest profesjonalnym środowiskiem programistycznym (IDE) od firmy Microsoft, stworzoną do tworzenia własnych aplikacji. Edytor współpracuje z wieloma językami programowania oraz posiada graficzny interfejs użytkownika. Posiada również duży wybór platform, na którą ma być napisana aplikacja: komputery stacjonarne z systemem Windows, telefony wyżej wymienionej firmy lub konsole do gier XBOX. Program posiada obfity zbiór rozszerzeń, które mogą pomóc nam w pisaniu kodu. Istnieje również sporo wtyczek, które kooperują z innymi programami jak GitHub czy Unity 3D. Interfejs jest bardzo czytelny i zrozumiały.



Rysunek 2.8. Kod przemodelowany przez program ReSharper.

ReSharper firmy JetBrains jest rozszerzeniem do Visual Studio, które oferuje dużo możliwości użytkownikom próbujących własnych sił w pisaniu kodu. Dodatek ten oferuje m.in. szybką naprawę kodu lub też zaproponowanie innej, zazwyczaj skuteczniejszej solucji. Dla mnie jednym z najlepszych jego funkcji jest poprawa estetyki kodu. Przy kombinacji paru klawiszy nasz cały kod w danym pliku jest automatycznie transformowany tak, aby zajmował mniej miejsca i był bardziej przejrzysty.

Rozdział 3

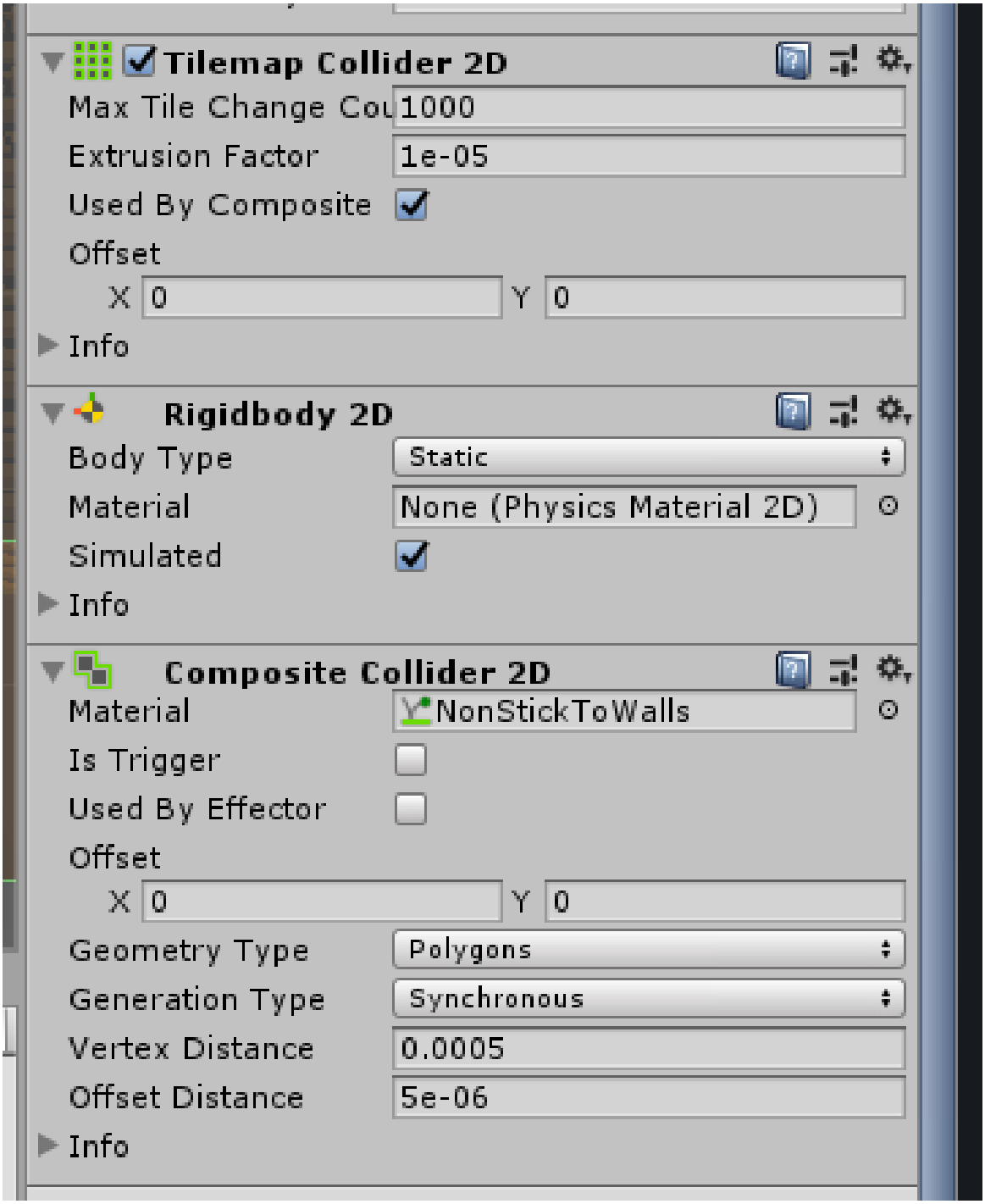
Projekt poziomu

Do stworzenia mapy, po której będzie poruszał się gracz wykorzystam specjalne narzędzie wbudowane w program Unity o nazwie `Tile Palette`. Dodawanie każdego elementu otoczenia dodany jako oddzielny obiekt wykorzystywałby o wiele więcej zasobów programu, tworzyłby niepotrzebną listę elementów w zakładce Hierarchy a przede wszystkim, każdy obiekt trzeba by było optymalizować oddzielnie.

Aby móc pracować z wyżej wymienionym narzędziem, musimy stworzyć nowy plik `Tilemap`, który znajdziemy w zakładce `2D Object`. Tile Palette importuje wybrane przez nas pliki graficzne i układa je na swojej mapie. Zbudowana jest ona na siatce, która pomoże nam rysować otoczenie. Teraz możemy wybrać kafelek z mapy, który chcemy użyć, przechodzimy do zakładki tworzenia gry i rysujemy planszę porównywalnie łatwo do programu graficznego. Elementy ustawiane są z precyzją jednej kratki na specjalnie zobrazowanej siatce, dzięki czemu łatwo i szybko możemy stworzyć nowe elementy mapy. Do wyboru mamy szereg możliwości rysowania, takich jak pędzel, prostokąt, lub wypełnienie.

Aby nasza powierzchnia działała jako podłoże, przez które gracz nie może przejść musimy nadać naszym obiektom komponent `Timemap Collider 2D`. Możemy jednak zauważyć, że każdy narysowany kafelek ma osobny obszar kolizyjny. Może to powodować, że postać będzie miała problem przechodzić między nimi i będzie się zatrzymywać. Aby temu zapobiec, zaznaczamy opcję `Used By Composite` i dodajemy komponent `Composite Collider 2D` wraz z `Rigidbody2D`, który jest niezbędny do złączenia kolizji obiektów. W `Rigitbody2D` wybieramy `Body Type` jako `Static`, aby plansza nie podlegała prawu grawitacji. Zaznaczamy również opcję `Simulated`. Jeżeli chcemy, aby nasze elementy posiadały w sobie jakiś materiał[1], na przykład materiał zapobiegający trzymaniu się ścian pionowych, możemy dodać go w polu `Material` w komponencie `Composite Collider 2D`. Teraz wszystkie obszary kolizyjne powinny złączyć się w jeden dzięki czemu unikniemy niepotrzebnego zatrzymywania się postaci oraz innych błędów wynikających ze źle zaimplementowanej kolizji.

[1] Materiałem w Unity jest sposób w jaki dwuwymiarowa tekstura jest odwzorowywana w ogólności na trójwymiarowy obiekt. W szczególności wszystkie obiekty dwuwymiarowe mogą być w Unity traktowane jako trójwymiarowe. W naszym kontekście zwykle materiał to po prostu tekstura.



Rysunek 3.1. Poprawne ustawienie kolizji podłoża stworzonego z Tile Palette.

Rozdział 4

Implementacja gry

Najbardziej rozbudowanym skryptem w mojej pracy jest kontroler `Player`. To postać, którą porusza gracz jest najważniejsza, jest ciągle w centrum uwagi i to ona ma styczność z największą ilością obiektów. Dzięki komponentowi `RigitBody2D`, nasza postać jest traktowana jako obiekt dynamiczny i sterowalny a `BoxCollider2D` pozwala nam ustawić kolizję między graczem a podłożem, przez co postać stoi na stworzonej przez nas platformie. Żeby nadać ruch postaci, musimy napisać skrypt, który będzie przesuwał naszym graczem podczas wciskania klawiszy.

W momencie stworzenia pustego skryptu C# w Unity, automatycznie zostanie wygenerowana nowa klasa z dwiema funkcjami:

```
public class Player : MonoBehaviour
{
    void Start()
    {

    }

    void Update()
    {

    }
}
```

Są to specjalne funkcje zdarzeniowe, które pomogą nam w implementacji naszego projektu. Nazwy funkcji nie można zmienić na dowolną, ponieważ wtedy nie będzie spełniać swojego zadania. Każda z nich dzieli się na grupy, które są wykonywane w określonym czasie i o określonym priorytecie. Mamy do wyboru funkcje odpowiadające za edytor, które na przykład realizują się w momencie włączenia się gry (`Start`). Następne w kolejności są funkcje związane z fizyką oraz kolizjami (`FixedUpdate`, `OnCollision`). Niżej w hierarchii znajdują się funkcje odpowiadające za logikę gry, renderowanie sceny lub graficzny interfejs. Ważne jest zapoznanie się z nimi, aby móc w pełni wykorzystać potencjał silnika. Wiemy wtedy na przykład, że funkcja `FixedUpdate` wykonuje się szybciej niż Update i jest wykonywana w stałych odstępach czasu.

Deklarujemy sobie 2 zmienne prywatne w klasie `Player`, które będą pobierać wartości z komponentów Unity do naszego obiektu:

```
private Animator anim;
private Rigidbody2D rb2d;
```

Obie zmienne trzeba przypisać do danych im narzędzi z poziomu kodu.

```
private void Start()
{
    rb2d = gameObject.GetComponent<Rigidbody2D>();
    anim = gameObject.GetComponent<Animator>();
}
```

Teraz w momencie uruchomienia aplikacji nasze obie zmienne będą miały przypisane przez nas parametry. Tak dodanymi komponentami możemy operować z poziomu kodu. Deklarujemy w kodzie zmienne:

```
public float speed = 50f;
public float maxspeed = 3;
```

Aby dodać ruch postaci, wykorzystamy funkcję warunkową `FixedUpdate`:

```
 private void FixedUpdate()
{
    var inputH = Input.GetAxis("Horizontal");
    rb2d.AddForce(Vector2.right * speed * inputH);

    if (rb2d.velocity.x > maxspeed) //ogarniczenie predkosci w prawo
        rb2d.velocity = new Vector2(maxspeed, rb2d.velocity.y);
    if (rb2d.velocity.x < -maxspeed) //ogarniczenie predkosci w lewo
        rb2d.velocity = new Vector2(-maxspeed, rb2d.velocity.y);
}
```

Jak widać, użyłem tutaj funkcji GetAxis(). Do pobrania wciśnięcia klawisza możemy posłużyć się również funkcjami GetButton() albo GetKey(), jednakże obie te wymienione funkcje zwracają wartość boolowską: przycisk jest wciśnięty lub nie. GetAxis() daje nam w wyniku liczbę w formacie float w granicy od -1 do 1. Możemy ustawić również z poziomu Unity alternatywne przyciski, zmieniające wartość na pozytywną i negatywną, ustawić czułość oraz martwy punkt ruchu przy niskim współczynniku wciśnięcia. W przyszłości można poszerzyć sterowanie postacią do użycia kontrolera, wtedy poruszanie się joystickiem będzie o wiele bardziej precyzyjne. Jeżeli przycisk odpowiadający za kierunek horyzontalny zostanie naciśnięty, postać przemieszcza się w odpowiednim dla niego kierunku.

Aby obrazek postaci mógł skierować się w lewą stronę, dodajemy w funkcji `Update` następujące warunki:

```
if (Input.GetAxis("Horizontal") < -0.1f)
    transform.localScale = new Vector3(-2, 2, 2);
if (Input.GetAxis("Horizontal") > 0.1f)
    transform.localScale = new Vector3(2, 2, 2);
```

Dzięki temu nasza postać będzie obracała obrazkiem w kierunku poruszania się obiektu.

Postać w grze platformowej powinna również mieć możliwość skoku. Jednak zanim zaprogramujemy skok, warto stworzyć nowy BoxCollider na nogach naszego bohatera. Gracz musi dotykać podłoża, w celu wykonania skoku. Jeżeli ta funkcja nie będzie spełniona, główny bohater będzie mógł skakać w powietrzu w nieskończoność. Jako, że jest to niepożądana przez

nas mechanika, deklarujemy zmienną boolowską, czyli taką, która odpowiada nam na pytanie "prawda/fałsz".

```
public bool grounded;
```

Dzięki temu będziemy mogli określić, czy postać ma kolizję z podłożem. Tworzymy nowy skrypt o nazwie "GroundCheck". Deklarujemy sobie zmienną gracza:

```
private Player player;
```

W funkcji Start, dodajemy przypisanie obiektu gracza do componentu:

```
private void Start()
{
    player = gameObject.GetComponentInParent<Player>();
}
```

Tworzymy teraz 3 funkcje poza Startem:

```
private void OnTriggerEnter2D(Collider2D collision)
{
    player.grounded = true;
}

private void OnTriggerStay2D(Collider2D collision)
{
    player.grounded = true;
}

private void OnTriggerExit2D(Collider2D collision)
{
    player.grounded = false;
}
```

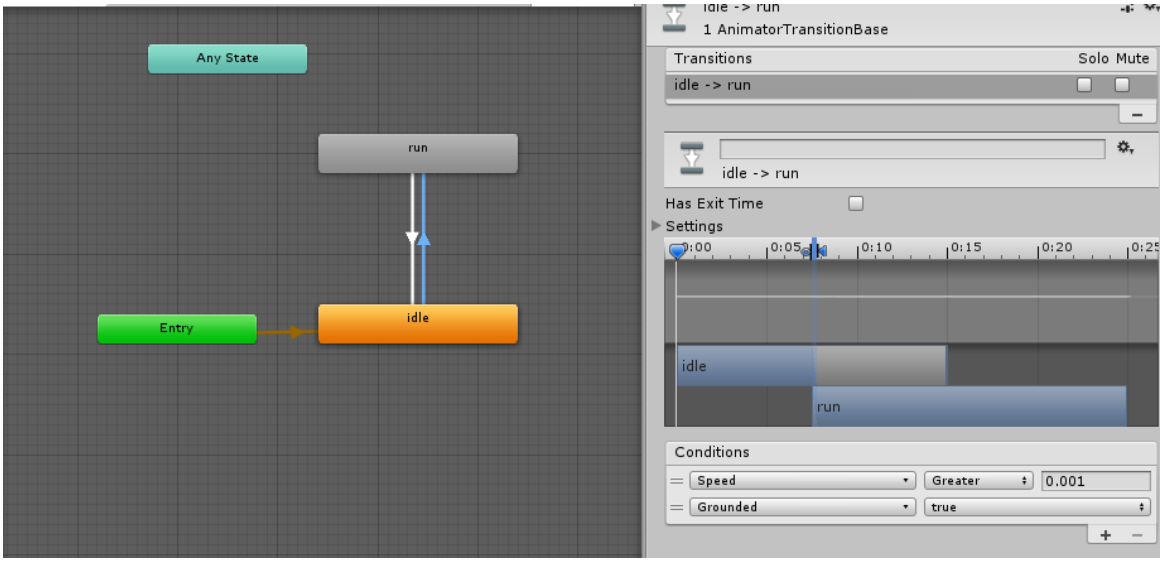
Każda z trzech kondycji sprawdza, czy dodatkowy blok kolizyjny dotyka podłoża i w jaki sposób ma to być interpretowane przez program. Podczas wejścia w kolizje i pozostanie w niej, kod zwróci nam wartość uziemienia gracza jako prawdę, gdy jednak zejdziemy z platformy i nasz czujnik nie będzie w kontakcie z podłogą, otrzymamy w zmiennej wartość negatywną.

Żeby animacje wyświetlały się pomyślnie, ustalimy zmienne dla animatora. Dzięki nim, będziemy mogli określić, która animacja będzie wyświetlana. Wracamy do skryptu `Player` i w funkcji `Update` deklarujemy:

```
anim.SetBool("Grounded", grounded);
anim.SetFloat("Speed", Mathf.Abs(rb2d.velocity.x));
```

Funkcja Mathf.Abs sprawi, że w wyniku będziemy otrzymywać wartość bezwzględną, co pomoże nam lepiej określić warunki odtwarzanej aplikacji. W zakładce animator tworzymy dwie zmienne tego samego typu, jak były zadeklarowane w skrypcie.

Po włączeniu gry możemy obserwować jak wartość Speed zmienia się podczas poruszania się a program wykrywa, czy postać stoi na innym obiekcie. Przeciągając animację biegu do naszego Animatora, dodajemy go na planszę animacyjną. Możemy stworzyć warunek przejścia pomiędzy dwoma animacjami: Znajdziemy tutaj opcje decydujące o tym jakie warunki muszą



Rysunek 4.1. Tworzenie warunku przejścia animacji.

zostać spełnione do zmiany animacji, jak szybkie jest przejście między nimi albo czy animacja musi zostać wykonana w całości, zanim przejdzie do innej. Aby z animacji bezruchu przejść w animację poruszania się, wybieramy w zakładce `Conditions` nasze zmienne i ustawiamy im odpowiednie parametry. Postać musi znajdować się na ziemi oraz wartość jej prędkości musi być większa niż 0.001. Gdy włączymy aplikację, nasza postać zmieni swoją animację, lecz nie wraca do animacji bezruchu przy zerowej prędkości. Trzeba pamiętać w tym przypadku również o "drodze powrotnej" dla animacji i warunkach jej przejścia.

Gdy dodaliśmy warunki zmiany naszej zmiennej boolowskiej `grounded`, możemy przejść do implementacji skoku postaci. Dodajemy nową zmienną do klasy Player:

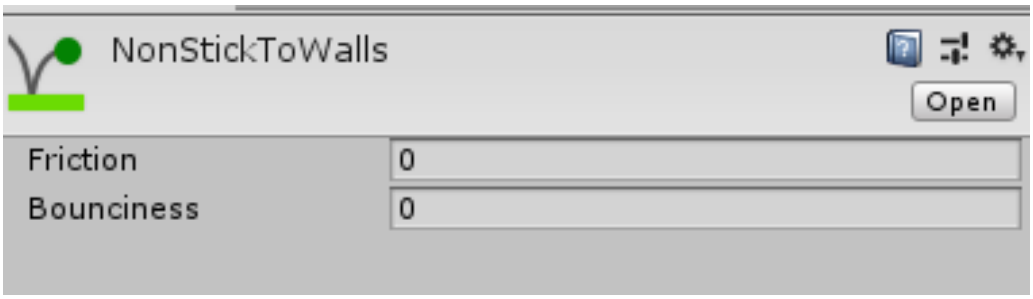
```
public float jumpPower = 150f;
```

Tutaj można wpisać dowolną moc z jaką nasza postać ma skakać. W funkcji Update, pod kontrolerem ruchu horyzontalnego możemy wpisać dalej:

```
if (Input.GetButtonDown("Jump"))
{
    if (grounded) //Zwykly skok
    {
        rb2d.AddForce(Vector2.up*jumpPower);
    }
}
```

Jeżeli przycisk skoku został wciśnięty program sprawdzi czy zachodzi kolizja między postacią a podłożem, czyli postać stoi. Jeżeli tak, obiektowi z komponentem RigitBody2D dodajemy siłę, z którą wystrzeli naszą postać w górę.

Po uruchomieniu gry możemy zauważyć, że nasza postać porusza się i wykonuje skok. Problemem, który możemy zauważyć to przyklejanie się postaci do ściany, gdy będąc w powietrzu gracz będzie z nią w kolizji trzymając przycisk poruszania się. Aby temu zapobiec, musimy stworzyć `Physics Material 2D`.



Rysunek 4.2. Opcje Physics Material 2D.

Możemy zmienić tarcie oraz odbijanie się od materiału, które będzie przypisane do określonej fizyki obiektu. Ustawiamy wszystkie wartości do 0 i przypisujemy do naszych elementów planszy. Teraz można zauważyć, że postać znajdująca się na powierzchni bez tarcia nie zatrzymuje się. Aby to naprawić musimy napisać warunek, który będzie wytrącał prędkość postaci w celu zatrzymania się, gdy przycisk poruszania się nie będzie wciśnięty. Przechodzimy zatem do skryptu `Player`:

```
private void FixedUpdate()
{
    Vector3 easeVelocity = rb2d.velocity;
    easeVelocity.y = rb2d.velocity.y;
    easeVelocity.z = 0.0f;
    easeVelocity.x *= 0.75f;

    if (grounded) rb2d.velocity = easeVelocity;
}
```

Tworzymy wektor o nazwie `easeVelocity`, gdzie zostaje przypisana wartość prędkości obiektu z `RigitBody2d`. Zależy nam jedynie na zmianie wartości w płaszczyźnie X, dlatego wartość naszej prędkości mnożona jest co 0.75 co każdą klatkę. Warunek z uziemieniem jest potrzebny, ponieważ nie chcemy spowalniać postaci podczas skoku.

Dla większej mobilności, postać będzie mogła wykonać dodatkowy skok w powietrzu. Deklarujemy sobie funkcję boolowską:

```
public bool candoubleJump;
```

Wtedy cała funkcja odpowiadająca skok, powinna wyglądać następująco:

```
if (Input.GetButtonDown("Jump"))
{
    if (grounded) //Zwykly skok
    {
        rb2d.AddForce(Vector2.up * jumpPower);
        candoubleJump = true;
    }
    else
    {
        if (candoubleJump) //Doublejump
        {
            candoubleJump = false;
            rb2d.velocity = new Vector2(rb2d.velocity.x, 0);
            rb2d.AddForce((Vector2.up * jumpPower / 2 )+
            (Vector2.up * jumpPower / 4));
        }
    }
}
```

Funkcja jest uruchamiana przy wciśnięciu klawisza skoku, czyli spacji. Jeżeli występuje kolizja między graczem a podłogą, postać wykonuje skok a zmienna `candoubleJump` przyjmuje wartość prawdziwą. Jeżeli ponownie wciśniemy spację, funkcja omija pierwszą instrukcję warunkową ze względu na to, że nie stykamy się z podłożem. Program sprawdza, czy wartość zmiennej `candoubleJump` jest prawdziwa i jeżeli jest, ustawiamy ją na fałszywą w celu zniwelowania ponownych skoków. Dodajemy również bohaterowi moc, która kieruje go w górę. W moim przypadku drugi skok jest równy 3/4 wartości pierwszego skoku.

Kamera jest nierozłączym elementem rozgrywki. To ona pokazuje nam całą scenerię gry, dlatego dobre ustawienie kamery jest bardzo ważne w każdej projekcie. Objekt `Kamera` jest automatycznie tworzony przy robieniu nowego projektu i dodawany jest do niego komponent kamery. Ustawiając go według naszych preferencji, możemy przejść teraz do poruszania kamerą. Stworzymy teraz skrypt, odpowiadający za podążanie kamery za graczem.

Deklarujemy 4 zmienne:

```
public class CameraFollow : MonoBehaviour
{
    public GameObject player;
    public float smoothTimeX;
    public float smoothTimeY;
    private Vector2 velocity;
}
```

Pierwsza zmienna będzie wskazywać naszego gracza. Dwie kolejne typu float odpowiadają za wygładzanie ruchu kamery w pionie i poziomie a prywatny wektor będzie przesuwał naszą kamerę. W funkcji `FixedUpdate` ustawiamy pozycję poziomą i pionową:

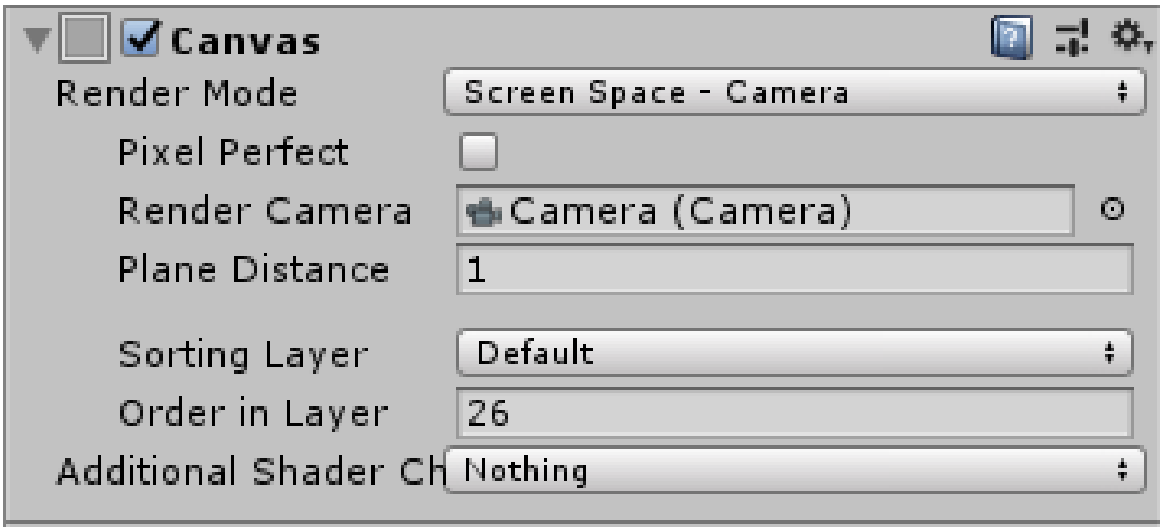
```
var posX = Mathf.SmoothDamp
  (transform.position.x, player.transform.position.x,
    ref velocity.x, smoothTimeX);

var posY = 0.25f + Mathf.SmoothDamp
  (transform.position.y, player.transform.position.y,
    ref velocity.y, smoothTimeY);

transform.position = new Vector3(posX, posY, transform.position.z);
```

Według tych ustawień kamera będzie wycentrowana na naszym bohaterze. Dla lepszego efektu, chciałbym żeby kamera znajdowała się lekko nad bohaterem, dlatego do zmiennej `posY` dodałem wartość 0.25 w typie float. Możemy zapisać skrypt i zabrać się za stworzenie przycisku pauzy.

Moja funkcja zatrzymania będzie wywoływana klawiszem "Esc". Będzie powodować, że wszystkie obiekty znajdujące się na scenie zostaną zatrzymane a gracz dostanie komunikat o tym, że gra jest zatrzymana. W polu `Hierarchy` wybieramy nowy obiekt który chcemy dodać. Przechodzimy do zakładki `UI`, z której wybieramy opcję `Canvas`. To będzie nasze płótno główne, którego zadaniem będzie przetrzymywać wszystkie elementy odpowiadające za interfejs graficzny, takie jak licznik zdrowia, ilość zebranych przedmiotów albo wyświetlanie okna pauzy. Nazwiemy je `Huds`.



Rysunek 4.3. Opcje komponentu Canvas.

Z listy `Render Mode` wybieramy `Screen Space - Camera` i przeciągamy naszą kamerę do pola `Render Camera`, aby za każdym razem płótno było wyświetlane w miejscu które obejmuje kamera. Tworzymy kolejny `Image` o nazwie `PauseUI` i wstawiamy to jako dziecko naszego płótna. W pierwszej kolejności chcemy, aby przy włączeniu pauzy gra miała wizualnie efekt nieaktywnej. Najprostszym sposobem aby to uzyskać, jest delikatna maska szarości na ekranie.

Dodany obraz powinien automatycznie posiadać w komponencie załadowany szary obrazek. W zupełności taki prosty obrazek nam wystarczy, wystarczy zmienić mu przeźroczystość oraz kolor. W opcjach komponentu znajdziemy tam wszystkie ustawienia. Potem, możemy dodać tekst za pomocą obiektu `Text` z zakładki `UI`, który przypisujemy pod `PauseUI`.

Teraz widok naszej gry będzie prezentował się mniej więcej tak:

Rysunek 4.4. Widok z kamery.

Jak można zauważyć przy uruchomieniu gry nasze menu pausy jest aktywne, mimo że gra cały czas jest w trybie niezapauzowanym. Tworzymy nowy skrypt, który nazwiemy `PauseMenu`. Tworzymy w nim 2 zmienne:

```
private bool paused;
public GameObject PauseUI;
```

Pierwsza zmienna wartości bool będzie określać, czy gra powinna być zatrzymana. Druga zmienna będzie przyjmować wartość płótna, którą mu przypiszemy. W funkcji `Start` wpisujemy:

```
PauseUI.SetActive(false);
```

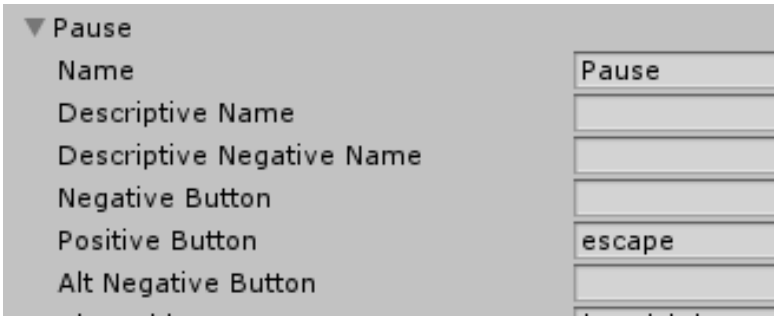
Dzięki temu, menu pauzy nie będzie wyświetlać się w momencie włączenia gry. W funkcji Update używamy 3 funkcji warunkowych:

```
if (Input.GetButtonDown("Pause")) paused = !paused;
if (paused)
    {
       PauseUI.SetActive(true);
       Time.timeScale = 0;
    }
if (!paused)
    {
```

```
        PauseUI.SetActive(false);
        Time.timeScale = 1;
    }
```

Skrypt sprawdza, czy przycisk `Pause` został użyty i jeżeli tak, ustawia wartość `paused` jako przeciwną. W tym momencie nie jest on nigdzie przypisany. Później sprawdzana jest wartość funkcji boolowskiej. Jeżeli zmienna `paused` jest ustawiona jako prawda, nasze płótno wyświetla się na ekranie, a czas gry zostaje zmieniony na 0. Oznacza to, że każdy obiekt pozbawiony zostaje ruchu i fizyka gry zostaje chwilowo wyłączona. Jeżeli zmienna jest jako fałszywa, nasz interfejs będzie nieaktywny, a czas gry z powrotem otrzyma wartość 1.

Przejdźmy do ustalenia przycisku zatrzymania gry. W programie Unity przechodzimy w zakładkę `Edit` i wybieramy opcję `Project Settings`. Po lewej stronie okienka wybieramy `Input`. Tutaj znajdują się wszystkie przypisane przyciski w naszym projekcie. Teraz wystarczy zduplikować ostatnią zmienną i nadać jej odpowiednie parametry:



Rysunek 4.5. Widok z kamery.

Teraz zostało nam dodać skrypt do naszej kamery i ustawić `PauseUI` w nowo dodanym komponencie. Teraz, przy wciśnięciu klawisza `ESC` nasza gra powinna pauzować się, a komunikat o zatrzymanej grze wyświetlać na ekranie.

W celu ułatwienia ukończenia poziomu gracz będzie mógł popełnić określoną ilość błędów, zanim gra zostanie zakończona niepowodzeniem. W tym celu stworzymy system zdrowia reprezentowany graficznie.

Przechodzimy do skryptu `Player`, implementujemy zmienne:

```
public int curHealth;
public int maxHealth = 3;
```

Pierwsza zmienna będzie przechowywać nasze obecne zdrowie. Druga będzie ogranicznikiem, która spowoduje, że nasze zdrowie nie będzie większe niż ta wartość. Do funkcji `Start` ustawiamy warunek:

```
curHealth = maxHealth;
```

Nasze zdrowie na początku gry będzie mieć wartość maksymalną.

```
if (curHealth > maxHealth) curHealth = maxHealth;
if (curHealth <= 0) Death();
```

Teraz, gdy nasze zdrowie przekroczy ustalony limit, zostanie zredukowane do maksymalnej wartości. Drugi warunek powoduje wywołanie funkcji `Death`, którą będziemy teraz tworzyć.

```
private void Death()
{
    SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex);
}
```

Zadaniem tej funkcji jest przeładowanie naszej sceny, co spowoduje reset poziomu. Teraz, kiedy nasze zdrowie będzie równe 0, będziemy musieli przechodzić poziom od początku.

W tym momencie licznik zdrowia dla gracza jest niewidoczny. Zaimportujemy teraz grafikę, która będzie odzwierciedlać zdrowie gracza na płótnie.

Rysunek 4.6. Wyświetlacz wartości zdrowia.

Zamiast paska zdrowia zdecydowałem się na stworzenie obiektu, podzielonego graficznie na 3 części. Suma niebieskich części serca odpowiada jednemu punktowi zdrowia. Gdy tracimy życie, jedna część zmienia kolor na szary.
Tworzymy nowe płótno i nazywamy je `HUD_HP`. Stworzenie nowego canvasa będzie nam potrzebne, ponieważ nasz licznik składa się z dwóch obrazków - jednego statycznego tła oraz serca, które zamierza zmieniać kolor zależnie od wartości zmiennej. Dodajemy do nowego płótna tło jako normalny obiekt i ustawiamy komponent `Rect Transform` do przemieszczenia go w lewy górny róg. Dodajemy teraz `Image`, który możemy znaleźć w zakładce UI. Ustawiamy teraz jako źródła obrazu nasze serduszko, odpowiednio skalujemy oraz przemieszczamy. Tworzymy nowy skrypt o nazwie `HUD` i deklarujemy zmienne:

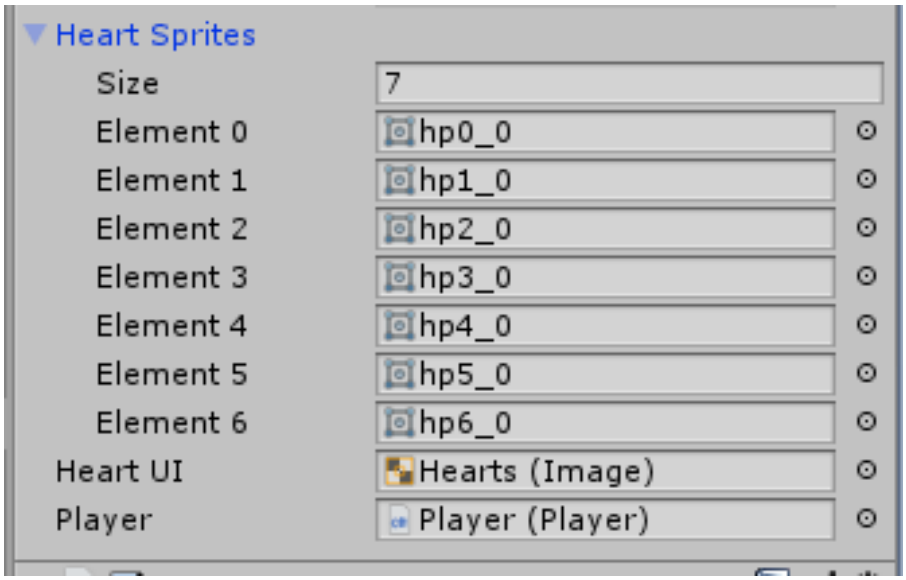
```
public Sprite[] HeartSprites;
public Image HeartUI;
public Player player;
```

Zmienna `HeartSprites` nie jest zwykłą zmienną wykorzystującą obrazek, a tablicą. Będziemy przechowywać w niej wszystkie grafiki reprezentujące nasz stan zdrowia. Druga zmienna przekazuje informację, który obiekt będzie miał zmieniane animacje a ostatnia wskazuje na naszego bohatera.

W funkcji `Update` wpisujemy:

```
HeartUI.sprite = HeartSprites[player.curHealth];
```

`player.curHealth` jest zmienną typu int. Oznacza to, że mając 2 zdrowia obrazkiem, który wyświetli się w obiekcie `HeartUI` jest `HeartSprites[2]`, czyli obrazek z tabeli z numerem indeksu 2. Zapisujemy skrypt i przypisujemy go jako komponent do naszej kamery.



Rysunek 4.7. Ustawienia statusu zdrowia.

Ustawiamy wielkość naszej tablicy w opcji `Size`, czyli ilość obrazów które mają być wyświetlane. W moim przypadku będzie to 7 mimo, że maksymalne zdrowie bohatera ustawiliśmy jako 3. Dzięki temu w późniejszych etapach po zebraniu jakiegoś przedmiotu możemy powiększyć limit. Wstawiamy teraz obrazy serduszek zachowując odpowiednią kolejność: w indeksie 0 znajduje się graficzny odpowiednik braku zdrowia, w 1 mamy jedno zdrowie i tak robimy we wszystkich przypadkach. Na koniec przeciągamy do skryptu naszego bohatera. Teraz po uruchomieniu gry możemy przejść do skryptów używanych przez naszego bohatera i zmienić mu ilość aktualnego zdrowia. Jak można zauważyć obrazki zmieniają się a gdy zdrowie ustawione zostanie na 0, gra zresetuje się.

Celem gracza będzie przejście przez poziomy, zbierając obiekty. W moim przypadku, będą to nakrętki. Po dodaniu nowego obiektu nadajemy mu animację, komponent `BoxCollider2D` oraz zaznaczamy opcję "OnTrigger". Dzięki temu, nasza postać nie będzie blokować się przy kolizji z nim, lecz będziemy mogli nadać akcję danemu obiektowi. Dodajemy tag o nazwie `PickUp` i ustawiamy go w naszym nowo utworzonym obiekcie. Tworzymy nowy skrypt `GameMaster`, który będzie przechowywał liczbę zebranych obiektów oraz wyświetlał je na ekranie.

```
public int nuts;
public Text Text;
```

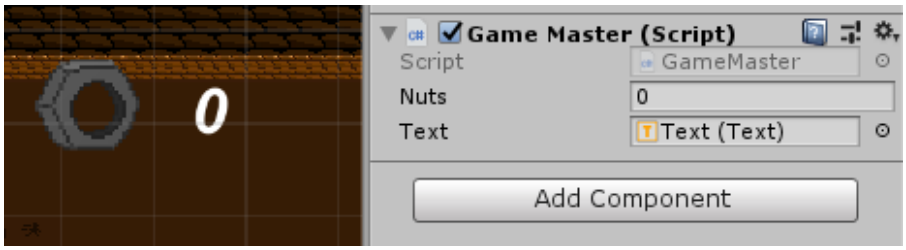
`Text` będzie wskazywał na tekst na płótnie, który ma wyświetlać ilość zebranych przez nas nakrętek. W funkcji `Update` wpisujemy:

```
Text.text = "" + nuts;
```

Zadaniem tej funkcji będzie wypisywanie wartości zmiennej typu integer `nuts` na polu tekstowym, który przypniemy do komponentu.

Tworzymy licznik na ekranie za pomocą nowego płótna, który będzie nazwany `HUD_Nut` w którym możemy dodać wybraną dla nas grafikę. Z menu `UI` wybieramy dodanie elementu `Text`, który wrzucamy pod naszym nowym canvasem i optymalizujemy go do naszej kamery.

Dodajemy pusty element, który nazwiemy `GameMaster` po czym przypisujemy mu komponent skryptu o tej samej nazwie.



Rysunek 4.8. Wygląd licznika nakrętek wraz z ustawieniem komponentu.

Po przypisaniu naszego tekstu w momencie włączenia aplikacji możemy zauważyć, że liczba na ekranie zmienia się kiedy zmienna będzie mieć różne wartości.

Teraz zajmiemy się dodawaniem punktów i kolizją między bohaterem. Przechodzimy do skryptu `Player`, deklarujemy zmienną odwołującą się do naszego obiektu `GameMaster`

```
public GameMaster gameMaster;
```

Tworzymy nową funkcję:

```
private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.CompareTag("PickUp"))
    {
        Destroy(collision.gameObject);
        gameMaster.nuts += 1;
    }
}
```

Funkcja będzie wykonana, gdy postać będzie miała kontakt z obiektem posiadającym komponent `BoxCollider2D` z włączoną opcją `IsTrigger`. Jeżeli obiekt który dotykamy posiada tag `PickUP`, wtedy zostaje on zniszczony, a nasza zmienna licząca zebrane nakrętki jest powiększana o 1.

Aby rozgrywka nie była była zbyt prosta, po planszy będą porozstawiane zostaną kolce, które będą zabierać graczowi zdrowie, kiedy będzie z nimi kolidował i powodować delikatny odrzut. Dodajemy je na naszej planszy i przechodzimy do pisania skryptu `Spikes`. Deklarujemy naszego gracza:

```
public Player player;
```

Potem, tworzymy nową funkcję kolizyjną:

```
    private void OnTriggerEnter2D(Collider2D col)
    {
        if (col.CompareTag("Player"))
        {
            player.Damage(1);
        }
    }
```

Teraz, gdy nasza przeszkoda zetknie się z obiektem posiadającym tag "Player", otrzymamy 1 punkt obrażeń.

Zaimplementujemy drzwi, które otwierają się dopiero gdy zbierzemy odpowiednią ilość nakrętek. Po stworzeniu obiektu i dodaniu mu tagu `Door`, tworzymy nowy skrypt o tej samej nazwie.

```
private Animator anim;
public GameMaster gameMaster;
public bool open;
public int value = 1;
```

Pierwsza zmienna będzie potrzebna do użycia komponentu `Animator`. Następnie, deklarujemy zmienną, która ze skryptu `GameMaster` będzie pobierać ilość posiadanych przez nas nakrętek. Dzięki temu, możemy później porównać, czy gracz zebrał odpowiednią ilość do otworzenia drzwi. Zmienna boolowska będzie odpowiadać za stan drzwi, dzięki której `Animator` będzie mógł je otworzyć. Zmienna `value` przechowuje wartość, do której będziemy przyrównywać ilość zebranych przez nas obiektów.

W funkcji `Start` ustawiamy:

```
anim = gameObject.GetComponent<Animator>();
open = false;
```

Przypisujemy obiekt do danej animacji, oraz ustawiamy wartość boolowską na fałszywą.

Teraz, aby otworzyć drzwi, potrzebujemy opisać akcję w funkcji `Update`.

```
anim.SetBool("open", open);
if(gameMaster.nuts >= value)
    {
        open = true;
    }
```

Ustawiamy animację `open` ze zmienną boolowską o tej samej nazwie. Funkcja sprawdza, czy ilość posiadanych przez nas obiektów jest większa bądź równa wartości potrzebnej do otworzenia drzwi. Jeżeli warunek został spełniony, zmienna `open` zmienia wartość na prawdziwą. Przypisujemy skrypt naszemu stworzonemu z obiektu drzwi i ustawiamy optymalną wartość po której otworzą się.

W Animatorze ustawiamy pustą animację początkową. Ustalamy warunek przejścia na otwierające się drzwi przy pomocy zmiennej boolowskiej `open`.

Rozdział 5

Testowanie

Gdy gra została już zaimplementowana, należy sprawdzić czy wszystkie elementy działają poprawnie i odpowiednio ze sobą współgrają. Ze względu na prostotę projektu nie zostały zaimplementowane testy jednostkowe a wszystkie błędy i niepoprawności kodu były sprawdzane przeze mnie w trakcie tworzenia gry.

Wykonałem jednak testowanie tzw. metodą czarnej skrzynki, gdyż w przypadku gier duży nacisk kładziony jest na interakcję aplikacji z użytkownikiem. W tym celu poprosiłem również o opinię osoby trzecie, które nie miały udziału w tworzeniu projektu, żeby mogły określić swoje pierwsze wrażenia związane z grą. Poprosiłem również o informację, jakie elementy powinny zostać zmienione oraz czy według nich można byłoby umilić i wydłużyć rozgrywkę.

`Jacek`
`Plusy`: Ładnie narysowana postać
`Minusy`: Niektóre kolizje są źle zoptymalizowane
`Własne sugestie`: Dodanie mocy skoku w zależności od przytrzymania klawisza

`Marcin`
`Plusy`: Klimatyczna muzyka
`Minusy`: Odrzut przy kolizji z kolcami jest zbyt duży
`Własne sugestie`: Brak

`Artur`
`Plusy`: Animacje postaci są ciekawie narysowane
`Minusy`: Animacje czasami się psują
`Własne sugestie`: Dłuższy etap

`Michał`
`Plusy`: Sterowanie postacią
`Minusy`: Brak
`Własne sugestie`: Więcej efektów dźwiękowych na przykład przy zbieraniu obiektów

Rozdział 6

Podsumowanie

Gra platformowa, która była tematem mojej pracy inżynierskiej, została ukończona. Nauczyłem się obsługiwać program Unity i wykorzystywać jego elementy w programowaniu z językiem C#. Podniosłem swoje kompetencje w rysowaniu obiektów pikselowych oraz korzystaniu z programu do przesyłania danych GitBash. Podczas pracy nad tym projektem udało mi się dowiedzieć wielu rzeczy odnośnie środowiska, które będą użyteczne i pomocne podczas tworzenia kolejnych konceptów. Podczas implementacji nowego projektu będę próbował naprawić błędy związane z animacjami odgrywanymi przez bohatera, kolizją między obiektami oraz będę miał na celu jeszcze bardziej urozmaicić rozgrywkę.

Bibliografia

[1] GucioDevs, *Youtube* `https://www.youtube.com/channel/UC4MDTn6WkqvBfBW_4cFZp2g`, dostęp online: 03.01.2020.

[2] *Dokumentacja Unity* `https://docs.unity3d.com/Manual/index.html`.

[3] Jeremy Gibson Bond, *Projektowanie gier przy użyciu środowiska Unity i języka C#. Od pomysłu do gotowej gry, wydanie II*, Helion 2018.

[4] Jeremy Tony Gaddis, *Visual C# dla zupełnie początkujących. Owoce programowania. Wydanie IV*, Helion 2019.