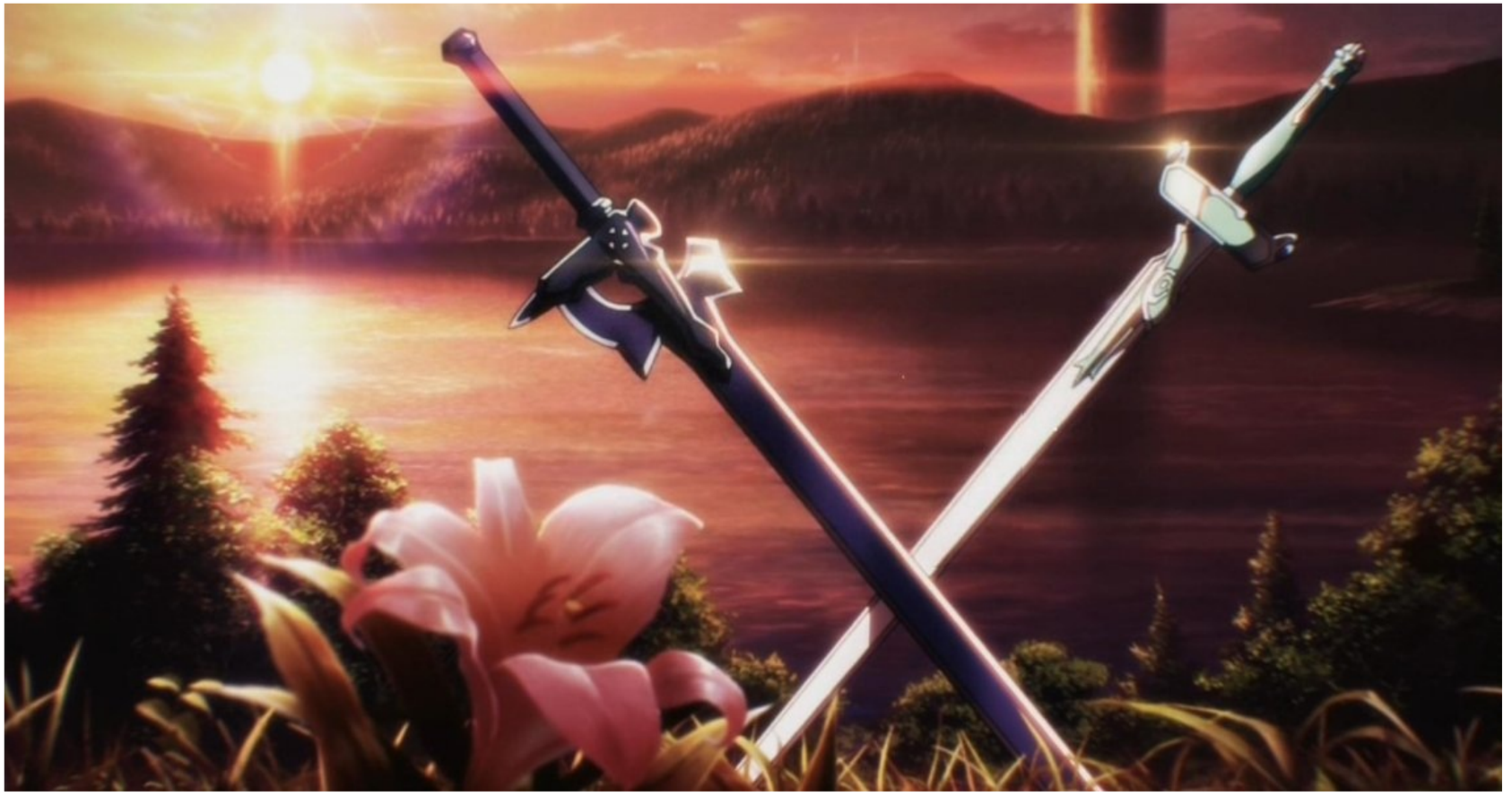


Piszemy RPGo-Platformówkę (10) – Masz mój miecz! (sheadovas/poradniki/proj_platf_rpg/10-masz-moj-miecz/)

Lip 04, 2017 / proj_platf_rpg (sheadovas/category/poradniki/proj_platf_rpg/)

Część pierwsza prac nad ekwipunkiem

Hej, dzisiaj zajmiemy się wprowadzeniem pomysły ekwipunku (a raczej jego zaimplementowaniem), właściwe jego wdrożenie do gry ukrywa się w kolejnej części *(....)* i *moją tarczę!*

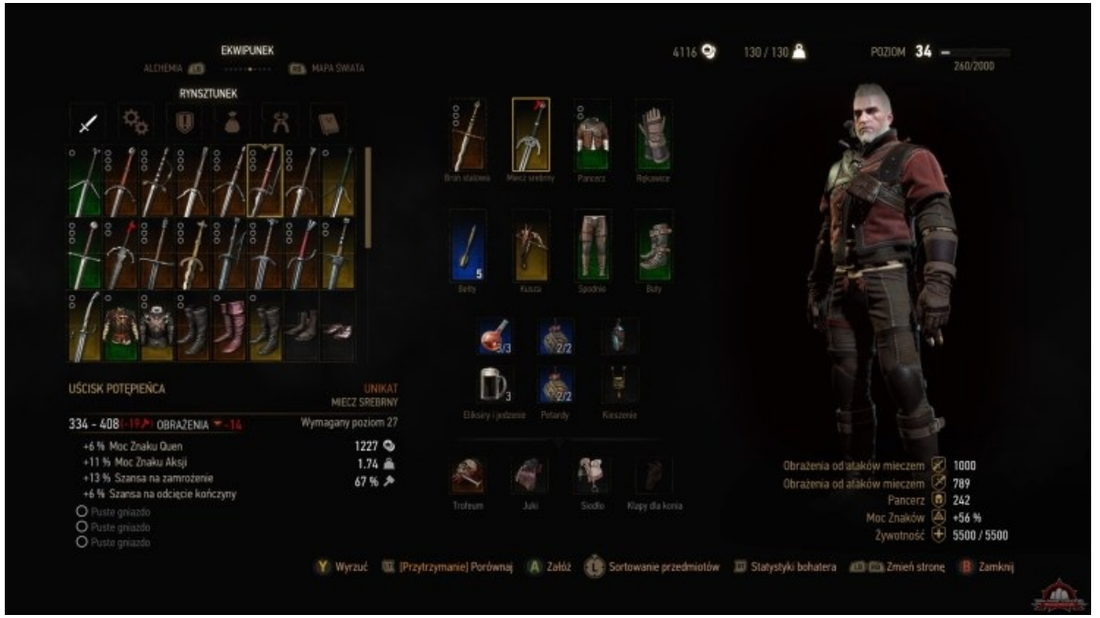
Omawiany kod dotyczy zmian do wizualizowanych [tutaj (https://github.com/sheadovas/proj_platf_rpg/compare/f1980f4f321f6fa501bc12d134bd48645bbf9bbc...8adb363b8bb472fed4be224e33e9a3a740c8f522)], a demo można pobrać [stąd (https://github.com/sheadovas/proj_platf_rpg/releases/tag/1.7)].

Planujemy

Przed rozpoczęciem właściwej pracy należy zastanowić się nad tym jak sam ekwipunek powinien wyglądać i odpowiedzieć sobie na pytanie: czym właściwie jest ekwipunek?

Po chwili zastanowienia jesteśmy w stanie stwierdzić, że jest to zbiór (lista) rzeczy, które posiada gracz. Mogą składać się na nie np. pieniądze oraz przedmioty różnego rodzaju, np: broń, pancerz, jedzenie, mikstury, itd.

Sam ekwipunek, czy raczej plecak może mieć nałożone ograniczenie w postaci maksymalnej masy jaką możemy udźwignąć, co za tym idzie narzucamy na przedmioty wymaganie dotyczące ich uogólnienia do pojedynczej, abstrakcyjnej klasy posiadającej pewne wspólne cechy.



(https://i1.wp.com/szymonsiarkiewicz.pl/wp-content/uploads/2017/07/ekwix.jpg)
Ekwipunek (Wiedźmin 3)

Zacznijmy od „góry” i zacznijmy zastanawiać się jak będzie wyglądał ekwipunek, aby „zejść” do pojedynczego przedmiotu.

Ekwipunek

Jak wspomniałem wcześniej – ekwipunek to zbiór przedmiotów. Mogą być przypisane do np. gracza, ale mogą tworzyć także osobny przedmiot np. plecak. W ogólnej postaci widzimy go jako listę przedmiotów wraz z maksymalną pojemnością (wg wagi) oraz osobnym slotem na zebrane złoto.

Dalszymi szczegółami zajmiemy się przy części z kodem.

Przedmiot

Sam przedmiot jest o wiele ciekawszy, co warto przy nim zauważyć to posiada:

- własną nazwę i opis,
- swój zestaw właściwości (np. używalny, jadalny, ekwipowalny jako broń itp) i mogą się one ze sobą łączyć,
- jakość (unikalność), np: „zepsuty”, „zwykły”, „epicki” – wpływa ona na bonus (także negatywny) związany ze sposobem użycia przedmiotu,
- wartość (oczywiste),
- wagę (oczywiste),
- sposób użycia – nieważne czy ogranicza się to do wyekwipowania przedmiotu, czy wypicia mikstury leczniczej.

Oprócz tego przedmiotu mogą posiadać swoje własne własności i powinny posiadać ogólny interfejs umożliwiający ich użycie z klasy podstawowej.

Dokładniej przedmiotom przyjrzymy się już za chwilę.

Daj mi miecz!

Mając na uwadze powyższe spostrzeżenia zacznijmy dokładniej przyglądać się nowym feature'om (wraz z pisaniem kodu), tym razem zaczniemy od Przedmiotów.

Klasa Item

Przedmiot jest implementowany przez abstrakcyjną klasę *Item* łączącą wspólne funkcjonalności każdego przedmiotu, prezentuje się ona następująco:

Item C#

```
using UnityEngine;

abstract public class Item : MonoBehaviour
{
  const int MAX_STACK_SIZE = 16;

  public enum ItemProperty
  {
    NONE      = 0,

    EATABLE   = (1 << 1),
    EQUIPABLE = (1 << 2),
    STACKABLE = (1 << 3),

    ARMOUR    = (1 << 4),
    WEAPON    = (1 << 5),

    DISABLED  = (1 << 6) // "broken"
  };

  public enum ItemQuality
  {
    BROKEN = 0,
    NORMAL = 1,
    SUPER  = 2
  };

  public float prize = 10; // per unit

  [HideInInspector]
  public int eid = -1; // equipment id

  public ItemQuality quality
```

W powyższym listingu celowo pominąłem implementacje metod, którymi zajmiemy się za chwilę, gorąco zachęcam się do samodzielnego zapoznania się z powyższym listingiem, szczególną uwagę należy zachować przy komentarzach które wyjaśniają parę pomysłów (i tak wrócimy do nich za chwilę, ale lepiej zrozumieć pewne idee samemu).

Jeżeli przeczytałeś powyższy listing jak prosiłem, to widzisz że idee są kalką z tego co opisałem wcześniej. Objaśnienia zacznę od metod po kolei, dopiero na końcu objaśnię jeszcze raz ideę blokowania przedmiotów (*useLock*).

Uwaga! Poniższe objaśnienie nie zawiera omówienia każdej metody, analizę tych najprostszych zostawiam jako zadanie domowe ;)

Właściwości przedmiotów

ItemProperty & Item Quality C#

```
public enum ItemProperty
{
  NONE      = 0,

  EATABLE   = (1 << 1),
  EQUIPABLE = (1 << 2),
  STACKABLE = (1 << 3),

  ARMOUR    = (1 << 4),
  WEAPON    = (1 << 5),

  DISABLED  = (1 << 6) // "broken"
};
```

W omawianej implementacji zakładam, że przedmioty mogą posiadać własności, które mogą decydować o ich sposobie użycia, a same własności mogą się łączyć, np. możemy stworzyć przedmiot (pałkę), który możemy założyć (EQUIPABLE) jako broń (WEAPON) albo jako zbroję w miejscu tarczy (ARMOUR).

W tym celu powstał zestaw łączących się opcji, samą ideę czegoś takiego opisywałem [tutaj (sheadovas/poradniki/howto/przekazywanie-opcji-jako-jeden-parametr/)]. W przypadku tej implementacji napisałem API obsługujący ten feature:

C#

```
 public void SetProperties(ItemProperty properties)
 {
   m_properties = properties;
 }

 public bool HasProperty(ItemProperty property)
 {
   if ((m_properties & property) != ItemProperty.NONE)
     return true;

   return false;
 }
```

Jakość przedmiotów (oraz psucie i naprawa)

C#

```
public enum ItemQuality
{
  BROKEN = 0,
  NORMAL = 1,
  SUPER  = 2
};
```

Ten feature odpowiada za mnożenie statystyk danego przedmiotu, każdy stopień jakości to wartość mnożnika bonusu uzyskiwanego z posiadanej jakości, np. jeżeli mikstura życia bazowo odnawia 10HP, ale jest jakości SUPER to w efekcie odnowi 20HP. Z kolei mikstura, która jest zepsuta (BROKEN) nie będzie mogła być użyta i będzie wymagała naprawy.

C#

```C#
  public int GetStatsMultiplier()
  {
    // disable lock
    // important note:
    //   we dont know the context of use GetStats()
    //   & we fooly belive in that the call is made once when needed to get calculated value
    //   see: GetRestorationHP() from ItemFood.cs
    m_useLock = false;

    return (int)m_quality;
  }

  public void Fix()
  {
    disable_property(ItemProperty.DISABLED);
  }
```

Używanie przedmiotów

Use() C#

```C#
  public virtual void Use(ItemProperty useContext)
  {
    if (HasProperty(ItemProperty.DISABLED) || !HasProperty(useContext))
    {
      // item is broken, so cannot be used
      // or if item haven't ctx property
      on_item_use_failure();
      return;
    }

    else if (HasProperty(ItemProperty.STACKABLE) && m_useLock)
    {
      // item is locked, just skip use & belive in user multi-click
      return;
    }

    else
    {
      // more actions should be implemented by user
      on_item_use(useContext);
    }
  }
```

Jak wspomniałem w poprzednim rozdziale metody mogą posiadać wiele funkcjonalności oraz ogólnie mogą być użyte na wiele sposobów. Metoda *Use()* nie stara się implementować ich wszystkich (tym zajmą się klasy dziedziczące po tej klasie), tutaj jedynie weryfikujemy poprawność kontekstu użycia, a więc: jeżeli przedmiot jest zepsuty, albo chcemy użyć jedzenie jako broń (a nie powinniśmy móc) to uniemożliwiamy taką akcję w zarodku.

Blokowanie przedmiotów

```C#
```

```
// use lock - concept
// we need to protect 1-use items against destroy
//
// example scenario:
//  1) we are using potion by Use()
//  2) if we have > 0 items in stack
//    2.1) we can get restoring mana amount by GetMana() method
//  3) else
//    3.1 item is destroyed (because is empty)
//    3.2 we cannot use GetMana() method because Item is destroyed :(
//
// solution:
//   lock item destroying during time between Use() & GetMana(), then check conditions
//
// notice:
//   lock is set in child class, because we don't know correct context of setting lock on item
//   guard in Item does not guarantee, that correct context for item is correct of setting lock
//   so we have to do it manually in each class
protected bool m_useLock = false;
```

Jest to feature obchodzący poniekąd problem z rzeczami, które możemy „stackować" (i mogą się zużywać), a więc np posiadać w jednym slocie wiele przedmiotów tego samego typu.

Na czym polega problem: z racji, że przedmioty nie muszą być używane bezpośrednio przez postać gracza oraz mogą być używane na wiele sposobów (np przez kliknięcie w UI), to nie mogą po prostu zwrócić wartości (np. *Use()* dla mikstury mogłoby zwracać ilość życia do odnowienia). Samo zwrócenie wskaźnika bonusu odbywa się przez użycie metody *GetStatsMultiplier()*.

Problem pojawia się w sytuacji, gdy mamy zużywalne przedmioty i zużyjemy je wszystkie. Wtedy nie możemy zniszczyć samego obiektu przedmiotu od razu, tylko powinniśmy zaczekać aż odpowiedni obiekt pobierze samą wartość (innymi słowy: obiekt nie może zostać zniszczony pomiędzy wywołaniem *Use()* oraz *GetStatsMultiplier()*).

Myślę, że teraz idea „blokowania" przedmiotów powinna być bardziej jasna, w razie czego pytajcie w komentarzach ;)

Klasa Equipment

Ta klasa jest zdecydowanie prostsza od klasy *Item*, składa się na nią zaledwie słownik i interfejs umożliwiający operowanie na ekwipunku.

Equipment C#

```
using UnityEngine;
using System.Collections.Generic;
using UnityEngine.UI;

public class Equipment : MonoBehaviour
{
  public int gold
  {
    get { return m_gold; }
    set { update_gold(value); }
  }

  public float capacity
  {
    get { return m_capacity; }
  }

  public float weight
  {
    get { return m_weight; }
  }

  public Item[] defaultItems; // default items for eq added after eq creation

  [SerializeField]
  private int m_gold = 0;

  [SerializeField]
  private float m_capacity = 100;
  private float m_weight = 0;

  private int m_nextId = 0;

```

Ponownie jak poprzednio zachęcam do samodzielnej analizy.

Uwaga! Poniższe objaśnienie nie zawiera omówienia każdej metody, analizę tych najprostszych zostawiam jako zadanie domowe ;)

Unikalność przedmiotów

Pewnie co uważniejsi zauważyli, że każdy przedmiot posiada własne ID, które jednak nie było używane w klasie *Item*. ID będzie służyło nam do identyfikacji w danym ekwipunku i co ważne: **jest unikalne w obrębie tylko jednego ekwipunku!** A więc możliwa jest sytuacja gdy dwie postaci posiadające przedmioty będą miały przedmioty z tym samym ID.

Dodawanie przedmiotów

C#

```
  public bool AddItem(Item item)
  {
    // Adds item only if can store additional items (based on item weight)
    if (weight + item.weight <= capacity)
    {
      m_items.Add(m_nextId++, item);
      item.transform.SetParent(m_itemsParent);
      return true;
    }

    return false;
  }
```

Jest to trywialny feature, którego zadaniem jest sprawdzenie czy posiadamy odpowiednią ilość miejsca oraz jego zaktualizowanie w przypadku gdy mamy odpowiednią ilość miejsca.

Oprócz tego warto zauważyć, że to tutaj przydzielane jest ID do przedmiotów oraz że przedmioty tworzą pewną hierarchię i są przypisane jako dzieci tablicy wszystkich przedmiotów w ekwipunku

Usuwanie przedmiotów

```C#
59  public bool DeleteItem(int eid, bool destroy = false)
60  {
61    // Check if item is available
62    // if yes, check it should be destryoed from game
63    //  if yes, then destroy it
64    //  else change parent
65    if (m_items.ContainsKey(eid))
66    {
67      Item item = m_items[eid];
68      if (destroy)
69      {
70        Destroy(item.gameObject, 1);
71      }
72      else
73      {
74        item.transform.SetParent(transform.parent);
75      }
76
77      update_weight(weight - item.weight);
78      m_items.Remove(eid);
79
80      return true;
81    }
82
83    return false;
84  }
```

Usuwanie przedmiotów jest nieco trudniejsze, ponieważ wymaga sprawdzenia czy dany przedmiot powinien zostać zniszczony (np. w przypadku jego zużycia), czy też pozbywamy się go tylko z ekwipunku (np. został wyrzucony z ekwipunku, sprzedany – więc fizycznie jest wciąż dostępny w świecie gry).

ItemFood – przykład

Jako przykład klasy korzystającej z powyższych featerów stworzyłem klasę *ItemFood*, której implementacja prezentuje się następująco:

```C#
```

```
using UnityEngine;

public class ItemFood : Item
{
  public float restore_hp = 10;

  public float GetRestorationHP()
  {
    if (quantity == 0)
    {
      Destroy(this, 1.0f);
    }

    return restore_hp * GetStatsMultiplier();
  }

  public void Eat()
  {
    Use(ItemProperty.EATABLE);
  }

  protected override void on_item_use(ItemProperty useContext)
  {
    switch(useContext)
    {
      case ItemProperty.EATABLE:
        // only correct value
        // TODO restore caller hp
        eat();
        break;

      default:
        // should be never called, because of guard in parent
```

Jest to dość prosty przykład, widzimy tutaj wrapper na metodę *Use()* w postaci metody *Eat()*, która zajmuje się skonsumowaniem potrawy i zablokowaniem jej przed zniszczeniem. Nawet pobieżna analiza powyższego listingu powinna wystarczyć do zrozumienia powyższej klasy ;)

Podsumowanie

To tyle jeżeli chodzi o ten wpis, jak widzicie sam ekwipunek może być dość rozbudowany. W razie niejasności, wątpliwości, pomysłów – piszcie.

W kolejnej części zajmiemy się łączeniem ekwipunku GUI (a więc tego co dzisiaj zrobiliśmy) z samą grą, a więc umożliwimy podnoszenie przedmiotów, interakcję z nimi oraz zintegrujemy wcześniej napisaną broń z nowym systemem.

Na koniec tradycyjnie zachęcam do pobrania dema (link powyżej, ekwipunek można podejrzeć pod klawiszem [I]), komentowania, śledzenia bloga przez social-media oraz dzielenia się nim ze znajomymi.

Code ON!