Introduction à Unity

Réalisation d’un labyrinthe procédural

La création procédurale et les labyrinthes

D'après Wikipédia : *"En informatique, la **génération procédurale** (ou le modèle procédural) est la création de contenu numérique (niveau de jeu, modèle 3D, dessins 2D, animation, son, musique, histoire, dialogues) à une grande échelle (en grande quantité), de manière automatisée répondant à un ensemble de règles définies par des algorithmes. Le modèle procédural s'appuie sur les informations d'un algorithme pour créer."*

C'est un outil de plus en plus utilisé pour la création de jeux vidéo, nombreux sont ceux qui l'utilisent, allant de la simple altération de modèle 3D pour créer de la diversité, à la création de planètes et de galaxies, vous pourrez trouver de nombreux exemples sur itch.io, genre de Youtube des jeux vidéos.

Et quoi de mieux que la génération de labyrinthe pour mettre la création procédurale en pratique? (sûrement plein d'autres choses). Le cœur de ce TD sera donc, via de la génération procédurale, de créer des labyrinthes, et d'implémenter un jeu autour de ceux-ci.
A la fin de ce module, le projet à rendre sera donc un jeu de votre création centré sur la mécanique de création de labyrinthe procédural.

Voici les différentes étapes que nous allons suivre dans ce TD pour la production du jeu :

Cœur du jeu
- Génération de labyrinthe
- Instanciation du labyrinthe
- Navigation
- Contrôle du joueur
- Fin du niveau

Base de l'interface
- Écran titre
- Menu pause

Complétion du jeu
- Interface in-game
- Remplissage du labyrinthe
- Gestion du son
- Scénario
- Design

Cœur du jeu

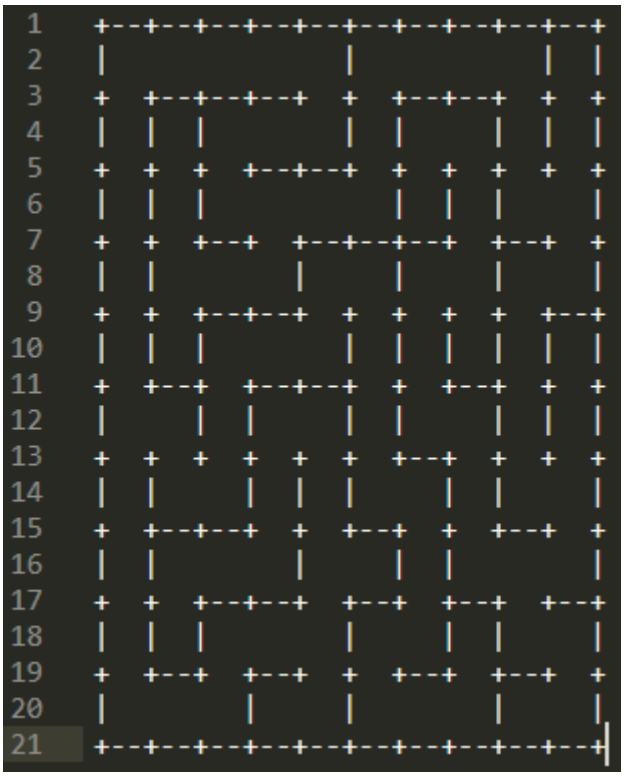
Génération de labyrinthe

La première étape de ce projet sera la création de labyrinthe. Pour ce faire, commencez par vous rendre sur ce site, qui vous donnera de nombreux exemples d'algorithmes pour en créer, bien détaillés et documentés.
Votre première tâche consistera donc à choisir un des algorithmes de ce site, et de l'implémenter en C# dans Unity.

Pour ce faire, je vous conseille de créer deux scripts dans Unity, un contenant un MonoBehaviour qui permettra d'instancier les différents éléments du labyrinthe (`LevelManager.cs` , par ex), et un script qui contiendra l'algorithme qui s'occupera de la création et de la gestion des données du labyrinthe (`MazeGenerator.cs`). Ce second script n'aura pas besoin d'être un MonoBehaviour.

Faites en sorte que `LevelManager.cs` puisse appeler une méthode de construction de labyrinthe depuis `MazeGenerator.cs`, et que celui-ci retourne un tableau contenant l'ensemble des cases du labyrinthe. L'utilisation de `Flags` et `Enum` pour définir le contenu du tableau serait une solution des plus adaptées.

Afin de tester que le générateur de labyrinthe fonctionne, sans avoir à implémenter entièrement `LevelManager.cs` , vous pouvez implémenter une méthode permettant de décrire le contenu du labyrinthe dans un fichier texte.



Instanciation du labyrinthe

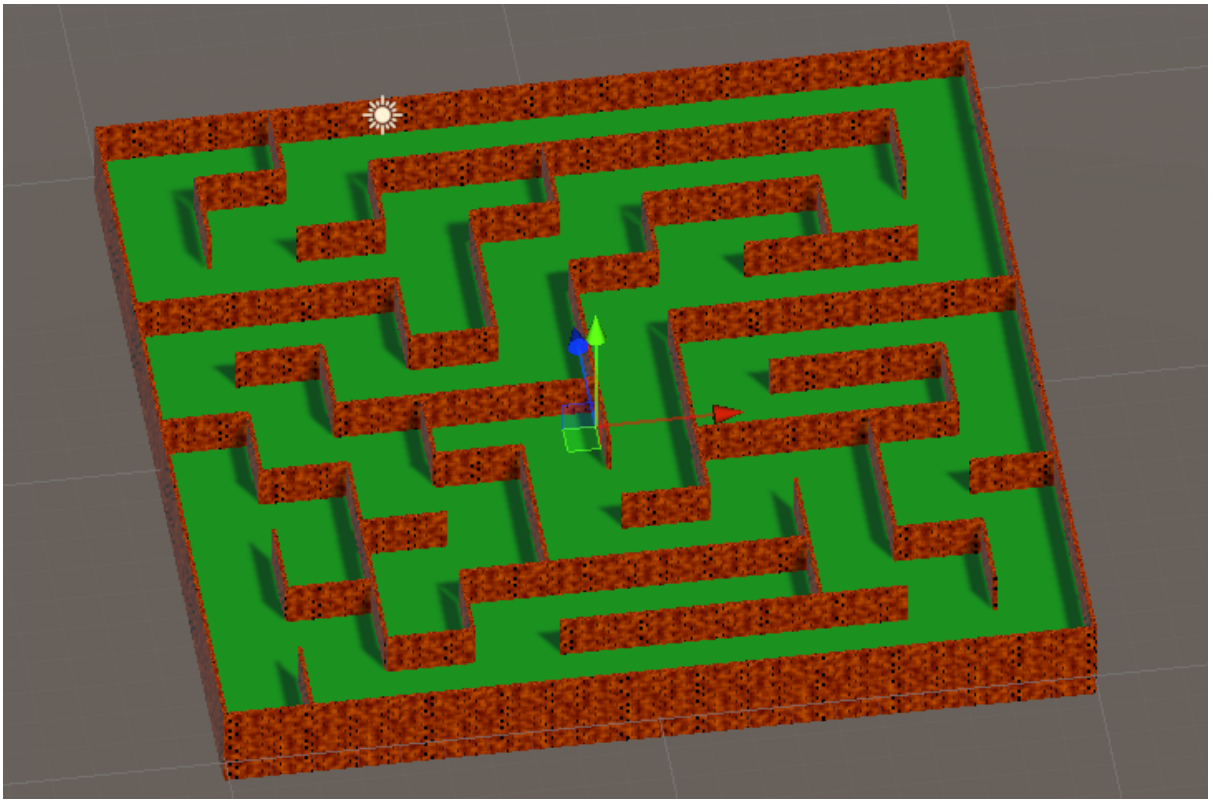
Une fois les données du labyrinthe obtenues, passons à l'instanciation de celui-ci.
Commencez par créer deux prefabs dans Unity, l'un servira de bloc pour le sol du labyrinthe, et l'autre pour les murs (deux cubes de couleur différente feront l'affaire, pour commencer).

Ajoutez des champs sérialisés dans `LevelManager.cs` afin de pouvoir utiliser ces prefabs depuis le script.

En parcourant le tableau retourné par `MazeGenerator.cs` , identifiez le centre de chaque case du labyrinthe en fonction de sa taille et de son échelle. Grâce au centre de chaque cellule, vous pouvez calculer la position et la taille que doivent prendre les différents blocs à instancier : les blocs du sol doivent recouvrir l'entièreté du sol de la case, et les murs doivent être placés sur les côtés de la cellule.

Petit conseil pour l'instanciation des côtés : instanciez le mur au centre de la cellule, et décalez le d'une unité sur le côté, ensuite, utilisez la méthode `Transform.RotateAround` pour déplacer le mur du côté nécessaire, en le faisant tourner autour du centre de la cellule.

En répétant ce processus pour chaque cellule, vous devriez obtenir une version 3D de votre labyrinthe dans Unity.



Navigation

NavMesh

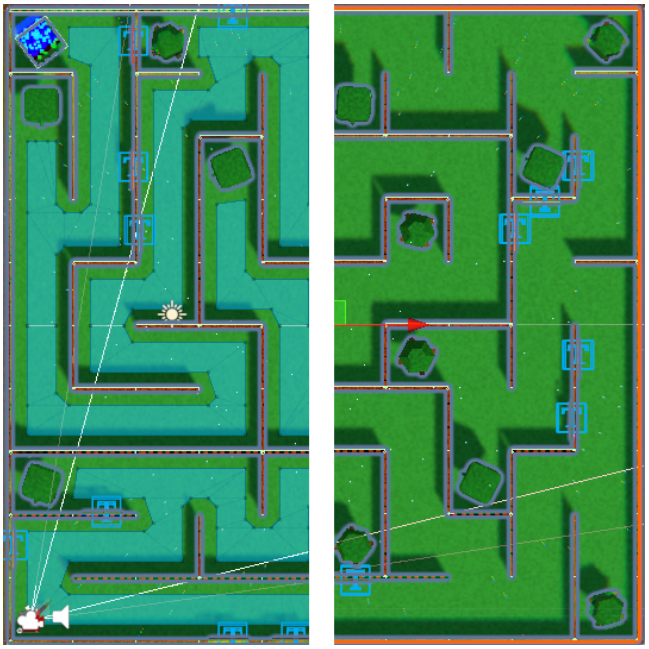
Dans Unity, `NavMesh` est un composant qui permet de définir l'espace dans lequel un `NavMeshAgent` (par exemple le joueur ou des PNJ) peut se déplacer. Son utilisation va ici nous permettre de vérifier que la création et l'instanciation du labyrinthe se sont bien déroulées, de manière automatisée.

Afin de pouvoir utiliser pleinement les `NavMesh`, il vous sera nécessaire d'ajouter ces sources à votre projet. Elles contiennent des méthodes et composants Unity permettant d'améliorer l'utilisation des `NavMesh`, notamment en donnant la possibilité de les construire à la volée depuis un script (ce qui n'est normalement possible que depuis l'Editor).

En reprenant le code réalisé plus tôt pour créer votre labyrinthe dans la scène, faites en sorte que tous les blocs soient instanciés en tant qu'enfant d'un même parent. C'est sur ce parent que vous devrez ajouter un composant de type `NavMeshSurface` , soit depuis l'Editor, soit depuis votre script.

Une fois le labyrinthe instancié, il vous suffira de générer le `NavMesh` de celui-ci, simplement en appelant la méthode `BuildNavMesh()` du `NavMeshSurface` .

En démarrant le projet dans l'Editor, si vous sélectionnez l'objet portant le `NavMeshSurface` depuis la hiérarchie, vous pourrez voir dans la scène la représentation du `NavMesh`.



NavMesh visible à gauche de l'image, en bleu

Détection des impasses

Pour définir les points de départ et d'arrivée du labyrinthe, nous allons utiliser le `NavMesh` pour calculer quel est le chemin le plus long (point de sadisme ici, il est simplement bon de penser à la durée de vie du jeu).

Pour commencer, il faut définir un point précis du labyrinthe comme étant le point de départ (soit au hasard, soit en sélectionnant un des coins par exemple). Ce point défini, placez-y un nouveau `GameObject` (une sphère, un cube, un modèle 3D, etc...) qui servira de personnage jouable. Cet objet devra posséder un composant de type `NavMeshAgent` , qui lui permettra de naviguer dans les limites du `NavMesh` .
En déplaçant la caméra en enfant de ce nouvel objet, vous obtiendrez facilement une caméra de type FPS / TPS.

Également, il vous faudra modifier `MazeGenerator.cs` , afin que celui-ci ajoute un `Flag` "`DeadEnd`" sur les cellules qui ne sont accessibles que par un seul côté (et qui sont donc des impasses). En récupérant toutes les positions des impasses, potentiellement au moment de définir le centre des cases du labyrinthe, vous pourrez comparer la distance entre la position de départ, et celle de chaque impasse, puis retenir la plus longue distance pour définir l'impasse correspondante comme étant le point d'arrivée du niveau.

Pour obtenir la distance entre un `NavMeshAgent` , et une position du NavMesh , il faudra pour chaque point utiliser `NavMeshAgent.SetDestination()` , puis utiliser la méthode d'extension suivante pour obtenir la distance.

```
public static float GetPathRemainingDistance(this NavMeshAgent navMeshAgent)
{
    if (navMeshAgent.pathPending ||
        navMeshAgent.pathStatus == NavMeshPathStatus.PathInvalid ||
        navMeshAgent.path.corners.Length == 0)
        return -1f;

    float distance = 0.0f;
    for (int i = 0; i < navMeshAgent.path.corners.Length - 1; ++i)
    {
        distance += Vector3.Distance(navMeshAgent.path.corners[i], navMeshAgent.path.corners[i + 1]);
    }

    return distance;
}
```

Enfin, pour tester que tout fonctionne, utilisez `NavMeshAgent.SetDestination()` sur la position de fin du labyrinthe ; votre personnage devrait se déplacer tout seul du point de départ au point d'arrivée du labyrinthe !

Contrôle des déplacements

Ce qu'il manque à présent pour avoir une base jouable pour le jeu, c'est d'implémenter les mouvements du personnage.

Sur le `GameObject` servant de joueur, ajoutez un nouveau script `PlayerController.cs` , il servira de cerveau pour les mécaniques du joueur.

Avant d'implémenter les déplacements, il faut savoir qu'il existe une contrainte lorsque que l'on veut modifier la position d'un objet qui est sous l'emprise d'un `NavMeshAgent` : les déplacements doivent s'effectuer via la variable `velocity` du `NavMeshAgent` . Pour faire avancer le joueur le plus simplement possible, on peut donc récupérer le vecteur `Forward` du joueur, et l'ajouter à la vélocité de l'agent lorsque l'on appuie sur la touche ↑ du clavier.
Reste maintenant à gérer les rotations du joueurs, pas besoin cette fois de passer par la vélocité de l'agent, on peut simplement modifier la rotation du joueur lors de l'appuis sur une touche.

Fin du niveau

Maintenant que l'on peut se déplacer jusqu'à la fin du niveau, il serait bon de pouvoir détecter cet événement, et d'en faire quelque chose. En ajoutant `RigidBody` et `Collider` (et en les configurant correctement), sur le joueur, et sur un objet à la fin du niveau, vous serez en mesure de détecter la collision entre les deux, et par exemple de recharger la scène entière, afin de recommencer le niveau, dans un nouveau labyrinthe procédural par exemple.

Vous devriez maintenant pouvoir naviguer dans un labyrinthe procédural dans Unity, félicitations !

Base de l’interface

Écran titre

Un jeu sans écran titre, c’est un peu comme une maison sans porte : on y a facilement accès mais c’est quand même pas très pratique.

Pour réaliser un écran titre, commencez par créer une nouvelle scène. Dans cette nouvelle scène, vous allez pouvoir ajouter un `Canvas`, base des interfaces dans Unity. En enfant de cet élément, vous pouvez ajouter Image, Texte et Bouton pour concevoir votre écran-titre.

Afin de pouvoir lancer le jeu en appuyant sur un bouton, il vous faudra faire deux choses. Tout d’abord, attachez un nouveau script au canvas, permettant de charger la scène suivante (celle réalisée plus tôt) via une méthode public. Ensuite, dans l’Inspector du bouton, vous pourrez trouver dans le composant `Button`, une section OnClick(). C’est cette section qui permet au bouton d’appeler des méthodes lorsque l’on clique dessus. Glissez dans le champ en bas à gauche le `GameObject` tenant votre script de chargement de la scène, et sélectionnez dans le dropdown à droite la méthode public définie plus tôt : d’un simple clic, vous êtes maintenant en mesure de démarrer le jeu !

Menu pause

De la même manière que pour l’écran titre, l’écran de pause est un élément présent dans tous les jeux, ou presque. Celui-ci permet généralement de gérer les options du jeu (volumes, contrôles, etc...), d’accéder aux informations du jeu (temps de jeu, crédits, etc...) ou tout simplement de quitter le jeu.

Similairement à l’écran titre, il vous faudra créer dans votre scène de jeu un `Canvas` qui supportera les éléments de l’écran de pause. Cet écran devra s’afficher lorsque l’on appuie sur une touche, et disparaître lorsque l’on appuie à nouveau dessus, ou en cliquant sur un bouton (vous pouvez simplement activer/désactiver le `GameObject` supportant le `Canvas` depuis un script).

Complétion du jeu

Comme énoncé au début de ce document, ce TD a pour but de vous guider dans les bases de la réalisation du projet, mais seulement les bases ne suffiront pas pour faire de votre projet un bon jeu.

La section suivante contiendra donc différentes indications qui pourront vous guider vers la finalisation de votre projet. Notez bien que c'est principalement cette section qui permettra d'évaluer votre maîtrise des bases de Unity, mais c'est aussi une occasion pour vous de montrer vos talents de créateurs de jeux vidéo!

Interface in-game

L'interface in-game d'un jeu est l'interface qui est visible en continu dans le jeu. Elle permet de transmettre diverses informations au joueur. Pour ce projet de labyrinthe, il pourrait être intéressant de laisser un certain temps au joueur pour qu'il finisse le niveau, ou bien d'ajouter un système de boussole dans l'interface afin de repérer plus facilement la fin du niveau, ou encore un système de barre de vie si vous désirez ajouter des ennemies au jeu...

Remplissage du labyrinthe

En parlant d'ennemies, la gestion procédurale du labyrinthe pourrait aussi être l'occasion d'y ajouter des ennemies à des endroits précis, ou plus simplement de remplir le labyrinthe avec des éléments de décors pour que la navigation dans celui-ci crée moins un sentiment d'errance.

Gestion du son

Afin d'apporter plus d'immersion dans le jeu, il est temps d'y ajouter du son : pour les projectiles, les collisions, la musique de fond, etc… Pour gérer le son dans Unity, deux composants sont nécessaires, un `AudioListener` (normalement déjà présent sur la caméra de la scène), et des `AudioSources`, qu'il vous faudra ajouter sur les objets qui selon vous doivent émettre un son. Si vous utilisez des sons que vous n'avez pas produits vous même, il vous faudra les lister, ainsi que leurs créateurs, dans un écran des crédits (par exemple accessible via l'écran titre du jeu). Egalement, pensez à utiliser des sons et des musiques libres de droit, ce qui pourra vous permettre de publier le jeu si vous le désirez, une fois terminé.

Scénario

Un des points clef d'un jeu est de parvenir motiver le joueur à progresser dans le jeu : est-ce que votre jeu sera un jeu d'exploration contemplatif, où l'on se promène dans un labyrinthe aux décors reposants et doux, ou bien sera-t il un rogue like où le moindre faux pas pourra être fatal?
Sans forcément créer un univers complet pour votre jeu, essayez de faire en sorte que celui-ci soit attractif, et ne soit pas juste composé des cubes blancs par défaut de Unity.

Design

Faites en sorte de rendre votre jeu visuellement attractif : vous pourrez au choix dessiner et modéliser vous même les différents assets du jeu, ou bien utiliser les sources d'assets disponibles en ligne. Vous pouvez par exemple utiliser la fenêtre Asset Store de Unity, ou toutes autres sources, comme par exemple Kenney Assets. Si vous désirez créer vous même les assets, il existe de très nombreuses solutions, gratuites ou payantes : Blender, Krita, Aseprite, Asset Forge, Photoshop, etc… Encore une fois, si vous utilisez des assets qui ne sont pas les vôtres, pensez à citer les sources dans un écran des crédits.

Rendu du projet

Ce projet servira à vous évaluer -en partie- pour ce module. Vous pourrez le rendre individuellement ou en groupe de deux.

Afin de faciliter votre travail d'équipe et de pouvoir efficacement me transmettre le projet, merci de mettre en place un git pour gérer les sources de votre jeu (si vous n'avez pas l'habitude, il existe un très grand nombre de clients pour git, comme TortoiseGit, SourceTree, GitKraken, etc...).
Pour l'utilisation de Git, afin d'éviter les fichiers Unity qui ne sont pas nécessaires lors du partage des sources, vous pourrez utiliser un fichier .gitignore comme celui-ci.

Votre code devra être commenté, et les fichiers devront être ordonnés correctement dans le projet Unity. Si vous utilisez des sources qui ne sont pas les vôtres, pensez à les créditer.

sources externes non créditées = points perdus sur le projet

La date du rendu du projet est le 19 novembre 2021 à minuit.
Pour pouvoir vous évaluer sur ce module, vous devrez m'envoyer un accès git à votre code par email à l'adresse gallien.kevin[at]outlook.com, publier votre jeu sur itch.io, et enfin inscrire votre jeu à la game jam (https://itch.io/jam/td-generation-de-labyrinthe).
Votre jeu devra pouvoir être builder pour Windows, et en plus, si vous le désirez, en WebPlayer.

Si vous avez des questions sur certains points du projet, n'hésitez pas à me contacter par email (gallien.kevin[at]outlook.com) ou par Teams ou Discord.