Piszemy RPGo-Platformówkę (1) – Nauka chodzenia (sheadovas/poradniki/proj_platf_rpg/1-nauka-chodzenia/)

Mar 11, 2017 / proj_platf_rpg (sheadovas/category/poradniki/proj_platf_rpg/)

W dzisiejszej części zajmiemy się jedną z bardziej kluczowych kwestii w grach platformowych, a mianowicie chodzeniem :)

Hej, już na wstępie chciałbym zaznaczyć, że projekt już leży na repo, omawiany kod w tym wpisie dotyczy wersji z commit'a [ba6e8d7 – Nauka chodzenia (https://github.com/sheadovas/proj_platf_rpg/commit/ba6e8d714fa97bea376f9e7f19900b8295a32ea6)]. W dalszej części wpisu zajmiemy się aspektami teoretycznymi wraz z ich implementacją w praktyce. Zapraszam do lektury.

Krok, po kroku... (babysteps)

Zacznijmy od tematu dzisiejszego wpisu, a więc chodzenia, czy też zmiany położenia. Mi osobiście ciężko jest sobie wyobrazić gry platformowej bez możliwości poruszania, jednocześnie z tą definicją kojarzy mi się przynajmniej kilka możliwych sposobów poruszania: przy użyciu myszy, klawiatury, pada, teraz dochodzi nam jeszcze VR.

Implementacja każdego z wymienionych systemów zupełnie osobno wydaje się zupełnie bez sensu i niepotrzebnym powtórzeniem tej samej pracy: po wciśnięciu przycisku „strzałki w lewo" na klawiaturze i przemieszczenie gałki na padzie „w lewo" powinno spowodować, że postać gracza poruszy się w identyczny* sposób w lewo, tzn większość mechaniki poruszania jest identyczna dla obu przycisków, zmienia się jedynie urządzenie z którego korzystamy.

> * sterowanie może się różnić tym, że na klawiaturze nie jesteśmy w stanie kontrolować stopnia nacisku, tzn albo klawisz jest wciśnięty albo nie; na padzie możemy zbadać stopień nachylenia gałki, a więc mocno odchylona gałka może wpływać na np prędkość poruszania w lewo

Jesteśmy w stanie też zauważyć pewne podobieństwo pomiędzy graczem i wrogiem: obaj mają pewny zestaw umiejętności, uzbrojenia, różny poziom życia, siły, itd. Tak na dobrą sprawę różnią się tylko jednym elementem: wróg jest sterowany przez komputer, a więc przez jakąś „sztuczną inteligencję", tzw AI.

Mając na uwadze powyższy fakt, to możemy stwierdzić, że Gracz (Player) i Wróg (Enemy), to ogólniej Postać (Character), która może być sterowana przez różne urządzenia: Klawiaturę, Pada, Sztuczną Inteligencję (komputer).

Oczywiście Gracz będzie zawierał cechy, których nie będzie posiadał Wróg i *vice versa*, ale na tym poziomie abstrakcji, który nas dzisiaj interesuje (poruszanie) dalsze różnice są nieistotne i pomijalne.

Nauka chodzenia

Skoro mamy już za sobą kilka potencjalnie przydatnych faktów, to warto się zastanowić jak można ujednolicić całość do interfejsu(!) wspólnego dla różnych „urządzeń".

> W tej części darujemy sobie implementację Pada oraz AI. Dzisiaj jedynie uczymy się chodzić, ale pozostawiamy sobie „furtkę" na prostą implementację innych systemów w niedalekiej przyszłości.

Z racji, że piszemy grę platformową (a później RPG), to nasza postać musi:

- poruszać się w lewo i prawo;
- skakać.

Może jeszcze oczywiście wykonywać inne rzeczy (wślizgi, podwójne skoki), ale to jest na wpis dotyczący specjalnych umiejętności.

Interfejs ujednolicający urządzenia, powinien udostępniać informacje o:

- żądaniu poruszania w poziomie: czy postać powinna poruszyć się w lewo/prawo (bo np. została kliknięta strzałka w lewo);
- żądaniu z informacją o tym, czy postać powinna skoczyć (bo np. została kliknięta spacja);
- swój typ, bo fajnie mieć dostęp z poziomu kodu do informacji o tym przez co dana postać jest poruszana (np. „Klawiatura”);

Oprócz tego całość powinna być spięta funkcją służącą do obsługi danego urządzenia. Z poziomu kodu może wyglądać to np. tak:

C#

```
public interface ICharacterController
{
  string controllerType { get; }

  float moveDirection { get; }
  bool isJumpClicked { get; }

  void Control();
}
```

Jeżeli chodzi o implementację tego dla Unity i klawiatury, to sprowadza się to do kodu postaci:

C#

```
using UnityEngine;
public class ManualKeyboardController : MonoBehaviour, ICharacterController
{
  const float MOV_LEFT = -1.0f;
  const float MOV_RIGHT = 1.0f;
  const float MOV_STOP = 0.0f;

  public KeyCode keyMoveLeft = KeyCode.LeftArrow;
  public KeyCode keyMoveRight = KeyCode.RightArrow;
  public KeyCode keyJump = KeyCode.Space;

  float m_movdir = 0.0f;
  bool m_jump = false;


  public string controllerType
  {
    get
    {
      return "MANUAL_KEYBOARD";
    }
  }

  public float moveDirection
  {
    get
    {
      return m_movdir;
    }
  }

  public bool isJumpClicked
  {
```

Aby nie zagłębiać się w szczegóły tej dość prostej implementacji, to zwrócę jedynie uwagę na kilka ciekawszych fragmentów:

C#

```
34  public bool isJumpClicked
35  {
36    get
37    {
38      if(m_jump)
39      {
40        m_jump = false;
41        return true;
42      }
43
44      return false;
45    }
46  }
```

Ten parametr sprawia, że w przypadku gdy wykryto kliknięcie przycisku skoku, to flaga informująca o tym jest zerowana, tak aby nie trzymała starej wartości z informacją o wciśniętym klawiszu (nie ma możliwości anulowania skoku, więc jest to jedyny sposób jej wyczyszczenia).

C#

```
48  public void Control()
49  {
50    if(Input.GetKeyDown(keyMoveLeft))
51    {
52      m_movdir = MOV_LEFT;
53    }
54    else if(Input.GetKeyDown(keyMoveRight))
55    {
56      m_movdir = MOV_RIGHT;
57    }
58
59    if ((m_movdir == MOV_LEFT && Input.GetKeyUp(keyMoveLeft)) ||
60       (m_movdir == MOV_RIGHT && Input.GetKeyUp(keyMoveRight)))
61    {
62      m_movdir = MOV_STOP;
63    }
64
65    if(Input.GetKeyDown(keyJump))
66    {
67      m_jump = true;
68    }
69  }
```

W tej funkcji przechwytujemy wciśnięcie klawiszy i ustawiamy wartości odpowiednim parametrom gdy wykryto wciśnięcie.

Warte uwagi jest to, że powyższa implementacja (po lekkich modyfikacjach) jest także implementacją pada, różniłaby się jedynie wartościami *m_movdir*, wtedy przyjmowałaby liczby z zakresu [-1.0,1.0] (decydowałyby one o stopniu odchylenia gałki na padzie), a nie jak obecnie {-1, 0, 1}.

Stawiamy pierwsze kroki...

W tym momencie mógłbym zakończyć swój wpis, gdybym nie chciał pokazać że to co napisaliśmy ma sens. Teraz zajmiemy się graczem (Postacią) i podobnie jak w poprzednim przypadku zastanowimy się jak ogólnie może wyglądać Postać (PlayableCharacter).

To co każda postać w naszej grze będzie posiadała, to:

- swój typ (Gracz, Wróg, NPC);
- sposób sterowania (AI, klawiaturę);

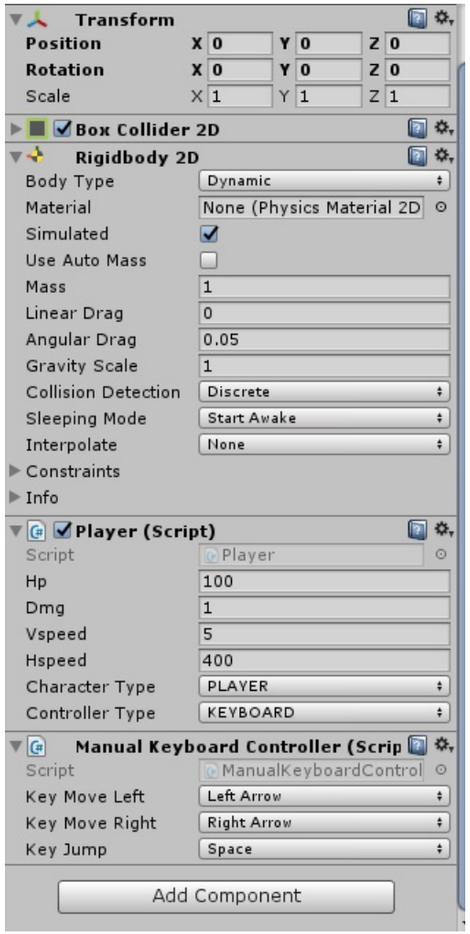
- obecnie nieistotne: życie, siłę ataku, masę;
- modyfikowalną (stałą) prędkość poruszania;
- siłę (wysokość) wyskoku.

Kod w Unity (w całości) wygląda jak poniżej, później przyjrzymy się jego ciekawszym fragmentom:

```C#
using UnityEngine;

public class PlayableCharacter : MonoBehaviour
{
  public enum CharacterType
  {
    NONE, PLAYER, ENEMY, OTHER
  };

  public enum ControllerType
  {
    KEYBOARD, AI
  };

  [SerializeField]
  protected int m_hp = 100;

  [SerializeField]
  protected float m_dmg = 1;

  [SerializeField]
  protected float m_vspeed = 5;

  [SerializeField]
  protected float m_hspeed = 400;

  [SerializeField]
  protected CharacterType m_characterType = CharacterType.NONE;

  [SerializeField]
  protected ControllerType m_controllerType = ControllerType.KEYBOARD;

  protected ICharacterController m_controller;
```

Z różnych względów chcemy chcemy mieć dostęp do informacji o używanych przez postać urządzeniach, oraz chcemy znać jej typ:

```C++
  public enum CharacterType
  {
    NONE, PLAYER, ENEMY, OTHER
  };

  public enum ControllerType
  {
    KEYBOARD, AI
  };
```



(https://i1.wp.com/www.shead.ayz.pl/wp-content/uploads/2017/03/babysteps_1.png)

Zabraniamy jednocześnie używania klasy *PlayableCharacter* do tworzenia prawdziwych obiektów (na tym etapie da się to obejść), stąd typ postaci *NONE*.

C#

```
139  protected virtual void FixedUpdate()
140  {
141    m_controller.Controll();
142
143    Move();
144    Jump();
145  }
```

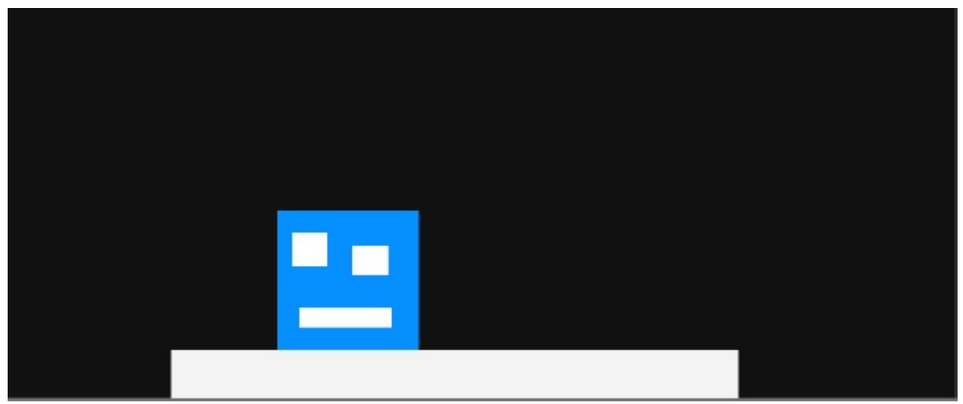
Prosta obsługa zaimplementowanych systemów, najpierw aktualizujemy informacje o wciśniętych klawiszach, a później wykonujemy odpowiednie akcje. W późniejszych pracach nadpiszemy sobie tą klasę dla nieco innego sterowania dla gracza, wroga, itp. ale w tym momencie nam to wystarczy.

C#

```
159   protected ICharacterController probe_character_controller()
160   {
161     ICharacterController conn = null;
162     switch (m_controllerType)
163     {
164       case ControllerType.KEYBOARD:
165         conn = GetComponent<ManualKeyboardController>();
166         break;
167
168       default:
169         Debug.LogError("Invalid controller type!", this);
170         break;
171     }
172
173     if(conn == null)
174     {
175       Debug.LogError("Cannot find proper controller attached to object!", this);
176     }
177
178     return conn;
179   }
```

Ta metoda służy do znalezienia kontrolera wybranego z poziomu edytora Unity i podłączenia go do skryptu.

Jeżeli w tym momencie uruchomimy grę w Unity (tzn ustawimy wszystko tak żeby działało), to powinno nam się pojawić coś takiego:



(https://i0.wp.com/www.shead.ayz.pl/wp-content/uploads/2017/03/babysteps_2.png)

Serdecznie zapraszam do pobrania gotowego projektu z [repozytorium (https://github.com/sheadovas/proj_platf_rpg)] i do zabawy z nim. Zapraszam także do wersji [binarnej (https://github.com/sheadovas/proj_platf_rpg/releases/download/1.1/windows.7z)] (póki co tylko *Windows*), chociaż tutaj można jedynie poruszać się na boki i skakać.

Zakończenie

Jeżeli chodzi o „naukę chodzenia” to tyle co chciałem Wam przekazać w tym wpisie, w dalszych częściach zajmiemy się nieco większymi wyzwaniami (bardziej ambitnymi).

Oprócz tego chciałbym Was zachęcić do podzielenia się swoim komentarzem, jeżeli macie jakieś pytania i uwagi to zapraszam do ich dodawania. A jeżeli lubicie być na bieżąco to zapraszam Was na mojego Facebooka i Twittera (linki w menu po prawej stronie).

Do przeczytania w kolejnym wpisie,

Code ON!