Date : 12/09/2023

Séance 1 : Introduction à la programmation

Objectifs :

- Initiation à l'algorithmique
- Initiation à l'environnement Visual Studio
- Initiation à la programmation en C#

1. Exercices

a) **Les algorithmes et les jeux vidéo :** allez sur le site blockly-games et jouez le jeu **labyrinthe** jusqu'à son dernier niveau. À chaque étape franchie, vous devez faire une capture d'écran des blocs utilisés pour la solution proposée et du code source correspondant. Faites le maximum des niveaux que vous pouvez pendant 30 minutes et ajoutez vos réponses dans un fichier Word. Lorsque le temps prévu est écoulé, comparez vos réponses entre les niveaux et discutez avec vos collègues. Que retrouvez-vous ?

Déposez votre fichier sur la plateforme de cours.

Lien : https://blockly-games.appspot.com/maze

b) **Logiciel Visual Studio** : vérifiez que votre ordinateur contient une installation de l'environnement de développement « Visual Studio Community 2022 » (**VSC**). Faites le téléchargement et l'installation dans le cas contraire :

Lien : https://visualstudio.microsoft.com/fr/vs/

c) Vous devez installer les modules suivants (Figure 1) :

- Développement .NET Desktop (Figure 2)
- Développement de jeux avec Unity (Figure 3)

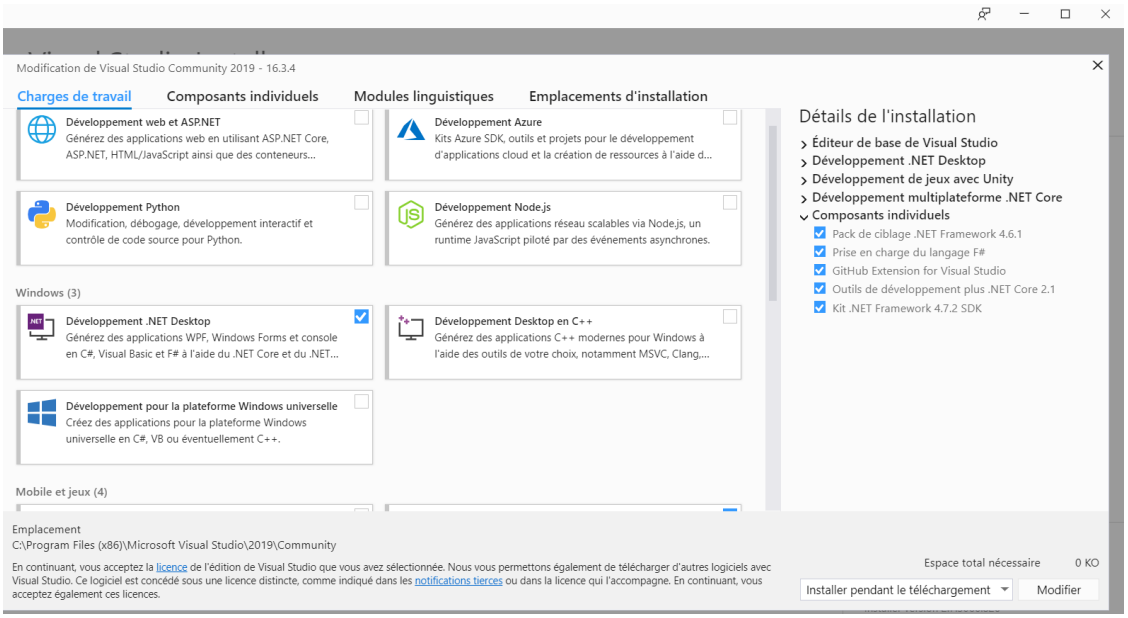


Figure 1. Interface pour l'installation des modules complémentaires sur VSC

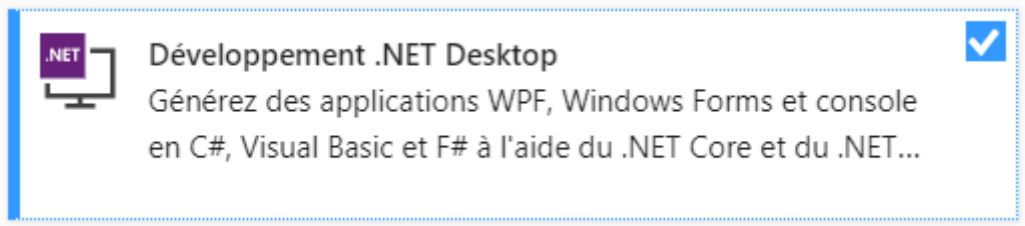


Figure 2. Module « Développement .NET Desktop »

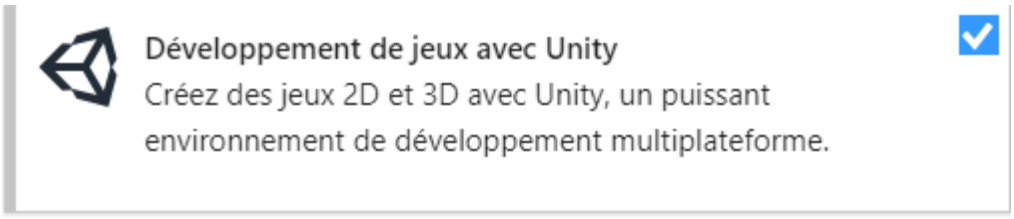


Figure 3. Module « Développement de jeux avec Unity

d) **Mon premier programme** : créez un projet de type « Application Console (**.NET Framework**) » et testez le code ci-dessus (Menu Déboguer / Exécuter sans débogage).

```
//Exemple : Hello World !
using System;
namespace TD1{
    class Program    {
        static void Main(string[] args)        {
            Console.WriteLine("Bonjour Madame/Monsieur!");
        }
    }
}
```

Date : 12/09/2023

Séance 2 – Programmation orientée objet

Objectifs :

- Rappel des concepts de la programmation orientée objet (POO) en langage C# : classes, attributs et méthodes
- Mise en pratique des concepts de POO pour le codage d'un jeu de tour par tour.

Concepts :

- **Classe** : ensemble des méthodes et des attributs qui modélisent une entité manipulée par une application. Une entité représente un élément spécifique d'une application. Par exemple, un personnage, un élément de la scène ou même des fonctionnalités pour assurer la connexion réseau d'un jeu.

Par exemple,

```
public class Personnage {
      //attributs
      //méthodes
}
```

- **Attributs** : données internes à une classe. Autrement dit, ce sont des variables définies dans le contexte (ou portée) global d'une classe. Par exemple,

```
private     int         vie;
public      string      nomObjet;
public      void        afficheContenu(){…}
private     void        calculerVitesse (){…}
```

- **Méthodes** : fonctions définies dans le contexte d'une classe.

```
public void marche (){  //méthode
            Console.WriteLine (« Je marche »)
}
```

- **Modificateur d'accès** : mots-clés utilisés pour permettre ou interdire l'accès externe aux attributs et méthodes d'une classe.
  - ✓ **private** : attributs et méthodes accessibles seulement aux méthodes de la classe.
  - ✓ **public** : attributs et méthodes également accessibles à des entités extérieures à la classe.
- **Méthodes de types « get » et « set »** : méthodes utilisées pour contrôler l'accès externe aux attributs d'une classe.
  - Type « get » : ce type de méthode est utilisé pour renvoyer la valeur d'un attribut d'une classe à une entité extérieure, comme la fonction « main » ou à une méthode d'une autre classe. Vous devez coder une méthode « get » par attribut.
  - Type « set » : il permet à une entité extérieure à classe de changer la valeur d'un attribut. Vous devez coder une méthode « set » par attribut.

Attention : l'utilisation des méthodes « **get/set »** pour l'accès aux attributs d'une classe est fortement conseillée à la place de l'utilisation du modificateur d'accès « public ».

Vue de l'ensemble :

```
public class Personnage {
private int vie;        //attribut
private double vitesse; //attribut
public  int id;         //attribut
      public Personnage () { //constructeur
            vie = 100 ;
            vitesse = 5 ;
      }
      public void marche (){  //méthode
            Console.WriteLine (« Je marche ») ;
      }
      public void arrete (){ //méthode
            Console.WriteLine (« Je m'arrete ») ;
      }
      public int getVie(){ //méthode de type « get »
            return vie ; }
      public void setVie(int nouvelleValeur){ //méthode de type « set »
            vie = nouvelleValeur ;
      }
}
```

- **Instance** : objet créé à partir d'un modèle défini par une classe. Nous pouvons considérer une instance comme une variable d'un « type » défini par une classe.

Pour créer une instance d'une classe, vous devez utiliser le mot-clé **new**. Par exemple,

```
Personnage monPerso = new Personnage();
```

- **Accès** aux **méthodes** et **attributs** d'une classe : pour utiliser (ou accéder) aux méthodes et attributs d'une classe, vous devez utiliser **l'opérateur « . »**. Par exemple,

```
static void Main(string[] args)
{
      Personnage monPerso     = new Personnage();
      //appel de la méthode « getVie() »
      int pointsVie           = monPerson.getVie(); /
      //accès à l'attribut Personnage.id
      Console.WriteLine(« Identifiant : »  + monPerson.id) ;
      Console.WriteLine(« Vie actuelle : » + pointsVie) ;
      ////appel de la méthode « setVie(int) »
      monPerso.setVie( 50 );
}
```

Exercices :

1. Identifiez les attributs et les variables :

```
public class Boss {
      public      int pointsAttaque;                  // ?
      private     int id;                             // ?
      string      nom ;                               // ?
      int         vitesse;                            // ?
      public Boss () {
            pointsAttaque     = 200 ;
            int vitesse       = 5 ;                   // ?
            string code       = nom + id ;            // ?
            Console.WriteLine(code) ;
      }
}
```

2. Créez un projet sur *Visual Studio* et ajoutez le code source ci-dessous. Ensuite, changez les modificateurs d'accès de la classe « Ennemi » pour la rendre compatible avec le code source défini sur la « boucle du jeu vidéo ». Utilisez des méthodes « get/set » pour donner accès aux attributs de la classe.

Classe « Ennemi »

```
public class Ennemi {
      public      int         pointsAttaque;
      public      double      vitesse;
      private     int         id;
      public      int         vie;
      public Ennemi() {
            vie = 100 ;
            vitesse = 5 ;
            pointsAttaque = 200 ;
      }
      private void attaque(){ //méthode
            Console.WriteLine(« J'attaque : - » + pointsAttaque) ;
      }
      private void defend (){ //méthode
            Console.WriteLine(« Je me défend ») ;
      }
      private void incrementerPointAttaque(){ //méthode
            Console.WriteLine(« Je prépare mon attaque ! ») ;
            pointsAttaque++;
      }
}
```

Boucle du jeu vidéo :

```
static void Main(string[] args){
      Personnage monPerso = new Personnage();
      Ennemi      ennemi1 = new Ennemi() ;
      while(monPerso.getVie() > 0) {
            monPerso.marche();
            ennemi1.attaque() ;
            Console.WriteLine("Vie personnage :"      + monPerso.getVie());
            Console.WriteLine("Vie ennemi :"          + ennemi1.vie);
            Console.WriteLine("L'ennemi attaque :" + ennemi1.pointsAttaque);
            monPerso.setVie(monPerso.getVie() -20);
            ennemi1.incrementerPointAttaque() ;
      }
      Console.WriteLine("C'est fini. Game Over!");
}
```

3. Les diagrammes de classes permettent d'avoir une vision globale de l'ensemble des classes qui modélisent une application, leurs attributs (-), leurs méthodes (+) et leurs relations. Le diagramme ci-dessous représente les classes : *ennemi*, *personnage* et *allié*.

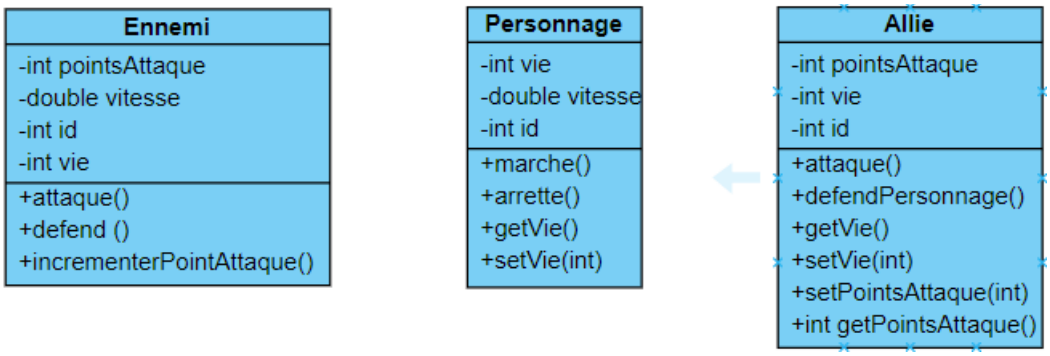


Figure 1. Diagrammes des classes du jeu exemple

- Ajoutez à votre projet les classes « Personnage » et « Allie » avec les méthodes et attributs définis dans le diagramme ci-dessus. Vous pouvez utiliser le code source des fonctions de même nom dans les classes « Ennemi » et « Personnage » comme exemple pour la nouvelle classe.
- Ajoutez une instance de la classe « Allie » au code de la boucle du jeu vidéo.
- Montrez le fonctionnement de la nouvelle classe par un appel à l'une de ses méthodes dans la boucle du jeu vidéo.

4. **Relier les points** : maintenant, nous allons faire interagir les objets créés à partir des classes : *Ennemi*, *Personnage* et *Allié*. L'objectif du jeu est que le personnage survive un nombre fini de tours du jeu. Le nombre de tours, N, sera un paramètre du jeu.
   1. Assurez-vous que votre jeu crée une instance de chaque classe.
   2. Codez la mécanique suivante pour le jeu :
      a) À chaque tour de boucle, le jeu demandera l'action à réaliser à l'utilisateur.
         - ✓ Touche « H » : soigne le personnage, mais subit les dégâts de l'attaque de l'ennemi au même temps.
         - ✓ Touche « Z » : soigne l'allié et subis les dégâts de l'attaque de l'ennemi.

- ✓ Touche « D » : demande à l'allié de faire le bouclier et de prendre les dommages causés par l'ennemi à la place du personnage. Le personnage se soigne eu même temps.

Attention, une fois que les points de vie de l'allié sont finis, cette action n'est plus disponible.

- ✓ Touche « A » : demande à l'allié d'attaquer l'ennemi. Le personnage subit les dommages infligés par l'ennemi.
- ✓ Autre touche : le personnage subit les dommages.

Astuce : la fonction « Console.ReadLine() » capture la touche appuyée par l'utilisateur.

b) Ensuite, l'ennemi essayera d'attaquer le personnage et le succès de son attaque dépendra de l'action choisie par l'utilisateur.
c) À la fin du tour, le jeu affichera les points de vie et d'attaque des éléments du jeu (instances de classes : ennemi, allié et personnage)
d) L'utilisateur gagne lorsque l'ennemi est vaincu ou que le nombre de tours est atteint. L'utilisateur perd une fois que la vie du personnage atteint zéro.

Perfectionnement :

5. Révisez votre projet et faites le nécessaire pour que la communication (ou liaison) entre les instances de classes soit faite par des méthodes afin d'éviter de donner accès direct aux valeurs des attributs d'un objet.
6. Paramétrez la valeur de **N** et les attributs des classes de façon à avoir un jeu cohérent. Par exemple :

Ennemi	Allié	Personnage
vie : 100	vie : 75	vie : 75
dommages d'attaque : 25	dommages d'attaque : 10	capacité de se soigner : 15

7. Pour les plus expérimentés, ajoutez une touche de hasard au jeu.
   - Pour chaque attaque (ennemi ou alliée), tirez un nombre au hasard pour déterminer si l'attaque réussit. Par exemple, si le nombre tiré est pair, les dommages seront égaux au maximum des points d'attaque de l'attaquant et zéro sinon.
   - Utilisez le nombre tiré au hasard (entre 0 et 100) comme base pour calculer les dommages infligés par une attaque. Par exemple, si le nombre 10 est sorti, l'attaquant causera des dommages équivalant à 10% de ses points d'attaque.

Astuce : Les commentaires nous permettent de décrire l'objectif et le fonctionnement de notre code. Un code source bien commentés facilite sa maintenance future. Deux options existent pour commenter votre programme :

a) /* bloc de texte en plusieurs lignes */

b) // texte sur une seule ligne.

Utilisez des commentaires pour décrire l'objectif et les paramètres de votre programme dans le code source.

Résumé :

À l'issue de cette séance, vous devez maitriser les notions suivantes :

- Les concepts de classe, méthode, attribut et instance.

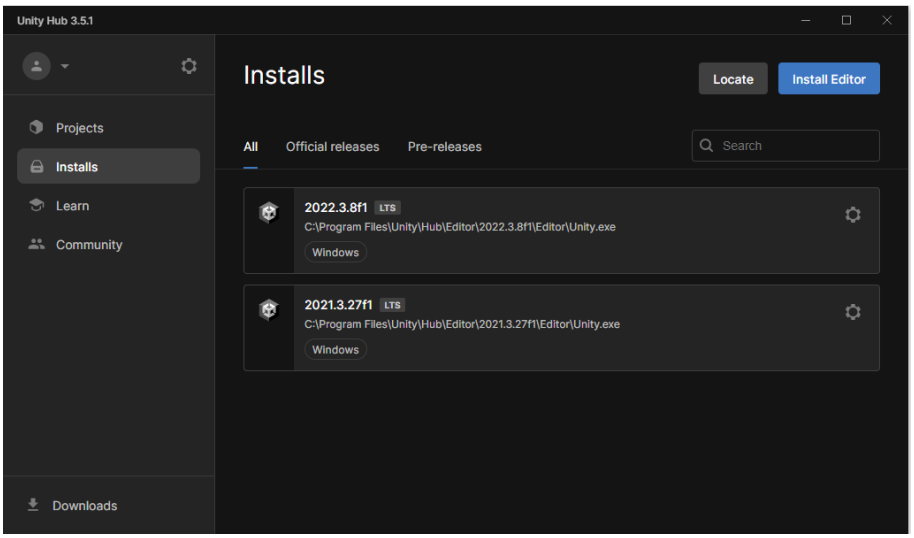
- L'enjeu derrière l'interdiction/permission d'accès aux attributs et méthodes d'une classe et l'intérêt d'ajouter des méthodes de type « get » et « set » à une classe.
- Codage d'une classe à partir d'un diagramme.
- Création d'une application où des instances de différentes classes interagissent en utilisant leurs méthodes.

Date : 21/19/2023

Séance 3 – Classes sur Unity

Avant de commencer :

- Téléchargez « Unity Hub » : https://unity.com/download
- Utilisez le bouton « Install Editor » pour télécharger la dernière version LTS d'Unity



Objectifs :

- Apprendre à utiliser de classes prédéfinies des bibliothèques Unity
- Création de nouvelles classes sur Unity

Concepts :

Classes sur Unity : le moteur de jeu Unity propose un ensemble de classes pour faciliter la modélisation des jeux vidéo. Ces classes représentent des composants d'un jeu vidéo comme des objets 2D, des objets 3D, des effets spéciaux, de l'audio, *etc*.

- **Classe « GameObject »** : c'est la classe utilisée pour modéliser des éléments de la scène d'un jeu vidéo. Une bonne partie des éléments que nous utiliserons pour modéliser nos jeux sur Unity sont des classes de type « GameObject ». Vous trouverez sur le lien ci-dessous une description détaillée des attributs et méthodes fournis par cette classe. Lien : https://docs.unity3d.com/ScriptReference/GameObject.html
- **Classe « Transform »** : cette classe permet de stocker et manipuler des informations relatives à la position et au déplacement des objets dans la scène, comme les attributs de « position », « rotation » et « scale », en français : position, rotation et échelle.
- **Classe « MeshRender »** : cette classe offre des attributs et méthodes pour contrôler le rendu des éléments de votre jeu vidéo.

Script comportemental : code en langage C# utilisé pour ajouter des comportements (ou des fonctionnalités) à une instance d'un *GameObject*. En pratique, lorsqu'on définit un script sur Unity, on est en train de définir une nouvelle classe. Nous sommes libres pour ajouter les méthodes et les attributs que nous voulons sur ces classes.

Néanmoins, le moteur de jeu Unity propose un modèle de script (ou classe) qui nous permet de profiter des comportements prédéfinis pour les éléments du jeu.

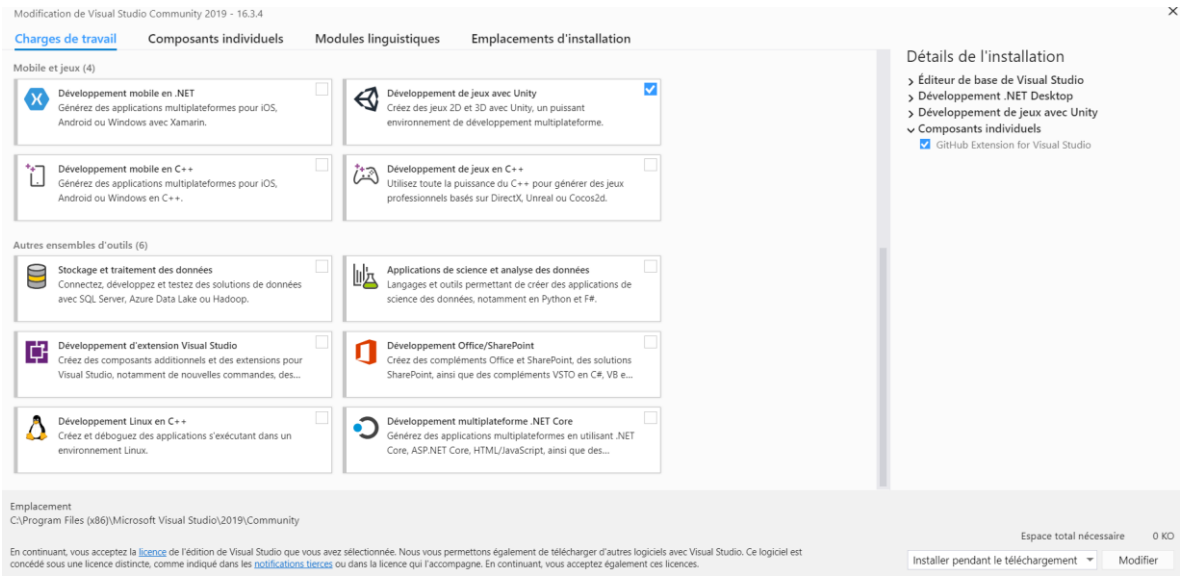
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class MonScript : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {    }
    // Update is called once per frame
    void Update()
    {    }
}
```

Par exemple, lorsqu'on définit les méthodes « Start » et « Update » avec les signatures proposées par le modèle, on a les comportements suivants :

- La méthode « Start » est appelée automatiquement par Unity avant l'exécution de la première image (« frame ») du jeu vidéo. Elle peut être utilisée pour activer certaines fonctionnalités des éléments d'un jeu juste avant le démarrage du jeu.
- La méthode « Update » est appelée à chaque image de votre jeu vidéo. Cette méthode est donc utile pour toutes les mises à jour ordinaires de votre jeu, comme :
    - ✓ Déplacement des objets non physiques.
    - ✓ Minuteur/chronomètre simples
    - ✓ Traitement des interactions avec l'utilisateur.

Visual Studio pour Unity

Avant de commencer les exercices, vérifiez que vous avez installé le module « Développement de jeux avec Unity » sur Visual Studio Community. Si ce n'est pas le cas, vous devez lancez « Visual Studio Installer » pour installer ce module.



Démonstration

- Création d'un projet
- Ajout d'un objet de type « GameObject » dans la scène
- Modification des composants d'un objet (« Transform », « MeshRenderer »).
- Création d'un script
- Visualisation des attributs d'un script sur « Inspector »
- Fonction « Debug.Log »

Comment ajouter un script sur un objet ?

Plusieurs options sont possibles :

a) Clic droit sur la zone de l'onglet « Asset » sous « projet ». Choisissez l'option « Create » et ensuite la sous-option « C# script ». Ensuite, sélectionnez le fichier créé et déplacez-le sur l'objet.
b) Sélectionnez l'objet à intégrer dans le script. Cliquez sur le bouton « Add Component » dans l'onglet « Inspector » et choisissez l'option « New script ».
c) Cliquez sur l'objet à intégrer dans le script et ensuite sur le menu « Component ». Sélectionnez soit l'option « script » pour créer un nouveau script ou le nom d'un script existant sur la liste.

Exercices :

1. **Utilisation des « assets »** : Maintenant, nous utiliserons des ressources préexistantes pour construire le jeu, des « assets ». Téléchargez les matériels du cours « Create with code » de Unity Learn.

Lien : https://connect-prd-cdn.unity.com/20191004/11d568c2-0b8a-4eca-9219-52d05a07eba1/Prototype%201%20-%20Direct%20Download.zip?_ga=2.261656845.1451003127.1571749928-1045493195.1570353603

- Décompressez l'archive téléchargée. Ensuite, allez sur le menu « Asset » d'Unity, et choisissez l'option « Import Package ». Chargez le fichier « Prototype-1_Starter-Files.unitypackage »
- Placez les assets « Prototype 1 » ou « Ground road » et « Veh_Ute_Red_Z » sur la scène. Ils se trouvent dans la fenêtre « Assets ». Ensuite, renommez ces assets avec des noms plus pertinents. Par exemple, voiture et environnement.
- Lancez le jeu pour valider l'exercice.

2. Cet exercice a pour objectif d'ajouter du mouvement à l'objet voiture. Le contrôle du mouvement des objets sur Unity est normalement fait à partir du composant « Transform ». Mais, comment y accéder ?

- **Instance « gameObject »** : dans tous les scripts Unity, un raccourci vers l'objet « GameObject » sur lequel le script C# est rattaché est automatiquement créé. Ce raccourci, nommé « gameObject » (attention à la première lettre en minuscule), donnes un accès direct aux attributs et méthodes de l'instance du GameObject en question, ainsi qu'aux composants de l'objet.

La construction suivante permet de récupérer l'attribut « position » de l'objet :

```
gameObject.transform.position
```

- La position des objets dans un jeu 3D sur Unity est représentée par un vecteur à trois dimensions : x, y, z. Le code suivant incrémente de « 1 » la position de l'objet sur l'axe y.

```
Vector3 vecteur = new Vector3(1,0,0);
gameObject.transform.position+= vecteur ;
```

- Ajoutez un script, nommé « déplacement », sur l'objet « voiture » et intégrez le code ci-dessus à la méthode que vous jugerez la plus pertinente pour déplacer la voiture. Testez le fonctionnement du code !
- Maintenant, essayez de modifier le code ci-dessus pour déplacer la voiture, automatiquement, dans l'axe de la route.

3. **Time.deltaTime** : vous avez peut-être remarqué que le déplacement de la voiture s'est fait trop vite ou trop lentement, selon la vitesse de votre ordinateur. Multipliez la mise à jour du vecteur « position » par « Time.deltaTime ». Cet attribut de la classe Time permet de contrôler la mise à jour de paramètres du jeu selon la vitesse de l'ordinateur. Est-ce que le déplacement vous semble plus naturel maintenant ?
4. **Réglage de la vitesse** : le mouvement de la voiture est actuellement constant. Ajoutez un attribut à votre script pour contrôler la vitesse de la voiture. Cet attribut doit être intégré au calcul de la mise à jour de la vitesse de la voiture et il doit être modifiable via « Inspector ». Testez différentes valeurs pour l'attribut « vitesse » pendant l'exécution de notre jeu pour valider votre code.
5. **Utilisation de classes** : créez un nouveau script et nommez-le « GestionVoiture ». Ce script doit contenir la classe décrite ci-dessous qui sert à calculer la consommation d'essence de la voiture et à signaler son arrêt dès que le réservoir est vide.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class GestionVoiture
{
    private double essence;

    public GestionVoiture() { //Constructeur
        essence = 10;
        Debug.Log("Essence :" + essence);
    }
    public double getEssence() {
        return essence;
    }
    public void setEssence(double valeur) {
        essence = valeur;
    }
    public bool roule(double consommation) { //à compléter }
}
```

- Complétez la méthode « roule ». Elle doit décompter la valeur du paramètre « consommation » de l'attribut « essence » et évaluer l'état du réservoir d'essence. Si

la valeur de l'attribut « essence » est supérieure à zéro, la méthode renvoie « true » et « false » dans le cas contraire (réservoir vide).

- Ajoutez une instance de la classe « GestionVoiture » au script responsable du déplacement de la voiture. Ce script doit arrêter le déplacement de la voiture dès que la méthode « roule » signale que le réservoir d'essence est vide.

Perfectionnement

6. Pour les plus expérimentés, ajoutez à votre voiture les fonctionnalités suivantes liées à la conduite de la voiture :
   - Accélérer ou décélérer la voiture en utilisant les touches Z et S.
   - Tourner à droite et à gauche en utilisant les touches Q et D.

Vous pouvez utiliser le code source ci-dessous pour capturer l'interaction du joueur avec les flèches du clavier.

```
if (Input.GetKeyUp(KeyCode.LeftArrow))    {…}
else if (Input.GetKeyUp (KeyCode.RightArrow)){…}
else if (Input.GetKeyUp (KeyCode.UpArrow))    {…}
else if (Input.GetKeyUp (KeyCode.DownArrow)) {…}
```

7. Rajoutez des véhicules contrôlés par le jeu à votre projet. Ces véhicules doivent parcourir la route en sens inverse.
8. Rajoutez des obstacles sur la route, telles que des boîtes. Utilisez le composant de type « RigidBody » aux obstacles afin de rajouter des fonctionnalités liées à la physique de collisions à votre jeu.

Date : 13/09/2023

Séance 4 – Calcul vectoriel

Précédemment dans les séances 2 et 3 :

Pendant les séances précédentes vous étiez amené à coder le déplacement d'un objet sur Unity. Votre script doit probablement ressembler au code source ci-dessous. Le code ci-dessous déplace l'objet d'intérêt suivant l'axe Z avec une vitesse de 4.5 mètres. La direction de déplacement est définie par un vecteur de type « Vector3 ». Dans notre exemple, nous avons utilisé la méthode « Time.deltaTime » pour considérer le temps passé entre l'appel précédent et l'appel actuel à la méthode « Update » dans le calcul de la nouvelle position de l'objet d'intérêt.

```
public class deplacement : MonoBehaviour {
 public float vitesse = 4.5f;
 void Update(){
     gameObject.transform.position += new Vector3(0,0,1)*vitesse*Time.deltaTime;
 }
}
```

Objectifs :

- Rappel des opérations sur des vecteurs
- Introduction aux classes Vector2 et Vector3
- Mise en pratique des concepts appris

Concepts :

Le moteur de jeu Unity utilise des vecteurs pour réaliser différentes tâches liées à la mise en place et le déroulement d'un jeu vidéo, tels que le calcul de la position des objets ou de la direction et de la vitesse de leur mouvement. La Figure 1 montre les repères Unity pour des jeux en 2D et 3D.

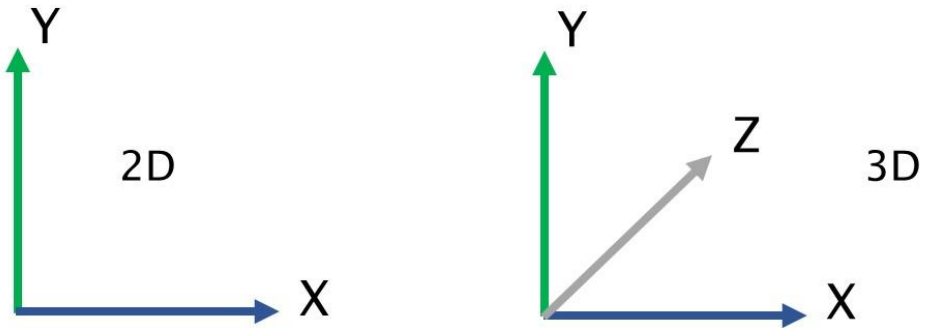


Figure 1. Repères Unity en 2D (gauche) et 3D (droite)

Classes « Vector2 » et « Vector3 » : la bibliothèque Unity met à notre disposition des classes pour calculer la majorité des opérations liées à la manipulation des vecteurs. Comme ces noms le suggèrent, la classe *Vector2* est utilisée pour représenter des vecteurs en 2 dimensions, < x, y >, et la classe *Vector3* pour des vecteurs en 3D dimensions, < x, y, z >.

Pour créer un vecteur sur Unity, vous devez utiliser l'opérateur « **new** ». L'exemple ci-dessous montre la création des variables pour stocker les vecteurs < 3,5 > et < 2,-1 >.

```
Vector2 v2D_a = new Vector2 (3, 5); // vecteur 2D
Vector2 v2D_b = new Vector2 (2, -1); // vecteur 2D
```

Pour extraire les coordonnées d'un vecteur, nous utilisons les attributs « x » et « y » pour un vecteur 2D et « x », « y » et « z » pour les vecteurs 3D.

```
Debug.Log(v2D_a.x);
Debug.Log(v2D_a.y);
```

L'addition entre deux vecteurs crée un nouveau vecteur où chaque dimension est la somme des dimensions correspondent des vecteurs sommés. Considérons l'exemple suivant :

```
a = <3,5>
b = <2,-1>
c = a + b
c = <3+2, 5+(-1)>
c = <5, 4> //résultat
```

La Figure 2 illustre l'addition des vecteurs « a » et « b », en noir, et leur résultat en gris, le vecteur « c ».

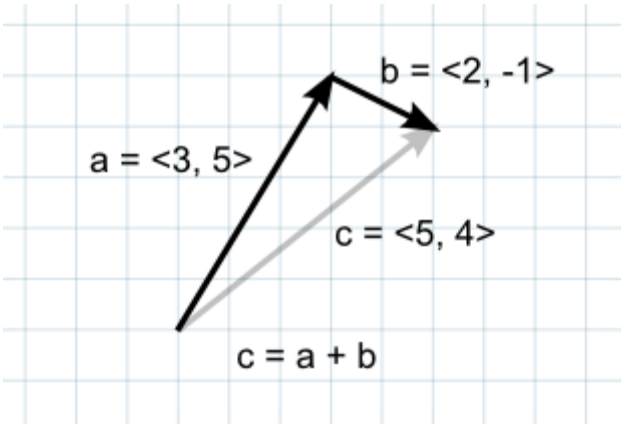


Figure 2. Sommes de vecteurs. Source : Unity.com

Les opérations d'addition et de soustraction entre deux variables de type vecteur sur Unity sont faites avec les opérateurs « + » et « - ». Vous trouverez ci-dessous un exemple d'addition.

```
Vector2 v2D_c;
v2D_c = v2D_a + v2D_b;
Debug.Log("Résultat: " + v2D_c);
//ou
Debug.Log("Résultat (v2).x : " + v2D_c.x +", .y :"+ v2D_c.y)
```

Déplacement des objets : il y a plusieurs façons de déplacer un objet sur Unity. Précédemment, nous avons directement modifié l'attribut « position » de l'objet. Une alternative, c'est d'utiliser la méthode « Transform.Translate() ». L'exemple ci-dessous produit un déplacement équivalant à celui produit par la solution précédente, mais en utilisant la méthode « Translate() ».

Exemple d'utilisation de la méthode « Translate »

```
transform.Translate ( new Vector3(0,0,1) * Time.deltaTime * vitesse );
```

Jusqu'à présent, nous avons créé un vecteur pour représenter la direction que nous voulions donner au mouvement de notre objet. Néanmoins, les classes de type vecteur sur Unity proposent plusieurs raccourcis pour faciliter le déplacement des objets dans la scène par

rapport au repère du monde. Le Tableau 1 montre quelques raccourcis proposés par la classe « Vector3 » :

Tableau 1. Raccourcis de direction proposés par la classe « Vector3 »

Raccourci	Équivalent à (x, y, z)
Vector3.back	Vector3(0, 0, -1).
Vector3.down	Vector3(0, -1, 0).
Vector3.forward	Vector3(0, 0, 1).
Vector3.left	Vector3(-1, 0, 0).
Vector3.right	Vector3(1, 0, 0).
Vector3.up	Vector3(0, 1, 0).
Vector3.zero	Vector3(0, 0, 0).
Vector3.one	Vector3(1, 1, 1).

Nous pouvons donc réécrire les exemples précédents de déplacement avec le raccourci « Vector3.forward » pour déplacer l'objet d'intérêt vers l'avant.

Exemple 1

```
transform.Translate ( Vector3.forward * Time.deltaTime * vitesse );
```

Exemple 2

```
gameObject.transform.position+= Vector3.forward * vitesse * Time.deltaTime;
```

De façon similaire, les raccourcis « Vector3.up », « Vector3.down », « Vector3.left » et « Vector3.right » définissent le déplacement de l'objet vers le haut, vers le bas, vers la gauche et vers la droite de la scène, respectivement.

Rotation des objets : nous pouvons également faire tourner un objet autour d'un axe à l'aide de la méthode « Transform.Rotate() ». Cette méthode prend comme arguments l'axe à utiliser pour la rotation (sous la forme d'un vecteur) et la vitesse de rotation.

```
transform.Rotate (Vector3.up, vitesseRotation * Time.deltaTime);
```

Repère de l'objet : les raccourcis présentés ci-dessus se basent sur le repère du monde. Mais, il est parfois nécessaire d'utiliser le repère de l'objet. Par exemple, lorsque nous voulons faire en sorte que l'objet se déplace en fonction de son orientation. Autrement dit, qu'il avance vers la direction qu'il pointe. Dans ce cas, nous devons utiliser les raccourcis proposés par le composant « Transform » de l'objet d'intérêt : « *.forward* », « *.right* » et « *.up* ».

Le moteur de jeu Unity propose aussi des méthodes pour nous aider à réaliser des fonctionnalités courantes d'un jeu vidéo. Dans ce cadre, vous pouvez utiliser deux méthodes importantes telles que « MoveTowards » et « LookAt ».

Méthode « Vector3.MoveTowards » : cette méthode calcule le vecteur pour déplacer un objet A vers un objet B. Son résultat peut être ensuite utilisé pour calculer la nouvelle position de l'objet à déplacer.

Exemple d'utilisation de la méthode « MoveTowards(Vector3,Vector3, float) » :

```
Vector3.MoveTowards(objetDInteret.transform.position, gameObject.transform.position, 20);
```

Méthode « LookAt ». La classe « Transform » met à notre disposition la méthode « LookAt » qui tourne l'objet courant vers l'objet d'intérêt. Pour l'utiliser, nous devons appeler la méthode « LookAt » à partir du composant « transform » de l'objet qui observe. Cette méthode prend comme argument le composant « transform » de l'objet à regarder.

Exemple d'utilisation de la méthode « Transform.LookAt » :

```
void update(){
      //transform : composant « Transform » de l'objet qui       regarde
      //objet_d_interet : instance de type GameObject de l'objet à regarder
      transform.LookAt(objet_d_interet.transform);
}
```

Exercices

Calcul des vecteurs

1. Calculez à la main les opérations entre vecteurs ci-dessous :
   a) Addition entre vecteurs 2D :

```
Position actuelle : <+5,+8>
Vitesse : <+3,+2>
```

   b) Soustraction des vecteurs en 2D :

```
Position : <-1,-3>
Déplacement : <-2,+2>
```

   c) Addition entre vecteurs 3D

```
Position actuelle : <-2,-1,+5>
Vitesse : <+1,+4,+3>
```

   d) Soustraction des vecteurs en 3D :

```
Position : <2,-4,+1>
Déplacement : <-1,-1,+3>
```

Vecteurs sur Unity

2. Créez un nouveau projet 3D sur Unity. Créez un script comportemental et utilisez les classes « Vector2 » et « Vector3 » pour coder les opérations entre vecteurs ci-dessus et valider vos résultats. Les résultats des opérations doivent être stockés eux aussi sous la forme de variables de type vecteur. Votre code doit être placé dans la méthode « Start ». Créez une instance de GameObject « Empty » pour attacher votre script et le tester.
3. Ensuite, ajoutez un « GameObject » de type « Cube » sur votre scène et placez-le sur la position <0,0,0>. Modifiez sa taille (scale) pour l'allonger selon l'axe Z, par exemple (1, 1, 10). Faites-le avancer de 3 unités par seconde. N'oubliez pas de la méthode « Time.deltaTime » sur le calcul du déplacement de votre jeu. Testez votre jeu !
4. Ensuite, sélectionnez la caméra du jeu (objet « Maint Camera ») et paramétrez-la avec les valeurs ci-dessous. Vous devez retrouver un résultat similaire à la Figure 3.
   - Position : <0,50,0>
   - Rotation : <90,0,0>

- Projection : Perspective
- Field of View : 120 degrés

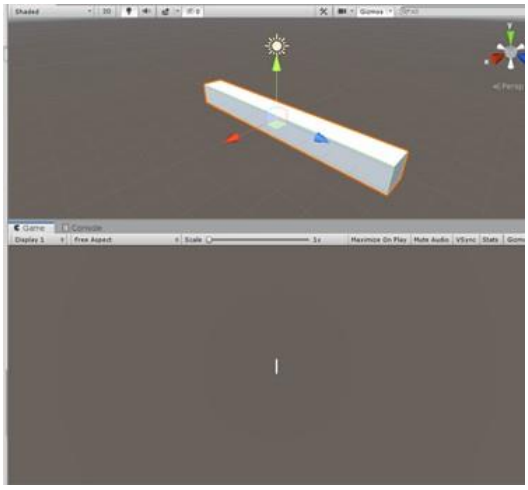


Figure 3. Résultat attendu

5. Commentez votre code pour déplacer l'objet. Maintenant, faites tourner votre objet autour de lui même de 45 dégrées à chaque « frame ». Testez votre code !
6. Finalement, combinez les deux fonctionnalités précédentes (Translate et Rotate) pour faire l'objet tourner en cercle autour de la scène.

Projet voiture

Reprenez votre projet Unity des séances 3 et 4 pour les exercices à suivre. Assurez-vous que la voiture se déplace automatiquement à chaque « frame ». La vitesse doit être un attribut du script qui déplace la voiture.

7. **Placement de la caméra :** sélectionnez la caméra du jeu (« main camera ») et placez-la à côté de la voiture du joueur. Utilisez les outils « Move tool » et « Rotate tool » pour obtenir un résultat qui ressemble à l'image ci-dessous.

Figure 4. Outils pour la manipulation des GameObject

Figure 5. Position de la caméra

8. **Regarder l'objet d'intérêt** : dans certains jeux, il est souvent nécessaire que la caméra suive le personnage du jeu. Pour accomplir cette tâche, ajoutez un script à caméra du jeu, nommez-le « regard.cs ». Ensuite, créez un attribut, « objet_cible » dans ce script pour stocker une référence vers le « GameObject » à suivre par la caméra. Utilisez les méthodes vues pour tourner la caméra vers la voiture dès que la voiture se déplace (voir Figure 6).

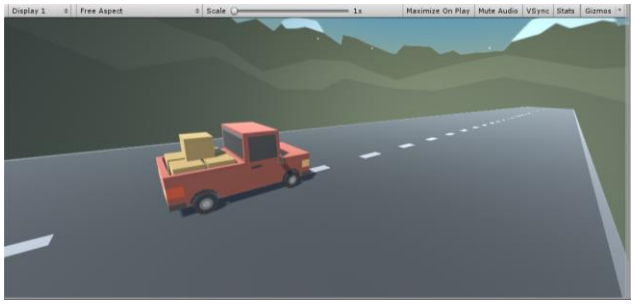


Figure 6. Retournement de la caméra

Pour lier l'objet « voiture » au script « regard.cs », cliquez sur l'objet « camera » de votre projet (sous le volet « Hierarchy »), puis déplacez l'objet « voiture » (aussi sous le volet « Hierarchy ») vers l'attribut « objet cible » du script, qui est visible sous le volet « Inspector ».

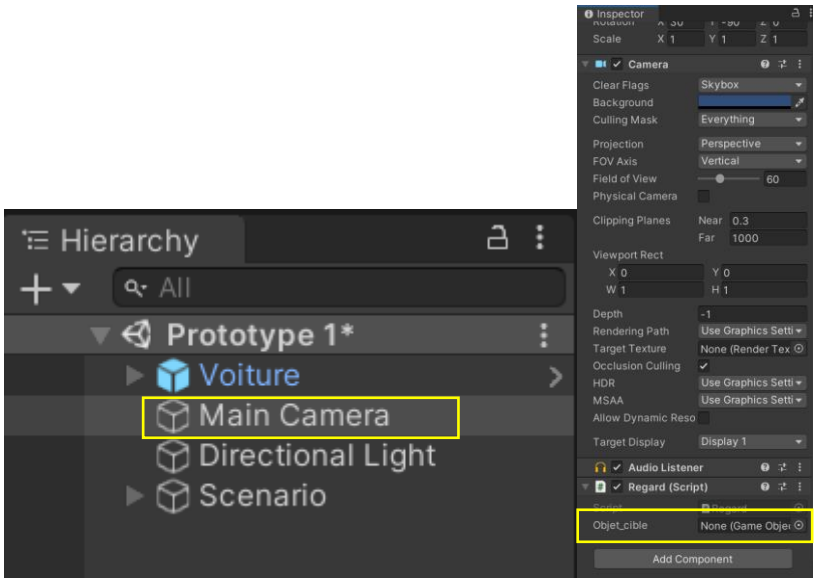


Figure 7. Volets « Hierarchy » et « Inspector »

9. **Une caméra qui accompagne le véhicule**. Faites que la caméra du jeu se déplace avec la voiture une fois que la voiture s'éloigne trop de son champ de vue. Utilisez les méthodes vues dans cette séance et l'attribut « objet_cible » pour accomplir cet exercice. L'attribut « objet_cible » doit être paramétrable via l'Inspector.

   Testez votre code avec la voiture du joueur.

Défis : faites que la caméra se déplace avec la voiture mais continue de lui être parallèle.

10. **Contrôle de la voiture** : ajoutez un script, « Deplacement.cs », pour contrôler le mouvement de la voiture. Les contrôles doivent permettre d'accélérer ou décélérer la voiture (0 à 50 m/s, méthode « Translate ») ou de la tourner vers la gauche ou la droite

(méthode « Rotate »). Vous pouvez utiliser le code source ci-dessous pour capturer l'interaction du joueur avec les flèches du clavier.

```
if    (Input.GetKeyUp(KeyCode.LeftArrow))    {…}
else if (Input.GetKeyUp (KeyCode.RightArrow)){…}
else if (Input.GetKeyUp (KeyCode.UpArrow))    {…}
else if (Input.GetKeyUp (KeyCode.DownArrow)) {…}
```

Perfectionnement :

11. **Des roues qui roulent :** Pour l'instant, la voiture contrôlée se déplace, mais ses roues ne bougent pas. Écrivez un script pour faire tourner une roue. Ensuite, appliquez ce script à l'une des roues de la voiture pour tester votre code. Une fois réussi, appliquez le script aux autres roues. N'oubliez pas de modérer la vitesse des rotations par les temps passés à travers les appels à la méthode « Update ».
12. **Vitesse partagée** : Faites en sorte que lorsque nous accélérons/décèlerons la voiture, les roues adoptent une vitesse proportionnelle à celle de la voiture.
13. **Défis II** : Faites en sorte que les roues avant tournent une fois que nous tournons la voiture vers la gauche ou la droite.

Date : 21/09/2023

Séance 5 – Cycle de vie d'un objet

Objectifs :

- Introduire les notions de cycle de vie d'un objet
- Mise en pratique des concepts décrits sur le moteur de jeu Unity

Concepts :

a) **Le cycle de vie d'un objet** (ou instance) d'une classe peut se diviser en trois étapes : sa création (ou instanciation), son utilisation et sa suppression (ou destruction). Le paradigme de programmation orientée objet définit des méthodes pour gérer les étapes des construction et destruction des objets, respectivement, les méthodes *constructeur* et *destructeur*.

Constructeur : cette méthode sert à paramétrer l'objet d'une classe avant son utilisation. Il est appelé une seule fois lors de l'instanciation de l'objet. Une classe peut avoir plusieurs constructeurs. Le constructeur par défaut ne prend pas d'arguments. Cependant, nous pouvons déclarer autant des constructeurs que nous en avons besoin, tant que chaque constructeur prend comme argument un ensemble de paramètres uniques par rapport aux autres constructeurs de la même classe.

Exemple de constructeur :

```
public class CoffreVoiture
{
    private float capacite;
//définition du constructeur par défaut
    public CoffreVoiture() {
        Debug.Log("Constructeur appelé");
        capacite = 0;
    }
//définition du constructeur alternatif
    public CoffreVoiture(float capaciteCoffre)
    {
        capacite = capaciteCoffre;
        Debug.Log("Constructeur appelé avec : "+capaciteCoffre+" litres");
    }
}
```

Déclaration d'objet avec appel au constructeur par défaut :

```
CoffreVoiture CoffreVoiture = new CoffreVoiture() ;
```

Déclaration d'objet avec appel au constructeur alternatif :

```
CoffreVoiture CoffreVoitureVoiture = new CoffreVoiture(5f) ;
```

Néanmoins, lorsque nous écrivons des scripts comportementaux Unity (MonoBehaviour), nous devons adopter les méthodes **« Awake »** et/ou **« Start »** pour la paramétrisation d'un objet **à la place** du **constructeur**. Ces méthodes seront aussi appelées automatiquement quand un objet sera instancié et **une seule fois** chacune dans leur cycle de vie.

La méthode « **Awake** » est couramment utilisée pour établir des références internes à un objet, comme les valeurs des attributs et des liens vers des objets extérieurs à la classe.

La méthode **« Start »** est lancée après la méthode « **Awake** » et juste avant l'exécution de la première « frame » du jeu. Elle sera donc utile pour le paramétrage de votre objet aussi. En revanche, la méthode « **Start** » est exécutée seulement si l'attribut booléen « *enabled* » du script comportemental vaut « *true* ». Notons que les méthodes « **Awake** » de tous les objets de la scène seront lancées avant que la première méthode « **Start** » ne soit appelée pour un objet.

À titre d'exemple d'utilisation des méthodes « **Awake** » et « **Start** », nous pouvons étudier le cas des jeux de tir. Dans ce contexte-là, la méthode « **Awake** » pourra être utilisée pour ajouter le joueur au jeu et approvisionner son inventaire, tant que la méthode « **Start** » sera utilisée pour lui accorder la permission de tirer sur ses ennemis.

Destructeur : cette méthode, aussi connu sous le nom de « finaliser », nous permet de faire les derniers traitements avant que l'objet ne soit détruit. Elle est lancée automatiquement par le « Coffre Voiture » de jeu lorsqu'un objet est signalé pour la destruction. Elle doit être utilisée essentiellement pour faire les derniers nettoyages relatifs à l'objet détruit, comme par exemple, signaler la destruction de ses sous-objets. Le destructeur d'une classe porte le même nom que la classe, mais avec le préfix « ~ ».

Exemple de déclaration d'un destructeur.

```
public class CoffreVoiture
~CoffreVoiture(){ //définition du destructeur
        Debug.Log("Destructeur appelé");
    }
}
```

De la même façon que pour le constructeur, un script « **MonoBehaviour** » doit utiliser une méthode alternative pour la destruction de l'objet : la méthode « **OnDestroy**() ». L'exemple ci-dessous montre une version alternative de la classe « CoffreVoiture » qui utilise les méthodes proposées par « MonoBehaviour » pour initialiser et détruire l'objet.

```
public class CoffreVoitureMB : MonoBehaviour {
    void Awake(){ }//constructeur – partie 1
    void Start(){ }//constructeur – partie 2
    void Update() { }
    void OnDestroy(){}//destructeur
}
```

Comme dans le framework .NET, le moteur de jeu Unity utilise une entité appelée « Garbage Collector » (GB) pour gérer l'allocation et la libération de mémoire de votre application. Chaque fois que vous créez un objet, la CLR alloue de la mémoire pour l'objet. Toutefois, la quantité de mémoire à notre disposition n'est pas illimitée. Le GB est donc exécuté périodiquement pour libérer de la mémoire allouée qui n'est plus utilisée. Une mémoire est considérée prête à être récupérée (ou recyclée) lorsqu'aucune variable ne fait plus référence à elle.

Nous pouvons utiliser la méthode « Destroy(objet) » pour demander explicitement la destruction des objets de types proposés par la bibliothèque Unity, tels que « Object », « GameObject » et « MonoBehavior », entre autres.

Exemple de destruction des objets - Type GameObject et Object

```
CoffreVoitureMB coffreVoitureMB1 = new CoffreVoiture();//création
Destroy(coffreVoitureMB1);//destruction
```

Néanmoins, la méthode « Destroy » ne convient pas à des objets créés à partir de classes standard (qui ne suivent pas le modèle « MonoBehaviour »). De plus, nous n'avons pas à cet instant une méthode pour demander explicitement la destruction de ces objets.

Les bonnes pratiques nous conseillent d'affecter la valeur « null » à un attribut (ou à une variable) qui fait référence à un objet dès qu'elle n'utilise plus cet objet. Une fois qu'aucun attribut (ou variable) ne fait référence à l'objet, le GB se chargera automatiquement de lancer sa destruction.

Exemple de destruction - Classe standard

```
CoffreVoiture = null;
```

Notons que c'est l'entité GB qui détermine le moment propice pour lancer une opération de collecte et de nettoyage de mémoire et que cette action peut avoir des conséquences sur la performance de votre jeu. Nous verrons plus tard des techniques utilisées pour bien gérer l'utilisation de mémoire dans un jeu.

Exercice 1 :

Pour les exercices suivants, créez un projet de jeu 3D sur Unity. Ensuite, ajoutez un objet de type « GameObject Empty » à votre scène avec un script comportemental nommé « Gestionnaire Jeu ».

a) **Constructeurs et destructeurs standard** :

1. Créez une nouvelle classe standard, nommé « Coffre Voiture », avec un constructeur par défaut et un constructeur alternatif. Ces constructeurs doivent afficher les messages suivants : « Le coffre de la voiture a été créé » et « Coffre de la voiture a été créé avec l'argument suivant : X ».
2. Créez des instances de cette classe dans le script « Gestionnaire Jeu » pour pratiquer l'utilisation des constructeurs par défaut et alternatif. Ces instances doivent être codées comme des attributs du script.
3. Ajoutez une méthode de type destructeur à la classe « Coffre Voiture » avec le message suivant : « le coffre de la voiture sera maintenant détruit! ».
4. Faites en sorte que les instances que vous avez créées de cette classe soient détruites lorsqu'on appuie sur une touche du clavier.

```
/*Le code ci-dessous vous permet de tester lorsque la touche « D » a été relâchée après avoir été pressée.*/
Input.GetKeyUp(KeyCode.D)
```

Exercice 2 :

Reprenons maintenant notre projet Unity de la séance précédente avec une voiture.

b) **Constructeur pour script** :

1. Ajoutez un nouveau script « *Cycle Vie.cs* » à la voiture de votre projet de la séance précédente pour gérer sa construction et destruction.

2. Ajoutez les messages de texte suivants aux méthodes équivalentes au constructeur sur les scripts Unity : « *La voiture se réveille* » et « *La voiture finit son paramétrage juste avant son utilisation* ».
3. Sélectionnez votre voiture et décochez la case à côte du nom de ce script sur « *Inspector* ». Cette action désactivera le lancement de votre script vis-à-vis du moteur de jeu Unity.
4. Lancez votre jeu et vérifiez les messages de texte affichés.
5. Maintenant, arrêtez votre jeu, activez le script et relancez le jeu.

Questions :

- Avez-vous constaté des différences entre les résultats des instructions 4) et 5) ?
- Pouvez-vous faire appel aux méthodes Awake et/ou Start avec un argument ?

c) **Destructeur pour script :** ajoutez la méthode équivalente au destructeur au script « CycleVie.cs » avec le message : « *La voiture est en voie de destruction* ». De façon similaire à l' « *Exercice 1* **»**, ajoutez le code source nécessaire pour déclencher la destruction de la voiture à partir d'une touche du clavier. Faites attention que cette touche soit différente de celle utilisée par l'exercice précédent.

d) **Liaison entre scripts** :

1. Intégrez la classe « **Coffre Voiture** » au script « CycleVie.cs » sous la forme d'un attribut et initialisez-le dans la méthode appropriée.
2. Lancez votre jeu et déclenchez la destruction de l'objet « voiture ». Qu'observez-vous ?

Exercice 3 :

a) Téléchargez le prototype de jeu 2 proposé par Unity : Ressource Unity

b) Créez un nouveau projet de type « 3D » sur Unity et importez le « prototype 2 » dans votre projet.

Vous pouvez supprimer la scène rajoutée par défaut à votre projet afin de garder seulement la scène incluse dans le prototype.

c) Ajoutez un objet de type « Human », trois de type « Animals » et 1 objet de type « Food » dans votre scène.

Figure 1. Prototype 2 – Source : Unity Learn 2023

Utiliser la fenêtre « Inspector » afin de retourner de 180º les objets que représentent des animaux.

d) Créez un script « Contrôle Joueur » pour contrôler votre personnage et ajoutez à ce script la possibilité de déplacer horizontalement votre personnage (humain) dans la scène.

Vous pouvez utiliser les méthodes « GetAxis » ou « Translate » pour réaliser cette action.

e) Faites le nécessaire afin que le personnage ne puisse pas se déplacer au-delà du terrain de du jeu.

Parmi les solutions possibles, vous pouvez vérifier que la nouvelle position calculée pour le personnage est valide, et la modifier dans le cas contraire.

f) Créez un script qui déplace un objet vers l'avant (nommé « déplacement ») et rajoutez-le à l'objet de type « food ». Ce type d'objet jouera le rôle de projectile dans le projet.

Vous pouvez utiliser la méthode « Translate » et la direction « vector3.foward » pour accomplir cette action.

g) Créez un nouveau modèle d'objet (« prefab ») à partir de votre projectile (version avec le script) et enregistrez-le dans un dossier nommé « Prefabs »à l'intérieur du dossier « Assets ».

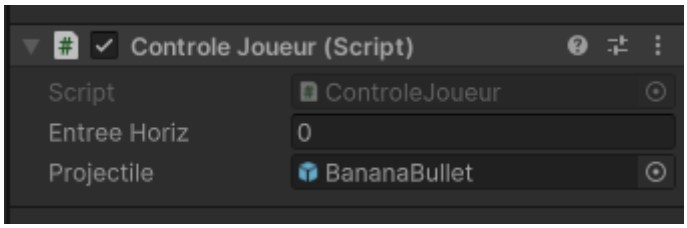
h) Rajoutez au script « Contrôle Joueur » la possibilité de tirer un projectile en utilisant la barre d'espace.

Pseudo-algorithme :

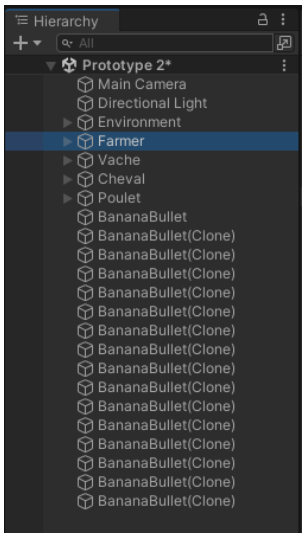
1. Utilisez la méthode « Input.GetKeyDown » afin de capturer l'événement d'appuyer sur la touche « espace ».
2. Créez un attribut de type « GameObject » dans votre script afin de définir le « prefab » à utiliser en tant que « projectile ».
3. Utilisez la méthode « Instantiate » afin de créer dynamiquement une instance de l'objet à tirer en utilisant l'objet stocké dans l'attribut « projectile ».

```
Instantiate(projectile,transform.position, projectile.transform.rotation);
```

4. Affectez le modèle d'objet crée, ici « BananaBullet », à l'attribut « projectile » en utilisant l'onglet « Inspector » d'Unity.



Vous pouvez constater qu'une fois tirés, les projectiles restent dans votre scène, même s'ils ne sont plus affichés.



i) Créez un script que détruit l'objet « GameObject » auquel il est attaché à partir du moment que l'objet sort de la scène. Rajoutez ce script au « modèle » de projectile créé précédemment (ou créez un nouveau modèle).

Vous pouvez utiliser la méthode « Destroy » vue au début de la séance pour détruire les projectiles.

j) Utilisez la procédure apprise pendant les exercices précédents afin de déplacer automatiquement les animaux du haut vers le bas de la scène (questions f à i de l'exercice 3). Faites le nécessaire afin que les objets de type « animal » soient aussi détruits lorsqu'ils dépassent les bords du terrain.

Considérations finales

Une partie des exercices de ce TD a été adaptée de l'unité 2 du Parcours Unity Learn - Junior Programmer. N'hésitez pas à visiter leur formation en ligne afin d'avoir accès au cours complet, ainsi qu'à avancer en autonomie sur les sujets que n'ont pas été traité en cours.

Références

- https://docs.microsoft.com/fr-fr/dotnet/csharp/programming-guide/classes-and-structs/destructors
- https://answers.unity.com/questions/513439/when-should-i-use-constructors-vs-using-awake-or-s.html
- https://docs.unity3d.com/ScriptReference/MonoBehaviour.Awake.html
- https://docs.microsoft.com/fr-fr/dotnet/standard/garbage-collection/

Date : 10/10/2023

Séance 6 – Moteur physique et gestion des collisions

Objectifs

- Introduire la gestion des collisions sur Unity
- Mise en pratique de concepts décrits sur le moteur de jeu Unity

Moteur de physique : le moteur de jeu vidéo Unity contient un ensemble des fonctionnalités pour nous aider à simuler la physique du monde réelle. Pour utiliser le moteur physique Unity, nous devons ajouter un composant « Rigidbody » à une instance d'un objet « GameObject ». Parmi les attributs plus utilisés d'un composant de type « Rigidbody », nous avons : « is kinematic », « use gravity » et « constraints ».

- **« Is Kinematic »** : cet attribut de type booléen définit si un objet « Rigidbody » est affecté par le moteur de physique.
- **« Use Gravity »** : cet attribut de type booléen définit si un objet « Rigidbody » est affecté par la gravité.
- **Les contraintes** sur un composant « Rigidbody » permettent de paramétrer les dégrées de liberté de son mouvement (ou déplacement). Les attributs « Freeze position » et « Freeze rotation » interdisent, respectivement, la translation et la rotation d'un objet.

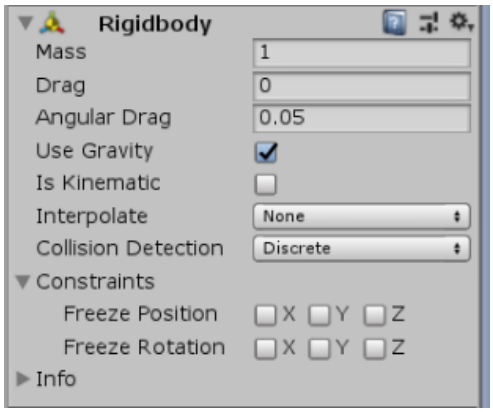


Figure 1. . Paramètres d'un composant « Rigidbody » sur la fenêtre « Inspector »

Gestion de collisions : le moteur de jeu Unity propose aussi des classes (ou des composants) pour gérer les collisions entre des objets. Pour avoir accès à cette fonctionnalité, une instance d'un GameObject doit contenir un composant de type « Collider ». Nous avons différents types de « Collider », par exemple, « SphereCollider » , « BoxCollider » et « MeshCollider ». Un composant « Collider » est comme une enveloppe (2D ou 3D selon le type de jeu) autour de l'objet. Elle peut être plus petite, exactement de la taille de l'objet ou plus grande. Nous devons la modéliser de façon à couvrir l'ensemble du volume autour de l'objet où nous voulons déclencher une collision. Pour modifier un « Collider », nous devons sélectionner l'objet « GameObject » cible, ensuite aller sur le menu « Inspector » et appuyer sur l'option « Edit Collider ».

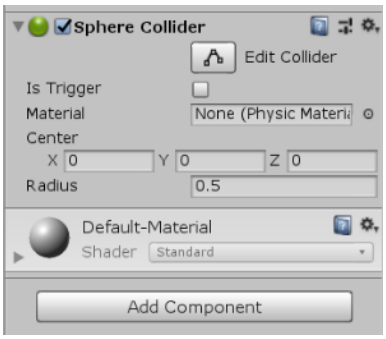


Figure 2. Paramètres d'un composant « SphereCollider » sur la fenêtre « Inspector »

Pour capturer des collisions entre des objets, nous utilisons des méthodes proposées par « MonoBehavior ». Les méthodes «Collider.OnCollisionEnter() », « Collider.OnCollisionStay()» et « Collider.OnCollisionExit() » sont appelées automatiquement lorsqu'un objet rentre en collision, reste en collision et fini sa collision avec un autre objet, respectivement. Ces méthodes possèdent comme argument un objet de type « collision » qui porte des informations relatives à la collision, telles qu'un pointer (non modifiable) vers l'objet « GameObject » tiers qui a fait une collision avec l'objet qui contient le script, les points de contact de la collision, entre autres informations. L'exemple ci-dessous montre comment capturer le nom de l'objet tiers qui vient de faire une collision avec l'objet cible. Cette méthode est appelée une seule fois pendant l'événement de la collision.

Exemple

```
private void OnCollisionEnter(Collision collision)
{
 Debug.Log("J'ai fait une collision avec l'objet:"
            + collision.rigidbody.name);
}
```

Les méthodes basées « *collision* » sont recommandées lorsque nous voulons capturer des collisions entre les objets affectés par le moteur physiques. Néanmoins, nous pouvons aussi capturer des collisions avec des objets qui ne sont pas affectés par le moteur physique.

Pour définir qu'un objet ne sera pas affecté par la physique lors d'une collision, nous devons cocher l'attribut « is trigger » de son composant « Collider ». Lorsque l'attribut « Is trigger » est activé, nous utiliserons les méthodes « OnTriggerEnter() », « OnTriggerExit() » et « OnTriggerStay() » pour traiter des collisions à la place des méthodes de la famille « OnCollision ».

Exemple

```
private void OnTriggerEnter(Collider other){
Debug.Log("J'ai fait une collision avec l'objet :" + other.name);
}
```

Méthode FixedUpdate : précédemment nous avons toujours utilisé la méthode « Update() » dans les scripts comportementaux pour capturer l'interaction avec l'utilisateur et faire des mises à jour de la scène, tel que le déplacement des objets. Néanmoins, dès que nous utilisons le moteur physique Unity, des objets « GameObject » avec un composant « Rigid Body », nous devons réaliser certaines actions dans la méthode « FixedUpdate » à la place de « Update ».

La méthode « Update » doit continuer à être utilisée pour des mises à jour ordinaires du jeu, comme le déplacement des objets non physiques, des chronomètres simples et du traitement des interactions avec l'utilisateur. La méthode « FixedUpdate » doit être utilisée pour toutes

les actions qui affectent des objets avec des composants « Rigidbody », car tous les calculs physiques sont faits par Unity juste après l'appel à cette méthode.

Notons que la méthode « Update() » est appelée une fois par « frame ». En revanche, la méthode « FixedUpdate() » peut être appelée zéro, une ou plusieurs fois par frame. Cette différence est due au fait que l'intervalle entre deux appels à la méthode « update » dépend du temps de traitement de la « frame » précédente. Par conséquent, les délais entre les appels à la méthode « Update() » seront fréquemment différents. En revanche, l'intervalle entre deux appels à la méthode « FixedUpdate » est paramétré et donc constant.

Le délai entre deux appels à la méthode « FixedUpdate » peut être modifié dans le menu « Édit > Settings > Time > Fixed Timestep ».

Le code source ci-dessous montre un exemple de répartition de code source entre les méthodes « Update » et « FixedUpdate » pour le déplacement d'un objet contrôlé par le joueur. L'implémentation proposée utilise la méthode « Input.GetAxis() » pour récupérer l'interaction avec l'utilisateur dans la méthode « Update ». L'information récupérée est stockée dans des attributs de la classe utilisée. Les valeurs de ces attributs seront utilisées dans les méthodes « deplacement() » et « rotation() » qui sont appelées dans la méthode « FixedUpdate() ». Les définitions de ces deux méthodes sont données plus tard dans la section « déplacement des objets physiques ».

```
private float deplacementAxeVertical ;
private float deplacementAxeHorizontal ;
private void Update(){
      deplacementAxeVertical   = Input.GetAxis ("Vertical");
      deplacementAxeHorizontal = Input.GetAxis ("Horizontal");
}
private void FixedUpdate(){
      deplacement() ;
      rotation() ;
}
```

Déplacement des objets physiques : lorsque nous utilisons des objets qui sont affectés par le moteur physique Unity, nous devons utiliser les méthodes proposées par la classe « Rigidbody » pour déplacer nos objets à la place de ceux définis par la classe « Transform ». Par exemple,

- *Rigidbody.MovePosition* (*Vector3*) : elle déplace un objet à partir du vecteur 3D passé en argument.

```
//implémentation de la méthode « deplacement() »
float vitesseDeplacement = 10f ;
Rigidbody objet_rb = GetComponent<Rigidbody>() ;
Vector3 mouvement = transform.forward * deplacementAxeVertical *
vitesseDeplacement * Time.deltaTime;
objet_rb.MovePosition(objet_rb.position + mouvement);
```

- *Rigidbody.MoveRotation(Quaternion)* : cette méthode applique à l'objet la rotation passée en argument sous la forme d'un objet « Quaternion ». La classe « Quaternion » est utilisée pour représenter des rotations sur Unity.

La méthode « Rigidbody.MoveRotation(Quaternion) » a l'avantage de ne pas être affectée par le problème de « gimbal lock » (https://fr.wikipedia.org/wiki/Blocage_de_cardan).

```
// implémentation de la méthode « rotation » ;
float vitesseRotation = 5f ;
Rigidbody objet_rb = GetComponent<Rigidbody>() ;
objet_rb = GetComponent<Rigidbody>() ;
float rotation = deplacementAxeHorizontal * vitesseRotation * Time.deltaTime;
Quaternion q_rotation = Quaternion.Euler (0f, rotation, 0f);
objet_rb.MoveRotation (objet_rb.rotation * q_rotation);
```

Exercice 1 :

Téléchargez le projet « GestionCollision » sur la plateforme de cours et effectuez les actions proposées par les exercices 1 à 6.

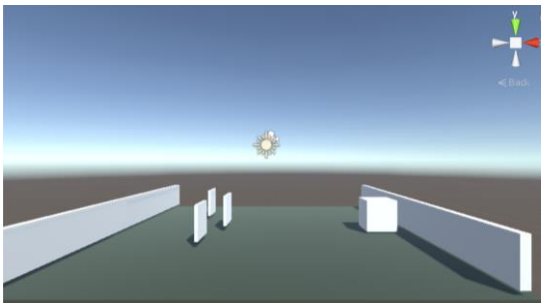


Figure 3. Scène proposée

1. L'objet « terrain » et l'un des murs tombent lorsque le jeu est lancé. Paramétrez-les de façon qu'il ne soit plus affecté par la gravité.
2. Le code source actuel déplace l'objet « joueur » à l'aide de son composant « Transform » de la méthode « Update ». Modifiez le code source pour utiliser les méthodes proposées pour le déplacement des objets physiques.

   a) Attention à bien répartir le code entre les méthodes « Update() » « FixedUpdate() », et préférez l'utilisation de la méthode « GetAxis() » à la place de la méthode « GetKey() ».

   b) Pensez aussi à modifier les contraintes du composant « Rigidbody » de l'objet joueur pour autoriser des changements de direction de déplacement.

3. Vous pouvez utiliser le code source ci-dessous pour arrêter le déplacement (attribut « velocity ») et la rotation (attribut « angularVelocity) du composant Rigidbody e l'objet "joueur" après son déplacement.

```
« rigid_body_objet ».velocity          = Vector3.zero;
« rigid_body_objet ».angularVelocity   = Vector3.zero;
```

4. Créez des obstacles pour l'objet « joueur » comme la montre la Figure 3. Faites en sorte que lorsque l'objet « joueur » rentre en collision avec l'un des obstacles, cet obstacle soit affecté par le moteur de physique et qu'il change de couleur selon les règles suivantes :
   - Jaune : moment de la collision et collision en cours
   - Couleur originale : collision finie

Conseil : commencez par afficher un message sur la console lorsqu'une collision est détectée pour une méthode donnée de traitement de collision. Ensuite, essayez de capturer les différentes phases de collision. Une fois que vous avez terminé les étapes précédentes, passez au changement de couleurs.

5. Entourez la scène des murs et ajoutez plusieurs objets de type « obstacle ». Paramétrez les contraintes des composants « Rigidbody » afin d'éviter que les obstacles et les murs tombent de la scène.

Perfectionnement

6. Créez un nouvel objet basé sur le « GameObject » de type « Capsule ». Lorsque l'objet « joueur » traverse ce nouvel objet, l'objet « joueur » devient plus grand et l'objet traversé disparaît. L'objet traversé ne doit pas subir les effets du moteur de physique.
7. Finalement, lorsque l'objet « joueur » a consommé un objet « Capsule », il pourra détruire un objet « obstacle ». Le jeu terminera dès qu'il n'y a plus d'obstacle dans la scène. Attention, lors d'une collision, l'objet « Collision » passé en argument contient une instance de GameObject. En revanche, cette instance est en mode lecture seulement.

N'hésitez pas à utiliser la méthode « Find » de la class « GameObject » pour chercher l'instance à détruire.

Exercice 2 :

Reprenons maintenant notre projet Unity de la séance précédente (les animaux de la ferme). Jusqu'à présent, vous avez placé manuellement les animaux dans la scène avant le démarrage du jeu. Dans cet exercice, votre premier objectif sera de gérer les collisions entre les animaux et la nourriture lancée par le joueur. Ensuite, nous allons améliorer notre jeu afin de le munir de la capacité de créer des animaux dans la scène de façon automatique, afin de simuler un effet des vagues « d'ennemies ». À la fin de ce TD, la position initiale de chaque animal sera choisie aléatoirement, ainsi que le modèle utilisé pour le créer et les animaux seront créés dans la scène automatiquement et à un intervalle régulier.

Modèles d'animal :

1. Créez un script en utilisent les méthodes de détection de collision vues pendant la séance afin de faire disparaître de la scène les animaux lorsqu'ils sont nourris par le personnage (collision entre un animal et la nourriture) .
2. Ajoutez votre script ainsi qu'un composant « Collider » au modèle (« prefab ») de chaque animal utilisé dans votre scène. Vous devez ajouter aussi un objet de type « Collider » au modèle de projectile.

   - Le type d'objet « Collider » choisi doit être adapté à l'objet utilisé.
   - Vous devez créer un nouveau modèle « prefab » pour chacun de trois animaux choisis contenant le script créé, ou mettre à jour vos modèles précédents.

Gestion de la création des animaux (Animal Spawn Manager)

3. Créez un objet vide « Empty Game Object », ainsi qu'un script nommé « Gestionnaire Jeu » pour gérer la création des animaux. Associez ce script à l'objet créé.
4. Rajoutez une méthode, nommée « Creation Animal », au script « Gestionnaire Jeu » avec l'objectif de créer un objet à partir des arguments suivants : un modèle « prefab », une position et une rotation dans la scène.

   - Vous pouvez utiliser la méthode « Instantiate » pour créer les objets de type "GameObject".
   - Le code source attendu est similaire à celui utilisé pour créer des projectiles.
   - Le modèle « prefab » à utiliser doit être un attribut de votre script.

5. Utilisez la méthode « Creation Animal » pour créer une instance d'un modèle "prefab" d'un animal choisie par vous, à chaque fois que nous appuyons sur la touche « C ». La position de l'animal doit être choisie au hasard.

Placement au hasard des animaux dans la scène : vous pouvez fixer deux dimensions de la position choisie pour placer les animaux dans la scène (e.g., Y et Z), et tirer au hasard la dimension restante en utilisant la méthode « Random.Range(…) ». Dans le cas de notre jeu, vous pouvez tirer au hasard la dimension X du vecteur qui contient la position de l'objet.

Je vous conseille de définir l'intervalle de valeurs fournies à la fonction « Range » en utilisant des attributs de votre script.

6. Lorsque vous avez validé le fonctionnement de votre script « Gestionnaire Jeu» pour la création des animaux à la demande, modifiez votre code source afin d'utiliser la méthode « InvokeRepeating » pour créer les animaux dans la scène de façon totalement automatique à un intervalle de temps régulier.

La méthode « InvokeRepeating » permet de faire des appels réguliers à une fonction. Cependant, la fonction appelée ne peut pas atteindre des arguments.

https://docs.unity3d.com/ScriptReference/MonoBehaviour.InvokeRepeating.html

Jusqu'à présent, nous avons utilisé toujours le même modèle d'animal, et nous avons créé plusieurs instances de ce modèle dans la scène, placés de façon aléatoire.

7. Modifiez votre code source afin de choisir le modèle d'animal à utiliser pour créer un objet aléatoirement, parmi un ensemble de modèles indiqués au préalable (fenêtre « Inspector »).

- Vous pouvez modifier l'attribut de type « GameObject » qui indiquait un modèle « prefab » à utiliser afin qu'il puisse comporter une liste d'objets de type « GameObject ».
- N'oubliez pas d'indiquer les modèles « prefab » à utiliser par votre script.

Considérations finales

Une partie des exercices de ce TD a été adaptée de l'unité 2 du Parcours Unity Learn - Junior Programmer. N'hésitez pas à visiter leur formation en ligne afin d'avoir accès au cours complet, ainsi qu'à avancer en autonomie sur les sujets que n'ont pas été traité en cours.

Références

- https://docs.unity3d.com/ScriptReference/Rigidbody-isKinematic.html
- https://docs.unity3d.com/ScriptReference/Rigidbody-constraints.html
- https://docs.unity3d.com/ScriptReference/Collision.html
  https://docs.unity3d.com/ScriptReference/Rigidbody.MovePosition.html
- https://docs.unity3d.com/ScriptReference/Quaternion.html
- https://docs.unity3d.com/ScriptReference/Random.Range.html

Date : 16/10/2023

Séance 7 – Relations entre classes et objets sur Unity

Objectifs :

- Introduire les notions des relations entre classes
- Mise en pratique de concepts décrits sur le moteur de jeu Unity

Concepts :

Relations d'association : les relations d'association modélisent le passage d'information (ou communication) entre les objets des différentes classes. Une association peut être entre deux classes ou entre plusieurs. Sans savoir, vous avez déjà utilisé des relations dans nos projets de discipline.

Par exemple, dans plusieurs occasions nous avons utilisé un script (qui devient un objet lorsqu'il est attaché à un GameObject) pour modifier les attributs d'un objet « Transform », responsable pour le déplacement d'un « GameObject ».

Nous avons aussi utilisé des relations pour passer des informations entre un objet script qui contrôlait le déplacement d'une voiture vers un objet qui géré la quantité d'essence restante, la classe « *GestionVoiture* ». Ce dernier objet renvoyait en retour si la voiture pouvait continuer à rouler.

Nous nous intéressons maintenant à deux types particuliers d'association : **l'agrégation** et la **composition**. Ceux deux types de relations modélisent la relation objet composite /composant (ou tout/partie). La différence pratique entre elles peut être faite par rapport au cycle de vie des objets modélisés. Si l'objet composant n'existe pas sans l'objet composite, c'est une relation de composition. Néanmoins, si le composant continuer à exister après la destruction de l'objet composite, la relation est une agrégation.

Par exemple, les objets qui modélisent les roues d'un objet « voiture » n'ont pas de raison d'exister lorsque l'objet voiture est détruit. Donc, une composition. Imaginons maintenant le cas d'un jeu de course avec plusieurs pistes et plusieurs voitures. Une piste aura des relations d'agrégation avec plusieurs voitures, vu qu'à la fin de chaque course, les voitures pourront passer à une autre piste, mais la piste cessera d'exister.

Exemple d'une relation de composition

```
public class Voiture : MonoBehaviour
{
    public GameObject roueAvantDroite   = null;
    public GameObject roueAvantGauche   = null;
    public GameObject roueArriereDroite = null;
    public GameObject roueArriereGauche = null;
```

Nous pourrions aussi utiliser le code source au-dessus pour coder une relation d'agrégation. C'est le projet du jeu qui dira quel type d'association nous utiliserons et la façon que nous allons signaler la destruction des objets composés qui fera la différence entre ces deux types de relations dans le code source.

Navigation des objets composites : lorsque vous ajoutez un GameObject 3D dans la scène (un cube, une sphère ou un objet plus avancé) vous pouvez regarder sur la fenêtre « Inspector » qu'il contient plusieurs composants par défaut, tels que : un MeshRenderer, un Transform et un BoxCollider. Nous avons couramment augmenté cette liste avec les scripts, par exemple.

La bibliothèque de classes Unity nous propose plusieurs méthodes pour nous aider à naviguer les objets dans un GameObject avec le langage C#, tels que : *GetComponent*, *GetComponenetInChildren* and *Find*.

Méthode « GetComponent » : elle nous permet d'accéder à un composant d'un objet « GameObject ». Par exemple, le MeshRenderer, le Transform et le BoxCollider. L'objet sera cherché par rapport au type saisi entre les symboles « < » et « > ».

Exemple :

```
Renderer apparence = getComponent<Renderer>();
```

Méthode GetComponentInChildren : cette méthode renvoie la liste des composants du type spécifié qui appartient au GameObject cherché et à ses sous-objets. Le code ci-dessous initialise une liste avec les composants « Transform » de tous les sous-objets contenu dans « gameObject », son « Transform » inclus.

Exemple :

```
Transform[] liste2 = gameObject.GetComponentsInChildren<Transform>();
```

La boucle **foreach** est bien adaptée pour itérer sur une liste d'éléments. Par exemple :

```
foreach (Transform transf in liste2)
{
  Debug.Log("Trnsf:" + transf.name + "," + transf.GetType());
}
```

GameObject.Find : cette méthode cherche un objet « GameObject » dans la scène par son nom. Le code ci-dessous cherche dans la scène un objet appelé « MaCamera ».

```
GameObject lien = GameObject.Find("MaCamera");
```

Pour tester si un objet a été trouvé, vous pouvez simplement utiliser la variable qui recevra le résultat de la recherche comme condition dans un test conditionnel.

```
GameObject lien = GameObject.Find("MaCamera");
if (lien){
     Debug.Log("L'objet a été trouvé");
}
else{
     Debug.Log("L'objet n'a pas été trouvé");
}
```

Exercice 1

Créez un nouveau projet de jeu 3D sur Unity et réalisez les actions suivantes :

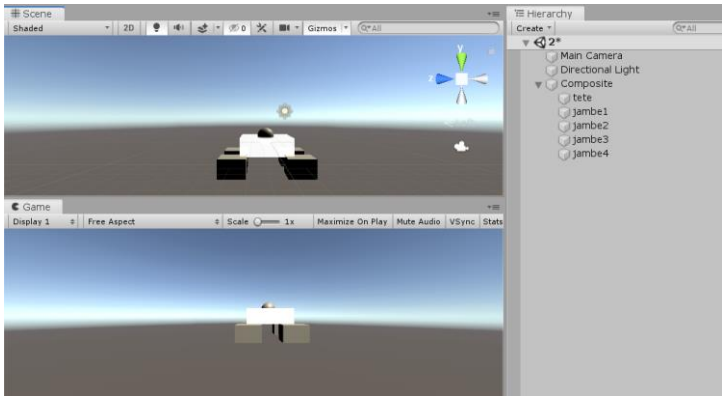
1. **Accès à un composant**. Ajoutez un GameObject 3D à votre scène. Ajoutez un script à votre objet qui cherche le composant « MeshRenderer » du GameObject et change sa couleur lorsqu'une touche du clavier est appuyée. Vous pouvez utiliser la classe « Color »[1] pour choisir la couleur à affecter à l'objet.

[1] https://docs.unity3d.com/ScriptReference/Color.html

Accès à l'attribut couleur du composant « MeshRenderer » :

```
Renderer -> material -> color
```

2. **Objet composite**. Créez un objet 3D composé d'autres objets 3Ds comme l'exemple ci-dessous. Nommez chaque sous-objet différemment. Utilisez les méthodes de navigation vues pour afficher les noms de chaque sous-objets d'un objet composite sur la console. Attention, le script doit être attaché à l'objet composite. Ensuite, faites que la taille des sous-objets « jambes » soit réduite de 25 % à chaque fois que nous appuyons sur la touche « R ». Vous devez chercher les sous-objets cible par nom.



Exercice 2 - Création d'un jeu de type « Coureur » en perspective 2,5D

Import et placement des élément dans la scène

1. Téléchargez le prototype de jeu 3 proposé par Unity : « Prototype 3 - Runner »
2. Créez un nouveau projet de jeu de type « 3D » et importez le « prototype 3 » dans votre projet.

   Vous pouvez supprimer la scène rajoutée par défaut à votre projet afin de garder seulement la scène incluse dans le projet par le prototype.

3. Vous pouvez changer l'arrière-plan de votre jeu en modifient le champ « Sprite » dans le composant « Sprite Renderer » de l'objet « Background ».
4. Choisissez un personnage (« Characters ») et placez-le dans la scène. Ajoutez les composants « RigidBody » et « BoxCollider » au personnage.
5. Ajoutez un script à votre personnage. Ce script comportera les fonctions nécessaires pour lui contrôler. Nous l'appellerons « Controle Personnage ».

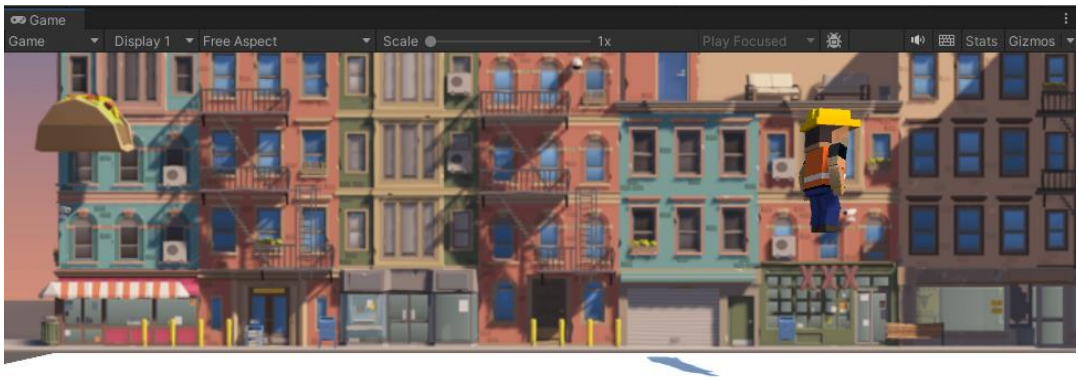
Contrôle du personnage

Nous allons maintenant ajouter la fonctionnalité de « sauter » au personnage. Rajoutez les éléments suivants au script « Controle Personnage ».

6. Créez un attribut afin de stocker une référence au composant « RigidBody » du personnage. Par exemple, « instanceRigidBody ».
7. Utilisez la méthode « GetComponent » dans la méthode « Start » du script afin d'initialiser l'attribut « instanceRigidBody » avec une référence à l'objet « RigidBody » du personnage.
8. Dans la méthode « Update », utilisez la méthode « AddForce » du composant « RigidBody » afin de faire le personnage sauter lorsque qu'on appuie sur la touche « espace » du clavier.

```
//L'exemple de code ci-dessous applique à l'objet une force de « 50 » unités vers le bas de la scène
instanceRigidBody .AddForce(Vector3.down*50, ForceMode.Impulse) ;
```

- Choisissez les valeurs adaptées pour la direction et l'intensité de la force, ainsi que le type de force.
- Pensez à utiliser un attribut pour stocker la valeur d'intensité de la force à être appliquée à votre personnage afin de permettre de tester différents valeurs à partir de l'onglet « Inspector ».



9. Modifiez le paramètre « Physics.gravity » dans la méthode « Start » du script « Controle Personnage »afin de modifier l'effet de la gravité sur le saut du personnage.

```
//le code ci-dessous augment la force de la gravité de 3 fois.
Physics.gravity *= 3 ;
```

Pensez à ajouter un attribut à votre script afin de stocker le facteur de modification de la gravité que vous voulez appliquer à votre jeu.

10. Choisissez des valeurs pour l'attribut d'intensité de force de saut et le modificateur de gravité afin d'obtenir un saut avec un comportement naturel.

Dans l'état actuel du jeu, votre personnage peut sauter indéfiniment, et ainsi monter dans la scène sans restriction. Nous allons maintenant utiliser la détection de collisions afin d'éviter ce comportement.

11. Créez un attribut booléen dans le script afin de savoir si le personnage touche le sol.

L'attribut prend la valeur de vrai si le personnage touche le sol, et faux sinon.

12. Choisissez la méthode de collision adaptée afin d'identifier lorsque le personnage rentre en collision avec le sol.
13. Ajoutez une contrainte au code source qui gère le saut du personnage afin d'éviter que le personnage puisse sauter s'il ne touche pas le sol.

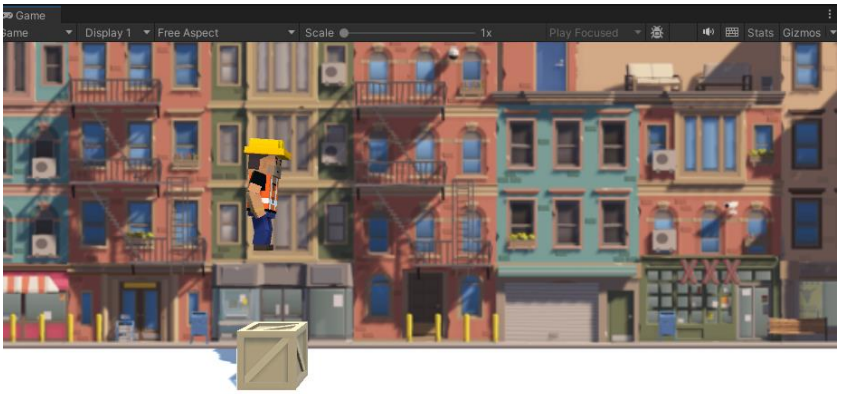
Création d'un modèle d'obstacle

Maintenant que le personnage est capable de sauter dans le jeu, nous allons créer des obstacles afin de rendre son parcours plus amusant.

14. Ajoutez un objet du dossier « /Course Library/Obstacles/ » au jeu.
15. Rajoutez des composants « RigidBody » et « Collider » à l'objet choisi.
16. Rajoutez un script à l'objet qui lui déplace vers la gauche.

La vitesse de déplacement de l'objet doit être paramétrable via la fenêtre « Inspector »

17. Testez votre objet « obstacle » dans la scène. Lorsque l'objet est fonctionnel, créez un modèle « prefab » à partir de cet objet.

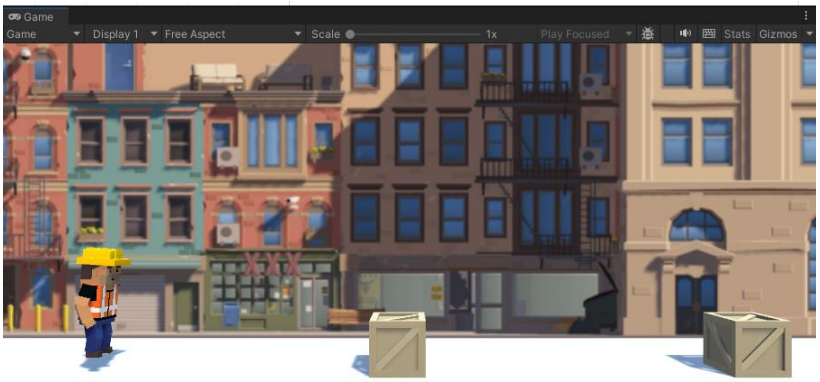


Création automatique d'obstacles

Nous disposons maintenant d'un modèle d'obstacle pour notre jeu. Notre prochain objectif est de créer automatiquement des obstacles dans la scène à intervalles réguliers.

1. Ajoutez un objet de type « Empty GameObject » à votre scène pour vous en servir en tant que gestionnaire du jeu. Ajoutez un script à cet objet afin de contrôler la création des obstacles dans la scène.
2. Utilisez les méthodes « Instantiate » et « InvokeRepeating » vues pendant la séance précédente afin de créer des obstacles dans la scène en utilisant le modèle d'obstacle créé.

- Le modèle d'obstacle à utiliser doit être modifiable via la fenêtre « Inspector ».
- Différemment de la séance précédente, la position de création des obstacles est fixe.
- Utilisez un attribut afin de définir la position initiale des obstacles.
- N'oubliez pas que la méthode « InvokeRepeating » n'accepte pas d'appeler une fonction qui requiert d'arguments.



Animation de l'arrière-plan et gestion du jeu

Pendant cet exercice, nous allons travailler sur l'animation de l'arrière-plan du jeu.

1. Ajoutez à l'arrière-plan du jeu (objet « background ») une copie du script créé précédemment qui déplace un objet à gauche.

Votre arrière-plan se déplace maintenant en créant une animation, mais il arrive éventuellement au bout de la scène.

2. Ajoutez un deuxième script à l'objet « background », nommé « RepetitionArrierePlan ». Ce script sera responsable pour remettre l'objet « background » à sa position au démarrage du jeu à chaque fois que l'arrière-plan du jeu sort du cadre de la scène.

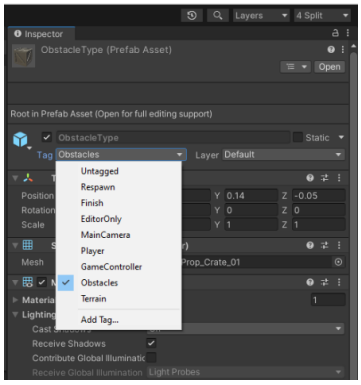
> Vous pouvez rajouter un attribut « RepetitionArrierePlan » afin de stocker la position de l'arrière-plan au démarrage du jeu.

Condition de fin du jeu

Maintenant que nous avons ajouté des comportements au personnage, aux obstacles et à l'arrière-plan, nous allons travailler sur la détermination de la fin du jeu et son effet sur les éléments du jeu.

3. Ajoutez une étiquette (*tag*) nommée « Obstacle » au modèle « prefab » des obstacles afin de pouvoir les identifier par un identifiant unique. Vous pouvez faire de même pour le terrain du jeu (étiquette « Terrain) et le personnage (étiquette « Player »).

> Pour rajouter une étiquette à un objet, sélectionnez l'objet (ou « prefab ») d'intérêt et utilisez l'option « Add tag » sur le champ « Tag », accessible via la fenêtre « Inspector ».



Script « Contrôle Personnage »

4. Ajoutez un attribut de type booléen au script qui contrôle le personnage. Cet attribut aura pour objectif d'identifier l'état du jeu (faux : le jeu est en cours, vrai : le jeu est fini).
5. Modifiez le code source du script qui contrôle le personnage afin qu'il puisse prendre en compte des différents types de collision, en utilisant le système d'étiquettes mis en place.

- Si le personnage fait une collision avec le sol, il est à nouveau capable de sauter.
- S'il fait une collision avec un obstacle, le message « Fin du jeu » est affiché à l'aide de la méthode « Debug.Log() ». Pensez à modifier l'attribut qui indique la fin du jeu.

Scripts « Gestion Jeu » et « DeplacerGauche »

6. Faites le nécessaire afin d'arrêter la création d'objets ainsi que le déplacement de l'arrière-plan lorsque le jeu est fini.

- Vous pouvez créer un attribut du même type que le script qui contrôle le personnage (e.g., « ControlePersonnage ») dans les scripts qui gère la création d'objets et le déplacement des objets à gauche.
- La méthode GameObject.Find (« nom de l'objet ») peut vous aider à obtenir une référence à l'objet « personnage ». En utilisant cette référence, vous pouvez obtenir une référence vers le script de gestion du personnage avec la méthode « GetComponent<TYPEOBJET>() », et ainsi suivre la valeur de l'attribut « fin du jeu ».
- Vous pouvez utiliser la méthode « CancelInvoke(« nom de la méthode ») » pour arrêter la création d'obstacles.

Script « DeplacerGauche »

Vous avez probablement remarqué que les obstacles créés pendant le jeu s'accumulent dans votre scène. Vous pouvez les visualiser en regardant la fenêtre « Hierarchy ».

7. Modifiez le script « Déplacer à gauche » afin qu'il détruise les objets de type « obstacle » qui sortent du cadre de la scène.

- Pensez à utiliser le système de « tag » afin de détruire seulement les objets « obstacle ».
- Utilisez un attribut de votre script afin de définir à partir de quelle position (ou dimension), vous considérez qu'un objet est à l'extérieur de la scène et doit être ainsi détruit.

Considérations finales

Une partie des exercices de ce TD a été adaptée de l'unité 3 du Parcours Unity Learn - Junior Programmer. N'hésitez pas à visiter leur formation en ligne afin d'avoir accès au cours complet, ainsi qu'à avancer en autonomie sur les sujets que n'ont pas été traité en cours.

Références

- https://docs.unity3d.com/ScriptReference/GameObject.GetComponent.html
- https://learn.unity.com/tutorial/lesson-3-1-jump-force?uv=2021.3&pathwayId=5f7e17e1edbc2a5ec21a20af&missionId=5f7648a4edbc2a5578eb67df&projectId=5cf9639bedbc2a2b1fe1e848#5ce35aa5edbc2a29e31b3c71

Date : 16/10/2023

Séance 8 – Animations, effets sonores et des particules

Pendant cette séance, nous allons enrichir le projet de la séance précédente avec des animations, et des effets de particules et sonores.

Exercice 1 – Animation du personnage

Dans cet exercice, nous allons travailler sur le composant d'animation du personnage.

1. Ouvrez la gestion des animations du personnage en cliquant deux fois sur le composant « Animator » associé à l'objet « GameObject » du personnage.
2. Prenez le temps de vérifier les animations disponibles ainsi que les conditions de transition modélisées.

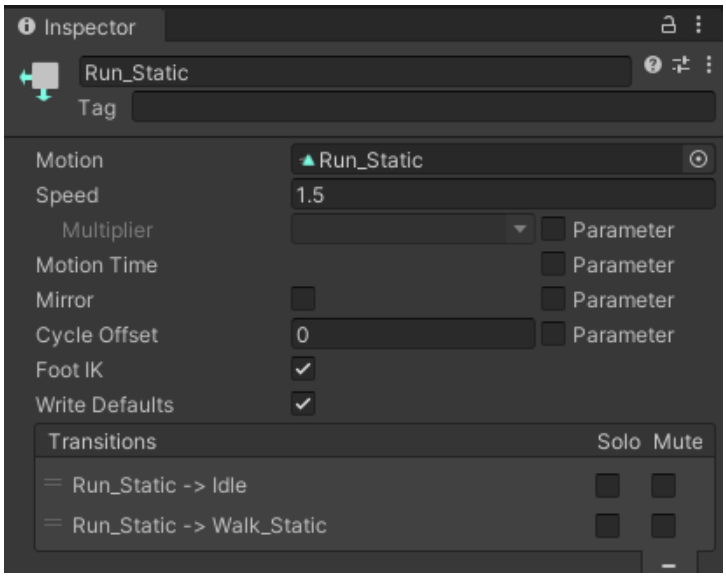
   - Vous pouvez cliquer deux fois sur une animation pour visualiser ses paramètres sur l'onglet « Inspector ».
   - La touche « Alt » vous permet de naviguer sur la fenêtre contenant les animations.

3. Définissez l'état d'animation « *Run_static* » comme l'animation par défaut du personnage

   Clic droit sur l'animation ciblé et ensuite sélectionnez l'option « Layer default state ».

4. Modifiez la vitesse de l'animation afin de la rapprocher de la vitesse de l'arrière-plan, et d'ainsi obtenir un jeu fluide.

   - Vous pouvez modifier la vitesse d'animation en sélectionnent l'animation cible, e.g., « run_static », et ensuite en modifiant le champ « vitesse », sous la fenêtre « Inspector ».
   - Vous pouvez aussi modifier la vitesse du personnage dans la variable « speed_ » sous l'onglet « Parameters ».



5. Dans le script de contrôle du personnage, ajoutez et initialisez un attribut de type « Animator » afin de garder une référence au composant « Animator » du personnage.

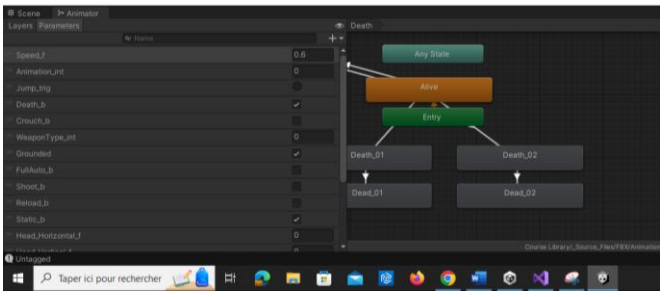
Vous pouvez utiliser la méthode « GetComponent » pour récupérer le composant « Animator » associé au personnage du jeu.

6. Utilisez la méthode « SetTrigger » du composant « Animator » afin de déclencher une animation lorsque le personnage saute.

   - Vous devez saisir le nom de l'animation (état) en argument de la méthode « SetTrigger ».
   - L'animation à utiliser s'appelle « Jump_trig »

7. Revisitez les attributs « mass », « force », et « gravity » de l'objet « RigidBody » du personnage si nécessaire, afin d'obtenir un comportement de saut compatible avec l'animation ajoutée.
8. Ajoutez maintenant une animation pour représenter la mort du personnage lorsqu'il fait une collision avec un obstacle.
   a. Utilisez la méthode « SetBool » afin d'indiquer que la variable « Death_b » déviant vrai.
   b. Vous devez aussi utiliser la méthode « SetInteger » afin d'indiquer le type d'animation de mort que vous voulez utiliser (« DeathType_int).

```
//Exemple de code
playerAnim.SetBool("Death_b", true) ;
palyerAnim.SetInteger   ("DeathType_int",1) ;
```



Exercice 2 – Effets de particules

Dans cet exercice, nous allons rajouter des effets de particules dans notre jeu.

Prise en main :

- Placez un effet de particules du dossier « /Assets/Course Library/Particles/ » dans la scène afin de comprendre son fonctionnement. Vous remarquerez qu'une interface s'affiche. Cette interface vous permet de jouer, redémarrer et arrêter la réalisation d'un effet de particules.

9. Maintenant, ajoutez l'effet « explosion smoke » en tant que sous-objet de votre personnage. Placez-le au niveau des pieds du personnage.

Assurez-vous que l'option « Play on awake » de l'objet effet est décochée. Nous allons déclencher l'effet en utilisant en réponse aux actions réalisées par le personnage.

10. Ajoutez un attribut de type « ParticleSystem » au script qui contrôle le personnage.
11. Affectez l'objet « effet de particules » à l'attribut créé dans le script de contrôle du personnage.
12. Utilisez la méthode « Play() » de l'objet « ParticleSystem » dans le code source du jeu afin de déclencher cet effet lorsqu'un objet de type « obstacle » fait une collision avec le personnage ».
13. Testez votre jeu. Lorsque vous avez validé le fonctionnement de l'effet de particules, faites de même afin de déclencher un effet de type « poussière » lorsque le personnage touche le sol.

> Attention, l'effet de type « poussière » doit être arrêté lorsque le personnage saute ou est atteint par un obstacle.

Exercice 3 – Musiques et effets sonores

Nous allons maintenant ajouter des effets sonores au jeu.

14. Additionnez un composant de type « Audio Source » à la caméra du jeu.

> Vous pouvez utiliser l'option « Add Component » sur l'onglet « Inspector » de l'objet « Main Camera ».

15. Ajoutez un clip audio de votre préférence à l'attribut « AudioClip » du composant « AudioSource » de la caméra.

> - Vous pouvez trouver plusieurs clips audio dans le dossier « Assets/Course Library/ Sound/ ».
> - Pensez à baisser le volume du composant « Audio Source » afin de bien entendre tous les effets sonores du jeu.
> - Vous pouvez cocher l'option « Loop » du composant « Audio Source » afin de rejouer en boucle la musique choisie.

Nous allons maintenant ajouter des effets sonores aux actions du personnage du jeu.

16. Dans le script de gestion de votre personnage, rajoutez deux attributs de type « AudioClip » afin de stocker des effets sonores pour les animations de « sauter » et de « faire une collision avec un obstacle ».

> Le choix de l'effet sonore associé à chaque état doit pouvoir se faire via l'onglet « Inspector ».

17. Affectez un clip audio à chaque attribut créé.

> /Assets/Course Library/Sound/

18. Ajoutez un composant « Audio Source » au personnage du jeu.

19. Ajoutez un attribut à votre script de « contrôle du personnage » afin de garder une référence à l'objet « AudioSource » du personnage.
20. Initialisez l'attribut de type « AudioSource » avec l'objet « AudioSource » du personnage.
21. Utilisez la méthode « PlayOneShot » disponible via l'objet de type « AudioSource » afin de déclencher des effets sonores lorsque le personnage saute ou fait une collision avec un obstacle.

Considérations finales

Les exercices de ce TD ont été adaptée de l'unité 3 du Parcours Unity Learn - Junior Programmer. N'hésitez pas à visiter leur formation en ligne afin d'avoir accès au cours complet, ainsi qu'à avancer en autonomie sur les sujets que n'ont pas été traité en cours.

Références

- https://docs.unity3d.com/ScriptReference/GameObject.GetComponent.html
- https://learn.unity.com/tutorial/lesson-3-1-jump-force?uv=2021.3&pathwayId=5f7e17e1edbc2a5ec21a20af&missionId=5f7648a4edbc2a5578eb67df&projectId=5cf9639bedbc2a2b1fe1e848#5ce35aa5edbc2a29e31b3c71