Piszemy RPGo-Platformówkę (6) – Dajcie mi przeciwnika! (sheadovas/poradniki/proj_platf_rpg/6-dajcie-mi-przeciwnika/)

Maj 14, 2017 / proj_platf_rpg (sheadovas/category/poradniki/proj_platf_rpg/)

Wprowadzamy przeciwników sterowanych przez komputer

Hej, dzisiaj porozmawiamy sobie o implementacji przeciwnika do gry.

Przeciwnicy – poetycznie (teoria)

Idea

Przeciwnicy sterowani przez komputer są nieodzowną częścią większości gier akcji. Mogą służyć na wiele sposobów: od wypełniacza rozgrywki, gdzie ich jedynym celem jest łatwa śmierć (np. „miniony” w grach typu MOBA) – aż do bardziej zaawansowanej, unikatowej i zmiennej w zachowania, co czyni przeciwników trudniejszymi do pokonania (np. sztuczna inteligencja tzw. „bossów” kończących poziom).

(https://i1.wp.com/szymonsiarkiewicz.pl/wp-content/uploads/2017/05/minions_lol.jpg)

Miniony w grze „League of Legends”

Warto też zauważyć, że to nie jest „prawdziwa” sztuczna inteligencja, gdy mówimy o AI w grach to zazwyczaj mamy na myśli zestaw predefiniowanych zachowań, które się wyzwalają po zajściu odpowiednich warunków (np. jeżeli gracz znajdzie się w odpowiedniej odległości od przeciwnika, to ten atakuje go szarżą).

Obecnie nie wplata się uczenia maszynowego do gier (ML w implementacji jest dużo trudniejszy i bardziej wymagający niż oskryptowanie paru zachowań, które można napisać dość szybko i stosunkowo łatwo). Także i w tym wpisie pod „AI” będziemy rozumieli „zbiór oskryptowanych zachowań”.

Jak już wspomniałem powyżej, sztuczna inteligencja to zbiór pewnych zachowań, których wykonanie zależy od spełnienia warunków wstępnych. Pseudokod dla sztucznej inteligencji w grze możemy przedstawić następująco:

Ogólny schemat AI

```
AI {
  Behaviour[]
}

Behaviour {
  if condition_set is met, then
    action_do_something()
}
```

Oczywiście nikt nie mówi, że pod jeden zestaw warunków nie można podłączyć wiele akcji, które można np ze sobą łączyć, przeplatać (spełnienie jednego zestawu warunków powoduje łańcuchowe spełnienie kolejnych => wykonywanie kolejnych zestawów akcji), bądź losować jedną reakcję z wielu dostępnych (w celu urozmaicenia rozgrywki – przeciwnik staje się bardziej nieprzewidywalny).

Dla przykładu: jeżeli przeciwnik jest ostrzeliwany ogniem (warunek), to może:

1. uciekać zygzakiem aż znikanie z pola widzenia (prosta akcja)
2. zacząć szukać schronienia (akcja) i rzuci z niego granat w kierunku gracza aby uciekł (łańcuchowa akcja)

Wybranie opcji (1) lub (2) można rozstrzygnąć przy pomocy losowości, która może zostać okraszona dodatkowymi warunkami wstępnymi (np. prawdopodobieństwo wybrania danej opcji jest zależne od swojego stanu życia, odległości od gracza, itp.).

Idąc dalej zastanówmy się jak mogą wyglądać same akcje:

Ogólny schemat Akcji

```
Action {
  walk()
  run()
  fight()
  ...
}
```

Jak widzimy powyżej, powinny być dość proste, tak aby można było je łączyć w bardziej skomplikowane, np *fight_long_distance()* składałaby się mniejszych akcji, które polegałyby na odpowiednim podejściu do gracza, wycelowaniu w niego, strzelaniu (i ucieczce lub podjęciu walki na bliski dystans gdyby on podszedł).

Wraz z przeciwnikami bardzo często idzie w parze poziom trudności gry – im mniejszy, tym przeciwnicy powinni być głupsi (oczywiście dochodzą też takie parametry jak np. ilość życia, ale nimi nie będziemy się teraz zajmować). Jest to świetny przykład na podstawie którego możemy zastanowić się nad dodatkowymi parametrami charakteryzującymi sztuczną inteligencję.

W pisaniu sztucznej inteligencji część rzeczy trzeba emulować, jak np. czas reakcji (*reaction_time*), bo np. „czysty” przeciwnik komputerowy (bez opóźnionego czasu reakcji) jest w stanie od razu zareagować na zmianę warunków i dostosować swoje zachowanie, a nie zawsze jest to pożądane. Dla przykładu: nie chcemy aby przeciwnik wystrzelił w stronę gracza ze snajperki w tej samej chwili co ten wejdzie w jego zasięg.

Emulowanie czasu reakcji ma też dodatkowy efekt uboczny: eliminuje po części sytuacje gdy postać zachowuje się kuriozalnie, bo co chwila zmieniają się warunku uruchamiające skrajne reakcje (np. podchodzimy do wroga, ten do nas podbiega na pewną odległość, strzela, ale stwierdza że jest jednak za blisko więc znowu odbiega, odchodzimy kawałek, stwierdza że musi do nas przybiec, ...).

Innym parametrem może być też sam poziom inteligencji – im wyższy tym szybszy czas reakcji, inna pula decyzji możliwych do podjęcia (mniej „głupie”, wystawiające na niebezpieczeństwa zachowania).

Praktyczniej...

A jak to wszystko się ma do tego co do tej pory napisaliśmy? Przypomnijmy sobie interfejs kontrolera:

ICharacterControllers.cs C#

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

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

  void Control();
}
```

Czyż nie przypomina on schematów powyżej? Jeżeli nie to rzućmy okiem na niego raz jeszcze, tym razem z komentarzami:

ICharacterControllers.cs C#

```
public interface ICharacterController  // AI
{                                      // {
  string controllerType { get; }       //    type of ai (reaction_type, intelligence_level)
                                       //    Behaviours = [
  float moveDirection { get; }         //      action_walk()
  bool isJumpClicked { get; }          //      action_jump()
  bool isRunningKeyClicked { get; }    //      action_run()
  bool isAttackClicked { get; }        //      action_attack()
                                       //    ]
  void Control();                      //    condition checker
}                                      // }
```

Jak widzimy ogólny schemat został zachowany, doszła nam choćby metoda *Control()* sprawdzająca zajście odpowiednich warunków i kolejkująca odpowiednie akcje.

Jeżeli dobrze pamiętacie, to ten sam interfejs pozwalał nam zaimplementować poruszanie gracza przy użyciu np. klawiatury – tam też zastosowaliśmy ten sam schemat, tylko warunki wstępne były niezależne od tego co się dzieje w logice gry, a z tym co chce zrobić gracz.

Świetną bazą do napisania sztucznej inteligencji jest oczywiście klasa *PlayableCharacter* w której musimy jedynie podmienić część domyślnych zachowań, a reszta zacznie działać „sama”! :)

Dajcie mi przeciwnika (praktyka)

Implementacja AI została dodana pod commitem [a7c1d78 (https://github.com/sheadovas/proj_platf_rpg/commit/a7c1d7883026fe7d7dbb786339ce7fa145bec745)]. Implementacja poszczególnych zachowań AI ogranicza się do 3 kroków:

1. Dziedziczenia po *ICharacterController* wraz ze stworzeniem odpowiednich zachowań.
2. Dodania nowego typu kontrolera w *PlayableCharacter.*
3. Edycji przeciwnika z tworzonego menu, wybrania odpowiedniego typu obsługiwanego kontrolera.

Dla przykładu zróbmy sobie przeciwnika, który jedynie co robi to „chodzi w kółko” niczym Goomba w Mario:

AI_Walker.cs C#

```
using UnityEngine;

public class AI_Walker : MonoBehaviour, ICharacterController
{
  public string controllerType
  {
    get
    {
      return "AI_WALKER";
    }
  }

  public bool isAttackClicked
  {
    get
    {
      return false;
    }
  }

  public bool isJumpClicked
  {
    get
    {
      return false;
    }
  }

  public bool isRunningKeyClicked
  {
    get
    {
      return false;
```

Jedyne zmiany względem podstawowej wersji znajdują się od linii 47, a służą jedynie do wydania polecenia: „zmień kierunek ruchu co 2 sekundy”.

Nieco ciekawiej jest w sytuacji gdy chcemy zaimplementować berserkera, a więc postać która gdy wykryje w swoim zasięgu gracza to wykonuje szarżę:

AI_Berserker.cs C#

```
using UnityEngine;

public class AI_Berserker : MonoBehaviour, ICharacterController
{
  public string controllerType
  {
    get
    {
      return "AI_BERSERKER";
    }
  }

  public bool isAttackClicked
  {
    get
    {
      return m_playerInView;
    }
  }

  public bool isJumpClicked
  {
    get
    {
      return false;
    }
  }

  public bool isRunningKeyClicked
  {
    get
    {
      return m_playerInView;
```

Do wykrywania czy gracz jest w zasięgu wroga użyłem *BoxCollider'ów* po „jednej stronie" postaci, dzięki czemu wykrywamy gracza tylko wtedy gdy może „zobaczyć" gracza. W powyższym listingu zaimplementowałem także nieco inne liczenie czasu potrzebnego do zmiany kierunku ruchu postaci.

Oczywiście dodaliśmy także nowe wpisy w *PlayableCharacter* aby wszystko mogło działać:

PlayableCharacter.cs C#

```
  public enum ControllerType
  {
    KEYBOARD, AI_WALKER, AI_BERSERKER
  };
```

PlayableCharacter.cs C#

```
219   protected ICharacterController ProbeCharacterController()
220   {
221     ICharacterController conn = null;
222     switch (m_controllerType)
223     {
224       case ControllerType.KEYBOARD:
225         conn = GetComponent<ManualKeyboardController>();
226         break;
227
228       case ControllerType.AI_WALKER:
229         conn = GetComponent<AI_Walker>();
230         break;
231
232       case ControllerType.AI_BERSERKER:
233         conn = GetComponent<AI_Berserker>();
234         break;
235
236       default:
237         Debug.LogError("Invalid controller type!", this);
238         break;
239     }
```

Myślę, że powyższe listingi są dość proste i zrozumiałe, na koniec części praktycznej jeszcze mała poprawka sprawiająca że tekstura postaci jest skierowana zgodnie z jej kierunkiem ruchu:

PlayableCharacter.cs

```
185   protected virtual void Flip()
186   {
187     Vector3 scale = gameObject.transform.localScale;
188     if (m_rigidbody.velocity.x < 0.0f)
189     {
190       scale.x = 1.0f;
191     }
192     else
193     {
194       scale.x = -1.0f;
195     }
196     gameObject.transform.localScale = scale;
197   }
```

Podsumowanie

Po tej części mamy już spory kawał gry i większą część za sobą. Przydałoby się jeszcze wyświetlać najważniejsze informacje w ładny sposób – tym (UI) zajmiemy się w kolejnej części ;)

Liczę, że wpis się Wam podobał, tradycyjnie zapraszam do zagrania w demo, podzielenia się komentarzem oraz śledzeniem bloga na mediach społecznościowych (jeżeli jeszcze tego nie robicie).

Code ON!