TABLE DES MATIERES

Installation d’unity 77
Manipulations de base 77
Création d'un nouveau projet 77
Navigation dans l'éditeur 77
Utiliser C# avec Unity 79
Travailler avec des scripts C# 79
Présentation de l'éditeur Visual Studio 79
Ouverture d'un fichier C# 79
Synchronisation des fichiers C# 79
Explorer la documentation 79
Programmation 81
Définition des variables 81
Un premier script : 82
Comprendre les méthodes 84
Présentation des classes 85
Travailler avec des commentaires 86
Variables, les types, et méthodes 88
Débogage du code 88
Comprendre les variables 88
Déclarer des variables 88
Déclarations de types et de valeurs 88
Déclarations de type uniquement 88
Utilisation des modificateurs d'accès 89
Travailler avec des types 89
Types d'éléments intégrés courants 89
Conversions de types 90
Déclarations déduites 90
Types personnalisés 90
Nommer les variables (casse CAMEL) 90
Nommer les variables (casse PASCAL) 91
Comprendre la portée des variables 91
Présentation des opérateurs 91
Dans les langages de programmation, les symboles d'opérateurs représentent les fonctions arithmétiques, d'affectation, relationnelles et logiques que les types peuvent exécuter. Les opérateurs arithmétiques représentent les fonctions mathématiques de base, tandis que les opérateurs d'affectation exécutent à la fois des fonctions mathématiques et d'affectation sur une valeur donnée. Les opérateurs relationnels et logiques évaluent les conditions entre plusieurs valeurs, comme plus grand que, moins que et égal à. 91
Arithmétique et devoirs 91
Définition des méthodes 92
Conventions d'appellation 92
Exécution sequentielle du code 92
Spécification des paramètres 92

Disséquer les méthodes courantes d'UnitY 95
La méthode Start() 96
La méthode Update() 96
Résumé ..... **Erreur ! Signet non défini.**
Petit quiz - variables et méthodes ..... **Erreur ! Signet non défini.**
Déclarations de sélection 96
Les collections en un coup d'œil 108
Déclarations d'itération 117
Résumé 123
Présentation de la POO 126
Définition des classes 126
Déclarer des structures 125
Comprendre les types de référence et de valeur 127
Intégrer l'esprit orienté objet 129
Application de la POO dans Unity 135
Résumé 140
Petit quiz - tout ce qui est OOP 141
Un abécédaire de la conception de jeux 142
Construire un niveau 147
Les bases de l'éclairage 161
Animer dans Unity 163
Résumé 170
Petit quiz sur les fonctionnalités de base d'Unity 170
Gestion des déplacements des joueurs 171
Déplacement du lecteur à l'aide du composant Transform 172

INSTALLATION D'UNITY

L'installation de « Unity » est couverte via ce lien ou le module « Unity Learn » : Unity Essential.

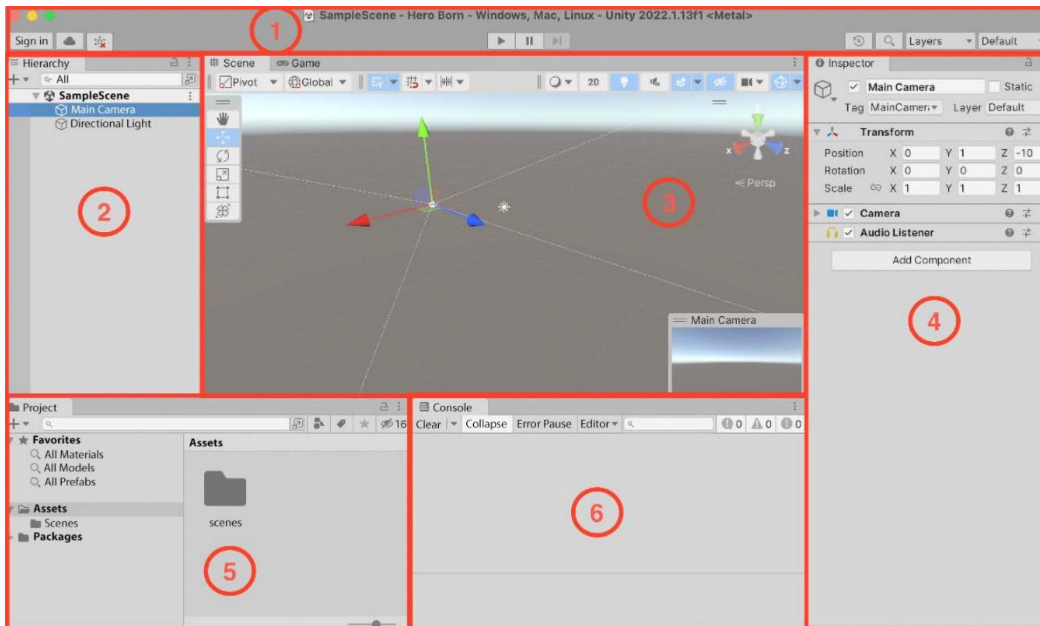
Documentation indispensable de Unity.

MANIPULATIONS DE BASE

CREATION D'UN NOUVEAU PROJET

Compétence : Get started with the Unity Hub.

NAVIGATION DANS L'EDITEUR



Nous allons donc examiner chacun de ces panneaux plus en détail :

La **BARRE D'OUTILS (1)** est la partie supérieure de l'éditeur Unity. À partir de là, vous pouvez vous connecter à un compte Unity, gérer les services, collaborer avec une équipe (groupe de boutons à l'extrême gauche), jouer et mettre en pause le jeu (boutons du centre). Le groupe de boutons le plus à droite contient une fonction de recherche, des **LAYERMASKS, etc**

La fenêtre **HIERARCHY (2)** affiche tous les éléments présents dans la scène du jeu. Dans le projet de démarrage, il ne s'agit que de la caméra par défaut et de la lumière directionnelle, mais lorsque nous créerons notre environnement prototype, cette fenêtre commencera à se remplir avec les objets que nous ajouterons dans la scène.

Les fenêtres **GAME** et **SCENE (3)** sont les aspects les plus visuels de l'éditeur. Considérez la fenêtre **SCENE** comme votre scène, où vous pouvez déplacer et arranger des objets 2D et 3D. Lorsque vous appuyez sur le bouton **Play**, la fenêtre **GAME** prend le relais et affiche la vue de la **SCENE** et toutes les interactions programmées. Vous pouvez également utiliser la vue de la **SCENE** lorsque vous êtes en mode de jeu.

La fenêtre de **L'INSPECTOR (4)** permet de visualiser et modifier les propriétés des objets de la scène. Si vous sélectionnez CAMERA PRINCIPALE dans la **HIERARCHY,** vous verrez plusieurs parties affichées, qu'Unity appelle des composants (**COMPONENTS**), tous accessibles depuis **L'INSPECTOR**.

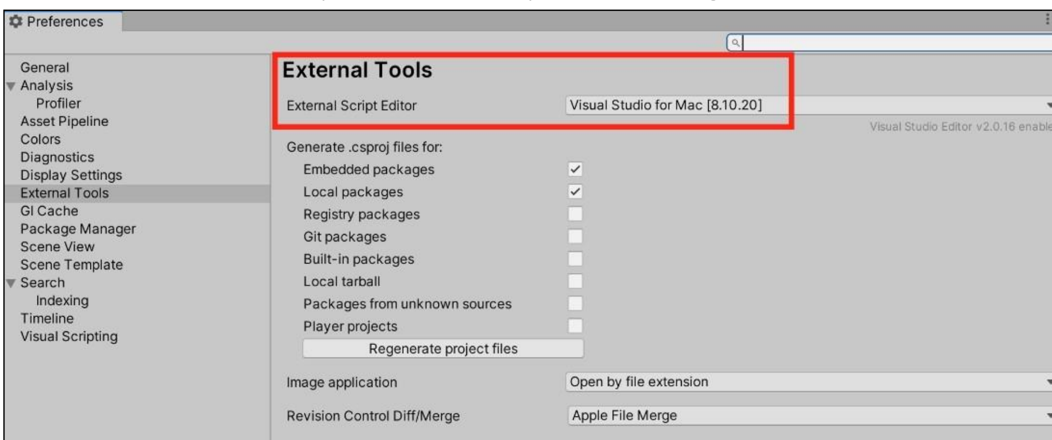
La fenêtre **PROJECT (5)** contient toutes les ressources qui se trouvent actuellement dans votre projet.

La fenêtre **CONSOLE (6)** est l'endroit où s'affiche toute sortie que nous voulons que nos scripts impriment : Cela permet de tester ou débugger le code.

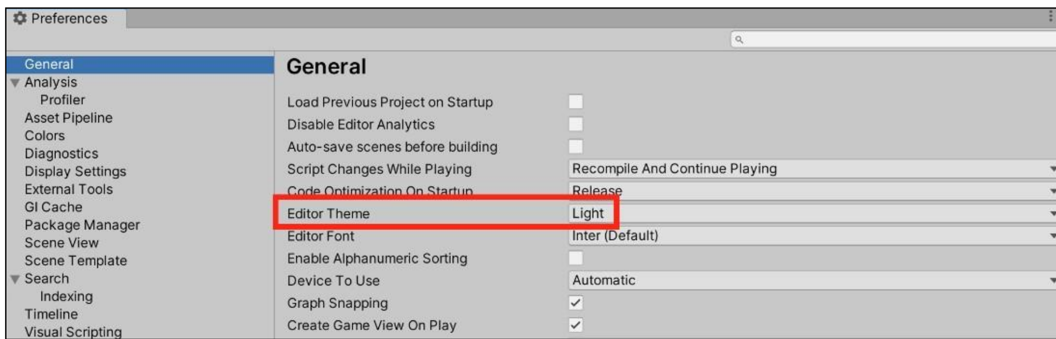
Si l'une de ces fenêtres est fermée par accident, vous pouvez la rouvrir à tout moment à partir du menu **Window>General**

Description plus détaillée des fonctionnalités.

Avant de continuer, il est important que Visual Studio soit configuré comme éditeur de script pour votre projet. Allez dans le menu **Edit>Preferences** et vérifiez que l'éditeur de script externe est réglé sur Visual Studio 2022



Enfin, si vous souhaitez passer du mode clair au mode foncé, allez dans le menu **Edit>Preferences** et modifiez



UTILISER C# AVEC UNITY

Unity est le moteur dans lequel vous allez créer des ***SCRIPTS C#*** et des ***GAMEOBJECTS***, mais la programmation proprement dite s'effectue dans un autre programme appelé Visual Studio -> **MVS**

TRAVAILLER AVEC DES SCRIPTS C#

Il existe plusieurs façons de créer des ***SCRIPTS C#*** à partir de l'éditeur :

- Sélectionnez dans le menu **Assets>Create>C# Script**
- Sous l'onglet **PROJECT**, sélectionnez l'icône **+** et choisissez **C# Script.**
- Cliquez avec le bouton droit de la souris sur le dossier **Assets** dans l'onglet **PROJECT** et sélectionnez **Create>C# Script** dans le menu contextuel.
- Sélectionnez n'importe quel objet de jeu dans la fenêtre **HIERARCHY** et cliquez le bouton en bas **sur Add a component > New script**.

> ***Il faut toujours placer les scripts dans un sous-dossier nommé Scripts du dossier Assets***

PRESENTATION DE L'EDITEUR VISUAL STUDIO

Un double – clic sur le script ouvre Visual Studio.

OUVERTURE D'UN FICHIER C#

Attention aux erreurs de dénomination :

> ***Il faudra nommer correctement tout de suite le script sous réserve de devoir renommer le nom de la classe associée au script. Utiliser C# avec Unity nous oblige à ce que le nom du fichier de script (extension .cs automatique) soit le même que le nom donné à la classe.***

SYNCHRONISATION DES FICHIERS C#

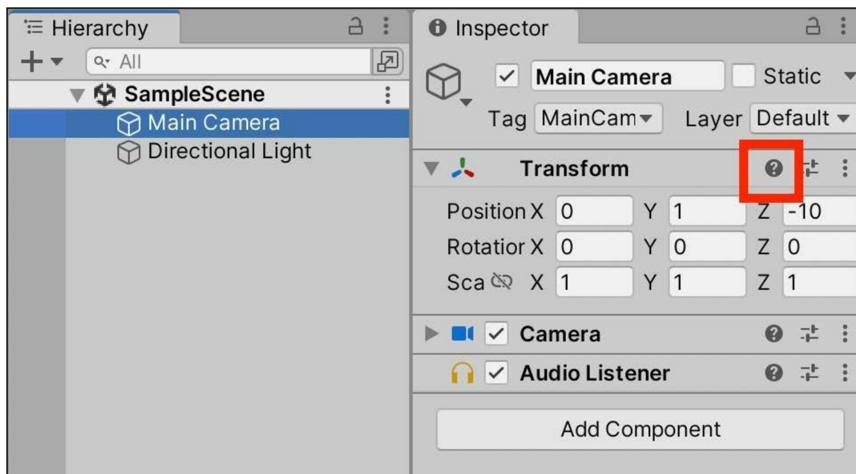
Dans le cadre de leur relation, Unity et Visual Studio communiquent entre eux pour synchroniser leur contenu. Cela signifie que si vous ajoutez, supprimez ou modifiez un fichier de script dans une application, l'autre application verra automatiquement les changements.

EXPLORER LA DOCUMENTATION

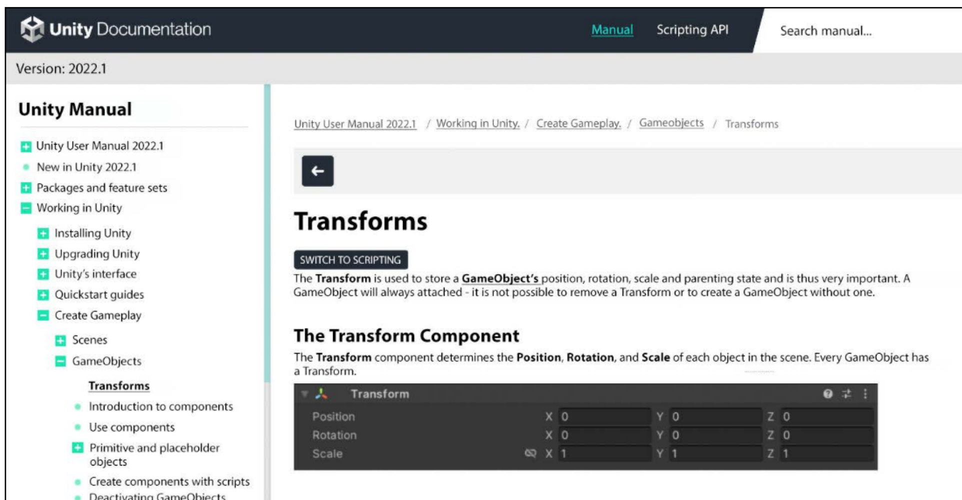
Il est important de prendre de bonnes habitudes dès le début lorsqu'on utilise de nouveaux langages de programmation ou de nouveaux environnements de développement : la documentation en fait partie, et garder à l'esprit que l'on ne peut pas tout retenir.

ACCEDER A LA DOCUMENTATION D'UNITY

1. Dans l'onglet **HIERARCHY**, sélectionnez le ***GAMEOBJECT*** **Main Camera**.
2. Passez à l'onglet **INSPECTOR** et cliquez sur l'icône d'information (point d'interrogation, **?**) en haut à droite du composant **Transform :**

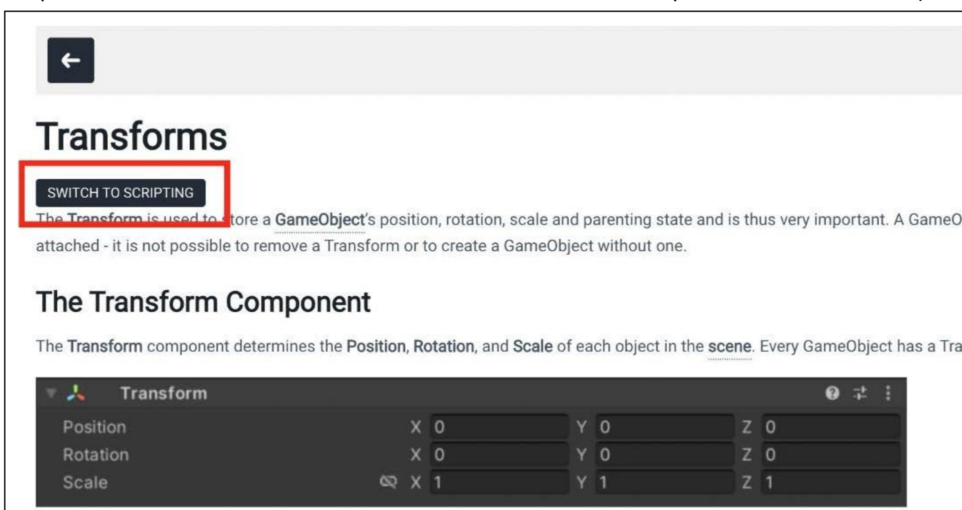


3. Vous verrez un navigateur web ouvert sur la page **Transforms** du manuel de référence :



Si nous voulons des exemples concrets de codage liés au composant **Transform** ? C'est très simple : il suffit de demander à la Référence de script :

1. Cliquez sur le lien **SWITCH TO SCRIPTING** sous le nom du composant ou de la classe (**Transforms**, dans ce cas) :



2. Ce faisant, le manuel de référence passe automatiquement à la référence des scripts :

Transform

class in UnityEngine / Inherits from:Component/ Implemented in:UnityEngine.CoreModule

SWITCH TO MANUAL

Description

Position, rotation and scale of an object.

Every object in a Scene has a Transform. It's used to store and manipulate the position, rotation and scale of the object. Every Transform can have a paren
apply position, rotation and scale hierarchically. This is the hierarchy seen in the Hierarchy pane. They also support enumerators so you can loop through

```
using UnityEngine;

public class Example : MonoBehaviour
{
    // Moves all transform children 10 units upwards!
    void Start()
    {
        foreach (Transform child in transform)
        {
            child.position += Vector3.up * 10.0f;
        }
    }
}
```

3. Comme vous pouvez le constater, outre l'aide au codage, il existe également une option permettant de revenir au manuel de référence si nécessaire.

La Référence pour les scripts est un document volumineux, parce qu'il doit l'être. Cependant, cela ne signifie pas que vous devez le mémoriser ou même être familier avec toutes les informations qu'il contient pour commencer à écrire des scripts. Comme son nom l'indique, il s'agit d'une référence et non d'un test.

Si vous vous perdez dans la documentation, ou si vous ne savez pas où chercher, vous pouvez également trouver des solutions au sein de la riche communauté de développement Unity dans les endroits suivants :

- Forum Unity
- Unity Answers
- Unity Discord

LOCALISATION DES RESSOURCES C#

Documentation Microsoft Learn : contient un grand nombre de tutoriels, de guides de démarrage rapide et d'articles pratiques.

PROGRAMMATION

Tout langage de programmation est perçu au départ comme quelque chose de plutôt repoussant de prime abord, mais tous les langages de programmation sont constitués des mêmes éléments essentiels. Variables, méthodes et classes (ou objets) constituent l'ADN de la programmation conventionnelle ; la compréhension de ces concepts simples ouvre la voie à un monde entier d'applications diverses et complexes.

Si vous êtes novice en programmation, il y a beaucoup d'informations à assimiler au début, et il faudra ne pas hésiter à reprendre les notions, se les approprier en se fixant de objectifs très simples. Avec les nouvelles technologies, les exemples de codes sont nombreux, et s'il faut ne pas se priver de cette source, il faut rester humble et coder soi-même petit à petit : c'est le meilleur moyen d'apprendre de manière cyclique en copiant, en essayant, en recherchant ce qui ne va pas, etc …

DEFINITION DES VARIABLES

Commençons par une question simple : qu'est-ce qu'une variable ? Selon le point de vue que l'on adopte, il y a plusieurs façons de répondre à cette question :

- Sur le plan conceptuel, une variable est l'unité de base de la programmation, comme un atome l'est dans le monde physique (à l'exception de la théorie des cordes). Tout commence par des variables, et les programmes ne peuvent exister sans elles.
- Techniquement, une variable est une petite section de la mémoire de votre ordinateur qui contient une valeur **assignée**. Chaque variable garde une trace de l'endroit où ses informations sont stockées (c'est ce qu'on appelle une ADRESSE MEMOIRE), de sa valeur et de son type (par exemple, des nombres, des mots ou des listes).

- En pratique, une variable est un conteneur. Vous pouvez en créer de nouvelles quand vous le souhaitez, les remplir de choses, les déplacer, modifier ce qu'elles contiennent et y faire référence si nécessaire. Elles peuvent même être vides et rester utiles !

Complétez cette information par une recherche dans la documentation de Microsoft

Rédigez dans ce cadre

Les variables peuvent contenir différents types d'informations. En C#, les variables peuvent contenir des chaînes de caractères (texte), des entiers (nombres), des booléens (valeurs binaires représentant soit le vrai, soit le faux), et des objets comme ceux que l'on a vu jusqu'à présent (**GameObject** **Main Camera** et **component** **transform**)

Les noms sont importants

La mise en forme et le nom que l'on donne aux variables sont très importants car elles sont utilisées dans l'ensemble de l'application.

Les variables agissent comme des espaces réservés

Lorsque vous créez et nommez une variable, vous créez un espace réservé pour la valeur que vous souhaitez stocker.

UN PREMIER SCRIPT :

> Dans un projet nommé **PremierProjet**, la scène est nommée **PremiereScene**.
> Ajoutez un GameObject nommé **UnCube.**
> Ajoutez un script nommé **testA** : on notera que c'est un composant comme un autre.
> Liez le script **testA** au GameObject nommé **UnCube**

Pour ce faire, il faut :

> Visualiser dans l'arborescence de la fenêtre **PROJECT** votre script.
> Glisser le script et le déposer sur l'objet dans la fenêtre **HIERARCHY**.
> Vérifier dans **INSPECTOR** que le script apparaît bien comme composant.

Les directives using permettant à C# d'accéder à des bibliothèques UNITY.

La classe **testA** est basée sur une classe appelée *MonoBehaviour* qui va permettre à notre script de gérer l'objet et les composants qui lui sont attachés.

Start() et *Update()* sont des méthodes non obligatoires exécutées respectivement 1 fois au démarrage et 1 fois par Frame (Image) dont la fréquence est proportionnelle à la vitesse du processeur.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
public class testA : MonoBehaviour
{
    // Déclarations de variables au niveau de la classe
    // Start is called before the first frame update
    void Start()
    {
        // votre code ici
    }
    // Update is called once per frame
    void Update()
    {
        // votre code ici
    }
}
```

> Dans la méthode Start, ajoutez les lignes ci-dessous pour déclarer des variables

ATTENTION : pensez dans MVS à sauvegarder systématiquement avant de retourner dans UNITY (CTRL + MAJ + S pour tout enregistrer), sinon, le script ne sera pas synchronisé avec Unity, et la compilation se fera avec l'ancienne version.

```
public int Age = 30;
public int AnneeNaissance = 1965;
private int AnneeActuelle = 2023;
private string PhraseAffiche = string.Empty;
```

Affichons la valeur des variables dans la console avec les lignes suivantes dans Start()

```
Debug.Log(Age);
Debug.Log("L'année de naissance est : " + AnneeNaissance);
```

Exécutez le code en prenant soin d'activer la fenêtre **CONSOLE**.
Observez les résultats qui doivent être conformes à ce que l'on attend du code.
Stoppez le programme.
Enregistrez vos fichiers dans MVS.

Dans MVS, avant de basculer dans Unity, tapez SHIFT+CTRL+S

Il est également important de noter que les variables dites *public* apparaissent dans l'inspecteur Unity, contrairement aux variables dites *private*.

Dans MVS, on peut voir que les variables utilisées dans le code sont en gras.

Quand vous cliquez sur Age, les occurrences de Age apparaissent en surbrillance.

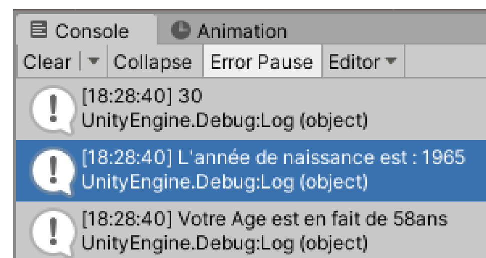
On peut remarquer que si on calcule l'âge, la variable Age ne devrait pas être de 30.

Ajoutez le code dans Start() à la suite des lignes existantes.

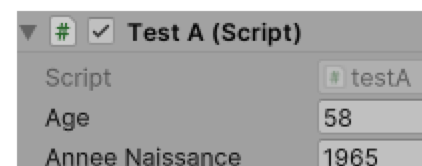
```
Age = AnneeActuelle - AnneeNaissance;
PhraseAffiche = "Votre Age est en fait de " + Age + " ans";
Debug.Log(PhraseAffiche);
```

Exécutez le code

La console affiche bien la phrase attendue, la variable Age à été initialisée à 30, mais a été recalculée à 58.



De même, observez que dans l'inspecteur, les valeurs de nos variables s'affichent aussi : cela va permettre de modifier les valeurs de données et d'affficher des résultats de manière simple en alternative à Debug.Log(). La valeur 58 disparaît ensuite lorsque le programme est arrêté au profit de la valeur 30.



Modifiez le code `public int Age = 30;` en `public int Age;`
Exécutez à nouveau.
Pendant l'exécution, modifiez la valeur de l'année de naissance.
Stoppez l'exécution.

L'Age n'est pas recalculé, car la variable Age est affectée lors de l'exécution de Start(). L'instruction qui calcule l'âge n'est plus exécutée.

Déplacez le calcul de l'Age dans Update()
Testez en modifiant l'année de naissance.

L'inspecteur doit afficher l'Age.

Pour commenter des lignes de codes, après les avoir sélectionnées, tapez CTRL+K+C
Pour retirer les commentaires, tapez CTRL+K+U

> Mettez Debug.log en commentaire car la console est polluée par l'affichage infini du contenu de la variable PhraseAffiche.

COMPRENDRE LES METHODES

Les méthodes sont à l'origine des actions

Tout comme les variables, la définition des méthodes de programmation peut être fastidieuse ou dangereusement brève ; voici une autre approche en trois volets à envisager :

- D'un point de vue **conceptuel**, les méthodes sont la façon dont le travail est effectué dans une application.
- **Techniquement**, une méthode est un bloc de code contenant des instructions exécutables qui s'exécutent lorsque la méthode est appelée par son nom. Les méthodes peuvent recevoir des arguments (également appelés paramètres), qui peuvent être utilisés à l'intérieur de la portée de la méthode.
- En **pratique**, une méthode est un conteneur pour un ensemble d'instructions qui s'exécutent à chaque fois qu'elle est exécutée. Ces conteneurs peuvent également recevoir des variables en entrée, qui ne peuvent être référencées qu'à l'intérieur de la méthode elle-même.

Dans l'ensemble, les méthodes sont l'ossature de tout programme : elles relient tout et presque tout est construit à partir de leur structure.

LES METHODES EVITENT LA REPETITION DU CODE

En C#, une méthode est un bloc de code qui effectue une tâche spécifique lorsqu'elle est appelée. Elle peut prendre des paramètres en entrée et retourner une valeur en sortie, ou ne rien retourner du tout. Les méthodes sont généralement utilisées pour organiser le code en petites unités de fonctionnalité qui peuvent être appelées depuis d'autres parties du programme, ce qui permet de simplifier la logique du code et de le rendre plus facile à maintenir. Les méthodes peuvent également être utilisées pour encapsuler du code réutilisable qui peut être utilisé dans différentes parties du programme.

ATTENTION : vous allez ajouter une deuxième script : que faire du premier ? comment garder plusieurs scripts en cours d'exécution ?

Pour faire simple, le script associé à un objet peut être désactivé de sorte que si plusieurs scripts sont liés à un même objet, on peut facilement interrompre l'exécution de chaque script. On peut aussi créer un objet par script, mail faudra aussi désactiver l'objet (on peut le rendre invisible) ou le script. Au bout d'un certain temps, on peut aussi créer plusieurs scènes.

> Créez testB et liez testB au cube : par défaut un script est actif après avoir été lié.
> Désactivez testA du cube.
> Après avoir copié correctement le code dans testB, testez si tout fonctionne bien.

Observons le code ci-dessous de testB

```
public class testB : MonoBehaviour
{
    public int AgeActuel;
    public int AgeFutur;
    public int AnneeNaissance = 1965;
    private int AnneeActuelle = 2023;
    private int AnneeFutur = 2030;
    public string PhraseAffiche = string.Empty;

    void Update()
    {
        AgeActuel = CalculeAge(AnneeNaissance,AnneeActuelle);
        AgeFutur = CalculeAge(AnneeNaissance,AnneeFutur,ref PhraseAffiche);
    }
    private int CalculeAge(int dtNais, int dt)
    {
        int age = dt - dtNais;
        return age;
    }
    private int CalculeAge(int dtNais, int dt, ref string phrase)
    {
        int age = CalculeAge(dtNais, dt);
        phrase = "En " + dt + " vous avez " + age + " ans";
        return age;
    }
}
```

Nous avons une méthode CalculAge() qui calcule la différence entre deux années que l'on passe en paramètre ou argument, les paramètres passés à la méthode sont typés en entier (int ici).

La deuxième méthode qui porte le même nom est une **surcharge** de la première qui va permettre d'ajouter une **fonctionnalité** à la méthode de base : nous construisons une phrase qui va permettre un affichage en clair. Cette surcharge est un ajout d'une fonctionnalité optionnelle (mot clé **ref** pour permettre à la méthode d'accéder à **l'adresse mémoire** de la variable et en modifier sa valeur). Cela permet de ne pas dupliquer le code puisque la deuxième appelle la première et ajoute la nouvelle affectation de la variable phrase.

Le mot-clé (modificateur d'accès) private devant la méthode implique que celle-ci ne peut être appelée qu'à l'intérieur de la classe elle-même, pas à partir d'une autre classe.

PRESENTATION DES CLASSES

Les variables stockent des informations et les méthodes exécutent des actions, mais notre boîte à outils de programmation est encore quelque peu limitée. Nous avons besoin d'un moyen de créer une sorte de super conteneur, contenant des variables et des méthodes qui peuvent être référencées à l'intérieur du conteneur lui-même. C'est là qu'interviennent les classes :

- D'un point de vue conceptuel, une classe contient des informations, des actions et des comportements connexes dans un seul et même conteneur. Elles peuvent même communiquer entre elles.
- Techniquement, les classes sont des structures de données. Elles peuvent contenir des variables, des méthodes et d'autres informations programmatiques, qui peuvent toutes être référencées lorsqu'un objet de la classe est créé.
- En pratique, une classe est un modèle. Elle définit les règles et règlements applicables à tout objet (appelé instance) créé à l'aide du modèle de la classe.

UNE CLASSE UNITY COMMUNE

Chaque script créé dans Unity est une classe, comme le montre le mot-clé **class** :

```
public class testB : MonoBehaviour
{

}
```

MonoBehaviour signifie simplement que cette classe peut être attachée à un GameObject dans la scène Unity, et les deux crochets marquent les limites de la classe - tout le code à l'intérieur de ces crochets appartient à cette classe.

Les classes peuvent exister par elles-mêmes, ce que nous verrons lorsque nous créerons des classes autonomes

Les termes scripts et class sont parfois utilisés de manière interchangeable dans les ressources Unity.

> ***Par souci de cohérence les fichiers C# seront des scripts s'ils sont attachés à des GameObjects et des classes si ils sont autonomes.***

LES CLASSES SONT DES SCHEMAS DIRECTEURS

En C#, une classe est un modèle ou un plan pour créer des objets qui ont des propriétés et des comportements communs. Une classe est définie en regroupant des variables, des méthodes et d'autres membres de données et de fonctions qui sont utilisés pour créer des instances d'objets.

Les propriétés d'une classe sont les caractéristiques ou les attributs qui décrivent l'objet, comme son nom, son identifiant unique, sa couleur, sa taille, etc. Les méthodes sont les comportements ou les actions que l'objet peut effectuer, telles que l'affichage de ses propriétés, la modification de ses propriétés, la communication avec d'autres objets, etc.

EXEMPLE D'UNE CLASSE POUR GERER UN VEHICULE

Dans cet exemple, nous allons créer une classe "Vehicle" (véhicule) avec des propriétés telles que la vitesse, la direction, la couleur et une méthode pour changer la direction.

```
public class Vehicle
{
    // Propriétés
    public double Speed { get; set; }
    public string Direction { get; set; }
    public string Color { get; set; }

    // Méthodes
    public void ChangeDirection(string newDirection)
    {
        this.Direction = newDirection;
    }
}
```

Dans cette classe, nous avons trois propriétés: `Speed`, `Direction`, et `Color`. `Speed` est une propriété numérique qui stocke la vitesse actuelle du véhicule. `Direction` est une propriété de chaîne de caractères qui stocke la direction actuelle du véhicule. `Color` est une propriété de chaîne de caractères qui stocke la couleur du véhicule.

La méthode `ChangeDirection` permet de changer la direction du véhicule en prenant une chaîne de caractères en entrée qui représente la nouvelle direction.

Vous pouvez utiliser cette classe pour créer des instances de véhicules spécifiques, en définissant leurs propriétés et en appelant leurs méthodes.

Voici un exemple de code qui crée une instance de la classe `Vehicle` et utilise sa méthode `ChangeDirection` pour changer sa direction :

```
Vehicle myVehicle = new Vehicle();
myVehicle.Speed = 50.0;
myVehicle.Direction = "north";
myVehicle.Color = "red";

myVehicle.ChangeDirection("east");
```

TRAVAILLER AVEC DES COMMENTAIRES

Les commentaires sont essentiels pour la lisibilité, la compréhension, la maintenance et le débogage du code. En ajoutant des commentaires clairs et concis, vous pouvez aider les autres développeurs à comprendre votre code plus rapidement et efficacement, tout en améliorant la qualité globale de votre code.

```
// permet de mettre une ligne en commentaire
/* ceci est un
commentaire sur
plusieurs lignes */
```

En C#, il est recommandé de documenter chaque méthode à l'aide de commentaires XML pour améliorer la lisibilité, la compréhension et la maintenance du code. Les commentaires XML sont des commentaires spéciaux qui commencent par trois barres obliques (///), suivis d'une description de la méthode et de ses paramètres, ainsi que des informations sur les exceptions potentielles qui peuvent être levées.

Voici un exemple de documentation d'une méthode C# à l'aide de commentaires XML.

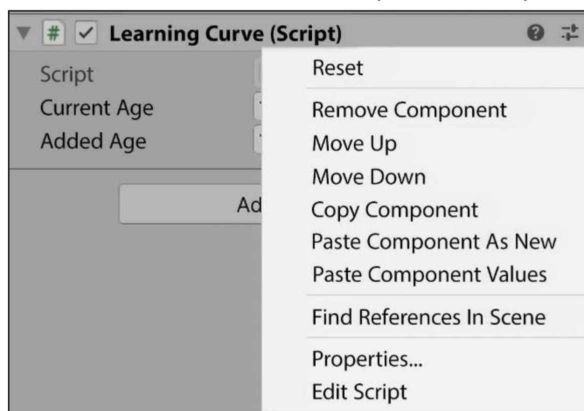
Une fois mis en place, les informations fournies seront affichées lors du passage de la souris sur les appels de la méthode et ses paramètres.

```
    /// <summary>
    /// Calcule la différence d'age entre une année dt et une année de naissance dtNais
    /// </summary>
    /// <param name="dtNais">Année de naissance</param>
    /// <param name="dt">Année de référence</param>
    /// <returns>Age calculé</returns>
    private int CalculeAge(int dtNais, int dt)
    {
        int age = dt - dtNais;
        return age;
    }
```

Lorsque vous êtes en **mode développement**, toutes les modifications que vous apportez aux variables sont sauvegardées par Unity. Cela signifie que si vous quittez Unity et que vous le redémarrez, les modifications seront conservées.

Les modifications que vous apportez aux valeurs du panneau **Inspecteur** en **mode lecture** ne modifient pas votre script, mais elles remplacent les valeurs que vous avez attribuées à votre script en **mode développement**.

Toutes les modifications apportées en **mode lecture** seront toujours réinitialisées automatiquement lorsque vous arrêterez le **mode lecture**. Si vous devez annuler des modifications effectuées dans le panneau **Inspecteur**, vous pouvez rétablir les valeurs par défaut (parfois appelées valeurs **initiales**) du script. Cliquez sur l'icône à trois points verticaux située à droite de n'importe quel composant, puis sélectionnez **Réinitialiser**, comme indiqué dans la capture d'écran suivante :



Cela devrait vous donner une certaine tranquillité d'esprit : si vos variables deviennent incontrôlables, il y a toujours la possibilité de réinitialiser le système.

UN COUP DE POUCE DE MONOBEHAVIOUR

Tous les scripts n'ont pas besoin d'hériter de MonoBehaviour - ce n'est nécessaire que pour ceux que vous voulez ajouter aux GameObjects dans vos scènes Unity.

Le sujet de l'héritage de classes est un peu avancé à ce stade de votre parcours de programmation ; considérez-le comme la classe MonoBehaviour qui prête quelques-unes de ses variables et méthodes à votre script.

Nous verrons plus loin comment écrire des classes qui n'héritent pas de MonoBehaviour.

Cherchez des informations sur Start() et Update() dans la documentation Unity. Existe-t-il d'autres méthodes spécifiques à Unity.

Votre réponse ici.

Cherchez la documentation sur MonoBehaviour

Votre réponse ici.

VARIABLES, LES TYPES, ET METHODES

DEBOGAGE DU CODE

Nous avons travaillé avec `Debug.log()` pour afficher des données ou des variables dans la console. Cela fonctionne aussi pour les objets.

Pour un débogage plus complexe, utilisez `Debug.LogFormat()` : cela vous permettra de placer des variables à l'intérieur du texte imprimé en utilisant des espaces réservés.

V1 : Ceux-ci sont marqués par une paire de crochets, chacun contenant chacun le nom de la variable.

V2 : Une alternative consiste à placer entre crochet un n° d'ordre, puis de placer les variables en paramètres à passer dans l'ordre fixé.

> Dans le script testB, ajoutez ou modifiez la procédure Start()

```
private void Start()
{
    // V1
    Debug.LogFormat($"Naissance : {AnneeNaissance} - Année Actuelle : {AnneeActuelle}");
    // V2
    Debug.LogFormat("Naissance : {0} - Année Actuelle : {1} », AnneeNaissance, AnneeActuelle);
}
```

COMPRENDRE LES VARIABLES

Dans le chapitre précédent, nous avons vu comment les variables sont écrites et nous avons abordé les fonctionnalités de haut niveau qu'elles fournissent. Cependant, il nous manque encore la syntaxe qui rend tout cela possible, alors commençons par les bases : la déclaration des variables.

DECLARER DES VARIABLES

Les variables n'apparaissent pas simplement au début d'un script C# ; elles doivent être déclarées conformément à certaines règles et exigences. À son niveau le plus élémentaire, une déclaration de variable doit satisfaire aux exigences suivantes :

- Le type de données que la variable stockera doit être spécifié
- La variable doit avoir un nom unique
- S'il existe une valeur assignée, elle doit correspondre au type spécifié
- La déclaration de la variable doit se terminer par un point-virgule

Le respect de ces règles se traduit par la syntaxe suivante : `dataType UniqueName = valeur ;`

Les variables doivent porter des noms uniques afin d'éviter tout conflit avec des mots qui ont déjà été pris par C#, et qui sont appelés **mots-clés**. Vous trouverez la liste complète des mots-clés protégés dans la documentation Microsoft.

DECLARATIONS DE TYPES ET DE VALEURS

Le scénario le plus courant pour la création de variables est celui dans lequel toutes les informations requises sont disponibles au moment de la déclaration. Par exemple, si nous connaissons l'âge d'un joueur, l'enregistrer serait aussi simple que de faire ce qui suit : `int AgeActuel = 32 ;`

Dans ce cas, toutes les conditions de base sont remplies :

- Un type de données est spécifié, à savoir **int** (abréviation de **integer**, qui est un mot pour désigner un nombre entier).
- Un nom unique est utilisé, à savoir `AgeActuel`
- 32 est un nombre entier, qui correspond au type de données spécifié
- La déclaration se termine par un point-virgule

Cependant, dans certains cas, vous voudrez déclarer une variable sans connaître immédiatement sa valeur. Nous aborderons ce sujet dans la section suivante.

DECLARATIONS DE TYPE UNIQUEMENT

Considérez un autre scénario : vous connaissez le type de données que vous souhaitez stocker dans une variable ainsi que son nom, mais pas sa valeur. La valeur sera calculée et affectée ailleurs, mais vous devez toujours déclarer la variable en tête du script. Cette situation est : `int AgeActuel;`

Seuls le type (int) et le nom unique (`AgeActuel`) sont définis, mais l'instruction reste valide car nous avons respecté les règles. Si aucune valeur n'est attribuée, des valeurs par défaut seront attribuées en fonction du type de la variable. Dans ce cas, `AgeActuel`

sera mis à 0, ce qui correspond au type **int**. Dès que la valeur réelle de la variable devient disponible, elle peut facilement être définie dans une instruction séparée en faisant référence au nom de la variable et en lui attribuant une valeur : `AgeActuel = 32 ;`

UTILISATION DES MODIFICATEURS D'ACCES

En C#, les modificateurs d'accès sont utilisés pour contrôler la visibilité des membres d'une classe, c'est-à-dire leur accessibilité à partir d'autres parties du code. Il existe quatre modificateurs d'accès en C# : public, private, protected et internal.

- **public** : Les membres déclarés avec le modificateur public sont accessibles à partir de n'importe quelle partie du programme. Les membres publics peuvent être appelés à partir de n'importe quelle classe ou méthode, qu'elles soient situées dans le même projet ou dans un projet différent.
- **private** : Les membres déclarés avec le modificateur private sont accessibles uniquement à partir de l'intérieur de la classe dans laquelle ils sont déclarés. Les membres privés ne peuvent pas être appelés depuis une autre classe ou méthode, même si elles sont situées dans le même projet.
- **protected** : Les membres déclarés avec le modificateur protected sont accessibles à partir de la classe dans laquelle ils sont déclarés et de ses classes dérivées (héritées). Les membres protégés ne peuvent pas être appelés à partir d'une classe qui n'est pas dérivée de la classe dans laquelle ils sont déclarés.
- **internal** : Les membres déclarés avec le modificateur internal sont accessibles à partir de n'importe quelle partie du code dans le même assemblage (un assemblage est un fichier .dll ou .exe). Les membres internes ne peuvent pas être appelés depuis un autre assemblage.

Il est important de noter que la visibilité par défaut des membres en C# est private. Cela signifie que si vous ne spécifiez pas de modificateur d'accès pour un membre, il sera automatiquement déclaré comme privé. Il est donc recommandé d'utiliser explicitement le modificateur d'accès approprié pour chaque membre de votre classe afin de rendre votre code plus clair et plus facilement compréhensible.

Toute variable qui n'est pas marquée comme **public** est considérée comme **private** par défaut et n'apparaîtra pas dans la base de données Unity dans la fenêtre **INSPECTOR**.

> ***Si vous incluez un modificateur, la recette syntaxique mise à jour se présentera comme suit :*** *`accessModifier dataType UniqueName = value ;`*

TRAVAILLER AVEC DES TYPES

L'attribution d'un type spécifique à une variable est un choix important, qui se répercute sur toutes les interactions de la variable au cours de sa vie. Le C# étant ce qu'on appelle un langage fortement typé, chaque variable doit avoir un type de données sans exception. En comparaison, les langages de programmation comme JavaScript, par exemple, ne sont pas sûrs quant au type de données. Cela signifie qu'il existe des règles spécifiques lorsqu'il s'agit d'effectuer des opérations avec certains types, et des règles lorsqu'il s'agit de convertir un type de variable donné en un autre.

TYPES D'ELEMENTS INTEGRES COURANTS

Tous les types de données dans C# descendent (ou **dérivent**, en termes programmatiques) d'un ancêtre commun : **SYSTEM.OBJECT**. Cette hiérarchie, appelée Common Type System (CTS), signifie que les différents types ont de nombreuses fonctionnalités communes. Le tableau suivant présente certaines des options de type de données les plus courantes et les valeurs qu'elles stockent :

Type	Contents of the variable
int	A simple integer, such as the number 3
float	A number with a decimal, such as the number 3.14
string	Characters in double quotes, such as,“Watch me go now”
bool	A Boolean, either **true** or **false**

En plus de spécifier le type de valeur qu'une variable peut stocker, les types contiennent des informations supplémentaires sur eux-mêmes, notamment les suivantes :

- Espace de stockage requis
- Valeurs minimales et maximales
- Opérations autorisées
- Emplacement dans la mémoire
- Méthodes accessibles
- Type de base (dérivé)

Travailler avec tous les types offerts par C# est un exemple parfait d'utilisation de la documentation plutôt que de la mémorisation. Très vite, l'utilisation des types personnalisés, même les plus complexes, vous semblera une seconde nature.

CONVERSIONS DE TYPES

Nous avons déjà vu que les variables ne peuvent contenir que des valeurs de leur type déclaré, mais il peut arriver que vous deviez combiner des variables de types différents. Dans la terminologie de la programmation, on appelle cela des conversions, qui se présentent sous deux formes principales :

Les conversions **implicites** ont lieu automatiquement, généralement lorsqu'une valeur plus petite peut être insérée dans un autre type de variable sans qu'il soit nécessaire de l'arrondir. Par exemple, tout nombre entier peut être implicitement converti en valeur double ou flottante sans code supplémentaire :

```
int MyInteger = 3;
float MyFloat = MyInteger;
Debug.Log(MyInteger);
Debug.Log(MyFloat);
```

Les conversions **explicites** sont nécessaires lorsqu'il y a un risque de perdre les informations d'une variable lors de la conversion. Par exemple, si nous voulions convertir une valeur double en valeur int, nous devrions la convertir explicitement en ajoutant le type de destination entre parenthèses avant la valeur que nous voulons convertir.

Cela indique au compilateur que nous sommes conscients que des données (ou de la précision) peuvent être perdues :

```
int ExplicitConversion = (int)3.14 ;
```

Dans cette conversion explicite, 3.14 serait arrondi à 3, perdant ainsi les valeurs décimales.

C# fournit des méthodes intégrées pour convertir explicitement des valeurs en types courants. Par exemple, n'importe quel type peut être converti en chaîne de caractères avec la méthode ToString(), tandis que la classe Convert peut gérer des conversions plus complexes.

DECLARATIONS DEDUITES

Heureusement, C# peut déterminer le type d'une variable à partir de la valeur qui lui est attribuée. Par exemple, le mot-clé var peut indiquer au programme que le type de la donnée, Temperature, doit être déterminé par sa valeur de 32.0, qui est un nombre décimal : `var Temperature = 32.0f ;`

Bien que cela soit pratique dans certaines situations, ne vous laissez pas entraîner dans une habitude de programmation paresseuse consistant à utiliser des déclarations de variables déduites pour tout. Cela ajoute beaucoup d'incertitude à votre code, alors qu'il devrait être clair comme de l'eau de roche. Les déclarations de variables déduites ne doivent être utilisées que lorsque vous testez un code et que vous ne connaissez pas le type de données stockées. Une fois que vous le savez, il est recommandé de modifier la déclaration de la variable en fonction du type spécifique afin d'éviter les erreurs d'exécution ultérieures.

TYPES PERSONNALISES

Lorsque nous parlons de types de données, il est important de comprendre dès le départ que les nombres et les mots (appelés valeurs littérales) ne sont pas les seuls types de valeurs qu'une variable peut stocker. Par exemple, une classe, une structure ou une énumération peuvent être stockées en tant que variables.

NOMMER LES VARIABLES (CASSE CAMEL)

La casse Camel (ou notation en chameau en français) est une convention de nommage pour les identificateurs de variables, de méthodes, de classes et d'autres éléments de code en informatique. La casse Camel est largement utilisée en C# et dans d'autres langages de programmation tels que Java et JavaScript.

La casse Camel consiste à écrire le premier mot en minuscules, puis chaque mot suivant en majuscules sans espace, de sorte que le nom ressemble à une bosse de chameau. Par exemple, voici quelques exemples d'identificateurs de code en casse Camel :

- firstName
- lastName
- accountBalance
- orderDetails
- customerAddress

La casse Camel est utilisée pour améliorer la lisibilité du code en rendant les identificateurs plus faciles à lire. En utilisant des majuscules pour séparer les mots, les identificateurs sont plus faciles à distinguer les uns des autres et à identifier leurs composants individuels.

Il est important de noter que la casse Camel est une convention de nommage et n'affecte pas le comportement du code lui-même. Cependant, il est recommandé de suivre les conventions de nommage telles que la casse Camel pour améliorer la lisibilité et la compréhension du code par les autres développeurs qui travaillent sur le même projet.

NOMMER LES VARIABLES (CASSE PASCAL)

La casse Pascal consiste à écrire le premier mot et chaque mot suivant en majuscules sans espace. Contrairement à la casse Camel, qui met le premier mot en minuscules, la casse Pascal met tous les mots en majuscules. Par exemple, voici quelques exemples d'identificateurs de code en casse Pascal :

- FirstName
- LastName
- AccountBalance
- OrderDetails
- CustomerAddress

COMPRENDRE LA PORTEE DES VARIABLES

À l'instar des modificateurs d'accès, qui déterminent les classes extérieures pouvant accéder aux informations d'une variable, la portée d'une variable est le terme utilisé pour décrire l'emplacement d'une variable donnée et son point d'accès au sein de la classe qui la contient.

Il existe trois niveaux principaux de portée des variables en C :

- La portée globale se réfère à une variable qui peut être accédée par un programme entier. C# ne supporte pas directement les variables globales, mais le concept est utile dans certains cas
- La portée d'une classe ou d'un membre fait référence à une variable qui est accessible partout dans la classe qui la contient.
- La portée locale fait référence à une variable qui n'est accessible qu'à l'intérieur d'une méthode ou d'un bloc spécifique du code dans lequel il est créé.

PRESENTATION DES OPERATEURS

Dans les langages de programmation, les symboles d'opérateurs représentent les fonctions arithmétiques, d'affectation, relationnelles et logiques que les types peuvent exécuter. Les opérateurs arithmétiques représentent les fonctions mathématiques de base, tandis que les opérateurs d'affectation exécutent à la fois des fonctions mathématiques et d'affectation sur une valeur donnée. Les opérateurs relationnels et logiques évaluent les conditions entre plusieurs valeurs, comme plus grand que, moins que et égal à.

ARITHMETIQUE ET DEVOIRS

Les symboles des opérateurs arithmétiques vous sont déjà familiers : + pour l'addition,- pour la soustraction,/ pour la division,* pour la multiplication

Les opérateurs de C# suivent l'ordre conventionnel des opérations, c'est-à-dire qu'ils évaluent d'abord les parenthèses, puis les exposants, puis la multiplication, puis la division, puis l'addition et enfin la soustraction.

Par exemple, les équations suivantes donneront des résultats différents, même si elles contiennent les mêmes valeurs et les mêmes opérateurs :

5 + 4 - 3 / 2 * 1 = 8

5 + (4 - 3) / 2 * 1 = 5

Les opérateurs fonctionnent de la même manière lorsqu'ils sont appliqués à des variables qu'à des valeurs littérales. Les opérateurs d'affectation peuvent être utilisés en remplacement de n'importe quelle opération mathématique en utilisant n'importe quel symbole arithmétique et égal ensemble. Par exemple, si nous voulions multiplier une variable, vous pourriez utiliser le code suivant :

```
int x = 32;
x = x * 2;
```

La deuxième façon de procéder, alternative, est présentée ici :

```
int x = 32 ;
x *= 2 ;
```

Le symbole égal est également considéré comme un opérateur d'affectation dans C#. Les autres symboles d'affectation suivent le même schéma syntaxique que notre exemple de multiplication précédent : +=, -=, et /= pour ajouter et assigner, soustraire et assigner, et diviser et assigner, respectivement.

Les chaînes de caractères sont un cas particulier en ce qui concerne les opérateurs, car elles peuvent utiliser le symbole d'addition pour créer un texte en patchwork, comme suit : `string NomComplet = "Toto " + "Dupont" ;`

Notez que les opérateurs arithmétiques ne fonctionnent pas avec tous les types de données. Par exemple, les opérateurs * et / ne fonctionnent pas sur les chaînes de caractères, et aucun de ces opérateurs ne fonctionne sur les booléens. Ayant appris que les types ont des règles qui régissent le type d'opérations et d'interactions qu'ils peuvent avoir, nous allons tenter notre

DEFINITION DES METHODES

Dans le chapitre précédent, nous avons brièvement abordé le rôle que jouent les méthodes dans nos programmes, à savoir qu'elles stockent et exécutent des instructions, tout comme les variables stockent des valeurs. Nous devons maintenant comprendre la syntaxe des déclarations de méthodes et la manière dont elles conduisent l'action et le comportement dans nos classes.

Comme pour les variables, les déclarations de méthodes ont leurs exigences de base, qui sont les suivantes :

- Le type de données qui seront renvoyées par la méthode (les méthodes ne doivent pas toutes renvoyer quelque chose, donc cela peut être void).
- Un nom unique, commençant par une majuscule
- Une paire de parenthèses après le nom de la méthode
- Une paire de parenthèses marquant le corps de la méthode (où les instructions sont stockées)

DECLARATION DES METHODES

Les méthodes peuvent également disposer des quatre mêmes modificateurs d'accès que les variables, ainsi que des paramètres d'entrée. Les paramètres sont des variables qui peuvent être transmises aux méthodes et auxquelles on peut accéder à l'intérieur de celles-ci. Le nombre de paramètres d'entrée que vous pouvez utiliser n'est pas limité, mais chacun doit être séparé par une virgule, indiquer son type de données et avoir un nom unique.

Exemple d’une méthode (équivaut à une fonction) qui retourne un résultat dans la variable **result** :

```
accessModifier returnType UniqueName(parameterType parameterName)
{
   // Corps de la méthode
   returnType result ;
   // inst 1
   // inst 2
   ……
   return result ;
}
```

Exemple d’une méthode qui ne retourne rien (équivaut à une procédure) :

```
accessModifier void UniqueName(parameterType parameterName)
{
   // Corps de la méthode

   // inst 1
   // inst 2
   ……
}
```

CONVENTIONS D'APPELLATION

Comme les variables, les méthodes ont besoin de noms uniques et significatifs pour les distinguer dans le code. Les méthodes conduisent à des actions, c'est donc une bonne pratique de les nommer en gardant cela à l'esprit. Par exemple, FonctionRepartition() ou DensiteProba() ressemble à une commande, ce qui se lit bien lorsque vous l'appelez dans un script, alors qu'un nom tel que Resume() ou FaitLe() est insipide et ne donne pas une image très claire de ce que la méthode va accomplir. Comme les variables, les noms de méthodes sont écrits dans la casse Pascal.

EXECUTION SEQUENTIELLE DU CODE

Nous avons vu que les lignes de code s'exécutent séquentiellement dans l'ordre où elles sont écrites L'appel d'une méthode demande au programme de faire un détour par les instructions de la méthode, de les exécuter une à une, puis de reprendre l'exécution séquentielle à l'endroit où la méthode a été appelée.

SPECIFICATION DES PARAMETRES

Dans les méthodes CalculAge(), nous avons utilisé des paramètres.

Dans un script nommé testC, réfléchissez à une classe Personne : quelles membres (propriétés et méthodes) pourriez-vous imaginer. Pour les propriétés, pensez à ce qui est nécessaire pour la décrire en tant qu'être humain, personnage de jeu, etc … Pour les méthodes, que fait la personne comme action, est-ce un étudiant ? un enseignant ? un joueur ? un personnage non joueur ? (amical, ennemi) : de quelles paramètres la méthode a-t-elle besoin pour exécuter l'action que vous avez défini.

EXEMPLE (SOURCE CHATGPT)

```
using UnityEngine;
public class Player : MonoBehaviour
{
    // Champs privés
    private string playerName;
    private int playerLevel;
    private int playerHealth;

    // Propriétés publiques en lecture seule
    public string PlayerName => playerName;
    public int PlayerLevel => playerLevel;
    public int PlayerHealth => playerHealth;

    // Méthode publique pour prendre des dégâts
    public void TakeDamage(int damage)
    {
        playerHealth -= damage;
        if (playerHealth <= 0)
        {
            Die();
        }
    }

    // Méthode publique pour tuer le joueur
    public void Die()
    {
        Destroy(gameObject);
    }

    // Méthode publique pour augmenter le niveau du joueur
    public void LevelUp()
    {
        playerLevel++;
        playerHealth = Mathf.RoundToInt(playerHealth * 1.5f);
    }

    // Méthode publique pour initialiser le joueur avec un nom, un niveau et une santé donnés
    public void InitializePlayer(string name, int level, int health)
    {
        playerName = name;
        playerLevel = level;
        playerHealth = health;
    }
}
```

Dans cet exemple, nous avons déclaré une classe Player qui hérite de la classe MonoBehaviour du moteur Unity. La classe contient des champs privés pour stocker le nom, le niveau et la santé du joueur, ainsi que des propriétés publiques en lecture seule pour permettre l'accès aux champs sans permettre la modification directe.

Nous avons également défini plusieurs méthodes publiques pour effectuer des actions sur le joueur, telles que prendre des dégâts, mourir, augmenter de niveau et initialiser le joueur avec des valeurs spécifiques. Les méthodes sont toutes publiques afin qu'elles puissent être appelées depuis d'autres classes.

En utilisant cette classe, nous pouvons créer un objet joueur dans le moteur Unity et l'initialiser en appelant la méthode InitializePlayer avec des valeurs appropriées. Ensuite, nous pouvons appeler d'autres méthodes pour effectuer des actions sur le joueur en réponse à des événements dans le jeu, tels que prendre des dégâts, augmenter de niveau ou mourir.

En résumé, une classe pour un joueur en utilisant le moteur Unity peut contenir des propriétés, des champs et des méthodes pour stocker et manipuler les données du joueur. Les méthodes peuvent être utilisées pour effectuer des actions sur le joueur en réponse à des événements dans le jeu, tels que prendre des dégâts, augmenter de niveau ou mourir.

SPECIFICATION DES VALEURS DE RETOUR

En plus d'accepter des paramètres, les méthodes peuvent renvoyer des valeurs de n'importe quel type C#.

Les méthodes dont le type de retour est void peuvent toujours utiliser le mot-clé return sans qu'aucune valeur ou expression ne leur soit attribuée. Lorsque la ligne contenant le mot-clé return est atteinte, la méthode cesse d'être exécutée. Ceci est utile dans les cas

où vous souhaitez vérifier l'existence d'une ou plusieurs valeurs avant de continuer ou vous prémunir contre les plantages de programme.

UTILISATION DES VALEURS DE RETOUR

Lorsqu'il s'agit d'utiliser les valeurs de retour, deux approches sont possibles :

- Créer une variable locale pour capturer (stocker) la valeur retournée.
- Utilisez la méthode d'appel elle-même comme substitut de la valeur renvoyée, en l'utilisant comme une variable. La méthode d'appel est la ligne de code qui déclenche les instructions, qui, dans notre exemple, serait GenerateCharacter("Spike", CharacterLevel). Vous pouvez même passer une méthode d'appel à une autre méthode en tant qu'argument si nécessaire.

La première option est préférée dans la plupart des cercles de programmation pour sa lisibilité. Le fait de jeter des appels de méthode sous forme de variables peut rapidement devenir problématique, surtout lorsque nous les utilisons comme arguments dans d'autres méthodes.

EXEMPLE : CONVERSION CELSIUS EN FAHRENHEIT

> Ajoutez un script testConvDegCelsius.
> Ajoutez une méthode qui retourne les degrés Fahrenheit avec les degrés Celsius en paramètre.

La méthode :

```
public float ConvertCelsiusToFahrenheit(float degC)
{
    float degF;
    degF = (degC * 1.8f) + 32f;
    return degF;
}
```

La méthode pourra être appelée en dehors de la classe, on lui passera le paramètre de type float degC, on utilise une variable locale degF de type float pour faire le calcul. La méthode retournera cette variable locale.

Appeler la méthode :

```
void Start()
{
    float DegC = 30.0f;
    // Pour faire un test simple, appeler la méthode et vérifier ce qui se passe, on peut simplement faire :
    Debug.Log(ConvertCelsiusToFahrenheit(DegC));
   // déclarer une variable locale pour recueillir le résultat, puis afficher la variable DegF
   float DegF = ConvertCelsiusToFahrenheit(DegC);
  Debug.Log(DegF) ;
}
```

> Modifiez le code pour afficher degC et degF dans l'inspecteur
> Test n°1 : La conversion sera faite seulement au démarrage de la scène.
> Test n°2 : La conversion sera faite en continu.

DISSEQUER LES METHODES COURANTES D'UNITY

Nous avons vu les méthodes par défaut les plus courantes qui sont livrées avec un script Unity C# : Start() et Update(). Contrairement aux méthodes que nous définissons nous-mêmes, les méthodes appartenant à la classe MonoBehaviour sont appelées automatiquement par le moteur Unity selon leurs règles respectives. Dans la plupart des cas, il est important d'avoir au moins une méthode MonoBehaviour dans un script pour lancer votre code.

Vous avez fait à la fin du chapitre 1 une recherche dans la documentation Unity : on y trouve une liste assez impressionnante de méthodes disponibles. Il est important de regarder l'ordre dans lequel les méthodes sont exécutées par UNITY.

Lorsque l'on veut apprendre la programmation de base en C#, Unity nous offre un environnement de développement très simple et puissant et nous n'avons pas besoin d'approfondir plus que nécessaire ce qui est disponible via MonoBehaviour.

Lorsque nous voulons aller plus loin en programmation, profiter de MonoBehaviour avec la compréhension des objets et composants est vraiment intéressant. Unity est un véritable outil de développement performant, accessible gratuitement, énormément documenté et utilisable avec d'autres langages ou outils : appeler des scripts python, aller chercher sur un serveur Raspberry des données de capteur via MQTT (ou autre protocole réseau), construire une interface utilisateur. On peut le voir comme un fédérateur ou une alternative à d'autres outils moins bien suivis, moins fiables, plus difficiles à mettre en œuvre. On peut grâce à des scripts communiquer directement avec Arduino, même si il sera plus pertinent de coder l'acquisition de données en

langage Arduino et rendre disponible les données avec Unity. De manière professionnelle, les applications industrielles sont bien sûr très développées dans tous les domaines (Réalité Virtuelle ou augmentée, modes opératoirs, simulation, etc).

Il faut progresser lentement et mettre en œuvre des choses simples : nous allons essayer de séparer aussi ce que l'on apprend au niveau de la programmation ou des mécanismes du moteur Unity.

LA METHODE START()

Unity appelle la méthode Start() sur la première image où un script est activé pour la première fois. Comme les scripts MonoBehaviour sont presque toujours attachés à des **GAMEOBJECTS** dans une scène, leurs scripts attachés sont activés en même temps qu'ils sont chargés lorsque vous appuyez sur **Play**. Dans notre projet, vous avez associé les scripts à un ou plusieurs objets : ils sont activés ou non afin de ne pas surcharger notre fenêtre **CONSOLE**.

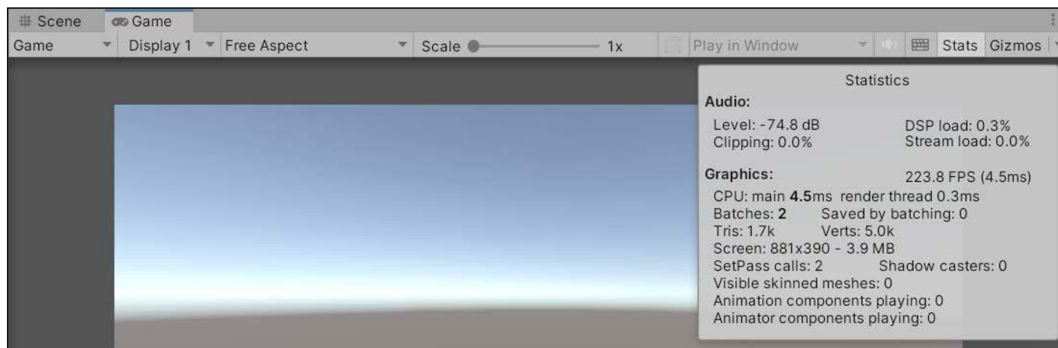
Start ne se déclenche qu'une seule fois, ce qui en fait un excellent outil pour afficher des informations ponctuelles sur la console et faire des essais de code.

LA METHODE UPDATE()

Lors de l'exécution du jeu, la fenêtre de la **scène** est affichée plusieurs fois par seconde, ce que l'on appelle le **taux de rafraîchissement** ou le nombre **d'images par seconde** (**FPS)**.

Après l'affichage de chaque image, la méthode Update() est appelée par Unity, ce qui en fait l'une des méthodes les plus exécutées dans votre jeu. Elle est donc idéale pour détecter les entrées de souris et de clavier ou pour exécuter la logique de jeu.

Pour connaître le nombre de FPS sur votre machine, lancez la lecture dans Unity et cliquez sur l'onglet **Stats** dans le coin supérieur droit de l'affichage du **jeu :**



FLUX DE CONTROLE ET COLLECTIONS

L'une des principales fonctions d'un ordinateur est de contrôler ce qui se passe lorsque des conditions prédéterminées sont remplies. Lorsque vous cliquez sur un dossier, vous vous attendez à ce qu'il s'ouvre ; lorsque vous tapez au clavier, vous vous attendez à ce que le texte reflète votre frappe. L'écriture du code C# pour les applications ou les jeux n'est pas différente : ils doivent tous deux se comporter d'une certaine manière dans un état donné, et d'une autre manière lorsque les conditions changent. En termes de programmation, c'est ce qu'on appelle le flux de contrôle, ce qui est approprié car il contrôle le flux d'exécution du code dans différents scénarios.

En plus de travailler avec des instructions de contrôle, nous allons nous pencher sur les types de données de collection. Les collections sont une catégorie de types qui permettent de stocker plusieurs valeurs et groupes de valeurs dans une seule variable. Ce chapitre est divisé en plusieurs parties :

- Déclarations de sélection
- Travailler avec des collections de tableaux, de dictionnaires et de listes
- Instructions d'itération avec les boucles for, foreach et while
- Correction des boucles infinies

DECLARATIONS DE SELECTION

Les problèmes de programmation les plus complexes peuvent souvent se résumer à un ensemble de choix simples qu'un jeu ou un programme évalue et sur lesquels il agit. Visual Studio et Unity n'étant pas en mesure de faire ces choix par eux-mêmes, c'est à vous qu'il revient d'écrire ces décisions.

Les instructions de sélection if-else et switch vous permettent de spécifier des chemins de dérivation, basés sur une ou plusieurs conditions, et les actions que vous souhaitez entreprendre dans chaque cas. Traditionnellement, ces conditions sont les suivantes

- Détection de l'entrée de l'utilisateur
- Évaluation d'expressions et logique booléenne
- Comparaison de variables ou de valeurs littérales

Dans la section suivante, vous commencerez par la plus simple de ces instructions conditionnelles, if-else.

L'INSTRUCTION IF-ELSE

Les instructions if-else sont le moyen le plus courant de prendre des décisions dans le code. Dépouillée de toute sa syntaxe, l'idée de base d'une instruction if-else est la suivante : *Ifi m ; conKiíion is mcí, cxccuíc íhis block ofi coKc ; ifi ií's noí, cxccuíc íhis oíhcu block ofi coKc.* Considérez les instructions if-else comme des portes, les conditions étant leurs clés. Pour passer, la clé doit être valide. Dans le cas contraire, l'entrée sera refusée et le code sera envoyé à la prochaine porte possible. Examinons la syntaxe de déclaration d'une de ces portes.

Une instruction if-else valide requiert les éléments suivants :

- Le mot-clé if au début de la ligne
- Une paire de parenthèses pour maintenir la condition
- Le corps d'une déclaration à l'intérieur de parenthèses courbes Cela ressemble à

```
ceci :si(la condition est vraie)
 {
      Exécuter un bloc de code
 }
```

En option, une instruction else peut être ajoutée pour enregistrer l'action que vous souhaitez entreprendre lorsque l'instruction if échoue. Les mêmes règles s'appliquent à l'instruction else :

```
autre
{
      Exécuter un autre bloc de code
}
```

Sous forme de plan, la syntaxe se lit presque comme une phrase, c'est pourquoi c'est l'approche recommandée :

```
si(la condition est vraie)
{
      Exécuter ce bloc de
code } autre
{
      Exécuter ce bloc de code
}
```

Comme il s'agit d'excellentes introductions à la pensée logique, du moins en programmation, nous allons décomposer les trois variations if-else plus en détail. L'ajout de ce code à votre script LearningCurve.cs est facultatif pour l'instant, car nous entrerons dans les détails dans les exercices suivants :

1. Une simple instruction if peut exister par elle-même dans les cas où vous ne vous souciez pas de ce qui se passe si la condition n'est pas remplie. Dans l'exemple suivant, si hasDungeonKey est défini à true, un journal de débogage sera imprimé ; s'il est défini

```
public class LearningCurve : MonoComportement
{ public bool hasDungeonKey = true ;

      void Start()
      { if(hasDungeonKey)
            {
                  Debug.Log("Vous possédez la clé sacrée - entrez") ;
            }
      }
}
```

Lorsqu'on dit d'une condition qu'elle est remplie, je veux dire qu'elle est évaluée à vrai, ce qui est souvent appelé une condition de réussite.

à false , aucun code ne sera exécuté :

2. Ajoutez une instruction else dans les cas où une action doit être entreprise, que la condition soit vraie ou fausse. Si hasDungeonKey était faux, l'instruction if échouerait et l'exécution du code passerait à l'instruction else :

```
public class LearningCurve : MonoComportement
{ public bool hasDungeonKey = true ;

      void Start()
      { if(hasDungeonKey)
            {
                  Debug.Log("Vous possédez la clé sacrée - entrez") ;
            } autre
            {
                  Debug.Log("Vous n'avez pas encore fait vos preuves") ;
            }
      }
}
```

3. Dans les cas où vous avez besoin de plus de deux résultats possibles, ajoutez une instruction else-if avec ses parenthèses, ses conditions et ses crochets. Il est préférable de le montrer plutôt que de l'expliquer, ce que nous ferons dans l'exercice suivant.

Gardez à l'esprit que les instructions if peuvent être utilisées seules, mais que les autres instructions ne peuvent pas exister seules. Vous pouvez également créer des conditions plus complexes à l'aide d'opérations mathématiques de base, telles que :

- > (plus grand que)
- < (moins que)
- >= (supérieur ou égal)
- <= (inférieur ou égal)
- == (équivalent)

Par exemple, une condition de (2 > 3) renverra false et échouera, tandis qu'une condition de (2 < 3) renverra true et réussira. Écrivons une instruction if-else qui vérifie la quantité d'argent dans la poche d'un personnage, en renvoyant des journaux de débogage différents pour trois cas différents : plus de 50, moins de 15, et n'importe quoi d'autre :

1. Ouvrez LearningCurve et ajoutez une nouvelle variable publique int nommée CurrentGold. Fixez sa valeur entre 1 et 100 :

```
public int CurrentGold = 32 ;
```

2. Créez une méthode publique sans valeur de retour, appelée Thievery :

```
public void Thievery()
{
}
```

3. Dans la nouvelle fonction, ajoutez une instruction if pour vérifier si CurrentGold est supérieur à 50, et imprimez un message sur la console si c'est le cas :

```
if(CurrentGold > 50)
{
    Debug.Log("Vous roulez dedans !") ;
}
```

4. Ajoutez une instruction else-if pour vérifier si CurrentGold est inférieur à 15 avec un journal de débogage différent :

```
else if (CurrentGold < 15)
{
    Debug.Log("Il n'y a pas grand-chose à voler...") ;
}
```

5. Ajoutez une instruction else sans condition et un protocole final par défaut :

```
autre
{
    Debug.Log("On dirait que votre sac à main est au bon endroit") ; }
```

6. Appelez la méthode Thievery à l'intérieur de Start :

```
void Start()
{
    Vol() ;
}
```

7. Enregistrez le fichier, vérifiez que votre méthode correspond au code ci-dessous et cliquez sur **Play** :

```
public void Thievery()
{ if(CurrentGold > 50)
    {
        Debug.Log("Vous roulez dedans !") ;
    } else if (CurrentGold < 15)
    {
        Debug.Log("Il n'y a pas grand-chose à voler...") ;
    } autre
    {
                                Debug.Log("On dirait que votre sac à main est au bon endroit")
        ;
    }

}
```

Avec CurrentGold fixé à 32 dans mon exemple, nous pouvons décomposer la séquence de code comme suit :

1. L'instruction if et le journal de débogage sont ignorés parce que CurrentGold n'est pas supérieur à 50.

2. L'instruction else-if et le journal de débogage sont également ignorés parce que CurrentGold n'est pas inférieur à 15.
3. Comme 32 n'est ni inférieur à 15 ni supérieur à 50, aucune des conditions précédentes n'a été remplie, l'instruction else s'exécute et le troisième journal de débogage s'affiche :

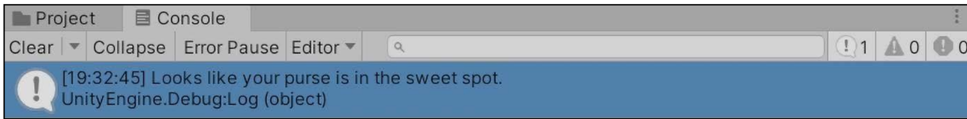


Figure 4.1 : Capture d'écran de la console montrant la sortie de débogage

Pour uuoviKc n comulcíc vicw ofi íhc Uníí ; cKiíou, nll ouru scuccnshoís nuc ínkcn in fiullLscuccn moKc. Pour les images en couleur de tous les livres, utilisez le lien suivant : https://packt.link/7yy5V.

Après avoir essayé d'autres valeurs pour CurrentGold, voyons ce qui se passe si nous voulons tester une condition d'échec.

Utilisation de l'opérateur NOT

Les cas d'utilisation ne nécessitent pas toujours la vérification d'une condition positive, ou vraie, et c'est là que l'opérateur NOT entre en jeu. Représenté par un simple point d'exclamation, l'opérateur NOT permet aux conditions négatives, ou fausses, d'être remplies par les instructions if ou else-if. Cela signifie que les conditions suivantes sont identiques :

```
if(variable == false)

// ET

if(!variable)
```

Comme vous le savez déjà, vous pouvez vérifier les valeurs booléennes, les valeurs littérales ou les expressions dans un formulaire if condition. L'opérateur NOT doit donc naturellement être adaptable.

Prenons l'exemple suivant de deux valeurs négatives différentes, hasDungeonKey et Type d'arme, utilisé dans une instruction if :

```
public class LearningCurve : MonoBehaviour
{ public bool hasDungeonKey = false ; public string
    weaponType = "Arcane Staff" ;

    void Start()
    { if(!hasDungeonKey)
        {
            Debug.Log("Vous ne pouvez pas entrer sans la clé sacrée") ;
        }

        if(weaponType ! = "Longsword")
        {
                Debug.Log("Vous ne semblez pas avoir le bon type de
arme...") ;
        }
    }
}
```

Nous pouvons évaluer chaque affirmation comme suit :

- La première affirmation peut être traduite par : *Ifi* hasDungeonKey *is* false*, íhc* if *sínícmcní cvnlunícs ío* true *nnK cxccuícs iís coKc block.*

Si vous vous demandez comment une valeur fausse peut être évaluée comme vraie, pensez-y de la manière suivante : l'instruction if ne vérifie pas si la valeur est vraie, mais si l'expression elle-même est vraie. hasDungeonKey peut être définie comme fausse, mais c'est ce que nous vérifions, donc elle est vraie dans le contexte de la condition if.

- La deuxième affirmation peut être traduite par : *Ifi íhc síuing vnluc ofi* weaponType n'est pas égal.

  ío Longsword*, íhcn cxccuíc íhis coKc block*.

  Si vous insérez ce code dans LearningCurve.cs, les résultats correspondront à la capture d'écran suivante :

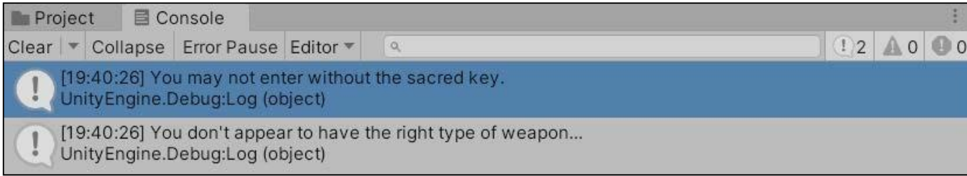


Figure 4.2 : Capture d'écran de la console montrant la sortie de l'opérateur NOT

Cependant, si vous êtes toujours confus, copiez le code que nous avons étudié dans cette section dans LearningCurve.cs et jouez avec les valeurs des variables jusqu'à ce que cela ait un sens.

Jusqu'à présent, nos conditions de branchement étaient relat#vement simples, mais le langage C permet également d'imbriquer les instructions conditionnelles les unes dans les autres pour obtenir des situations plus complexes.

Imbrication des déclarations

L'une des fonctions les plus intéressantes des instructions if-else est qu'elles peuvent être imbriquées les unes dans les autres, créant ainsi des itinéraires logiques complexes dans votre code. En programmation, on les appelle des arbres de décision. Comme dans un vrai couloir, il peut y avoir des portes derrière d'autres portes, créant ainsi un labyrinthe de possibilités :

```
public class LearningCurve : MonoBehaviour
{ public bool weaponEquipped = true ; public string
    weaponType = "Longsword" ;

    void Start()
    { if(weaponEquipped)
        { if(weaponType == "Longsword")
            {
                Debug.Log("Pour la reine !") ; }
        }




    autre
    {
        Debug.Log("Les poings ne vont pas fonctionner contre les armures...") ;
    }
}

}
```

Décortiquons l'exemple précédent :

1. Tout d'abord, une instruction if vérifie si nous avons weaponEquipped. À ce stade, le code ne se préoccupe que de savoir si c'est vrai, et non pas de quel type d'arme il s'agit.
2. La deuxième instruction if vérifie le type d'arme et imprime le journal de débogage associé.
3. Si la première instruction if est évaluée à false, le code passe à l'instruction else et à son journal de débogage. Si la deuxième instruction if évalue false, rien n'est imprimé car il n'y a pas d'instruction else.

La responsabilité du traitement des résultats logiques incombe à 100 % au programmeur. C'est à vous de déterminer les branches ou les résultats possibles de votre code.

Ce que vous avez appris jusqu'à présent vous permettra de résoudre sans problème des cas d'utilisation simples. Cependant, vous aurez rapidement besoin d'instructions plus complexes, et c'est là que l'évaluation de conditions multiples entre en jeu.

Évaluation de conditions multiples

Outre l'imbrication des instructions, il est également possible de combiner plusieurs vérifications de conditions en une seule instruction if ou else-if à l'aide des opérateurs logiques AND OR :

- L'opérateur ET est représenté par deux caractères esperluette, &&. Toute condition utilisant l'opérateur AND signifie que toutes les conditions doivent être évaluées comme vraies pour que l'instruction if s'exécute.
- L'opérateur OR est représenté par deux caractères en forme de pipe, ||. Une instruction if utilisant l'opérateur OR s'exécute si une ou plusieurs de ses conditions sont vraies.
- Les conditions sont toujours évaluées de gauche à droite.

Dans l'exemple suivant, l'instruction if a été mise à jour pour vérifier la présence de l'armeEquipped et weaponType, qui doivent tous deux être vrais pour que le bloc de code s'exécute :

```
if(weaponEquipped && weaponType == "Longsword")
{
        Debug.Log("Pour la reine !") ;
}
```

Les opérateurs AND OR peuvent être combinés pour vérifier plusieurs conditions dans n'importe quel ordre. Il n'y a pas non plus de limite au nombre d'opérateurs que vous pouvez combiner. Veillez toutefois, lorsque vous les utilisez ensemble, à ne pas créer des conditions logiques qui ne s'exécuteront jamais.

Il est temps de mettre à l'épreuve tout ce que nous avons appris jusqu'à présent sur les instructions "si". Relisez donc cette section si vous en avez besoin, puis passez à la section suivante.

Cimentons ce sujet avec une petite expérience de coffre au trésor :

1. Déclarer trois variables en haut de LearningCurve : PureOfHeart est un bool et doit valoir true, HasSecretIncantation est également un bool et doit valoir false, et RareItem est une chaîne de caractères dont la valeur est laissée à votre

   ```
   public bool PureOfHeart = true ; public bool HasSecretIncantation = false ; public string
   RareItem = "Relic Stone" ;
   ```

   appréciation :

2. Créez une méthode publique sans valeur de retour appelée OpenTreasureChamber :

   ```
   public void OpenTreasureChamber()
   {
   }
   ```

3. Dans OpenTreasureChamber, déclarez une instruction if-else pour vérifier si PureOfHeart est vrai *nnK* que RareItem correspond à la valeur de la chaîne que vous lui avez attribuée :

   ```
   if(PureOfHeart && RareItem == "Relic Stone")
   {
   }
   ```

4. Créer une instruction if-else imbriquée dans la première, en vérifiant si HasSecretIncantation est faux :

   ```
   if(!HasSecretIncantation)
   {
       Debug.Log("Vous avez l'esprit, mais pas la connaissance") ; }
   ```

5. Ajouter des journaux de débogage pour chaque cas if-else.

6. Appeler la méthode OpenTreasureChamber à l'intérieur de Start :

```
void Start()
{
    OpenTreasureChamber() ;
}
```

7. Enregistrez, vérifiez que votre code correspond au code ci-dessous et cliquez sur **Play** :

```
public class LearningCurve : MonoBehaviour
{ public bool PureOfHeart = true ; public bool
    HasSecretIncantation = false ; public string RareItem
    = "Relic Stone" ;

    void Start()
    {
        OpenTreasureChamber() ;
    }

    public void OpenTreasureChamber()
    { if(PureOfHeart && RareItem == "Relic Stone")
        { if(!HasSecretIncantation)
            {
                    Debug.Log("Vous avez l'esprit, mais pas la
connaissances")
             ;
            } autre
            {
                Debug.Log("Le trésor est à toi, digne héros !") ; } } autre
        {
            Debug.Log("Revenez quand vous aurez ce qu'il faut") ;
        }
    }
}
```

Si vous faites correspondre les valeurs des variables à la capture d'écran précédente, le journal de débogage de l'instruction if imbriquée sera imprimé. Cela signifie que notre code a passé la première instruction if en vérifiant deux conditions, mais qu'il a échoué à la troisième :

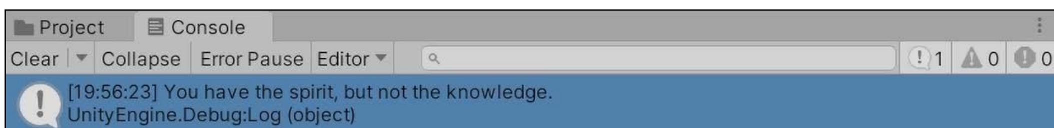


Figure 4.3 : Capture d'écran de la sortie de débogage dans la console

Vous pourriez vous arrêter là et utiliser des instructions if-else encore plus grandes pour tous vos besoins conditionnels, mais cela ne sera pas efficace à long terme. Une bonne programmation consiste à utiliser le bon outil pour le bon travail, et c'est là que l'instruction switch entre en jeu.

L'INSTRUCTION DE COMMUTATION

Les instructions if-else sont un excellent moyen d'écrire une logique de décision. Toutefois, lorsqu'il y a plus de trois ou quatre actions ramifiées, elles ne sont tout simplement pas réalisables. En un rien de temps, votre code peut finir par ressembler à un nœud emmêlé difficile à suivre et à mettre à jour.

Les instructions switch prennent en compte les expressions et nous permettent d'écrire des actions pour chaque résultat possible, mais dans un format beaucoup plus concis que les instructions if-else.

Les déclarations de commutation doivent comporter les éléments suivants :

- Le mot-clé switch suivi d'une paire de parenthèses contenant sa condition
- Une paire de parenthèses bouclées
- Un énoncé de cas pour chaque chemin possible se terminant par deux points : lignes de code ou méthodes individuelles, suivies du mot-clé break et d'un point-virgule.
- Un énoncé de cas par défaut se terminant par deux points : lignes de code ou méthodes individuelles, suivies du mot-clé break et d'un point-virgule.

Sous forme de plan, il se présente comme suit :

```
switch(matchExpression)
{ case matchValue1 : Exécution du
        bloc de code break ;

        case matchValue2 : Exécution du
                bloc de code break ;
```

```
        par défaut :
            Exécution d'une rupture de bloc
            de code ;
}
```

Les mots-clés mis en évidence dans le schéma précédent sont les éléments importants. Lorsqu'une instruction case est définie, tout ce qui se trouve entre les deux points et le motclé break agit comme le bloc de code d'une instruction if-else. Le mot-clé break indique simplement au programme de quitter entièrement l'instruction switch une fois que le cas sélectionné s'est déclenché. Voyons maintenant comment l'instruction détermine le cas à exécuter, ce que l'on appelle le filtrage.

Correspondance des modèles

Dans les instructions de commutation, la correspondance des motifs fait référence à la manière dont une **expression de correspondance** est validée par rapport à plusieurs instructions de cas. Une expression de correspondance peut être de n'importe quel type qui n'est pas null ou nothing ; toutes les valeurs de l'instruction case doivent correspondre au type de l'expression de correspondance.

Par exemple, si nous avons une instruction switch qui évalue une variable entière, chaque cas devrait spécifier une valeur entière pour qu'elle soit vérifiée.

L'instruction Case dont la valeur correspond à l'expression est exécutée. Si aucune est trouvé, c'est le cas par défaut qui se déclenche. Essayons-le par nous-mêmes !

Cela fait beaucoup de syntaxe et d'informations nouvelles, mais il est utile de les voir en action. Créons un simple une déclaration de commutation pour les différentes actions qu'un personnage pourrait entreprendre :

1. Créez une nouvelle variable publique de type chaîne de caractères nommée CharacterAction et donnez-lui la valeur suivante "Attaque" :

```
public string CharacterAction = "Attack" ;
```

2. Créez une méthode publique sans valeur de retour appelée PrintCharacterAction :

```
public void PrintCharacterAction()
{
}
```

3. Déclarez une instruction switch dans la nouvelle méthode et utilisez CharacterAction comme expression de correspondance :

```
switch(CharacterAction)
{
}
```

4. Créez deux instructions de cas pour "Heal" et "Attack" avec des journaux de débogage différents. N'oubliez pas d'inclure le mot-clé break à la fin de chacune

```
cas "Guérir" :
        Debug.Log("Potion envoyée.")
        ; break ; case
"Attack" :
        Debug.Log("Aux armes
        !") ; break ;
```

d'entre elles :

5. Ajouter un cas par défaut avec un journal de débogage et un break :

```
par défaut :
        Debug.Log("Boucliers levés") ; break ;
```

6. Appeler la méthode PrintCharacterAction à l'intérieur de Start :

```
void Start()
{
        PrintCharacterAction() ;
}
```

7. Enregistrez le fichier, assurez-vous que votre code correspond à la capture d'écran ci-dessous et cliquez sur **Play** :

```
public string CharacterAction = "Attack" ;

void Start()
{
        PrintCharacterAction() ;
}

public void PrintCharacterAction()
{ switch(CharacterAction)
        { cas "Guérir" :
                        Debug.Log("Potion envoyée.") ; break ;
                case "Attack" :
                        Debug.Log("Aux armes
                        !") ;
                            break ;
                    default :
                            Debug.Log("Boucliers levés") ; break ;
            }
    }
```

Comme CharacterAction est défini sur "Attack", l'instruction switch exécute le deuxième cas et affiche son journal de débogage :

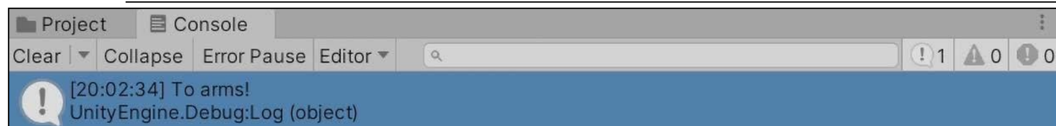


Figure 4.4 : Capture d'écran de la sortie de l'instruction de commutation dans la console

Changez CharacterAction en "Heal" ou en une action non définie pour voir la première action.
et les cas de défaut en action.

Il peut arriver que vous ayez besoin de plusieurs cas de commutation, mais pas tous, pour effectuer la même action. C'est ce que l'on appelle les **cas d'application** et c'est le sujet de notre prochaine section.

Cas de repli

Les instructions switch peuvent exécuter la même action pour plusieurs cas, de la même manière que nous avons spécifié plusieurs conditions dans une seule instruction if. Le terme pour cela est appelé fall-through ou, parfois, fall-through cases. Les cas d'application vous permettent de définir un seul ensemble d'actions pour plusieurs cas. Si un bloc de cas est vide ou contient du code sans le mot-clé break, il passe au cas directement inférieur. Cela permet de conserver un code de commutation propre et efficace, sans duplication des blocs de cas.

Les cas peuvent être écrits dans n'importe quel ordre, de sorte que la création de cas de passage augmente considérablement la lisibilité du code. et l'efficacité.

Simulons un scénario de jeu de table avec une instruction de commutation et un cas d'échec, où un Le jet de dés détermine le résultat d'une action spécifique :

1. Créez une variable publique int nommée DiceRoll et attribuez-lui la valeur 7 :

```
public int DiceRoll = 7 ;
```

2. Créez une méthode publique sans valeur de retour appelée RollDice :

```
public void RollDice()
{
}
```

3. Ajoutez une instruction switch avec DiceRoll comme expression de correspondance :

```
switch(DiceRoll)
{
}
```

4. Ajoutez trois cas pour les jets de dés possibles à 7, 15 et 20, avec une déclaration de cas par défaut à la fin.
5. Les cas 15 et 20 devraient avoir leurs propres journaux de débogage et instructions de rupture, tandis que le cas 7 devrait passer au cas 15 :

```
cas 7 :
cas 15 :
        Debug.Log("Dommages médiocres, pas mauvais.") ; break ;
cas 20 :
        Debug.Log("Coup critique, la créature tombe !") ; break ;
par défaut :
        Debug.Log("Vous avez complètement raté votre coup et vous êtes tombé sur la
        tête") ; break ;
```

6. Appeler la méthode RollDice à l'intérieur de Start :

```
void Start()
{
      RollDice() ;
}
```

7. Enregistrez le fichier et exécutez-le dans Unity.

Si vous voulez voir le cas de chute en action, essayez d'ajouter un journal de débogage au cas 7, mais sans le mot-clé break.

Avec un DiceRoll fixé à 7, l'instruction switch correspondra au premier cas, qui tombera et exécutera le cas 15 parce qu'il n'y a pas de bloc de code ni d'instruction break. Si vous donnez à DiceRoll la valeur 15 ou 20, la console affichera les messages correspondants, et toute autre valeur déclenchera l'exécution du cas par défaut à la fin de l'instruction :

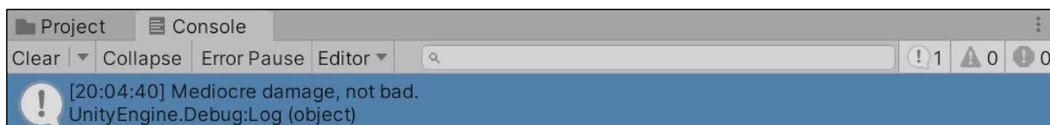


Figure 4.5 : Capture d'écran du code de l'instruction fall-through switch

Les instructions switch sont extrêmement puissantes et peuvent simplifier les logiques de décision les plus complexes. Si vous souhaitez approfondir la recherche de motifs de commutation, consultez : https://docs.microsoft.com/enus/dotnet/csharp/language-reference/ keywords/switch.

C'est tout ce que nous avons besoin de savoir sur la logique conditionnelle pour le moment. Revoyez donc cette section si nécessaire, puis testez vous sur le quiz suivant avant de passer aux collections !

POP QUIZ 1- SI, ET, OU MAIS

Testez vos connaissances en répondant aux questions suivantes :

1. Quelles sont les valeurs utilisées pour évaluer les instructions "if" ?
2. Quel opérateur peut transformer une condition vraie en condition fausse ou une condition fausse en condition vraie ?
3. Si deux conditions doivent être remplies pour que le code d'une instruction if s'exécute, quel opérateur logique utiliseriez-vous pour joindre les conditions ?
4. Si une seule des deux conditions doit être vraie pour que le code d'une instruction if soit exécuté, quel opérateur logique utiliseriez-vous pour joindre les deux conditions ?

N'oubliez pas de comparer vos réponses aux miennes dans l'annexe *Fou Quiz "nswcus"* pour voir où vous en êtes !

Ceci fait, vous êtes prêt à entrer dans le monde des types de données de collection. Ces types vont ouvrir un tout nouveau sous-ensemble de fonctionnalités de programmation pour vos jeux et vos programmes C # !

LES COLLECTIONS EN UN COUP D'ŒIL

Jusqu'à présent, nous n'avons eu besoin de variables que pour stocker une seule valeur, mais il existe de nombreuses situations dans lesquelles un groupe de valeurs est nécessaire. Les **types de collection** dans C # comprennent les tableaux, les dictionnaires et les listes. Chacun a ses forces et ses faiblesses, que nous aborderons dans les sections suivantes.

TABLEAUX

Les tableaux sont la collection la plus élémentaire offerte par C #. Il s'agit de conteneurs pour un groupe de valeurs, appelées "*clcmcnís*" dans la terminologie de la programmation, dont chacune peut être accédée ou modifiée individuellement :

- Les tableaux peuvent stocker n'importe quel type de valeur ; tous les éléments doivent être du même type.

- La longueur, ou le nombre d'éléments qu'un tableau peut avoir, est fixée lors de sa création et ne peut pas être modifiée.

  être modifiée par la suite.
- Si aucune valeur initiale n'est attribuée lors de sa création, chaque élément se verra attribuer une valeur par défaut. Les tableaux stockant des nombres prennent par défaut la valeur zéro, tandis que tous les autres types prennent la valeur null ou rien.

Les tableaux sont le type de collection le moins flexible en C . Cela s'explique principalement par le fait que les éléments ne peuvent pas être ajoutés ou supprimés une fois qu'ils ont été créés. Cependant, ils sont particulièrement utiles pour stocker des informations qui ne sont pas susceptibles de changer. Ce manque de flexibilité les rend plus rapides que les autres types de collection.

La déclaration d'un tableau est similaire à celle des autres types de variables avec lesquels nous avons travaillé, mais elle comporte quelques modifications :

- Les variables de type tableau nécessitent un type d'élément spécifié, une paire de crochets et un nom unique

  nom.
- Le mot-clé new est utilisé pour créer le tableau en mémoire, suivi du type de valeur et d'une autre paire de crochets. La zone de mémoire réservée correspond à la taille exacte des données que vous avez l'intention de stocker dans le nouveau tableau.
- Le nombre d'éléments que le tableau stockera se trouve à l'intérieur de la deuxième paire de crochets. Sous forme

  de plan, cela ressemble à ceci :

```
elementType[] name = new elementType[numberOfElements] ;
```

Prenons un exemple où nous devons stocker les trois meilleurs scores de notre jeu :

```
int[] TopPlayerScores = new int[3] ;
```

TopPlayerScores est un tableau d'entiers qui stocke trois éléments entiers. Comme nous n'avons pas ajouté de valeurs initiales, chacune des trois valeurs de TopPlayerScores vaut 0. Cependant, si vous modifiez la taille du tableau, le contenu du tableau d'origine est perdu, alors soyez prudent.

Vous pouvez assigner des valeurs directement à un tableau lors de sa création en les ajoutant à l'intérieur d'une paire de crochets à la fin de la déclaration de la variable. C dispose d#une méthode longue et d'une méthode courte pour effectuer cette opération, mais les deux

```
// Initialisateur Longhand int[] TopPlayerScores = new int[] {713,
549, 984} ;

// Initialisateur de raccourcis int[] TopPlayerScores = {
713, 549, 984 } ;
```

L'initialisation des tableaux à l'aide de la syntaxe abrégée est très courante, c'est pourquoi je l'utiliserai dans le reste du livre. Toutefois, si vous souhaitez vous rappeler les détails, n'hésitez pas à utiliser la syntaxe explicite de l'initialisateur en écriture longue, comme indiqué ci-dessus.

méthodes sont également valables :

Maintenant que la syntaxe de déclaration n'est plus un mystère, voyons comment les éléments d'un tableau sont stockés et accessibles.

Indexation et indices

Chaque élément d'un tableau est stocké dans l'ordre qui lui a été attribué, ce qui est appelé son **index**. Les tableaux sont indexés à zéro, ce qui signifie que l'ordre des éléments commence à 0 au lieu de 1. L'index d'un élément est sa référence ou son emplacement.

Dans TopPlayerScores, le premier entier, 452, est situé à l'index 0, 713 à l'index 1 et 984 à l'index 2 :

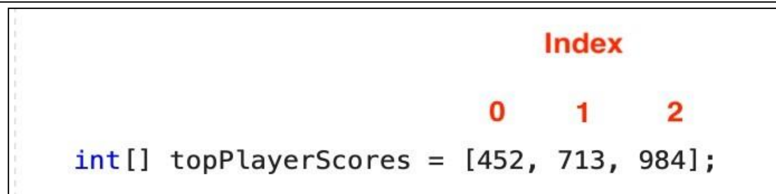


Figure 4.6 : Index des tableaux mis en correspondance avec leurs valeurs

Les valeurs individuelles sont localisées par leur indice à l'aide de l'**opérateur d'indice**, qui est une paire de crochets contenant l'indice des éléments.

Par exemple, pour récupérer et stocker le deuxième élément du tableau TopPlayerScores, nous utiliserons le nom du tableau suivi de crochets d'indice et de l'index 1 :

```
// La valeur du score est fixée à 713 int score = TopPlayerScores[1]
;
```

L'opérateur d'indice peut également être utilisé pour modifier directement la valeur d'un tableau comme n'importe quelle autre variable, ou même être passé comme une expression en soi :

```
TopPlayerScores[1] = 1001 ;
```

Les valeurs de TopPlayerScores seraient alors 452, 1001 et 984.

Tableaux multidimensionnels

Les tableaux sont également un excellent moyen de stocker des éléments sous la forme d'un tableau (lignes et colonnes dans le monde réel). On parle de tableaux multidimensionnels car chaque élément ajouté apporte une nouvelle dimension aux données. Les exemples de tableaux ci-dessus ne contiennent qu'un élément par index, ils sont donc unidimensionnels. Si nous voulions qu'un tableau contienne, par exemple, une coordonnée *x* et une

```
// Le tableau de coordonnées comporte 3 lignes et 2 colonnes int[,] Coordinates = new int[3,2] ;
```

coordonnée *;* dans chaque élément, comme en cours de mathématiques au collège, nous pourrions créer un tableau bidimensionnel comme suit :

Remarquez que nous avons utilisé une virgule à l'intérieur des crochets pour indiquer que le tableau est bidimensionnel et que nous avons ajouté deux champs d'initialisation, qui sont également séparés par une virgule.

Nous pouvons également initialiser directement un tableau multidimensionnel avec des valeurs, en créant un tableau de *x* et de *Des* coordonnées comme celles qui précèdent pourraient être abrégées comme suit :

```
        int[,] Coordinates = new int[3,2] {
         {5,4},
         {1,7},
         {9,3}
    } ;
```

Vous pouvez voir que nous avons trois lignes, ou éléments, chacune contenant 2 colonnes de données pour les valeurs *x* et *;* valeurs. Voici maintenant la partie délicate de la gymnastique mentale : vous devez

considérer les tableaux multidimensionnels comme des tableaux *de* tableaux. Dans l'exemple ci-dessus, chaque élément est toujours stocké à un index commençant par 0 et allant vers le haut, mais chaque élément est un tableau au lieu d'une valeur unique. Concrètement, la valeur 4 dans la première colonne de la première ligne est située à l'indice 0, et la valeur réelle de 4 est située au premier élément du tableau de cette ligne, soit 1 :

```
int[,] Coordinates = new int[3, 2]
{   Columns
     0  1
    {5, 4},   ——  Index 0 (or row 0)
    {1, 7},   ——  Index 1 (or row 1)
    {9, 3}    ——  Index 2 (or row 2)
};
```

Figure 4.7 : Tableau multidimensionnel mappé avec des index

Dans le code, nous utiliserions l'indice suivant, en commençant par l'indice de la ligne, suivi de l'indice de la ligne. l'index de la colonne :

```
// Recherche de la valeur dans la première ligne, première colonne int coordinateValue = Coordinates[0, 1] ;
```

La modification d'une valeur dans un tableau multidimensionnel se fait de la même manière que pour un tableau ordinaire : nous utilisons l'indice de la valeur que nous voulons mettre à jour, puis nous lui attribuons une nouvelle valeur :

```
// La valeur de la première ligne, première colonne est maintenant 10 Coordonnées [0, 1] = 10 ;
```

Un #tableau C peut avoir jusqu'à 32 dimensions, ce qui est beaucoup, mais les règles pour les créer sont les mêmes - ajouter une virgule supplémentaire pour chaque dimension à l'intérieur des crochets de type au début de la variable et une virgule supplémentaire et le nombre d'éléments dans l'initialisation. Par exemple, un tableau à trois dimensions

```
int[,,] Coordinates = new int[3,3,2] ;
```

Ceci est un peu avancé pour nos besoins, mais vous pouvez accéder à un code de tableau multidimensionnel plus complexe à l'adresse suivante : https://learn.microsoft.com/dotnet/csharp/ programmingguide/arrays/multidimensional-arrays.

ressemblerait à ceci :

Exceptions de portée

Lorsque les tableaux sont créés, le nombre d'éléments est fixé et immuable, ce qui signifie que nous ne pouvons pas accéder à un élément qui n'existe pas. Dans l'exemple TopPlayerScores, la longueur du tableau est de 3, et les indices valides sont donc compris entre 0 et 2.

Tout indice de 3 ou plus est hors de la portée du tableau et génère une erreur bien nommée Erreur IndexOutOfRangeException dans la console :

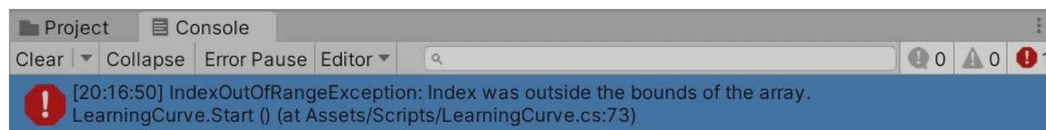


Figure 4.8 : Capture d'écran de l'exception "index hors plage".

Les bonnes habitudes de programmation imposent d'éviter les exceptions de plage en vérifiant si la valeur souhaitée se trouve dans la plage d'index d'un tableau, ce que nous verrons dans la section *lícuníion sínícmcnís.*

Vous pouvez toujours vérifier la longueur d'un tableau, c'est-à-dire le nombre d'éléments qu'il contient, à l'aide de la fonction Longueur de la propriété :

```
TopPlayerScores.Length ;
```

Les tableaux ne sont pas les seuls types de collection proposés par C. Dans la prochaine section, nous aborderons les listes, # qui sont plus flexibles et plus courantes dans le paysage de la programmation.

LISTES

Les listes sont étroitement liées aux tableaux, car elles rassemblent plusieurs valeurs du même type dans une seule variable. Elles sont beaucoup plus faciles à gérer lorsqu'il s'agit d'ajouter, de supprimer et de mettre à jour des éléments, mais leurs éléments ne sont pas stockés de manière séquentielle. Elles sont également mutables, ce qui signifie que vous pouvez modifier la longueur ou le nombre d'éléments que vous stockez sans écraser toute la variable. Cela peut parfois entraîner un coût de performance supérieur à celui des tableaux.

Le coût des performances correspond à la part du temps et de l'énergie d'un ordinateur que prend une opération donnée. De nos jours, les ordinateurs sont rapides, mais ils peuvent encore être surchargés par de gros jeux ou applications.

Une variable de type liste doit répondre aux exigences suivantes :

- Le mot-clé List, son type d'élément à l'intérieur des flèches gauche et droite, et un nom unique.
- Le nouveau mot-clé pour initialiser la liste en mémoire, avec le mot-clé List et le type d'élément entre les caractères fléchés.
- Une paire de parenthèses terminée par un point-virgule

Sous forme de plan, il se lit comme suit :

```
List<elementType>    name  = new  List<elementType>() ;
```

La longueur de la liste peut toujours être modifiée, il n'est donc pas nécessaire de préciser le nombre d'éléments.
qu'il contiendra lors de sa création.

Comme les tableaux, les listes peuvent être initialisées dans la déclaration de la variable en ajoutant les valeurs des éléments à l'intérieur d'une paire de crochets :

```
List<elementType> name = new List<elementType>() { value1, value2 } ;
```

Les éléments sont stockés dans l'ordre dans lequel ils sont ajoutés (au lieu de l'ordre séquentiel des valeurs elles-mêmes), sont indexés à zéro comme les tableaux et sont accessibles à l'aide de l'opérateur d'indice.

Commençons par créer notre propre liste pour tester les fonctionnalités de base offertes par cette classe.
Nous commencerons par un exercice d'échauffement en créant une liste des membres d'un groupe dans un jeu de rôle fictif. jeu :

1. Créez une nouvelle liste de type chaîne dans Start, appelée QuestPartyMembers, et initialisez-la avec les noms de trois personnages :

```
List<string> QuestPartyMembers = new List<string>() {
        "Grim le barbare",
        "Merlin le sage",
        "Sterling le chevalier".
} ;
```

2. Ajoutez un journal de débogage pour imprimer le nombre de membres du parti dans la liste à l'aide de la fonction Count méthode :

```
Debug.LogFormat("Membres du groupe : {0}", QuestPartyMembers.Count) ;
```

3. Enregistrez le fichier et jouez-le dans Unity.

Nous avons initialisé une nouvelle liste, appelée QuestPartyMembers, qui contient maintenant trois valeurs de type chaîne, et nous avons utilisé la méthode Count de la classe List pour imprimer le nombre d'éléments.

Remarquez que vous utilisez Count pour les listes, mais Length pour les tableaux.

Figure 4.9 : Capture d'écran de la sortie des éléments de la liste dans la console

Connaître le nombre d'éléments d'une liste est très utile ; cependant, dans la plupart des cas, cette information n'est pas suffisante. Nous voulons être en mesure de modifier nos listes en fonction des besoins, ce que nous allons voir par la suite.

Accès et modification des listes

Les éléments d'une liste sont accessibles et modifiables comme des tableaux avec un opérateur d'indice et un index, tant que l'index se trouve dans la plage de la classe Liste. Cependant, la classe Liste dispose d'une variété de méthodes qui étendent ses fonctionnalités, telles que l'ajout, l'insertion et la suppression d'éléments.

En restant dans la liste QuestPartyMembers, ajoutons un nouveau membre à l'équipe :

```
QuestPartyMembers.Add("Craven le Nécromancien") ;
```

La méthode Add() ajoute le nouvel élément à la fin de la liste, ce qui amène la méthode

Les membres de QuestPartyMembers sont au nombre de quatre et l'ordre des éléments est le suivant :

```
{
        "Grim le Barbare, Merlin le Sage,
        Sterling le Chevalier, Craven le
        Nécromancien
} ;
```

Pour ajouter un élément à un endroit spécifique d'une liste, nous pouvons passer l'index et la valeur souhaitée

à ajouter à la méthode Insert() :

```
QuestPartyMembers.Insert(1, "Tanis the Thief") ;
```

Lorsqu'un élément est inséré à un indice précédemment occupé, tous les éléments de la liste voient leur indice augmenter de 1. Dans notre exemple, "Tanis le voleur" est maintenant à l'indice 1, ce qui signifie que "Merlin le sage" est maintenant à l'indice 2 au lieu de 1, et ainsi de suite :

```
{
        "Grim le Barbare", "Tanis le
        Voleur",
```

```
        "Merlin le sage, Sterling le
        chevalier, Craven le
        nécromancien
};
```

La suppression d'un élément est tout aussi simple ; tout ce dont nous avons besoin, c'est de l'index ou de la valeur littérale, et de la fonction List fait le travail :

```
// Ces deux méthodes supprimeraient l'élément requis
QuestPartyMembers.RemoveAt(0) ;
QuestPartyMembers.Remove("Grim the Barbarian") ;
```

À l'issue de nos modifications, QuestPartyMembers contient désormais les éléments suivants, indexés à partir de 0 à 3 :

```
{
        "Tanis le voleur, Merlin le sage,
        Sterling le chevalier, Craven le
        nécromancien
};
```

Si vous lancez le jeu maintenant, vous verrez également que la longueur de la liste des participants est de 4 au lieu de 3 !

Il existe de nombreuses autres méthodes de la classe List qui permettent de vérifier les valeurs, de rechercher et de trier les éléments et de travailler avec des plages. Une liste complète des méthodes, avec leur description, est disponible ici : https://docs.microsoft.com/en-us/dotnet/api/system. collections.generic.list-1?view=netframework-4.7.2.

Si les listes sont idéales pour les éléments à valeur unique, il arrive que vous deviez stocker des informations ou des données contenant plus d'une valeur. C'est là que les dictionnaires entrent en jeu.

DICTIONNAIRES

Le type **Dictionnaire** s'éloigne des tableaux et des listes en stockant des paires de valeurs dans chaque élément, au lieu de valeurs uniques. Ces éléments sont appelés paires clé-valeur : la clé sert d'index, ou de valeur de consultation, pour la valeur correspondante. Contrairement aux tableaux et aux listes, les dictionnaires ne sont pas ordonnés. Toutefois, ils peuvent être triés et ordonnés dans diverses configurations après leur création.

La déclaration d'un dictionnaire est presque la même que celle d'une liste, mais avec un détail supplémentaire : les deux la clé et le type de valeur doivent être spécifiés à l'intérieur des symboles de flèche :

```
Dictionary<keyType, valueType> name = new Dictionary<keyType, valueType>() ;
```

Pour initialiser un dictionnaire avec des paires clé-valeur, procédez comme suit :

- Utiliser une paire de parenthèses à la fin de la déclaration
- Ajouter chaque élément à l'intérieur de sa paire de crochets, la clé et la valeur étant séparées par une virgule.
- Séparer les éléments par une virgule, sauf le dernier élément où la virgule est facultative

:

```
Dictionary<keyType, valueType> name = new Dictionary<keyType,
valueType>() {
        {key1, value1},
        {clé2, valeur2}
};
```

Une remarque importante à prendre en compte lors du choix des valeurs de clé est que chaque clé doit être unique et ne peut être modifiée. Si vous devez mettre à jour une clé, vous devez modifier sa valeur dans la déclaration de la variable ou supprimer toute la paire clé-valeur et en ajouter une autre dans le code, ce que nous verrons par la suite.

Tout comme les tableaux et les listes, les dictionnaires peuvent être initialisés sur une seule ligne sans problème avec Visual Studio. Toutefois, il est bon de prendre l'habitude d'écrire chaque paire clé-valeur sur sa propre ligne, comme dans l'exemple précédent, tant pour la lisibilité que pour la santé mentale.

Créons un dictionnaire pour stocker les objets qu'un personnage pourrait transporter :

1. Déclarez un dictionnaire dont le type de clé est string et le type de valeur int, appelé ItemInventory dans la méthode Start.
2. Initialisez-le avec new Dictionary<string, int>(), et ajoutez trois paires clé-valeur de votre choix. Assurez-vous que chaque élément se trouve dans sa paire de

```
Dictionary<string, int> ItemInventory = new Dictionary<string,
```

   crochets :

```
int>() {
    { "Potion", 5 },
    { "Antidote", 7 },
    { "Aspirine", 1 }
};
```

3. Ajoutez un journal de débogage pour imprimer la propriété ItemInventory.Count afin que nous puissions voir comment les articles sont stockés :

```
Debug.LogFormat("Items : {0}", ItemInventory.Count) ;
```

4. Enregistrez le fichier et jouez.

Ici, un nouveau dictionnaire, appelé ItemInventory, a été créé et initialisé avec trois paires clévaleur. Nous avons spécifié les clés sous forme de chaînes de caractères et les valeurs correspondantes sous forme d'entiers, et nous avons imprimé le nombre d'éléments que contient actuellement ItemInventory :

Figure 4.10 : Capture d'écran du nombre de dictionnaires dans la console

Comme pour les listes, nous devons pouvoir faire plus que simplement imprimer le nombre de paires clé-valeur dans un dictionnaire donné. Nous étudierons l'ajout, la suppression et la mise à jour de ces valeurs dans la section suivante.

Travailler avec des paires de dictionnaires

Les paires clé-valeur peuvent être ajoutées, supprimées et consultées dans les dictionnaires à l'aide des méthodes d'indice et de classe. Pour récupérer la valeur d'un élément, utilisez l'opérateur d'indice avec la clé de l'élément - dans l'exemple suivant, la valeur 5 sera attribuée

```
int numberOfPotions = ItemInventory["Potion"] ;
```

à numberOfPotions :

La valeur d'un élément peut être mise à jour en utilisant la même méthode - la valeur associée à "Potion" serait désormais de 10 :

```
Inventaire d'objets["Potion"] = 10 ;
```

Des éléments peuvent être ajoutés aux dictionnaires de deux manières : avec la méthode Add et avec l'opérateur subscript. La méthode Add prend en compte une clé et une valeur et crée un nouvel élément clé-valeur, pour autant que leurs types correspondent à la déclaration du

```
ItemInventory.Add("Throwing Knife", 3) ;
```

dictionnaire :

Si l'opérateur d'indice est utilisé pour attribuer une valeur à une clé qui n'existe pas dans un dictionnaire, le compilateur l'ajoutera automatiquement en tant que nouvelle paire clévaleur. Par exemple, si nous voulions ajouter un nouvel élément pour "Bandage", nous pourrions le faire avec le code suivant :

```
ItemInventory["Bandage"] = 5 ;
```

Cela soulève un point crucial concernant le référencement des paires clé-valeur : il est préférable d'être certain qu'un élément existe avant d'essayer d'y accéder, afin d'éviter d'ajouter par erreur de nouvelles paires clé-valeur. L'association de la méthode ContainsKey à une instruction if est la solution la plus simple, puisque ContainsKey renvoie une valeur booléenne en fonction de l'existence de la clé. Dans l'exemple suivant, nous nous assurons

```
if(ItemInventory.ContainsKey("Aspirin"))
{
    ItemInventory["Aspirine"] = 3 ;
}
```

que la clé "Aspirine" existe à l'aide d'une instruction if avant de modifier sa valeur :

Enfin, une paire clé-valeur peut être supprimée d'un dictionnaire à l'aide de la méthode Remove(), qui prend un paramètre clé :

```
ItemInventory.Remove("Antidote") ;
```

Comme les listes, les dictionnaires offrent une variété de méthodes et de fonctionnalités pour faciliter le développement, mais nous ne pouvons pas toutes les couvrir ici. Si vous êtes curieux, vous trouverez la documentation officielle à l'adresse suivante : https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2?view=netframework-4.7.2.

Les collections sont en sécurité dans notre boîte à outils, il est donc temps de faire un autre quiz pour s'assurer que vous êtes prêt à passer au prochain grand sujet : les déclarations d'itération.

POP QUIZ 2 - TOUT SUR LES COLLECTIONS

1. Qu'est-ce qu'un élément dans un tableau ou une liste ?

2. Quel est le numéro d'index du premier élément d'un tableau ou d'une liste ?

3. Un même tableau ou une même liste peuvent-ils stocker différents types de données ?
4. Comment ajouter des éléments à un tableau pour faire de la place à d'autres données ?

N'oubliez pas de comparer vos réponses aux miennes dans l'annexe *Fou Quiz "nswcus"* pour voir où vous en êtes !

Les collections étant des groupes ou des listes d'éléments, elles doivent être accessibles de manière efficace. Heureusement, C dispose de#plusieurs déclarations d'itération, dont nous parlerons dans la section suivante.

DECLARATIONS D'ITERATION

Nous avons accédé à des éléments individuels de la collection grâce à l'opérateur d'indice et aux méthodes de type collection, mais que faire lorsque nous devons parcourir l'ensemble de la collection élément par élément ? En programmation, cela s'appelle l'**itération**, et C propose plusieurs types# d'instructions qui nous permettent de parcourir en boucle (ou d'itérer sur, si vous voulez être technique) les éléments d'une collection. Les instructions d'itération sont semblables à des méthodes, en ce sens qu'elles stockent un bloc de code à exécuter. Toutefois, contrairement aux méthodes, elles peuvent exécuter leurs blocs de code de manière répétée tant que leurs conditions sont remplies. **pour les boucles**

La boucle for est le plus souvent utilisée lorsqu'un bloc de code doit être exécuté un certain nombre de fois avant que le programme ne se poursuive. L'instruction elle-même prend en compte trois expressions, chacune ayant une fonction spécifique à exécuter avant que la boucle ne s'exécute. Comme les boucles for gardent la trace de l'itération en cours, elles conviennent mieux aux tableaux et aux listes.

Jetez un coup d'œil au modèle de déclaration en boucle suivant :

```
pour (initialisateur ; condition ; itérateur)
{ bloc de code ;
}
```

Voyons ce qu'il en est :

1. Le mot-clé for commence la déclaration, suivi d'une paire de parenthèses.
2. À l'intérieur des parenthèses se trouvent les gardiens : l'initialisateur, la condition et l'itérateur expressions.
3. La boucle commence par l'expression d'initialisation, qui est une variable locale créée pour garder une trace du nombre d'exécutions de la boucle - elle est généralement fixée à 0 car les types de collection sont indexés à zéro.
4. Ensuite, l'expression de la condition est vérifiée et, si elle est vraie, elle passe à l'itérateur.
5. L'expression de l'itérateur est utilisée pour augmenter ou diminuer (**incrémenter** ou **décrémenter**) l'initialisateur, ce qui signifie que la prochaine fois que la boucle évaluera sa condition, l'initialisateur sera différent.

L'augmentation et la diminution d'une valeur par 1 sont appelées respectivement **incrémentation** et **décrémentation** (-- diminue une valeur de 1, et ++ l'augmente de 1).

Tout cela semble beaucoup, alors voyons un exemple pratique avec la liste QuestPartyMembers que nous avons créée plus tôt :

```
List<string> QuestPartyMembers = new List<string>()
{ "Grim le Barbare", "Merlin le Sage", "Sterling le Chevalier"} ; int listLength = QuestPartyMembers.Count ;

for (int i = 0 ; i < listLength ; i++)
{
      Debug.LogFormat("Index : {0} - {1}", i, QuestPartyMembers[i]) ; }
```

Reprenons la boucle et voyons comment elle fonctionne :

1. Tout d'abord, l'initialisateur dans la boucle for est défini comme une variable int locale nommée i avec une valeur initiale de 0.
2. Deuxièmement, nous stockons la liste de la liste dans une variable afin que la boucle n'ait pas besoin de vérifier la longueur à chaque fois, ce qui est une bonne pratique pour les performances.
3. Pour s'assurer de ne jamais obtenir d'exception hors plage, la boucle for s'assure que la boucle ne s'exécute une nouvelle fois que si i est inférieur au nombre d'éléments de QuestPartyMembers :
    - Avec les tableaux, nous utilisons la propriété Length pour déterminer le nombre d'éléments qu'ils contiennent

- Avec les listes, nous utilisons la propriété Count

4. Enfin, i est augmenté de 1 à chaque fois que la boucle est exécutée avec l'opérateur ++.
5. Dans la boucle for, nous venons d'imprimer l'index et l'élément de la liste à cet index en utilisant i.
6. Remarquez que i est en phase avec l'indice des éléments de la collection, puisqu'ils commencent tous deux à 0 :

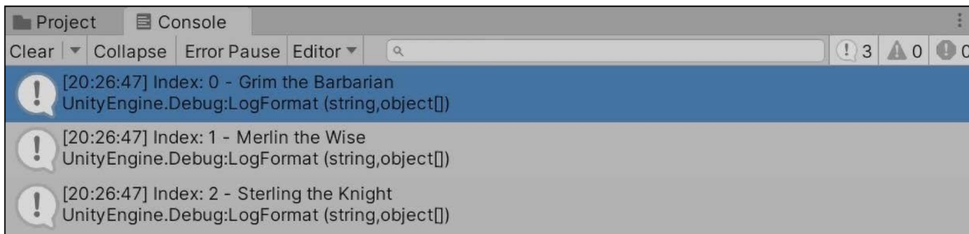


Figure 4.11 : Capture d'écran des valeurs de la liste imprimées avec une boucle for

Traditionnellement, la lettre i est utilisée comme nom de variable d'initialisation. Si vous avez des variables pour, les noms de variables utilisés doivent être les lettres j, k, l, et ainsi de suite. Essayons nos nouvelles instructions d'itération sur l'une de nos collections existantes.

Pendant que nous parcourons en boucle les membres de QuestPartyMembers, voyons si nous pouvons identifier quand un certain élément est itéré et ajouter un journal de débogage spécial juste pour ce cas :

1. Déplacer la liste QuestPartyMembers et la boucle for dans une fonction publique appelée FindPartyMember et l'appeler dans Start.
2. Ajoutez une instruction if sous le journal de débogage dans la boucle for pour vérifier si l'état actuel de l'application La liste des membres de questPartyMember correspond à "Merlin le sage" :

```
if(QuestPartyMembers[i] == "Merlin le Sage")
{
    Debug.Log("Heureux que tu sois là Merlin !") ;
}
```

3. Si c'est le cas, ajoutez un journal de débogage de votre choix, vérifiez que votre code correspond à la capture d'écran ci-dessous, et appuyez sur "play" :

```
void Start()
{
    FindPartyMember() ;
}

public void FindPartyMember()
{
    List<string> QuestPartyMembers = new List<string>() {
        "Grim le Barbare", "Merlin le
        Sage",
```

```
            "Sterling le chevalier"
        } ;

        int listLength = QuestPartyMembers.Count ;
        QuestPartyMembers.Add("Craven le Nécromancien") ;
        QuestPartyMembers.Insert(1, "Tanis le Voleur") ;
        QuestPartyMembers.RemoveAt(0) ;
        //QuestPartyMembers.Remove("Grim the Barbarian") ; Debug.LogFormat("Membres

        du parti : {0}", listLength) ;

        for(int i = 0 ; i < listLength ; i++)
        {
            Debug.LogFormat("Index : {0} - {1}",
    i, QuestPartyMembers[i]) ;

            if(QuestPartyMembers[i] == "Merlin le Sage")
            {
                Debug.Log("Heureux que tu sois là Merlin !") ; }
        }
    }
```

La sortie de la console devrait être presque la même, sauf qu'il y a maintenant un journal de débogage supplémentaire - un journal qui n'a été imprimé qu'une seule fois lorsque c'était au tour de Merlin de parcourir la boucle. Plus précisément, lorsque i était égal à 1 dans la deuxième boucle, l'instruction if s'est déclenchée et deux journaux ont été imprimés au lieu d'un seul :

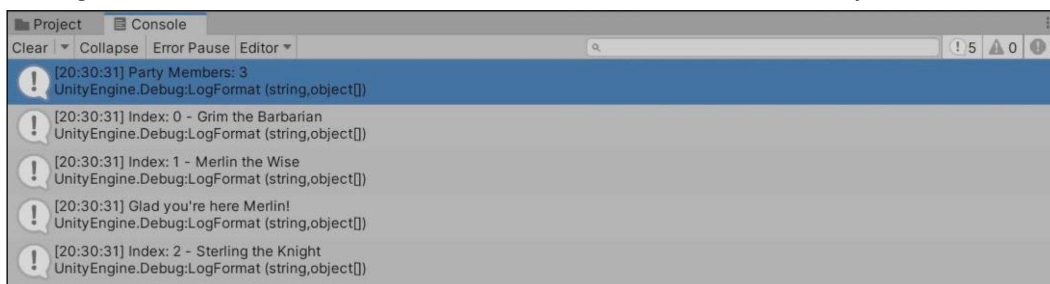


Figure 4.12 : Capture d'écran de la boucle for imprimant les valeurs de la liste et les instructions if correspondantes

L'utilisation d'une boucle for standard peut être très utile dans la bonne situation, mais il y a rarement une seule façon de faire les choses en programmation, et c'est là que l'instruction foreach entre en jeu.

BOUCLES FOREACH

Les boucles foreach prennent chaque élément d'une collection et le stockent dans une variable locale, ce qui le rend accessible à l'intérieur de l'instruction. Les boucles foreach peuvent être utilisées avec des tableaux et des listes, mais elles sont particulièrement utiles avec les dictionnaires, étant donné que les dictionnaires sont des paires clé-valeur au lieu d'index numériques.

Sous forme de schéma, une boucle foreach se présente comme suit :

```
foreach(elementType localName in collectionVariable)
{ bloc de code ;
}
```

Reprenons l'exemple de la liste QuestPartyMembers et faisons l'appel pour chacun de ses éléments :

```
List<string> QuestPartyMembers = new List<string>()
{ "Grim le barbare", "Merlin le sage", "Sterling le chevalier"} ;

foreach(string partyMember in QuestPartyMembers)
{
        Debug.LogFormat("{0} - Here !", partyMember) ;
}
```

Vous pouvez également utiliser le mot-clévar pour déterminer automatiquement le type de collection que vous parcourez, comme suit :

```
foreach (var partyMember in QuestPartyMembers)
{
        Debug.LogFormat("{0} - Here !", partyMember") ;
}
```

Nous pouvons décomposer cela comme suit :

- Le type d'élément est déclaré comme une chaîne de caractères, qui correspond aux valeurs de QuestPartyMembers
- Une variable locale, appelée partyMember, est créée pour contenir chaque élément au fur et à mesure que la boucle se répète.
- Le mot-clé in, suivi de la collection que nous voulons parcourir en boucle, dans ce cas,

QuestPartyMembers , termine le tout :

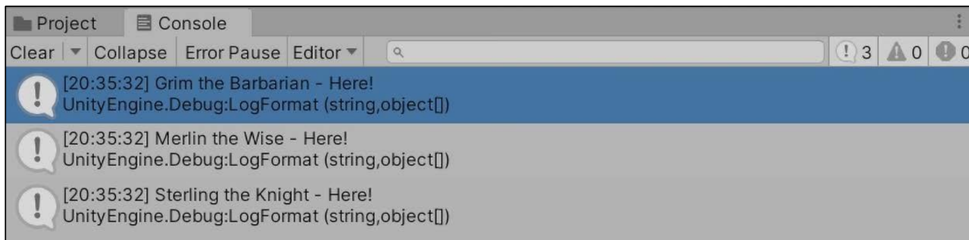


Figure 4.13 : Capture d'écran d'une boucle foreach imprimant les valeurs d'une liste

C'est beaucoup plus simple que la boucle for. Cependant, lorsqu'il s'agit de dictionnaires, il existe des différences importantes que nous devons mentionner, notamment la façon de traiter les paires clé-valeur en tant que variables locales.

Boucle sur les paires clé-valeur

Pour capturer une paire clé-valeur dans une variable locale, nous devons utiliser le bien nommé type KeyValuePair, en assignant les types clé et valeur pour qu'ils correspondent aux types correspondants du dictionnaire. Comme KeyValuePair est son type, il se comporte comme n'importe quel autre type d'élément, en tant que variable locale.

Par exemple, parcourons en boucle le dictionnaire ItemInventory que nous avons créé plus tôt dans la section *Dicíionnuics* et déboguer chaque valeur-clé comme une description d'article de magasin :

```
Dictionary<string, int> ItemInventory = new Dictionary<string, int>() {
        {"Potion", 5},
        {"Antidote", 7},
        { "Aspirine", 1}
};

foreach(KeyValuePair<string, int> kvp in ItemInventory)
{
        Debug.LogFormat("Item : {0} - {1}g", kvp.Key, kvp.Value) ; }
```

Nous avons spécifié une variable locale KeyValuePair, appelée kvp, ce qui est une convention de dénomination courante en programmation, comme l'appel à l'initialisateur i de la boucle for, et nous avons défini les types de clé et de valeur à string et int pour correspondre à ItemInventory.

Pour accéder à la clé et à la valeur de la variable locale kvp, nous utilisons les propriétés KeyValuePair de Key et la valeur, respectivement.

Dans cet exemple, les clés sont des chaînes et les valeurs sont des entiers, que nous pouvons imprimer comme le nom et le prix de l'article :

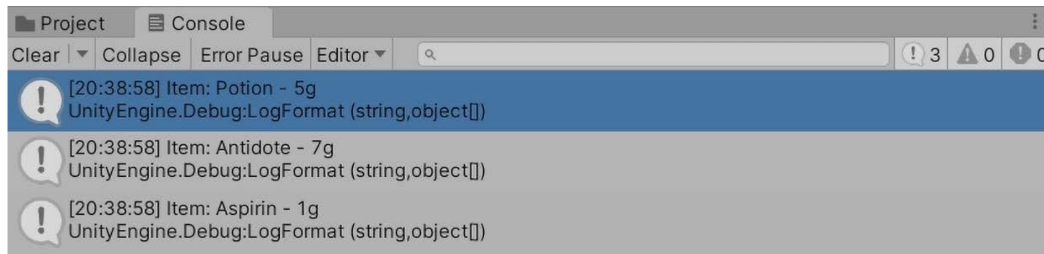


Figure 4.14 : Capture d'écran d'une boucle foreach imprimant les paires clé-valeur d'un dictionnaire

Si vous vous sentez particulièrement aventureux, essayez le défi facultatif suivant pour mettre en pratique ce que vous venez d'apprendre.

L'EPREUVE DU HEROS - TROUVER DES ARTICLES ABORDABLES

À l'aide du script précédent, créez une variable pour stocker la quantité d'or que possède votre personnage fictif et voyez si vous pouvez ajouter une instruction if à l'intérieur de la boucle foreach pour vérifier les objets que vous pouvez vous offrir.

Conseil : utilisez kvp.Value pour comparer les prix avec ce que vous avez dans votre portefeuille.

BOUCLES "WHILE

Les boucles while sont similaires aux instructions if en ce sens qu'elles s'exécutent tant qu'une seule expression ou condition est vraie.

Les comparaisons de valeurs et les variables booléennes peuvent être utilisées comme conditions de type "while", et elles peuvent être modifié par l'opérateur NOT.

La syntaxe de la boucle while dit : *Whilc m ; conKiíion is íuuc, kccu uunning m ; coKc block inKcfiniícl;* :

```
Initialisateur while (condition)
{
```

```
        bloc de code ;
        itérateur ;
}
```

Avec les boucles while, il est courant de déclarer une variable d'initialisation, comme dans une boucle for, et de l'incrémenter ou de la décrémenter manuellement à la fin du bloc de code de la boucle. Nous procédons ainsi pour éviter une boucle infinie, dont nous parlerons à la fin du chapitre. En fonction de votre situation, l'initialisateur fait généralement partie de la condition de la boucle.

Les boucles while sont très utiles lorsque l'on code en C#, mais elles ne sont pas considérées comme une bonne pratique dans Unity parce qu'elles peuvent avoir un impact négatif sur les performances et qu'elles doivent systématiquement être gérées manuellement.

Prenons un cas d'utilisation courant dans lequel nous devons exécuter du code tant que le joueur est en vie, puis déboguer lorsque ce n'est plus le cas :

1. Créez une variable publique appelée PlayerLives de type int et donnez-lui la valeur 3 :

```
public int PlayerLives = 3 ;
```

2. Créer une nouvelle fonction publique appelée HealthStatus :

```
public void HealthStatus()
{
}
```

3. Déclarez une boucle while dont la condition est de vérifier si PlayerLives est supérieur à 0 (c'est-à-dire que le joueur est toujours en vie) :

```
while(PlayerLives > 0)
{
}
```

4. À l'intérieur de la boucle while, déboguez quelque chose pour nous faire savoir que le personnage est toujours en train de frapper, puis décrémentez PlayerLives de 1 à l'aide de l'opérateur -- :

```
Debug.Log("Still alive !")
; PlayerLives-- ;
```

5. Ajouter un journal de débogage après les parenthèses de la boucle while afin d'imprimer quelque chose lorsque nos vies s'épuisent :

```
Debug.Log("Joueur KO...") ;
```

6. Appeler la méthode HealthStatus dans Start :

```
void Start()
{
      HealthStatus() ;
}
```

Votre code devrait ressembler à ce qui suit :

```
public int PlayerLives = 3 ;

void Start()
{
      HealthStatus() ;
}

public void HealthStatus()
{ while(PlayerLives > 0)
      {
            Debug.Log("Still alive !")
            ; PlayerLives-- ;
      }

      Debug.Log("Joueur KO...") ;
}
```

Avec PlayerLives commençant à 3, la boucle while s'exécutera trois fois. Au cours de chaque boucle, le journal de débogage, "Still alive !", se déclenche, et une vie est soustraite de PlayerLives.

Lorsque la boucle while s'exécute une quatrième fois, notre condition échoue car PlayerLives est égal à 0, donc le bloc de code est ignoré et le journal de débogage final s'imprime :

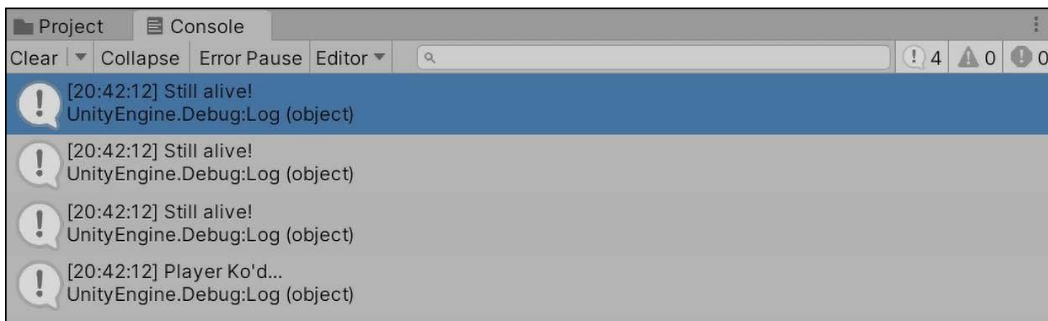


Figure 4.15 : Capture d'écran de la sortie de la boucle while dans la console

Si vous ne voyez pas plusieurs journaux de débogage "Still alive !", assurez-vous que le bouton "**Collapse**" de la fenêtre

La barre d'outils de **la console n**'est pas sélectionnée.

La question est maintenant de savoir ce qui se passe si une boucle ne s'arrête jamais de s'exécuter. Nous aborderons cette question dans la section suivante.

JUSQU'A L'INFINI ET AU-DELA

Avant de terminer ce chapitre, nous devons comprendre un concept extrêmement important en ce qui concerne les instructions d'itération : les *loos infinis*. C'est exactement ce à quoi elles ressemblent : lorsque les conditions d'une boucle rendent impossible l'arrêt de son exécution et la poursuite du programme. Les boucles infinies se produisent généralement dans les boucles for et while lorsque l'itérateur n'est pas augmenté ou diminué ; si la ligne de code PlayerLives était omise de l'exemple de la boucle while, Unity se figerait et/ou se planterait, reconnaissant que PlayerLives serait toujours 3 et exécutant la boucle à l'infini.

Les itérateurs ne sont pas les seuls coupables dont il faut se méfier ; le fait de définir des conditions dans une boucle for qui n'échoueront jamais, ou qui seront évaluées à faux, peut également provoquer des boucles infinies. Dans l'exemple des membres du parti, tiré de la section *Loouing íhuough kc;Lvnluc unius*, si nous avions fixé la condition de la boucle for à i < 0 au lieu de i

< QuestPartyMembers.Count, i serait toujours inférieur à 0, tournant en boucle jusqu'à ce que Unity se plante.

RESUME

Alors que nous clôturons ce chapitre, nous devrions réfléchir à tout ce que nous avons accompli et à ce que nous pouvons construire avec ces nouvelles connaissances. Nous savons comment utiliser des vérifications simples de type "if-else" et des instructions plus complexes

de type "switch", qui permettent de prendre des décisions dans le code. Nous pouvons créer des variables qui contiennent des collections de valeurs avec des tableaux et des listes ou des paires clé-valeur avec des dictionnaires.

Cela permet de stocker efficacement des données complexes et groupées. Nous pouvons même choisir la bonne instruction de bouclage pour chaque type de collection, tout en évitant soigneusement les plantages dus aux boucles infinies.

Si vous vous sentez surchargé, c'est tout à fait normal - la pensée logique et séquentielle fait partie de l'exercice de votre cerveau programmateur.

Le prochain chapitre complétera les bases de la programmation sur C en abordant les classes, les structures et la **programmation orientée objet** (**POO**). Nous mettrons tout ce que nous avons appris jusqu'à présent dans ces sujets, préparant ainsi notre première véritable plongée dans la compréhension et le contrôle des objets dans le moteur Unity.

Rejoignez-nous sur discord !

Lisez ce livre en compagnie d'autres utilisateurs, d'experts en développement de jeux Unity et de l'auteur luimême.

Posez des questions, apportez des solutions aux autres lecteurs, discutez avec l'auteur via des sessions "Ask Me Anything" et bien plus encore. Ask Me Anything et bien plus encore.

Scannez le code QR ou visitez le lien pour rejoindre la communauté.

https://packt.link/csharpwithunity

5

Travailler avec des classes, des structures, et OOP

Pour des raisons évidentes, l'objectif de ce livre n'est pas de vous donner un mal de crâne à cause d'une surcharge d'informations. Cependant, les sujets suivants vous feront sortir de la cabine du débutant et vous amèneront à l'air libre de la **programmation orientée objet** (**POO)**. Jusqu'à présent, nous nous sommes appuyés exclusivement sur des types de variables prédéfinis qui font partie du langage C : des chaînes de caractères, des listes et des dictionnaires sous le capot qui sont des classes, ce qui explique pourquoi nous pouvons les créer et utiliser leurs propriétés grâce à la notation par points. Cependant, le fait de s'appuyer sur des types intégrés présente une faiblesse flagrante : l'impossibilité de s'écarter des plans déjà établis par C.

La création de vos classes vous donne la liberté de définir et de configurer des modèles de votre conception, en capturant des informations et en menant des actions spécifiques à votre jeu ou à votre application. Par essence, les classes personnalisées et la POO sont les clés du royaume de la programmation ; sans elles, les programmes uniques seront rares.

Dans ce chapitre, vous allez acquérir une expérience pratique de la création de classes à partir de zéro et discuter du fonctionnement interne des variables de classe, des constructeurs et des méthodes. Vous découvrirez également les différences

entre les objets de type référence et de type valeur, et comment ces concepts peuvent être appliqués dans Unity. Les sujets suivants seront abordés plus en détail au fur et à mesure que vous avancerez dans le cours :

- Présentation de la POO
- Définition des classes
- Déclarer des structures
- Comprendre les types de référence et de valeur

- Intégrer l'esprit orienté objet
- Application de la POO dans Unity

PRESENTATION DE LA POO

La programmation orientée objet est le principal paradigme de programmation que vous utiliserez lorsque vous coderez en C#. Si les classes et les structures sont les plans de nos programmes, la POO est l'architecture qui tient le tout ensemble. Lorsque nous parlons de la POO comme d'un paradigme de programmation, nous disons qu'elle a des principes spécifiques sur la façon dont le programme global doit fonctionner et communiquer.

Essentiellement, la POO se concentre sur les objets plutôt que sur la logique séquentielle pure - les données qu'ils contiennent, la façon dont ils conduisent l'action et, surtout, la façon dont ils communiquent entre eux.

DEFINITION DES CLASSES

Dans *Chnuícu 2*, *Thc DuilKing Dlocks ofi Fuogunmming*, nous avons brièvement parlé de la façon dont les classes sont des plans pour les objets (dans ce cas, des objets dans le code, et non des GameObjects dans Unity) et mentionné qu'elles peuvent être traitées comme des types de variables personnalisées. Nous avons également appris que le script LearningCurve est une classe, mais une classe spéciale qu'Unity peut attacher aux objets de la scène. La principale chose à retenir avec les classes est qu'elles sont de type **référence, c'est-à-dire** que lorsqu'elles sont assignées ou passées à une autre variable, c'est l'objet original qui est référencé, et non une nouvelle copie. Nous aborderons ce point après avoir discuté des structures dans la section *Dcclnuing síuucís.* Cependant, avant tout cela, nous devons comprendre les bases de la création de classes.

Pour l'instant, nous allons mettre de côté le fonctionnement des classes et des scripts dans Unity et nous concentrer sur la façon dont ils sont créés et utilisés en C#. Les classes sont

```
accessModifier classe UniqueName
{
    Variables
    Constructeurs
    Méthodes
}
```

Toutes les variables ou méthodes déclarées à l'intérieur d'une classe appartiennent à cette classe et sont accessibles par le biais de son nom unique.

créées à l'aide du mot-clé class, comme suit :

Pour que les exemples soient aussi cohérents que possible tout au long de ce chapitre, nous créerons et modifierons une classe de personnage simple, comme celle d'un jeu typique. Nous nous éloignerons également des captures d'écran de code pour vous habituer à lire et à interpréter le code tel que vous le verriez "dans la nature". Cependant, la première chose dont nous avons besoin est une classe personnalisée, alors créons-en une.

Nous aurons besoin d'une classe pour nous entraîner avant de comprendre leur fonctionnement interne, alors créons un nouveau script C# et partons de zéro :

1. Cliquez avec le bouton droit de la souris sur le dossier Scripts que vous avez créé dans *Chnuícu 1*, *Gcíiing ío Know Youu Enviuonmcní*, et choisissez **Créer** > **C# Script**.
2. Nommez le script Character, ouvrez-le dans Visual Studio et supprimez tout le code généré.

   à l'exception des trois premières lignes qui commencent par le mot-clé using.
3. Déclarez une classe publique appelée Character suivie d'un ensemble d'accolades {}, puis enregistrez le fichier. Le code de votre classe doit correspondre exactement au code suivant :

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

public class Character
{
}
```

Nous avons supprimé le code généré car nous n'aurons pas besoin d'attacher ce script à un objet de jeu Unity.

Le personnage est désormais enregistré en tant que blueprint de classe publique. Cela signifie que n'importe quelle classe du projet peut l'utiliser pour créer des personnages. Cependant, il ne s'agit que d'instructions - la création d'un personnage nécessite une étape supplémentaire. Cette étape de création est appelée **instanciation** et fait l'objet de la section suivante.

INSTANCIER DES OBJETS DE CLASSE

L'instanciation est l'acte de créer un objet à partir d'un ensemble spécifique d'instructions, que l'on appelle une **instance**. Si les classes sont des plans, les instances sont les maisons construites à partir de leurs instructions ; toute nouvelle instance de Character est son objet, tout comme deux maisons construites à partir des mêmes instructions restent deux structures physiques différentes. Ce qui arrive à l'une n'a aucune répercussion sur l'autre.

Dans *Chnuícu 4*, *Coníuol Flow nnK Collccíion T;ucs*, nous avons créé des listes et des dictionnaires, qui sont des classes par défaut fournies avec C#, en utilisant leurs types et le mot-clé new. Nous pouvons faire la même chose pour les classes personnalisées telles que Character, ce que nous allons faire maintenant.

Nous avons déclaré la classe Character comme publique, ce qui signifie que vous pouvez créer une instance de Character dans n'importe quelle autre classe. Puisque LearningCurve fonctionne déjà, déclarons un nouveau personnage dans la méthode Start().

Ouvrez LearningCurve et déclarez une nouvelle variable de type Character, appelée hero, dans la fonction Start() méthode :

```
Personnage hero = nouveau personnage() ;
```

Voyons cela étape par étape :

1. Le type de variable est spécifié comme Character, ce qui signifie que la variable est une instance de cette classe.
2. La variable s'appelle hero et est créée à l'aide du mot-clé new, suivi du nom de la classe Character et de deux parenthèses (). C'est ici que l'instance réelle est créée dans la mémoire du programme, même si la classe est vide pour l'instant.

3. Nous pouvons utiliser la variable hero comme n'importe quel autre objet avec lequel nous avons travaillé jusqu'à présent. Lorsque la classe Character aura ses propres variables et méthodes, nous pourrons y accéder à partir de hero en utilisant la notation point.

Vous auriez tout aussi bien pu utiliser une déclaration inférée lors de la création de la variable hero, comme suit :

```
var hero = new Character() ;
```

Notre classe de personnage ne peut pas faire grand-chose sans champs de classe. Vous ajouterez des champs de classe et d'autres éléments dans les prochaines sections.

AJOUTER DES CHAMPS DE CLASSE

L'ajout de variables, ou de champs, à une classe personnalisée n'est pas différent de ce que nous avons déjà fait avec LearningCurve. Les mêmes concepts s'appliquent, y compris les modificateurs d'accès, la portée des variables et l'attribution de valeurs. Cependant, toutes les variables appartenant à une classe sont créées avec l'instance de la classe, ce qui signifie que si aucune valeur ne leur est attribuée, elles prendront par défaut la valeur zéro ou null. En général, le choix des valeurs initiales dépend des informations qu'elles stockent :

- Si une variable doit avoir la même valeur initiale à chaque fois qu'une instance de classe est créée, la définition d'une valeur initiale est une bonne idée. Cela peut

s'avérer utile pour les points d'expérience ou le score de départ, par exemple.

- Si une variable doit être personnalisée dans chaque instance de classe, comme CharacterName, laissez sa valeur non assignée et utilisez un constructeur de classe (un sujet que nous aborderons dans la section *Using consíuucíous*).

Chaque classe de personnage aura besoin de quelques champs de base ; c'est à vous de les ajouter.

Incorporons deux variables pour contenir le nom du personnage et le nombre de points d'expérience de départ :

1. Ajoutez deux variables publiques à l'intérieur des accolades de la classe Character :

   une variable de type chaîne pour le nom et une variable de type entier pour les points d'expérience.

2. Laissez la valeur du nom vide, mais fixez les points d'expérience à 0 afin que chaque personnage commence au bas de l'échelle :

```
public class Character
{ public string name ; public int exp = 0 ;
}
```

3. Ajouter un journal de débogage dans LearningCurve juste après l'initialisation de l'instance de personnage. Utilisez-le pour imprimer le nom du nouveau personnage et les variables exp en utilisant la notation par points :

```
Personnage hero = nouveau personnage() ;
Debug.LogFormat("Hero : {0} - {1} EXP", hero.name, hero.exp) ;
```

4. Lorsque le héros est initialisé, le nom se voit attribuer une valeur nulle qui apparaît comme un espace vide dans le journal de débogage, tandis que l'exp s'imprime comme 0. Remarquez que nous n'avons pas eu à attacher le script Character à des GameObjects dans la scène ; nous les avons simplement référencés dans LearningCurve et Unity s'est chargé du reste. La console va maintenant déboguer les informations de notre personnage, qui est référencé comme suit :

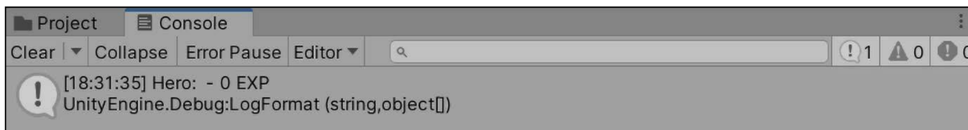


Figure 5.1 : Capture d'écran des propriétés d'une classe personnalisée imprimées dans la console

Pour uuoviKc n comulcíc vicw ofi íhc Uníí ; cKiíou, nll ouru scuccnshoís nuc ínkcn in fiullLscuccn moKc. Pour les images en couleur de tous les livres, utilisez le lien suivant : https://packt.link/7yy5V.

À ce stade, notre classe fonctionne, mais elle n'est pas très pratique avec ces valeurs vides. Vous aurez besoin de pour y remédier avec ce que l'on appelle un constructeur de classe.

UTILISATION DES CONSTRUCTEURS

Les constructeurs de classe sont des méthodes spéciales qui se déclenchent automatiquement lors de la création d'une instance de classe, de la même manière que la méthode Start s'exécute dans LearningCurve. Les constructeurs construisent la classe conformément à son plan :

- Si aucun constructeur n'est spécifié, C# en génère un par défaut. Le constructeur par défaut donne à toutes les variables la valeur de leur type par défaut : les valeurs numériques sont fixées à 0, les booléens à false et les types de référence (classes) à null.
- Les constructeurs personnalisés peuvent être définis avec des paramètres, comme n'importe quelle autre méthode, et sont utilisé pour définir les valeurs des variables de classe lors de l'initialisation.
- Une classe peut avoir plusieurs constructeurs.

Les constructeurs s'écrivent comme des méthodes ordinaires, à quelques différences près : par exemple, ils doivent être publics, n'ont pas de type de retour et le nom de la méthode est toujours le nom de la classe. À titre d'exemple, ajoutons un constructeur de base sans paramètre à la classe Character et attribuons au champ name une valeur autre que null.

Ajoutez ce nouveau code directement sous les variables de la classe, comme suit :

```
public class Character
{ public string name ; public int
      exp = 0 ;

      public Character()
      { nom = "Non attribué" ;
      }
}
```

Lancez le projet dans Unity et vous verrez l'instance du héros utiliser ce nouveau constructeur. Le journal de débogage indiquera que le nom du héros **n'est pas assigné** au lieu d'une valeur nulle :

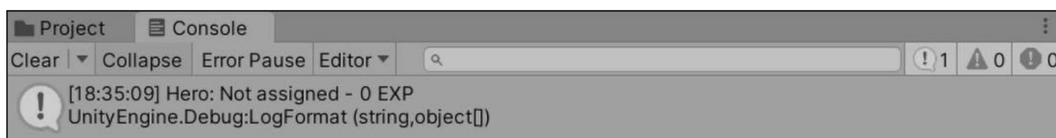


Figure 5.2 : Capture d'écran des variables de classe personnalisées non assignées imprimées sur la console

C'est un bon progrès, mais nous avons besoin que le constructeur de la classe soit plus flexible. Cela signifie que nous devons doit pouvoir passer des valeurs afin qu'elles puissent être utilisées comme valeurs de départ, ce que vous ferez par la suite.

La classe Character commence à se comporter comme un véritable objet, mais nous pouvons encore l'améliorer en ajoutant un second constructeur qui prend un nom lors de l'initialisation et le place dans le champ name :

1. Ajoutez un autre constructeur à Character qui prend un paramètre de type chaîne de caractères, appelé name.
   Le fait d'avoir plusieurs constructeurs dans une même classe s'appelle la **surcharge des constructeurs**.
2. Attribuer le paramètre à la variable nom de la classe à l'aide du mot clé this :

```
public class Character
{ public string name ; public int exp =
            0 ;

            public Character()
            { nom = "Non attribué" ;
            }

            public Character(string name)
            { this.name = name ;
            }
}
```

3. Pour des raisons de commodité, les constructeurs ont souvent des paramètres qui partagent un nom avec une variable de classe. Dans ce cas, il convient d'utiliser le mot-clé this pour spécifier quelle variable appartient à la classe. Dans l'exemple cidessus, this.name fait référence à la variable name de la classe, tandis que name est le paramètre ; sans le mot-clé this, le compilateur émettra un avertissement car il ne sera pas en mesure de les différencier. Pour plus de clarté, vous auriez également pu utiliser le mot-clé this dans le constructeur par défaut où nous avons défini la propriété name à Not assigned.
4. Créer une nouvelle instance de personnage dans LearningCurve, appelée héroïne.

```
Personnage heroine = new Character("Agatha") ;
Debug.LogFormat("Hero : {0} - {1} EXP", heroine.name, héroïne.exp) ;
```


Utilisez le constructeur personnalisé pour passer un nom lors de l'initialisation et imprimez les détails dans la console :

5. Lorsqu'une classe possède plusieurs constructeurs ou qu'une méthode possède plusieurs variantes, Visual Studio affiche un ensemble de flèches dans la fenêtre contextuelle d'autocomplétion, que l'on peut faire défiler à l'aide des touches fléchées :

Figure 5.3 : Capture d'écran des constructeurs de méthodes multiples dans Visual Studio

6. Nous pouvons maintenant choisir entre le constructeur de base et le constructeur personnalisé lorsque nous initialisons une nouvelle classe de personnage. La classe de personnage elle-même est désormais beaucoup plus flexible lorsqu'il s'agit de configurer différentes instances pour différentes situations :

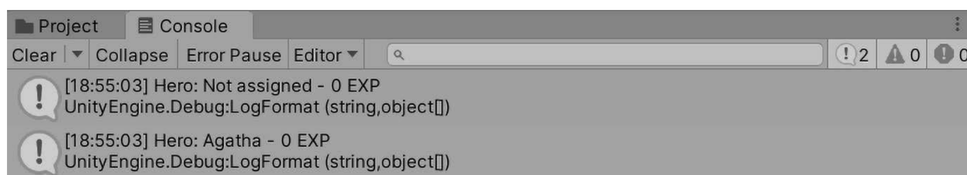


Figure 5.4 : Capture d'écran de plusieurs instances de classes personnalisées imprimées dans la console

C'est maintenant que le vrai travail commence ; notre classe a besoin de méthodes capables de faire des choses utiles en plus d'agir en tant qu'installation de stockage pour les variables. Votre prochaine tâche consiste à mettre cela en pratique.

DECLARER LES METHODES DE LA CLASSE

Ajouter des méthodes à des classes personnalisées n'est pas différent de les ajouter à LearningCurve. Cependant, c'est une bonne occasion de parler d'un élément fondamental de la bonne programmation - ne **pas se répéter** (**DRY**). DRY est une référence pour tout code bien écrit. Essentiellement, si vous vous retrouvez à écrire la même ligne, ou les mêmes lignes, encore et encore, il est temps de repenser et de réorganiser. Cela prend généralement la forme d'une nouvelle méthode pour contenir le code répété, ce qui facilite la modification et l'appel de cette fonctionnalité ailleurs dans le script en cours ou même à partir d'autres scripts.

En termes de programmation, on parle d'**abstraction d'**une méthode ou d'une fonctionnalité.

Nous avons déjà pas mal de code répété, alors jetons un coup d'œil et voyons où nous pouvons augmenter le nombre d'heures de travail. la lisibilité et l'efficacité de nos textes.

Nos journaux de débogage répétés sont l'occasion parfaite d'extraire du code directement dans le fichier Classe de personnage :

1. Ajoutez une nouvelle méthode publique avec un type de retour void, appelée PrintStatsInfo, à la méthode Classe de personnage.
2. Copiez et collez le journal de débogage de LearningCurve dans le corps de la méthode.
3. Remplacez les variables par name et exp, puisqu'elles peuvent maintenant être référencées directement dans la classe :

```
public void PrintStatsInfo()
{
        Debug.LogFormat("Hero : {0} - {1} EXP", this.name, this.exp) ; }
```

4. Remplacez le journal de débogage des caractères que nous avons précédemment ajouté à LearningCurve par des appels de méthode à PrintStatsInfo, et cliquez sur **Play**

:

```
Character hero = new Character() ; hero.PrintStatsInfo() ;

Personnage héroïne = nouveau
personnage("Agatha") ; héroïne.PrintStatsInfo() ;
```

5. Maintenant que la classe Character a une méthode, n'importe quelle instance peut y accéder librement en utilisant la notation point. Puisque le héros et l'héroïne sont tous deux des objets distincts, PrintStatsInfo affiche leur nom respectif et leur valeur exp sur la console.

Ce comportement est préférable à l'utilisation des journaux de débogage directement dans LearningCurve. C'est toujours une bonne idée de regrouper les fonctionnalités dans une classe et d'agir par le biais de méthodes. Cela rend le code plus lisible - comme nos objets Personnage qui donnent une commande lors de l'impression des journaux de débogage, au lieu de répéter le code.

L'ensemble de la classe Character devrait ressembler au code suivant :

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

public class Character
{ public string name ; public int
        exp = 0 ;



        public Character()
        { nom = "Non attribué" ;
        }

        public Character(string name)
        { this.name = name ;
        }

        public void PrintStatsInfo()
        {
                        Debug.LogFormat("Hero : {0} - {1} EXP", this.name, this.exp) ;
        }
}
```

Avec les classes, vous êtes sur la bonne voie pour écrire un code modulaire qui soit lisible, léger et réutilisable. Il est maintenant temps de s'attaquer à l'objet cousin de la classe, la structure !

DECLARER DES STRUCTURES

Les structures sont similaires aux classes en ce sens qu'elles sont également des plans pour les objets que vous souhaitez créer dans vos programmes. La principale différence réside dans le fait qu'il s'agit de **types de valeur**, ce qui signifie qu'ils sont transmis par valeur et non par référence, comme le sont les classes. Lorsque les structures sont assignées ou passées à une autre variable, une nouvelle copie de la structure est créée, de sorte que l'original n'est pas référencé du tout. Nous reviendrons plus en détail sur ce point dans la section suivante. Tout d'abord, nous devons comprendre le fonctionnement des structures et les règles spécifiques qui s'appliquent lors de leur création.

Les structures sont déclarées de la même manière que les classes et peuvent contenir des champs, des méthodes et des constructeurs :

```
accessModifier struct UniqueName
{
    Variables
    Constructeurs
    Méthodes
}
```

Comme les classes, les variables et les méthodes appartiennent exclusivement à la structure et sont accessibles par son nom unique.

Cependant, les structures ont quelques limites :

- Les variables ne peuvent pas être initialisées avec des valeurs à l'intérieur de la déclaration struct à moins qu'elles ne soient marquées avec le modificateur static ou const - vous pouvez en savoir plus à ce sujet dans *Chnuícu 10, kcvisiíing T;ucs, McíshoKs,*

```
public struct Author
    { string name = "Harrison" ; int age =
        32 ;
    }
```

nnK Clnsscs. Par exemple, le code suivant provoquerait une erreur :

- Les constructeurs sans paramètres ne sont pas autorisés. Par exemple, le code suivant provoquerait également une erreur :

```
public struct Author
{ public Author()
    {
    }
}
```

- Les structures sont dotées d'un constructeur par défaut qui fixe automatiquement toutes les variables à leur valeur par défaut en fonction de leur type.

Chaque personnage a besoin d'une bonne arme, et ces armes conviennent parfaitement à un objet struct plutôt qu'à une classe. Nous verrons pourquoi dans la section *UnKcusínnKing ucficucncc nnK vnluc í;ucs* de ce chapitre. Cependant, vous allez d'abord en créer une pour vous amuser.

Nos personnages vont avoir besoin de bonnes armes pour mener à bien leurs quêtes, qui sont de bonnes candidates pour une structure simple :

1. Cliquez avec le bouton droit de la souris sur le dossier Scripts, ch#oisissez **Créer** et sélectionnez **C Script**.
2. Nommez-le Weapon, ouvrez-le dans Visual Studio et supprimez tout le code généré après avoir utilisé UnityEngine.
3. Déclarez une structure publique appelée Weapon, suivie d'un ensemble d'accolades, puis enregistrez le fichier.
4. Ajouter un champ pour le nom de type string et un autre champ pour le dommage de type int.
5. Les classes et les structures peuvent être imbriquées les unes dans les autres, mais cette pratique est généralement déconseillée car elle encombre le code :

```
public struct Weapon
{ public string name ; public int damage ;
}
```

6. Déclarez un constructeur avec le nom et les paramètres de dommages, et définissez les champs de la structure à l'aide de la fonction

le mot-clé this :

```
public Weapon(string name, int damage)
{ this.name = name ;
        this.damage = damage ;
}
```

7. Ajoutez une méthode de débogage sous le constructeur pour imprimer les informations relatives à l'arme :

```
public void PrintWeaponStats()
{
        Debug.LogFormat("Arme : {0} - {1} DMG", this.name, this. damage)
;
}
```

8. Dans LearningCurve, créez une nouvelle structure Weapon en utilisant le constructeur personnalisé et la nouvelle structure puis utiliser la méthode PrintWeaponStats pour déboguer les valeurs de la structure :

```
Weapon huntingBow = new Weapon("Hunting Bow", 105) ; huntingBow.PrintWeaponStats() ;
```

9. Notre nouvel objet huntingBow utilise le constructeur personnalisé et fournit des valeurs pour les deux éléments suivants lors de l'initialisation.

C'est une bonne idée de limiter les scripts à une seule classe, mais il est assez courant de voir des structs
qui sont utilisés exclusivement par une classe incluse dans le fichier.

Maintenant que nous disposons d'un exemple d'objet de référence (classe) et d'objet de valeur (structure), il est temps de se familiariser avec chacun de leurs détails. Plus précisément, vous devrez comprendre comment chacun de ces objets est transmis et stocké en mémoire.

COMPRENDRE LES TYPES DE REFERENCE ET DE VALEUR

Hormis les mots-clés et les valeurs initiales des champs, nous n'avons pas vu beaucoup de différences entre les classes et les structures jusqu'à présent. Les classes conviennent mieux pour regrouper des actions complexes et des données qui changeront tout au long d'un programme ; les structures sont un meilleur choix pour les objets simples et les données qui resteront constantes la plupart du temps, comme les valeurs qui restent les mêmes tout au long du projet. Outre leur utilisation, ils sont fondamentalement différents dans un domaine clé, à savoir la manière dont ils sont transmis ou assignés entre les variables. Les classes sont des **types de référence**, c'est-à-dire qu'elles sont transmises par référence ; les structures sont des **types de valeur**, c'est-à-dire qu'elles sont transmises par valeur.

TYPES DE REFERENCE

Lorsque les instances de notre classe Character sont initialisées, les variables hero et heroine ne contiennent pas d'informations sur leur classe, mais une référence à l'emplacement de l'objet dans la mémoire du programme. Si nous assignons hero ou heroine à une autre variable de la même classe, c'est la référence mémoire qui est assignée, et non les données du personnage. Cela a plusieurs implications, la plus importante étant que si nous avons plusieurs variables stockant la même référence mémoire, une modification de l'une d'entre elles les affecte toutes.

Les sujets de ce type sont mieux démontrés qu'expliqués ; c'est à vous d'en faire l'expérience dans un exemple pratique, ensuite.

Il est temps de tester que la classe Character est un type de référence :

1. Déclarez une nouvelle variable Character dans LearningCurve appelée villain. Attribuer villain à la variable et utiliser la méthode PrintStatsInfo pour imprimer les deux ensembles d'informations.
2. Cliquez sur **Play** et regardez les deux journaux de débogage qui s'affichent dans la console :

```
Personnage héros = nouveau personnage() ; Personnage méchant= héros ;
```

```
hero.PrintStatsInfo() ;
```

3. Les deux journaux de débogage seront identiques car villain a été assigné à hero lors de sa création. À ce stade, hero et villain pointent tous deux vers l'endroit où hero est stocké en mémoire :

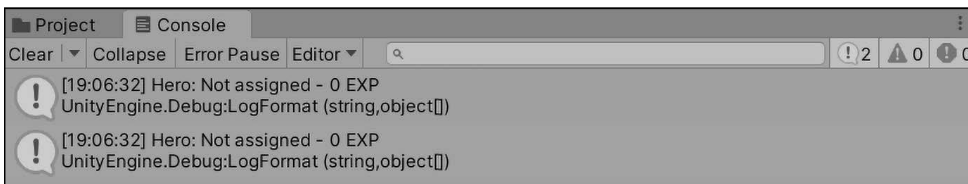


Figure 5.5 : Capture d'écran des statistiques de la structure imprimées sur la console

4. Maintenant, changez le nom du méchant en quelque chose d'amusant et cliquez à nouveau sur **Jouer :**

```
Personnage villain = hero ; villain.name = "Sir Kane the Bold" ;
```

5. Vous verrez que hero et hero2 ont maintenant le même nom, même si les données d'un seul de nos personnages ont été modifiées :

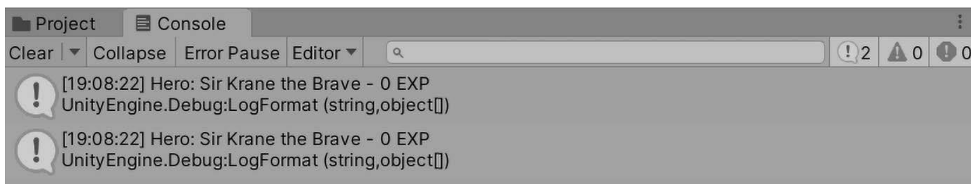


Figure 5.6 : Capture d'écran des propriétés d'une instance de classe imprimées sur la console

La leçon à retenir est que les types de référence doivent être traités avec soin et ne pas être copiés lorsqu'ils sont affectés à de nouvelles variables. Toute modification apportée à une référence se répercute sur toutes les autres variables contenant la même référence.

Si vous essayez de copier une classe, vous devez soit créer une nouvelle instance distincte, soit reconsidérer la question de savoir si une structure ne serait pas un meilleur choix pour votre modèle d'objet. Vous aurez un meilleur aperçu des types de valeurs dans la section suivante.

TYPES DE VALEURS

Lorsqu'un objet struct est créé, toutes ses données sont stockées dans la variable correspondante, sans référence ni connexion à son emplacement en mémoire. Les structures sont donc utiles pour créer des objets qui doivent être copiés rapidement et efficacement, tout en conservant leur identité propre. Essayez ceci avec notre structure Weapon dans l'exercice suivant.

Créons un nouvel objet arme en copiant huntingBow dans une nouvelle variable et en mettant

à jour ses données pour voir si les changements affectent les deux structures :

1. Déclarez une nouvelle structure Weapon dans LearningCurve, et attribuez-lui la valeur initiale huntingBow :

```
Arme huntingBow = nouvelle Arme("Arc de chasse", 105) ; Arme warBow = huntingBow ;
```

2. Imprimez les données de chaque arme à l'aide de la méthode de débogage :

```
huntingBow.PrintWeaponStats() ; warBow.PrintWeaponStats() ;
```

3. De la manière dont ils sont configurés maintenant, huntingBow et warBow auront les mêmes journaux de débogage, tout comme nos deux personnages l'avaient avant que nous ne changions les données :

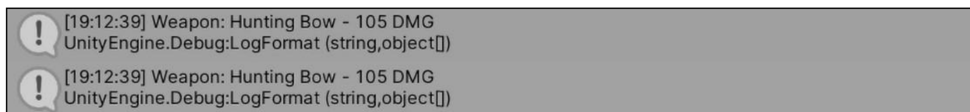


Figure 5.7 : Capture d'écran des instances struct imprimées sur la console

4. Modifiez les champs warBow.name et warBow.damage avec les valeurs de votre choix et cliquez sur Rejouer :

```
Arme warBow = huntingBow ; warBow.name = "War Bow" ;
warBow.damage = 155 ;
```

5. La console montrera que seules les données relatives à warBow ont été modifiées, et que huntingBow conserve ses données d'origine.

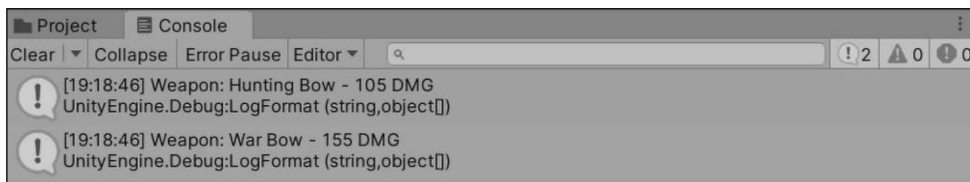


Figure 5.8 : Capture d'écran des propriétés de la structure mises à jour et affichées sur la console

Ce qu'il faut retenir de cet exemple, c'est que les structures sont facilement copiées et modifiées en tant qu'objets distincts, contrairement aux classes, qui conservent les références à l'objet d'origine. Maintenant que nous comprenons un peu mieux le fonctionnement des structs et des classes, et que nous avons confirmé le comportement des types référence et valeur dans leur habitat naturel, nous sommes bien placés pour commencer à parler de l'un des sujets de codage les plus importants, la POO, et de la façon dont elle s'inscrit dans le paysage de la programmation.

INTEGRER L'ESPRIT ORIENTE OBJET

Les objets du monde physique fonctionnent à un niveau similaire à celui de la POO ; lorsque vous voulez acheter une boisson gazeuse, vous prenez une canette de soda, et non le liquide luimême. La canette est un objet, qui regroupe des informations et des actions connexes dans un ensemble autonome. Cependant, il existe des règles concernant les objets, tant en programmation qu'à l'épicerie - par exemple, qui peut y avoir accès. Les différentes variations et les actions génériques jouent toutes un rôle dans la nature des objets qui nous entourent.

En termes de programmation, ces règles sont les principaux principes de la POO : l'**encapsulation**, l'**héritage** et l'utilisation de l'information. **le polymorphisme**. Nous allons aborder ces sujets dans les prochaines sections !

ENCAPSULATION

L'un des avantages de la POO est qu'elle permet l'encapsulation, c'est-à-dire la définition de l'accessibilité des variables et des méthodes d'un objet au code extérieur (ce que l'on appelle parfois le **code d'appel**). Prenons l'exemple de notre canette de soda : dans un distributeur automatique, les interactions possibles sont limitées. Comme la machine est verrouillée, n'importe qui ne peut pas s'approcher et en prendre une ; si vous avez la bonne monnaie, vous aurez un accès provisoire à la canette, mais dans une quantité spécifiée. Si la machine ellemême est enfermée dans une pièce, seule la personne possédant la clé de la porte saura que la canette de soda existe.

La question que vous vous posez maintenant est la suivante : comment fixer ces limites ? La réponse est simple : nous avons toujours utilisé l'encapsulation en spécifiant des modificateurs d'accès pour les variables et les méthodes de nos objets.

Si vous avez besoin d'un rafraîchissement, retournez visiter la section *"cccss moKificus"* dans *Chnuícu 3*, *Diving inío Vnuinblcs, T;ucs, nnK McíhoKs*

Essayons un exemple simple d'encapsulation pour comprendre comment cela fonctionne en pratique. Notre classe Character est publique, tout comme ses champs et ses méthodes. Cependant, que se passerait-il si nous voulions une méthode capable de réinitialiser les données d'un personnage à leurs valeurs initiales ? Cette méthode pourrait s'avérer utile, mais elle pourrait s'avérer désastreuse si elle était appelée accidentellement, ce qui en fait un candidat parfait pour un membre d'objet privé :

1. Créez une méthode privée appelée Reset, sans valeur de retour, dans la classe Character. Remettez les variables name et exp à "Not assigned" et 0, respectivement :

```
private void Reset()
{ this.name = "Non attribué" ; this.exp = 0 ;
```

```
}
```

2. Essayez d'appeler Reset() à partir de LearningCurve après avoir imprimé les données de hero2 :

```
14
15          hero.PrintStatsInfo();
16          hero2.PrintStatsInfo();
17          hero2.Reset();
Error: 'Character.Reset()' is inaccessible due to its protection level
```

Figure 5.9 : Capture d'écran d'une méthode inaccessible dans la classe Character

Si vous vous demandez si Visual Studio est cassé, ce n'est pas le cas. Le fait de marquer une méthode ou une variable comme privée la rend inaccessible en utilisant la notation par points ; elle ne peut être appelée qu'à l'intérieur de la classe ou de la structure à laquelle elle appartient. Si vous la saisissez manuellement et que vous survolez Reset(), vous verrez apparaître un message d'erreur indiquant que la méthode est protégée.

Pour appeler cette méthode privée, nous pouvons ajouter une commande Reset() dans le constructeur de la classe ou dans toute autre méthode de la classe des caractères :

```
public Character()
{
      Reset() ;
}
```

L'encapsulation permet des configurations d'accessibilité plus complexes avec les objets ; cependant, pour l'instant, nous allons nous en tenir aux membres publics et privés. Lorsque nous commencerons à étoffer notre prototype de jeu dans le prochain chapitre, nous ajouterons différents modificateurs si nécessaire.

Parlons maintenant de l'héritage, qui sera votre meilleur ami lorsque vous créerez des hiérarchies de classes dans vos futurs jeux.

HERITAGE

Une# classe C peut être créée à l'image d'une autre classe, en partageant ses variables membres et ses méthodes, mais en étant capable de définir ses propres données. En POO, on parle d'**héritage, et c'est** un moyen puissant de créer des classes apparentées sans avoir à répéter le code. Reprenons l'exemple des sodas : il existe sur le marché des sodas génériques qui possèdent toutes les mêmes propriétés de base, puis des sodas spéciaux. Les sodas spéciaux partagent les mêmes propriétés de base, mais leur marque, ou leur emballage, les différencie. Lorsque vous les regardez côte à côte, il est évident qu'il s'agit de deux canettes de soda, mais il est tout aussi évident qu'elles ne sont pas identiques.

La classe d'origine est généralement appelée classe de base ou classe mère, tandis que la classe qui en hérite est appelée classe dérivée ou classe enfant. Tous les membres de la classe de base marqués par les modificateurs d'accès public, protégé ou interne font automatiquement partie de la classe dérivée, à l'exception des constructeurs. Les constructeurs de classe appartiennent toujours à la classe qui les contient, mais ils peuvent être utilisés à partir des classes dérivées afin de réduire au minimum la répétition du code. Ne vous préoccupez pas trop des différents scénarios de classe de base pour l'instant. Essayons plutôt un exemple de jeu simple.

La plupart des jeux ont plus d'un type de personnage, alors créons une nouvelle classe appelée Paladin qui hérite de la classe Character. Vous pouvez ajouter cette nouvelle classe au script Character ou en créer un nouveau. Si vous ajoutez la nouvelle classe au script Character, assurez-vous qu'elle se trouve en dehors des crochets de la classe Character :

```
public class Character
{
      // Tout notre code précédent...
}

public class Paladin : Personnage
{

}
```

Tout comme LearningCurve hérite de MonoBehavior, tout ce que nous avons à faire est d'ajouter les deux points : et la classe de base dont nous voulons hériter, et C# fait le reste. Désormais, toutes les instances de Paladin auront accès aux propriétés name et exp ainsi qu'à la méthode PrintStatsInfo.

On considère généralement que la meilleure pratique consiste à créer un nouveau script pour les différentes classes au lieu de les ajouter aux scripts existants. Cela permet de séparer les scripts et d'éviter d'avoir trop de lignes de code dans un seul fichier (ce qu'on appelle un **fichier volumineux**).

C'est très bien, mais comment les classes héritées gèrent-elles leur construction ? Vous pouvez le découvrir dans la section section suivante.

Constructeurs de base

Lorsqu'une classe hérite d'une autre classe, elle forme une sorte de pyramide dont les variables membres descendent de la classe mère vers tous ses enfants dérivés. La classe mère ne connaît aucun de ses enfants, mais tous les enfants connaissent leur parent. Cependant, les

constructeurs de la classe mère peuvent être appelés directement à partir des constructeurs de la classe enfant, moyennant une simple modification de la syntaxe :

```
public class ChildClass : ClasseParent
{ public ChildClass() : base()
        {
        }
}
```

Le mot-clé base représente le constructeur parent - dans ce cas, le constructeur par défaut. Cependant, puisque base représente un constructeur et qu'un constructeur est une méthode, une classe enfant peut passer des paramètres à son constructeur parent en remontant la pyramide.

Puisque nous voulons que tous les objets Paladin aient une variable nom, et que Character a déjà un constructeur qui gère cela, nous pouvons appeler le constructeur de base directement depuis la classe Paladin et nous épargner la réécriture d'un constructeur :

1. Ajoutez un constructeur à la classe Paladin qui prend un paramètre de type chaîne de caractères, appelé name. Utilisez les deux points (:) et le mot-clé base pour appeler le

```
public class Paladin : Personnage
{ public Paladin(string name) : base(name)
        {

        }
}
```

   constructeur parent, en lui passant nom :

2. Dans LearningCurve, créez une nouvelle instance de Paladin appelée knight. Utilisez le constructeur de base pour assigner une valeur. Appelez PrintStatsInfo à partir de knight et regardez la console :

```
Paladin chevalier = nouveau Paladin("Sir Arthur") ; chevalier.PrintStatsInfo() ;
```

3. Le journal de débogage sera le même que celui de nos autres instances de personnages, mais avec le nom que nous avons attribué au constructeur de Paladin :

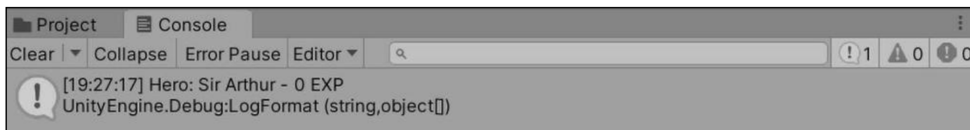


Figure 5.10 : Capture d'écran des propriétés du constructeur du caractère de base

Lorsque le constructeur Paladin se déclenche, il transmet le paramètre name au constructeur Character, qui définit la valeur de name. En fait, nous avons utilisé le constructeur Character pour faire le travail d'initialisation de la classe Paladin, rendant le constructeur Paladin uniquement responsable de l'initialisation de ses propriétés uniques, qu'il n'a pas à ce stade.

Outre l'héritage, il peut arriver que vous souhaitiez créer de nouveaux objets à partir d'une combinaison d'autres objets existants. Pensez aux *LEGO®* ; vous ne commencez pas à construire à partir de rien - vous avez déjà des blocs et des structures de différentes couleurs avec lesquels vous pouvez travailler. En termes de programmation, c'est ce qu'on appelle la **composition**, dont nous parlerons dans la section suivante.

COMPOSITION

Outre l'héritage, les classes peuvent être composées d'autres classes. Prenons l'exemple de notre structure Weapon. Paladin peut facilement contenir une variable Weapon à l'intérieur de lui-même et avoir accès à toutes ses propriétés et méthodes. Pour ce faire, nous allons mettre à jour Paladin pour qu'il prenne en charge une arme de départ et lui assigne sa valeur

```
public class Paladin : Personnage
{ public Weapon weapon ;

    public Paladin(string name, Weapon weapon) : base(name)
    { this.weapon = weapon ;
    }
}
```

dans le constructeur :

Comme l'arme est propre au Paladin et non au personnage, nous devons définir sa valeur initiale dans le constructeur. Nous devons également mettre à jour l'instance du chevalier pour inclure une variable Weapon. Retournons donc dans LearningCurve.cs et utilisons huntingBow :

```
Paladin knight = nouveau Paladin("Sir Arthur", huntingBow) ;
```

Si vous lancez le jeu maintenant, vous ne verrez rien de différent parce que nous utilisons la méthode PrintStatsInfo de la classe Character, qui ne connaît pas la propriété weapon de la classe Paladin. Pour résoudre ce problème, nous devons parler de **polymorphisme**.

POLYMORPHISME

Le polymorphisme est le mot grec pour *mnn;LshnucK* et s'applique à la POO de deux manières distinctes :

- Les objets de classe dérivés sont traités de la même manière que les objets de classe parents. Par exemple, un tableau d'objets Personnage peut également contenir des objets Paladin, puisqu'ils dérivent de Personnage.
- Les classes mères peuvent marquer les méthodes comme virtuelles, ce qui signifie que leurs instructions peuvent être modifiées par les classes dérivées à l'aide du mot-clé override. Dans le cas de Character et Paladin, il serait utile de pouvoir déboguer différents messages de PrintStatsInfo pour chacun d'entre eux.

Le polymorphisme permet aux classes dérivées de conserver la structure de leur classe mère tout en ayant la liberté d'adapter les actions à leurs besoins spécifiques. Toute méthode que vous marquez comme virtuelle vous donnera la liberté du polymorphisme d'objet. Prenons ces nouvelles connaissances et appliquons-les à notre méthode de débogage de personnage.

Modifions Character et Paladin pour qu'ils affichent différents journaux de débogage à l'aide de PrintStatsInfo :

1. Modifier PrintStatsInfo dans la classe Character en ajoutant le mot-clé virtual entre publics et nuls :

```
public virtual void PrintStatsInfo()
{
    Debug.LogFormat("Héros : {0} - {1} EXP", nom, exp) ;
}
```

2. Déclarez la méthode PrintStatsInfo dans la classe Paladin en utilisant le mot-clé override. Ajoutez un journal de débogage pour imprimer les propriétés de Paladin de la manière que vous souhaitez :

```
public override void PrintStatsInfo()
{
    Debug.LogFormat("Salut {0} - prenez votre {1} !", this.name,
            this.weapon.name) ;
}
```

3. Cela peut ressembler à du code répété, ce qui, nous l'avons déjà dit, n'est pas une bonne chose, mais il s'agit d'un cas particulier. Ce que nous avons fait en marquant PrintStatsInfo comme virtuel dans la classe Character, c'est d'indiquer au compilateur que cette méthode peut avoir plusieurs formes en fonction de la classe appelante.
4. Lorsque nous avons déclaré la version surchargée de PrintStatsInfo dans Paladin, nous avons ajouté le comportement personnalisé qui ne s'applique qu'à cette classe. Grâce au polymorphisme, nous n'avons pas à

choisir quelle version de PrintStatsInfo nous voulons appeler à partir d'un objet Character ou Paladin - le compilateur le sait déjà :

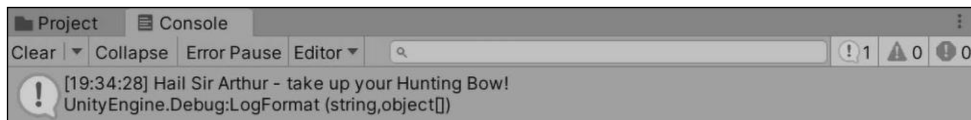


Figure 5.11 : Capture d'écran des propriétés des caractères polymorphes

C'était beaucoup à assimiler, je le sais. Passons donc en revue quelques-uns des principaux points de la POO à mesure que nous approchons de la fin de l'année. la ligne d'arrivée :

- La POO consiste à regrouper des données et des actions connexes dans des objets - des objets qui peuvent communiquer et agir indépendamment les uns des autres.
- L'accès aux membres de la classe peut être défini à l'aide de modificateurs d'accès, tout comme les variables
- Les classes peuvent hériter d'autres classes, créant ainsi des hiérarchies descendantes de parents/enfants.
  relations
- Les classes peuvent avoir des membres d'autres types de classes ou de structures
- Les classes peuvent remplacer toutes les méthodes parentales marquées comme virtuelles, ce qui leur permet d'effectuer des actions personnalisées tout en conservant le même schéma directeur.

La POO n'est pas le seul paradigme de programmation qui peut être utilisé avec le langage C - vous pouvez trouver des exemples pratiques d'utilisation de la POO.

explications des autres approches principales ici : http://cs.lmu.edu/~ray/notes/paradigmes.

Toute la POO que vous avez apprise dans ce chapitre est directement applicable au monde C. Cependant, nous devons encore mettre cela en perspective avec Unity, et c'est ce à quoi vous allez consacrer le reste du chapitre.

APPLICATION DE LA POO DANS UNITY

Si vous fréquentez suffisamment les langages OOP, vous finirez par entendre la phrase "*cvcu;íhing is nn objccí*" chuchotée comme une prière secrète entre développeurs. Selon les principes de la POO, tout dans un programme devrait être un objet, mais les GameObjects dans Unity peuvent représenter vos classes et vos structures. Cependant, cela ne veut pas dire que tous les objets dans Unity doivent être dans la scène physique, donc nous pouvons

toujours utiliser nos nouvelles classes programmées dans les coulisses.

LES OBJETS ONT DE LA CLASSE

Dans *Chnuícu 2*, *Thc DuilKing Dlocks ofi Fuogunmming*, nous avons vu comment un script est transformé en composant lorsqu'il est ajouté à un GameObject dans Unity. Pensez-y en termes de principe OOP de composition : les GameObjects sont les conteneurs parents, et ils peuvent être constitués de plusieurs composants. Cela peut sembler contradictoire avec l'idée d'une classe C par script mais, en vérité, il s'agit plus d'une ligne directrice pour une meilleure lisibilité que d'une exigence réelle. Les classes peuvent être imbriquées les unes dans les autres, mais cela devient vite compliqué. Cependant, le fait d'avoir plusieurs composants de script attachés à un seul objet de jeu peut être très utile, en particulier lorsqu'il s'agit de classes de gestionnaire ou de comportements.

Essayez toujours de réduire les objets à leurs éléments les plus élémentaires, puis utilisez la composition pour construire des objets plus grands et plus complexes à partir de ces petites classes. Il est plus facile de modifier un objet de jeu composé de petits éléments interchangeables qu'un objet volumineux et encombrant.

Jetons un coup d'œil à l'**appareil photo principal** pour voir comment cela fonctionne :

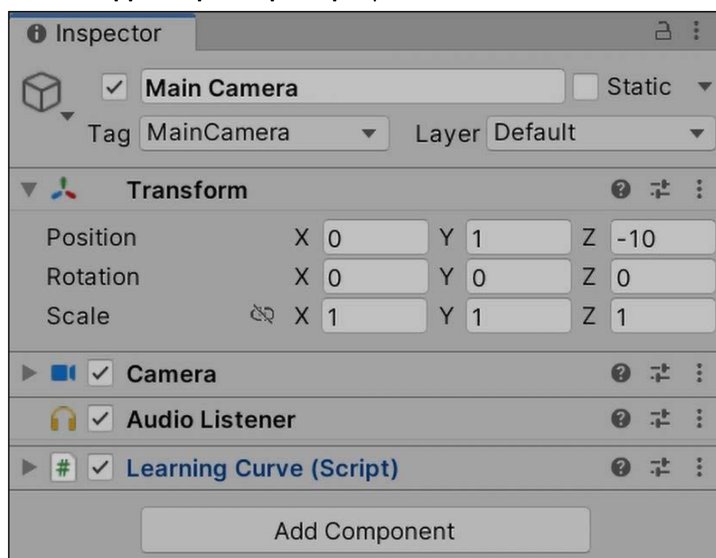


Figure 5.12 : Capture d'écran de l'objet Caméra principale dans l'inspecteur

Chaque composant de la capture d'écran précédente (**Transform**, **Camera**, **Audio Listener**, et le script **Learning Curve**) a commencé comme une classe dans Unity. Comme les instances de Character ou Weapon, ces composants deviennent des objets dans la mémoire de l'ordinateur lorsque nous cliquons sur **Play**, avec leurs variables membres et leurs méthodes.

Si nous attachons LearningCurve (ou n'importe quel script ou composant) à 1 000 GameObjects et que nous cliquons sur **Play**, 1 000 instances distinctes de LearningCurve seront créées et stockées en mémoire.

Nous pouvons même créer nos instances de ces composants en utilisant leur nom comme type de données. Comme les classes, les classes de composants Unity sont des types de référence et peuvent être créées comme n'importe quelle autre variable. Cependant, la recherche et l'affectation de ces composants Unity sont légèrement différentes de ce que vous avez vu jusqu'à présent. Pour cela, vous devrez comprendre un peu mieux le fonctionnement des GameObjects dans la section suivante.

ACCES AUX COMPOSANTS

Maintenant que nous savons comment les composants agissent sur les GameObjects, comment pouvons-nous accéder à leurs instances spécifiques ? Heureusement pour nous, tous les GameObjects dans Unity héritent de la classe GameObject, ce qui signifie que nous pouvons utiliser leurs méthodes membres pour trouver tout ce dont nous avons besoin dans une scène. Il existe deux façons d'assigner ou de récupérer des GameObjects actifs dans la scène en cours :

1. Par le biais des méthodes GetComponent() ou Find() de la classe GameObject, qui fonctionnent avec des variables publiques et privées. Cependant, il est important de faire attention à ces deux méthodes ; pour des performances optimales et de bonnes pratiques, le résultat de l'appel GetComponent() doit toujours être sauvegardé dans ses propres variables et Find() doit être utilisé avec parcimonie et jamais dans une boucle Update().
2. En glissant-déposant les GameObjects eux-mêmes depuis le panneau Projet directement dans les emplacements de variables de l'onglet **Inspecteur.** Cette option ne fonctionne qu'avec les variables publiques dans C, puisque ce sont les seules qui apparaîtront dans l'**inspecteur**. Si vous décidez qu'une variable privée doit être affichée dans l'**inspecteur**, vous pouvez la marquer avec l'attribut SerializeField.

Pour en savoir plus sur les attributs et SerializeField, consultez la documentation Unity : https://docs.unity3d.com/ScriptReference/SerializeField.html.

Examinons la syntaxe de la première option.

Accès aux composants dans le code

L'utilisation de GetComponent est assez simple, mais la signature de sa méthode est légèrement différente des autres méthodes que nous avons vues jusqu'à présent :

```
GameObject.GetComponent<ComponentType>() ;
```

Tout ce dont nous avons besoin est le type de composant que nous recherchons, et la classe GameObject renverra le composant s'il existe et null s'il n'existe pas. Il existe d'autres variantes de la méthode GetComponent, mais celle-ci est la plus simple car nous n'avons pas besoin de connaître les spécificités de la classe GameObject que nous recherchons.

C'est ce qu'on appelle une méthode générique, dont nous parlerons plus en détail dans *Chnuícu 13*, *Exulouing Gcncuics, Dclcgnícs, nnK Dc;onK*. Cependant, pour l'instant, contentons-nous de travailler avec la transformation de la caméra.

Puisque LearningCurve est déjà attaché à l'objet **Main Camera**, prenons le composant Transform de la caméra et stockons-le dans une variable publique. Le composant Transform contrôle la position, la rotation et l'échelle d'un objet dans Unity, c'est donc un exemple pratique :

1. Dans LearningCurve, ajouter une nouvelle variable publique de type Transform, appelée CamTransform :

```
public Transform CamTransform ;
```

2. Initialiser CamTransform dans Start en utilisant la méthode GetComponent de la classe GameObject. Utilisez le mot-clé this, puisque LearningCurve est attaché au même composant GameObject que le composant Transform.
3. Accédez à la propriété localPosition de CamTransform et déboguez-la en utilisant la notation par points (notez que nous stockons le composant dans sa propre variable pour des

```
void Start()
{
        CamTransform = this.GetComponent<Transform>() ;
        Debug.Log(CamTransform.localPosition) ;
}
```

raisons de performance) :

4. Nous avons ajouté une variable Transform publique non initialisée au sommet de LearningCurve et l'avons initialisée en utilisant la méthode GetComponent à l'intérieur de Start. GetComponent trouve le composant Transform attaché à ce composant GameObject et le renvoie à CamTransform. CamTransform stockant désormais un objet Transform, nous avons accès à toutes les propriétés et méthodes de sa classe, y compris localPosition dans la capture d'écran suivante :

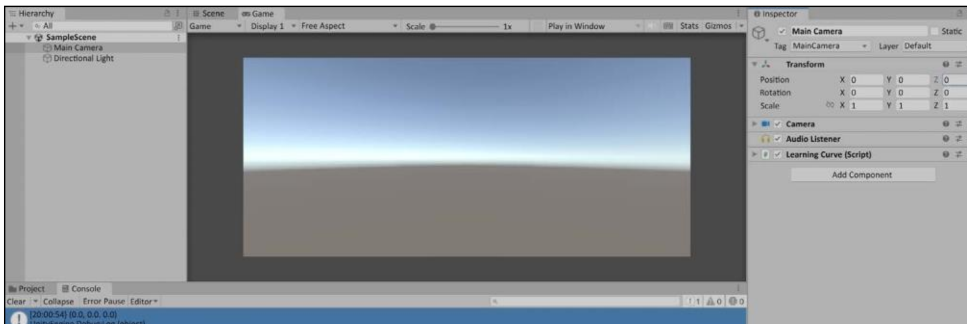


Figure 5.13 : Capture d'écran de la position de transformation imprimée sur la console

La méthode GetComponent est fantastique pour récupérer rapidement des composants, mais elle n'a accès qu'aux composants du GameObject auquel le script appelant est attaché. Par exemple, si nous utilisons GetComponent à partir du script LearningCurve attaché à la **caméra principale**, nous ne pourrons accéder qu'aux composants **Transform**, **Camera** et **Audio Listener.**

Si nous voulons référencer un composant sur un GameObject séparé, tel que **Directional Light**, nous devons d'abord obtenir une référence à l'objet à l'aide de la méthode Find. Tout ce qu'il faut, c'est le nom d'un GameObject, et Unity renverra le GameObject approprié pour que nous puissions le stocker ou le manipuler.

Pour référence, le nom de chaque GameObject se trouve en haut de l'onglet **Inspecteur** lorsque l'objet est sélectionné :

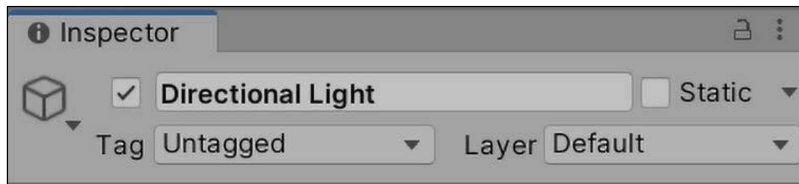


Figure 5.14 : Capture d'écran de l'objet Lumière directionnelle dans l'inspecteur

La recherche d'objets dans les scènes de votre jeu est cruciale dans Unity, c'est pourquoi vous devrez vous entraîner. Prenons l'exemple du nous devons travailler avec des objets et nous entraîner à trouver et à attribuer leurs composants.

Utilisons la méthode Find et récupérons l'objet **Lumière directionnelle** à partir de Courbe d'apprentissage :

1. Ajouter deux variables à LearningCurve sous CamTransform - une de type GameObject et un autre de type Transform :

```
public GameObject DirectionLight ; public Transform LightTransform ;
```

2. Trouver le composant Lumière directionnelle par son nom et l'utiliser pour initialiser DirectionLight dans la méthode Start() :

```
void Start()
{
DirectionLight = GameObject.Find("Directional Light") ;
}
```


3. Fixez la valeur de LightTransform au composant Transform attaché à DirectionLight, et déboguez sa position locale. Comme DirectionLight est maintenant son GameObject, GetComponent fonctionne parfaitement :

```
LightTransform = DirectionLight.GetComponent<Transform>() ; Debug.Log(LightTransform.localPosition) ;
```

4. Avant de lancer le jeu, il est important de comprendre que les appels de méthodes peuvent être enchaînés pour réduire le nombre d'étapes du code. Par exemple, nous pourrions initialiser LightTransform en une seule ligne en combinant Find et GetComponent sans passer par DirectionLight :

```
GameObject.Find("Lumière directionnelle").GetComponent<Transform>() ;
```

Un mot d'avertissement : les longues lignes de code enchaîné peuvent entraîner une mauvaise lisibilité et de la confusion lorsque l'on travaille sur des applications complexes. En règle générale, il est préférable d'éviter les lignes plus longues que cet exemple.

Bien que la recherche d'objets dans le code fonctionne toujours, vous pouvez aussi simplement faire glisser et déposer les objets. dans l'onglet **Inspecteur**. Nous allons voir comment procéder dans la section suivante.

Glisser-déposer

Maintenant que nous avons abordé la façon de faire qui nécessite beaucoup de code, jetons un coup d'œil rapide à la fonctionnalité de glisser-déposer d'Unity. Bien que le glisser-déposer soit beaucoup plus rapide que l'utilisation de la classe GameObject dans le code, Unity perd parfois les connexions entre les objets et les variables établies de cette manière lors de l'enregistrement ou de l'exportation des projets, ou lors des mises à jour d'Unity.

Lorsque vous avez besoin d'assigner rapidement quelques variables, vous pouvez tout à fait profiter de cette fonctionnalité. Dans la plupart des cas, je conseille de s'en tenir au code.

Modifions LearningCurve pour montrer comment affecter un composant GameObject par glisser-déposer :

1. Commentez la ligne de code suivante, où nous avons utilisé GameObject.Find() pour récupérer et assigner l'objet **Directional Light** à la variable DirectionLight :

```
//DirectionLight = GameObject.Find("Directional Light") ;
```

2. Sélectionnez le GameObject **Caméra principale**, faites glisser la **Lumière directionnelle** vers la Lumière directionnelle. dans le composant **Courbe d'apprentissage,** et cliquez sur **Lecture** :

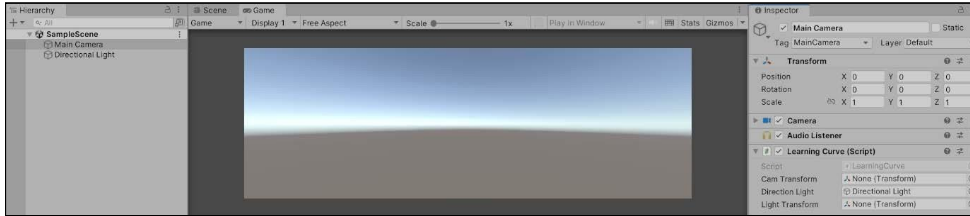


Figure 5.15 : Capture d'écran du glissement de la lumière directionnelle vers la propriété du script

3. Le GameObject **Directional Light** est maintenant assigné à la variable DirectionLight. Aucun code n'a été impliqué car Unity a assigné la variable en interne, sans modification de la classe LearningCurve.

Il est important de comprendre certaines choses lorsque vous décidez d'assigner des variables en utilisant le glisser-déposer ou la méthode GameObject.Find(). Tout d'abord, la méthode Find() est légèrement plus lente, ce qui expose votre jeu à des problèmes de performances si vous appelez la méthode plusieurs fois dans plusieurs scripts. Deuxièmement, vous devez vous assurer que vos GameObjects ont tous des noms uniques dans la hiérarchie de la scène ; si ce n'est pas le cas, cela peut entraîner des bugs désagréables dans les situations où vous avez plusieurs objets du même nom ou si vous changez les noms des objets eux-mêmes.

RESUME

Notre voyage dans les classes, les structures et la POO marque la fin de la première section sur les principes fondamentaux du langage C#. Vous avez appris à déclarer vos classes et vos structures, ce qui constitue l'échafaudage de chaque application ou jeu que vous réaliserez. Vous avez appris à déclarer vos classes et vos structures, ce qui constitue l'échafaudage de toutes les applications et de tous les jeux que vous réaliserez. Vous avez également identifié les différences dans la manière dont ces deux objets sont transmis et accédés et comment ils sont liés à la POO. Enfin, vous vous êtes familiarisé avec les principes de la POO - création de classes à l'aide de l'héritage, de la composition et du polymorphisme.

L'identification de données et d'actions connexes, la création de plans pour leur donner une forme et l'utilisation d'instances pour construire des interactions sont des bases solides pour aborder n'importe quel programme ou jeu. Ajoutez à cela la possibilité d'accéder aux composants et vous aurez l'étoffe d'un développeur Unity.

Le prochain chapitre abordera les bases du développement de jeux et du scripting du comportement des objets directement dans Unity. Nous commencerons par définir les

exigences d'un simple jeu d'aventure en monde ouvert, nous travaillerons avec des GameObjects dans la scène et nous terminerons avec un environnement en boîte blanche prêt à accueillir nos personnages.

PETIT QUIZ - TOUT CE QUI EST OOP

1. Quelle méthode gère la logique d'initialisation à l'intérieur d'une classe ?
2. Les structs étant des types de valeurs, comment sont-ils transmis ?
3. Quels sont les trois principaux principes de la POO ?
4. Quelle méthode de la classe GameObject utiliseriez-vous pour trouver un composant sur le même objet ? comme classe appelante ?

N'oubliez pas de comparer vos réponses aux miennes dans l'annexe *Fou Quiz "nswcus"* pour voir où vous en êtes !

Rejoignez-nous sur discord !

Lisez ce livre en compagnie d'autres utilisateurs, d'experts en développement de jeux Unity et de l'auteur luimême.

Posez des questions, apportez des solutions aux autres lecteurs, discutez avec l'auteur via des sessions "Ask Me Anything" et bien plus encore. Ask Me Anything et bien plus encore.

Scannez le code QR ou visitez le lien pour rejoindre la communauté.

https://packt.link/csharpwithunity

6

Se salir les mains avec L'unité

La création d'un jeu va bien au-delà de la simple simulation d'actions dans le code. La conception, l'histoire, l'environnement, l'éclairage et l'animation jouent tous un rôle important dans la mise en scène de vos joueurs. Un jeu est avant tout une expérience, quelque chose que le code seul ne peut pas offrir.

Unity s'est placé à l'avant-garde du développement de jeux au cours de la dernière décennie en mettant des outils avancés à la disposition des programmeurs et des non-programmeurs. L'animation et les effets, l'audio, la conception de l'environnement et bien d'autres choses encore sont tous disponibles directement dans l'éditeur Unity, sans une seule ligne de code. Nous aborderons ces sujets au fur et à mesure que nous définirons les besoins, l'environnement et les mécanismes de notre jeu. Cependant, nous avons d'abord besoin d'une introduction à la conception d'un jeu.

La théorie de la conception des jeux est un vaste domaine d'étude et l'apprentissage de tous ses secrets peut prendre toute une carrière. Cependant, nous n'aborderons que les bases ; tout le reste est à explorer ! Ce chapitre nous prépare au reste du livre et couvre les sujets suivants :

- Un abécédaire de la conception de jeux
- Construire un niveau
- Les bases de l'éclairage

- Animer dans Unity

UN ABECEDAIRE DE LA CONCEPTION DE JEUX

Avant de se lancer dans un projet de jeu, il est important d'avoir une idée de ce que l'on veut construire. Parfois, les idées sont très claires dans votre esprit, mais dès que vous commencez à créer des classes de personnages ou des environnements, les choses semblent s'éloigner de votre intention initiale. C'est là que la conception du jeu vous permet de planifier les points de contact suivants :

- **Concept** : L'idée générale et la conception d'un jeu, y compris son genre et son style de jeu.
- **Mécanismes de base** : Les caractéristiques jouables ou les interactions qu'un personnage peut prendre dans le jeu. Les mécanismes de jeu les plus courants sont le saut, le tir, la résolution d'énigmes ou la conduite.
- **Schémas de contrôle** : Une carte des boutons et/ou des touches qui permettent aux joueurs de contrôler leur le personnage, les interactions avec l'environnement et d'autres actions exécutables.
- **Histoire** : Le récit sous-jacent qui alimente un jeu, créant une empathie et un lien entre les joueurs et l'univers dans lequel ils évoluent.
- **Style artistique** : L'aspect général du jeu, cohérent entre les personnages, les menus, les niveaux et les environnements.
- **Conditions de victoire et de défaite** : Les règles qui régissent la manière dont le jeu est gagné ou perdu, et qui consistent généralement en des objectifs ou des buts qui ont le poids d'un échec potentiel.

Ces sujets ne constituent en aucun cas une liste exhaustive de ce qui entre dans la conception d'un jeu. Cependant, ils constituent un bon point de départ pour élaborer ce que l'on appelle un **document de conception de jeu**, qui est votre prochaine tâche !

DOCUMENTS RELATIFS A LA CONCEPTION DU JEU

En cherchant sur Google des documents de conception de jeux, vous obtiendrez un flot de modèles, de règles de formatage et de directives de contenu qui peuvent inciter un nouveau programmeur à tout abandonner. En réalité, les documents de conception sont adaptés à l'équipe ou à l'entreprise qui les crée, ce qui les rend beaucoup plus faciles à rédiger que ce qu'Internet voudrait vous faire croire.

En général, il existe trois types de documentation sur la conception :

- **Document de conception du jeu** (**GDD)** : Le GDD contient tout, de la façon dont le jeu est joué à son atmosphère, son histoire et l'expérience qu'il tente de créer. Selon le jeu, ce document peut compter quelques pages ou plusieurs centaines.
- **Document de conception technique** (**DCT)** : Ce document se concentre sur tous les

aspects techniques du jeu, du matériel sur lequel il fonctionnera à la façon dont les classes et l'architecture du programme doivent être construites. Comme pour le GDD, la longueur du document varie en fonction du projet.

- **Une page** : Généralement utilisé dans des situations de marketing ou de promotion, un document d'une page est essentiellement un instantané de votre jeu. Comme son nom l'indique, il ne doit occuper qu'une seule page.

Il n'y a pas de bonne ou de mauvaise manière de mettre en page un GDD, c'est donc un bon endroit pour laisser libre cours à votre créativité. Ajoutez des images de documents de référence qui vous inspirent ; faites preuve de créativité dans la mise en page - c'est ici que vous pouvez définir votre vision.

Le jeu sur lequel nous allons travailler dans le reste de ce livre est assez simple et ne nécessitera rien d'aussi détaillé qu'un GDD ou un TDD. Au lieu de cela, nous allons créer un onepager pour garder une trace des objectifs de notre projet et de quelques informations de base.

LE HEROS NE ONE-PAGER

Pour nous permettre de rester sur la bonne voie, j'ai élaboré un document simple qui présente les bases du prototype du jeu. Lisez-le avant de continuer, et essayez d'imaginer la mise en pratique de certains des concepts de programmation que nous avons appris jusqu'à présent :

Concept
Game prototype focused on stealthily avoiding enemies and collecting health items - with a little FPS on the side.

Gameplay
Main mechanic centers around using line-of-sight to stay one step ahead of patrolling enemies and collecting required items.

Combat will consist of shooting projectiles at enemies, which will automatically trigger an attack response.

Interface
Control scheme for movement will be the WASD or arrow keys using the mouse for camera control. Shooting mechanic will use the Space bar, and item collection will work off of object collisions.

Simple HUD will show items collected and remaining ammo, as well as a standard health bar.

Art Style
Level and character art style will be all primitive GameObjects for fast and efficient, no-frills development. These can be swapped out at a later date with 3D models or terrain environments if needed.

Figure 6.1 : Document d'une page de Hero Born

Pour uuoviKc n comulcíc vicw ofi íhc Uní ; cKiíou, nll ouru scuccnshoís nuc ínkcn in fiullLscuccn moKc. Pour les images en couleur de tous les livres, utilisez le lien suivant : https://packt.link/7yy5V.

Maintenant que vous avez une vue d'ensemble des piliers de notre jeu, vous êtes prêt à construire un prototype de niveau pour accueillir l'expérience de jeu.

CONSTRUIRE UN NIVEAU

Lorsque vous construisez les niveaux de votre jeu, il est toujours bon d'essayer de voir les choses du point de vue de vos joueurs. Comment voulez-vous qu'ils voient l'environnement, qu'ils interagissent avec lui et qu'ils se sentent lorsqu'ils s'y promènent ? Vous construisez littéralement le monde dans lequel votre jeu existe, alors soyez cohérent.

Avec Unity, vous pouvez utiliser des formes 3D de base pour créer des environnements simples, l'**outil** plus avancé **ProBuilder** ou un mélange des deux. Vous pouvez même importer des modèles 3D d'autres programmes, tels que Blender, pour les utiliser comme objets dans vos scènes.

Unity propose une excellente introduction à l'outil ProBuilder à l'adresse suivante : https://unity.com/features/probuilder.

Vous pouvez également utiliser des outils tels que Blender pour créer les éléments de votre jeu, que vous trouverez à l'adresse suivante

à l'adresse suivante : https://www.blender.org/features/modeling/.

Pour *Hcuo Doun*, nous nous en tiendrons à une simple arène intérieure, facile à parcourir, mais avec quelques recoins pour se cacher. Vous assemblerez tout cela à l'aide de **primitives (**formes d'objets de **base** fournies par Unity), car elles sont faciles à créer, à mettre à l'échelle et à positionner dans une scène.

CREATION DE PRIMITIVES

En regardant les jeux auxquels vous jouez régulièrement, vous vous demandez probablement comment vous allez pouvoir créer des modèles et des objets si réalistes qu'il vous semblera possible de les attraper à travers l'écran. Heureusement, Unity dispose d'un ensemble de GameObjects primitifs que vous pouvez sélectionner pour créer des prototypes plus rapidement. Ces objets ne seront pas très sophistiqués ou de haute définition, mais ils vous sauveront la vie si vous apprenez les ficelles du métier ou si vous n'avez pas d'artiste 3D dans votre équipe de développement.

Si vous ouvrez Unity, vous pouvez aller dans le panneau **Hiérarchie** et cliquer sur **+** > **Objet 3D**, et vous verrez toutes les options disponibles, mais seulement la moitié d'entre elles sont des primitives ou des formes communes, comme indiqué dans la capture d'écran suivante :

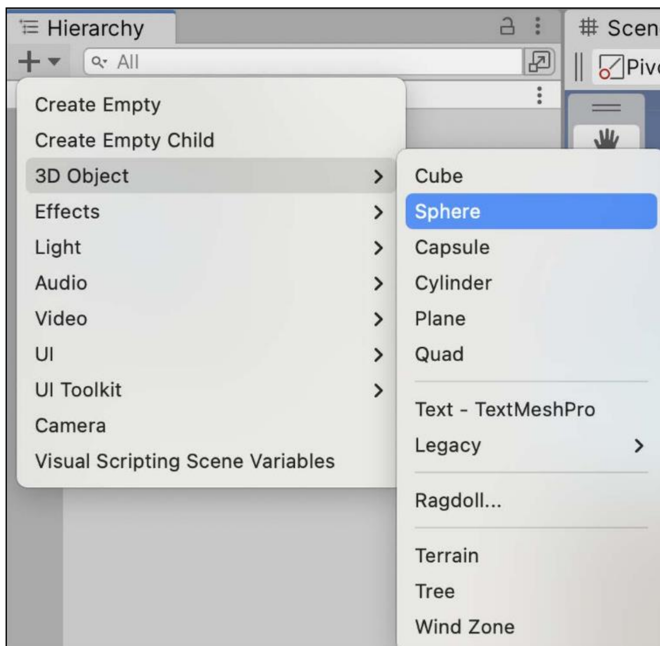


Figure 6.2 : Fenêtre de hiérarchie Unity avec l'option Objet 3D sélectionnée

Les autres options d'**objets 3D**, telles que **Terrain**, **Zone de vent** et **Arbre**, sont un peu trop avancées pour ce dont nous avons besoin, mais n'hésitez pas à les expérimenter si cela vous

Vous pouvez en savoir plus sur la création d'environnements Unity à l'adresse suivante : https://docs . unity3d.com/Manual/CreatingEnvironments.html.

intéresse.

Avant d'aller trop loin, sachez qu'il est généralement plus facile de se déplacer lorsqu'il y a un plancher en dessous Commençons donc par créer un plan de masse pour notre arène en suivant les étapes suivantes :

1. Dans le panneau **Hiérarchie**, cliquez sur **+** > **Objet 3D** > **Plan**.
2. Sélectionnez le nouvel objet dans l'onglet **Hiérarchie** et renommez le GameObject en Ground dans l'onglet **Inspecteur** ou en appuyant sur *Enícu*.

3. Dans le menu déroulant **Transformer**, réglez l'**échelle sur** 3 pour les axes **X**, **Y** et **Z :**

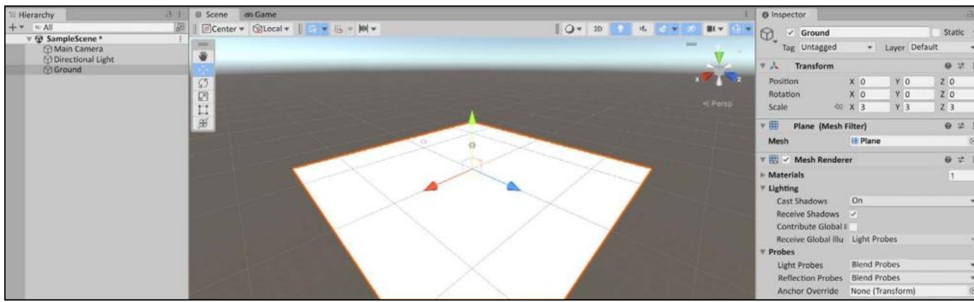


Figure 6.3 : Editeur Unity avec un plan de masse

4. Si l'éclairage de votre scène semble plus faible ou différent de l'image précédente, sélectionnez **Lumière directionnelle** dans le panneau **Hiérarchie** et réglez la valeur **Intensité** du composant **Lumière directionnelle** sur 1 :

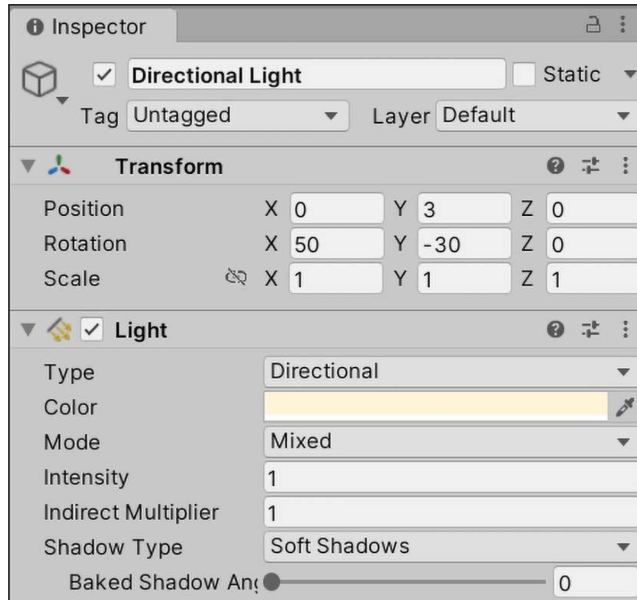


Figure 6.4 : Objet Lumière directionnelle sélectionné dans le panneau Inspecteur

Ici, nous avons créé un GameObject avion, et nous avons augmenté sa taille pour faire plus de place à nos futurs personnages. Ce plan se comportera comme un objet 3D lié à la physique de la vie réelle, ce qui signifie que les autres objets sauront qu'il est là et ne tomberont pas simplement à travers le sol dans l'oubli. Nous en dirons plus sur le système physique d'Unity et son fonctionnement dans *Chnuícu T*, *Movcmcní, Cnmcun Coníuols, nnK Collisions*. Pour l'instant, nous devons commencer à penser en 3D.

PENSER EN 3D

Maintenant que nous avons notre premier objet dans la scène, nous pouvons parler de l'espace 3D, et plus précisément de la façon dont la position, la rotation et l'échelle d'un objet se comportent en trois dimensions. Si vous vous souvenez de vos cours de géométrie au lycée, un graphique avec un système de coordonnées *x* et *;* devrait vous être familier. Pour placer un point sur le graphique, vous avez besoin d'une valeur *x* et d'une valeur *;*.

Unity prend en charge le développement de jeux en 2D et en 3D, et si nous réalisions un jeu en 2D, nous pourrions nous en tenir à cette explication. Cependant, lorsqu'il s'agit d'un espace 3D dans l'éditeur Unity, nous disposons d'un axe supplémentaire, appelé axe *z*. L'axe z représente la profondeur, ou la perspective. L'axe *z* représente la profondeur, ou la perspective, ce qui confère à notre espace et aux objets qu'il contient leur qualité 3D.

Cela peut être déroutant au début, mais Unity dispose de quelques aides visuelles intéressantes pour vous aider à y voir plus clair. Dans le coin supérieur droit du panneau **Scène**, vous verrez une icône géométrique avec les axes *x*, *;* et *z* marqués en rouge, vert et bleu, respectivement. Tous les GameObjects de la scène affichent leurs flèches d'axe lorsqu'ils sont sélectionnés dans la fenêtre **Hiérarchie :**

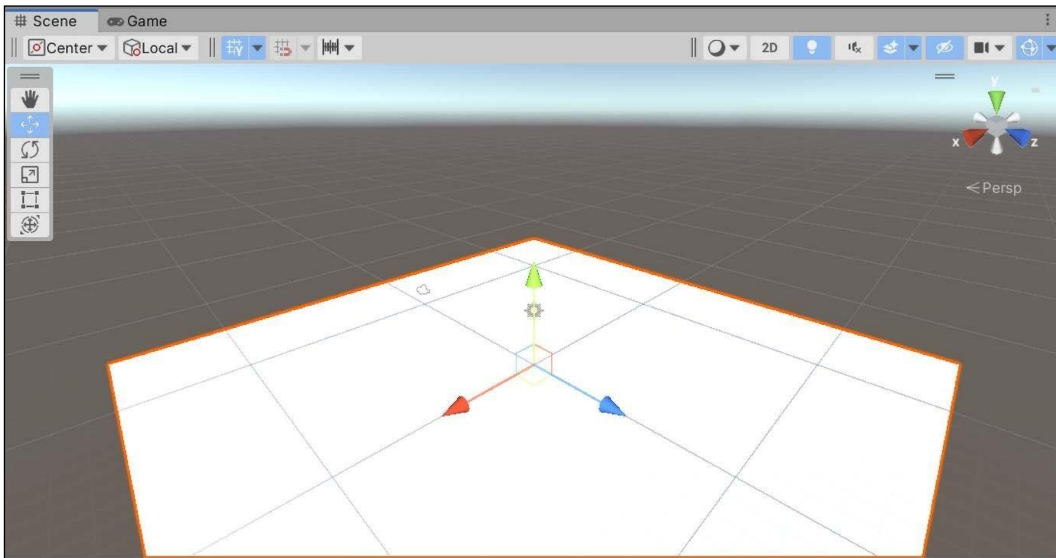


Figure 6.5 : Vue de la scène avec le gizmo d'orientation en surbrillance

L'orientation actuelle de la scène et des objets placés à l'intérieur est toujours affichée. En cliquant sur l'un des axes colorés, l'orientation de la scène passe à l'axe sélectionné. Essayezle vous-même pour vous familiariser avec le changement de perspective.

Si vous examinez le composant **Transform de** l'objet **Ground** dans la fenêtre de l'**inspecteur**, vous verrez que la position, la rotation et l'échelle sont toutes déterminées par ces trois

axes.

La position détermine l'emplacement de l'objet dans la scène, la rotation régit son angle et l'échelle s'occupe de sa taille. Ces valeurs peuvent être modifiées à tout moment dans la sous-

fenêtre de l' **inspecteur** ou dans un script C# :

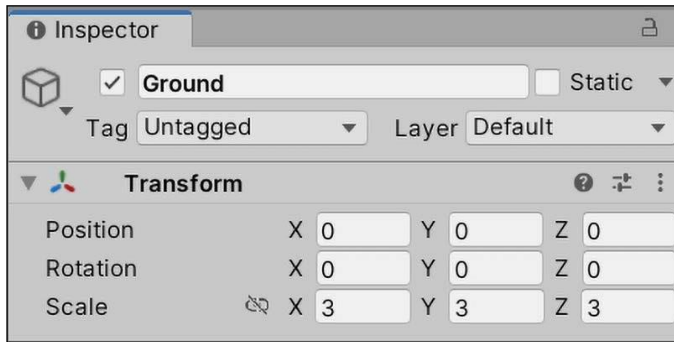


Figure 6.6 : Objet sol sélectionné dans la hiérarchie

Pour l'instant, le sol est un peu ennuyeux. Changeons cela avec un matériau.

MATERIAUX

Notre plan de sol n'est pas très intéressant pour l'instant, mais nous pouvons utiliser les **matériaux** pour donner un peu de vie au niveau. Les matériaux sont chargés de définir les propriétés du GameObject telles que la couleur et la texture ; le matériau est transmis au shader, qui utilise le shader pour rendre les propriétés du matériau à l'écran. Les **shaders** sont chargés de combiner les données d'éclairage et de texture en une représentation de l'apparence du matériau.

Chaque GameObject est doté d'un **matériau** et d'un **shader** par défaut (illustrés ici dans l'**inspecteur**), en réglant sa couleur sur le blanc standard :

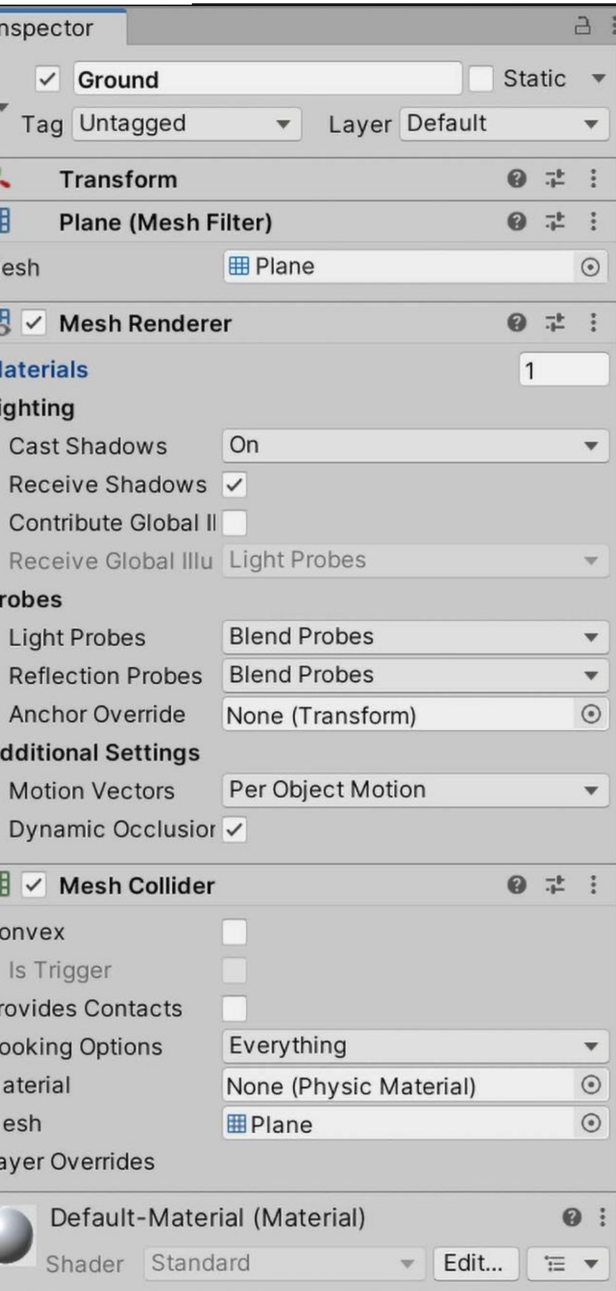


Figure 6.7 : Matériau par défaut d'un objet

Pour modifier la couleur d'un objet, nous devons créer un matériau et le faire glisser sur l'objet que nous voulons modifier. N'oubliez pas que tout est un objet dans Unity - il en va de même pour les matériaux. Les matériaux peuvent être réutilisés sur autant de GameObjects que nécessaire, mais toute modification apportée à un matériau se répercute sur tous les objets auxquels il est attaché. Si nous avons plusieurs objets ennemis dans la scène avec un matériau qui leur donne tous une couleur rouge, et que nous changeons la couleur du matériau de base en bleu, tous nos ennemis seront alors bleus.

Le bleu attire l'attention ; changeons la couleur du plan de masse pour l'assortir et créons un nouveau matériau pour transformer le plan de masse d'un blanc terne en un bleu foncé et vibrant :

1. Créez un nouveau dossier dans le panneau **Projet** en *cliquant haut* > **Créer** > **Dossier**, et nommez-le Matériaux.
2. Dans le dossier **Materials**, *uighíLclick* > **Create** > **Material**, et nommez-le Ground_Mat.
3. Sélectionnez le nouveau matériau dans le panneau **Projet** et examinez ses propriétés dans l'**inspecteur**. Cliquez sur la case de couleur à côté de la propriété **Albedo**, sélectionnez votre couleur dans la fenêtre de sélection des couleurs qui s'affiche (le grand carré au milieu définit la couleur actuelle), puis fermez-la.

4. Faites glisser l'objet Ground_Mat depuis le panneau **Project** et déposez-le sur le GameObject Ground dans le panneau **Hierarchy :**

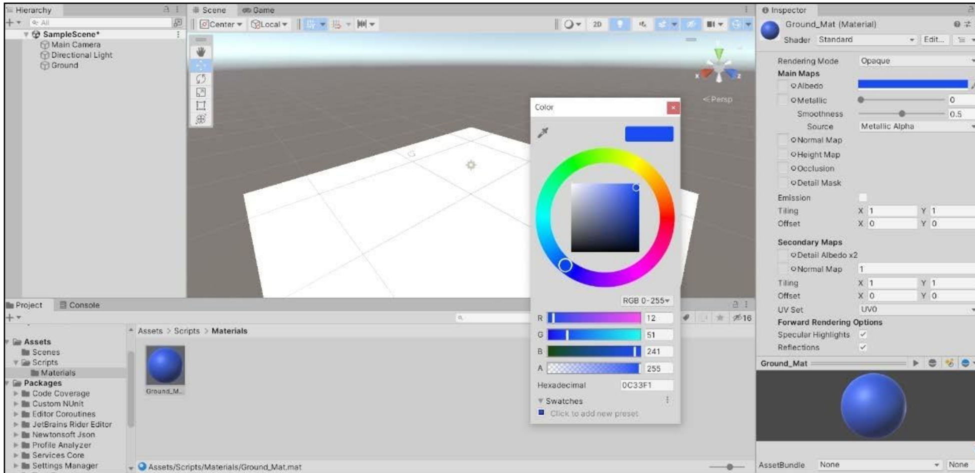


Figure 6.8 : Sélecteur de couleur de matériau

Le nouveau matériau que vous avez créé est maintenant une ressource du projet. Le fait de glisser-déposer Ground_Mat du panneau **Projet** sur le GameObject Ground dans la **Hiérarchie** a changé la couleur du plan, ce qui signifie que toute modification apportée à Ground_Mat sera répercutée sur le GameObject Ground.

Si vous sélectionnez Ground dans la **hiérarchie**, vous verrez également que le composant **Ground_Mat (Material)** tout en bas est maintenant utilisé par le **nuanceur standard** pour rendre la propriété de couleur de l'avion :

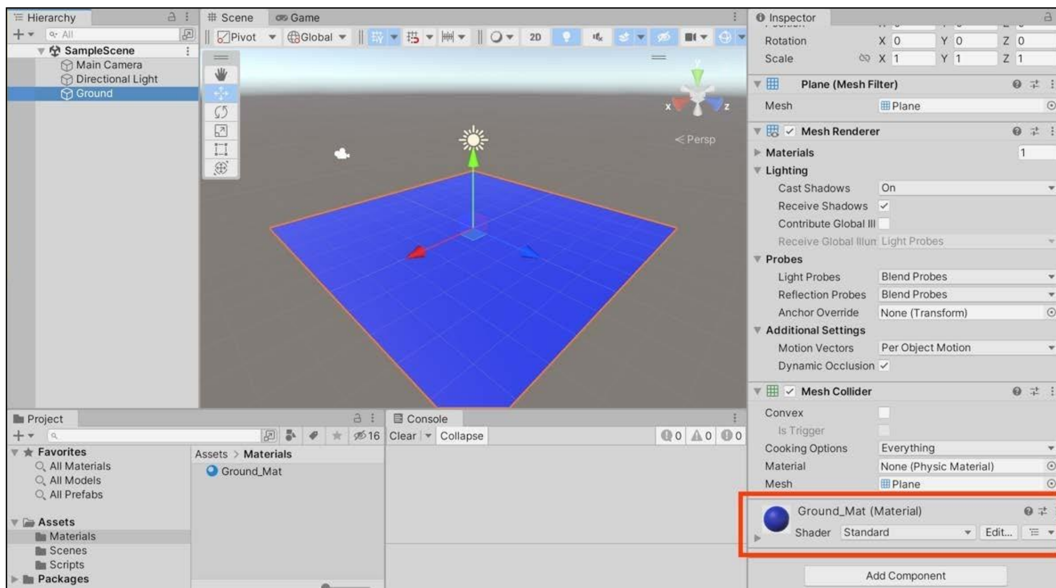


Figure 6.9 : Plan du sol avec le matériau coloré mis à jour

Le sol est notre toile ; cependant, dans l'espace 3D, il peut supporter d'autres objets 3D sur sa surface. Il vous appartiendra de le peupler d'obstacles amusants et intéressants pour vos futurs joueurs, ce que nous ferons dans la section suivante, lorsque nous en apprendrons un peu plus sur le white-boxing !

BOITE BLANCHE

Le white-boxing est un terme de conception qui désigne la présentation d'idées à l'aide d'espaces réservés, généralement dans l'intention de les remplacer par des éléments finis à une date ultérieure. Dans la conception de niveaux, la pratique du white-boxing consiste à bloquer un environnement avec des GameObjects primitifs pour avoir une idée de l'aspect que vous souhaitez lui donner. C'est une excellente façon de commencer les choses, en particulier pendant les phases de prototypage de votre jeu.

Avant de plonger dans Unity, j'aimerais commencer par un simple croquis de la disposition et de la position de base de mon niveau. Cela nous donne un peu de direction et nous aidera à mettre en place notre environnement plus rapidement.

Dans le dessin suivant, vous pourrez voir l'arène que j'ai en tête, avec une plateforme surélevée au milieu, accessible par des rampes, et des petites tourelles dans chaque coin :

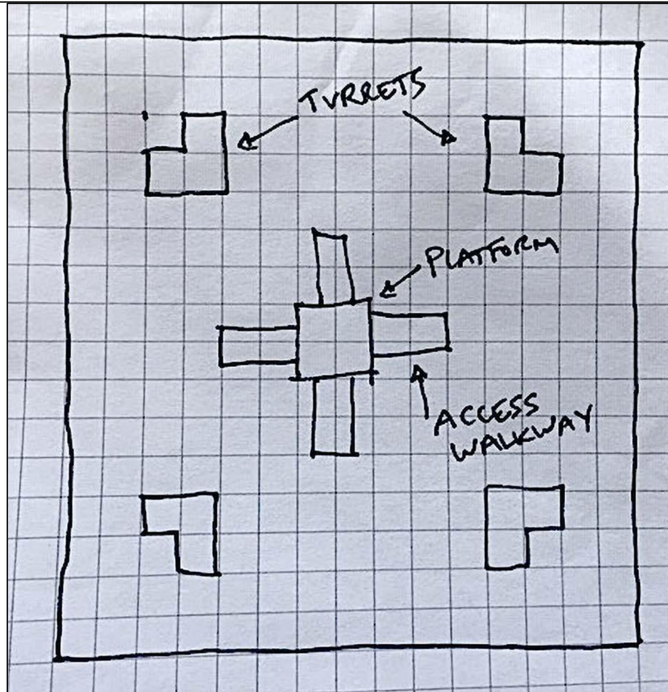


Figure 6.10 : Croquis de l'arène du niveau Hero Born

Ne vous inquiétez pas si vous n'êtes pas un artiste - moi non plus. L'important est de mettre vos idées sur papier afin de les consolider dans votre esprit et de régler les problèmes éventuels avant de vous lancer dans Unity.

Avant de vous lancer à corps perdu dans la production de cette esquisse, vous devrez vous familiariser avec quelques raccourcis de l'éditeur Unity afin de faciliter la création de boîtes blanches.

Outils d'édition

Lorsque nous avons abordé l'interface Unity dans *Chnuícu 1*, *Gcíiing ío Know Youu Enviuonmcní*, nous avons survolé certaines fonctionnalités de la barre d'outils, que nous devons revoir afin de savoir comment manipuler efficacement les GameObjects. Vous les trouverez dans le coin supérieur gauche de l'éditeur Unity :

Figure 6.11 : Barre d'outils de l'éditeur Unity

Décortiquons les différents outils disponibles dans la barre d'outils de la capture d'écran précédente :

1. **Vue** : Cette fonction vous permet d'effectuer un panoramique et de modifier votre position dans la scène en cliquant et en faisant glisser votre souris.
2. **Déplacer** : cette option permet de déplacer des objets le long des axes *x*, *;* et *z* en faisant glisser leurs flèches respectives.

3. **Rotation** : Cette fonction permet d'ajuster la rotation d'un objet en tournant ou en faisant glisser ses marqueurs respectifs.
4. **Échelle** : Cette fonction permet de modifier l'échelle d'un objet en le faisant glisser sur des axes spécifiques.
5. **Transformation de rectangle** : Cette fonction combine les fonctionnalités des outils de **déplacement**, de **rotation** et de **mise à l'échelle.**
6. **Transformer** : Cette fonction vous permet d'accéder à la position, à la rotation et à l'échelle d'un objet en une seule fois.

Vous trouverez de plus amples informations sur la navigation et le positionnement des objets de jeu dans le panneau **Scène** à l'adresse suivante : https://docs.unity3d.com/Manual/ PositioningGameObjects.html. Il convient également de noter que vous pouvez déplacer, positionner et mettre à l'échelle des objets à l'aide du composant **Transform**, comme nous l'avons vu plus haut dans la section

Il est possible d'effectuer des panoramiques et de naviguer dans la scène avec des outils similaires, mais pas à partir de l'éditeur Unity lui-même :

- Pour regarder autour de vous, maintenez le bouton droit de la souris enfoncé et faites-le glisser pour faire pivoter la caméra.
- Pour vous déplacer tout en utilisant l'appareil photo, maintenez le bouton droit de la souris enfoncé et utilisez les touches *W*, *"*, *S* et *D* pour vous déplacer respectivement vers l'avant, l'arrière, la gauche et la droite. Si vous êtes sur un Mac et que vous utilisez votre pavé tactile au lieu d'une souris, cliquez et maintenez deux doigts enfoncés tout en appuyant sur les touches *W*, *"*, *S* et *D.*
- Appuyez sur la touche *F* pour effectuer un zoom avant et vous concentrer sur un objet de jeu sélectionné dans la **hiérarchie.** panneau.

Ce type de navigation dans la scène est plus communément connu sous le nom de mode survol. Lorsque je vous demande de vous concentrer sur un objet ou un point de vue particulier ou de vous y rendre, utilisez une combinaison de ces fonctions.

La navigation dans la vue **Scène** peut être une tâche à part entière, mais tout se résume à une pratique répétée. Pour une liste plus détaillée des fonctions de navigation dans la scène, visitez :

Même si le plan du sol ne permet pas à notre personnage de tomber à travers, nous pouvons toujours marcher sur le bord à ce stade. Votre tâche consiste maintenant à murer l'arène de façon à ce que le joueur dispose d'une zone de locomotion confinée.

Procès du héros - pose de cloisons sèches

À l'aide des cubes primitifs et de la barre d'outils, placez quatre murs autour du niveau en utilisant les outils **Déplacement**, **Rotation** et **Échelle** pour délimiter l'arène principale :

1. Dans le panneau **Hiérarchie**, sélectionnez **+** > **Objet 3D** > **Cube** pour créer le premier mur et nommez-le
   Mur_01.
2. Réglez sa valeur d'échelle sur 30 pour l'axe x, 1,5 pour l'axe *;* et 0,2 pour l'axe *x.*

Ainsi, notre avion d'une longueur de 3 est de la même longueur que n'importe quel autre objet de longueur 30.

3. L'objet Mur_01 étant sélectionné dans le panneau **Hiérarchie**, passez à l'outil **Position** dans le coin supérieur gauche et utilisez les flèches rouge, verte et bleue pour positionner le mur au bord du plan de masse.
4. Répétez le *sícus 1L3* jusqu'à ce que vous ayez quatre murs entourant votre zone :

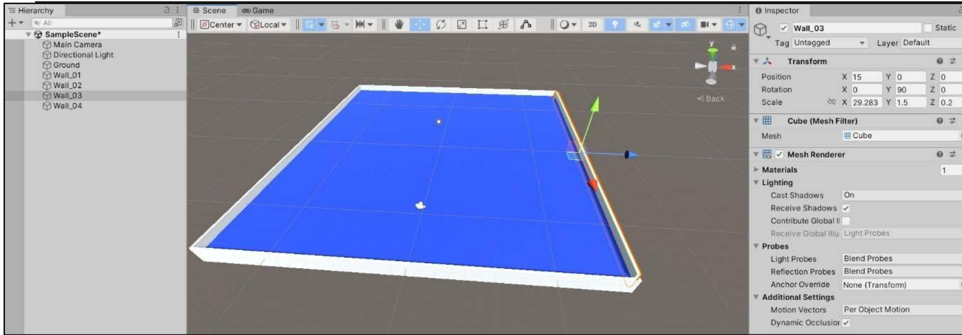


Figure 6.12 : Arène de plain-pied avec quatre murs et un plan de sol

À partir de ce chapitre, je donnerai des valeurs de base pour la position, la rotation et l'échelle des murs, mais n'hésitez pas à faire preuve d'audace et de créativité. Je souhaite que vous expérimentiez les outils de l'éditeur Unity afin d'être plus rapidement à l'aise.

C'était un peu de construction, mais l'arène commence à prendre forme ! Avant de passer à l'ajout d'obstacles et de plateformes, vous devez prendre l'habitude de nettoyer votre hiérarchie d'objets. Nous verrons comment cela fonctionne dans la section suivante.

Garder la hiérarchie propre

Normalement, ce genre de conseil serait placé dans un paragraphe à la fin d'une section, mais s'assurer que la hiérarchie de votre projet est aussi organisée que possible est si important qu'il a besoin de sa propre sous-section. Idéalement, vous voudrez que tous les GameObjects apparentés soient regroupés sous un seul **objet parent**. Pour l'instant, ce n'est pas un risque car nous n'avons que quelques objets dans la scène ; cependant, lorsque le nombre d'objets atteindra des centaines dans un projet de grande envergure, vous aurez du mal.

La façon la plus simple de garder votre hiérarchie propre est de stocker les objets apparentés dans un objet parent, comme vous le feriez avec des fichiers dans un dossier sur votre bureau. Notre niveau comporte quelques objets qui mériteraient d'être organisés, et Unity nous facilite la tâche en nous permettant de créer des GameObjects vides. Un objet vide est un conteneur (ou dossier) parfait pour contenir des groupes d'objets apparentés, car il n'est accompagné d'aucun composant - c'est une coquille.

Prenons notre plan de sol et nos quatre murs et regroupons-les sous un GameObject vide commun :

1. Sélectionnez **+** > **Créer un vide** dans le panneau **Hiérarchie** et nommez le nouvel objet Environnement.
2. Faites glisser et déposez le plan du sol et les quatre murs dans l'**environnement**, pour en faire des objets enfants.
3. Sélectionnez l'objet vide **Environnement** et vérifiez que ses positions **X**, **Y** et **Z** sont toutes réglées sur 0 :

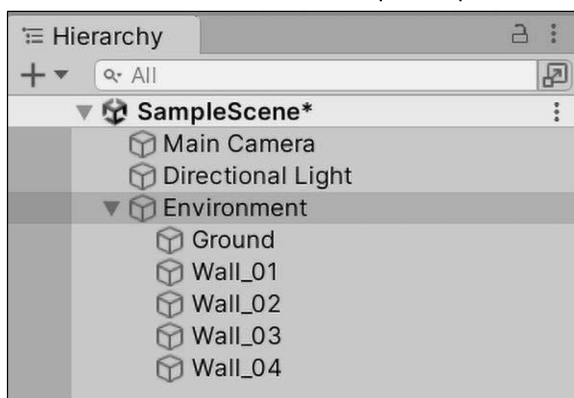


Figure 6.13 : Panneau de hiérarchie montrant le parent GameObject vide

L'environnement existe dans l'onglet **Hiérarchie** en tant qu'objet parent, avec les objets arène comme enfants. Nous pouvons maintenant développer ou fermer la liste déroulante de l'objet **Environnement** à l'aide de l'icône en forme de flèche, ce qui permet de désencombrer le panneau **Hiérarchie.**

Il est important de fixer les positions **X**, **Y** et **Z** de l'objet **Environnement** à 0, car les positions des objets enfants sont désormais relatives à la position du parent. Cela nous amène à une question intéressante : quels sont les points

d'origine de ces positions, rotations et échelles que nous définissons ? La réponse est qu'ils dépendent de l'espace relatif que nous utilisons, qui, dans Unity, est soit **World**, soit **Local** :

- **L'espace mondial** utilise un point d'origine défini dans la scène comme référence constante pour tous les objets du jeu. Dans Unity, ce point d'origine est (0, 0, 0), ou 0 sur les axes *x*, *;* et *x.*
- **L'espace local** utilise le composant Transform parent de l'objet comme origine, ce qui modifie essentiellement la perspective de la scène. Unity fixe également cette origine locale à (0, 0, 0). Imaginez que la transformation parente est le centre de l'univers et que tout le reste gravite autour d'elle.

Ces deux orientations sont utiles dans différentes situations, mais pour l'instant, la réinitialisation à cette Le point de contact permet de mettre tout le monde sur un pied d'égalité.

Travailler avec des préfabriqués

Les Prefabs sont l'un des composants les plus puissants que vous rencontrerez dans Unity. Ils sont très utiles non seulement dans la construction de niveaux, mais aussi dans l'écriture de scripts. Les Prefabs sont des GameObjects qui peuvent être sauvegardés et réutilisés avec tous les objets enfants, composants, scripts C# et paramètres de propriété intacts. Une fois créé, un Prefab est comme un modèle de classe ; chaque copie utilisée dans une scène est une instance distincte de ce Prefab. Par conséquent, toute modification apportée au Prefab de base modifiera également toutes les instances actives non modifiées de la scène.

L'arène semble un peu trop simple et complètement ouverte, ce qui en fait un endroit parfait pour tester la création et l'édition de Prefabs. Comme nous voulons quatre tourelles identiques dans chaque coin de l'arène, elles constituent un cas parfait pour un Prefab.

Encore une fois, je n'ai pas inclus de valeurs précises de position de barrière, de rotation ou d'échelle parce que je veux que vous vous approchiez au plus près des outils de l'éditeur Unity.

À l'avenir, lorsque vous verrez une tâche qui vous attend et qui ne comporte pas de position spécifique,

Créons les tourelles en suivant les étapes suivantes :

1. Créez un objet parent vide à l'intérieur de l'objet parent **Environnement** en sélectionnant **+** >. **Créer un vide** et le nommer Barrière.
2. Créez deux cubes en sélectionnant **+** > **Objet 3D** > **Cube** deux fois, puis positionnez-les et mettez-les à l'échelle pour obtenir une base en forme de V. Modifiez l'**échelle X** du premier cube à 3 et celle du second à 4 si vous n'êtes pas sûr de la taille qu'ils doivent avoir.
3. Créez deux autres primitives cube (il n'est pas nécessaire de les redimensionner) et placez-les sur les extrémités de la base de la tourelle :

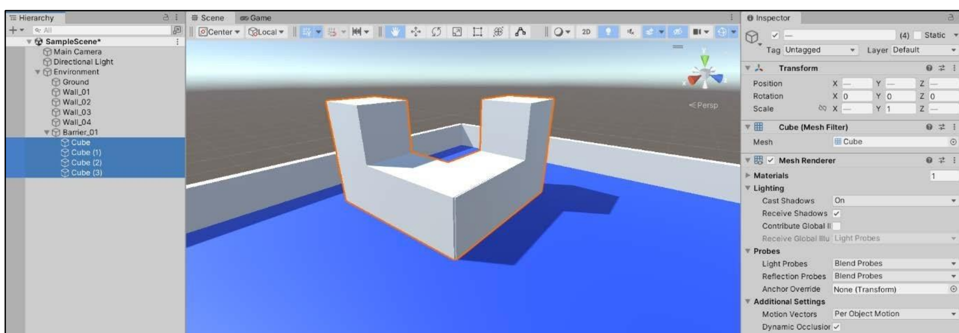
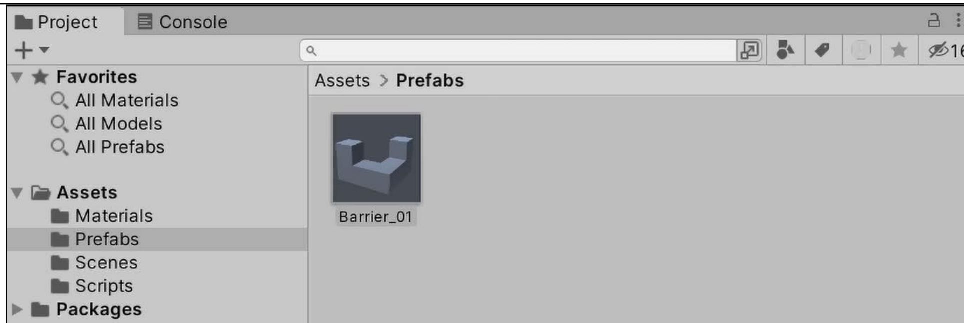


Figure 6.14 : Capture d'écran de la tourelle composée de cubes

4. Dans le dossier principal **Assets**, créez un nouveau dossier nommé Prefabs et faites glisser le fichier **Barrier** GameObject du panneau **Hiérarchie** vers le dossier **Prefabs de** la vue **Projet :**



Fiqure 6.15 : Barrière Prefab dans le dossier Prefabs

La barrière et tous ses objets enfants sont désormais des Prefabs, ce qui signifie que nous pouvons les réutiliser en faisant glisser des copies depuis le dossier Prefabs ou en dupliquant celui qui se trouve dans la scène. La **barrière** est devenue bleue dans l'onglet **Hiérarchie** pour indiquer son changement de statut, et a également ajouté une rangée de boutons de fonction **Prefab** dans l'onglet **Inspecteur** sous son nom :

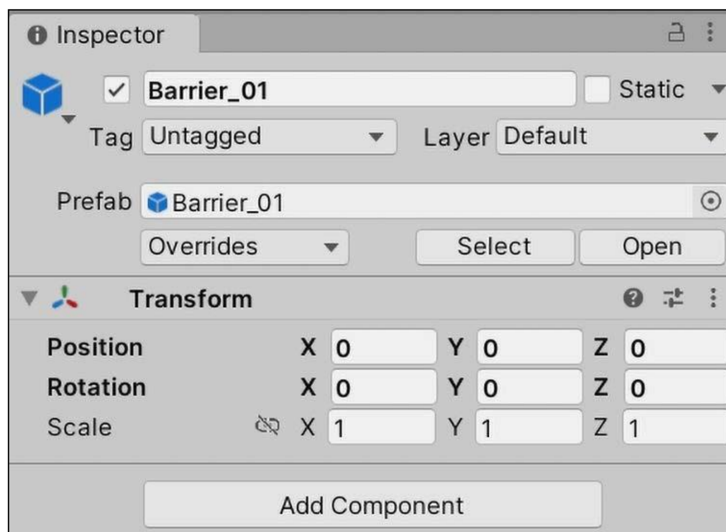


Figure 6.16 : Barrière_01 Prefab en surbrillance dans le panneau Inspecteur

Toute modification apportée à l'objet préfabriqué d'origine, **Barrière**, affectera désormais toutes les copies présentes dans la scène. Puisque nous

Nous avons besoin d'un cinquième cube pour compléter la barrière. Mettons à jour et sauvegardons le préfabriqué pour le voir à l'œuvre.

Maintenant, notre tourelle a un énorme espace au milieu, ce qui n'est pas idéal pour couvrir notre personnage, alors mettons à jour le **Barrier** Prefab en ajoutant un autre cube et en appliquant le changement :

1. Créez une primitive **Cube** et placez-la à l'intersection de la base de la tourelle.
2. La nouvelle primitive **Cube** sera marquée comme étant grise avec une petite icône **+** à côté de son nom dans l'écran d'accueil.

Onglet **Hiérarchie**. Cela signifie qu'il ne fait pas encore officiellement partie de la préfabrication :

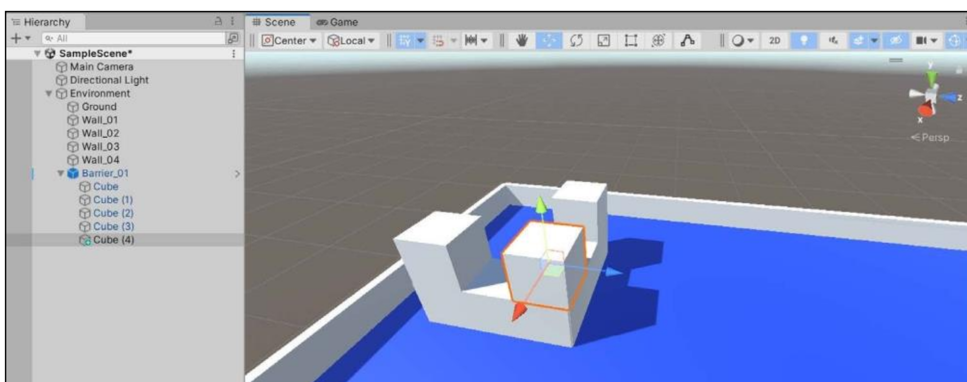


Figure 6.17 : Mise à jour d'un nouveau préfabriqué dans la fenêtre Hiérarchie

3. Faites un clic droit sur la nouvelle primitive **Cube** dans le panneau **Hiérarchie** et sélectionnez **Added GameObject (Objet de jeu ajouté**)

   > **Appliquer au préfabriqué "Barrière_01** " :

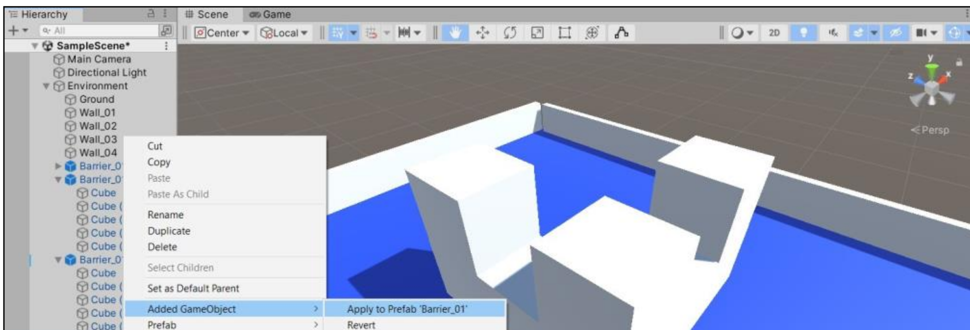


Figure 6.18 : Option permettant d'appliquer les modifications de la préfabrication à la préfabrication de base

Le **Barrier** Prefab est maintenant mis à jour pour inclure le nouveau cube, et toute la hiérarchie du Prefab devrait être à nouveau bleue. Vous avez maintenant un Prefab de tourelle qui ressemble à la capture d'écran précédente ou, si vous vous sentez d'humeur aventureuse, à quelque chose de plus créatif. Cependant, nous voulons qu'il y en ait dans tous les coins de l'arène. C'est à vous de les ajouter !

Maintenant que nous avons une barrière Prefab réutilisable, construisons le reste du niveau pour qu'il corresponde à l'esquisse que nous avions au début de la section :

1. Dupliquez le Prefab **Barrier** trois fois et placez chacun d'eux dans un coin différent de l'arène. Vous pouvez le faire en faisant glisser plusieurs objets **Barrière** du dossier **Prefabs** dans la scène, ou en faisant un clic droit sur **Barrière** dans la **hiérarchie** et en sélectionnant **Dupliquer**.
2. Créez un nouveau GameObject vide à l'intérieur de l'objet parent **Environnement**, nommez-le Plate-forme_élevée et réglez sa position sur les axes *x*, *;* et *x* à 0.
3. Créez un **Cube** en tant qu'objet enfant de Raised_Platform et mettez-le à l'échelle (*x* : 5, *;* : 2, *x* : 5) pour former une plate-forme comme le montre la *figure 6.19* cidessous.
4. Créez un **plan** et mettez-le à l'échelle pour obtenir une rampe (*x* : 10, *;* : 0.1, *x* : 5) :
   - Conseil : faites pivoter le plan autour de l'axe *x* pour créer un plan angulaire.
   - Ensuite, il faut le positionner de manière à ce qu'il relie la plate-forme au sol
5. Dupliquer l'objet rampe en utilisant *CmK* + *D* sur Mac, ou *Cíul* + *D* sur Windows. Répétez ensuite les étapes de rotation et de positionnement.
6. Répétez l'étape précédente deux fois de plus, jusqu'à ce que vous ayez quatre rampes au total menant à la plate-forme :

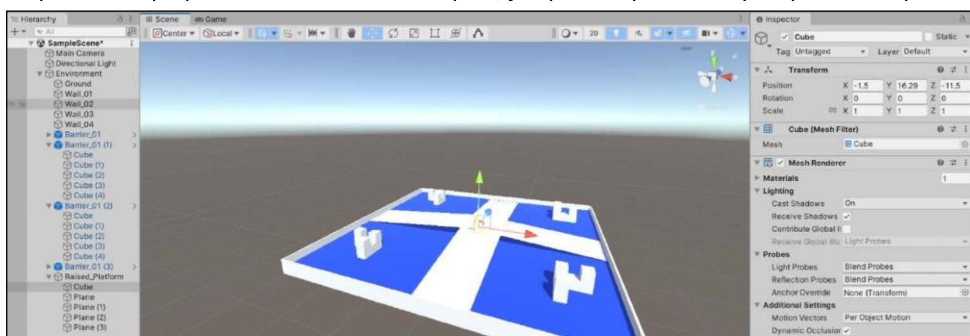


Figure 6.19 : Plate-forme surélevée GameObject parent

Vous avez maintenant réussi à créer votre premier niveau de jeu en boîte blanche ! Ne vous emballez pas trop vite, car nous ne faisons que commencer. Tous les bons jeux ont des objets que les joueurs peuvent ramasser ou avec lesquels ils peuvent interagir. Dans le défi suivant, vous devez créer un objet de santé et en faire un Prefab.

L'EPREUVE DU HEROS - CREER UN PICK-UP DE SANTE

Rassembler tout ce que nous avons appris jusqu'à présent dans ce chapitre peut vous prendre quelques minutes, mais le jeu en vaut la chandelle. Créez l'article de ramassage

comme suit :

1. Créez un objet de jeu **Capsule** en sélectionnant **+** > **Objet 3D** > **Capsule** et nommez-le Health_ Pickup.
2. Réglez l'échelle à 0,3 pour les axes *x, ;* et *x,* puis passez à l'outil **Déplacer** et positionnezle près de l'une de vos barrières.
3. Créer et attacher un nouveau **matériau de** couleur jaune à l'objet **Health_Pickup.**
4. Faites glisser l'objet **Health_Pickup** du panneau **Hiérarchie** vers le dossier **Prefab.**

La capture d'écran suivante donne un exemple de ce à quoi doit ressembler le produit fini :

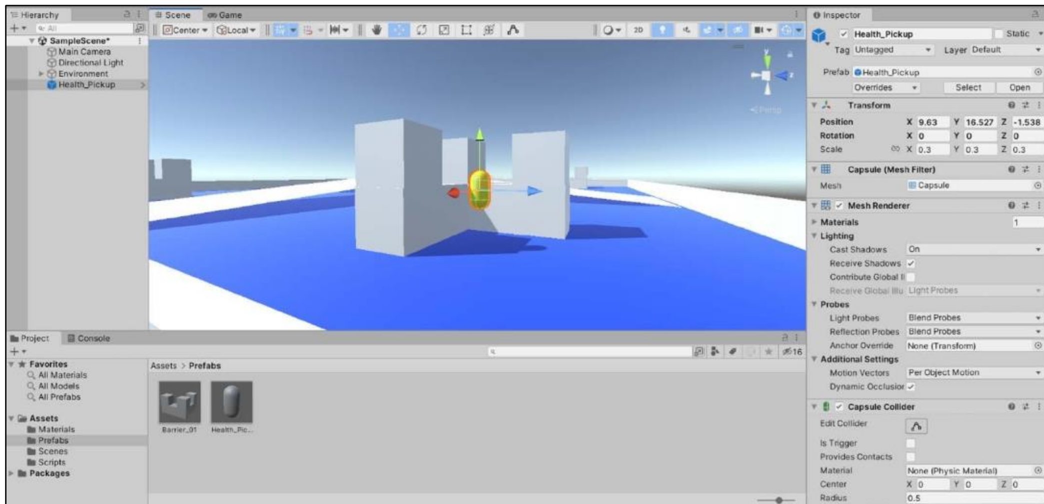


Figure 6.20 : Élément de ramassage et barrière Prefab dans la scène

Voilà qui conclut notre travail sur la conception et la mise en page des niveaux pour l'instant. Prochainement, vous allez suivre un cours accéléré sur l'éclairage avec Unity, et nous apprendrons à animer notre objet plus tard dans le chapitre.

LES BASES DE L'ECLAIRAGE

L'éclairage dans Unity est un vaste sujet, mais il peut être résumé en deux catégories : temps réel et précalculé. Les deux types de lumières prennent en compte des propriétés telles que la couleur et l'intensité de la lumière, ainsi que la direction vers laquelle elle est orientée dans la scène, qui peuvent toutes être configurées dans le panneau **Inspecteur.** La différence réside dans la manière dont le moteur Unity calcule l'action des lumières :

- **L'éclairage en temps réel** est calculé à chaque image, ce qui signifie que tout objet passant sur son chemin projettera des ombres réalistes et se comportera généralement comme une source de lumière réelle. Cependant, cela peut ralentir considérablement votre jeu et coûter une quantité exponentielle de puissance de calcul, en fonction du nombre de lumières dans votre scène.
- **L'éclairage précalculé**, quant à lui, stocke l'éclairage de la scène dans une texture appelée **lightmap**, qui est ensuite appliquée, ou cuite, dans la scène. Bien que cela permette d'économiser de la puissance de calcul, l'éclairage cuit est statique. Cela signifie qu'il ne réagit pas de manière réaliste et ne change pas lorsque les objets se déplacent dans la scène.

Il existe également un type d'éclairage mixte appelé **Precomputed Realtime Global Illumi- nation**, qui comble le fossé entre les processus en temps réel et précalculés. Il s'agit d'un sujet avancé spécifique à Unity, nous ne le couvrirons donc pas dans ce livre, mais n'hésitez pas à consulter la documentation à l'adresse suivante : https://docs.unity3d.com/Manual/GIIntro.html.

Voyons maintenant comment créer des objets lumineux dans la scène Unity elle-même.

CREER DES LUMIERES

Par défaut, chaque scène est livrée avec un composant de lumière directionnelle qui sert de source principale d'éclairage, mais les lumières peuvent être créées dans la hiérarchie comme n'importe quel autre GameObject. Même si l'idée de contrôler des sources de lumière vous est peut-être inconnue, il s'agit d'objets dans Unity, ce qui signifie qu'ils peuvent être positionnés, mis à l'échelle et tournés pour répondre à vos besoins :

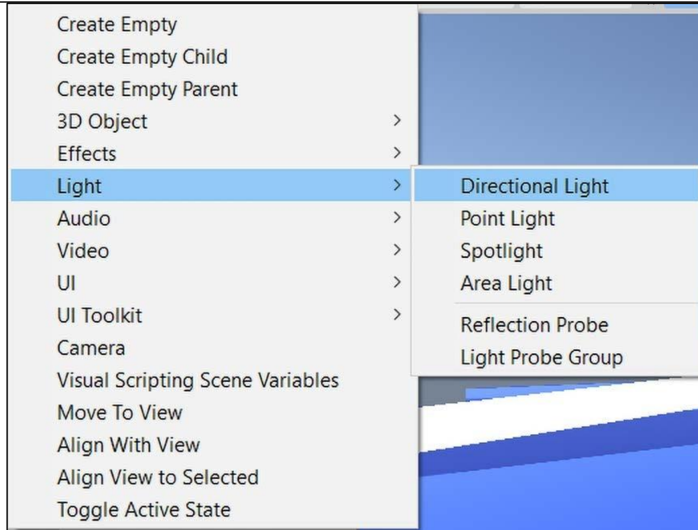


Figure 6.21 : Option du menu de création d'éclairage

Examinons quelques exemples d'objets lumineux en temps réel et leurs performances :

- **Les lumières directionnelles** sont idéales pour simuler la lumière naturelle, comme le soleil. Elles n'ont pas de position réelle dans la scène, mais leur lumière frappe tout comme si elle était toujours dirigée dans la même direction.
- **Les lumières ponctuelles** sont essentiellement des globes flottants qui envoient des rayons lumineux dans toutes les directions à partir d'un point central. Elles ont des positions et des intensités définies dans la scène.
- **Les projecteurs** envoient de la lumière dans une direction donnée, mais ils sont verrouillés par leur angle et concentrés sur une zone spécifique de la scène. Dans le monde réel, ce sont des projecteurs.
- **Les éclairages de surface** ont la forme d'un rectangle et émettent la lumière de leur surface à partir d'un seul côté du rectangle.

Les **sondes de réflexion** et les **groupes de sondes lumineuses** ne sont pas nécessaires pour *Hcuo Doun* ; cependant, si vous êtes intéressé, vous pouvez en savoir plus à l'adresse suivante : https://docs.unity3d.com/ Manual/ReflectionProbes.html et https://docs.unity3d.com/Manual/ LightProbes.html.

Comme tous les GameObjects dans Unity, les lumières ont des propriétés qui peuvent être ajustées pour donner à une scène un aspect spécifique. l'ambiance ou le thème.

PROPRIETES DES COMPOSANTS LEGERS

La capture d'écran suivante montre le composant **Light** sur la lumière directionnelle de notre scène. Toutes ces propriétés peuvent être configurées pour créer des environnements immersifs, mais les propriétés de base que nous devons connaître sont la **couleur**, le **mode** et l'**intensité**. Ces propriétés régissent la teinte de la lumière, les effets en temps réel ou calculés et la force générale :

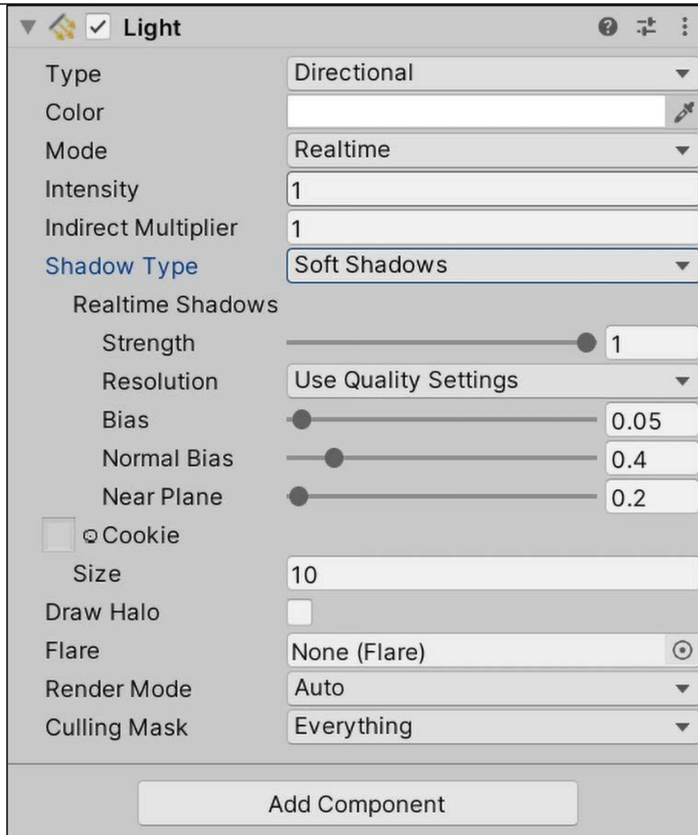


Figure 6.22 : Composant lumière dans la fenêtre de l'inspecteur

Comme pour les autres composants Unity, ces propriétés sont accessibles par le biais de scripts et de la classe Light, qui peut être consultée à l'adresse suivante : https://docs.unity3d.com/ ScriptReference/Light.html.

Essayez par vous-même en sélectionnant **+** | **Lumière** | **Lumière ponctuelle** et voyez comment cela affecte l'éclairage de la zone. Après avoir joué avec les paramètres, supprimez la lumière ponctuelle en cliquant dessus avec le bouton droit de la souris dans le panneau **Hiérarchie** et en choisissant **Supprimer**.

Maintenant que nous en savons un peu plus sur l'éclairage d'une scène de jeu, intéressonsnous à l'ajout d'animations !

ANIMER DANS UNITY

L'animation d'objets dans Unity peut aller d'un simple effet de rotation à des mouvements et actions complexes de personnages. Vous pouvez créer des animations dans le code ou à l'aide des fenêtres Animation et Animator :

- La fenêtre **Animation** est l'endroit où les segments d'animation, appelés clips, sont créés et gérés à l'aide d'une ligne de temps. Les propriétés des objets sont enregistrées le long de cette ligne de temps et sont ensuite lues pour créer un effet d'animation.
- La fenêtre **Animator** gère ces clips et leurs transitions à l'aide d'objets appelés contrôleurs d'animation.

Vous trouverez plus d'informations sur la fenêtre **Animator** et ses contrôleurs à l'adresse suivante : https://docs.unity3d.com/Manual/AnimatorControllers.html.

En créant et en manipulant vos objets cibles dans des clips, vous ferez bouger votre jeu en un rien de temps. Pour notre petit voyage dans les animations Unity, nous allons créer le même effet de rotation en code, et en utilisant l'Animator.

CREER DES ANIMATIONS EN CODE

Pour commencer, nous allons créer une animation dans le code pour faire pivoter notre objet de santé. Comme tous les GameObjects ont un composant Transform, nous pouvons saisir le composant Transform de notre objet et le faire pivoter indéfiniment.

Pour créer une animation en code, vous devez suivre les étapes suivantes :

1. Créez un nouveau script dans le dossier Scripts, nommez-le ItemRotation et ouvrez-le dans Visual Studio Code.
2. En haut du nouveau script et à l'intérieur de la classe, ajoutez une variable int publique contenant la valeur 100 appelée RotationSpeed, et une variable Transform privée appelée itemTransform. Si vous ne spécifiez pas de niveau d'accès, Visual Studio suppose qu'il s'agit d'une variable privée, mais la meilleure pratique consiste à utiliser

```
public int RotationSpeed = 100 ; private Transform itemTransform ;
```

   des modificateurs d'accès explicites pour s'assurer que votre code est clair comme de l'eau de roche :
3. Dans le corps de la méthode Start(), saisissez le composant Transform du GameObject et affectez-le à itemTransform :

```
itemTransform = this.GetComponent<Transform>() ;
```

4. Dans le corps de la méthode Update(), appelez itemTransform.Rotate. Cette méthode de la classe Transform prend en compte trois axes, un pour les rotations *x, ;* et *x* que vous souhaitez exécuter. Comme nous voulons que l'élément pivote d'un bout à l'autre, nous utiliserons l'axe *x* et laisserons les autres axes à 0 :

```
itemTransform.Rotate(RotationSpeed * Time.deltaTime, 0, 0) ;
```

Vous remarquerez que nous multiplions notre RotationSpeed par quelque chose appelé Time.deltaTime. C'est la façon standard de normaliser les effets de mouvement dans Unity afin qu'ils soient fluides quelle que soit la vitesse d'exécution de l'ordinateur du joueur. En général, vous devriez toujours multiplier vos vitesses de mouvement ou de rotation par

Time.deltaTime.

5. De retour dans Unity, sélectionnez l'objet Health_Pickup dans le dossier Prefabs du panneau **Projects** et descendez jusqu'au bas de la fenêtre de l'**inspecteur.** Cliquez sur **Add Component (Ajouter un composant)**, recherchez le script ItemRotation, puis appuyez sur *Enícu* :

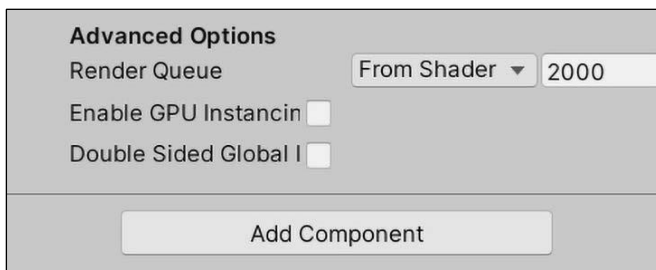


Figure 6.23 : Bouton Ajouter un composant dans le panneau Inspecteur

6. Maintenant que notre Prefab est mis à jour, déplacez la **caméra principale de** façon à voir l'objet Health_ Pickup et cliquez sur **Play** !

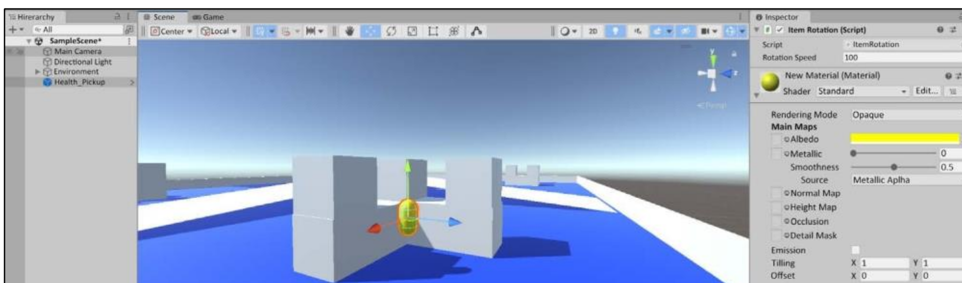


Figure 6.24 : Capture d'écran de la caméra focalisée sur l'élément de santé

Comme vous pouvez le voir, le ramasseur de santé tourne maintenant autour de son axe *x* dans une animation continue et fluide ! Maintenant que vous avez animé l'objet en code, nous allons dupliquer notre animation en utilisant le système d'animation intégré à Unity.

CREER DES ANIMATIONS DANS LA FENETRE UNITY ANIMATION

Avant d'aller plus loin, il est important de choisir la méthode de code **OU le** système d'animation d'Unity pour *n singlc nnimníion - et* non les deux. Sinon, ces deux systèmes finiront par se battre l'un contre l'autre dans votre projet.

Tout GameObject auquel vous souhaitez appliquer un clip d'animation doit être attaché à un composant **Animator** avec un **contrôleur d'animation** défini. S'il n'y a pas de contrôleur dans le projet lorsqu'un nouveau clip est créé, Unity en créera un et l'enregistrera dans le panneau **Projet,** que vous pourrez ensuite utiliser pour gérer vos clips. Votre prochain défi consiste à créer un nouveau clip d'animation pour l'article de ramassage.

Nous allons commencer à animer le Health_Pickup Prefab en créant un nouveau clip d'animation, qui fera tourner l'objet dans une boucle infinie. Pour créer un nouveau clip d'animation, nous devons suivre les étapes suivantes :

1. Naviguez vers **Fenêtre** > **Animation** > **Animation** pour ouvrir le panneau **Animation** et faites glisser l'onglet **Animation** à côté de la **console**.
2. Assurez-vous que l'élément Health_Pickup est sélectionné dans la **hiérarchie**, puis cliquez sur **Create** dans le panneau **Animation :**

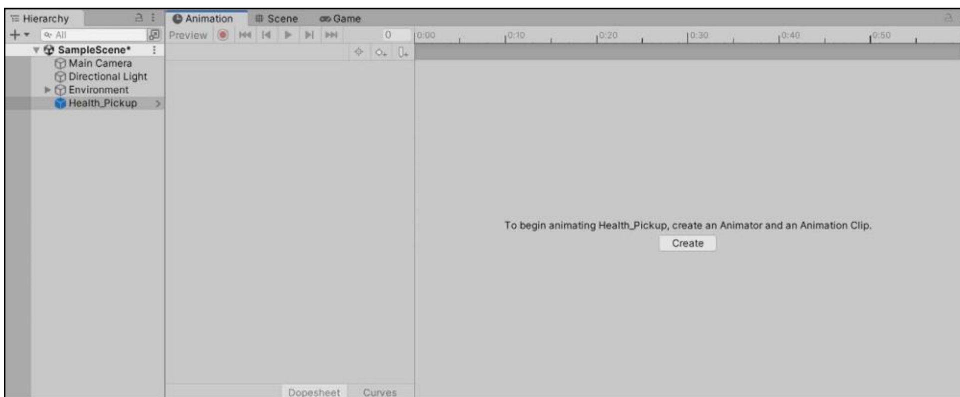


Figure 6.25 : Capture d'écran de la fenêtre Unity Animation

3. Créez un nouveau dossier sous **Assets dans** la liste déroulante suivante, nommez-le Animations, puis nommez le nouveau clip Pickup_Spin :

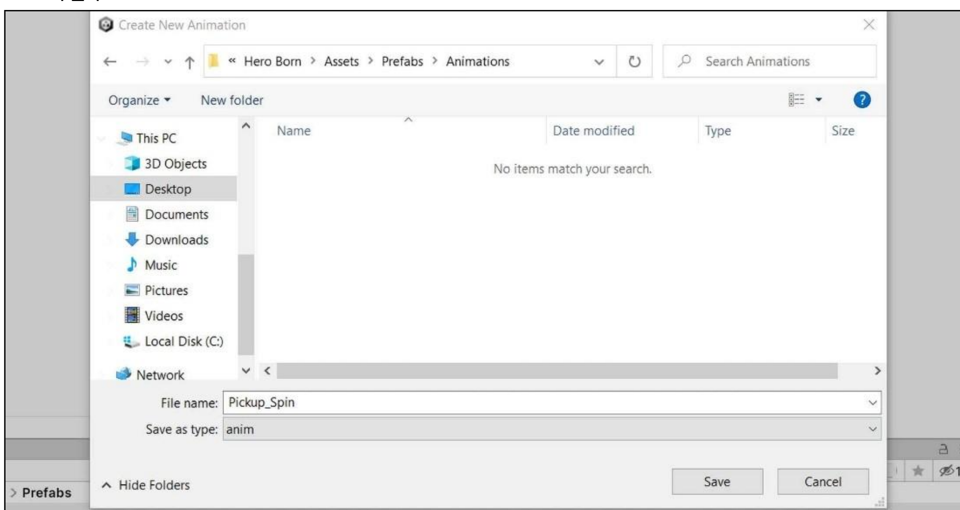


Figure 6.26 : Capture d'écran de la fenêtre Créer une nouvelle animation

4. Assurez-vous que le nouveau clip apparaît dans le panneau **Animation :**

Figure 6.27 : Capture d'écran de la fenêtre Animation avec un clip sélectionné

5. Comme nous n'avions pas de contrôleur **Animator**, Unity en a créé un pour nous dans le dossier Animation appelé **Health_Pickup**. Lorsque **Health_Pickup** est sélectionné, notez dans le panneau **Inspector** que lorsque nous avons créé le clip, un composant **Animator** a également été ajouté au Prefab pour nous, mais qu'il n'a pas encore été officiellement sauvegardé dans le Prefab avec le contrôleur **Health_Pickup** défini.
6. Notez que l'icône **+** apparaît en haut à gauche du composant **Animator**, ce qui signifie qu'il ne fait pas encore partie du préfabriqué **Health_Pickup :**

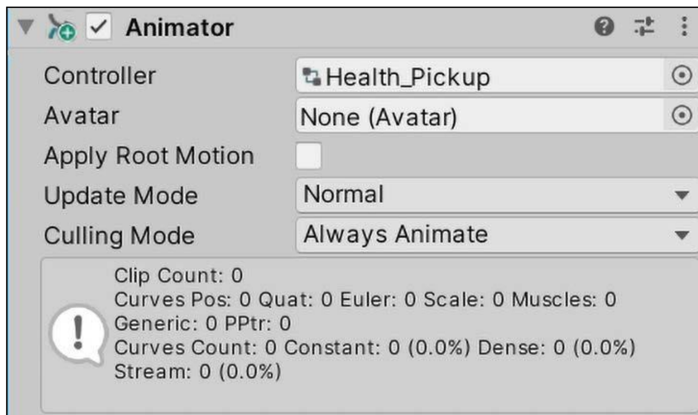


Figure 6.28 : Composant Animator dans le panneau Inspecteur

7. Sélectionnez l'icône à trois points verticaux en haut à droite et choisissez **Added Component** > **Apply to Prefab 'Health_Pickup'** (**Composant ajouté** > **Appliquer au préfabriqué 'Health_Pickup'**) :

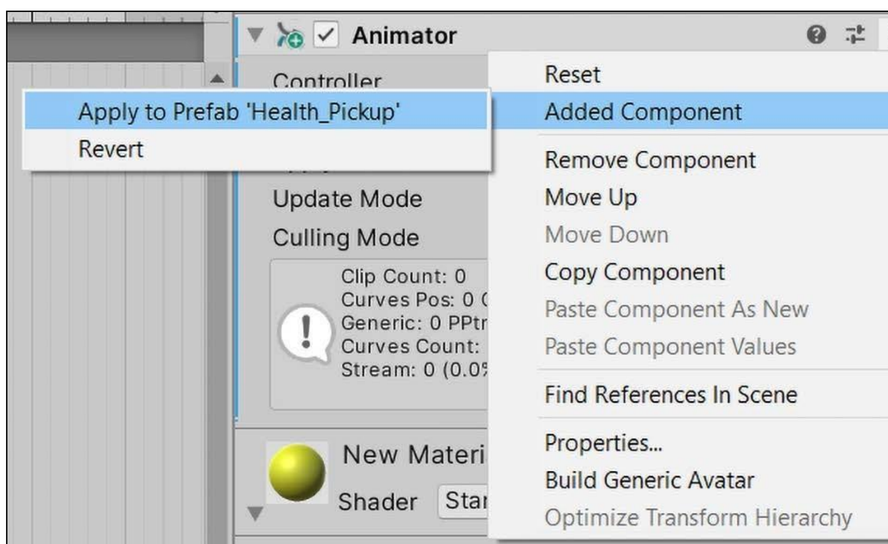


Figure 6.29 : Capture d'écran d'un nouveau composant appliqué au préfabriqué

Maintenant que vous avez créé et ajouté un composant **Animator** à la préfabrication **Health_Pickup**, il est temps de commencer à enregistrer des images d'animation. Lorsque vous pensez à des clips d'animation, comme dans les films, vous pensez à des images. Au fur et à mesure que le clip se déplace dans ses images, l'animation progresse, ce qui donne un effet de mouvement. C'est la même chose dans Unity ; nous devons enregistrer notre objet cible dans différentes positions à travers différentes images pour qu'Unity puisse jouer le clip.

ENREGISTREMENT D'IMAGES CLES

Maintenant que nous avons un clip avec lequel travailler, vous verrez une ligne de temps vierge dans la fenêtre **Animation.** Essentiellement, lorsque nous modifions la rotation *x de* notre **Health_Pickup** Prefab, ou toute autre propriété pouvant être animée, la ligne de temps enregistre ces changements sous forme d'images clés. Unity assemble ensuite ces images clés en une animation complète, de la même manière que les images individuelles d'un film analogique s'assemblent pour former une image animée.

Regardez la capture d'écran suivante et souvenez-vous de l'emplacement du bouton **Enregistrer** et de la ligne de temps :

Figure 6.30 : Capture d'écran de la fenêtre Animation et de la ligne de temps des images clés

Maintenant, faisons tourner notre objet. Pour l'animation de rotation, nous voulons que le **Health_Pickup** Prefab effectue une rotation complète de 360 degrés sur son axe *x* toutes les secondes, ce qui peut être fait en définissant trois images clés et en laissant Unity s'occuper du reste :

1. Sélectionnez l'objet **Health_Pickup** dans la fenêtre **Hiérarchie**, choisissez **Ajouter une propriété** > **Transformer**, puis cliquez sur le signe **+** en regard de **Rotation** :

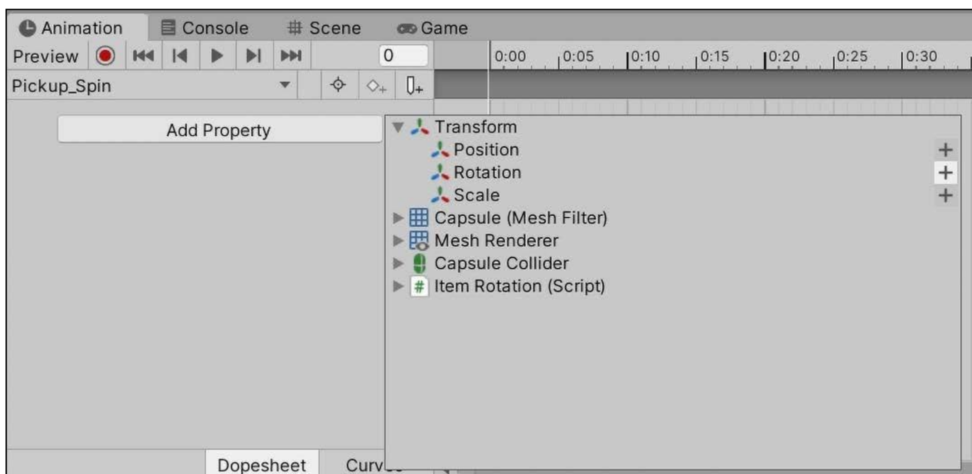


Figure 6.31 : Capture d'écran de l'ajout d'une propriété Transform pour l'animation

2. Cliquez sur le bouton **Enregistrer** pour lancer l'animation :
   - Placez votre curseur à 0:00 sur la ligne de temps, mais laissez la rotation *x* du Health_Pickup Prefab à 0 dans l'**inspecteur** - **les** champs modifiables apparaîtront en rouge afin que vous n'animiez pas une propriété par erreur :

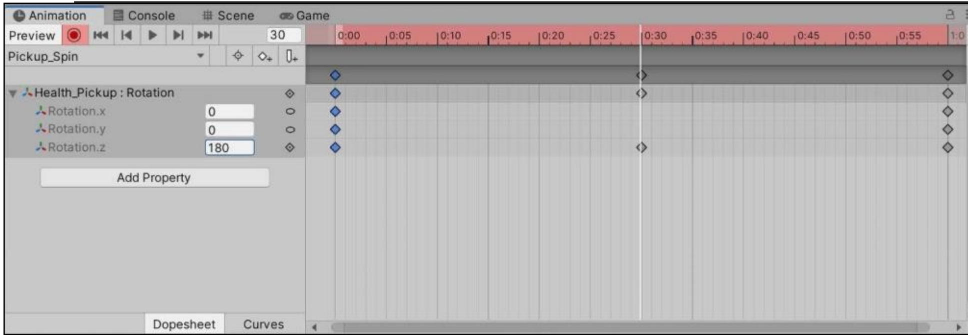


Figure 6.32 : Capture d'écran de la propriété animable dans l'inspecteur

- Placez votre curseur à 0:30 sur la ligne de temps et réglez la rotation *x* sur 180.
- Placez votre curseur à 1:00 sur la ligne de temps et réglez la rotation *x* sur 360 :

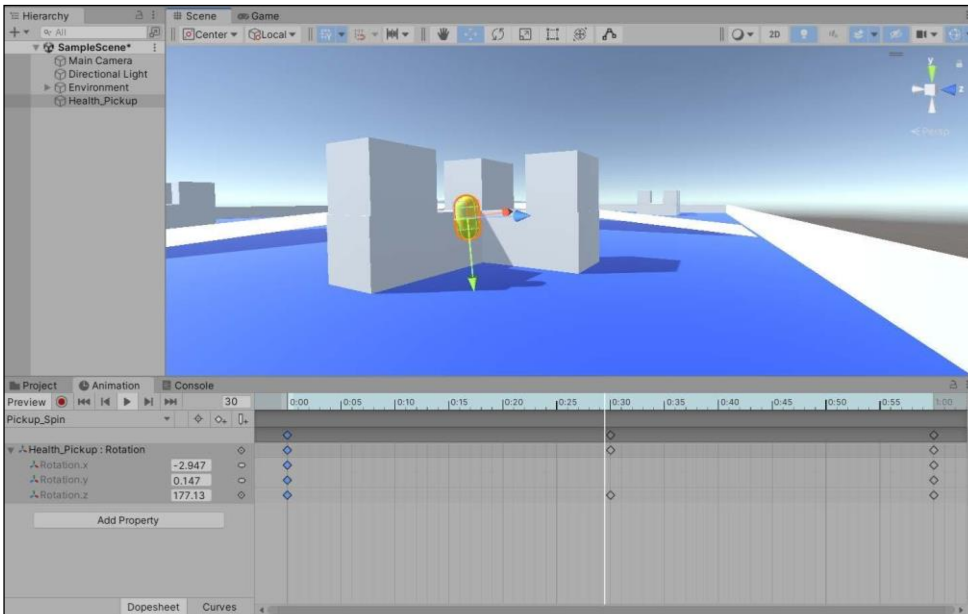


Figure 6.33 : Capture d'écran de l'enregistrement des images clés de l'animation

3. Cliquez sur le bouton **Enregistrer** pour terminer l'animation.
4. Cliquez sur le bouton "**Play"** à droite du bouton "**Record**" pour voir l'animation se dérouler en boucle.

Vous remarquerez que notre animation **Animator** remplace celle que nous avons écrite en code plus tôt. Ne vous inquiétez pas, il s'agit d'un comportement attendu. Vous pouvez cliquer sur la petite case à droite de n'importe quel composant dans le panneau de l'**inspecteur** pour l'activer ou le désactiver. Si vous désactivez le composant **Animator**, **Health_Pickup** tournera à nouveau autour de l'axe *x à* l'aide de notre code.

L'objet **Health_Pickup** pivote désormais sur l'axe *x* entre 0, 180 et 360 degrés toutes les secondes, créant ainsi une animation de rotation en boucle. Si vous jouez au jeu maintenant, l'animation tournera indéfiniment jusqu'à ce que le jeu soit arrêté :

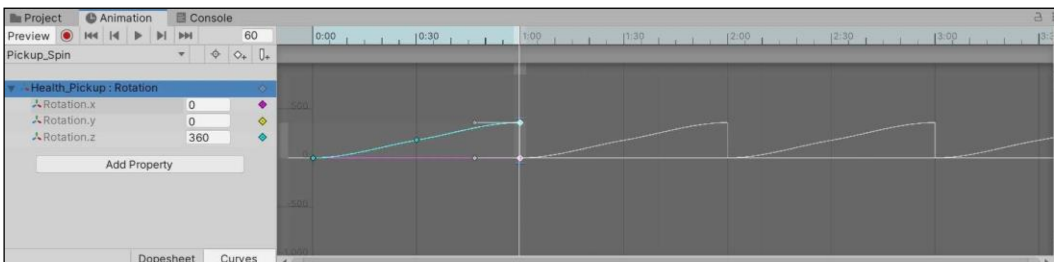


Figure 6.34 : Capture d'écran d'une animation jouée dans la fenêtre Animation

Toutes les animations ont des courbes, qui déterminent les propriétés spécifiques de l'exécution d'une animation. Nous ne ferons pas grand-chose avec ces courbes, mais il est important d'en comprendre les bases. Nous les aborderons dans la section suivante.

COURBES ET TANGENTES

En plus d'animer une propriété d'objet, Unity nous permet de gérer la façon dont l'animation se déroule dans le temps grâce aux courbes d'animation. Jusqu'à présent, nous avons utilisé le mode **Dopesheet,** que vous pouvez modifier en bas de la fenêtre **Animation.** Si vous cliquez sur la vue **Curves** (illustrée dans la capture d'écran suivante), vous verrez un graphique différent avec des points d'accentuation à la place des images clés enregistrées.

Nous voulons que l'animation de la rotation soit fluide - ce que nous appelons linéaire - et nous laisserons donc tout tel quel. Cependant, il est possible d'accélérer, de ralentir ou de modifier l'animation à tout moment en faisant glisser ou en ajustant les points du graphique de la courbe dans n'importe quelle direction :

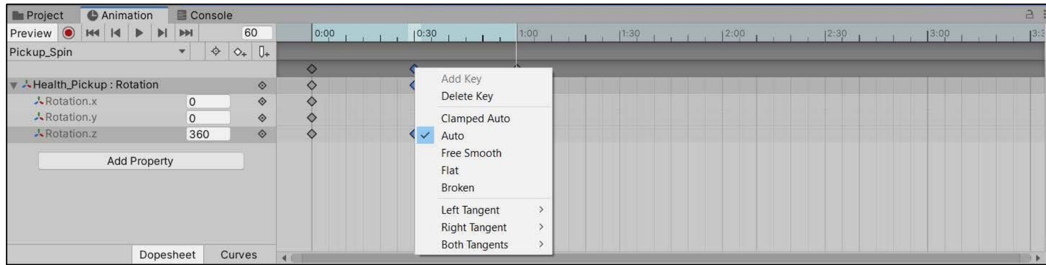


Figure 6.35 : Capture d'écran de la ligne de temps Courbes dans la fenêtre Animation

Les courbes d'animation gérant l'évolution des propriétés dans le temps, il nous faut encore trouver un moyen de corriger le bégaiement qui se produit chaque fois que l'animation **Health_Pickup** se répète. Pour ce faire, nous devons modifier la tangente de l'animation, qui gère la façon dont les images clés se fondent les unes dans les autres.

Ces options sont accessibles en cliquant avec le bouton droit de la souris sur n'importe quelle image clé de la ligne de temps dans le **Dopesheet**

que vous pouvez voir ici :

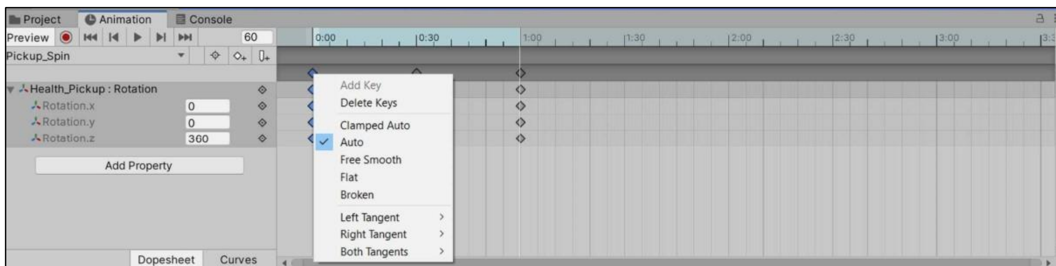


Figure 6.36 : Capture d'écran des options de lissage des images clés

Les courbes et les tangentes sont des fonctions intermédiaires/avancées, c'est pourquoi nous ne nous y attarderons pas trop. Si cela vous intéresse, vous pouvez consulter la documentation sur les courbes d'animation et les options de tangente à l'adresse suivante : https://docs.unity3d.com/Manual/ animeditor-AnimationCurves.html.

Si vous jouez l'animation de rotation telle qu'elle est actuellement, il y a une légère pause entre le moment où l'objet termine sa rotation complète et celui où il en commence une nouvelle. Votre tâche consiste à lisser cette pause, ce qui est l'objet du prochain défi.

Ajustons les tangentes sur la première et la dernière image de l'animation de manière à ce que la rotation de l'image se fasse en douceur.

L'animation s'intègre parfaitement lorsqu'elle se répète :

1. Cliquez avec le bouton droit de la souris sur les icônes en forme de diamant de la première et de la dernière image clé sur la ligne de temps de l'animation et

sélectionner **Auto** :

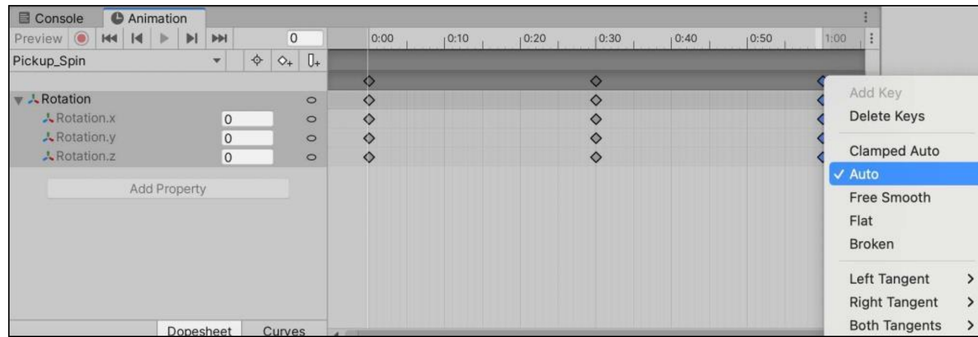


Figure 6.37 : Modification des options de lissage de l'image clé

2. Si ce n'est pas déjà fait, déplacez la **caméra principale de** façon à voir l'objet Health_ Pickup, puis cliquez sur **Play :**

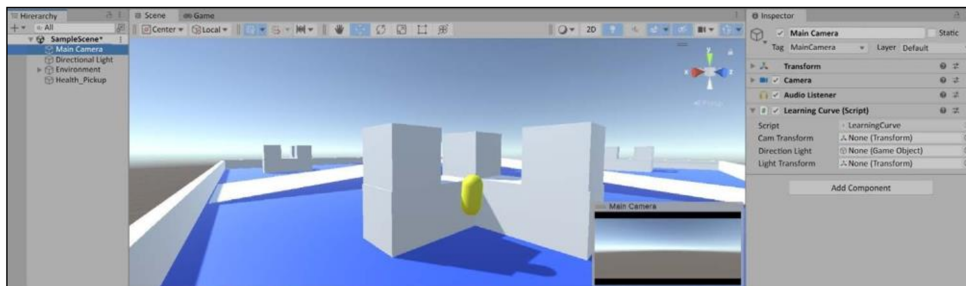


Figure 6.38 : Capture d'écran de la lecture de l'animation finale lissée

En réglant les tangentes de la première et de la dernière image clé sur **Auto**, vous demandez à Unity de rendre leurs transitions fluides, ce qui élimine le mouvement saccadé d'arrêt/redémarrage lorsque l'animation tourne en boucle.

C'est toute l'animation dont vous aurez besoin pour ce livre, mais je vous encourage à consulter la boîte à outils complète qu'offre Unity dans ce domaine. Vos jeux seront plus attrayants et vos joueurs vous remercieront !

RESUME

Nous sommes arrivés à la fin d'un autre chapitre qui comportait beaucoup d'éléments mobiles, ce qui a pu être beaucoup pour ceux d'entre vous qui sont nouveaux dans Unity.

Même si ce livre se concentre sur le langage C# et son implémentation dans Unity, nous devons prendre le temps d'avoir une vue d'ensemble du développement de jeux, de la documentation et des fonctionnalités du moteur qui ne sont pas des scripts. Bien que nous n'ayons pas eu le temps de couvrir en profondeur l'éclairage et l'animation, cela vaut la peine de les connaître si vous envisagez de continuer à créer des projets Unity.

Dans le prochain chapitre, nous reviendrons sur la programmation des mécanismes de base de *Hcuo Doun*, en commençant par la mise en place d'un objet joueur mobile, le contrôle de la caméra et la compréhension de la manière dont le système physique d'Unity régit le monde du jeu.

PETIT QUIZ SUR LES FONCTIONNALITES DE BASE D'UNITY

1. Les cubes, les capsules et les sphères sont des exemples de quel type de GameObject ?
2. Quel axe Unity utilise-t-il pour représenter la profondeur, qui donne aux scènes leur aspect 3D ?
3. Comment transformer un GameObject en un Prefab réutilisable ?
4. Quelle unité de mesure le système d'animation Unity utilise-t-il pour enregistrer les animations d'objets ?

N'oubliez pas de comparer vos réponses aux miennes dans l'annexe *Fou Quix "nswcus"* pour voir où vous en êtes !

Rejoignez-nous sur discord !

Lisez ce livre en compagnie d'autres utilisateurs, d'experts en développement de jeux Unity et de l'auteur luimême.

Posez des questions, apportez des solutions aux autres lecteurs, discutez avec l'auteur via des sessions "Ask Me Anything" et bien plus encore. Ask Me Anything et bien plus encore.

Scannez le code QR ou visitez le lien pour rejoindre la communauté.

https://packt.link/csharpwithunity

7

Mouvement, contrôle de la caméra, et collisions

L'une des premières choses que fait un joueur lorsqu'il commence un nouveau jeu est d'essayer les mouvements du personnage (si, bien sûr, le jeu a un personnage mobile) et les commandes de la caméra. C'est non seulement excitant, mais cela permet au joueur de savoir à quel type de jeu il peut s'attendre. Dans *Hcuo Doun, le* personnage sera un objet capsule que l'on pourra déplacer et faire pivoter à l'aide des touches *W*, *"*, *S*, *D* et des flèches, respectivement.

Nous commencerons par apprendre à manipuler le composant Transform de l'objet joueur, puis nous reproduirons le même schéma de contrôle du joueur en appliquant une force. Cela permet d'obtenir un effet de mouvement plus réaliste. Lorsque nous déplaçons le joueur, la caméra le suit à partir d'une position située légèrement derrière et au-dessus du joueur, ce qui facilite la visée lorsque nous mettons en œuvre le mécanisme de tir. Enfin, nous explorerons la façon dont les collisions et les interactions physiques sont gérées par le système physique d'Unity en travaillant avec notre Prefab de ramassage d'objets.

Tout cela sera mis en place à un niveau jouable, bien qu'il n'y ait pas encore de mécanisme de tir. Il nous donnera également un premier aperçu de l'utilisation de ₡ pour programmer des fonctions de jeu en reliant les sujets suivants :

- Gestion des déplacements des joueurs
- Déplacement du lecteur à l'aide du composant Transform
- Script du comportement de la caméra
- Travailler avec le système physique d'Unity

GESTION DES DEPLACEMENTS DES JOUEURS

Lorsque vous décidez de la meilleure façon de déplacer votre personnage dans votre monde virtuel, réfléchissez à ce qui sera le plus réaliste et à ce qui n'alourdira pas votre jeu avec des calculs coûteux. Dans la plupart des cas, il s'agit d'un compromis, et Unity n'échappe pas à la règle.

Les trois façons les plus courantes de déplacer un objet de jeu et leurs résultats sont les suivants :

- **Option A** : utiliser le composant Transform d'un GameObject pour le mouvement et la rotation. Il s'agit de la solution la plus simple et celle avec laquelle nous travaillerons en premier.
- **Option B** : utiliser la physique du monde réel en attachant un composant **Rigidbody** à un objet de jeu et en appliquant une force dans le code. Les composants Rigidbody ajoutent une physique réelle simulée à tout objet de jeu auquel ils sont attachés. Cette solution s'appuie sur le système physique d'Unity pour faire le gros du travail, ce qui permet d'obtenir un effet beaucoup plus réaliste. Nous mettrons à jour notre code pour utiliser cette approche plus loin dans ce chapitre afin de nous familiariser avec les deux méthodes.

Unity suggère de s'en tenir à une approche cohérente lors du déplacement ou de la rotation d'un objet de jeu ; manipulez soit le composant Transform soit le composant Rigidbody d'un objet, mais jamais les deux en même temps.

- **Option C** : Attacher un composant Unity prêt à l'emploi ou un Prefab, tel que le **Character Controller** ou le **First-Person Controller**. Cela permet d'éliminer le code standard et d'obtenir un effet réaliste tout en accélérant le temps de prototypage.

Nous ne travaillerons pas avec l'option C, mais il y a des tonnes d'actifs dans l'Asset Store et dans la communauté Unity que vous pouvez explorer !

Vous trouverez de plus amples informations sur le composant Character Controller et ses utilisations à l'adresse suivante : https://docs.unity3d.com/ScriptReference/CharacterController.html.

Le First-Person Controller Prefab est disponible dans le paquet Standard Assets, que vous pouvez télécharger à l'adresse suivante : https://assetstore.unity.com/packages/essentials/asset-

Comme vous commencez à peine à vous familiariser avec le mouvement des joueurs dans Unity, vous commencerez par utiliser le composant **Transform du** joueur dans la section suivante, puis vous passerez à la physique des corps rigides plus tard dans le chapitre.

DEPLACEMENT DU LECTEUR A L'AIDE DU COMPOSANT TRANSFORM

Nous voulons une aventure à la troisième personne pour *Hcuo Doun*, donc nous commencerons avec une capsule qui peut être contrôlée avec le clavier et une caméra pour suivre la capsule pendant qu'elle se déplace. Même si ces deux GameObjects fonctionneront ensemble dans le jeu, nous les garderons, ainsi que leurs scripts, séparément pour un meilleur contrôle.

Avant de pouvoir faire du scripting, vous devrez ajouter une capsule de joueur à la scène, ce qui est votre prochaine tâche.

Nous pouvons créer une belle capsule de lecteur en quelques étapes seulement :

1. Cliquez sur **+** > **Objet 3D** > **Capsule** dans le panneau **Hiérarchie** et nommez-la Player.
2. Positionnez la capsule du joueur dans le niveau comme vous le souhaitez, mais j'aime la placer près de l'une des rampes centrales. Il est également important de positionner la capsule au-dessus du plan du sol, donc assurez-vous de mettre la valeur Transform Y position à **1** dans l'**inspecteur**.
3. Sélectionnez le Player GameObject et cliquez sur **Add Component** en bas de l'onglet **Inspector.** Cherchez **Rigidbody** et cliquez sur *Enícu* pour l'ajouter. Nous n'utiliserons ce composant que plus tard, mais il est bon de configurer les choses correctement dès le début.

4. En bas du composant **Rigidbody**, développez la propriété **Constraints :**
   - Cochez les cases **Freeze Rotation** sur les axes *x*, *;* et *x* afin que le joueur ne puisse pas être tourné par accident lors de collisions ou d'autres interactions physiques. Nous voulons limiter la rotation au code que nous écrirons plus tard :

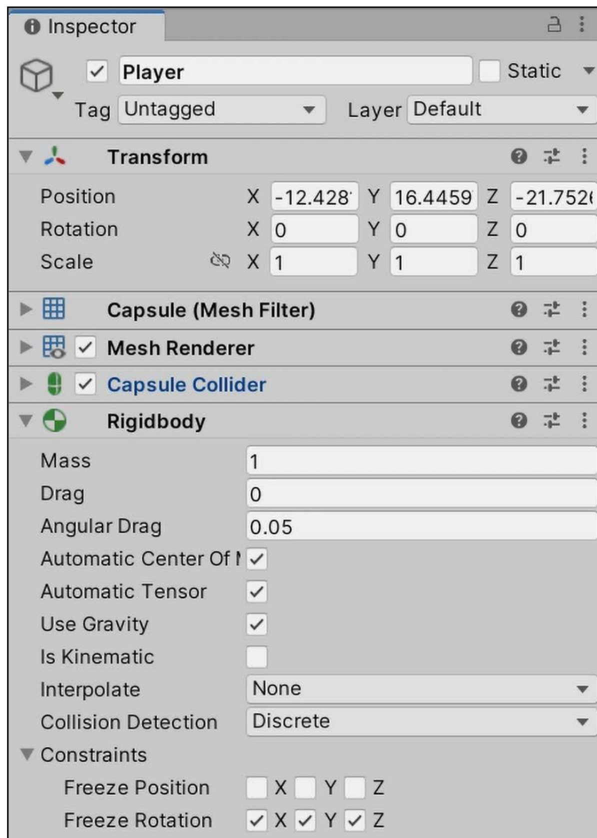


Figure 7.1 : Composant du corps rigide

Pour uuoviKc n comulcíc vicw ofi íhc Unií ; cKiíou, nll ouru scuccnshoís nuc ínkcn in fiullLscuccn moKc. Pour les images en couleur de tous les livres, utilisez le lien suivant : https://packt.link/7yy5V.

5. Sélectionnez le dossier Matériaux dans le panneau **Projet** et cliquez sur **Créer** | **Matériau**. Nommez-le Joueur_Mat.

6. Sélectionnez Player_Mat dans la **hiérarchie**, puis modifiez la propriété **Albedo** dans l'**inspecteur**.

   à un vert vif.

7. Faites glisser le matériau sur l'objet **Player** dans le panneau **Hiérarchie :**

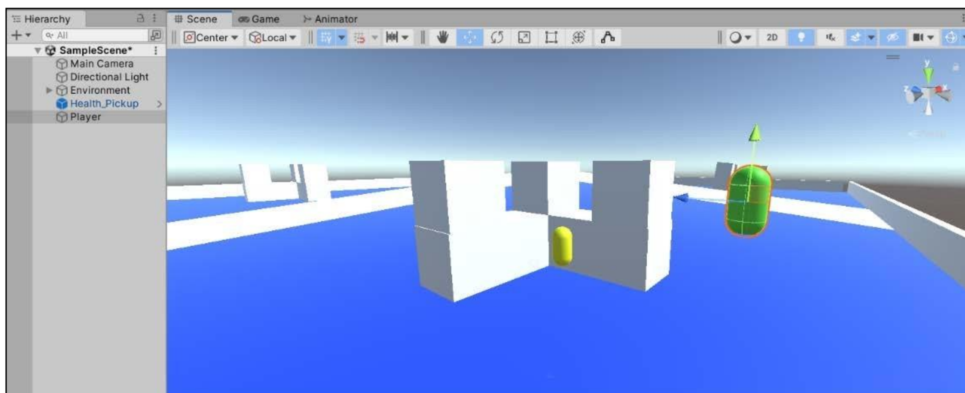


Figure 7.2 : Matériel du joueur attaché à une capsule

Vous avez créé l'objet **Player** à partir d'une primitive capsule, d'un composant **Rigidbody** et d'un nouveau matériau vert vif. Ne vous préoccupez pas encore de ce qu'est le composant **Rigidbody** - tout ce que vous devez savoir pour l'instant, c'est qu'il permet à notre capsule d'interagir avec le système physique. Nous entrerons dans les détails dans la section *Travailler avec íhc Unií ; uh;sics s;sícm* à la fin de ce chapitre. Avant d'en arriver là, nous devons parler d'un sujet très important dans l'espace 3D : les vecteurs.

COMPRENDRE LES VECTEURS

Maintenant que nous disposons d'une capsule de joueur et d'une caméra, nous pouvons commencer à étudier comment déplacer et faire pivoter un GameObject à l'aide de son composant **Transform.** Les méthodes Translate et Rotate font partie de la classe Transform fournie par Unity, et chacune a besoin d'un paramètre vectoriel pour exécuter sa fonction.

Dans Unity, les vecteurs sont utilisés pour contenir des données de position et de direction dans les espaces 2D et 3D, c'est pourquoi ils existent en deux variétés : Vector2 et Vector3. Ils peuvent être utilisés comme n'importe quel autre type de variable que nous avons vu ; ils contiennent simplement des informations différentes. Comme notre jeu est en 3D, nous utiliserons des objets Vector3, ce qui signifie que nous devrons les construire en utilisant les valeurs *x*, *;* et *x.*