TD4 : NavMesh, GUI et changement de scène

Florian Jeanne

Rémy Frenoy

Yann Soullard

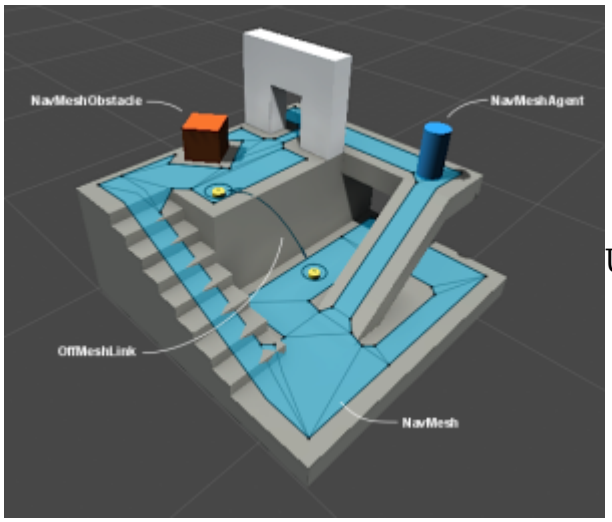
Rappel des trois premières séances de TD

Lors des trois premières séances de TD, nous avons appris à créer différents types d'objets dans Unity, à les déplacer de différentes manières (*AddForce*, *Translate*). Nous en connaissons davantage sur la gestion de la physique (*rigidbody, colliders, triggers*). Nous connaissons plusieurs fonctions de mise à jour (Update, FixedUpdate, LateUpdate). Nous avons également travaillé sur le positionnement de la caméra (fixe, héritage d'un objet, suivi des translations via un script dédié). Nous avons utilisé un *standard asset* de Unity (le First Person Character) et lui avons ajouté la capacité de saisir des objets en utilisant le principe du ray casting. Nous savons également modifier les curseurs pour fournir à l'utilisateur des retours sur son interaction avec l'environnement.

Introduction aux *NavMeshs*

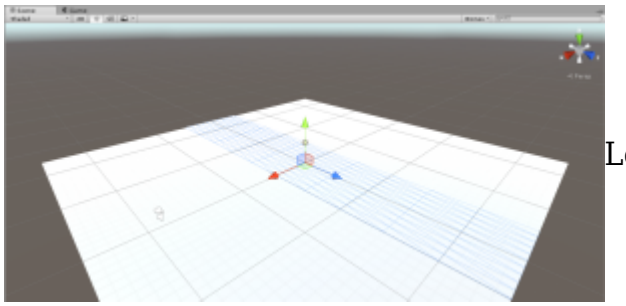
Il est souvent utile d'avoir des personnages ou des objets se déplaçant "tout seuls" dans une scène. Il faut pour cela utiliser des NavMesh et des NavMesh Agents. Un *NavMesh* est une surface que l'on définit comme navigable (sur l'image suivante, la zone bleue est navigable, les zones grises ne le sont pas), et les *NavMesh Agents* sont des intelligences artificielles capables de se déplacer sur ces surfaces (et uniquement sur ces surfaces).

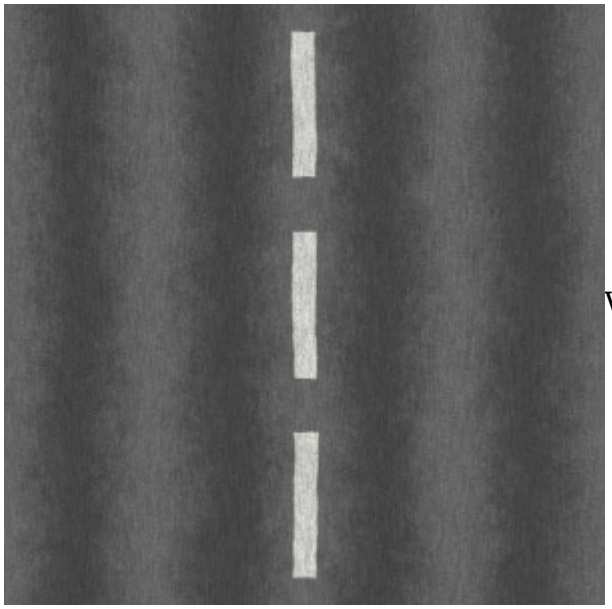
Ces outils permettent de construire des scènes complexes, avec notamment des obstacles, des endroits où la navigation sera ralentie (de la boue par exemple), etc…



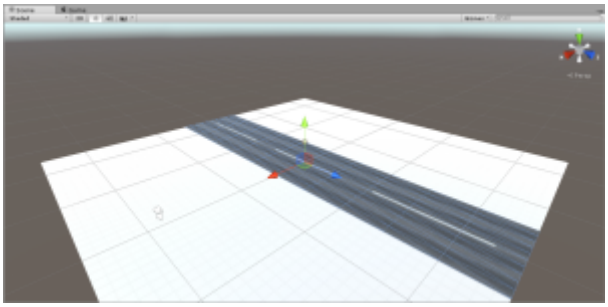
Une scène de navigation complexe, composée d'un mesh de navigation, d'un agent, d'un obstacle et d'un lien off-mesh.

Nous allons créer une route sur notre plan. Pour cela, ajoutez un nouveau plan (que vous nommerez "Road") sur votre scène, ajustez les variables *scale* de la manière suivante : (5 ; 1 ; 5) pour le *plane* de base et (1 ; 1 ; 5) pour "Road". On obtient quelque chose comme ceci:


Le plan "Road" sélectionné sur la scène. Vous pouvez lui ajouter cette texture :



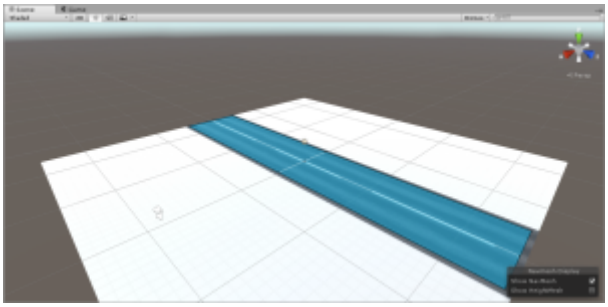
Vous obtenez ceci :



Le plan “Road” avec sa jolie texture

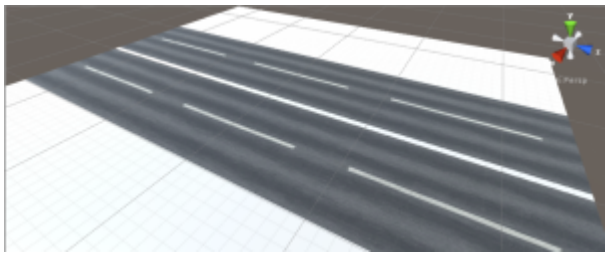
Attention à ce que le plan “Road” soit très légèrement surélevé (de 0.1 par exemple) par rapport à votre plan principal, sans quoi vous obtiendrez des bugs graphiques.

Nous voulons créer un *NavMesh* sur cette route. Pour cela, ouvrez la fenêtre *Navigation* (**Window > Navigation**). Cette fenêtre s’ouvre dans un onglet à côté de l’*inspector*. Dans la fenêtre *hierarchy*, sélectionnez votre objet “Road”, puis dans la fenêtre *navigation* cochez la case “Navigation static”. Sélectionnez ensuite “Bake” dans le coin inférieur droit de la fenêtre navigation. Vous obtenez le résultat suivant :

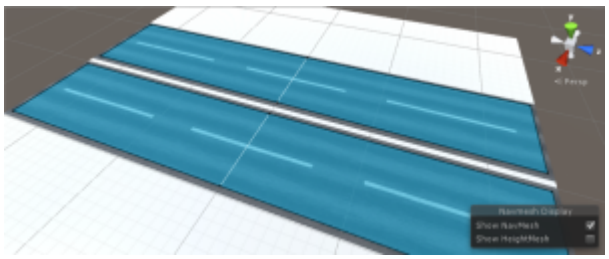


Un NavMesh est construit sur la route

Cette zone bleue signifie qu'un *NavMeshAgent* pourra se déplacer sur cette zone. Dupliquez maintenant votre route pour en avoir deux et faire une 2×2 voies.



Cependant, si vous retournez dans l'onglet *Navigation*, catastrophe !! Le *NavMesh* n'est plus sur la route… Et il n'a pas été dupliqué en copiant la route… En réalité, le *NavMesh* est indépendant des autres objets. Il vous faudra redéfinir votre *NavMesh* si vous modifiez l'environnement. Ainsi, enlevez le *NavMesh* existant en cliquant sur le bouton "Clear" dans la fenêtre *Navigation*, et redéfinissez un *NavMesh* qui couvre les deux routes. Vous devriez obtenir ceci :

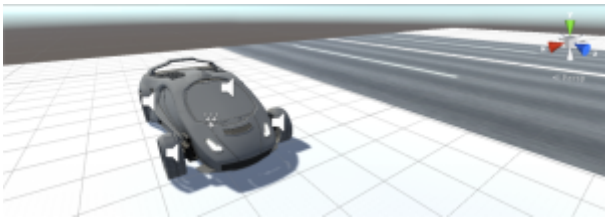


Attaquons-nous désormais aux *NavMeshAgents*.

Introduction aux *NavMeshAgents*

Sur une route, on trouve souvent des voitures. Importez donc le package **Assets > Import package > Vehicules** (ne sélectionnez que l'ensemble des dossiers "Car" et "CrossInputPlatform". CrossInputPlatform est une dépendance de la plupart des assets standards. Vous aurez des erreurs de compilation tant que vous n'aurez pas importé cet asset).

Depuis la fenêtre *Project*, glissez-déposez le prefab "Car" sur votre scène.



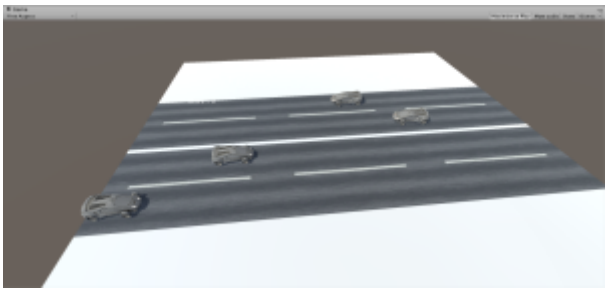
Même Batman a du réduire les dépenses…

Dans la fenêtre *inspector*, désactivez le script "Car User Control", puisque le but est justement que la voiture se déplace d'elle-même, sans intervention de l'utilisateur. Créez un script "NavMeshScript" dans le dossier *Scripts* du standard asset *Car*, et placez-le sur la voiture. Aussi, ajoutez un *NavMeshAgent* à votre voiture (**Component > Navigation**).

Dans ce code, vous devrez définir la destination du *NavMeshAgent* via la méthode SetDestination(), vers laquelle chaque *NavMeshAgent* va se diriger. Pour un premier test, vous pouvez la coder "en dur" (dans mon cas, (-2.5f, 0.0f, -4.5f) correspond à l'extrémité de ma route). Plutôt que d'utiliser cette méthode (assez moche vous en conviendrez), utilisez plutôt un *GameObject* vide et placez le à l'extrémité de la route où se trouve votre voiture. Ce sont alors les coordonnées de cette objet que vous récupérez dans le code pour définir la cible de votre NavMeshAgent. Pensez à détruire les voitures une fois qu'elles sont arrivées à destination, dans le cas contraire, l'accumulation d'objets peut provoquer des ralentissements sur votre ordinateur.

Votre voiture se déplace désormais toute seule sur la route jusqu'à atteindre sa cible. Nous voudrions toutefois instancier des voitures à une **fréquence aléatoire** pour avoir un semblant de trafic sur notre route. Depuis votre voiture, créez un prefab "NavMeshCar". Puisque vous avez déjà instancié dynamiquement des objets sur une scène, utilisez le même principe que dans le *CubeGenerator* du TD2 pour instancier des NavMeshCar sur un bord de la route, afin qu'elles se déplacent vers l'autre bord.

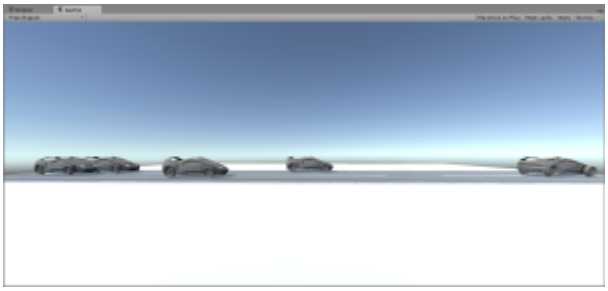
Copiez les mécanismes développées précédemment afin de pouvoir générer de manière aléatoire des voitures sur chaque voies :



Le périphérique Parisien à l'heure de pointe

Triggers et klaxon

En ajoutant un collider sur le prefab de votre NavMeshCar, par exemple un rectangle devant la voiture, en sélectionnant l'option *isTrigger* et en surchargeant la méthode *OnTriggerEnter()*, vous pouvez faire en sorte que les voitures klaxonnent lorsqu'elles se rapprochent dangereusement d'un objet. Pour jouer un son, il est possible d'utiliser la fonction AudioSource.Play(). Vous pouvez créer une source audio depuis un objet AudioClip, comme celui-ci pour le klaxon.



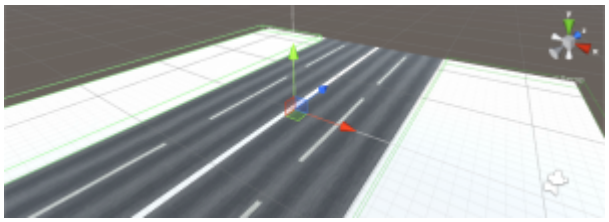
BIP BIIIIIIP

C’est l’heure du du-du-du-du.. euh du jeu.

Nous avons une véritable autoroute. Cependant, c’est un peu vide comme application. Nous allons donc créer un petit jeu. Ajoutez un *FirstPersonCharacter* comme dans le TD précédent, et positionnez-le sur le bord de la route. Le but du jeu sera de traverser les voies sans se faire renverser par une voiture. A chaque fois que le personnage traverse la route sans encombre, il gagne un point. Cependant, s’il se fait toucher par une voiture, son score est décrémenter de un point et il retourne à sa position initiale.

Pour gérer le score, nous allons créer un *Game Manager* qui stockera le score et les méthodes servant à l’incrémenter ou le décrémenter. Ce script peut être par exemple ajouter à un *GameObject* vide quelconque que l’on nommera “GM”.

Ensuite, positionnez un *trigger* de chaque côté de la route, de manière à ce que le personnage soit détecter une fois qu’il arrive de l’autre côté. On pourra par exemple créer un cube faisant toute la largeur du plan (voir image) et désactiver son *Mesh Renderer* afin de n’avoir que son *Collider* (dont le isTrigger est positionné à True).



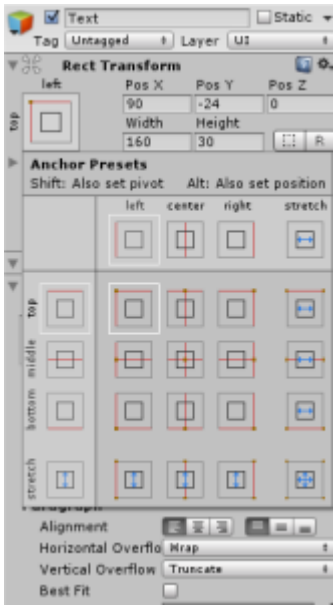
Associez-leur un script qui incrémente le score une fois qu'il passe d'un côté de la route à l'autre. Cependant, pour ne pas incrémenter le score plus d'une fois à chaque fois, on désactivera le *trigger* dans lequel le joueur vient de pénétrer, et on activera celui de l'autre côté de la route, ainsi de suite. Le premier *trigger*, c'est-à-dire celui dans lequel se trouve le personnage au départ, est de base désactivé. Faites néanmoins attention. On ne désactive pas le *GameObject* contenant le *trigger*, mais bien le composant en lui-même à l'aide de Collider.enabled.

GUI : *Graphical User Interface*

Votre jeu marche bien, le score fonctionne correctement mais nous ne l'affichons pas encore au joueur. C'est donc le moment de voir comment l'interface graphique est gérée sous Unity. Il existe différents éléments existants sous Unity pour composer votre GUI, vous pouvez retrouver différents tutoriaux à cette adresse : Doc Unity. Aujourd'hui nous allons seulement nous intéresser à trois types d'éléments de l'interface à savoir les boutons, le texte et les images.

Tout d'abord, il faut savoir que n'importe quel composant de l'interface graphique, donc un bouton ou autre, sera toujours intégré dans un *Canvas*. Ce *Canvas* est une zone délimitant notre interface. Il sera généré automatiquement lorsque vous créer un bouton par exemple.

Dans notre application, nous désirons afficher le score du joueur. Ajoutez donc un texte à votre scène (**GameObject > UI > Text**). Comme tous les éléments d'interface de Unity, il possède un Rect Transform qui vous permet de le positionner dans le *Canvas* et de définir manuellement sa taille. On y retrouve aussi les notions d'ancre ou encore de pivot qui vous permettent de placer votre éléments par rapport à l'objet parent, si et seulement si celui-ci possède également un *Rect Transform*, ce qui dans la pratique le sera dans 99.999999% des cas. On veut afficher le score en haut à gauche et ce, quelque soit la taille de l'écran. Nous allons donc figer le texte en haut à gauche de notre *Canvas* à l'aide de l'ancre correspondante.



L'objet *Text* créé comprend un composant *Text (script)*, qui lui-même contient un champ texte. Cependant, ce champ est utile si notre texte était statique. Or, nous voulons actualiser le score en temps réel, il faut donc que le texte soit dynamique. Pour cela nous allons créer un script. Ajoutez donc un script à l'objet *Text*. Il devra récupérer le score stocké dans notre *GameManager* et l'afficher en modifiant le texte de l'objet *Text* (vous suivez toujours ?). Cependant, pour pouvoir accéder à la propriété Text.text, nous devons inclure la librairie correspondant à l'interface utilisateur. Ajoutez donc tout en haut de votre script la ligne :

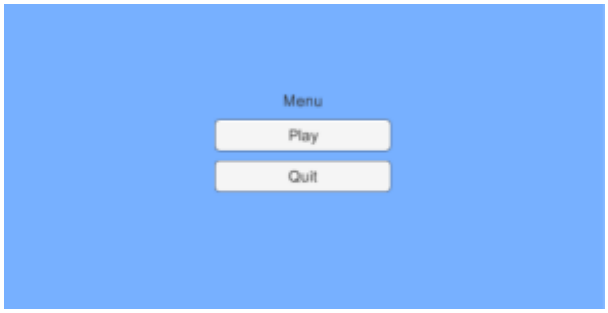
```
using UnityEngine.UI;
```

Une fois ceci terminé, votre score doit se mettre correctement à jour et s'afficher dans la foulée.

Chargement et lancement du jeu

Créez une nouvelle scène et ajoutez lui une image (**GameObject > UI > Image**). Modifiez sa taille de façon à ce qu'elle remplisse le *Canvas*. Vous pouvez faire cela directement dans le *Rect Transform*, au niveau des ancres prédéfinies en maintenant la touche Alt enfoncée.

Ajoutez-lui ensuite un texte et deux boutons comme ceci :



Nous devons cependant associer des actions aux boutons. Notre bouton Play devra charger la scène principale contenant notre jeu, alors que Quit fermera l'application.

Créez tout d'abord un script contenant deux méthodes publiques :

- Quit() qui servira à fermer l'application à l'aide de Application.Quit() ;
- Play() qui chargera donc le jeu à l'aide de Application.LoadLevel();

Vous pouvez ajouter ce script à la caméra par exemple, ce n'est pas très important ici. Ensuite, nous devons ajouter nos deux scènes créées dans les paramètres du projets afin qu'elles soient prises en compte pendant le build du projet. Pour cela, allez dans **File > Build Settings**. En ayant au préalable sauvegardé votre scène, cliquez sur le bouton "Add Current". Chargez votre scène principale et refaites la même opération. Vos scènes sont alors toutes les deux ajoutées au build du projet.

Pour info :

- Dans le script que nous venons de créer, Application.Quit() ne fonctionnera que si vous effectuez un build du projet, c'est à dire si vous générez un exécutable indépendant de Unity. Ainsi, si vous lancez l'application comme d'habitude dans l'éditeur, cette méthode de fonctionnera pas (comme précisé dans la doc Unity de la méthode Quit()) ;
- Pour la méthode LoadLevel(), vous pouvez mettre en paramètre le nom de votre scène principale ou son identifiant. Pour connaître ce dernier, il suffit d'aller dans les

paramètres du build comme expliqué précédemment et de relever le numéro associer à la scène correspondante.

Pour aller plus loin

Notre NavMesh est très rudimentaire (pour un cas aussi simple, il est même possible de faire sans, mais c'était l'occasion de vous familiariser avec cet outil qui peut s'avérer très utile et puissant). Vous pouvez l'améliorer de nombreuses façons, par exemple en faisant une route plus complexe (vous pouvez utiliser l'asset EasyRoads3D Free), ou en considérant le personnage comme un obstacle mobile, les voitures essayant alors de l'éviter plutôt que d'aller tout droit.