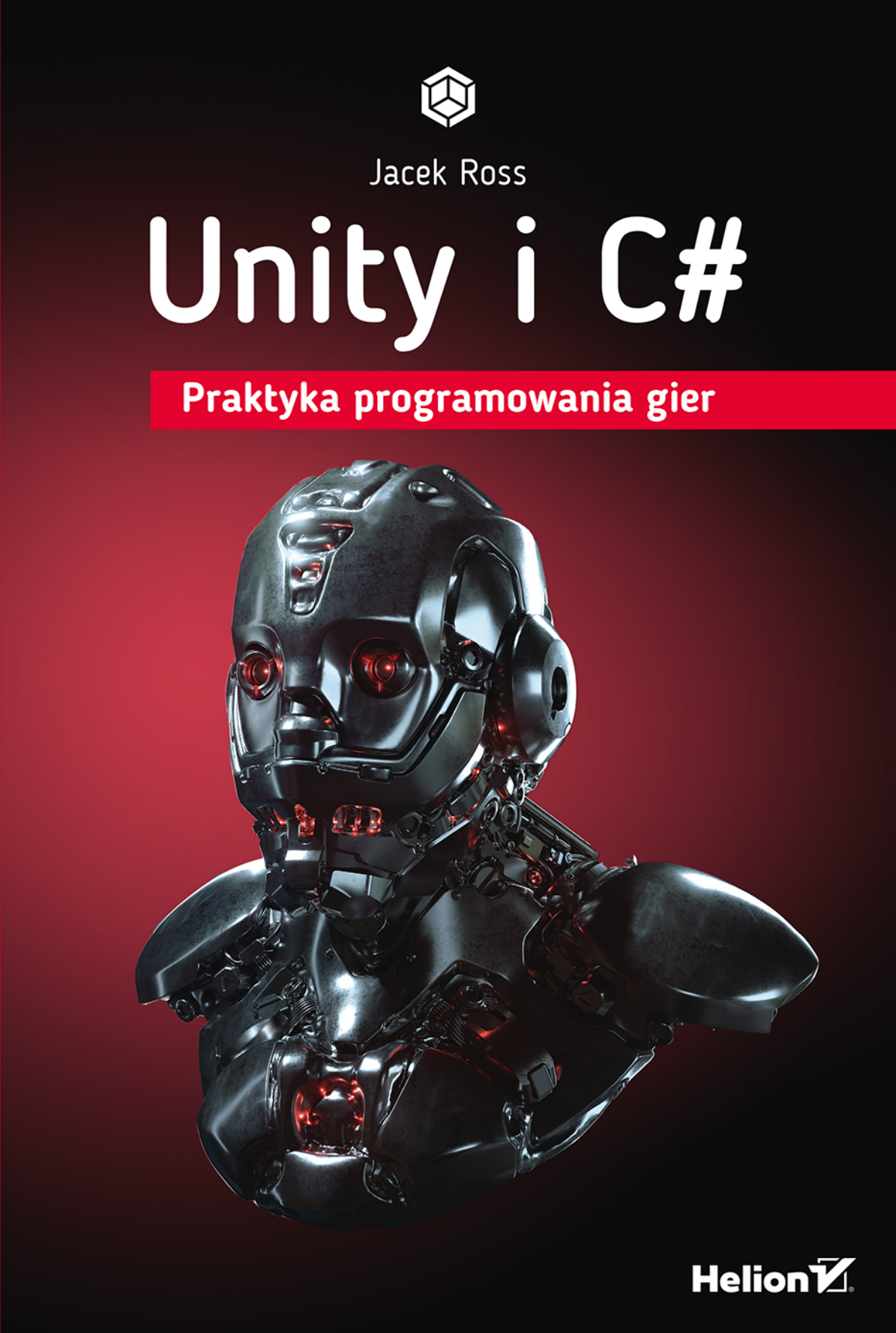
Jacek Ross
Unity i C#
Praktyka programowania gier
Helion

Redaktor prowadzący: Małgorzata Kulik

Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki
Grafika na okładce została wykorzystana za zgodą Shutterstock.com

Helion SA
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/uncppg
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Kody źródłowe wybranych przykładów dostępne są pod adresem:
ftp://ftp.helion.pl/przyklady/uncppg.zip

ISBN: 978-83-283-6586-5

Printed in Poland.

Spis treści

Rozdział 3.

Trzeci sprint. Plecak, podnoszenie przedmiotów, zapis stanu gry, leczenie chorób

Planowanie

W trzecim sprincie rozwiniemy mocno koncepcje przedmiotów, wprowadzając plecak — miejsce, w którym pacjent będzie mógł przechowywać zebrane przedmioty. Plecak będzie można rozwijać oraz wybierać, który przedmiot będzie aktywny. Aktywny przedmiot będzie można przekazać pacjentowi i jeśli przedmiot ten będzie posiadał cechy lecznicze — pacjent może zostać nim uleczony. Wprowadzimy również zapis stanu gry, co jest nieodzowne ze względu na zwiększający się poziom komplikacji gry. Po zakończeniu tego etapu będzie można powiedzieć, że nasza gra stanie się już grą pełnoprawną, którą można rozegrać, a nie tylko potencjalną. Stanie się tak dzięki domknięciu cyklu chorobowego możliwością leczenia. Już nie tylko pacjenci będą mogli się sami przypadkowo wyleczyć, ale gracz będzie miał na ich stan aktywny wpływ. Dodatkowo będziemy mogli posiadać wiele przedmiotów oraz zapiszemy trwale stan gry, dzięki czemu nie będzie się ona restartować za każdym uruchomieniem.

Historyjki użytkownika

1. Pozycja gracza na scenie oraz stan pacjentów i aktywnych przedmiotów zapisywane są przy wyjściu gry i odczytywane przy jej ponownym starcie. Dodatkowy zapis wykonuje klawisz *F5*.
2. Można podnieść przedmiot aktywny ze sceny klawiszem *E*, trafia on do plecaka. Podnoszeniu towarzyszy animacja zmniejszania się i rotowania przedmiotu.
3. Gracz może nieść do 5 przedmiotów w plecaku, który można zwijać i rozwijać klawiszem *P*. Po zwinięciu na ekranie widać tylko jeden przedmiot w postaci miniaturki w małej ramce. Po rozwinięciu w takich ramkach widać wszystkie przedmioty.
4. Jeden z przedmiotów w plecaku jest aktywny. Klawisze *[ ]* zmieniają aktywny przedmiot. Na ekranie można zobaczyć podpis aktywnego przedmiotu.
5. Można wyrzucić aktywny przedmiot z plecaka z powrotem na scenę.
6. Gracz może przekazać aktywny przedmiot pacjentowi, jeśli znajdzie się w jego pobliżu i naciśnie klawisz *E*.
7. Jeśli przedmiot ma cechy lecznicze — pacjent rozpocznie leczenie.
8. Gdy gracz zbliży się do skraju mapy na kilkadziesiąt metrów — zostanie cofnięty do nieco wcześniejszej pozycji oraz na ekranie wyświetli się zabawny komunikat mówiący o tym, że nie może iść zbyt daleko.

Komentarz do planowania

Prace do wykonania dzielą się na trzy wyraźne etapy:

- Przygotowanie i użycie kodu zapisującego stan gry. Na dalszych etapach rozwoju projektu będziemy tego kodu używać ponownie do zapisywania kolejnych aspektów, np. statystyk gry. Na początku użyjemy go do zapisania pozycji gracza na scenie oraz stanu jego plecaka. W drugiej kolejności — stanu pacjentów.
- Rozbudowa systemu przedmiotów o możliwość ich podnoszenia oraz zarządzanie nimi w ekwipunku — plecaku. Przedmiot będzie można również przekazać pacjentowi.
- Dokończenie systemu chorób i ich leczenia poprzez użycie przedmiotów leczniczych.

Aby zrealizować te punkty, konieczne będzie także dodanie do sceny większej liczby aktywnych przedmiotów oraz powiązanie ich z klasą `PrzedmiotLeczniczy`.

Zadania do wykonania

- Implementacja bazowej klasy zapisu stanu gry `MenedzerStanuGry`.
- Użycie klasy `MenedzerStanuGry` do zapisu pozycji gracza.
- Plecak i przedmioty w plecaku.
- Ułożenie przedmiotów na ekranie w panelach plecaka i zarządzanie wybranym panelem.
- Dodanie podpisu aktywnego przedmiotu.
- Obsługa efektów podnoszenia przedmiotu.
- Obsługa wyrzucania przedmiotu na scenę.
- Tworzenie instancji przedmiotu leczniczego w przedmiocie.
- Przekazywanie przedmiotów pacjentom.
- Leczenie pacjentów przedmiotami leczniczymi.
- Implementacja wyświetlania krótkiej wiadomości do gracza.
- Klasa `KontrolerGracza` oraz zawracanie gracza, gdy podejdzie zbyt blisko skraju mapy.

Prace nad sprintem

Implementacja bazowej klasy zapisu stanu gry MenedzerStanuGry

Zapis stanu gry będzie podsystemem umożliwiającym trwałe zapisanie danych w postaci serii zmiennych charakteryzujących się posiadaniem swojej nazwy (identyfikującej jednoznacznie zmienną) i wartości, np. liczbowej albo tekstowej.

Implementując zapis stanu gry, musimy przede wszystkim zdecydować, w jaki sposób technicznie chcemy zapisywać dane. Spośród wielu czynności, jakie nasza gra będzie wykonywać, ta jest jedną z najbardziej specyficznych dla platformy sprzętowej i systemu operacyjnego. A pamiętamy, że chcemy wyprodukować grę dla wielu zupełnie odmiennych systemów. Dobrze byłoby więc znaleźć metodę uniwersalną, tj. taką, w której Unity ma dla nas przygotowany ogólny interfejs do zapisu/odczytu i możemy go użyć niezależnie od systemu, a to, jak dokonuje on faktycznego zapisu/odczytu, ukryte jest w konkretnym pliku binarnym przygotowanym na konkretny system (i realizowane faktycznie w różny sposób, np. w innych lokalizacjach, czasem w pliku, czasem w rejestrze systemowym — nie interesuje nas jak). Taką metodą jest użycie klasy `PlayerPrefs`.

Z drugiej jednak strony zapis i odczyt stanu gry powinny być szybkie i wydajne. W projekcie takim jak *Symulator Zielarza* nie powinniśmy mieć z tym problemów, ale w bardziej rozbudowanych grach może wystąpić konieczność zoptymalizowania tych operacji. Wówczas korzystanie z `PlayerPrefs`, które na przykład w wersji Windows nie jest zbyt szybkie, może nie być wskazane i należy utworzyć metodę specyficzną dla systemu.

Jak postąpić? Dużo zależy od konkretnych potrzeb, konkretnego projektu. My zrobimy coś pośredniego, aby zademonstrować Czytelnikom różne możliwości. Utworzymy wersję dla Windows, która będzie obsługiwać zapis w specjalny sposób, a potem, w następnych rozdziałach metodę ogólną dla systemów mobilnych używającą klasy `PlayerPrefs`. Wybór konkretnej metody będzie sterowany dyrektywą warunkową kompilacji (co już prezentowaliśmy w rozdziale 2.).

Aby jednak w ogóle móc rozdzielić metodę zapisu i wykonywać ją na kilka sposobów, ale jednocześnie z innych miejsc projektu korzystać z zapisu w sposób prosty i jednolity, potrzebujemy **interfejsu**, który pokaże nam, jak z systemu zapisu i ukryje szczegóły implementacji. O interfejsie możemy myśleć jak o zdefiniowaniu sposobu dostępu do jakiejś grupy ogólnych funkcji, np. tutaj zapisu i odczytu stanu gry. Moglibyśmy go nie wydzielać, ale ładnie i czytelnie będzie podać wprost, w jaki sposób powinno się korzystać z tych ogólnych funkcji. Nasz interfejs nazwiemy `IStanGry` i można go zobaczyć na listingu 3.1.

LISTING 3.1. Interfejs IStanGry pokazujący sposób użycia podsystemu zapisu i odczytu stanu gry

```
public interface IStanGry
{
    // utworzenie nowej gry, skasowanie starego zapisu
    void UtworzNowaGre();
    // załadowanie stanu zmiennych gry z trwałego zapisu
    void ZaladujGre();
    // zapisanie stanu zmiennych gry do trwałego zapisu
    void ZapiszGre();
    // pobranie zmiennej typu int o nazwie nazwaKlucza
    int PobierzKluczInt(string nazwaKlucza);
    // pobranie zmiennej typu float o nazwie nazwaKlucza
    float PobierzKluczFloat(string nazwaKlucza);
    // pobranie zmiennej typu string o nazwie nazwaKlucza
    string PobierzKluczString(string nazwaKlucza);
    // zapisanie zmiennej „zmienna" pod nazwą nazwaKlucza
    void ZapiszKlucz(string nazwaKlucza, object zmienna);
    // usunięcie klucza o nazwie nazwaKlucza
    void UsunKlucz(string nazwaKlucza);
    // zwraca true, jeśli istnieje klucz o nazwie nazwaKlucza
    bool istniejeKlucz(string nazwaKlucza);
}
```

Jak już było napisane wyżej, będziemy realizować różny sposób zapisu i odczytu dla różnych wersji systemu operacyjnego. Podsystem zapisu powinien przedstawiać kilka cech (nie wszystkie na tak prostym przykładzie jak nasz projekt są absolutnie niezbędne, ale pokażemy je poglądowo):

- Wewnętrzny sposób zapisu i odczytu dla każdego systemu powinna realizować inna klasa.
- Z zewnątrz podsystemu użycie go powinno być proste i wygodne, a szczegóły implementacji ukryte.
- Na zewnątrz podsystemu nie powinno być informacji o tym, jakie systemy operacyjne obsługujemy (ani warunków decydujących o tym, której klasy używamy).

Aby pogodzić te warunki, napiszemy klasę bazową `MenedzerStanuGry` realizującą w sposób ogólny interfejs `IStanGry`, klasy wyprowadzone z niej realizujące specyfikę zapisu i odczytu po swojemu w zależności od tego, dla jakich systemów są przeznaczone, oraz specjalną klasę będącą pochodną wzorca fabryki: `FabrykaMenedzeraStanuGry`, która będzie odpowiedzialna za utworzenie instancji jednej z klas (odpowiedniej do aktualnie używanego systemu) i zwrócenie jej jako typ interfejsu `IStanGry`. Jeszcze jedno zmienimy w `MenedzerStanuGry` w stosunku do tego, jak zrobilibyśmy to domyślnie — uczynimy ją **klasą abstrakcyjną**. Klasa abstrakcyjna deklaruje pewne metody, ale nie posiada ich definicji (ciał funkcji) — zrzuca to na klasy wyprowadzone. Powoduje to, że nie można utworzyć instancji takiej klasy — zawsze tworzymy instancje klas wyprowadzonych. To doskonale pasuje do naszej sytuacji, a robimy to głównie dlatego, że część funkcji, które np. dokonują technicznie zapisu, nie ma sensu dla klasy ogólnej — one mają sens tylko w kontekście konkretnego systemu. Dzięki temu podział kompetencji w kodzie będzie taki:

- `IStanGry` — interfejs pokazujący, jak można korzystać z podsystemu zapisu i odczytu na zewnątrz. Ukrywa on szczegóły implementacji, które nie są niezbędne osobie korzystającej z podsystemu.
- `MenedzerStanuGry` — klasa abstrakcyjna implementująca tylko ogólne fragmenty zapisu i odczytu, to znaczy takie, które nie są zależne od rodzaju używanego systemu. Klasa ta deklaruje funkcje konkretnego zapisu i odczytu, które muszą być zdefiniowane w klasach potomnych.
- `MenedzerStanuGryWindows`, `MenedzerStanuGryMobilne` itd. — klasy potomne po `MenedzerStanuGry` implementują szczegóły techniczne zapisu i odczytu zależne od systemu operacyjnego.
- `FabrykaMenedzeraStanuGry` — zamyka w sobie logikę wyboru, jakiej konkretnej klasy potomnej użyć. W chwili obecnej logika ta zostanie zrealizowana przez dyrektywy kompilacji warunkowej, ale w przyszłości może to ulec zmianie. Klasa ta będzie używana poza podsystemem do tworzenia instancji klasy realizującej interfejs `IStanGry`.

Jeżeli brzmi to zbyt zawile, to polecam analizę przykładu zawierającego tylko strukturę pustych klas z kluczowymi tylko funkcjami oraz fabryki — listing 3.2.

LISTING 3.2. Przykład pustych jeszcze klas dokonujących zapisów i odczytów danych

```
public class FabrykaMenedzeraStanuGry
{
    public static IStanGry InstancjaMenedzera()
    {
#if UNITY_STANDALONE
            return new MenedzerStanuGryWindows();
#else
            return new MenedzerStanuGryMobilne();
#endif
    }
}
public abstract class MenedzerStanuGry : IStanGry
{
    // ***** konstruktor chroniony
    // ***** implementacja interfejsu IStanGry
    // ***** prywatne i chronione metody wewnętrzne
    // ***** metody wewnętrzne implementowane w klasach wyprowadzonych
    protected abstract void UsunPlik(string nazwaPliku);
    protected abstract string ZaladujPliki(string nazwaPliku);
    protected abstract void ZapiszPliki(string nazwaPliku);
}
public class MenedzerStanuGryWindows : MenedzerStanuGry
{
    public MenedzerStanuGryWindows() :base()
    {
    }
    protected override void UsunPlik(string nazwaPliku)
    {
    }
    protected override string ZaladujPliki(string nazwaPliku)
    {
    }
    protected override void ZapiszPliki(string nazwaPliku)
    {
    }
}
public class MenedzerStanuGryMobilne : MenedzerStanuGry
{
    public MenedzerStanuGryMobilne() :base()
    {
    }
    protected override void UsunPlik(string nazwaPliku)
    {
        throw new System.NotImplementedException("Jeszcze nie zaimplementowano metody
        ↪MenedzerStanuGryMobilne.UsunPlik");
    }
    protected override string ZaladujPliki(string nazwaPliku)
    {
        throw new System.NotImplementedException("Jeszcze nie zaimplementowano metody
        ↪MenedzerStanuGryMobilne.ZaladujPliki");
```

```
    }
    protected override void ZapiszPliki(string nazwaPliku)
    {
        throw new System.NotImplementedException("Jeszcze nie zaimplementowano metody
        ↪MenedzerStanuGryMobilne.ZapiszPliki");
    }
}
```

Najwyższy czas zaimplementować w klasie `MenedzerStanuGry` funkcje realizujące interfejs `IStanGry`. Decydujemy się przechowywać zmienne w postaci obiektów typu `string`. Sam zapis fizyczny może mieć postać tekstową albo binarną (albo może jakąś inną — nie wiemy przecież, jak będzie wyglądał dla różnych systemów) i nie ma to znaczenia dla sposobu ich reprezentacji w klasie `MenedzerStanuGry` — wybieramy `string` jako typ mogący dość wygodnie reprezentować różne typy. Można zastanawiać się, dlaczego do odczytu zmiennych mamy trzy funkcje, a do zapisu tylko jedną. W przypadku odczytu wygodnie będzie nam już wewnętrznie konwertować wartość zmiennej do jednego z typów: `int`, `float`, `string`, ale podczas zapisu możemy podać jako argument obiekt klasy ogólnej `object`, ponieważ niezależnie od tego, jaki będzie faktyczny typ argumentu, łatwo skonwertujemy go do obiektu typu `string`. Inne funkcje interfejsu realizujemy dość prosto, manipulujemy tablicą zmiennych albo wywołujemy wewnętrzne techniczne funkcje usuwania, zapisu i odczytu, które zdefiniujemy jako abstrakcyjne (a więc ich ciała będą dopiero w klasie wyprowadzonej). Można to zobaczyć na listingu 3.3.

LISTING 3.3. Implementacja interfejsu IStanGry w klasie MenedzerStanuGry

```
public abstract class MenedzerStanuGry : IStanGry
{
    protected Dictionary<string, string> _zmienne;
    public const string DOMYSLNA_NAZWA_ZAPISU = "autozapis";
    public const string DOMYSLNY_FOLDER_ZAPISU = "Zapis";
    // ***** konstruktor chroniony
    protected MenedzerStanuGry()
    {
        _zmienne = new Dictionary<string, string>();
    }
    // ***** implementacja interfejsu IStanGry
    public void UtworzNowaGre()
   {
        UsunPlik(DOMYSLNA_NAZWA_ZAPISU);
        _zmienne.Clear();
    }
    public void ZaladujGre()
    {
        string zawartosc = ZaladujPliki(DOMYSLNA_NAZWA_ZAPISU);
        KonwertujZawartoscDoZmiennych(zawartosc);
    }
    public void ZapiszGre()
    {
        ZapiszPliki(DOMYSLNA_NAZWA_ZAPISU);
    }
```

```
public int PobierzKluczInt(string nazwaKlucza)
{
    if (_zmienne.ContainsKey(nazwaKlucza))
    {
        if (_zmienne[nazwaKlucza] == "")
            return 0;
        try
        {
            return int.Parse(_zmienne[nazwaKlucza]);
        }
        catch (System.Exception e)
        { Debug.LogError("Błąd konwersji: " + nazwaKlucza + ":" + _zmienne[nazwaKlucza]); }
    }
    return 0;
}
public float PobierzKluczFloat(string nazwaKlucza)
{
    if (_zmienne.ContainsKey(nazwaKlucza))
    {
        if (_zmienne[nazwaKlucza] == "")
            return 0f;
        try
        {
            return float.Parse(_zmienne[nazwaKlucza]);
        }
        catch (System.Exception e)
        { Debug.LogError("Błąd konwersji: " + nazwaKlucza + ":" + _zmienne[nazwaKlucza]); }
    }
    return 0f;
}
public string PobierzKluczString(string nazwaKlucza)
{
    if (_zmienne.ContainsKey(nazwaKlucza))
    {
        return _zmienne[nazwaKlucza];
    }
    return "";
}
public void ZapiszKlucz(string nazwaKlucza, object zmienna)
{
    if (!_zmienne.ContainsKey(nazwaKlucza))
        _zmienne.Add(nazwaKlucza, zmienna.ToString());
    else _zmienne[nazwaKlucza] = zmienna.ToString();
}
public void UsunKlucz(string nazwaKlucza)
{
    if (_zmienne.ContainsKey(nazwaKlucza))
        _zmienne.Remove(nazwaKlucza);
}
public bool istniejeKlucz(string nazwaKlucza)
{
    return  _zmienne.ContainsKey(nazwaKlucza);
}
// ***** prywatne i chronione metody wewnętrzne
protected virtual string SciezkaZapisu()
```

```
    {
        return DOMYSLNY_FOLDER_ZAPISU;
    }
    private void KonwertujZawartoscDoZmiennych(string zawartosc)
    {
        _zmienne = new Dictionary<string, string>();
         string[] tokeny = zawartosc.Split('\n');
         for (int i = 0; i < tokeny.Length; i++)
         {
             int pozycjaWartosci = tokeny[i].IndexOf("=");
             if (pozycjaWartosci > 0)
                 ZapiszKlucz(tokeny[i].Substring(0, pozycjaWartosci),
                 ↪tokeny[i].Substring(pozycjaWartosci + 1));
         }
    }
    // ***** metody wewnętrzne implementowane w klasach wyprowadzonych
    protected abstract void UsunPlik(string nazwaPliku);
    protected abstract string ZaladujPliki(string nazwaPliku);
    protected abstract void ZapiszPliki(string nazwaPliku);
}
```

Zaimplementujemy jeszcze zapis i odczyt na Windows, aby móc już testowo zacząć korzystać z tego podsystemu. Zapiszemy zmienne do plików, w folderze zwracanym specjalną funkcją `SciezkaZapisu`, a do samych operacji wejścia/wyjścia użyjemy strumieni. Pokazano to na listingu 3.4.

LISTING 3.4. Implementacja zapisu i odczytu zmiennych stanu gry dla systemu Windows

```
public class MenedzerStanuGryWindows : MenedzerStanuGry
{
    public MenedzerStanuGryWindows() :base()
    {
    }
    protected override void UsunPlik(string nazwaPliku)
    {
        if (File.Exists(SciezkaZapisu() + "/" + nazwaPliku + ".dat"))
            File.Delete(SciezkaZapisu() + "/" + nazwaPliku + ".dat");
    }
    protected override string ZaladujPliki(string nazwaPliku)
    {
       if (!Directory.Exists(SciezkaZapisu()))
            Directory.CreateDirectory(SciezkaZapisu());
        string sciezka = SciezkaZapisu() + "/" + nazwaPliku + ".dat";
        if (File.Exists(sciezka))
        {
            StreamReader strumien = File.OpenText(sciezka);
            string zawartosc = strumien.ReadToEnd();
            strumien.Close();
            return zawartosc;
        }
        else return "";
    }
```

```
    protected override void ZapiszPliki(string nazwaPliku)
    {
        if (!Directory.Exists(SciezkaZapisu()))
            Directory.CreateDirectory(SciezkaZapisu());
        StringBuilder sb = new StringBuilder();
        foreach (string key in _zmienne.Keys)
            sb.Append(key + "=" + _zmienne[key] + '\n');
        string zawartosc = sb.ToString();
        StreamWriter strumien;
        strumien = File.CreateText(SciezkaZapisu() + "/" + nazwaPliku + ".dat");
        strumien.WriteLine(zawartosc);
        strumien.Close();
    }
}
```

Sposób, w jaki zaimplementowaliśmy interfejs `IStanGry` w klasie `MenedzerStanuGry`, gwarantuje nam niemal natychmiastowe użycie tego podsystemu w trakcie rozgrywki, ponieważ zmienne znajdują się w tablicy, do której dostęp jest dość szybki. Powolne mogą być odczyt i zapis stanu, czego nie możemy dokładnie przewidzieć, ponieważ ich implementacje będą różne na różnych systemach, a szybkość może znacznie różnić się na poszczególnych urządzeniach użytkowników. Jeżeli stanie się to dużą przeszkodą, to Czytelnik może np. zaimplementować system częściowego doczytywania i dopisywania zmiennych do plików albo rozbić je na wiele plików itp. Dobrym pomysłem będzie też zapewne zmiana zapisu tekstowego na binarny i zaciemnienie plików zapisu w taki sposób, aby gracz nie mógł nimi łatwo manipulować, lub wręcz zaszyfrowanie tych plików (rzecz konieczna w przypadku rozgrywek sieciowych).

Użycie klasy MenedzerStanuGry do zapisu pozycji gracza

Zapis i odczyt gry będzie prowadzony przez klasę `MenedzerGry`. Nie znaczy to, że musi ona dokonać faktycznie wszystkich odczytów i zapisów zmiennych stanu gry — może delegować te czynności na inne fragmenty kodu. Jednak główny punkt wejścia oraz dbanie o wywoływanie zapisów i odczytów spoczywać będą na tej właśnie klasie i jej funkcjach: `ZapiszGre`, `OdczytajGre`. Klasa `MenedzerUI` będzie wywoływać funkcję `MenedzerGry.ZapiszGre` w reakcji na naciśnięcie przez gracza przycisku *F5* (tak zapisano w historyjce użytkownika).

Zapis gry będzie dokonywany przy opuszczaniu gry, co `MenedzerGry` jako komponent Unity może wykryć, implementując funkcję należącą do cyklu życia komponentu `OnApplicationQuit`. Odczyt będziemy inicjować w pierwszym wykonaniu funkcji `Update`. Jeżeli Czytelnik uważa, że zapis dokonywany wyłącznie przy wychodzeniu z gry to za mało (nie zadziała przy gwałtownym zamknięciu aplikacji np. poprzez wyjątek bądź zanik zasilania na komputerze), to bardzo łatwo może zaimplementować automatyczny zapis, który będzie wywoływać metodę `ZapiszGre` z funkcji `Update` co pewien z góry ustalony czas.

Pierwsza implementacja metod zapisu i odczytu będzie zapisywać i odczytywać pozycję gracza, czyli wektor pozycji kontrolera pierwszoosobowego. Odpowiedni kod można zobaczyć na listingu 3.5.

LISTING 3.5. Zapis i odczyt stanu gry w klasie MenedzerGry

```
public class MenedzerGry: MonoBehaviour
{
    //...fragment kodu...
    private IStanGry _menedzerZapisu;
    public void ZapiszGre()
    {
        _menedzerZapisu.ZapiszKlucz("pozycja_gracza_x",
        ↪KontrolerGracza.transform.position.x);
        _menedzerZapisu.ZapiszKlucz("pozycja_gracza_y",
        ↪KontrolerGracza.transform.position.y);
        _menedzerZapisu.ZapiszKlucz("pozycja_gracza_z",
        ↪KontrolerGracza.transform.position.z);
       _menedzerZapisu.ZapiszGre();
    }
    public void OdczytajGre()
    {
       _menedzerZapisu.ZaladujGre();
        KontrolerGracza.transform.position = new Vector3(
            _menedzerZapisu.PobierzKluczFloat("pozycja_gracza_x"),
            ↪_menedzerZapisu.PobierzKluczFloat("pozycja_gracza_y"),
            ↪_menedzerZapisu.PobierzKluczFloat("pozycja_gracza_z"));
    }
    private void Awake()
    {
        //...fragment kodu...
        _menedzerZapisu = FabrykaMenedzeraStanuGry.InstancjaMenedzera();
    }
    private void Update()
    {
        if(_PierwszyUpdate)
        {
            //...fragment kodu...
            OdczytajGre();
            _PierwszyUpdate = false;
        }
    }
    private void OnApplicationQuit()
    {
        ZapiszGre();
    }
}
```

Rozbudowa `MenedzeraUI` o sprawdzenie naciśnięcia klawisza *F5* i wywołanie `Menedzer↪Gry.ZapiszGre` jest bardzo prosta i nie będziemy jej przedstawiać na listingu.

Plecak i przedmioty w plecaku

Zanim zaimplementujemy sam plecak/ekwipunek, zdefiniujemy sobie klasę `Przedmiot`. Czym będzie się ona różniła od klas `PrzedmiotLeczniczy` i `AktywnyObiekt`? Czy potrzebujemy kolejnej? Tak, ze względu na nieco inne znaczenie każdej z klas. Podsumujmy to:

- `AktywnyObiekt` — obiekt na scenie, który ma swoją nazwę (możemy ją zobaczyć, gdy najedziemy na niego wskaźnikiem myszki), może zmieniać położenie na scenie, a w następnych krokach będziemy generować takie obiekty dynamicznie w trakcie gry oraz zapisywać w stanie gry ich obecność i położenie. Nie ma jednak pewności, czy każdy taki obiekt będzie mógł być wzięty przez gracza. Może to być tylko interaktywny obiekt na scenie, np. klamka u drzwi, skrzynia do otworzenia itp.
- `PrzedmiotLeczniczy` — przedmiot, który może znaleźć się w posiadaniu gracza, a który ma cechy lecznicze wobec pacjentów i na leczeniu właśnie koncentruje się znaczenie tej klasy. Posiada pewną liczbę substancji leczniczych (bądź trujących). Niekoniecznie musi być obiektem na scenie, może np. będziemy tworzyć takie przedmioty dynamicznie i przekazywać je pacjentom bez pośrednictwa gracza i jego plecaka?
- `Przedmiot` — obiekt sceny, który może znaleźć się w plecaku gracza i podlegać manipulacjom takim jak tworzenie nowych przedmiotów itp. Powinien posiadać też klasę `AktywnyObiekt`, ale ta klasa koncentruje się na byciu częścią plecaka i to jest jej główna odpowiedzialność.

Rozdzielamy te klasy także zgodnie z zasadą pojedynczej odpowiedzialności, starając się zapewnić to, aby pojedyncza klasa odpowiadała za pewną zamkniętą grupę działań związanych z pewną konkretną, zwięzłą logiką: bycie przedmiotem w plecaku jest taką zamkniętą grupą działań, bycie przedmiotem ogólnie — nie jest.

Na początku implementacja klasy `Przedmiot` będzie prosta. Od razu przewidujemy zapotrzebowanie na pewne pola, właściwości i metody związane z plecakiem, ale dopiero dalszy rozwój kodu przyniesie rozbudowę tych miejsc. Patrz listing 3.6.

LISTING 3.6. Pierwsza implementacja klasy Przedmiot

```
public class Przedmiot : MonoBehaviour
{
    public string Nazwa
    {
        get
        {
            if (_komponentAO != null)
                return _komponentAO.Nazwa;
            else
                return "?";
        }
    }
```

```
    public Vector3 przesuniecieWPlecaku = Vector3.zero;
    public Vector3 rotacjaWPlecaku = Vector3.zero; // w mierze kątowej (Euler angles)
    public Vector3 skalaWPlecaku = new Vector3(0.2f, 0.2f, 0.2f);
    private AktywnyObiekt _komponentAO;
    private void Start()
    {
        _komponentAO = GetComponent<AktywnyObiekt>();
    }
}
```

Plecak to w dużym uproszczeniu lista przedmiotów (obiektów klasy `Przedmiot`), którą wyświetlamy częściowo bądź w całości na ekranie. Będziemy manipulować nim głównie od strony UI (gracz podnosi coś, używa, oddaje albo wyrzuca) oraz ładować/zapisywać w stanie gry. Pierwsza niezbędna do dalszych działań implementacja pokazana jest na listingu 3.7.

LISTING 3.7. Pierwsza implementacja klasy Plecak

```
public class Plecak : MonoBehaviour
{
    public const int ROZMIARPLECAKA = 5;    // maks. liczba przedmiotów w plecaku
    // przedmiot na pozycji „pozycja", null oznacza brak przedmiotu w plecaku na tej pozycji
    public Przedmiot this[int pozycja]
    {
        get {
            if (pozycja >= 0 && pozycja < ROZMIARPLECAKA)
                return _przedmioty[pozycja];
            else
                return null;
        }
        private set {
            _przedmioty[pozycja] = value;
        }
    }
    private Przedmiot[] _przedmioty;
    private int _wybranyPanel;
    // dodaje przedmiot do plecaka na pierwsze wolne miejsce albo na miejsce _wybranyPanel,
    // jeśli brak wolnych miejsc
    public void Dodaj(Przedmiot dodawanyPrzedmiot)
    {
        int panelNaKtoryDodajemy = _wybranyPanel;
        if (this[panelNaKtoryDodajemy] != null)
            for (int i = ROZMIARPLECAKA - 1; i >= 0; i--)
                if (_przedmioty[i] == null)
                    panelNaKtoryDodajemy = i;
        if (this[panelNaKtoryDodajemy] != null)
            PrzedmiotZostalUsuniety(this[panelNaKtoryDodajemy]);
        this[panelNaKtoryDodajemy] = dodawanyPrzedmiot;
        dodawanyPrzedmiot.transform.parent = transform;
        dodawanyPrzedmiot.transform.localScale = dodawanyPrzedmiot.skalaWPlecaku;
        dodawanyPrzedmiot.transform.localRotation = Quaternion.Euler(dodawanyPrzedmiot.
        ↪rotacjaWPlecaku);
       dodawanyPrzedmiot.transform.localPosition = dodawanyPrzedmiot.przesuniecieWPlecaku;
    }
```

```
    private void Awake()
    {
        _przedmioty = new Przedmiot[ROZMIARPLECAKA];
        _wybranyPanel = 0;
        transform.position = Camera.main.ScreenToWorldPoint(new Vector3(5, 5, 0.35f));
    }
    private void PrzedmiotZostalUsuniety(Przedmiot usuniety)
    {
        Destroy(usuniety.gameObject);
    }
}
```

Zanim wyjaśnię trudniejsze fragmenty tego kodu, najpierw utwórzmy odpowiednie obiekty na scenie i dodajmy obsługę plecaka do `MenedzeraGry`. Na scenie dodajemy obiekt o nazwie `Plecak` jako podobiekt obiektu *FirstPersonCharacter* i dodajemy do niego komponent `Plecak`. W kodzie klasy `MenedzerGry` dodajemy deklarację: `public Plecak ObiektPlecaka;`, a w obiekcie na scenie, który ma komponent `MenedzerGry`, ustawiamy to pole na przed chwilą dodany obiekt plecaka. Nasz plecak zostanie spozycjonowany w lewym, dolnym rogu, a dokładniej: o 5×5 pikseli od tego rogu. Robimy to z poziomu kodu za pomocą funkcji `Camera.screenToWorldPoint`, dzięki czemu możemy umieścić plecak dokładnie w takim punkcie przestrzeni sceny, który w momencie startu odpowiada lewemu, dolnemu rogowi ekranu, niezależnie od rozdzielczości, w której zostanie uruchomiona gra.

Najciekawszym fragmentem kodu listingu 3.7 jest z pewnością **indekser** — jest to fragment wyglądający jak większość właściwości klas, ale z użyciem słowa `this` (`public Przedmiot this[int pozycja]`). Taka konstrukcja umożliwi nam użycie obiektu klasy `Plecak`, jakby był tablicą, a więc jeśli będziemy mieli do czynienia z obiektem o nazwie `obiektPlecak`, to będziemy mogli napisać: `obiektPlecak[2]` i zostanie wówczas wywołany kod indeksera, a dokładniej metody `get`, która zwraca przedmiot o podanym indeksie (w tym przykładzie — o indeksie 2). To tak, jakbyśmy napisali `obiektPlecak._przedmioty[2]`, z tym że tablica `_przedmioty` jest prywatna, a więc niedostępna z zewnątrz. Takie ukrycie tablicy, a jednocześnie pozostawienie sobie możliwości indeksowania obiektów jest jednocześnie eleganckie i wygodne. Warto pamiętać jednak, aby indekser był intuicyjny i dotyczył czegoś, co jest główną tablicą czy głównym zastosowaniem klasy, a nie pobocznym, ponieważ inaczej przestanie być zrozumiały. Wewnątrz klasy również nie musimy już odnosić się do tablicy `_przedmioty`, lecz napisać `this[indeks]`, aby odczytać element o indeksie indeks. Ma to tę zaletę, że szczegóły obsługi zwracania i ustawiania elementu tablicy są w jednym miejscu i łatwo nimi zarządzać.

Lista przedmiotów ma ograniczenie stałą `ROZMIARPLECAKA` — to liczba przedmiotów, które mogą jednocześnie znaleźć się w plecaku. Gdy dodajemy nowy przedmiot, to wskakuje on na listę na pierwszą wolną pozycję (wolne pozycje oznaczone są wartością `null` na liście przedmiotów), a jeśli żadne miejsce nie jest wolne, przedmiot zamienia się miejscem z aktualnie wybranym przedmiotem (na razie stary przedmiot jest niszczony, później zrobimy to ładniej). Wybrany aktualnie przedmiot, a dokładniej jego indeks na liście oznaczamy zmienną `_wybranyPanel`. Przez chwilę nic się nie będzie z nią działo.

Mamy już dodawanie przedmiotu do plecaka funkcją `Plecak.Dodaj`, możemy ją podpiąć w klasie `Przedmiot`, a w `MenedzerUI` podpiąć dodawanie przedmiotu do plecaka po naciśnięciu klawisza *E*. Wykorzystujemy już wcześniej poznaną informację o tym, na co patrzy kamera. Jeśli patrzy na obiekt posiadający komponent `Przedmiot` — każemy dodać mu się do plecaka. Można to zobaczyć na listingu 3.8.

LISTING 3.8. Podpięcie dodawania przedmiotu w klasie MenedzerUI oraz Przedmiot

```
public class MenedzerUI : MonoBehaviour
{
    //...fragment kodu...
    void Update()
    {
        //...fragment kodu...
        if (Input.GetKeyDown(KeyCode.E))
        {
            if (patrzymy != null && patrzymy.GetComponent<Przedmiot>() != null)
                patrzymy.GetComponent<Przedmiot>().DodajDoPlecaka();
        }
    }
}
public class Przedmiot : MonoBehaviour
{
    //...fragment kodu...
    public void DodajDoPlecaka()
    {
        MenedzerGry.InstancjaMenedzeraGry.ObiektPlecaka.Dodaj(this);
    }
}
```

Możemy dodać na scenę kilka przedmiotów do testów. Mogą to być np. różne bryły, wystarczy, że będą posiadać komponenty `AktywnyObiekt` oraz `Przedmiot`. Szybkie testy pokazują jednak, że plecak nie działa jeszcze dobrze — chociaż przedmioty podnoszą się i wskakują do plecaka, a nadmiarowe są z niego usuwane, to wszystkie przedmioty z plecaka pokazują się w jednym miejscu. Musimy dodać obsługę wyświetlania paneli w plecaku i spozycjonować przedmioty względem nich.

Ułożenie przedmiotów na ekranie w panelach plecaka i zarządzanie wybranym panelem

Tworzymy sobie dwie grafiki w folderze *Assets/PrototypoweZasoby* — *RamkaPrzedmiotu.png* i *ZaznaczenieAktywnejRamki.png*, oba o rozmiarach 512×512 pikseli. Pierwsza pokazuje ramkę pojedynczego przedmiotu plecaka, a druga taką samą ramkę, ale z czerwonym zaznaczeniem aktywnej ramki. Ustawiamy im `TextureType` na *Sprite (2D and UI)* oraz `PixelsPerUnit` na 128. Następnie tworzymy dwa obiekty będące podobiektami do obiektu *Plecak* na scenie: *RamkaPrzedmiotu* i *ZaznaczenieAktywne*, którym dodajemy komponent `SpriteRenderer` i ustawiamy parametr `Sprite` na wcześniej utworzone grafiki. Dodamy teraz do klasy `Plecak`: dwa pola `WzorzecPaneluPlecaka` i `ZnacznikAktywnejRamki`,

którym wskażemy dwa nowododane obiekty sceny. `WzorzecPaneluPlecaka` posłuży nam jako wzorzec do generowania tylu ramek dla przedmiotów, ile będzie aktualnie wyświetlanych, a `ZnacznikAktywnejRamki` będzie pokazywany w miejscu tej ramki, która aktualnie będzie wybrana. Dodamy też właściwość `Rozszerzony`, która będzie określała, czy pokazujemy tylko aktualnie wybrany przedmiot, czy wszystkie. Nowe fragmenty kodu — patrz listing 3.9.

LISTING 3.9. Rozszerzenie kodu klasy Plecak o obsługę wyświetlania kilku przedmiotów

```
public class Plecak : MonoBehaviour
{
    public const int ROZMIARDUSZKAPLECAKA = 512; // rozmiar duszka panelu przedmiotu
    public GameObject WzorzecPaneluPlecaka;      // wzorzec do generowania paneli przedmiotów
    public GameObject ZnacznikAktywnejRamki;     // obiekt, którym będziemy zaznaczać aktywną ramkę
    private bool _rozszerzony;
    // informuje i ustawia plecak rozszerzony do wszystkich przedmiotów
    public bool Rozszerzony
    {
        get { return _rozszerzony; }
        set
        {
            if (_rozszerzony)
                Debug.Log("");
            _rozszerzony = value;
            OdswiezPozycjePaneliUI();
        }
    }
    private GameObject[] _panelePlecaka;
    private int _wybranyPanel;
    // dodaje przedmiot do plecaka na pierwsze wolne miejsce albo na miejsce _wybranyPanel, jeśli brak wolnych miejsc
    public void Dodaj(Przedmiot dodawanyPrzedmiot)
    {
        //...fragment kodu...
        OdswiezPozycjePrzedmiotow();
    }
    private void Awake()
    {
        _przedmioty = new Przedmiot[ROZMIARPLECAKA];
        _panelePlecaka = new GameObject[ROZMIARPLECAKA];
        if (ZnacznikAktywnejRamki == null)
            Debug.Log("Nie ustawiono znacznika aktywnego panelu");
        if (WzorzecPaneluPlecaka == null)
            Debug.Log("Nie ustawiono wzorca panelu plecaka");
        else
            GenerujPaneleUI();
        Rozszerzony = false;
        _wybranyPanel = 0;
        transform.position = Camera.main.ScreenToWorldPoint(new Vector3(5, 5, 0.35f));
    }
    private void GenerujPaneleUI()
    {
        if (_panelePlecaka != null)
            foreach (GameObject panel in _panelePlecaka)
                Destroy(panel);
```

```
        _panelePlecaka = new GameObject[ROZMIARPLECAKA];
        for (int i = 0; i < ROZMIARPLECAKA; i++)
        {
            _panelePlecaka[i] = Instantiate(WzorzecPaneluPlecaka, transform);
            _panelePlecaka[i].transform.localPosition = new Vector3(0f, 0f, 0f);
            _panelePlecaka[i].transform.localRotation = Quaternion.identity;
            _panelePlecaka[i].SetActive(true);
        }
        WzorzecPaneluPlecaka.SetActive(false);
    }
    private void OdswiezPozycjePaneliUI()
    {
        if (_panelePlecaka == null)
            GenerujPaneleUI();
        float rozmiarPanelu = (ROZMIARDUSZKAPLECAKA / 128f) *
        ↪WzorzecPaneluPlecaka.transform.localScale.x;
        for (int i = 0; i < ROZMIARPLECAKA; i++)
        {
            _panelePlecaka[i].SetActive(Rozszerzony | i == _wybranyPanel);
            if (Rozszerzony)
                _panelePlecaka[i].transform.localPosition = new Vector3(rozmiarPanelu *
                ↪0.5f + i * (rozmiarPanelu + 0.01f), rozmiarPanelu * 0.5f, 0);
            else
                _panelePlecaka[i].transform.localPosition = new Vector3(rozmiarPanelu *
                ↪0.5f, rozmiarPanelu * 0.5f, 0);
        }
        ZnacznikAktywnejRamki.SetActive(Rozszerzony);
        ZnacznikAktywnejRamki.transform.localPosition = _panelePlecaka[_wybranyPanel].
        ↪transform.localPosition;
        OdswiezPozycjePrzedmiotow();
    }
    private void OdswiezPozycjePrzedmiotow()
    {
        for (int i = 0; i < ROZMIARPLECAKA; i++)
        {
            if (this[i] != null)
            {
                this[i].gameObject.SetActive(_panelePlecaka[i].activeInHierarchy);
                this[i].transform.localPosition = _panelePlecaka[i].transform.
                ↪localPosition + this[i].przesuniecieWPlecaku;
            }
        }
    }
}
```

Ramki/panele dla przedmiotów tworzone są w funkcji `GenerujPaneleUI`, a ich pozycje w `OdswiezPozycjePaneliUI`, gdzie zmieniamy także pozycję ramki czerwonej, tej, która zaznacza nam wybraną pozycję. Są to oczywiście pozycje `localPosition`, czyli pozycje względem rodzica, którym jest obiekt `Plecak`. A ponieważ jego rodzicem jest z kolei kontroler pierwszoosobowy, to ramki przedmiotów poruszają się wraz z ruchem gracza. Przedmiotom ustawiamy pozycje w funkcji `OdswiezPozycjePrzedmiotow`. Jedna i druga funkcja ustawia także aktywność (widoczność) przedmiotów i ramek. W zależności od tego, czy plecak jest zwinięty czy rozwinięty — widzimy tylko wybrany panel albo wszystkie z zaznaczeniem wybranego na czerwono. Pozycje, rotacje i skale przedmiotów są jeszcze

aktualizowane o odpowiednie pola, które dodawaliśmy wcześniej w klasie `Przedmiot` (`przesuniecieWPlecaku`, `rotacjaWPlecaku` i `skalaWPlecaku`), dzięki czemu mamy możliwość sprawienia, że przedmiot o najdziwniejszym kształcie podporządkuje się i będzie dobrze wyglądać w plecaku.

Warto w tym momencie zauważyć, że przyjęliśmy podejście polegające na pokazywaniu w plecaku prawdziwych obiektów sceny. Podkreślam to jeszcze raz: plecak nie jest nakładką na UI wyświetlającą się tylko na ekranie, ale grupą obiektów naprawdę istniejących na scenie. Ma to swoje wady i zalety. Zaletą jest brak konieczności tworzenia ikonek oraz łatwiejsze przejście do projektu wirtualnej rzeczywistości (gdzie będziemy chcieli zawsze manipulować prawdziwymi obiektami na scenie). Wadą — reagowanie tych obiektów z innymi obiektami sceny oraz np. polem widzenia kamery. Wady można zminimalizować, np. usuwając obiektom kolidery, zmieniając im shadery itp. Jeśli jednak w konkretnym projekcie takie rozwiązanie okaże się zbyt kłopotliwe — Czytelnik będzie miał dwa wyjścia:

- Po dodaniu przedmiotu do plecaka usunąć ewentualne renderery 3D i inne powiązane z nimi komponenty, dodać komponent `SpriteRenderer` i wyświetlić na nim ikonkę 2D przedmiotu. Plecak nadal będzie obiektem na scenie, ale przedmioty będą dwuwymiarowe i łatwiej będzie uniknąć wad aktualnego rozwiązania.
- Zrobić plecak całkowicie od nowa w warstwie UI na kanwie operującej jako nakładka na ekran.

W naszym rozwiązaniu pozostały nam jeszcze do zrobienia implementacja zmiany aktualnie wybranego panelu przedmiotu oraz podpięcie reakcji na naciśnięcia klawiszy w klasie `MenedzerUI`: P rozwijającego i zwijającego plecak oraz klawiszy *[* i *]*, które będą zmieniać aktualnie wybrany panel przedmiotu. Całość zmian na listingu 3.10.

LISTING 3.10. Zmiany w klasie Plecak związane ze zmianą aktualnie wybranego panelu oraz obsługa klawiszy związanych z plecakiem w klasieMenedzerUI

```
public class Plecak : MonoBehaviour
{
    //...fragment kodu...
    // przesuwa wybrany panel o przesunięcie (1 w prawo, -1 w lewo)
    public void PrzesunWybranyPanel(int przesuniecie)
    {
        _wybranyPanel += przesuniecie;
        if (_wybranyPanel < 0)
            _wybranyPanel = ROZMIARPLECAKA - 1;
        _wybranyPanel %= ROZMIARPLECAKA;
        OdswiezPozycjePaneliUI();
    }
}
public class MenedzerUI : MonoBehaviour
{
    //...fragment kodu...
```

```
    void Update()
    {
        //...fragment kodu...
        if (Input.GetKeyDown(KeyCode.P))
            MenedzerGry.InstancjaMenedzeraGry.ObiektPlecaka.Rozszerzony = !MenedzerGry.
            ↪InstancjaMenedzeraGry.ObiektPlecaka.Rozszerzony;
        if (Input.GetKeyDown(KeyCode.LeftBracket))
            MenedzerGry.InstancjaMenedzeraGry.ObiektPlecaka.PrzesunWybranyPanel(-1);
        if (Input.GetKeyDown(KeyCode.RightBracket))
            MenedzerGry.InstancjaMenedzeraGry.ObiektPlecaka.PrzesunWybranyPanel(1);
    }
}
```

Możemy teraz uruchomić grę i z zadowoleniem zobaczyć, że plecak działa już bez zarzutu — można podnosić przedmioty i oglądać je w plecaku, który można zwijać i rozwijać, oraz przesuwać zaznaczanie. Plecak w działaniu można zobaczyć na rysunku 3.1.

RYSUNEK 3.1. *Zrzut ekranu z aktualnego stanu gry z działającym już plecakiem*

Dodanie podpisu aktywnego przedmiotu

Nie zrealizowaliśmy jeszcze fragmentu historyjki użytkownika, który mówi: „Na ekranie można zobaczyć podpis aktywnego przedmiotu”. Aby go zrealizować, dodamy na scenę dwa pola tekstowe, podobnie jak dodawaliśmy pola pokazujące nazwę przedmiotu ponad wskaźnikiem myszki. Nie będę podawać krok po kroku, jak to zrobić, ponieważ jest to analogiczne do tego, co już robiliśmy. Obejrzymy tylko dodany kod źródłowy. Dodajemy w nim dwa pola typu `Text` na podpis i jego cień oraz pole na kanwę. Kanwę

ustawiamy w kodzie tak, by była przesunięta na prawo od ostatniego panelu przedmiotu. Pola tekstowe ustawiamy w odpowiednich pozycjach względem kanwy w samym edytorze. Na końcu dodajemy funkcję `OdswiezInformacjeOAktywnymPrzedmiocie` i podpinamy ją w nowej funkcji `OdswiezWszystko`, do której dodajemy też wywołania innych funkcji odświeżających widok plecaka (robimy ten refaktoring, ponieważ tych funkcji jest już kilka i zrobił nam się brzydki ciąg wzajemnych wywołań, które lepiej przenieść w jedno miejsce). Wcześniejsze wywołania funkcji `OdswiezPozycjePaneliUI` zmieniamy na wywołania `OdswiezWszystko`. Całość zmian w kodzie można zobaczyć na listingu 3.11.

LISTING 3.11. Dodanie podpisu aktywnego przedmiotu — zmiany w klasie Plecak

```
public class Plecak : MonoBehaviour
{
    //...fragment kodu...
    public Text OpisAktywnegoPrzedmiotu;     // tekst mówiący o tym, jak nazywa się aktywnie wybrany przedmiot
    public Canvas KanwaOpisuAktywnegoPrzedmiotu;
    // informuje i ustawia plecak rozszerzony do wszystkich przedmiotów
    //...fragment kodu...
    public bool Rozszerzony
    {
        get { return _rozszerzony; }
        set
        {
            //...fragment kodu...
            OdswiezWszystko();
        }
    }
    //...fragment kodu...
    // przesuwa wybrany panel o przesunięcie (1 w prawo, -1 w lewo)
    public void PrzesunWybranyPanel(int przesuniecie)
    {
        //...fragment kodu...
        OdswiezWszystko();
    }
    private void OdswiezWszystko()
    {
        OdswiezPozycjePaneliUI();
        OdswiezPozycjePrzedmiotow();
        OdswiezInformacjeOAktywnymPrzedmiocie();
    }
    private void OdswiezInformacjeOAktywnymPrzedmiocie()
    {
        if(Rozszerzony && this[_wybranyPanel] != null)
        {
            OpisAktywnegoPrzedmiotu.text = this[_wybranyPanel].Nazwa;
            OpisAktywnegoPrzedmiotu.gameObject.SetActive(true);
        } else
        {
            OpisAktywnegoPrzedmiotu.gameObject.SetActive(false);
        }
    }
}
```

Obsługa efektów podnoszenia przedmiotu

Podnoszenie przedmiotu powinno rozpocząć się od krótkiej, efektownej animacji przedmiotu — podnoszenia się, rotowania i zmniejszania się przedmiotu. Dla czytelności kodu całość tego efektu będzie wykonywana przez osobny komponent, który utworzymy i dodamy do przedmiotu w odpowiednim momencie. Po zakończeniu efektu usuniemy komponent efektu i wywołamy znany już kod dodawania przedmiotu do plecaka.

Zmiany pozycji, rotacji i skali wprowadzamy w metodzie `Update`, w której sukcesywnie zliczamy czas, jaki upłynął od startu. Kiedy już znamy czas całkowity efektu, wiemy, w jakim momencie się znajdujemy (jaki procent całości czasu upłynął i jaki procent przesunięcia, rotacji i skali całkowitej mamy zastosować). Warto wspomnieć o metodzie `Vector3.Lerp` — potrafi ona zastąpić kilka linii kodu obliczeń matematycznych. Podajemy jej wektor startowy, wektor docelowy, a ona oblicza, jaki powinien być wektor pośredni, biorąc pod uwagę trzeci parametr — ułamek. Może to być ułamek sekundy, gdybyśmy podawali przesunięcia na sekundę, albo ułamek całego czasu, jak w naszym przykładzie — podajemy przesunięcia na cały czas trwania efektu. Funkcję `Lerp` posiada również `Quaternion`, ale dla odmiany używamy do obliczenia rotacji innej metody — ustawiamy obiekt w rotacji początkowej, a potem obracamy go metodą `Rotate` wokół osi pionowej o aktualny kąt, który obliczamy sobie, mnożąc całkowite przesunięcie przez iloraz czasu, który upłynął, do całkowitego czasu.

Po upłynięciu czasu informujemy komponent `Przedmiot` o tym, że efekt został zakończony, a on usuwa komponent `EfektPodnoszenia`. Metoda `Destroy` zastosowana na komponencie usuwa tylko ten komponent (w przeciwieństwie do zastosowania jej na właściwości `gameObject`, co usuwa cały obiekt ze sceny wraz ze wszystkimi jego komponentami). Dodanie komponentu także jest możliwe — można do tego celu użyć metody `AddComponent`. Całość zmian na listingu 3.12.

LISTING 3.12. Klasa EfektPodnoszenia oraz niezbędne do jej wprowadzenia zmiany w klasie Przedmiot

```
public class EfektPodnoszenia : MonoBehaviour
{
    private const float CZASCALKOWITY = 0.75f;
    private readonly Vector3 PRZESUNIECIECALKOWITE = new Vector3(0, 1f, 0);
    private const float OBROTCALKOWITY = 540;
    private Vector3 _pozycjaStartowa, _skalaStartowa;
    private Quaternion _rotacjaStartowa;
    private float _czasUplynal;
    void Start()
    {
        _czasUplynal = 0;
        _pozycjaStartowa = transform.localPosition;
        _rotacjaStartowa = transform.localRotation;
        _skalaStartowa = transform.localScale;
    }
```

```
    void Update()
    {
        _czasUplynal += Time.deltaTime;
        if (_czasUplynal >= CZASCALKOWITY)
            ZatrzymajEfekt();
        else
        {
            transform.localPosition = Vector3.Lerp(_pozycjaStartowa, _pozycjaStartowa +
            ↪PRZESUNIECIECALKOWITE, _czasUplynal / CZASCALKOWITY);
            transform.localRotation = _rotacjaStartowa;
            transform.Rotate(transform.up, OBROTCALKOWITY * _czasUplynal / CZASCALKOWITY);
            transform.localScale = Vector3.Lerp(_skalaStartowa, _skalaStartowa * (1 -
            ↪_czasUplynal / CZASCALKOWITY), _czasUplynal / CZASCALKOWITY);
        }
    }
    private void ZatrzymajEfekt()
    {
        GetComponent<Przedmiot>().FinalizujDodanieDoPlecaka();
    }
}
public class Przedmiot : MonoBehaviour
{
    //...fragment kodu...
    public void DodajDoPlecaka()
    {
        if (GetComponent<Collider>() != null)
            GetComponent<Collider>().enabled = false;
            gameObject.AddComponent<EfektPodnoszenia>();
    }
    public void FinalizujDodanieDoPlecaka()
    {
        Destroy(GetComponent<EfektPodnoszenia>());
        MenedzerGry.InstancjaMenedzeraGry.ObiektPlecaka.Dodaj(this);
    }
}
```

Warto zastanowić się chwilę nad kodem, który wrzuciliśmy do metody `Update`. Mamy tam kilka działań matematycznych, które nie muszą być szybkie. Czy to problem? W tym konkretnym przypadku nie, ponieważ wiemy, że będziemy używać tego komponentu tylko do efektów podnoszenia przedmiotów, a więc jednocześnie może być uruchomionych maksymalnie kilka instancji komponentu (jeśli gracz jest szybki, to w ciągu trwania efektu może spróbować podnieść coś jeszcze, ale raczej nigdy nie uda mu się podnieść w ten sposób więcej niż 2 – 3 przedmiotów). Taka ilość obliczeń w metodach `Update` będzie niezauważalna dla współczesnych procesorów, nawet mobilnych. Gdybyśmy jednak nagle zapragnęli używać tego komponentu dla kilkuset obiektów naraz, to mogłoby to mieć już duży wpływ na wydajność gry i należałoby rozważyć zmiany w kodzie albo inne podejście. Metoda `Update` to krytyczne miejsce naszej gry i zawsze powinniśmy dobrze rozumieć, co się w nich dzieje. Dotyczy to także wszystkich komponentów stworzonych przez kogoś innego, które włączamy do naszego projektu!

Obsługa wyrzucania przedmiotu na scenę

Wyrzucenie przedmiotu wiąże się z usunięciem go z plecaka oraz nadaniem mu ponownie właściwości, jakie posiadał (np. jeśli miał wyłączony kolider, to należy go włączyć, należy też przywrócić oryginalną skalę obiektu). Jak to jednak zrobić fizycznie, aby przedmiot poleciał w wirtualnym powietrzu i upadł na wirtualną ziemię? Ma wprawdzie kolider, ale nie reaguje fizycznie z ziemią, innymi przedmiotami i jest niewrażliwy na grawitację. Potrzebny jest do tego komponent o nazwie `Rigidbody` odpowiedzialny za właściwości **bryły sztywnej**. Komponent taki nie tylko umożliwia reagowanie obiektu na grawitację (i ma parametr określający masę obiektu, bez tego oczywiście nie byłoby mowy o działaniu grawitacji), lecz także ma szereg przydatnych właściwości i funkcji, np. można pchnąć go z pewną siłą. To wszystko, co będzie nam potrzebne, najpierw jednak musimy się upewnić, że przedmiot posiada taki komponent, a jeśli jest inaczej — utworzyć go.

Podczas obsługi wyrzucania obiektu w metodzie `Przedmiot.UsunZPlecaka`, po przywróceniu właściwego stanu przedmiotowi, przesuwamy jego pozycję na pozycję kamery plus wektor `forward` kamery, czyli około metra w kierunku, w którym patrzy gracz. Dzięki temu przedmiot wypadnie w jego polu widzenia, a nie np. nad jego głową czy tuż pod nogami. Następnie dodajemy nieco siły metodą `Rigidbody.AddForce`. Siła jest oczywiście wektorem, kierujemy go również w kierunku patrzenia kamery i mnożymy jeszcze przez skalar – siła musi być na tyle duża, żeby była zauważalna (wektor `forward` to wektor jednostkowy, jak na siłę liczoną w niutonach to za mało). I to wszystko. Wyrzucony przedmiot będzie odbijać się nie tylko od ziemi, lecz także innych przedmiotów, drzew, domku czy pacjentów. Można go później podnieść i wyrzucić inny. Łatwe powinno być też zaimplementowanie wyrzucania aktywnie wskazywanego przedmiotu z plecaka po naciśnięciu klawisza. Zostawiam to jako ćwiczenie Czytelnikowi.

Zmiany w kodzie — patrz listing 3.13.

LISTING 3.13. Zmiany w kodzie w klasach Przedmiot i Plecak

```
public class Przedmiot : MonoBehaviour
{
    //...fragment kodu...
    private Vector3 _oryginalnaSkala;
    public void DodajDoPlecaka()
    {
        _oryginalnaSkala = transform.localScale;
        if (GetComponent<Rigidbody>() != null)
           Destroy(GetComponent<Rigidbody>());
        if (GetComponent<Collider>() != null)
            GetComponent<Collider>().enabled = false;
        gameObject.AddComponent<EfektPodnoszenia>();
    }
    public void UsunZPlecaka()
    {
        if (GetComponent<Collider>() != null)
            GetComponent<Collider>().enabled = true;
        if (GetComponent<Rigidbody>() == null)
```

```
        {
            gameObject.AddComponent<Rigidbody>();
            GetComponent<Rigidbody>().mass = 1;
        }
        transform.localScale = _oryginalnaSkala;
        transform.parent = MenedzerGry.InstancjaMenedzeraGry.transform;
        transform.position = Camera.main.transform.position +
        ↪Camera.main.transform.forward;
        GetComponent<Rigidbody>().AddForce(Camera.main.transform.forward *150);
    }
}
public class Plecak : MonoBehaviour
{
    //...fragment kodu...
    private void PrzedmiotZostalUsuniety(Przedmiot usuniety)
    {
        usuniety.UsunZPlecaka();
    }
}
```

Tworzenie instancji przedmiotu leczniczego w przedmiocie

Klasa `Przedmiot` jest odpowiedzialna za reprezentację przedmiotów na scenie i w plecaku. Dodamy także to, że będzie pośrednikiem między przedmiotem na scenie a jego wewnętrznymi danymi i logiką, np. logiką leczenia. Czy powinniśmy to pośrednictwo rozdzielić? Obecnie wygląda na to, że nie, ponieważ logika ta będzie niewielka i wynikająca wprost z bycia obiektem na scenie. Dlatego klasa `Przedmiot` będzie posiadała pole typu `PrzedmiotLeczniczy` o nazwie `WewnetrznyPrzedmiotLeczniczy`. Problemem do rozstrzygnięcia jest jeszcze, jak to pole inicjalizować. Kluczem do tego jest klasa `MenedzerZielarstwa` i jej listy wzorców przedmiotów. Może ona zainicjować odpowiedni przedmiot leczniczy, a zidentyfikować możemy go po nazwie (nazwa przedmiotu leczniczego będzie musiała być identyczna jak typ aktywnego obiektu na scenie). Niestety `MenedzerZielarstwa` ładuje swoje dane dopiero podczas pierwszego wywołania metod `Update`, a więc nie wiemy, gdzie i jak podpiąć to ładowanie informacji o przedmiocie leczniczym w klasie `Przedmiot`. Dlatego musimy jeszcze zrobić drobny refaktoring i przenieść to ładowanie do metody `Awake`. Krok po kroku do wykonania są następujące czynności:

- Wywołanie ładowania danych klasy `MenedzerZielarstwa` zlokalizowane w `MenedzerGry.Update` przenosimy do `MenedzerGry. Awake`.
- W klasie `PrzedmiotLeczniczy` utworzymy konstruktor kopiujący, który będzie tworzyć klon przedmiotu podanego jako wzorzec.
- Do `MenedzerZielarstwa` dodajemy funkcję `UtworzPrzedmiotLeczniczy`, która szuka po nazwie i zwraca utworzoną instancję przedmiotu leczniczego bądź `null`, jeśli nie uda się zrobić dopasowania.
- W klasie `Przedmiot` w funkcji `Start` wywołujemy `MenedzerZielarstwa.UtworzPrzedmiotLeczniczy` i przypisujemy zwrot do właściwości `WewnetrznyPrzedmiotLeczniczy`.

Zmiany w kodzie realizujące powyższy plan — patrz listing 3.14 (zmiana w klasie `MenedzerGry` jest banalna i polega wyłącznie na przeniesieniu jednej linii kodu na koniec funkcji `Awake`, dlatego nie podajemy jej tu na listingu).

LISTING 3.14. Zmiany w kodzie klas MenedzerZielarstwa, PrzedmiotLeczniczy i Przedmiot

```
public sealed class MenedzerZielarstwa
{
    //...fragment kodu...
    // próbuje dopasować wzorzec przedmiotu leczniczego do podanego typu i jeśli mu się uda — tworzy
    // instancję klasy PrzedmiotLeczniczy i ją zwraca
    public PrzedmiotLeczniczy UtworzPrzedmiotLeczniczy(string szukanyTyp)
    {
        PrzedmiotLeczniczy wzorzec = null;
        foreach (PrzedmiotLeczniczy przedmiot in _wzorcePrzedmiotow)
            if (przedmiot.Nazwa == szukanyTyp)
                wzorzec = przedmiot;
        if (wzorzec != null)
            return new PrzedmiotLeczniczy(wzorzec);
        return null;
    }
}
public class PrzedmiotLeczniczy
{
    //...fragment kodu...
    // skopiuj z innego przedmiotu-wzorca
    public PrzedmiotLeczniczy(PrzedmiotLeczniczy wzorzec)
    {
        AktywneSubstancje = new List<Substancja>();
        Nazwa = wzorzec.Nazwa;
        CzasDzialania = wzorzec.CzasDzialania;
        foreach (Substancja wzorzecSubstancja in wzorzec.AktywneSubstancje)
            AktywneSubstancje.Add(new Substancja(wzorzecSubstancja));
    }
}
public class Przedmiot : MonoBehaviour
{
    //...fragment kodu...
    public PrzedmiotLeczniczy WewnetrznyPrzedmiotLeczniczy
    {
        get;  private set;
    }
    //...fragment kodu...
    private void Start()
    {
        _komponentAO = GetComponent<AktywnyObiekt>();
        WewnetrznyPrzedmiotLeczniczy = MenedzerZielarstwa.Instancja.
        ↪UtworzPrzedmiotLeczniczy(_komponentAO.Typ);
    }
}
```

Przekazywanie przedmiotów pacjentom

Przekazywanie pacjentom przedmiotów od strony UI zorganizowane będzie prosto. Wystarczy najechać wskaźnikiem myszki na pacjenta i nacisnąć klawisz *E*, a aktualnie wybrany przedmiot z plecaka zostanie mu przekazany. Oczywiście pod warunkiem, że taki przedmiot to przedmiot leczniczy. Pacjenci, aby mogli być leczeni, potrzebują mieć własną listę przedmiotów leczniczych. Przedmiot do przejęcia klasie `Pacjent` przekaże klasa `KomponentPacjent`, którą informować będzie `MenedzerUI` o naciśnięciu klawisza *E* na pacjencie (jak pamiętamy, mamy już w tej klasie rozpoznawanie, na jaki obiekt patrzymy, wystarczy dodać obsługę klasy `KomponentPacjent`). `KomponentPacjent` będzie decydował, czy przyjąć `Przedmiot`. Po wszystkim przedmiot musi oczywiście być usunięty z plecaka. W klasie `Plecak` musimy natomiast dodać właściwość `AktywnyPrzedmiot`, która powie nam, który przedmiot jest aktualnie wybrany, oraz funkcję `ZniszczPoCichuPrzedmiot`, która usunie przedmiot z plecaka, nie rzucając go na ziemię. Przedmiot całkiem niszczymy, usuwając go ze sceny, a pacjent dostanie tylko kopię obiektu klasy `PrzedmiotLeczniczy`. Zmiany są rozproszone na wiele klas. Pokazuje je listing 3.15.

LISTING 3.15. Zmiany umożliwiające przekazywanie przedmiotów pacjentom

```
public class Pacjent: Postac
{
    //...fragment kodu...
    private List<PrzedmiotLeczniczy> _przedmiotyLeczace; // lista przedmiotów aktualnie leczących pacjenta
    public Pacjent() : base()
    {
        //...fragment kodu...
        _przedmiotyLeczace = new List<PrzedmiotLeczniczy>();
    }
    public void DodajPrzedmiotLeczacy(PrzedmiotLeczniczy nowyPrzedmiot)
    {
        _przedmiotyLeczace.Add(nowyPrzedmiot);
    }
    public string ZrzutDebugZdarzen()
    {
        //...fragment kodu...
        foreach (PrzedmiotLeczniczy przedmiotL in _przedmiotyLeczace)
            log.Append(string.Format(" %% {0}: {1} \n\r", TlumaczCiagow.PodajCiag("Przedmiot
            ↪Leczacy"), TlumaczCiagow.PodajCiag("Przedmiot" + przedmiotL.Nazwa)));
        //...fragment kodu...
    }
}
public class KomponentPacjent : MonoBehaviour
{
    //...fragment kodu...
    // przekazuje pacjentowi przedmiot, zwraca true, jeśli przedmiot został przyjęty, lub false, jeśli nie
    public bool DodajPrzedmiotLeczacy(Przedmiot nowyPrzedmiot)
    {
        if (nowyPrzedmiot == null || nowyPrzedmiot.WewnetrznyPrzedmiotLeczniczy == null)
            return false;
        WewnetrznyPacjent.DodajPrzedmiotLeczacy(new PrzedmiotLeczniczy(nowyPrzedmiot.
        ↪WewnetrznyPrzedmiotLeczniczy));
        if (_infoPacjenta != null)
```

```
            _infoPacjenta.text = WewnetrznyPacjent.ZrzutDebugZdarzen();
        return true;
    }
}
public class Plecak : MonoBehaviour
{
    //...fragment kodu...
    // przedmiot na aktywnej pozycji „pozycja”, null oznacza brak przedmiotu na aktywnej pozycji
    public Przedmiot AktywnyPrzedmiot
    {
        get
        {
            return this[_wybranyPanel];
        }
    }
    //...fragment kodu...
    // usuwa przedmiot z plecaka, nie wyrzucając go na ziemię (zniszczony, przekazany komuś itp.)
    public void ZniszczPoCichuPrzedmiot(Przedmiot ktoryPrzedmiot)
    {
        int indeksZnaleziony = -1;
        for (int i = 0; i < ROZMIARPLECAKA; i++)
            if (_przedmioty[i] == ktoryPrzedmiot)
                indeksZnaleziony = i;
        if (indeksZnaleziony != -1)
        {
            _przedmioty[indeksZnaleziony] = null;
            Destroy(ktoryPrzedmiot.gameObject);
        }
    }
    //...fragment kodu...
}
public class MenedzerUI : MonoBehaviour
{
    //...fragment kodu...
    void Update()
    {
        //...fragment kodu...
        if (Input.GetKeyDown(KeyCode.E))
        {
            if (patrzymy != null)
            {
                if (patrzymy.GetComponent<Przedmiot>() != null)
                    patrzymy.GetComponent<Przedmiot>().DodajDoPlecaka();
                else if (patrzymy.transform.parent != null &&
                ↪patrzymy.transform.parent.GetComponent<KomponentPacjent>() != null)
                {
                    Przedmiot aktywny = MenedzerGry.InstancjaMenedzeraGry.
                    ↪ObiektPlecaka.AktywnyPrzedmiot;
                    bool przyjetoPrzedmiot = patrzymy.transform.parent.GetComponent
                    ↪<KomponentPacjent>().DodajPrzedmiotLeczacy(aktywny);
                    if (przyjetoPrzedmiot)
                        MenedzerGry.InstancjaMenedzeraGry.ObiektPlecaka.ZniszczPoCichuPrzedmiot
                        ↪(MenedzerGry.InstancjaMenedzeraGry.ObiektPlecaka.AktywnyPrzedmiot);
                }
            }
        }
    }
}
```

Leczenie pacjentów przedmiotami leczniczymi

Po tym wszystkim, co już wprowadziliśmy w tym sprincie, samo leczenie będzie już dość proste. Pacjent podczas obsługi godzinowej musi dla każdego swojego zdarzenia wyszukać, czy któryś z posiadanych przez niego przedmiotów zawiera substancję leczącą dane zdarzenie, oraz zainicjować zużycie się części przedmiotu leczniczego. Część tych działań będzie oddelegowana do klasy `PrzedmiotLeczniczy` dla lepszej czytelności kodu. Warto zauważyć, że przy wyszukiwaniu leczenia mamy do czynienia w sumie z kilkoma zagnieżdżonymi pętlami. Jest to sytuacja potencjalnie ryzykowna wobec wydajności działania gry. Na razie, o ile testy nie wykażą problemów, nie będziemy z tym nic robić. Zdarzenia przetwarzane są rzadko, a pacjentów jest na razie niewielu, więc problem nie powinien być poważny. Zmiany — patrz listing 3.16.

LISTING 3.16. Leczenie pacjentów przedmiotami przez nich posiadanymi

```
public class Pacjent: Postac
{
    //...fragment kodu...
    // obsługa zdarzeń medycznych godzinowo
    public void GodzinowaObslugaZdarzen()
    {
        //...fragment kodu...
        foreach(ZdarzenieMedyczne zdarzenie in _aktualneZdarzenia)
        {
            //...fragment kodu...
            // obsłużmy jeszcze witalność pacjenta
            zdarzenie.Sila = zdarzenie.Sila * (1 - PobierzZdolnosc("Witalnosc") / 5000f)
            ↪- 0.01f * (PobierzZdolnosc("Witalnosc") / 50f);
            // oraz leczenie przedmiotami
            zdarzenie.Sila -= PodajGodzinoweLeczeniePrzedmiotem(zdarzenie);
        }
        //...fragment kodu...
        ObsluzZuzycieGodzinowePrzedmiotow();
    }
    private float PodajGodzinoweLeczeniePrzedmiotem(ZdarzenieMedyczne zdarzenie)
    {
        foreach (PrzedmiotLeczniczy przedmiot in _przedmiotyLeczace)
        {
            float leczenie = przedmiot.PodajGodzinoweLeczenie(zdarzenie);
            if (leczenie != 0)
                return leczenie;
        }
        return 0;
    }
    private void ObsluzZuzycieGodzinowePrzedmiotow()
    {
        List<PrzedmiotLeczniczy> doUsuniecia = new List<PrzedmiotLeczniczy>();
        foreach(PrzedmiotLeczniczy przedmiot in _przedmiotyLeczace)
        {
            przedmiot.ZuzyjDawke();
            if (przedmiot.PozostalyCzasDzialania <= 0)
                doUsuniecia.Add(przedmiot);
        }
```

```
        foreach (PrzedmiotLeczniczy przedmiot in doUsuniecia)
            _przedmiotyLeczace.Remove(przedmiot);
    }
}
public class PrzedmiotLeczniczy
{
    //...fragment kodu...
    public void ZuzyjDawke()
    {
        PozostalyCzasDzialania--;
    }
    private float PodajGodzinoweLeczenie(ZdarzenieMedyczne zdarzenie)
    {
       foreach (Substancja subst in AktywneSubstancje)
       {
            foreach (LeczenieZdarzenia leczenie in subst.Leczenie)
                if (leczenie.WartoscGraniczna <= subst.Wielkosc && leczenie.TypZdarzenia
                ↪== zdarzenie.Nazwa && (leczenie.GdzieLeczy & zdarzenie.Lokalizacja) !=
                ↪0 && PozostalyCzasDzialania > 0)
                    return leczenie.Leczenie / CzasDzialania;
        }
        return 0;
    }
}
```

Implementacja wyświetlania krótkiej wiadomości do gracza

W różnych sytuacjach będziemy potrzebowali wyświetlić na ekranie komentarz do stanu gry, informację dla gracza itp. W tym sprincie wprowadzamy ograniczenie polegające na tym, że gdy gracz podejdzie zbyt blisko brzegu mapy, zostanie cofnięty wraz z zabawnym komunikatem. Wykorzystamy do tego właśnie tę nowo tworzoną funkcję. Wprowadzimy ją jako dodatkowy tekst wyświetlany obok wskaźnika kamery (poniżej niego), podobnie jak informację o nazwie przedmiotu, który pokazuje wskaźnik. Możemy na scenę dodać pole tekstowe, podobnie jak to robiliśmy dla nazwy wskazywanego przedmiotu. Musimy jednak przemyśleć kwestię kanwy oraz zrobienia niewielkiego refaktoringu w klasie `MenedzerUI`. Jeżeli dodamy teksty do istniejącej już kanwy, w której znajdują się teksty opisujące przedmiot, to po pierwsze nie będziemy mogli łatwo sterować widocznością kanwy opisu przedmiotów, jak to robimy do tej pory, po drugie będziemy mieli niezbyt czytelny kod i nadmiar pól publicznych. A jeśli zrobimy tylko dwie kanwy, to nie do końca poprawnie zadziała używana przez nas metoda `GetComponentInChildren` pobierająca kanwę. Dlatego zrobimy następujące zmiany:

- Wprowadzimy osobną kanwę dla pól opisu przedmiotu i krótkiej informacji.
- Obie kanwy będą przypisywane do pola publicznego w `MenedzerUI`, a teksty będą w polach prywatnych, dodatkowo jako tablice tekstów, a nie osobne pola.
- Teksty pobieramy metodą `GetComponentsInChildren` dla kanw.

Zmiany dla tego refaktoringu można zobaczyć na listingu 3.17.

LISTING 3.17. Refaktoring klasy MenedzerUI

```
public class MenedzerUI : MonoBehaviour
{
    //...fragment kodu...
    public Canvas KanwaOpisu, KanwaMalegoInfo;
    private Text[] _tekstyOpisuPrzedmiotu;
    private Text[] _tekstyMalegoInfo;
    void Start()
    {
        //...fragment kodu...
        if (KanwaMalegoInfo == null)
            Debug.LogError("Kawna małego info nie jest ustawiona!");
        if (KanwaOpisu == null)
            Debug.LogError("Kawna opisu przedmiotu nie jest ustawiona!");
        _tekstyMalegoInfo = KanwaMalegoInfo.GetComponentsInChildren<Text>();
        _tekstyOpisuPrzedmiotu = KanwaOpisu.GetComponentsInChildren<Text>();
        KanwaMalegoInfo.gameObject.SetActive(false);
        KanwaOpisu.gameObject.SetActive(false);
    }
    void Update()
    {
        //...fragment kodu...
        if (_poprzednioPatrzylismy != patrzymy)
        {
            if (patrzymy != null && patrzymy.GetComponent<AktywnyObiekt>() != null)
            {
                _wskaznikKamery.sprite = WskaznikWlaczony;
                foreach(Text t in _tekstyOpisuPrzedmiotu)
                    t.text = patrzymy.GetComponent<AktywnyObiekt>().Nazwa;
                KanwaOpisu.gameObject.SetActive(true);
            }
    //...fragment kodu...
    }
}
```

Obsługę dodanych tekstów i krótkiej wiadomości zrobimy tak, że nowa publiczna funkcja `PokazMaleInfo` będzie ustawiać tekst polom tekstowym krótkiej wiadomości, włączać widoczność ich kanwy oraz startować licznik `_licznikCzasuMalegoInfo`, który będzie obsługiwany w metodzie `Update`. Gdy licznik ten wskaże zero, wyłączymy kanwę krótkiej wiadomości, w ten sposób ukrywając ją na ekranie. Można to zobaczyć na listingu 3.18.

LISTING 3.18. Obsługa krótkich wiadomości do gracza w klasie MenedzerUI

```
public class MenedzerUI : MonoBehaviour
{
    public const float CZASMALEGOINFO = 3.5f;    // czas wyświetlania się małego komunikatu
                                                 // tekstowego w UI
    //...fragment kodu...
    private float _licznikCzasuMalegoInfo;
    void Update()
    {
        //...fragment kodu...
        if(_licznikCzasuMalegoInfo >= 0)
```

```
        {
            _licznikCzasuMalegoInfo -= Time.deltaTime;
            if(_licznikCzasuMalegoInfo <= 0)
                KanwaMalegoInfo.gameObject.SetActive(false);
        }
    }
    //...fragment kodu...
    public void PokazMaleInfo(string tekst)
    {
        foreach (Text t in _tekstyMalegoInfo)
            t.text = tekst;
        KanwaMalegoInfo.gameObject.SetActive(true);
        _licznikCzasuMalegoInfo = CZASMALEGOINFO;
    }
}
```

Klasa KontrolerGracza oraz zawracanie gracza, gdy podejdzie zbyt blisko skraju mapy

Gdy obsługiwaliśmy ładowanie stany gry w klasie `MenedzerGry`, przypisywaliśmy bezpośrednio pozycję gracza załadowaną z pliku obiektowi będącemu kontrolerem pierwszoosobowym. To mogło być dobre przy tak prostym kodzie, ale teraz, kiedy musimy wykonać historyjkę mówiącą o zawracaniu gracza znad brzegu mapy i z tego powodu zaczynamy manipulować kontrolerem pierwszoosobowym w znacznie większym stopniu, potrzebujemy rozbudowy i utworzenia osobnego komponentu zarządzającego położeniem i innymi cechami kontrolera pierwszoosobowego, a co za tym idzie — pozycją i widokiem gracza. Utworzymy komponent o nazwie `KontrolerGracza` i przypniemy go do obiektu na scenie o nazwie *FPSController*. Będzie on m.in. posiadał metodę publiczną `PrzesunGracza`, przyjmującą jako argument wektor położenia i zmieniającą położenie gracza na podany argument. Wykorzystamy to, sprawdzając co pewien czas, czy pozycja gracza nie wyszła poza ustalone przez nas granice mapy. Jeśli wyszła, to przesuwamy gracza z powrotem do ostatniej znanej nam pozycji, w której gracz znajdował się jeszcze w ustalonych granicach. Przy okazji użyjemy też dodanej niedawno funkcji wyświetlania graczowi krótkiej informacji i cofając go, wyświetlimy ją. (Poza tym, że tak jest zabawnie, to taki tekst pełni ważną rolę z punktu widzenia UX [ang. *user experience*], ponieważ cofając gracza bez słowa wyjaśnienia, wprowadzimy zamieszanie — gracz nie będzie wiedział, dlaczego widok nagle się zmienił, i najprawdopodobniej uzna, że gra zawiera poważne błędy. W nietypowych sytuacjach warto zwracać graczowi uwagę, co się dzieje, najlepiej na kilka sposobów). Kod klasy — patrz listing 3.19.

LISTING 3.19. Kod nowej klasy KontrolerGracza

```
public class KontrolerGracza: MonoBehaviour
{
    public const float MAXXBRZEGU = 250;
    public const float MAXZBRZEGU = 250;
    public const float MINXBRZEGU = -250;
    public const float MINZBRZEGU = -250;
```

```
    public const float CZESTOSCBADANIA = 4;
    private Vector3 _ostatniaBezpiecznaPozycja;
    private float _licznikCzasu;
    private bool _ruchWylaczony;
    private void Start()
    {
        _licznikCzasu = CZESTOSCBADANIA;
        _ruchWylaczony = false;
    }
    private void Update()
    {
        _licznikCzasu -= Time.deltaTime;
        if(_licznikCzasu <= 0)
        {
            _licznikCzasu = CZESTOSCBADANIA;
            ZbadajBrzegMapy();
        } else
        if (_ruchWylaczony)
        {
            _ruchWylaczony = false;
            GetComponent<FirstPersonController>().enabled = true;
        }
    }
    private void ZbadajBrzegMapy()
    {
        if (transform.position.x < MINXBRZEGU || transform.position.x > MAXXBRZEGU ||
        ↪transform.position.z < MINZBRZEGU || transform.position.z > MAXZBRZEGU)
        {
            PrzesunGracza(_ostatniaBezpiecznaPozycja);
            MenedzerUI.Instancja.PokazMaleInfo(TlumaczCiagow.PodajCiag("InfoBrzegowe"));
        }
        else
            _ostatniaBezpiecznaPozycja = gameObject.transform.position;
    }
    public void PrzesunGracza(Vector3 nowaPozycja)
    {
        GetComponent<FirstPersonController>().enabled = false;
        gameObject.transform.position = nowaPozycja;
        _ruchWylaczony = true;
    }
}
```

Tu warto jeszcze chwilę pochylić się nad sposobem, w jaki przesuwamy gracza. Dlaczego po prostu nie zmieniamy mu `transform.position`, tylko stosujemy skomplikowany mechanizm z manipulowaniem polem `enabled` na komponencie `FirstPersonController`? Robimy tak, ponieważ proste podejście zwyczajnie... nie zadziała. Zmianę pozycji kontrolera inicjujemy z metody `Update` (gdybyśmy to robili np. z `FixedUpdate`, to też nic by nie dało), ale nie zostanie ona wprowadzona w życie przed końcem klatki. Jednocześnie w tej samej klatce kontroler pierwszoosobowy wykonuje swoje przesunięcia postaci gracza, nieświadomie wracając na starą pozycję i psując nam zmianę. Być może z innym kontrolerem sama zmiana pozycji zadziałałaby, ale dopóki pracujemy na standardowym `FirstPersonController`, musimy na jedną klatkę go wyłączyć i dopiero gdy nasza zmiana

zadziała, włączyć ponownie. Komponent, który zostanie wyłączony (przez pole `enabled`), nie uczestniczy w wywoływaniu funkcji cyklu życia komponentu, np. `Update` czy `FixedUpdate`, więc kontroler pierwszoosobowy nie będzie przez tę klatkę korygować swojej pozycji. Gracz oczywiście nie zauważy braku możliwości ruchu trwającego jedną klatkę, tym bardziej że zmienia mu się widok w wyniku naszego przesunięcia.

Podsumowanie trzeciego sprintu

W trzecim sprincie opanowaliśmy mocniej system przedmiotów i związanego z nimi plecaka. Użycie przedmiotów pozwoliło na domknięcie cyklu rozgrywki z pacjentem poprzez umożliwienie leczenia go przedmiotami podniesionymi ze sceny. Wprowadziliśmy także sposoby na zapis i odczyt stanu gry, dzięki czemu rozgrywka nie restartuje się za każdym razem (nadal można to zrobić poprzez ręczne usunięcie pliku z zapisanym stanem gry). W trakcie prac można było zauważyć, że znacznie częściej niż wcześniej dotykają one wielu miejsc w kodzie, znacznie częściej są to istniejące miejsca, a rzadziej tworzymy nowe klasy i całkiem nowe podsystemy. Wiele prac wymaga najpierw mniejszych bądź większych refaktoringów, aby można było je sprawnie przeprowadzać. To naturalna konsekwencja rozwoju projektu i można się spodziewać, że w następnych sprintach tendencje te się nasilą. Typowym przykładem refaktoringu jest m.in. refaktoring, jaki wykonaliśmy pod koniec rozdziału w klasie `MenedzerUI`. Jej postać była rozsądna w poprzednim sprincie, ale rozszerzenia wymagały zmiany konstrukcji istniejących fragmentów, aby nowe mogły działać sprawnie i wyglądać czytelnie.

Skorowidz

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Z

Osiągnij wyższy poziom z Unity i C#!

- Poznaj zaawansowane techniki programowania
- Dowiedz się, jak realizować projekty informatyczne
- Naucz się tworzyć grę od strony praktycznej

Połączenie platformy Unity i języka C# zapewnia doskonałe środowisko do tworzenia i rozwijania różnego rodzaju gier komputerowych. To sprawia, że tandem ten jest niezwykle popularny wśród game developerów pragnących szybko i wydajnie osiągać profesjonalne efekty. Jednak sama znajomość narzędzi nie wystarczy, aby sprostać niełatwemu zadaniu zaprojektowania i zaprogramowania prawdziwej gry — by przekuć pomysł i umiejętności w prawdziwy produkt, trzeba czegoś więcej.

Niezbędne doświadczenie możesz zdobyć tylko w praktyce i na realnych przykładach, a takie zapewni Ci właśnie ta książka! Pozbawiona zbędnej teorii, oparta na prawdziwym przypadku i do bólu praktyczna, pozwoli Ci szybko poznać zaawansowane techniki tworzenia gier komputerowych oraz wdrożyć się w proces opracowywania projektu przy użyciu metodyki zwinnej. Krok po kroku, iteracja za iteracją będziesz towarzyszyć autorowi w pracy nad komercyjnym produktem — przejdziesz wszystkie fazy jego rozwoju: od programowania, poprzez usuwanie błędów, po publikację gotowej gry.

W książce:

- Tworzenie obiektów scen środowiska Unity oraz ich komponentów
- Zaawansowane techniki w języku C# przydatne twórcom gier
- Praktyczne zastosowanie różnych wzorców projektowych
- Zarządzanie obiektami, wirtualnym czasem gry i jej stanem
- Realizacja rozgrywki w widoku pierwszoosobowym
- Usuwanie błędów i testowanie balansu gry
- Dostosowanie gry do platform mobilnych oraz VR
- Publikacja gry w popularnych sklepach cyfrowych
- Zastosowanie metodyki zwinnej w projekcie gry

Programuj gry jak profesjonalista!

Helion

helion.pl

HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

ISBN 978-83-283-6586-5

9 788328 365865

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 59,00 zł