Piszemy RPGo-Platformówkę (5) – Walka (sheadovas/poradniki/proj_platf_rpg/5-walka/)

Maj 01, 2017 / proj_platf_rpg (sheadovas/category/poradniki/proj_platf_rpg/)

Wprowadzenie uzbrojenia

W dzisiejszym (nieco opóźnionym) wpisie (a raczej uzupełnieniu poprzedniego) zajmiemy się omówieniem pierwszym rodzajem wyposażenia, nieodzownie połączonym z walką, a więc bronią.

Tradycyjnie dla tej serii, omawiany kod dotyczy zmian do commit'a [2f1f893 (https://github.com/sheadovas/proj_platf_rpg/commit/2f1f8936096d09f9323e856b2414072dbc639e19)], a wersja demo jest już dostępna do [pobrania (https://github.com/sheadovas/proj_platf_rpg/releases/tag/1.5)].

Walka (część 2)

Pierwszym elementem od jakiego zaczniemy jest wprowadzenie „pojęcia" śmierci do naszej gry, tzn chcemy aby każdy obiekt posiadający statystyki po spełnieniu pewnych warunków był uznawany jako „martwy", a więc niemogący wchodzić w interakcję z resztą otoczenia (nie może otrzymywać obrażeń, poruszać się).

CharacterStats.cs C#

```
102   public bool isDead
103   {
104     get { return m_isDead; }
105   }
106
107   ...
108
109   protected void GetHit(CharacterStats attackerStats)
110   {
111     if (!hitable ||             // if cannot hurt object...
112         m_invulnerable ||       // if got hit...
113         isDead ||               // if is dead...
114         attackerStats.dmg == 0  // if base dmg is 0...
115       )
116       return;                   // ...then attacking is disabled, skip it!
117
118     m_hp = Mathf.Max(m_hp - attackerStats.dmg, 0.0f);
119
120     if (m_hp > 0)
121     {
122       StartCoroutine(StayInvulnOnHit()); // temporary disable hitting player
123     }
124     else
125     {
126       m_isDead = true;
127     }
128     ...
129   }
```

W teorii sprawdzenie stanu postaci mogłoby się ograniczać do przyrównania ilości życia z zerem (do tego też się sprowadza podstawowa implementacja tej funkcjonalności), ale być może będziemy później chcieli wprowadzić specjalne postaci, które mimo braku życia będą mogły wchodzić w jeszcze jakieś interakcje.

Kolejnym wartym uwagi fragmentem kodu jest zauważenie, że dany obiekt nie zawsze w danej chwili zadaje obrażenia co zakładała nasza poprzednia implementacja. Bowiem może dojść do sytuacji, gdy zajdzie kolizja pomiędzy np. bronią a postacią, ale broń będzie w stanie cooldown'u, a więc nie będzie zadawała obrażeń.

CharacterStats.cs C#

```
  protected void ReceiveDamage(CharacterStats enemy)
  {
    if (enemy == null)
      return;

    if (enemy.canAtttack) // receive damage only if enemy can attack
      GetHit(enemy);  // example skill is ready, character is not invuln.
  }

  virtual protected bool CanAttack()
  {
    // default behaviour:
    // enable attack only if player is not invulnerable
    return !m_invulnerable && !isDead;
  }
```

Idąc dalej chcemy wprowadzić atakowanie nie przy użyciu „aury” postaci, a poprzez wyciągnięcie broni na wciśnięcie przycisku, w tym celu dodajemy jego obsługę:

ICharacterController.cs C#

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

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

  void Control();
}
```

ManualKeyboardController.cs C#

```
using UnityEngine;

public class ManualKeyboardController : MonoBehaviour, ICharacterController
{
  ...
  public KeyCode keyAttack = KeyCode.LeftControl;
  ...
  protected bool m_isAttack = false;
  ...
  public bool isAttackClicked
  {
    get { return m_isAttack; }
  }

  public void Control()
  {
    ...
    /* Running */
    m_isRunning = Input.GetKey(keyRun);

    /* Attacking */
    m_isAttack = Input.GetKey(keyAttack);
  }
}
```

Aktywacja bronic odbywa się następująco:

PlayableCharacter.cs C#

```
198   protected virtual void Attack()
199   {
200     if (!m_controller.isAttackClicked)
201       return;
202
203     m_equippedWeapon.UseWeapon();
204   }
```

Tutaj pojawił nam się nowy obiekt klasy *Weapon*, który w powyższym listingu „uruchamia” naszą broń po wciśnięciu przycisku atak. Warto zauważyć, że do momentu gdy broń nie zostanie użyta to nie jest w stanie zadać obrażeń.

Weapon.cs C#

```
1  using UnityEngine;
2  using System.Collections;
3
4  public class Weapon : CharacterStats
5  {
6    public bool isInfinite
7    {
8      get { return m_isInfinite; }
9    }
10
11   public int ammunition
12   {
13     get { return m_ammunition; }
14   }
15
16   /* Default setup is set to short distance weapon (i.e. sword) */
17   [Header("Weapon")]
18   [SerializeField]
19   protected bool m_isInfinite = true;
20
21   [SerializeField]
22   protected int m_ammunition = 1;
23
24   [SerializeField]
25   protected float m_cooldown = 1.0f;
26
27   [SerializeField]
28   protected Animator m_animator;
29
30   protected bool m_isCooldown = false;
31
32
33   protected void Awake()
```

Tym samym zbliżyliśmy się do klasy *Weapon*, powyższa klasa umożliwia na stworzenie broni zarówno białej jak i palnej, mechanika aktywacji / dezaktywacji samej broni jest zaszyta także w samych Animacjach, które odpowiednio wyłączają i włączają kolizje z bronią.

Jak widzimy (zgodnie z moimi zapowiedziami) broń jest rozszerzeniem klasy *CharacterStats*, wprowadza ideę cooldownu oraz ilości dostępnej amunicji (po skończeniu której staje się bezużyteczna). Obie idee są proste i nie odbiegają zbytnio od poprzednich lekcji, a zatem pozwolę pominąć sobie ich dokładne objaśnienie.

Podsumowanie

To tyle co dla Was przygotowałem w tym wpisie, który jest *de facto* uzupełnieniem poprzedniego. W kolejnym pojawi się znacznie więcej ciekawych (i nowych) rzeczy, bo zabawimy się w stworzenie prostego AI.

Tradycyjnie zapraszam do zagrania w demo, dzielenia się komentarzami oraz śledzeniem bloga na social media (fb, twitter, g+).

Code ON!