Guide Unity

Société technique Poly Games

Itération 3

- Chali-Anne Lauzon –

Note : Ce document est basé sur les tutoriels Unity 3D Roll-a-ball et Space Shooter, disponibles sur le site officiel du framework, et ne se veut pas un substitut au support offert par sa communauté.

Contents

Comment fonctionne Unity?

Unity est un framework qui utilise le modèle «Entity Component»; une architecture qui favorise le modèle «Composition» plutôt que le modèle «Héritage»[1]. Chaque artéfact, ou élément (modèle 3D, animation, script, texture, etc.), est un objet indépendant et peut devenir «enfant» ou «parent» d'un autre objet. Par défaut, toute forme géométrique de base possède une position et une orientation (Transform) et un maillage pour le rendu (Mesh Renderer).

Illustrons tout cela par un exemple concret : dans le cadre d'un jeu, comment implémenter le joueur? Celui-ci devra posséder un script, qui déterminera son comportement vis-à-vis de son environnement, peut-être une animation et un modèle 3D attrayant ou un sprite. On peut voir à la figure 1 que le joueur est une sphère simple. En plus des éléments de base (Transform, Mesh Filter et Mesh Renderer), notre joueur possède aussi une boîte de collision (Sphere Collider), un corps rigide (RigidBody) et un script. Tous ces éléments sont «enfants» de «Player». Tout élément qui peut se déplacer dans l'espace verra sa position et son orientation modifiée dès que le joueur se déplace/tourne. Le joueur pourrait lui-même être «enfant» d'un autre objet.

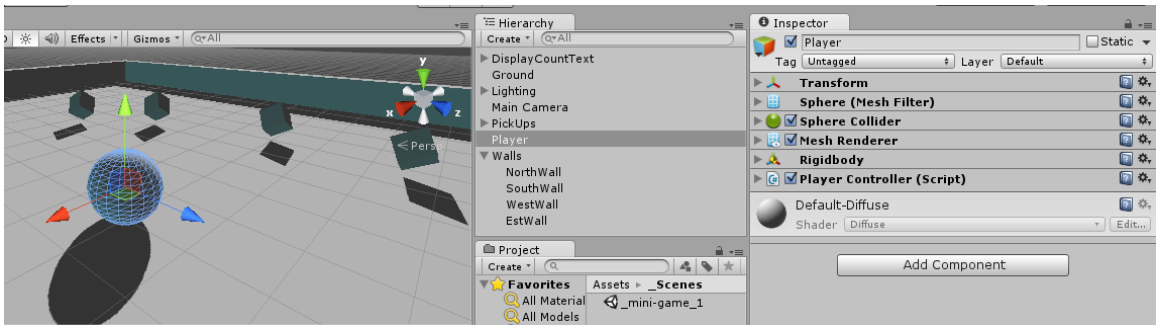

Figure 1 – Le joueur

[1] Il s'agit ici de patrons de conception.

Les fenêtres les plus utilisées dans Unity 3D

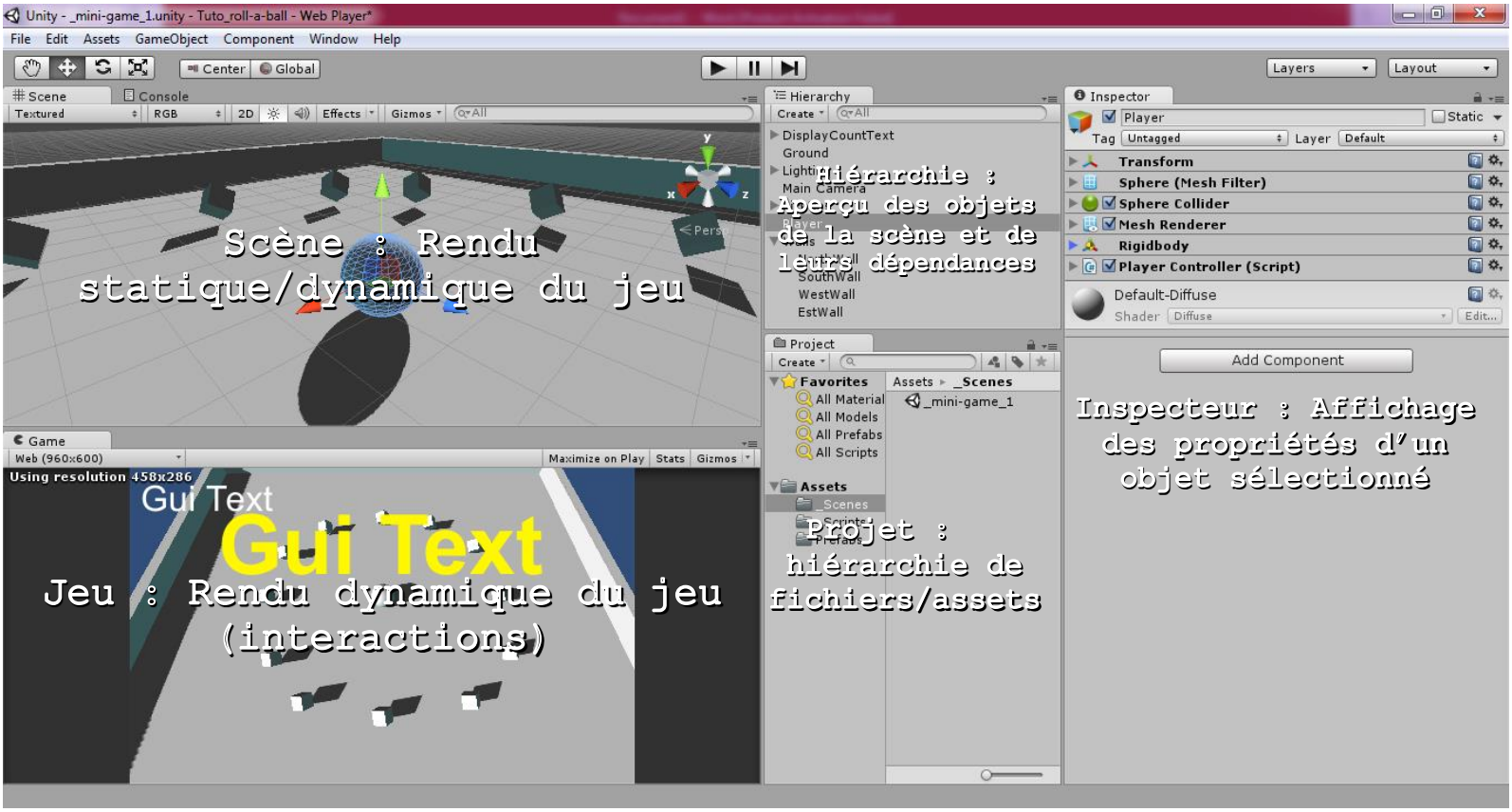

Figure 2 – Mise en forme d'un jeu

Dans Unity, toute l'architecture peut se faire à partir de ces fenêtres. Le framework supporte en effet le «drap-and-drop». On peut faire glisser un objet texture de la fenêtre «Project» sur un objet dans la scène pour la lui assigner automatiquement. Le même effet est obtenu si on glisse plutôt la texture sur l'objet dans la hiérarchie ou même dans l'inspecteur.

Notons que cela rend le partage de fichiers via des systèmes tels que Git plus complexes, puisqu'au fil du temps les dépendances peuvent entrer en conflit. La meilleure solution pour contrer cela reste une bonne communication au sein des équipes de développement.

Notions de base de programmation

Ce tutoriel ne se voulant pas exhaustif, seules les notions les plus élémentaires seront abordées ici :

- Les variables
- Les fonctions
- Les bonnes habitudes

Les variables

Une variable est une entité de base en programmation; elle sert à contenir une information. Il existe divers types de variables de base : *int*, *double* et *float* pour les informations numériques, *char* et *string* pour les caractères alphanumériques. On peut aussi avoir des tableaux, des listes chaînées, des hashmap, des dictionnaires et bien plus, mais ces notions ne seront pas abordées ici.

Dans Unity, le type de variable utilisé le plus souvent est *float*. Ce type permet de conserver des nombres à virgules négatifs et positifs allant de -3.4×10^{38} à 3.4×10^{38}[2]. Il faut ajouter un *f* minuscule à la fin d'un nombre en virgule flottante afin d'indiquer au programme qu'on utilise les *float* (voir figure 3).

```
float moveHorizontal = Input.GetAxis ("Horizontal");
float moveVertical = Input.GetAxis ("Vertical");

Vector3 movement = new Vector3 (moveHorizontal, 0.0f, moveVertical);
```

Figure 3 – Utiliser des float

L'IDE MonoDevelop permet l'auto-complétion des types de variables et des fonctions. Si on laisse son curseur sur une fonction, on peut obtenir sa déclaration (paramètres et valeur de retour s'il y a lieu). Pour chaque type ou fonction déjà existante, il est possible d'accéder à la documentation en ligne en appuyant sur CTRL+' sous Windows.

Les fonctions

Une fonction est une suite logique d'opérations. Une fonction peut prendre en entrée divers paramètres de types distincts et retourner un résultat sous forme de variable. Elle peut aussi ne rien retourner du tout.

Il existe dans Unity quelques fonctions de base qu'il est intéressant d'aborder; certaines apparaissent par défaut dans tout nouveau script de MonoDevelop :

- void Start() [défaut] : la méthode est appelée au lancement du jeu. On s'en sert surtout pour initialiser les références et les variables nécessaires à l'objet auquel se greffera le script.
- void Update() [défaut] : la méthode est appelée à chaque frame et contient généralement toute la logique du jeu.
- void FixedUpdate() : la méthode est appelée à chaque frame avant que le rendu visuel ne soit fait. Elle est idéale pour implémenter la physique du jeu (déplacements, collisions, etc.).

[2] http://msdn.microsoft.com/en-us/library/b1e65aza.aspx

- void OnTriggerEnter(Collider other) : la méthode sert à définir le comportement d'un objet qui possède une boîte/sphère/capsule de collision et qui entre en contact avec un autre objet qui en possède également une. Il faut que la boîte de collision du second objet soit un déclencheur (voir figure 4) et que l'objet possède un RigidBody.
- void LateUpdate() : la méthode est appelée à chaque frame une fois que les changements de tous les autres updates ont eu lieu. On s'en sert généralement pour les déplacements de caméra et les animations.

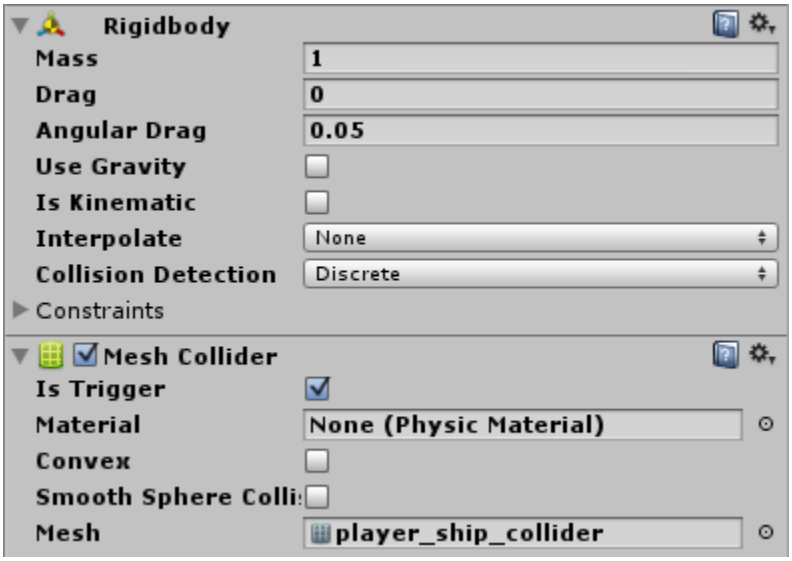

Figure 4 – Boîte de collision en tant que déclencheur

Les bonnes habitudes

Afin de rendre vos scripts et vos projets durables, voici quelques bons conseils :

1. Beaucoup de commentaires pertinents

Toute personne qui lira vos scripts, incluant vous, ne saura pas nécessairement ce que vos fonctions font, comment vous vouliez que telle fonctionnalité se greffe aille s'ajouter dynamiquement à un objet ou encore à quel objet devrait appartenir un script. Prenez alors le temps de mettre en commentaire (précéder de «//» le texte en commentaire sur la même ligne/fin de ligne et encadrer de «/*…*/» pour un commentaire sur plusieurs lignes) toute indication pertinente pouvant aider à la compréhension et/ou la relecture.

Les dépendances aux objets dans la hiérarchie ainsi que leur disposition attendue est hautement pertinente. Par exemple, si un script doit agir sur le parent de l'objet auquel il est assigné, mieux vaut le mentionner en commentaire afin de mieux retracer les erreurs prévisibles.

2. Des noms de variables/fonctions pertinents et concis

Tout au cours de la rédaction du code, il est recommandé d'utiliser des noms reliés à l'usage attendu de la variable/fonction afin de faciliter la compréhension de toute personne qui risque de travailler sur le projet. C'est pour votre propre bien.

Par exemple, si vous avez besoin d'une variable pour conserver le score, nommer la variable de type *int* «score» plutôt que «le_score_qui_sera_utilise_tout_au_long_du_jeu».

3. Séparer les modules par utilité

Pour chaque nouvelle fonctionnalité, écrivez une nouvelle fonction si possible. Une fonction pour avancer, une fonction pour faire bouger la caméra, une fonction pour partir le son, pour l'arrêter, etc. Au final, elles se retrouveront sans doute toutes dans Update() ou FixedUpdate() d'une façon ou d'une autre, mais ce sera beaucoup plus logique et lisible ainsi.

Si une fonction cause une erreur, il sera beaucoup plus facile de retrouver le chemin logique qui l'a causée.