Système de tuiles – Notions générales

Avantages:
+ Facile de gérer des environnements vastes
+ Des éléments de l'environnement peuvent être modifiés "avec mémoire" (Les objets qui quittent l'écran ne sont pas recyclés ou détruits comme dans un jeu « Endless Runner » ou « Spaceshooter » par exemple)
+ On peut même sauvegarder des environnements générés de façon aléatoire
+ Économie de mémoire (réutilisation des mêmes éléments graphiques, souvent le nombre total de tuile est fixe)
+ Collisions plus efficaces avec l'environnement
    - Système à tuiles traditionnel : Un personnage qui veut se déplacer n'a qu'à connaître la nature de la tuile qui se trouve directement devant lui pour déterminer s'il peut y entrer ou pas. Pas besoin donc de faire des tests de collision avec plein d'éléments de l'environnement à chaque frame. Cependant, les adversaires ou autres objets mobiles (ex: des projectiles) doivent parfois être détectés de façon conventionnelle.
    - Dans Unity : Des colliders et la physique sont utilisés pour gérer les collisions avec les tuiles. Les composants TileMapCollider2D et CompositeCollider2D aident à optimiser les collisions avec des tuiles regroupées dans un même *tile palette*. C'est bien plus efficace que d'avoir des colliders individuels appliqués à tout plein d'objets.

Désavantages:
- Demande beaucoup de précision dans la confection et le positionnement des tuiles
- Apparence "grille 2d" du jeu difficile à dissimuler
- Mouvements diagonaux parfois plus difficiles à gérer (surtout si le mouvement se fait sans colliders et physique).

Une tuile c'est:

- Un élément visuel de base carré de taille modeste (ou losange si le jeu est en affichage isométrique)
- La taille d'une tuile en pixels est souvent une élévation de 2 à la puissance *n* (16X16, 32X32, 64X64, etc.)
- Les tuiles sont généralement conçues pour que les "joints" entre elles ne soient pas trop apparents.
- Dans Unity, une tuile est le plus souvent un *sprite* associé à un *tile palette*
    - À titre indicatif, dans un logiciel comme Animate, on utiliserait un MovieClip pour chaque tuile.
    - Une approche plus « *hardcore* » consisterait à dessiner les tuiles directement en utilisant des opérations bitmap par programmation (on manipule directement les pixels). C'est une approche bien plus laborieuse à déployer pour un programmeur, mais elle permet de faire des optimisations surprenantes et ainsi faire fonctionner des jeux à tuiles complexes et vastes sur les machines peu performantes, sans avoir recours à des systèmes préétablis de gestions de tuiles et de physique.

Les systèmes de navigation à tuile sont plus faciles à concevoir si:

- Chaque tuile est soit un obstacle ou une tuile navigable (éviter les tuiles hybrides!)
- Les personnages ne peuvent s'arrêter qu'au centre des cases (c'est moins un problème dans Unity)

Deux types de navigation à tuile:

- Tuiles fixes et perso mobile (ex: Pacman, Zelda NES youtube.com/watch?v=wHaZrYX0kAU&t=0m45s)
- Tuiles mobiles et perso fixe (ex: Final Fantasy youtube.com/watch?v=oADjegTnKiA&t=8m50s, Pokémon youtube.com/watch?v=s_4zaj8EbFI&t=59m51s)

Une variante hybride est possible dans laquelle le perso est mobile et les tuiles aussi ce qui donne un effet de « caméra » qui suit le joueur tout en ne sortant pas de la zone de jeu (ex :Legend of Zelda : Minish Cap https://www.youtube.com/watch?v=UfEMNbjLd3Y

Inspiration de tuiles sur le net? *Dans Google, recherchez en mode image: tileset, tilesheet, tilemap, spritesheet*

Tuiles gratuites disponibles aussi sur OpenGameArt.org et itch.io (https://itch.io/game-assets/free)

Principales classes et composants utilisées pour les tuiles

Tile Palette	TileBase
Tilemap	Grid
TileMapCollider2D	Composite Collider 2D

Quelques fonctions de Tilemap à explorer pour manipuler les tuiles par prog…

GetTile()	SetTile()	SetTiles()
GetTilesBlock()	SetTilesBlock()	GetSprite()
SwapTile()	WorlToCell()	CellToWorld()

(source: https://docs.unity3d.com/ScriptReference/Tilemaps.Tilemap.html)

Solutions pour éviter les petits écarts parfois visibles entre les tuiles

- Solution 1 : Utiliser un asset Sprite Atlas pour optimiser l'organisation des sprites de tuiles
- Solution 2 : Dans le composant Grid utilisé, changer la valeur de Cell Gap pour -0.01 en x et en y
- Solution 3 : Ajouter un « padding » de pixels autour de chaque tuile dans Photoshop (plus laborieux, mais fonctionne à 100%, tout le temps!)

Déplacement dans Unity : Translate() vs RigidBody2D

Approche 1 : **Translate ou transform.position**

- Mise en œuvre dans un Update()
- N'applique pas la physique lors du déplacement
- Préférable pour animer des objets qui n'ont pas besoin de physique (exemple : fenêtre qui sort de l'écran, neige qui tombe sans collisions, nuages qui traversent le ciel, etc.)

Approche 2 : **RigidBody2D**

- Mise en œuvre dans un **FixedUpdate()**
- On applique un mouvement au **RigidBody2D** d'un objet
- _monRigidBody2D.**velocity** = valeur – ex : Donne une vitesse constante (attention : changer directement velocity contourne les autres calculs de physique, incluant la gravité! Ça peut causer des problèmes…)
- monRigidBody2D.**AddForce()** – ex : Accélérer un projectile, une boule de billard, etc.
- monRigidBody2D.**MovePosition()** – Souvent préféré : permet de déplacer un objet en modifiant sa position tout laissant Unity tenir compte des autres calculs de physique correctement. Attention : Si l'objet va trop vite, il peut quand même passer à travers les objets. Pour éviter cela, on peut activer *continuous* au lieu de *discrete* comme détection de collision au RigidBody2D (mais cela est plus exigeant pour le processeur).

Note : Update() est préférable à FixedUpdate() pour détecter les saisies (Input)

(*Référence : Unity Movement [RigidBody vs Translate]* https://www.youtube.com/watch?v=ixM2W2tPn6c)