TP Découverte Unity : Mario !

Table des matières

1 Présentation

1.1 But de l'activité

L'objectif est de vous faire découvrir l'ambiance d'une séance de TP d'informatique à l'EPITA. Ici, dans les salles machines, nous sommes loin (très loin) de l'ambiance du CDI du lycée (et encore plus de celle d'une salle de classe). Ici on s'entraide, on discute (pas toujours du sujet... oups... (ne dites rien à Lionel svp)). Bref, on ne va pas vous manger si vous osez parler, au contraire.

Les TPs de programmation sont toujours encadrés par des étudiants des années supérieures. Ils sont là pour nous assister et répondre à nos éventuelles questions. Aujourd'hui, vos assistants, c'est nous ! Alors surtout n'hésitez pas à nous poser des questions, nous demander d'éclaircir un point du sujet ou juste nous demander un peu (ou beaucoup) d'aide.

1.2 Unity

Aujourd'hui, nous allons apprendre à utiliser Unity et créer un jeu du style Mario Bros.

Pourquoi Unity ?

Parce qu'au deuxième semestre de l'année INFO SUP (la première année à l'EPITA) vous devrez réaliser un jeu ou une application en C# ou en CAML (deux langages de programmation que vous maîtriserez à la fin de l'année). Le choix le plus courant est de réaliser un jeu en C# avec Unity.

Mais c'est quoi Unity ?

Unity est un moteur de jeu. C'est un logiciel qui réalise des calculs de géométrie et de physique primordiaux au fonctionnement d'un jeu. Ils permettent de créer un jeu vidéo sans avoir à réinventer la roue à chaque fois. Ils nous simplifient la vie, notamment grâce à un simulateur en temps réel de ce que l'on est en train de créer. Bien sûr, Unity n'est pas le seul moteur de jeux vidéo, on peut également citer Unreal Engine, Blender Game Engine, et bien d'autres [a].

a. La liste complète : ICI

1.3 Découpage du sujet

En programmation, tout le monde ne progresse pas à la même vitesse. Pas de panique, à l'EPITA les cours d'informatique partent de zéro, vous n'avez besoin d'aucune connaissance en programmation. C'est pourquoi les sujets sont souvent "découpés" en deux parties : la première contient les compétences à maîtriser, et la deuxième qui permet d'aller plus loin avec des bonus. Ainsi, du néophyte au plus confirmé, vous passez tous un bon moment en vous challengeant pour toujours aller plus loin.

Ce sujet suit le même découpage. La première partie, le "véritable sujet", vous permet de découvrir Unity. La seconde, les bonus, est facultative et permettra aux plus rapides d'aller plus loin ou de continuer un autre jour chez vous.

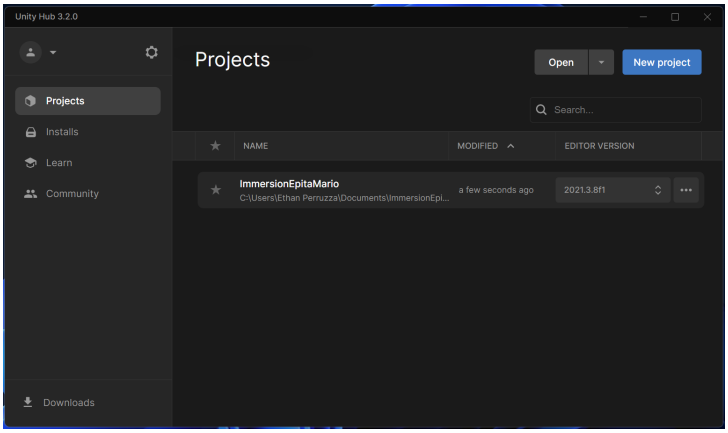
2 It's a-me, Mario !

En savoir plus

Les encadrés bleus "Pour aller plus loin" et "En savoir plus" que vous verrez tout le long du sujet contiennent des indications supplémentaires, à lire et à suivre par curiosité ou si vous vous sentez déjà à l'aise avec Unity.

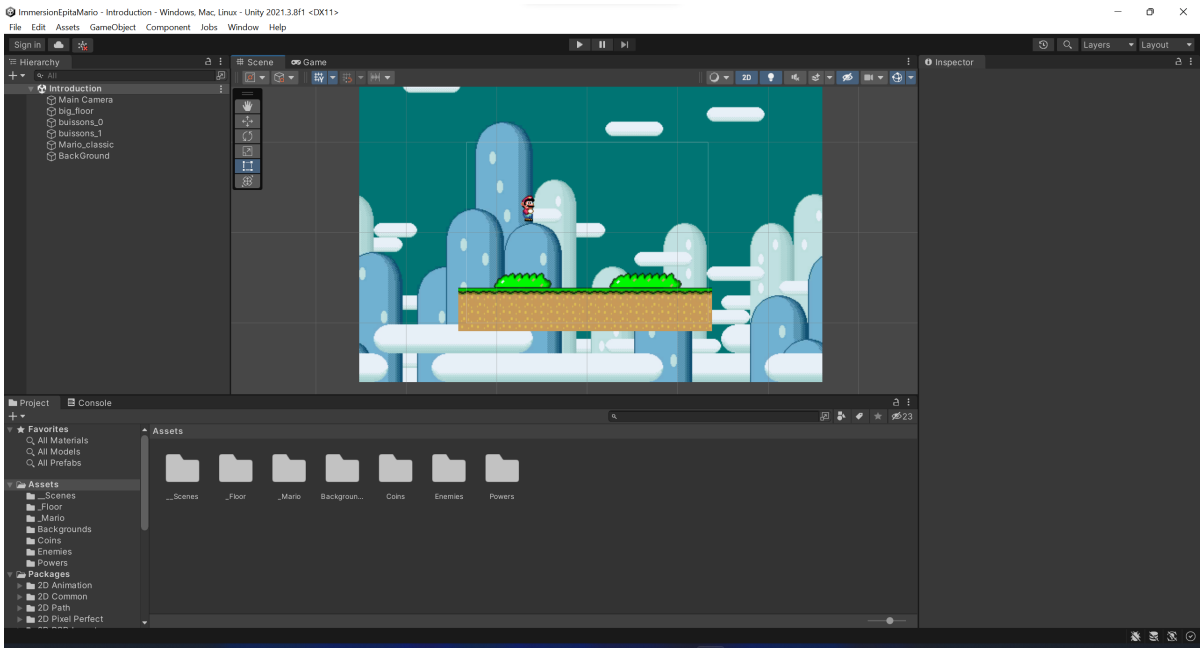
2.1 Unity Hub : ouverture du projet

Vous avez devant vous Unity Hub. C'est ici que sont résumés tous vos projets commencés. A partir de cette fenêtre, on peut créer un nouveau projet Unity ou ouvrir un projet existant. Pour aujourd'hui, il suffira d'ouvrir le projet "ImmersionEpitaMario" que nous avons créé pour vous.



2.2 Présentation de l'interface

Très bien, nous voici maintenant sur le logiciel Unity. On peut vite se sentir perdu mais pas de panique, on va commencer par découvrir les différentes fenêtres qui se sont ouvertes.



Les onglets et fenêtres :

Scene : Le plus grand, au centre, c'est dans cet onglet que le plus gros du travail va se passer. Il contient tous les objets qui composent notre jeu. On peut les déplacer, en ajouter ou en supprimer.

Hierarchy : Ici, la fenêtre sur la gauche, sont listés tous les objets présents dans la scène.

Project : Dans la fenêtre du bas, on retrouve tous les fichiers associés au projet. Le dossier **Asset** est crée automatiquement, vide au départ. Il contient toutes les images, scènes et fichiers de code que l'on utilise pour notre jeu.

Inspector : A droite, l'un des onglets les plus importants car il nous permettra de visualiser et modifier les propriétés d'un objet (sa position dans l'espace, son aspect, sa rigidité, sa gravité, etc.).

Game : Caché derrière l'onglet **Scene**, il permet de visualiser en temps réel ce que l'utilisateur verra lorsqu'il lancera le jeu. Plus tard, il nous permettra aussi de faire des simulations en temps réel du jeu.

Bienvenue chez vous

Tous les onglets sont modulables pour qu'ils puissent convenir à votre utilisation. Essayez de déplacer l'onglet **Game** à côté de l'onglet **Project**. Déposez-le par exemple entre les onglets **Project** et **Inspector** pour qu'il soit toujours visible.

En savoir plus

Il existe d'autres onglets que l'on peut ajouter et qui sont très utiles lorsque vous souhaiterez complexifier vos projets, aujourd'hui nous voyons les bases. Le dernier onglet visible, **Console**, permet par exemple de consulter les erreurs de notre code pendant une simulation.

2.3 Navigation dans la scène

Chose importante lorsque le projet s'agrandira : savoir se déplacer dans la scène. Pour cela, comme nous serons en 2D pour ce projet, ce n'est pas sorcier. Il suffit de faire un **Clic Droit Enfoncé** dans l'onglet **Scene** et de bouger la souris pour se déplacer dans la scène. Le **Simple Clic Gauche** permet de sélectionner un ou plusieurs objets déjà insérés dans la scène.

Pour aller plus loin : et en 3D ?

C'est un chouïa plus compliqué. Déjà, passons en 3D. Pour ce faire, cliquez sur le bouton bleu **2D** dans l'onglet **Scene**. Pour repasser en 2D, il suffira de rappuyer sur le même bouton.

Un Simple Clic permet de sélectionner un ou plusieurs objets déjà insérés dans la scène.

Un Clic Droit Enfoncé change l'orientation de la vue.

Un Clic Molette permet de se déplacer.

Un Roulement de Molette permet de zoomer ou dézoomer.

3 Let's-a go, little guys !

A vérifier avant de commencer

Pour cette section, vous devez vous trouver dans la scène **Introduction**.
Le nom de la scène sur laquelle vous travaillez est écrit en haut dans l'onglet **Hierarchy**.
Si vous n'êtes pas dans la bonne scène, allez dans le dossier "Assets/___Scenes" dans l'onglet **Project**, puis double-cliquez sur **Introduction**.

Comme vous pouvez le voir, un certain nombre d'éléments ont déjà été placés pour vous simplifier le travail. Alors commençons par lancer la simulation ! Rendez-vous dans l'onglet **Game**. Appuyez sur le bouton **Play** (en haut de l'écran).

Comment savoir si la simulation est en cours ou non ?

Quand le bouton **Play** est coloré en `bleu` : c'est que la simulation est en cours.
S'il est `gris` : la simulation est arrêtée.

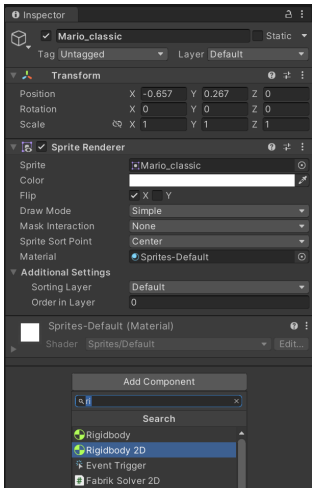
Normalement, rien ne doit se passer. Le jeu n'est qu'une image fixe avec laquelle vous ne pouvez pas interagir.

Important !

Il est crucial de bien stopper la simulation avant de faire des modifications, sinon elles disparaîtront quand vous aurez quitté la simulation.

3.1 Wahoo ! I fly !

Commençons par le plus dérangeant : Mario vole ?! En effet, pour qu'un objet tombe il faut lui attribuer une propriété de gravité. Dans Unity, les propriétés sont appelés des "Component". Retournez dans l'onglet **Scene**, cliquez sur Mario. Tout un tas d'informations apparaissent dans l'onglet **Inspector** (le nom de l'objet, sa position, etc.). Cliquez sur le bouton **Add Component** et recherchez et sélectionnez **RigidBody 2D**.



En savoir plus : pourquoi "Rigidbody 2D" ?

Le plus simple d'abord : 2D parce que nous travaillons en 2D. Il y en a deux parce que Unity permet également de faire de la 3D. La version 2D nécessite moins de ressources puisqu'il y a une dimension en moins à gérer.
Le component "Rigidbody" indique à Unity que cet objet est soumis aux lois de la physique. Cela inclut la gravité, mais également le fait d'être poussé si un objet ou un joueur nous fonce dessus, etc.

Relancez la simulation ! Et remarquez que Mario est bien soumis à la gravité.

3.2 Buh-bye ! I'm falling through the floor !

Mario est soumis à la gravité certes, mais il tombe à travers le sol ! C'est parce que pour Unity notre sol n'est pas solide, on peut passer à travers !

Ajoutez le component "**Box Collider 2D**" au sol.

En savoir plus : pourquoi "Box Collider 2D" ?

Cela permet de dire à Unity que cet objet est solide et que l'on ne peut pas passer à travers.
La taille du "collider [a]" peut être modifiée.
Si vous tapez "collider" dans la barre de recherche des components, vous remarquerez qu'il y a plusieurs types de collider. En réalité la seule chose qui change est leur forme, ce qui permet de mieux s'adapter au contexte.

a. surface dans laquelle un autre objet solide ne peut pénétrer

Si vous relancez la simulation à cette étape, vous remarquerez que Mario tombe toujours à travers le sol. En effet, bien que le sol soit maintenant solide et que tous les autres objets solides ne peuvent le pénétrer, nous n'avons jamais dit à Unity que Mario est solide. Pour cela, ajoutons un collider à Mario.

Conseil

Pour épouser au mieux les contours d'un personnage il est courant d'utiliser un **Capsule Collider 2D**. Le collider aura alors la forme d'une capsule, ce qui semble plus naturel qu'un rectangle.

Relancez la simulation. Mario n'est plus un fantôme !

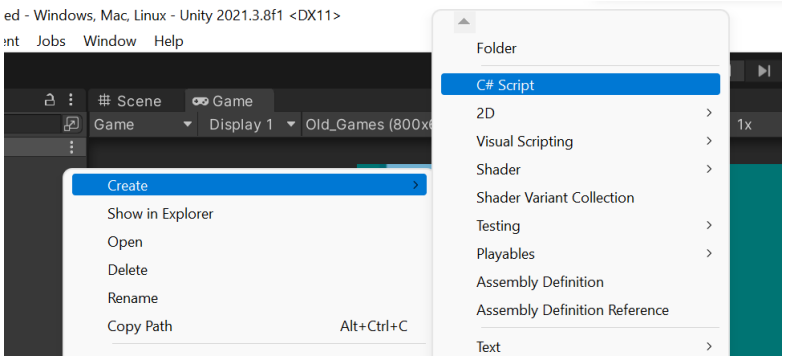
3.3 Mamma Mia ! I can't move !

Certes, Mario respecte enfin les lois fondamentales de la physique mais il ne peut toujours pas se déplacer... Alors il est grand temps de s'y mettre ! Nous voulons pouvoir faire bouger Mario horizontalement en utilisant les flèches du clavier ou les touches A et D.

3.3.1 Code

Dès lors que l'on souhaite faire quelque chose de plus spécifique sur Unity, il faut coder un peu pour créer nos propres "components". Ici, nous voulons créer une "component" qui dirige le personnage en fonction des touches pressées.

Pour cela nous allons commencer par créer un fichier script. Rendez-vous dans le dossier "_Mario/Scripts", cliquez droit, survolez jusqu'à "Create" et appuyer sur "C# Script". Nommez-le "MarioMove" (le nom du fichier n'a aucune importance, mais nous y référerons sous ce nom).



Attention : ne changez plus le nom d'un fichier une fois celui-ci validé

Il faut éviter de renommer un fichier code dans Unity. En effet lors de la création du fichier, Unity référence le code sous le premier nom que vous donnez au fichier. Ainsi, si vous changez le nom du fichier, il ne trouvera plus le code.
Évidemment, on peut toujours changer cette référence en modifiant une ou deux lignes dans le fichier mais nous ne nous attarderons pas là-dessus.

Double-cliquez sur le fichier pour l'ouvrir. Un nouveau logiciel s'ouvre : Visual Studio. C'est le logiciel par défaut pour coder en C# . On se concentrera sur le plus pertinent, ainsi nous n'étudierons pas l'interface du logiciel. Ce code doit apparaître après la création du fichier :

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class MarioMove : MonoBehaviour
6   {
7       // Start is called before the first frame update
8       void Start()
9       {
10
11      }
12
13      // Update is called once per frame
14      void Update()
15      {
16
17      }
18  }
```

C'est ce qui apparaît à chaque fois que l'on crée un fichier code.

— `class MarioMove` indique que tout le code écrit à l'intérieur des accolades sera relié à le component MarioMove.
— `void Start(){}` est une méthode : c'est une fonction qui sera appelée par Unity à un moment précis. La méthode Start est appelée quand on appuie sur Play. Ainsi, le code écrit dans cette fonction sera exécuté une seule fois.
— `void Update(){}` est également une méthode. Elle sera cependant appelée plusieurs fois : à chaque rafraîchissement d'image, donc plusieurs fois par secondes. Ainsi, le code écrit dans cette fonction sera donc exécuté très souvent.

En savoir plus

— `public` indique que ce code est "public" : on peut accéder à une fonction écrite dans ce fichier depuis n'importe quel autre fichier (cela ne serait pas le cas avec `private`, par exemple).
— `: MonoBehaviour` indique que ce fichier code hérite du fichier code `MonoBehaviour`. C'est un fichier créé par Unity qui permet d'accéder à toutes les fonctions spécifiques à Unity.
— Il existe d'autres méthodes qui sont appelées à d'autres moments et/ou dans d'autres conditions par Unity. On aura l'occasion d'en voir d'autres dans les bonus.

On écrira donc notre code dans la méthode "Update" puisque l'on souhaite savoir à tout moment si le joueur appuie ou non sur une touche du clavier.

Écrivons ces quelques lignes :

```
1  void Update()
2  {
3      Vector3 motion = new Vector3(Input.GetAxis("Horizontal"), 0f, 0f);
4      transform.position = transform.position + motion * Time.deltaTime;
5  }
```

Attention !

Il est écrit 0f ('zéro'f), si vous écrivez Of ('o'f), ça ne fonctionnera pas. En réalité, le "f" indique à Unity que vous venons de lui indiquer un nombre flottant (un nombre à virgule), cela reviendrait à écrire "0.0".

Pour aller plus loin : explications du code

Chaque objet dans Unity possède une propriété `transform` qui définit leur positionnement, rotation, etc. dans l'espace. Leur position (stockée dans `transform.position`) est un `Vector3`, un vecteur tridimensionnel de coordonnées (x,y,z). Pour l'instant on souhaite déplacer Mario uniquement sur l'axe horizontal (l'axe des x). La fonction `Input.GetAxis("Horizontal")` renvoie une valeur entre -1 et 1 en fonction du déplacement souhaité par le joueur.

Info : quand renvoie-t-elle -1 ou 1 ?

La valeur tend vers -1 si l'on appuie sur une touche reliée au déplacement horizontal gauche (par défaut, ici "A" ou "flèche gauche"). Ou vers 1 si l'on appuie sur une touche reliée au déplacement horizontal droit (par défaut, ici "D" ou "flèche droite"). Sinon, elle renvoie 0.

Ainsi lorsque l'on ajoute ce vecteur à celui qui définit la position de l'objet (`transform.position + motion`), on obtient un nouveau vecteur avec une nouvelle position. Cependant, pour que l'expérience soit plus fluide, on va d'abord multiplier le vecteur "motion" par le temps qu'il nous a fallu pour réaliser tous les calculs avant de l'ajouter à l'ancienne position (`transform.position + (motion * Time.deltaTime)`).
Enfin, on stocke le résultat (la nouvelle position) dans `transform.position` pour modifier la position de l'objet dans la scène.

En savoir plus : fact jeux vidéo

C'est pour cette raison que lorsqu'un jeu lag [a], vous avez l'impression de vous télé-porter. Mais ne vous méprenez pas, c'est au final une "bonne chose" car sinon vous n'avanceriez quasiment pas d'image en image.

a. lorsque le temps entre deux images est conséquent et donc visible à l'oeil nu

Mais que sont donc tous ces " ;" ?

Le cauchemar de tous les développeurs bien sûr !
En effet, en C# (comme dans certains autres langages de programmation), il ne faut jamais oublier de les mettre à la fin de chaque instruction. Sinon, c'est une erreur et le code ne fonctionnera pas.
Vous n'imaginez pas le nombre de cheveux qui sont arrachés chaque année à cause de ce caractère si insignifiant !
Heureusement, les logiciels vous indiquent quand vous en oubliez un ;)

Bravo ! Vous en avez fini avec le code (pour l'instant...) !

Sauvegarder le fichier

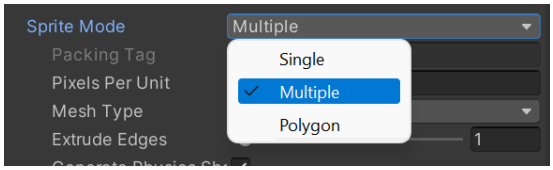
Pour que les lignes de code soient prises en compte, il faut absolument sauvegarder le fichier de code avant de retourner sur Unity, et ce chaque fois que vous voudrez appliquer des changements à la scène ! Pour ce faire, vous pouvez aller dans "File" dans la barre d'outils, puis cliquer sur "Save", ou alors simplement utiliser le raccourci clavier "Ctrl+S".

Il ne nous reste plus qu'à ajouter notre nouveau component personnalisé à notre personnage. Pour cela, de retour sur Unity, cliquez sur Mario et ajoutez-lui le component "MarioMove" comme nous l'avons fait précédemment avec "Rigidbody 2D" par exemple.

Testez, et magie ! Mario peut se déplacer !

3.3.2 Animation

Vous me direz que sa démarche est quelque peu étrange... Vous avez raison ! Remédions à cela !
Pour l'animer, nous allons avoir besoin de plusieurs images dans différentes positions. Ici, nous en utiliserons 3. Ces dernières sont dans le dossier "_Mario" dans une grande planche avec toutes les images de Mario, nous allons donc devoir "découper" les images que l'on désire utiliser.
Pour cela, rien de très compliqué, cliquez une fois sur l'image "ALL_Marios" (avec un 's'). Il va falloir indiquer à Unity que cette image contient plusieurs sprites [1]. Dans l'onglet Inspector, changez **Sprite Mode** de "Single" à "Multiple".

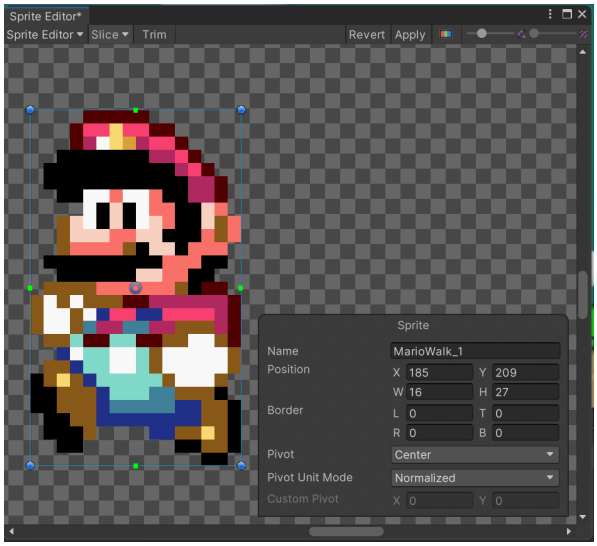


Maintenant, découpons les images. Cliquez sur le bouton **Sprite Editor** (situé un peu en dessous). Une nouvelle fenêtre s'ouvre : **Sprite Editor**. Les contrôles pour se déplacer sont les mêmes que pour l'onglet **Scene**. Zoomez sur la partie "Super Mario Walk" (les grands Marios) et sélectionnez la première sprite (cliquez gauche et restez appuyé pour former un rectangle de la taille appropriée). Une fois le clic relâché, un petit onglet s'ouvre, changez le nom du sprite pour : "MarioWalk_1". Ça y est, vous avez découpé la première image !

Comment changer le découpage ?

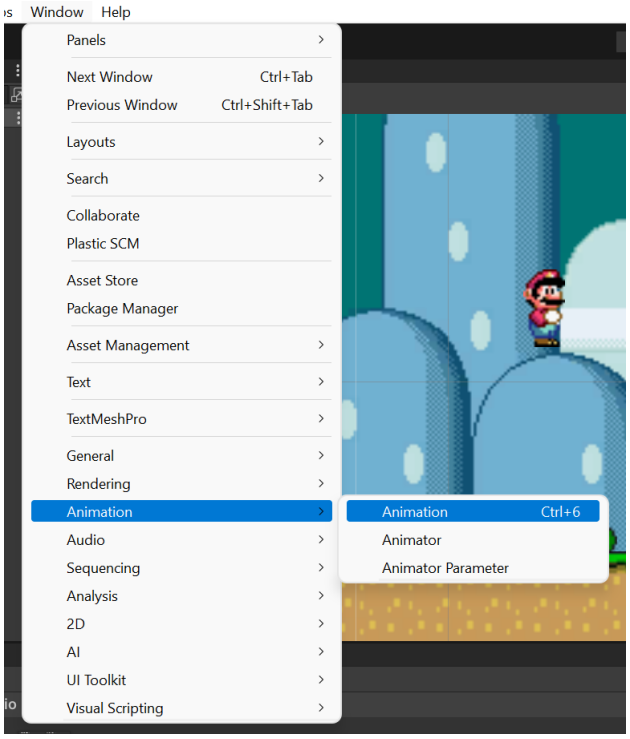
Il est important de réaliser un "bon découpage" en épousant au mieux les contours de l'image pour éviter des problèmes de collisions indésirées.
Pour modifier la taille du découpage, on utilisera les points bleus.

1. Un sprite, ou lutin, est dans le jeu vidéo un élément graphique qui peut se déplacer sur l'écran

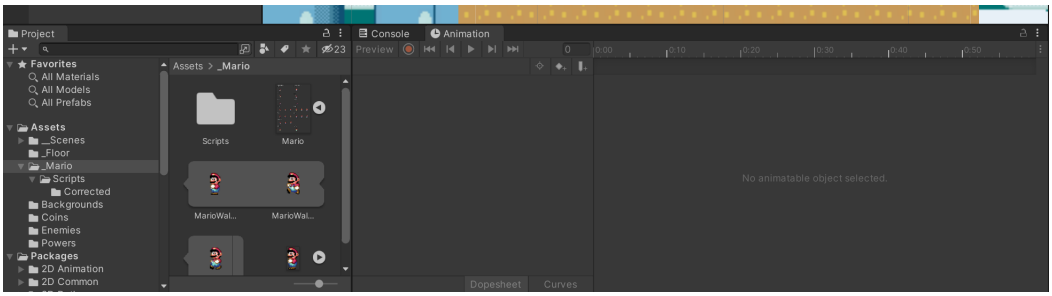


Faites de même avec les deux images suivantes et nommez-les respectivement "MarioWalk_2" et "MarioWalk_0". On nomme la dernière "0" car c'est la position initiale de Mario : il démarre de cette position et finira dans cette position. Une fois cette étape réalisée, cliquez sur **Apply** en haut à droite et vous pouvez fermer la fenêtre. Maintenant, si vous cliquez sur le petit menu déroulant de l'image, vous pouvez voir les 3 sprites que l'on vient de découper.

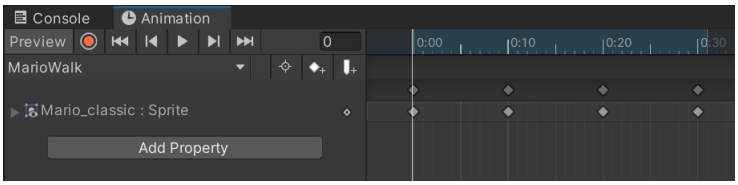
Passons à l'animation. Pour cela, nous allons ouvrir une nouvelle fenêtre : dans la barre d'outils en haut, allez dans **Window/Animation**, puis cliquez sur **Animation**.



Placez-la avec les fenêtres **Project** et **Console**, puis déplacez la fenêtre **Project** sur la gauche pour pouvoir accéder aux deux fenêtres en même temps. De la sorte :



Sélectionnez Mario dans l'onglet **Scene**. Dans l'onglet **Animation**, cliquez sur **Create** pour créer une nouvelle animation reliée à Mario. Enregistrez l'animation dans "_Mario/Animations" et nommez-la "MarioWalk".
Glissez/déposez la première image, "MarioWalk_0", à l'indication de temps 0 :00. Faites de même avec les deux autres images en les plaçant respectivement à 0 :10 et 0 :20. Enfin, reglissez/déposez la première image mais cette fois-ci à 0 :30. Vous devriez obtenir ce résultat :



Comment visualiser notre animation ?

Appuyez sur la touche espace pour lancer l'animation et à nouveau pour l'arrêter
Bien sûr, si vous n'êtes pas satisfaits de la vitesse de l'animation, vous pouvez rapprocher (ou éloigner) les losanges pour accélérer (ou décélérer) la vitesse de transition des images.

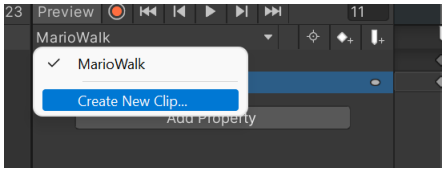
Pourquoi faut-il mettre deux fois la première image ?

Il faut positionner la première image au début est à la fin car l'animation va faire une boucle (une fois arrivée à la fin, elle se relance en recommençant au début). Ainsi, si la dernière image est différente de la première, à peine affichée, elle sera remplacée, résultant en un affichage à l'apparence buguée.
Vous pouvez vous amuser à tester en supprimant le dernier losange et en relançant l'animation.

En savoir plus sur les animations

Vous avez sûrement remarqué le bouton "Add Property" dans l'onglet **Animation**. Nous nous sommes contentés de changer l'apparence de Mario au cours de l'animation mais Unity nous permet également de le faire se déplacer par exemple. Pour le déplacer, nous aurions modifié une autre propriété de l'animation (la position). Cela peut notamment servir à animer une plateforme non-contrôlée par l'utilisateur.
Les petits losanges sont appelés des "clefs" d'animation. Ils indiquent à Unity, qu'à l'instant t, la propriété de l'objet doit se trouver à l'état x contenu dans la clef (dans notre cas, à 0 :10 seconde, l'image affichée doit être "MarioWalk_1").
Si nous avions choisi d'animer un déplacement, la clef contiendrait des informations de position, Unity saurait alors qu'à cet instant t, l'objet doit être à la position (x,y,z) contenue dans la clef : il ferait donc les calculs nécessaires pour déplacer l'objet à vitesse constante entre les deux clefs de position.

Créons une seconde animation (voir illustration ci-dessous) que l'on nommera "MarioImmobile" et dont la seule image sera "**Mario_Classic**" (vous comprendrez son utilité plus tard). Vous pouvez la placer à 0 :00.

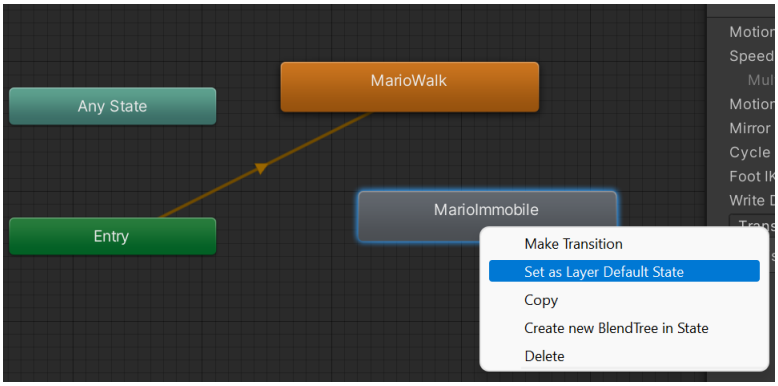


Automatiquement, dans le dossier "_Mario/Animations" s'est créé un **Animator Controller** [2] nommé : **Mario_Classic**. Double-cliquez dessus pour l'ouvrir. Un nouvel onglet apparaît (et oui ! encore un !), l'onglet **Animator**.

Conseil Navigation :

Pour vous déplacer dans cette fenêtre, restez appuyé sur la molette et bougez la souris.

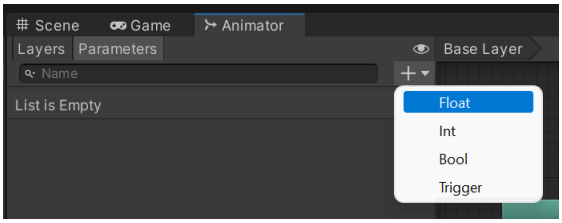
Commençons par définir l'animation **MarioImmobile** comme animation par défaut (l'état d'origine du personnage). Pour cela, cliquez-droit sur la boite et sélectionnez **Set as Layer Default State**.



2. Un Animator Controller vous permet d'organiser et de maintenir un ensemble de clips d'animation et de transitions d'animation associées pour un personnage ou un objet

Recliquez-droit sur cette même boite et sélectionnez cette fois **Make Transition** (pour créer une nouvelle transition) puis cliquez sur la boite **MarioWalk**. Cliquez sur la flèche nouvellement créée, puis dans l'onglet **Inspector**, décochez la case **Has Exit Time**. Lorsque cette case est cochée, Unity finira toujours l'animation avant de laisser le personnage retourner dans son état initial (ainsi, si l'on arrête d'avancer et que la case est cochée l'animation de marche continue alors qu'on ne bouge plus). Dans le menu déroulant **Settings** (juste en dessous), mettez **Transition Duration** à 0 (pour les mêmes raisons).
Faisons la même chose dans l'autre sens pour pouvoir passer de l'état marche à l'état immobile. Créez la flèche dans l'autre sens et modifiez les paramètres.

Nous allons maintenant créer les conditions pour basculer d'un état à l'autre. Pour cela, on aura besoin de la vitesse de déplacement de Mario. On crée donc une variable qui la contiendra. En haut à gauche de la fenêtre cliquez sur **Parameters** puis sur le + et enfin **float** [3] (on utilise un float car la vitesse sera comprise entre 0 et 1). Nommez la variable "Speed".



Cliquez sur la flèche allant de **MarioImmobile** à **MarioWalk**. Dans **Conditions**, cliquez sur + et modifiez la valeur de la condition pour 0.1. Pour l'autre flèche, faites l'inverse (la condition est : que la vitesse soit inférieure à 0.1).

Pourquoi 0.1 et pas 0 ?

Parce que si l'on utilise 0, alors l'animation ne s'arrêtera jamais, car la vitesse serait égale à 0 et non inférieure à 0.

Il ne reste plus qu'à modifier cette valeur. Retournons dans notre fichier code. Commençons par créer une variable qui va stocker le lien entre l'Animator et notre code (pour que l'on puisse modifier les valeurs de l'animator depuis notre code). On placera notre code juste après l'ouverture des accolades de la classe.

```
1 public class MarioMove : MonoBehaviour
2 {
3     public Animator MarioAnimator;
4
5     [...]
6 }
```

3. Un float est un nombre à virgule

En savoir plus : explications du code

- `Animator` est le type de la variable.
  Pour imager : imaginez un bocal en verre vide. Vous devez décider de ce que vous allez mettre dans le bocal mais vous ne pourrez plus jamais changer d'avis (c'est la règle). Sur ce bocal vous faites une marque au fer rouge indiquant que ce bocal ne contiendra jamais autre chose que de la farine.
  Ici, on a décidé que notre bocal ne contiendrait jamais autre chose qu'un Animator.
- `MarioAnimator` est le nom de la variable.
  Reprenons notre bocal. En réalité nous avons au fil des ans créé plusieurs bocaux de farine : 1 pour les crêpes, 1 autre pour les gâteaux, etc. Pour les différencier on a collé sur chaque bocal une étiquette avec le nom que l'on a donné au bocal. Celui pour les crêpes s'appelle Chandler_Bing et celui pour les gâteaux, Marbré_Au_Chocolat.
  Ici, on a appelé notre bocal d'Animator : MarioAnimator.

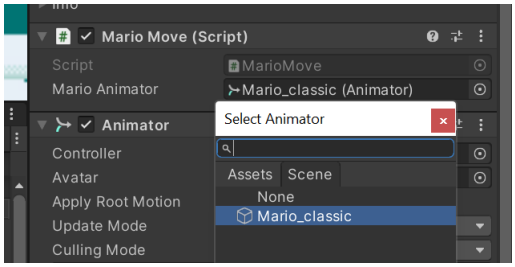
Ensuite, on va venir ajouter une ligne de code à la suite de notre déplacement pour changer la valeur de `Speed`.

```
1  void Update()
2  {
3      Vector3 motion = new Vector3(Input.GetAxis("Horizontal"), 0f, 0f);
4      transform.position = transform.position + motion * Time.deltaTime;
5
6      MarioAnimator.SetFloat("Speed", Mathf.Abs(motion.x));
7  }
```

En savoir plus : explications du code

Ici on vient changer la valeur du float `Speed` en lui attribuant `Mathf.Abs(motion.x)`, qui est la valeur absolue de x (qui est compris entre -1 et 1). `Mathf.Abs()` est une fonction déjà définie en C# qui renvoie la valeur absolue du nombre entre les parenthèses.

Retournons dans l'onglet **Scene**, cliquez sur Mario et dans le component du script est apparue une variable non-attribuée ("MarioAnimator"). Cliquez sur la cible à côté puis sélectionnez **Mario_classic**.



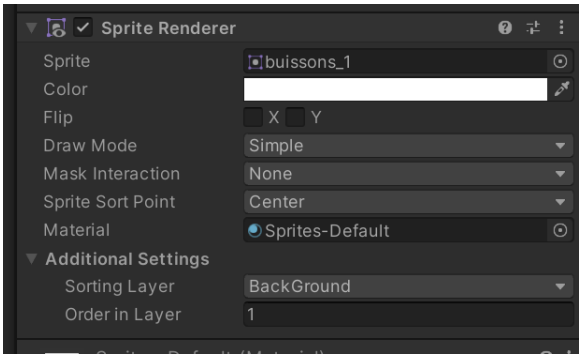
Testez, Mario marche !

3.4 Let's make it beautiful

Il nous reste encore quelques détails à régler : Mario marche mais derrière les buissons... Et puis on souhaite qu'il se retourne lorsqu'il va vers la gauche.

3.4.1 I'm hidden !

Pour régler ce premier problème il suffit de changer la couche d'affichage des buissons pour une couche inférieure. Cliquez sur un des buissons. Dans le component **Sprite Renderer** changez la **Sorting Layer** de **Default** à **Background**. Pour éviter un conflit d'affichage avec le fond qui est à l'ordre 0 du background. Changez la valeur d'ordre de 0 à 1 pour que le buisson soit toujours devant le fond.



3.4.2 Watch where you go !

Quand Mario va vers la gauche il faut que l'on retourne l'image de Mario sur l'axe x.
Pour cela, il va falloir toucher à nouveau un petit peu au code. On commence tout simplement par créer un lien entre le code et le **Sprite Renderer** de Mario (de la même façon que nous avons créé un lien entre l'Animator et le code pour pouvoir modifier des valeurs dans l'Animator).

```
1  public class MarioMove : MonoBehaviour
2  {
3      public Animator MarioAnimator;
4
5      public SpriteRenderer MarioSpriteRenderer;
6
7      [...]
8  }
```

Ensuite, on ajoute après notre déplacement un bout de code qui change l'orientation de Mario en fonction de son sens de déplacement.

```
1  void Update()
2  {
3      [...]
4
5      if (motion.x > 0.1)
6      {
7          MarioSpriteRenderer.flipX = true;
8      }
9      else if (motion.x < -0.1)
10     {
11         MarioSpriteRenderer.flipX = false;
12     }
13 }
```

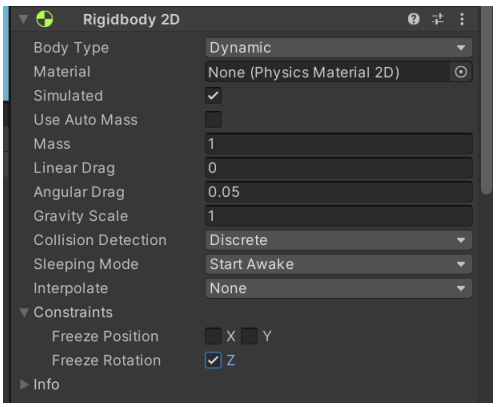
Si le mouvement est positif, alors on va vers la droite et si le mouvement est négatif, alors on va vers la gauche. Ainsi, on oriente le sprite en fonction du sens du déplacement. Cependant, si le mouvement est nul alors on ne fait rien car on ne souhaite pas que Mario se retourne toujours d'un côté.

Pour finir, de la même manière que précédemment pour l'Animator, on retourne dans la scène et on renseigne la valeur de la nouvelle variable vide en cliquant sur la cible. Encore une fois, on sélectionne **Mario_classic**.

Testez, et ça y est, notre Mario est parfait !

3.4.3 Mario in Wonderland

Si vous faites tomber Mario dans le vide, vous remarquerez que tout le niveau se met à basculer. En réalité, c'est Mario qui pique du nez vers le sol, la caméra ne fait que le suivre. Pour éviter ce problème, dans le component **RigidBody 2D** de Mario (dans la fenêtre **Inspector**), sous le menu déroulant **Constraints** : On coche la case **Freeze Rotation Z**. Qui empêchera Mario de tomber en avant ou en arrière.



3.5 Yiiiiipeee ! I want to jump !

Il est bien beau notre Mario, mais un Mario qui ne saute pas c'est pas très rigolo... remédions à cela ! On commence fort avec un chouïa de code dans notre fichier :

```
1  void Update()
2  {
3      if (Input.GetKey(KeyCode.Space) &&
4          GetComponent<Rigidbody2D>().velocity.y == 0)
5      {
6          GetComponent<Rigidbody2D>().velocity = new Vector2(0f, 3f);
7      }
8
9      [...]
10 }
```

On rajoute une nouvelle condition dans notre méthode `Update()`. Si l'on presse la barre d'espace : `Input.GetKey(KeyCode.Space)`, et que la vitesse sur l'axe y de Mario est nulle (il n'est ni déjà en train de sauter, ni de tomber : `GetComponent<Rigidbody2D>().velocity.y == 0`). Alors on le fait sauter : `GetComponent<Rigidbody2D>().velocity = new Vector2(0f, 3f);` en lui modifiant sa vitesse pour lui donner une impulsion verticale.

En savoir plus : les conditions logiques

Vous avez pu remarquer que lorsque l'on utilise un *if*, on lui adjoint une condition entre parenthèse (si le mouvement est supérieur à 0.1 ; si l'on presse la touche espace ; etc.).
Si cette condition est vraie, alors le code dans les {} s'exécute, dans le cas contraire, on saute tout ce qu'il y a dans les {} sans les exécuter.
On a parfois besoin de vérifier plusieurs conditions en même temps. Pour cela, on utilise des "OU" ou des "ET" (en fonction des cas). En C# , le "OU" s'écrit "`||`" et le "ET" s'écrit "`&&`".
Comme en mathématiques, la priorité des parenthèse s'applique. Ainsi pour : "`condition_1 && (condition_2 || condition_3)`" il suffit que `condition_1` et `condition_2` soit vraies ou que `condition_1` et `condition_3` soit vraies.

Challenge

Sur ce même principe, faites en sorte que l'on puisse également sauter en appuyant sur la "flèche vers le haut" du clavier `KeyCode.UpArrow`.

Vous pouvez tester, Mario saute déjà !

3.6 Build and run

Bravo ! Vous êtes arrivés à la dernière section du sujet ! Nous avons conçu notre jeu, il faut maintenant le compiler : on appelle ça un Build. Cela va créer un exécutable que vous pourrez lancer depuis n'importe quel ordinateur (même si ce dernier ne possède pas Unity, donc comme un "vrai" jeu).

Des Build partout

Vous pouvez faire un premier Build maintenant, mais si vous comptez encore améliorer votre jeu avec les bonus, vous devrez refaire un Build complet plus tard pour avoir la dernière version de votre jeu. Vous êtes donc libre de sauter cette partie maintenant et d'y revenir tout à la fin !

Dans la barre d'outils sous **File**, sélectionnez **Build Settings** puis cliquez sur **Build and Run** (en bas à droite de la fenêtre) et choisissez l'endroit de stockage du jeu (mettez-le sur votre clef USB ;)).

Comment lancer le jeu ?

Le jeu se lance automatiquement si vous avez appuyé sur **Build and Run** une fois la compilation terminée.
Pour le relancer, depuis n'importe quel autre ordinateur Windows, il suffira de retourner dans vos fichiers à l'emplacement choisi et de cliquer sur l'exécutable **ImmersionEpita-Mario.exe**.

En savoir plus : la fenêtre Build Settings

- Tout en haut de la fenêtre se trouvent les scènes du jeu. Pour que votre scène soit compilée dans le jeu il faut que la case soit cochée.
- La première scène à être lancée est la scène 0 (le numéro est à droite). Vous pouvez changer cet ordre. Pour changer de scène, il faut nécessairement que cela soit prévu dans votre code avec une touche ou un portail qui lance une autre scène.
- Pour ajouter une scène non présente dans la liste, il faut ouvrir la scène en question puis cliquer sur le bouton **Add Open Scenes** dans l'onglet **Build Settings**.
- Vous pouvez choisir pour quelle plateforme (Windows, PS4, Xbox, Mobile, etc.) vous souhaitez compiler votre jeu. Il faut cependant que lors de l'installation d'Unity vous ayez choisi de pouvoir exporter votre jeu vers cette plateforme.

4 Bonus

Les bonus sont facultatifs, mais ils ajoutent du piment à votre jeu, donc nous ne pouvons que vous conseiller de les faire ! Ils peuvent être réalisés dans n'importe quel ordre (sauf indication contraire). Ils sont néanmoins rangés dans un ordre logique.

4.1 I want to jump like a princess !

Mario saute, c'est bien, mais il ne change pas de position quand il est en l'air... Modifions cela !
Cette partie est purement esthétique, elle peut être sautée. Elle reste tout de même intéressante grâce à l'utilisation de booléens.

En savoir plus : qu'est ce qu'un booléen ?

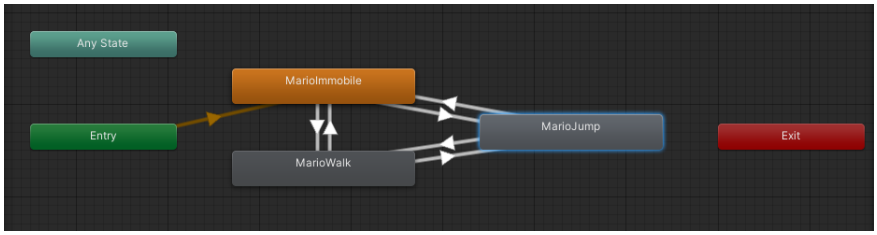
De 1844 à 1854 : Georges Boole [a], un célèbre logicien, crée l'algèbre binaire, nommée en son honneur l'algèbre booléenne.
Cette algèbre est celle de tous les ordinateurs, elle n'accepte que deux valeurs numériques : le 0 et le 1.
Un booléen est donc une valeur qui peut soit être 0 soit 1. Pour que cela soit plus parlant, par convention on utilise une célèbre analogie : 0 = Faux, 1 = Vrai.
Un booléen en informatique ne peut avoir que deux valeurs : **true** ou **false**.

a. Le lien de sa page Wikipédia ICI

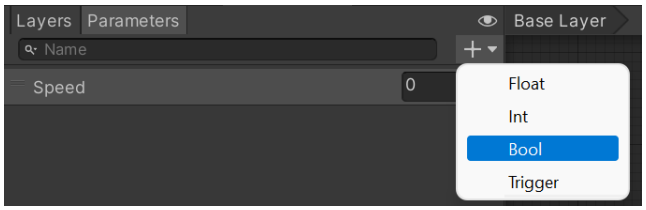
Changeons l'animation quand Mario saute pour une animation de saut. On commence par découper l'asset saut de notre planche d'asset (comme précédemment). Nommez-le "MarioJump". N'oubliez pas d'appuyer sur **Apply** une fois l'asset découpé.

Comme nous l'avons fait pour "MarioImmobile", créons une animation "MarioJump" avec pour seule image, l'Asset de saut. N'hésitez pas à retourner quelques pages en avant pour suivre à nouveau les instructions pour l'animation.

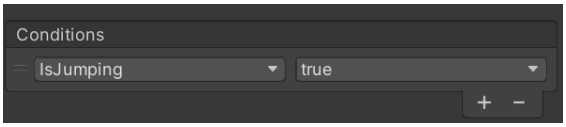
Rendez-vous dans l'**Animator**. Une nouvelle boîte **MarioJump** est apparue. Depuis **MarioImmobile** et **MarioWalk**, créez une flèche vers **MarioJump**. Créez également les flèches inverses. Vous devriez obtenir ce schéma :



Créez une nouvelle variable. Cependant, cette fois nous utiliserons un **Booléen**. Ainsi, dans **Parameters**, choisissez **Bool**. Comme ceci :



Nommez-la "IsJumping".
Pour toutes les nouvelles flèches : décochez la case **HasExitTime** et mettez **Transition duration** à 0.
Pour toutes les flèches entrant dans **MarioJump**, créez une condition avec **IsJumping** à **true** et pour toutes les flèches sortant de **MarioJump**, créez une condition avec **IsJumping** à **false**. Exemple avec une flèche entrante :



Retournons au code pour modifier cette variable et ainsi déclencher l'animation.

```
1  void Update()
2  {
3      [...]
4
5      if (GetComponent<Rigidbody2D>().velocity.y != 0)
6      {
7          MarioAnimator.SetBool("IsJumping", true);
8      }
9      else
10     {
11         MarioAnimator.SetBool("IsJumping", false);
12     }
13
14     [...]
15 }
```

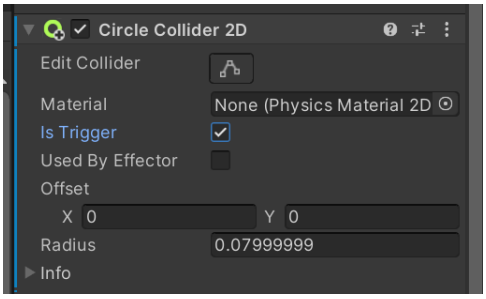
Si la vitesse sur l'axe y est différente de 0, Mario est en l'air, donc il est en train de sauter. Alors on lance l'animation de saut en passant la variable à true : (`MarioAnimator.SetBool("IsJumping", true);`. Sinon, il n'est pas en train de sauter, donc on arrête l'animation en passant la variable à false : `MarioAnimator.SetBool("IsJumping", false);`.

Vous pouvez re-tester et admettre que cette modification esthétique change quand même beaucoup (en tout cas moi je trouve !).

4.2 Money, money, money ...

4.2.1 Gold digger

Sur votre scène, il y a déjà une pièce, mais pour l'instant, vous ne pouvez pas la ramasser. Mario est avide de pièces en or... Il a bien travaillé jusque-là alors récompensons-le ! Commençons par ajouter un **collider** à la pièce pour que Mario puisse interagir avec cette dernière. Cependant, on une souhaite pas que Mario puisse se superposer avec la pièce, mais qu'il puisse la ramasser. Pour cela, cochons la case **Is Trigger** du component.



En savoir plus : à quoi sert la case *Is Trigger* ?

Cette case permet de dire à Unity que Mario peut pénétrer à l'intérieur de ce collider. De plus, lorsqu'un objet entrera dans la zone du collider, Unity exécutera la fonction `OnTriggerEnter2D()` dans le fichier code présent sur l'objet (s'il y en a un).

Créons un nouveau fichier code, dans le dossier **Assets/Coins/Scripts/**. Nommez-le "GetCoin" et ouvrez-le. Cette fois-ci nous ne toucherons pas aux fonctions créées automatiquement par Unity. Sous la fontion `Update()`, ajoutez ce code :

```
1  public class GetCoin : MonoBehaviour
2  {
3      [...]
4
5      private void OnTriggerEnter2D(Collider2D collision)
6      {
7          Destroy(gameObject);
8      }
9  }
```

En savoir plus : explications du code

- `private` indique que cette fonction est privée. Elle ne peut pas être appelée depuis un autre fichier de code.
- `void` signifie que cette fonction ne renvoie rien à celui qu'il l'appelle (ici elle sera appelée par Unity comme les deux précédentes, donc on ne s'en souciera pas).
- `OnTriggerEnter2D` est le nom de la fonction. C'est un fonction spécifique à Unity, appelée automatiquement lorsque l'on entre dans le collider de l'objet et que le case "Is Trigger" est cochée.
- `Collider2D collision` est un argument de la fonction : quelque chose qui doit être envoyé par celui qui appelle la fonction (ici Unity). C'est le collider de l'objet qui est rentrée en collision. Par exemple, si un personnage marche sur un cactus, cela permet de savoir qui c'est pour lui faire perdre de la vie, ou le faire mourir, etc.
- `Destroy()` est la fonction que l'on appelle lorsqu'un objet entre en collision avec la pièce. Elle détruit de la scène ce que l'on met entre les parenthèses.
- `gameObject` est l'objet sur lequel est le script, ici la pièce.

Ainsi, on indique à Unity que lorsqu'un objet entre dans le collider de la pièce, il doit détruire la pièce.

Ça y est, lorsque Mario entre en collision avec une pièce, il la ramasse et s'empresse de la mettre dans sa poche. (La légende raconte que la taille de sa poche serait infinie...)

4.2.2 Gold machine

C'est bien beau de pouvoir ramasser une pièce, mais Mario veut devenir riche ! Alors il y a deux solutions :

- Soit vous recommencez à chaque fois toutes les étapes ci-dessus pour créer plus de pièces
- Soit je vous montre comment utiliser les préfabs et c'est une affaire réglée en moins que temps qu'il n'en faut pour lire le sujet en entier. (Certes, ce n'est pas une bonne mesure de temps, je suis d'accord, disons en moins 2 minutes pour les plus rapides.)

On va partir sur la deuxième méthode !

A quoi sert un préfab ?

On veut pouvoir dupliquer notre pièce sans avoir à remettre toutes les propriétés et modifier les variables pour chaque pièce. Pour cela, on va créer un préfab, un objet préfabriqué. On pourra ensuite ultérieurement glisser/déposer ce préfab n'importe où dans la scène, et l'on aura un nouvelle pièce identique à la première qui nous a servi à créer le moule.

Créons un préfab **Coin**. Depuis l'onglet **Hierarchy**, glissez/déposez **Coin** dans le dossier **Coins/Prefabs**. Le cube dans l'onglet **Hierarchy** devient bleu, il est lié au préfab créé. Maintenant faites l'opération inverse : depuis le dossier **Coins/Prefabs**, glissez/déposez la pièce n'importe où dans l'onglet **Scene**. C'est bon, vous avez dupliqué la pièce !

Synchronisation des Préfabs

Attention ! Si vous modifiez une valeur, ajoutez ou supprimez un component d'un préfab dans la scène, le préfab ne sera pas modifié. A l'inverse, si vous modifiez le préfab directement, toutes les pièces qui y sont liées changeront elles aussi.

4.3 Your turn to play !

Dans le dossier **Floor/Prefabs** se trouve des prefabs de sol que nous avons créées pour que vous puissiez personnaliser votre niveau, alors amusez-vous ;)

N'oubliez pas de Build avant la fin !

N'oubliez pas de recompiler votre projet à nouveau avant la fin de la journée.
Il suffit de suivre à nouveaux les instructions de la partie Build (3.6)

Here we go !