Piszemy RPGo-Platformówkę (14) – Fireball! (sheadovas/poradniki/proj_platf_rpg/14-fireball/)

Sie 26, 2017 / proj_platf_rpg (sheadovas/category/poradniki/proj_platf_rpg/)

Dodajemy system umiejętności

Hej, w dzisiejszej części zajmiemy się napisaniem podwalin pod system umiejętności oraz zaimplementujemy umiejętność leczenia.

Tradycyjnie zachęcam do zagrania w [demo (https://github.com/sheadovas/proj_platf_rpg/releases/tag/1.11)] oraz samodzielnego zapoznania się z [napisanym kodem (https://github.com/sheadovas/proj_platf_rpg/compare/2e30c849d2a1949ee63cec7ea9b539fad0ffab0f...6555ed90bcd62b759421df0a4010b8812718ee7b)].

Umiejętności – teoretycznie

System umiejętności (choćby w najprostszej formie) jest dostępny w wielu grach – nie tylko RPG. Dzięki niemu graczowi jest łatwiej się wczuć w grę i odczuwać rozwój zarówno postaci (jest coraz silniejszy) – co przekłada się na satysfakcję z grania – jak i przyczynie się do lepszego poznawania świata gry (bo np. do przejścia w konkretne miejsce wymagana jest umiejętność przechodzenia przez ściany). Dla lepszego zobrazowania systemu umiejętności w grach spójrzmy na przykłady.

Systemy umiejętności w grach

Prosty

Pierwszym (bardziej prymitywnym / niewidocznym dla gracza) jest ten zastosowany w grze *Assassin's Creed: Brotherhood* gdzie od pewnego momentu gry jesteśmy w stanie wezwać swoich rekrutów, aby ci zlikwidowali za nas cele.

(https://i1.wp.com/szymonsiarkiewicz.pl/wp-content/uploads/2017/08/Apprentices.jpg)
Assassin Creed: Brotherhood

Niewątpliwie mamy tutaj swego rodzaju umiejętność: ta funkcjonalność była do pewnego momentu zablokowana (odblokowuje ją przejście przez zadanie), przy czym do jej wykonania wymagane jest posiadanie zasobów (wolnych rekrutów, czyli takich niewysłanych na misje) oraz odczekanie odpowiedniej ilości czasu (cooldown). Można też ją ulepszać przez zwiększenie liczby posiadanie rekrutów oraz wysyłanie ich na misje tak aby zwiększyli swoją umiejętność walki.

Zaawansowany

Drugim systemem umiejętności, tym „zaawansowanym" można nazwać każdy widoczny dla gracza system posiadający drzewko rozwoju. Tutaj wszystko zależy od twórców, zazwyczaj polegają one na ulepszaniu / odblokowywaniu konkretnych umiejętności przy użyciu punktów umiejętności (te najczęściej się zbiera przez awans postaci na wyższy poziom). Im wyższy poziom umiejętności – tym lepsza jego skuteczność.

Same umiejętności też mogą działać pasywnie / aktywnie, t.j. odpowiednio: zawsze / z koniecznością manualnej aktywacji. Bardzo często też do odblokowania jednej umiejętności wymagane jest odblokowanie innej.

Dla przykładu spójrzmy na kilka drzewek z gier:

(https://i0.wp.com/szymonsiarkiewicz.pl/wp-content/uploads/2017/08/anna-jasinski-skilltree01.jpg)

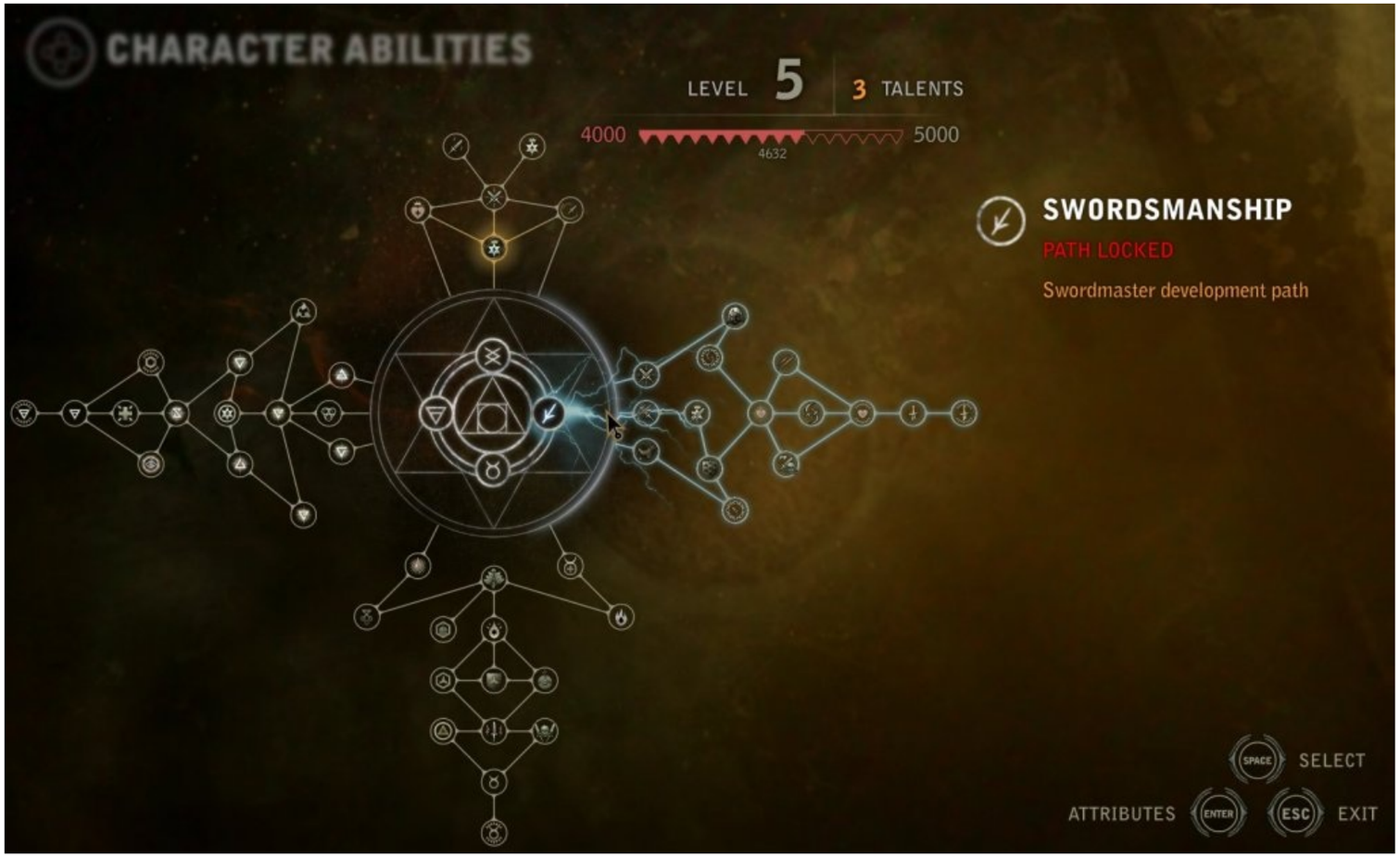
Ori and the Blind Forest

Ori and the Blind Forest przedstawia najprostszy z możliwych drzewek: rozwijamy 3 predefiniowane umiejętności. Każdą z nich możemy ulepszać jedynie w sposób liniowy (tzn aby ulepszyć jedno ulepszenie należy zebrać odpowiednią ilość punktów umiejętności oraz odblokować wszystkie poprzednie ulepszenia).

(https://i0.wp.com/szymonsiarkiewicz.pl/wp-content/uploads/2017/08/the_witcher_3_wild_hunt_character_new_rgb-100660323-orig.png)

Wiedźmin 3

Dużo ciekawiej sprawa przedstawia się w rodzimym *Wiedźminie 3*, tutaj mamy wiele poddrzew (ścieżek) rozwoju, gdzie do odblokowania danej umiejętności należy wcześniej zainwestować odpowiednią ilość punktów w tej samej gałęzi rozwoju. Co ciekawe aby z samej umiejętności móc skorzystać, to oprócz jej odblokowania należy uaktywnić ją przez przeciągnięcie jej do drzewa aktywnych umiejętności widocznego po prawej stronie ekranu (pomijam możliwość zwiększania efektu umiejętności przez łączenie ich w grupy jednego rodzaju i nakładanie specjalnych przedmiotów – mutagenów).



(https://i2.wp.com/szymonsiarkiewicz.pl/wp-content/uploads/2017/08/bQnNd.jpg)
Wiedźmin 2

„Najciekawsze”, a co za tym idzie: najtrudniejsze w implementacji są drzewa umiejętności, które wyglądają jak grafy. Tj. do jednej umiejętności prowadzi wiele ścieżek i wymagane jest odblokowane wszystkich (lub przynajmniej jednego) dochodzących węzłów umiejętności, aby odblokować jedną wymarzoną umiejętność. Przykładem takiego drzewa jest to z *Wiedźmina 2.*

Same drzewa umiejętności potrafią być jeszcze bardziej rozbudowane, te gigantyczne zazwyczaj spotyka się w grach MMORPG. Jeżeli ktoś jest ciekawy jak duże potrafią to być to zachęcam do rzucenia okiem na drzewko z *Path of Exile*.

~~Fireball!~~ Heal!

Skoro już wiemy „z czym się je” system umiejętności, to czas zabrać się do pracy. Tradycyjnie zabierzmy się za wyznaczenie celów:

- zajmujemy się umiejętnościami „aktywnymi”,
- do odblokowania / ulepszenia umiejętności będą nam służyły punkty umiejętności,
- do odblokowania umiejętności należy spełnić wymagania:
  1. gracz musi posiadać odpowiednią ilość punktów umiejętności.
  2. do odblokowania umiejętności może być wymagane posiadanie innej umiejętności na odpowiednim poziomie.

Kilka uwag odnośnie powyższych punktów:

- zaimplementujemy możliwość posiadania tylko jednej umiejętności aktywnej (tj. jeżeli mamy odblokowane umiejętności: „FIreball”, „Heal” to musimy wybrać w menu którą umiejętność chcemy móc użyć po wciśnięciu klawisza);
- umiejętności mają swoje poziomy, które wpływają na ich statystyki;
- trzecia kropka implikuje to, że nasze drzewo będzie miało wygląd grafu (drzewa) w sensie matematycznym (chociaż implementacja wizualna / UI na to nie będzie bezpośrednio wskazywała).

Skill

Zacznijmy od najbardziej podstawowego elementu jakim jest pojedyncza umiejętność (*Skill)*.

Tradycyjnie zachęcam do samodzielnej analizy, po czym zajmiemy się analizą konkretnych fragmentów.

Skill.cs C#

```
using UnityEngine;
using System.Collections;

public class Skill : MonoBehaviour
{
  public struct SkillRequirement
  {
    public Skill skill;
    public int requiredLevel;
  }

  public bool isUnlocked
  {
    get { return m_isUnlocked; }
  }

  public int skillLevel
  {
    get { return m_skillLevel; }
  }

  public string skillName = "Skill";

  protected int m_skillLevel = 0;
  protected bool m_isReadyToUse = true;
  protected PlayableCharacter m_owner;
  protected SkillTree m_skillTree;

  // end extra values can be here, like "mana cost"


  [SerializeField]
  protected float m_cooldown = 0;
```

Oczywistym faktem jest to, że ta klasa jest dopiero podstawą pod właściwe kody, więc do pewnego stopnia jest wirtualna.

Skill.cs C#

```
public class Skill : MonoBehaviour
{
  public struct SkillRequirement
  {
    public Skill skill;
    public int requiredLevel;
  }

  public bool isUnlocked
  {
    get { return m_isUnlocked; }
  }

  public int skillLevel
  {
    get { return m_skillLevel; }
  }

  public string skillName = "Skill";

  protected int m_skillLevel = 0;
  protected bool m_isReadyToUse = true;
  protected PlayableCharacter m_owner;
  protected SkillTree m_skillTree;

  // end extra values can be here, like "mana cost"


  [SerializeField]
  protected float m_cooldown = 0;

  [SerializeField]
  protected float m_skillStatsMultiplier = 1.5f;
```

Na powyższym listingu widzimy kolejno implementację wymagań dotyczących umiejętności (*SkillRequirement)*, parametry wyjściowe potrzebne w innych modułach (typu: informacja czy umiejętność jest odblokowana, poziom umiejętności, jej nazwa) oraz parametry pomocnicze (resztę parametrów omówimy za chwilę).

Skill.cs C#

```
  public virtual void Use(GameObject target)
  {
    if(!isUnlocked)
    {
      Debug.Log(string.Format("Cannot use {0} because skill is not unlocked", skillName), this);
      return;
    }

    // if skill still needs time to cooldown...
    if (!m_isReadyToUse)
      return;

    // begin waiting time
    Cooldown();

    // NOTICE rest is done by implementation in non-virtual class
  }
```

Każdą umiejętność można użyć (w jakiś sposób), jej użycie może być zależne od celu (

`GameObject target`

) np. możemy założyć, że umiejętność leczenia można użyć na sobie lub sojusznikach, tutaj ilość uzdrowionych punktów życia może być zależna od tego czy użyliśmy jej na sobie (leczymy mniej PŻ), czy na sojuszniku (leczymy więcej PŻ). Tego typu mechanika bardzo często używana jest w grach MMO, wymusza ona poniekąd granie w grupie, a nie „solo”.

Warto zauważyć, że powyżej mamy zaimplementowany jedynie prolog metody *Use* będący wspólny dla większości umiejętności. Poprawny sposób implementacji z użyciem powyższego prologu będzie pokazany poniżej.

W prologu najpierw sprawdzamy czy możemy użyć danej umiejętności, a więc: czy umiejętność jest odblokowana oraz czy cooldown już minął. Następnie uruchamiamy czas cooldownu. Dalsza część (a więc samo użycie) jest implementowane przez klasy nadrzędne – dziedziczące po *Skill*.

Skill.cs C#

```
67   public bool Unlock()
68   {
69     // check if we can unlock it
70     if (m_skillTree.availableSkillpoints < m_nextLevelCost)
71       return false;
72
73     // check we have to unlock it, if not check we are passing requirements
74     if (!m_isUnlocked && !CheckRequirements())
75       return false;
76
77     // unlock
78     m_skillTree.availableSkillpoints -= m_nextLevelCost;
79     m_nextLevelCost = CalculateNextLevelCost();
80     m_skillLevel++;
81     m_isUnlocked = true;
82
83     return true;
84   }
```

Sprawdzenie tego czy umiejętność możemy odblokować wymaga od nas odwołania się do drzewka umiejętności, w którym to przechowujemy punkty umiejętności. Na początku oczywiście sprawdzamy, czy posiadamy ich dostateczną ilość, po czym sprawdzamy czy wszystkie wymagania związane z umiejętnością są spełnione (a więc odblokowanie innych umiejętności na dany poziom):

Po wykonaniu tego kroku aktualizujemy samą umiejętności oraz koszt odblokowania następnego poziomu umiejętności.

Resztę metod z tej klasy pozostawiam do samodzielnej analizy (są raczej proste i w moim odczuciu nie wymagają dodatkowego komentarza).

SkillHeal

W tym momencie chciałbym wyjaśnić nazwę bieżącego „dużego” paragrafu (~~*Fireball!*~~ *Heal!*). Otóż, nie zaimplementujemy umiejętności kuli ognia (tytułowego bohatera tej lekcji), a zajmiemy się umiejętnością leczenia ;) (wynika to z faktu, że przy leczeniu nie potrzeba żądnych dodatkowych „fajerwerków” graficznych, których i tak nie pokazuję w poradniku).

SkillHeal.cs C#

```
1  using UnityEngine;
2
3  public class SkillHeal : Skill
4  {
5    public float restoreHP = 5;
6
7    public override void Use(GameObject target)
8    {
9      base.Use(target);
10
11     // self-healing skill, logarithmic increase
12     m_owner.stats.hp += restoreHP + restoreHP * Mathf.Log(m_skillLevel, 2);
13   }
14 }
```

Powyższa klasa jest niezwykle prosta. Widzimy na niej poprawne użycie predefiniowanego prologu (

`base.Use(taget)`

) oraz zwiększenie PŻ właściciela umiejętności o bazową ilość uzdrawiania życia wraz z bonusem wynikającym z posiadanego poziomu. To tyle w temacie :P

SkillTree

Przejdźmy teraz do „logicznej” części związanej z drzewkiem umiejętności. Gorąco zachęcam do samodzielnego zapoznania się z poniższym listingiem:

SkillTree.cs C#

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

public class SkillTree : MonoBehaviour
{
  public int availableSkillpoints = 0;

  // needed only in object creation as initial list
  public Skill[] availableSkills;

  [HideInInspector]
  public List<Skill> learntSkills;
  [HideInInspector]
  public List<Skill> unknownSkills;

  [SerializeField]
  private Transform m_containerLearnt;
  [SerializeField]
  private Transform m_containerUnknown;

  private PlayableCharacter m_owner;
  private Skill m_activeSkill;

  public bool Learn(Skill skill)
  {
    return skill.Unlock();
  }

  public void UseSkill()
  {
    // Use active skill, in our implementation we are allowing for only 1 active skill
    if(m_activeSkill != null)
    {
```

„Logiczne” drzewko umiejętności posiada m.in. ilość dostępnych punktów umiejętności, bazową pulę umiejętności których gracz może się nauczyć, właściciela oraz aktywną umiejętność.

Uwaga! Obecnie sama gra nie nagradza gracza punktami umiejętności: te obecnie są nadane „z góry” jako „hack” z poziomu sceny Unity. Sam system wymaga dodania tego feater’a co zostawiam jako zadanie domowe.

SkillTree.cs C#

```
59  private void Start()
60  {
61    if (availableSkills != null)
62    {
63      foreach (Skill skill in availableSkills)
64      {
65        skill.SetOwner(m_owner, this);
66
67        if (skill.isUnlocked)
68        {
69          learntSkills.Add(skill);
70          skill.transform.SetParent(m_containerLearnt);
71        }
72        else
73        {
74          unknownSkills.Add(skill);
75          skill.transform.SetParent(m_containerUnknown);
76        }
77
78        skill.transform.localScale = Vector3.one;
79      }
80
81      availableSkills = null;
82    }
83  }
```

Po uruchomieniu skryptu pula aktywnych umiejętności dzielona jest na 2 części: znanych umiejętności oraz tych do nauczenia. Dla lepszego wizualnego efektu znane umiejętności i nieznane są przypisane do odpowiednich obiektów na scenie jako dzieci.

SkillTree.cs C#

```
24  public bool Learn(Skill skill)
25  {
26    return skill.Unlock();
27  }
28
29  public void UseSkill()
30  {
31    // Use active skill, in our implementation we are allowing for only 1 active skill
32    if(m_activeSkill != null)
33    {
34      m_activeSkill.Use(null);
35    }
36  }
```

Sama nauka i użycie umiejętności jest prostym odwołanie się do odpowiednich metod danej umiejętności. Warto zauważyć, że dla *UseSkill* w obecnej implementacji kontekst użycia umiejętności (cel) jest zawsze ustawiany na *null*.

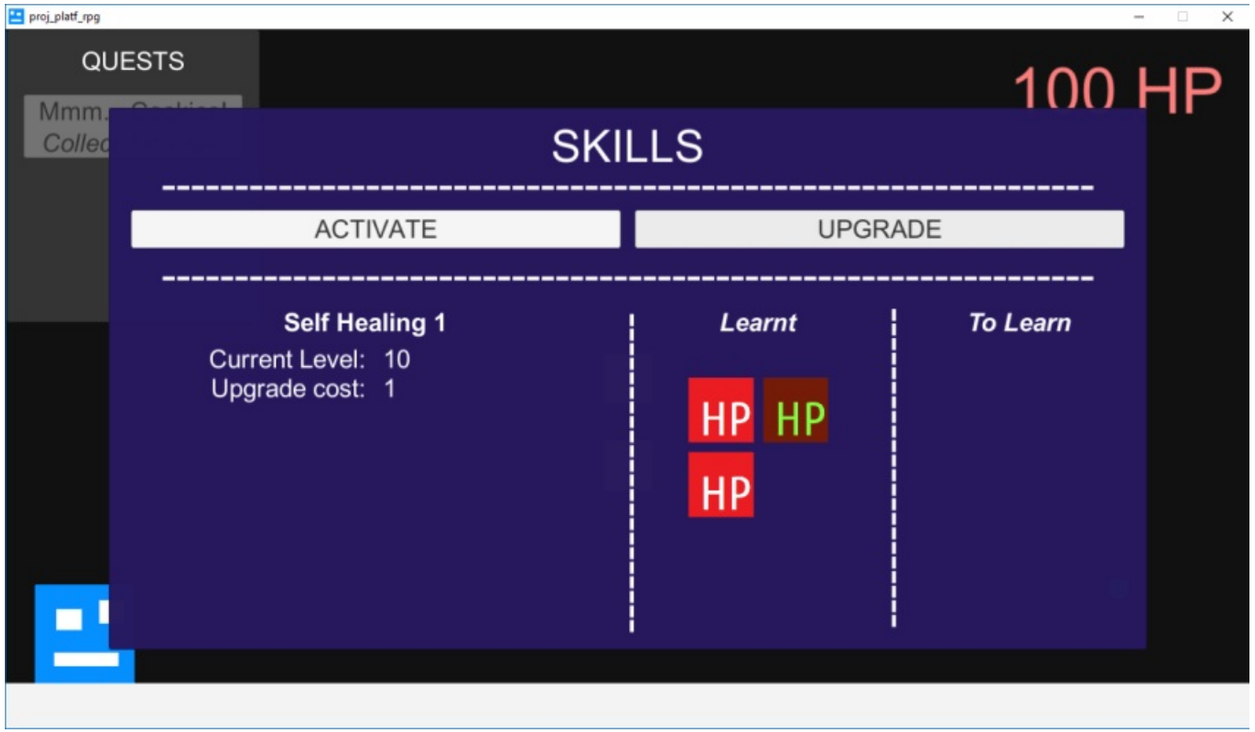
SkillTree.cs C#

```
38  public void SetActiveSkillOnSlot(Skill skill)
39  {
40    // here we can add param "slot_number" to this method
41    // that defines assigning slot id
42    // we have only one active skill, so we dont need this
43
44    if(skill.isUnlocked && learntSkills.Contains(skill))
45    {
46      // simple lock-check that evades activating skill from outside the skill tree
47      m_activeSkill = skill;
48    }
49  }
```

Aby móc użyć jakiejś umiejętności, to oprócz jej odblokowania należy ją również aktywować. W tym celu przed ustawieniem umiejętności jako aktywnej najpierw sprawdzamy, czy znajduje się w liście odblokowanych umiejętności w lokalnym drzewie.

SkillTreeVisual

Czas na część wizualną naszego drzewa, która wymaga doszlifowania pod kątem artystycznym – jednakże dla nas jest wystarczająca. Dla lepszego zrozumienia spójrzmy na screen reprezentujący całość już po implementacji (tak, nie jest za piękny ;)):



(https://i1.wp.com/szymonsiarkiewicz.pl/wp-content/uploads/2017/08/rpg-skills.png)
SkillTreeVisual

Elementy warte uwagi:

- przyciski „Activate”, „Upgrade” do aktywacji / ulepszania wybranej umiejętności,
- ramka z właściwościami wybranej umiejętności (nazwa, poziom, koszt ulepszenia),
- ramka z poznanymi umiejętnościami (tutaj: same umiejętności leczenia, bo tylko taką zaimplementowaliśmy), umiejętność szaro-zielona jest wybrana „do edycji”,
- ramka z umiejętnościami jeszcze nieznanymi, analogiczna do ramki „Learnt”.

SkillTreeVisual.cs C#

```
using UnityEngine;
using UnityEngine.UI;

public class SkillTreeVisual : MonoBehaviour
{
  public const string TEXT_BUTTOIN_LEARN_UNKNOWN = "LEARN";
  public const string TEXT_BUTTOIN_LEARN_KNOWN = "UPGRADE";

  public SkillTree skillTree;

  public Transform skillsLearnt;
  public Transform skillsUnknown;

  public Button skillBtnActivate;
  public Button skillBtnUpgrade;

  public Text skillSelectedName;
  public Text skillSelectedProps;

  protected Skill m_selectedSkill = null;
  protected Color m_oldColor;

  public void OnSkillSelect(Skill skill)
  {
    // restore color
    if(m_selectedSkill != null)
    {
      // reset color change
      m_selectedSkill.GetComponent<Image>().color = m_oldColor;
    }

    // update selection
    m_selectedSkill = skill;
```

Z racji, że powyższy listing jest bardzo mocno powiązany z Unity, to omówię jedynie ogólnie niektóre fragmenty. Resztę pozostawiam do analizy dla chętnych.

SkillTreeVIsual.cs C#

```
  protected void toggle_ui(bool active)
  {
    skillBtnActivate.interactable = active;
    skillBtnUpgrade.interactable = active;

    if(active)
    {
      // update gui props
      skillBtnUpgrade.GetComponentInChildren<Text>().text =
        m_selectedSkill.isUnlocked ?
        TEXT_BUTTOIN_LEARN_KNOWN :
        TEXT_BUTTOIN_LEARN_UNKNOWN;

      skillSelectedName.text = m_selectedSkill.skillName;
      skillSelectedProps.text = string.Format(
          "{0}\n{1}", m_selectedSkill.skillLevel, m_selectedSkill.CalculateNextLevelCost()
      );

      // check xp
      skillBtnUpgrade.interactable = m_selectedSkill.CheckRequirements();
    }
  }
```

Powyższa metoda służy do włączania/wyłączania *focusu* przyciski. W przypadku gdy ustawiamy go na *true* to oprócz aktywowania przycisków aktualizujemy również napisy w ramce z właściwościami umiejętnościami. Po czym w przypadku gdy spełniamy wymagania związane z ulepszeniem umiejętnościami, to aktywujemy przycisk do ulepszenia umiejętności.

SkillTreeVisual.cs

```
79  private void Start()
80  {
81    toggle_ui(false);
82
83    for(int i=0; i<skillsUnknown.childCount; ++i)
84    {
85      Transform child = skillsUnknown.GetChild(i);
86      init_buttons(child);
87    }
88
89    for (int i = 0; i < skillsLearnt.childCount; ++i)
90    {
91      Transform child = skillsLearnt.GetChild(i);
92      init_buttons(child);
93    }
94
95    // NOTICE We are not loading learnt skills from previous game / game save.
96    // NOTICE You can do this here :)
97  }
```

Na samym początku focus na przyciski jest wyłączony (ponieważ nie mamy wybranej żadnej umiejętności) oraz aktualizujemy przyciski umiejętności (m.in ustawiamy im z poziomu skryptu odpowiedni *onClick*).

SkillTreeVisual.cs C#

```
23  public void OnSkillSelect(Skill skill)
24  {
25    // restore color
26    if(m_selectedSkill != null)
27    {
28      // reset color change
29      m_selectedSkill.GetComponent<Image>().color = m_oldColor;
30    }
31
32    // update selection
33    m_selectedSkill = skill;
34
35    // modify & store color
36    Image img = m_selectedSkill.GetComponent<Image>();
37    m_oldColor = img.color;
38    img.color = new Color(0.5f, 1.0f, 0.25f);
39
40    // update ui
41    toggle_ui(true);
42  }
```

Wybranie umiejętności odbywa się przez kliknięcie na nią, po czym w pierwszym kroku zmieniamy kolory przycisku, dzięki czemu widzimy która umiejętność została kliknięta. Następnie aktualizujemy aktualnie wybraną umiejętność i aktywujemy focus przycisków (jednocześnie aktualizujmy ramkę właściwości umiejętności).

SkillTreeVIsual.cs C#

```
44  public void OnActivateClicked()
45  {
46    skillBtnActivate.interactable = false;
47    skillTree.SetActiveSkillOnSlot(m_selectedSkill);
48  }
49
50  public void OnUpgradeClicked()
51  {
52    skillTree.Learn(m_selectedSkill);
53    toggle_ui(true); // refresh ui
54  }
```

Reakcja na wciśnięcie klawiszy jest dość proste i ogranicza się głównie do wywołań odpowiednich metod wraz z aktualizacją UI.

Podsumowanie

Jeżeli chodzi o drzewka umiejętności, to już jest wszystko co chciałem Wam przekazać. Wyszło tego całkiem sporo, a jest tego nawet jeszcze więcej – sporo zostało „połknięte” przez to, że nie zajmujemy się Unity. Dlatego gorąco zachęcam wszystkich chętnych do rzucenia okiem na to jak całość wygląda w samym edytorze tego silnika :)

W tym miejscu – tradycyjnie – zachęcam do zagrania w demo, podzieleniem się swoim komentarzem pod tym wpisem, śledzeniem blogiem przez social-media (wszystkie linki są w panelu po prawej stronie) oraz udostępnieniem linku do tego wpisu, blogu wśród swoich znajomych (z góry: wielkie dzięki).

Kolejna, a zarazem ostatnia część już niedługo…

Code ON!