RAPPORT

- The Cinema Slot Machine -

SOMMAIRE

I. DESCRIPTION DU PROJET

Nous avons réalisé un jeu en 2D simulant une **machine à sous sur le thème du cinéma**. Le but est de gagner le plus d'argent possible en obtenant à chaque tour trois images identiques sur la machine. Chaque tour de jeu est payant, il faut donc veiller à garder un équilibre entre l'argent entrant et l'argent sortant. Le jeu se termine quand l'argent est écoulé.

Lorsque le joueur commence le jeu, il doit choisir **son film préféré** parmi trois films. La machine prend alors une **couleur** proche des teintes du film sélectionné. Il existe en tout 13 couleurs de machines différentes.

Le joueur commence avec une **somme de départ** lui permettant de jouer. Pour commencer, il doit **tirer le manche**. Plus il reste appuyé sur le manche, plus il augmente les chances que la machine tourne vite (vitesse du lancer). Si le joueur obtient une combinaison de deux ou trois images, il gagne un prix selon les **scores** indiqués en haut de la page. Les **prix par combinaisons** changent aussi aléatoirement à chaque partie. S'il lui reste de l'argent, le joueur peut alors rejouer. Sinon, c'est « Game Over ».

Nous avons choisi de faire une machine à sous parce que nous trouvions intéressant de recréer ce jeu que l'on peut rencontrer dans des fêtes foraines ou casinos, et, car il est simple à comprendre. Il était aussi plus facile d'y intégrer plusieurs notions de hasard. Nous avons aussi ajouté des fonctionnalités originales comme les couleurs différentes de machines ou le changement de gain par image, pour apporter plus de jouabilité, de possibilités et de hasard. Le thème du cinéma nous est venu en pensant à des **objets reconnaissables**. Ces objets, symboles de films, ajoutent un plus au jeu. Par exemple, le joueur aura peut-être envie de deviner les films cachés derrière ceux-ci.

Les images choisies pour être dans la machine à sous sont donc toutes des références à des films connus :

Le seigneur des anneaux

Forrest Gump

Seul au monde

Matrix

Retour vers le futur

Le fabuleux destin d'Amelie Poulain

Le Parrain

Au niveau des **technologies utilisées**, pour le design nous avons fait le design avec Adobe Illustrator et Photoshop. Pour le code, tout a été tout sur Unity : ce logiciel est très pratique pour la programmation de jeu et nous étions assez à l'aise dessus grâce à d'autres projets réalisés précédemment. Unity utilise des scripts en C# (voir ***VI. annexes : fichier code MathProb.cs en C#***).

II. VARIABLES ALÉATOIRES

a) Couleurs de la machine

Nous avons randomisé les **couleurs de la machine**. Nous avions 13 images de machines de couleurs différentes et nous voulions donc faire participer le joueur pour qu'il puisse influer sur la probabilité de tomber sur une machine d'une couleur particulière. Pour cela le joueur choisit un film : il aura alors plus de chance de tomber sur une couleur qui a un rapport avec la teinte du film choisi.

Pour modéliser cela, nous avons fait trois tableaux de 13 probabilités associés aux trois types de films. Ensuite, une machine est choisie aléatoirement selon ces pourcentages de chance (tableau de probabilité). Nous utilisons donc une **loi discrète avec pourcentages de chances**.

```
float[] probsColorsMachine = new float[] { 0.1f, 1f, 0f, 0f, 0f, 0f, 0f, 1f, 1f, 0f, 1f, 1f, 1f };

int colorMachine = MathProb.intInRangeProbs(1, 13, probsColorsMachine);

GameObject machineUI.GetComponent<SpriteRenderer>().sprite = Resources.Load<Sprite>("MACHINES/Machine_" + colorMachine);
```

b) Vitesse du lancer

Nous avons aussi randomisé la **vitesse du lancer**. La vitesse à laquelle défilent les images lorsqu'on tire sur le manche varie donc entre 2 ms et 30 ms. Cela aura une influence sur le résultat du lancer (*voir **III. Structures de corrélation***).

Pour modéliser cela, nous avons choisi d'utiliser une **loi binomiale**. La probabilité est par rapport à la force lors de l'appui sur le manche (qui est entre 0 et 1). Cette loi discrète, qui se rapproche toutefois d'une loi normale, correspond bien à la vitesse, car elle permet de faire différents échelons assez rapprochés. De plus, elle évite les extrémités et permet ainsi d'éviter un tirage trop lent ou trop rapide de manière récurrente.

```
float powerHandle = CalculateHoldDownForce(holdDownTime);

float timeInterval = (MathProb.intInRangeBinomial(2, 30, 1-powerHandle))/1000.0f;
```

c) Prix pour les combinaisons

Nous avons randomisé **les prix pour les combinaisons d'images**. Lorsqu'on a trois images identiques, on gagne un certain prix. Les combinaisons changent à chaque partie. Les prix peuvent prendre les valeurs 100, 400, 600, 800, 1500, 2000 et 5000.

Pour modéliser cela, nous avons choisi d'utiliser une **loi discrète uniforme**. On simule un tirage sans remise : on mélange un tableau (« Shuffle ») des valeurs des prix (en utilisant un tirage discret uniforme) et on l'associe aux images. Cette loi discrète correspond donc bien au fait de choisir entre un nombre restreint de possibilités tout en faisant un tirage sans remise.

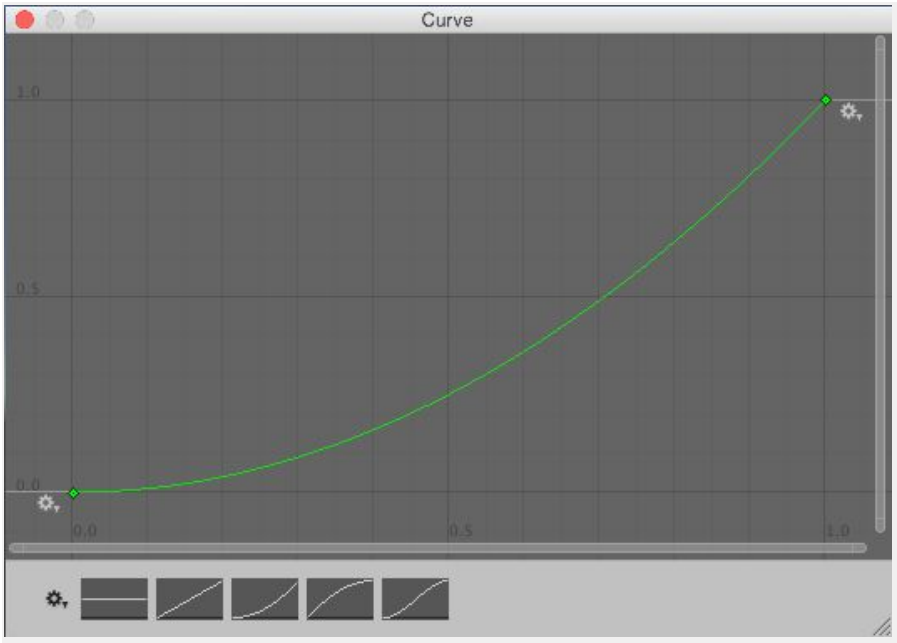
```
Dictionary<string, int> imagesScores = new Dictionary<string, int>()
{
        {"ring", 200},
        {"wilson", 400},
        {"hoverboard", 600},
        {"horse_head", 800},
        {"box_chocolates", 1500},
        {"garden_gnome", 2000},
        {"pills", 5000}
};

imagesScores = MathProb.randomizeDictionary(imagesScores);
int i = 0;
GameObject[] imagesScoresUI;
foreach (var item in imagesScores.OrderBy(key => key.Value))
{
        imagesScoresUI[i].GetComponent<SpriteRenderer>().sprite                    =
        Resources.Load<Sprite>("SCORES/Score_" + item.Key);
        i++;
}
```

d) Résultat du lancer

Nous avons randomisé le **résultat du lancer**. Le résultat est constitué de trois images parmi les 7 possibles. Si on obtient deux ou trois fois la même image, alors on gagne le prix associé. Pour cela, on fait un défilement de trois bandes avec sur chacune les 7 images. On influe donc aléatoirement sur le défilement de chaque bande (temps qu'elle va continuer à défiler).

Pour modéliser cela, nous avons fait le choix d'utiliser une **loi continue avec une courbe** Unity. On fait une courbe telle que celle-ci :



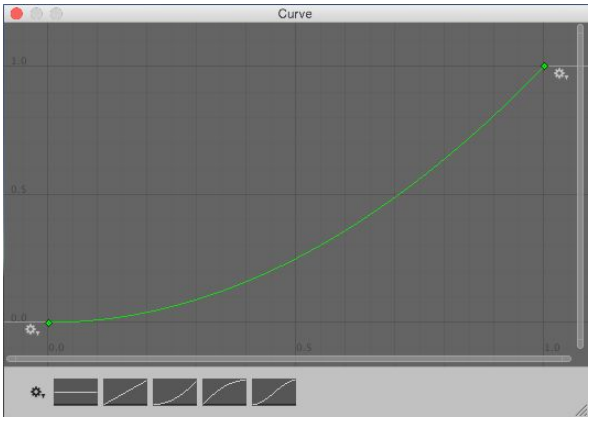
On prend ensuite au hasard sur la courbe un point en abscisse et on regarde son ordonnée. On aura ici plus de chance de tourner moins longtemps (donc plus près de 60 avancées que 120 avancées). De plus, cette courbe est corrélée à la vitesse (voir **III. Structures de corrélation**).

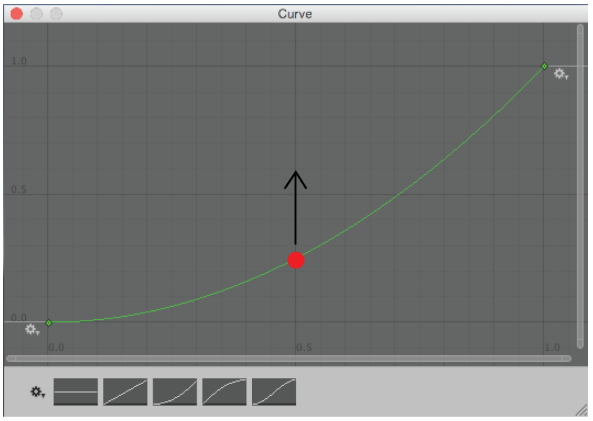
```
AnimationCurve curveProb = new AnimationCurve(new Keyframe(0.0f, 0.0f), new
Keyframe(0.5f, 0.25f), new Keyframe(1.0f, 1.0f));

int randomValue = Mathf.RoundToInt(MathProb.uniformFloatInRangeCurve(60,
120, curveProb));
```

III. STRUCTURE DE CORRÉLATION

Nous avons décidé d'implémenter une **structure de corrélation entre la vitesse et les résultats du tirage**. La vitesse en probabilité influence la courbe utilisée pour le résultat du tirage. Elle influe notamment en faisant augmenter ou baisser le point central de la courbe. Ainsi, si la vitesse est grande, le point central monte. On a donc plus de chance que les images tournent plus longtemps avant de s'arrêter.
Le but de cette manœuvre est de faire tourner la machine plus longtemps si elle va vite, et aussi laisser une part d'action au joueur. Il a un rôle qui lui permet d'influencer sur son jeu.





Le joueur influence aussi l'aléatoire en **choisissant un film** (influence la couleur de la machine a)) et en restant **plus ou moins appuyé sur le manche** (influence la vitesse b)).

IV. DIFFICULTÉS

Une des principales difficultés rencontrées pendant la conception de ce jeu fut l'**implémentation des lois de probabilités**. Nous avons eu du mal à transformer des lois mathématiques pour des choses plus concrètes appartenant au monde du jeu et de la programmation.

Par exemple, comment transformer une loi de Bernoulli en un tirage random entre des entiers, ou comment utiliser une loi de poisson pour choisir une image parmi plusieurs autres ? Nous avons donc dû repenser les lois de probabilités pour avoir le bon nombre de discrètes et de continu, tout en respectant bien notre cahier des charges. Il fallait bien souvent penser à l'envers !

Ce fut aussi compliqué de **réaliser un jeu** (concept, programmation et design), même petit, avec tous les autres projets que nous avions. Non n'avons donc pas fait le tirage aléatoire d'images pour la machine (*voir **fiche projet initiale***), car cela aurait demandé encore plus de temps pour faire d'autres images.

V. CONCLUSION

Ce projet nous a pris beaucoup de temps. De la conception à la programmation, en passant par le design, **créer un jeu ludique et qui fonctionne** n'est pas une tâche facile. Nous redoutions aussi les **lois de probabilités**, non pas dans la compréhension, mais dans l'incorporation de ces mathématiques au code.

Néanmoins, nous avons **réussi à faire fonctionner le jeu** : nous sommes contents du résultat. Nous avons **compris beaucoup de choses** pendant ce projet et avons pu **appliquer** concrètement des notions de mathématiques complexes.

VI. ANNEXES

...\Desktop\Cinema slot machine\Assets\SCRIPTS\MathProb.cs 1

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using System.Linq;
4  using UnityEngine;
5
6  public static class MathProb
7  {
8      // Loi continue uniforme
9      public static float uniformFloatInRange(float min , float max)
10     {
11         return Random.value * (max - min) + min;
12     }
13
14     // Loi continue (avec courbe)
15     public static float uniformFloatInRangeCurve(float min, float max,
         AnimationCurve curve)
16     {
17         return curve.Evaluate(Random.value) * (max - min) + min;
18     }
19
20     // Loi discrète uniforme
21     public static int uniformIntInRange(int min, int max)
22     {
23         return Mathf.RoundToInt(uniformFloatInRange(min , max));
24     }
25
26     // Loi discrète avec probabilités
27     public static int intInRangeProbs(int min, int max, float[] probs)
28     {
29         float total = 0;
30         float[] probs2 = new float[max];
31         for (int j = 0; j < probs2.Length; j++)
32         {
33             if (j >= probs.Length)
34             {
35                 probs2[j] = 0.5f;
36             }
37             else
38             {
39                 probs2[j] = probs[j];
40             }
41             total += probs2[j];
42         }
43
44         float randomPoint = Random.value * total;
45         for (int i = 0; i < probs.Length; i++)
46         {
47             if (randomPoint < probs2[i])
48             {
49                 return i + min;
50             }
51             else
52             {
53                 randomPoint -= probs2[i];
54             }
55         }
```

```
56
57          return (probs.Length - 1) + min;
58      }
59
60      // Loi discrète Bernoulli
61      public static bool boolInRangeBernoulli(int min, int max, float p)
62      {
63          return (uniformIntInRange(min, max)) <= (p * (max - min) + min) ?
              true : false;
64      }
65
66      // Loi discrète Bimoniale
67      public static int intInRangeBinomial(int min, int max, float p)
68      {
69          int sum = min;
70          for(int i=min; i<max; i++)
71          {
72              sum += boolInRangeBernoulli(min, max, p) ? 1 : 0;
73          }
74          return sum;
75      }
76
77      // Loi discrète uniforme avec test si existe déjà dans tableau
78      public static int uniqueRandomInt(int min, int max, int[] array) {
79          int val = uniformIntInRange(min, max);
80          while (array.Contains(val)) {
81              val = uniformIntInRange(min, max);
82          }
83          return val;
84      }
85
86      // Tableau aléatoire (Shuffle)
87      public static T[] randomizeArray<T>(T[] array)
88      {
89          for (int i = 0; i < array.Length; ++i)
90          {
91              int randomIndex = uniformIntInRange(0, array.Length-1);
92              T temp = array[randomIndex];
93              array[randomIndex] = array[i];
94              array[i] = temp;
95          }
96          return array;
97      }
98
99      // Dictionnaire Aléatoire (Shuffle)
100     public static Dictionary<TKey, TValue> randomizeDictionary<TKey, TValue>
          (this Dictionary<TKey, TValue> dict)
101     {
102         var dictValues = dict.Values.ToArray();
103         dictValues = randomizeArray(dictValues);
104         for(int i=0; i< dictValues.Length; i++)
105         {
106             dict[dict.ElementAt(i).Key] = dictValues[i];
107         }
108         return dict;
109     }
```