

# Sieciowa fabularna gra komputerowa - The Trinity

Kleina Mateusz, Podlawski Adrian

12 kwietnia 2017

## Spis treści

<b>1 Wstęp i opis</b>	<b>1</b>
1.1 Porównanie dostępnych rozwiązań . . . . .	1
<b>2 Projektowanie oraz przygotowanie środowiska</b>	<b>1</b>
2.1 Projekt Terenu . . . . .	1
2.2 Projekt Postaci sterowanych przez gracza . . . . .	1
2.3 Projekt systemu multplayer . . . . .	1
2.4 Instalacja niezbędnych narzędzi . . . . .	1
<b>3 Implementacja</b>	<b>1</b>
3.1 Budowa terenu . . . . .	2
3.2 Tworzenie Postaci . . . . .	7
3.3 Ruch kamery . . . . .	8
3.4 Fizyka . . . . .	11
3.5 Animacje . . . . .	13
3.6 Poruszanie bohaterów . . . . .	14
3.7 Poruszanie postaci wrogów . . . . .	15
3.8 Nadanie bohaterom indywidualnych umiejętności . . . . .	17
3.9 Obrażenia . . . . .	17
3.10 Interakcja ze światem oraz NPC . . . . .	17

## **1 Wstęp i opis**

### **1.1 Porównanie dostępnych rozwiązań**

## **2 Projektowanie oraz przygotowanie środowiska**

### **2.1 Projekt Terenu**

### **2.2 Projekt Postaci sterowanych przez gracza**

### **2.3 Projekt systemu mulitplayer**

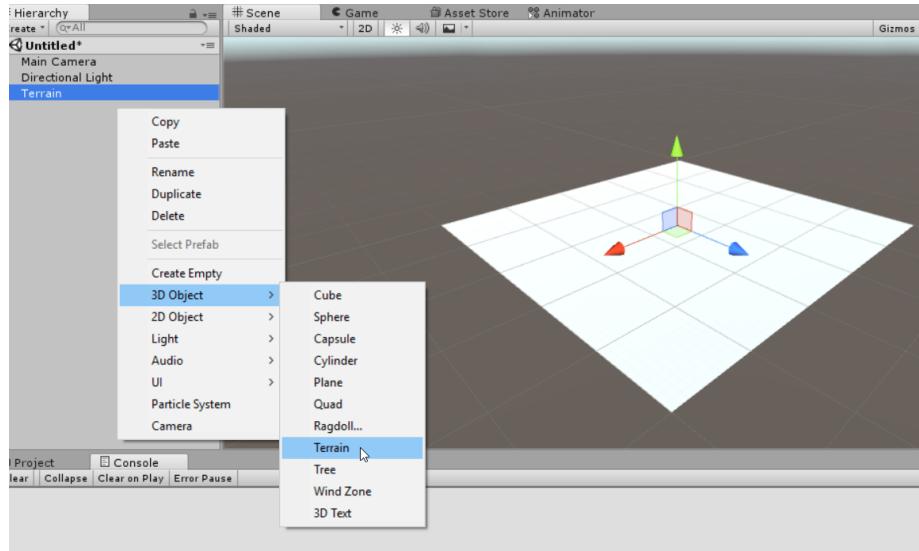
### **2.4 Instalacja niezbędnych narzędzi**

## **3 Implementacja**

### 3.1 Budowa terenu

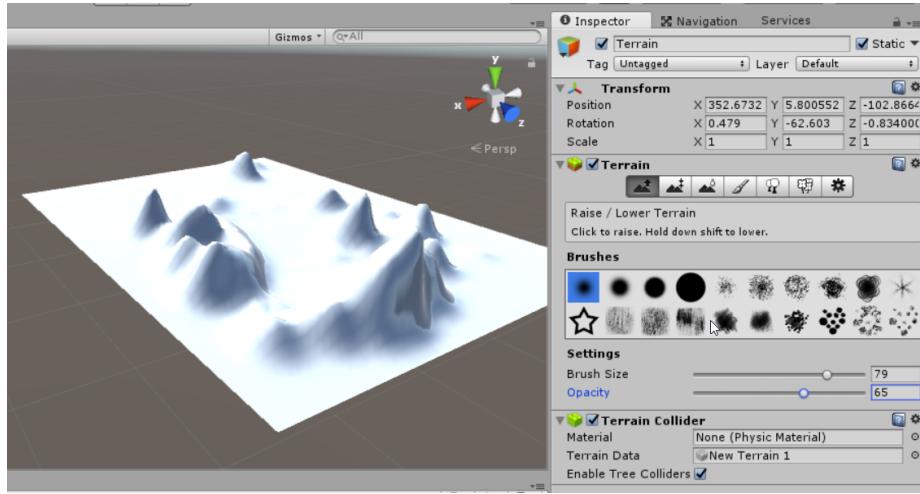
Budowa podstawowej mapy gry jest stosunkowo nieskomplikowanym procesem, dzięki czemu mogliśmy się tutaj skupić głównie na projektowaniu dróg, rozłożeniu obiektów i ksztaltowaniu mapy na potrzeby rozgrywki.

**Przygotowanie terenu** Do przygotowania terenu został użyty specjalnie do tego przeznaczony typ obiektu 3D o nazwie **Terrain**. Po utworzeniu takiego obiektu w hierarchii obiektów na ekranie ukaże się nieoteksturowana płaszczyzna.



Rysunek 1: Płaszczyzna terenu

**Formowanie kształtu** Na tak przygotowanej płaszczyźnie uformowane zostały nierówności przy użyciu palety narzędzi terenu. Używając opcji **Raise / Lower Terrain** utworzone zostały wypiętrzenia nadające kształt mapie gry. Charakter wypiętrzeń dostosowany został używając odpowiedniego pędzla z panelu **Brushes**, natomiast promień zniekształceń oraz siła efektu za pomocą parametrów kolejno **Brush Size** oraz **Opacity**.

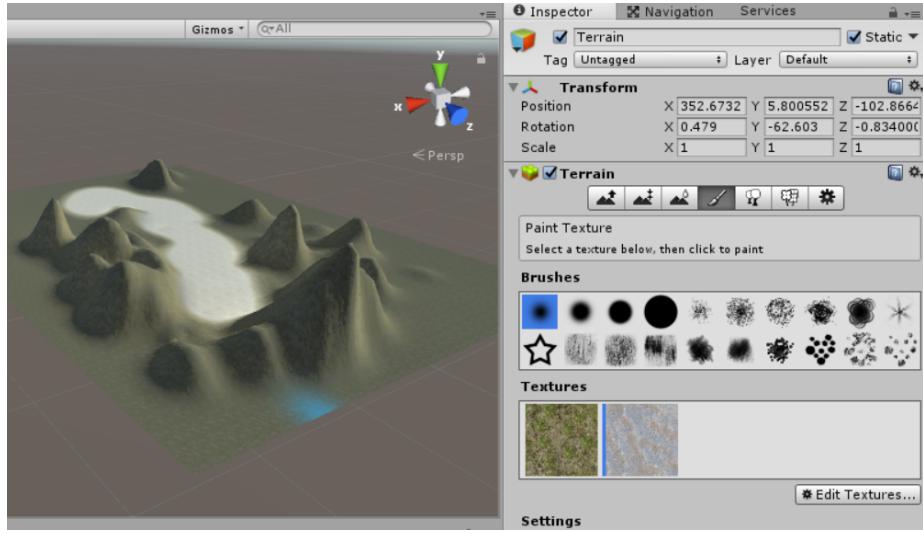


Rysunek 2: Modelowanie nierówności terenu

**Nakładanie tekstur** Kolejnym krokiem było utworzenie tekstury służącej do nadania naszemu terenowi koloru oraz faktury. Do tego celu pobrane zostały odpowiednie tekstury z wbudowanego sklepu assetów [Asset Store](#). Assety są rodzajem pakietów zawierających różnorakie obiekty, skrypty oraz tekstury, dostępne do pobrania z serwerów Unity.

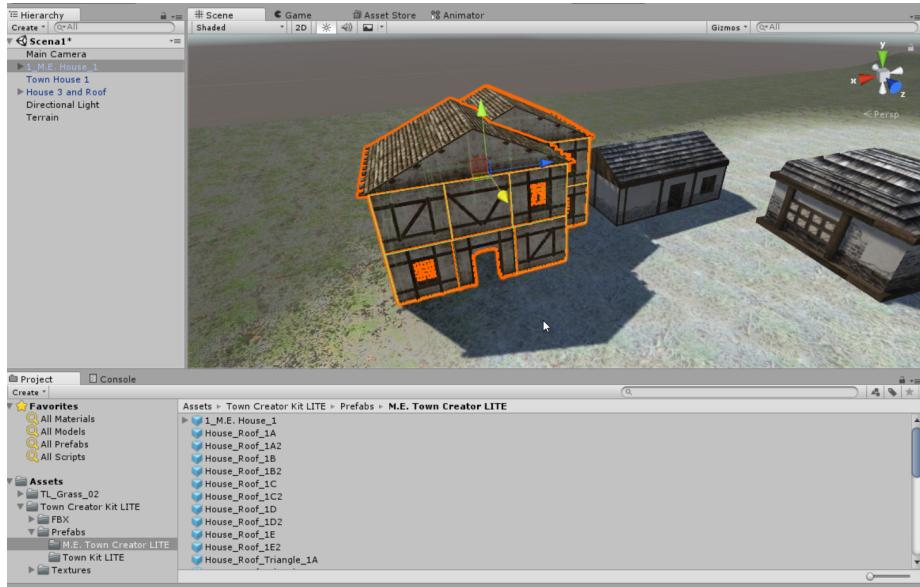
Po pobraniu odpowiedniej tekstury trawy, zostaje ona zainportowana do folderu [Assets](#) znajdującego się w głównym katalogu projektu.

Po wybraniu narzędzia [Paint Texture](#) ukazuje się panel [Textures](#) pozwalający na skonfigurowanie używanej przez narzędzie tekstury. Znajduje się tam przycisk [Edit Textures...](#) po kliknięciu którego otwiera się okno konfiguracyjne tekstury pozwalające na wybór tekstury podstawowej oraz tekstury przechowującej dane o chropowatościach. Po skonfigurowaniu tekstur i nałożeniu ich na teren, całość prezentuje się następująco.



Rysunek 3: Nakładanie tekstuur na mapę terenu

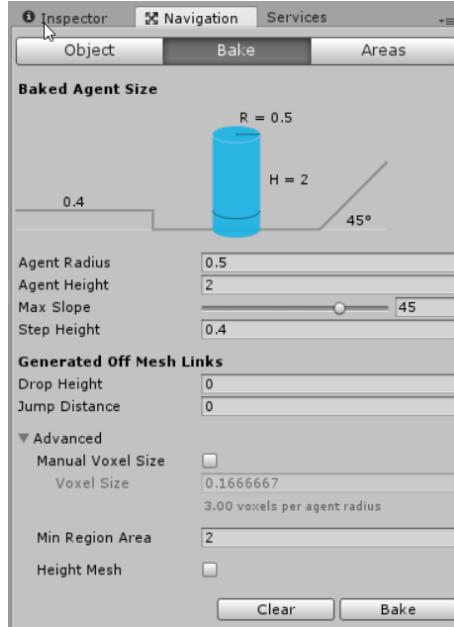
**Rozmieszczanie obiektów** Kolejnym krokiem w tworzeniu świata gry było umieszczenie na mapie sprefabrykowanych wcześniej obiektów (tzw. *Prefabs*, o tym później). Wszystkie obiekty użyte w pracy są dostępne za darmo w bibliotece obiektów Unity ([Asset Store](#)). Po pobraniu, obiekty znajdują się w podfolderze **Prefabs** zainstalowanej paczki. Obiekty umieszcza się na mapie metodą przeciągnij-upuść, dostosowując ich koordynaty używając narzędzi transformacji dostępnych w pasku narzędzi znajdującym się w górnej części okna edytora.



Rysunek 4: Umieszczanie obiektów na mapie

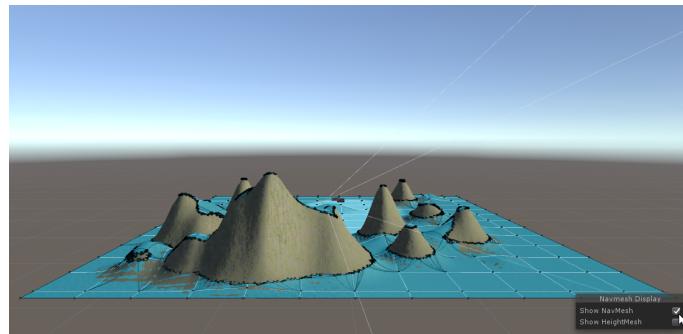
**Oddziaływanie terenu na inne obiekty** Kluczowym elementem tworzenia mapy świata gry są takie elementy jak oddziaływanie na postacie ograniczeń nachylenia terenu typu wzgórza, drzewa, czy budynki, uniemożliwiających dostanie się w niektóre miejsca. Unity w celu uproszczenia obliczeń posiada możliwość wygenerowania uproszczonej mapy dróg (tzw. *NavMesh*) na bazie modelu terenu, pozwalającej na dynamiczne omijanie przeszkód przez wroga (o tym później w sekcji 3.7 na stronie 15).

Aby utworzyć taką mapę, należy przejść do zakładki **Navigation**, gdzie w panelu **Bake** znajduje się lista parametrów dotyczących maksymalnego kąta nachylenia terenu, czy maksymalnej wysokości uskoku, którą obiekty sterowane przez komputer mogą pokonać.



Rysunek 5: Parametry mapy dróg

Po ustaleniu parametrów i kliknięciu przycisku **Bake**, mapa zostaje wygenerowana, natomiast w oknie widoku sceny, obszary dostępne do przemierzania oznaczone zostają niebieskim kolorem. Operację można powtarzać do uzyskania optymalnych efektów.



Rysunek 6: Poprawnie wygenerowany *NavMesh*

Tak przygotowana mapa posłużyła nam do projektowania dalszej części gry.

### **3.2 Tworzenie Postaci**

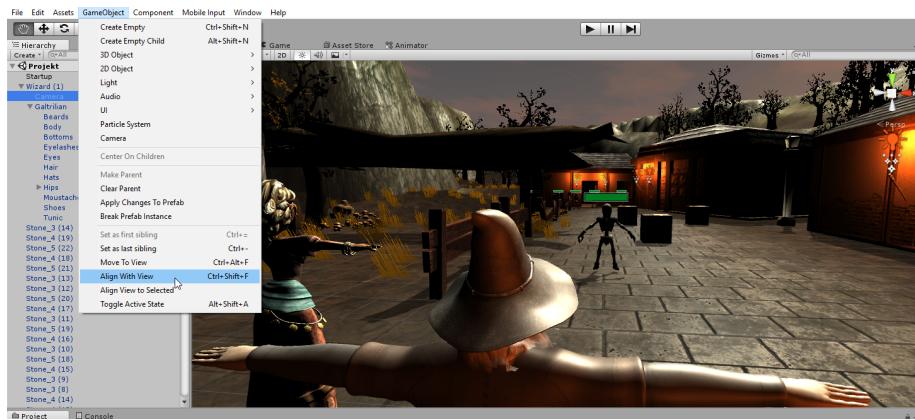
**« TO MIEJSCE WYPEŁNIA ADRIAN »**

### 3.3 Ruch kamery

**Pozycja kamery** Widok w grze jest trzecioosobowy, kamera obejmuje zarówno widzialny obszar jak i samego gracza. Pod tym względem jest to gra TPP (*Third Person Perspective*). Przykładowymi grami tego typu są bardzo znane produkcje, takie jak seria *Wiedźmin*, *Tomb Raider*, czy *GTA*.

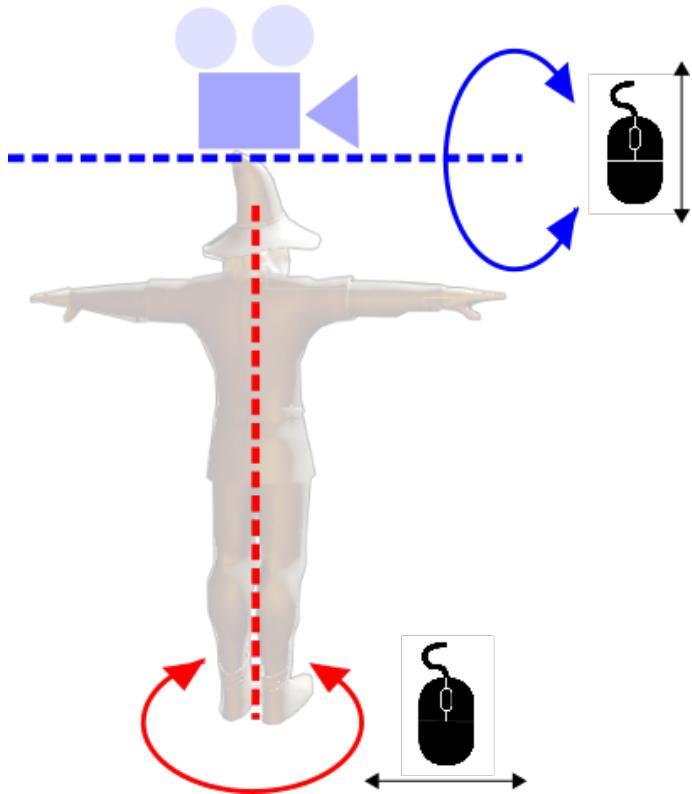
Aby uzyskać efekt kamery podążającej za graczem, obiekt kamery powinien zostać umieszczony wewnętrz obiektu gracza w hierarchii obiektów. W ten sposób koordynacje kamery będą ustawiane względem gracza, a sama kamera poruszać się będzie i obracać wraz z nadziednym obiektem.

Koordynacje kamery ustawiśmy na (0, 0, 0), w ten sposób kamera jest w środkowym punkcie obiektu nadziednego i wytarczy za pomocą narzędzi transformacji przesunąć ją do oczekiwanej pozycji (tak, aby obejmowała obraz zza pleców postaci). Inną, bardziej poręczną metodą jest odpowiednie dostosowanie widoku sceny i skopiowanie jego koordynatów do zaznaczonej w hierarchii kamery.



Rysunek 7: Dostosowanie pozycji kamery gracza względem aktualnego widoku sceny

**Mechanika obrotu kamery** Kolejnym etapem jest oprogramowanie ruchu kamery za pomocą ruchów myszy. Mysz jest używana również do obrotu postacią. Jako, że kamera “przyklejona” jest do postaci, aby uzyskać efekt rozglądzania się, modyfikujemy jedynie jej obrót w pionie, natomiast ruch myszy w poziomie obraca poziomo całą postać wraz z kamerą.



Rysunek 8: Schemat działania systemu sterowania kamerą

---

```

if (!photonView.isMine) return;
if (!stopCamera)
{
    Camera.main.transform.Rotate(new
        Vector3(-Input.GetAxis("Mouse Y") *
        rotationSensitivity * Time.deltaTime, 0, 0));
    rb.transform.Rotate(new Vector3(0, Input.GetAxis("Mouse
        X") * rotationSensitivity * Time.deltaTime, 0));
}

```

---

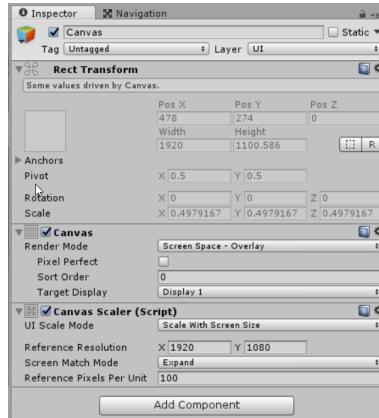
Listing 1: Obrót kamery za pomocą myszy

Obrót kamery polega na dodaniu wektora obrotu do odpowiedniej osi kamery, natomiast wielkość wektora określona jest przez wartość chwilowego przesunięcia myszy, skalowanego parametrem *rotationSensitivity* w celu określenia czułości myszki. Parametr *Time.deltaTime* zwraca czas od ostatniej wyrenderowanej klatki, co pozwala na uzyskanie jednolitej czułości myszki bez względu na ilość generowanych klatek na sekundę.

**Celownik** Aby ułatwić celowanie w przeciwników oraz interakcję z otoczeniem umieściliśmy w grze celownik, który w dalszym etapie tworzenia gry zmienia kolor informując o możliwej interakcji.

Celownik jest obiektem 2D nałożonym na ekran w formie *Sprite'a*. Sprity' to 2-wymiarowe, wcześniej przygotowane obiekty graficzne. W naszym wypadku posłużyliśmy się bezstratnym formatem PNG.

Do sceny został dodany obiekt, na który nałożyliśmy komponenty **Canvas** oraz **CanvasScaler**. Nakładają one na ekran dwuwymiarowe płótno oraz pozwalają na skalowanie go względem podanej rozdzielczości referencyjnej. We wnętrzu takiego płotna można umieszczać obrazy, elementy interfejsu oraz różnego rodzaju wskaźniki (np. Pasek postępu życia bohatera).



Rysunek 9: Konfiguracja płotna 2D do wyświetlania interfejsu gry

Wewnątrz płotna został umieszczony obrazek (komponent **Image**) celownika, wyrównany do środka ekranu.



Rysunek 10: Gotowy celownik gracza

### 3.4 Fizyka

Ważnym elementem gry jest fizyka, aby postaci mogły poruszać się, podskakiwać i reagować na kolizje.

Podstawowym komponentem zapewniającym obliczenia fizyczne jest **Rigidbody**. Komponent ten przyjmuje parametry takie jak wartość masy, tarcia, czy wyłączenie grawitacji.

Aby zdarzenia fizyczne mogły mieć miejsce, na elementy otoczenia oraz samą postać gracza nałożone powinny zostać komponenty **Collider**. Są to obszary określające granice kolizji. Mogą one mieć kształt kapsuły, sześciangu, bądź uproczczonego modelu samego obiektu. Pozwalają one silnikowi gry na przyspieszenie obliczeń i płynne reakcje na zdarzenia kolizji.



Rysunek 11: Przykład – collider mapy terenu

W przypadku postaci poruszających się po mapie, zablokowane zostały obroty we wszystkich osiach (parametr **Freeze Rotation** komponentu Rigidbody), dzięki czemu nie przewracają się one na bok przy nierównościach terenu i kolizjach z otoczeniem. Zdarzenia fizyczne wyliczane są jedynie w pionie (spadanie, skok itp.).

**Mechanika skakania postaci** Mając już zaimplementowaną fizykę, dodaliśmy możliwość skoku. Utworzona została metoda *jump()*, która dodaje do zdań fizycznych naszej postaci siłę w kierunku pionowym o wektorze ustalonym parametrem *jumpPower*. Natomiast na podstawie kolizji z terenem ustalane jest, czy postać stoi na ziemi, dzięki czemu nie można wykonać kolejnego skoku będąc już w powietrzu.

---

```
void Start()
{
    rb = GetComponent<Rigidbody>();
    anim = GetComponent<Animator>();
}

void jump()
{
    if (onGround)
    {
```

```
        onGround = false;
        rb.AddRelativeForce(new Vector3(0, jumpPower, 0));
        anim.SetTrigger("Jump");
    }
}

void OnTriggerEnter(Collider other)
{
    onGround = true;
}
```

---

Listing 2: Fragment algorytmu skoku postaci

### **3.5 Animacje**

**« TO MIEJSCE WYPEŁNIA ADRIAN »**

### 3.6 Poruszanie bohaterów

Poruszanie bohaterów działa na podobnej zasadzie jak mechanika skoku. Do zdarzeń fizycznych bohatera dodawana jest siła o wektorze skierowanym w kierunku określonym przez wciskany klawisz (W, S, A lub D).

W przypadku chodzenia, animacja zależna jest od kierunku, w którym porusza się postać (cofanie, chodzenie bokiem).

Dodatkowym elementem mechaniki poruszania jest bieg, który aktywowy jest poprzez wcisnięcie lewego klawisza Shift.

---

```
run = Input.GetKey(KeyCode.LeftShift);
anim.SetBool("Sprint", run);
float speed = run ? runSpeed : walkSpeed;

...
if (Input.GetKey(KeyCode.W))
{
    rb.AddRelativeForce(Vector3.forward * speed,
        ForceMode.VelocityChange);
}
```

---

Listing 3: Fragment kodu mechaniki biegania – reakcja na wcisnięcie klawisza **W**

### 3.7 Poruszanie postaci wrogów

Poruszanie wrogów odbywa się autonomicznie. Droga jest wyliczana, a koordynacje postaci zmieniane są zgodnie z wyliczoną drogą, dzięki czemu postacie wroga przemieszczają się płynnie w kierunku zdefiniowanego celu.

Unity posiada mechanizm wyznaczania dróg na mapie terenu (więcej informacji na ten temat w dziale 3.1 na stronie 2). Aby móc z niego skorzystać, dodaliśmy do postaci wroga agenta nawigacji **NavAgent**. Jest to komponent zajmujący się obliczeniami drogi i zarządzaniem przemieszczeniem i obrotem postaci w czasie. Podstawowym parametrem jest 3-wymiarowy wektor określający docelową pozycję, silnik gry zajmuje się wyliczeniem trasy prowadzącej do punktu.

Mechanika wrogów polega na wyszukiwaniu najbliższej znajdującej się postaci gracza, a następnie ustaleniu jej jako cel podążania wroga. W momencie, gdy wróg znajduje się odpowiednio blisko, rozpoczyna atak.

---

```
GameObject findNearestPlayer(float maxDistance)
{
    GameObject[] players =
        GameObject.FindGameObjectsWithTag("Player");
    GameObject nearestPlayer = null;
    float distance = maxDistance;
    foreach (GameObject player in players)
    {
        float distanceFromEnemy =
            Vector3.Distance(player.transform.position,
                             transform.position);
        if (distanceFromEnemy < distance)
        {
            distance = distanceFromEnemy;
            nearestPlayer = player;
        }
    }
    return nearestPlayer;
}
```

---

Listing 4: Algorytm zwracający wskaźnik do obiektu znajdującego się najbliżej, przy określonym maksymalnym promieniu poszukiwań



Rysunek 12: Obszary interakcji postaci wroga

W przypadku, gdy wróg „widzi”, postać gracza, jest w jego stronę stale odwrócony. Służy do tego algorytm, który najpierw wylicza kierunek w postaci wektora wartości w skali 0.0f..1.0f w przestrzeni 3D, a następnie zamienia wektor na kąt, do którego stopniowo dąży. Daje to efekt płynnego obracania postaci wroga z możliwością dostosowania prędkości obrotu, aby ruchy wyglądały naturalnie, a wróg dawał możliwość np. zajścia go od tyłu.

---

```
private void rotateTowards(Transform target)
{
    Vector3 direction = (target.position -
        transform.position).normalized;
    Quaternion lookRotation =
        Quaternion.LookRotation(direction);
    transform.rotation =
        Quaternion.Slerp(transform.rotation, lookRotation,
        Time.deltaTime * 5);
}
```

---

Listing 5: Algorytm zwracający wskaźnik do obiektu znajdującego się najbliżej, przy określonym maksymalnym promieniu poszukiwań

**3.8 Nadanie bohaterom indywidualnych umiejętności**

**3.9 Obrażenia**

**3.10 Interakcja ze światem oraz NPC**