Piszemy RPGo-Platformówkę (2) – Zwyciężaj albo giń! (sheadovas/poradniki/proj_platf_rpg/2-zwyciezaj-albo-gin/)

Mar 18, 2017 / proj_platf_rpg (sheadovas/category/poradniki/proj_platf_rpg/)

Piszemy mechanizm pozwalający na dynamiczne tworzenie zasad gry (wygranej / przegranej).

Hej Wszystkim! Dzisiaj postanowiłem wziąć na tapetę napisanie mechanizmu, który pozwala na wykrywanie końca gry.

Tradycyjnie dla tej serii chciałbym Was poinformować, że omawiamy kod dostępny pod commit'em: [d9e9c80 (https://github.com/sheadovas/proj_platf_rpg/commit/d9e9c80809bced013420ebd0aae5346e9b332892)]. Dodatkowo zachęcam do zajrzenia na opis commit'a, który w skrócie opisuje ideę napisanego kodu. Zbudowana aplikacja dla Windows znajduje się w zakładce [Releases (https://github.com/sheadovas/proj_platf_rpg/releases/tag/1.2)] (na Github).

Bajka (Wprowadzenie)

Jak to w grach bywa, żądzą nimi pewne zasady (zbiór reguł), które mogą zadecydować o tym, że wygraliśmy bądź przegraliśmy, przez co dostaniemy odpowiednią informację o sposobie ukończenia gry.

W najprostszym przypadku gra posiada jedną globalną zasadę, która decyduje o wygranej / przegranej, np. dojście do pewnego miejsca w grze (przejście do następnego poziomu) lub odpowiednio utratę wszystkich punktów zdrowia.

(https://i2.wp.com/www.shead.ayz.pl/wp-content/uploads/2017/03/dark_souls_you_died.jpg)
Ekran końca gry (Dark Souls)

Jednakże jak to bywa w życiu, nie wszystko musi być takie proste i do przejścia jednego poziomu może być wymagany inny zbiór reguł (np. niestracenie życia), a w kolejnym inny zestaw zasad (np. obrona punktu przez 10min). Może także dojść do sytuacji gdy reguł jest więcej lub są bardziej skomplikowane (np. zbierz 10 roślin i porozmawiaj z osobą X lub wygraj 10 walk na arenie i porozmawiaj z osobą Y).

> UWAGA: Napisanie systemu tego typu może się przydać szczególnie w grze RPG, gdzie relacje z innymi postaciami mogą mieć wpływ na dostępne możliwości ukończenia zadania, np. pomogliśmy wcześniej postaci X i nie musimy zbierać już dla niej roślin.

Napisanie mechanizmu pozwalającego na elastyczne zmienianie zasad pozwala na wprowadzenie w łatwy sposób większej rozmaitości. Możliwe jest także wprowadzenie mini-gier w dość łatwy sposób.

Zwyciężaj albo giń!

Skoro wstęp fabularny mamy za sobą to warto przez chwilę się zastanowić co właściwie chcemy osiągnąć.

Chcemy aby system:

1. posiadał uniwersalny (i dynamiczny) system umożliwiający wprowadzenie zasad prowadzących do wygranej lub/i przegranej;
2. umożliwiał przypisanie zdarzeń po zajściu danego zdarzenia (np. przejście do ekranu końcowego);
3. umożliwiał tworzenie własnych (także skomplikowanych) zasad;
4. udostępniał ujednolicony interfejs;

Małe objaśnienie co do (1), chcemy obsłużyć sytuacje gdy:

- spełnienie warunku sprawia że gracz wygrywa, jego niespełnienie nie powoduje przegranej (np. dojście do określonego punktu powoduje wygraną, ale nie-dojście do niego nie powoduje przegranej);
- odwrotnie do sytuacji poprzedniej (np. utrata wszystkich punktów życia gwarantuje przegraną, ale posiadania jakichś punktów życia nie sprawia, że gracz wygrał);
- (warunek złożony) warunek można spełnić na 2 sposoby: pozytywny (wygrana) i negatywny (przegrana) (np. „zbierz X elementów w Y czasu” – zebranie X elementów przed upływem Y czasu powoduje wygraną, niespełnienie tego warunku przed upływem czasu powoduje przegraną).

Klasa *GameOverCondition* dla powyżej zadanych warunków prezentuje się następująco (omówienie później):

C#

```
using UnityEngine;

public abstract class GameOverCondition : MonoBehaviour
{
  public delegate void Action();
  public enum ConditionResult { NONE, SUCCESS, FAILURE };

  protected bool m_canBeSucceed = false;
  protected bool m_canBeFailed = false;

  protected Action m_actionOnSuccess;
  protected Action m_actionOnFailure;
  protected Action m_actionsOnUpdateCondition;


  public abstract string GetProgressInfo();

  public void AddActionOnSuccess(Action action)
  {
    m_canBeSucceed = true;
    m_actionOnSuccess += action;
  }

  public void AddActionOnFailure(Action action)
  {
    m_canBeFailed = true;
    m_actionOnFailure += action;
  }

  public void AddActionOnUpdate(Action action)
  {
    m_actionsOnUpdateCondition += action;
  }
```

Omówienie kodu (i idei)

Pierwszym wartym uwagi fragmentem kodu jest linia:

C#

```
  public enum ConditionResult { NONE, SUCCESS, FAILURE };
```

Widzimy tutaj, że nasze warunki mogą znajdować się w 3 stanach:

1. Nierozstrzygniętym (NONE);

2. Zakończonym: wygraną (SUCCESS);
3. Zakończonym: przegraną (FAILURE).

Dzięki temu rozróżnieniu będzie możliwe tworzenie bardziej skomplikowanych zasad ukończenia gry o czym wspominałem już powyżej.

> UWAGA! Powyższa implementacja mimo nazywania się GameOverCondition, wcale nie musi się kończyć ekranem końca gry. Warto zauważyć, że może służyć jako system zdarzeń, a ten możemy użyć do skryptowania gry (np. jeżeli gracz wejdzie do miasta i jest zraniony, to NPC zaproponuje mu pomoc).

Nieco dalej widzimy zestawi deklaracji:

```C#
8  protected bool m_canBeSucceed = false;
9  protected bool m_canBeFailed = false;
10
11 protected Action m_actionOnSuccess;
12 protected Action m_actionOnFailure;
13 protected Action m_actionsOnUpdateCondition;
```

Linie 8 i 9 mówią nam o tym czy dany warunek może zwrócić informację o wygranej / przegranej. Jedynym sposobem podniesienia tych flag jest dodanie delegatów *Action* (

`public delegate void Action();`

) do odpowiednich obiektów: m_actionOnSuccess / *m_actionOnFailure*. Dodanie zdarzeń do konkretnych stany końcowe jest możliwe poprzez użycie jednej z poniższych funkcji:

```C#
18 public void AddActionOnSuccess(Action action)
19 {
20   m_canBeSucceed = true;
21   m_actionOnSuccess += action;
22 }
23
24 public void AddActionOnFailure(Action action)
25 {
26   m_canBeFailed = true;
27   m_actionOnFailure += action;
28 }
29
30 public void AddActionOnUpdate(Action action)
31 {
32   m_actionsOnUpdateCondition += action;
33 }
```

Zdarzenie *onUpdate* może służyć do wywoływania funkcji po spełnieniu pewnej części zadania (np. zebraniu 2 z 10 przedmiotów), jego implementacja jest opcjonalna, a sama metoda domyślnie nie jest obsługiwana.

Funkcja *CheckConditions()* wymusza sprawdzenie warunków końca gry oraz uruchamia odpowiednie zdarzenia po ich zajściu.

Na nieco większą uwagę zasługuje poniższa funkcja:

```C#
```

```
58  protected virtual ConditionResult verifyResult()
59  {
60    /*
61     * Success strategy:
62     * checks conditions for success, then failure
63     */
64    if (m_canBeSucceed && isSuccess())
65      return ConditionResult.SUCCESS;
66
67    if (m_canBeFailed && isFailure())
68      return ConditionResult.FAILURE;
69
70    return ConditionResult.NONE;
71  }
```

Widzimy tutaj, że zanim nastąpi sprawdzenie warunków końca gry, to najpierw weryfikujemy czy flagi *canBeSucceed* / *canBeFailed* są podniesione. Konsekwencje ich niepodniesienia były opisane wyżej (*Condition* nie jest w stanie zwrócić danej wartości).

Możemy też zauważyć, że powyższa implementacja jest „przyjazna” graczowi, a to dlatego, że w razie gdyby zaszła sytuacja gdy spełnione są zasady dla wygranej i przegranej to wzięte pod uwagę zostaną wyłącznie warunki wygranej (można to łatwo zmienić odwracając kolejność wykonania if’ów).

Oprócz tego mamy 2 abstrakcyjne metody, które posiadają zbiór reguł decydujący o wygranej, bądź przegranej.

C#

```
55  protected abstract bool isSuccess();
56  protected abstract bool isFailure();
```

Oprócz tych metod musimy zaimplementować jeszcze jedną:

C#

```
16  public abstract string GetProgressInfo();
```

Służy ona do zwracania bieżącego stanu Warunku w postaci string’a. Jest to mechanizm, który ma za zadanie pomóc w aktualizowaniu GUI.

PoC (przykład użycia)

Osobiście bardzo lubię gdy dany mechanizm jestem w stanie łatwo przetestować na gotowym kodzie, dlatego poniżej przedstawiam Warunek „kończący grę” uruchamiający się w momencie gdy gracz dojdzie do punktu (gorąco zachęcam do samodzielnej analizy).

> UWAGA: kompletną grę platformową (wszystkie mechanizmy) połączymy we wpisie „Diabeł tkwi w szczegółach”.

C#

```C#
using UnityEngine;

[RequireComponent(typeof(BoxCollider2D))]
public class ConditionPointReached : GameOverCondition
{
  private BoxCollider2D m_box2d;
  private bool m_playerReached = false;
  private string m_info = "";

  public override string GetProgressInfo()
  {
    return m_info;
  }


  protected override bool isFailure()
  {
    /* NOTICE */
    /* There is no possibility for failure, so always return false */
    return false;
  }

  protected override bool isSuccess()
  {
    return m_playerReached;
  }

  protected void processSuccess()
  {
    m_info = "Point Reached!";
    GameMaster.gm.NotifySuccess(this);
  }

```

```C#
```

```
using UnityEngine;

public class GameMaster : MonoBehaviour
{
  public static GameMaster gm = null;

  void Awake()
  {
    if (gm == null)
      gm = this;
  }

  public void NotifySuccess(GameOverCondition cond)
  {
    Debug.Log(cond.GetProgressInfo());
  }

  public void NotifyFailure(GameOverCondition cond)
  {
    Debug.Log(cond.GetProgressInfo());
  }

  private void Update()
  {
    /* FIXME for testing purpose only! */
    if (Input.GetKeyDown(KeyCode.Escape))
      Application.Quit();
  }
}
```

Miejsca warte szczególnej uwagi:

```C#
    AddActionOnSuccess(processSuccess);
```

Dodanie metody, która uruchomi się po wykryciu wygranej -> włączenie możliwości wygranej (nasz Warunek nie jest w stanie sprawić, że gracz przegra).

```C#
  private void OnTriggerEnter2D(Collider2D collision)
  {
    if (!m_playerReached && collision.tag == "Player")
    {
      m_playerReached = true;
      CheckConditions(); // force to update conditions
    }
  }
```

Powyższy listing dodatkowo ilustruje, że warunki nie są wykrywane automatycznie (*CheckConditions*) – domyślnie trzeba sprawdzić je manualnie. Jest to spowodowane optymalizacją, jeżeli zależy nam na automatycznym wykrywaniu to wystarczy wywołać metodę *CheckConditions()* w metodzie *Update* (dla Unity).

Reszta kodu mówi sama za siebie, dlatego zachęcam do samodzielnego przejrzenia kodu :)

Podsumowanie

Dzisiejsza lekcja przedstawiała mechanizm zezwalający na sterowanie zasadami gry. Zachęcam Was do samodzielnych eksperymentów z tą klasą, wspólnie pobawimy się nią (i innymi klasami) w części zatytułowanej „Diabeł tkwi w szczegółach”.

Tymczasem zapraszam Was do systemu komentarzy, gdzie możecie podzielić się ze mną swoją opinią (a także zapraszam Was do innych materiałów dostępnych na blogu).

Do przeczytania w kolejnym wpisie,

Code ON!