viki53

Zeste de savoir

Créer un jeu HTML5 avec Quintus

23 avril 2020

Table des matières

Première partie

À propos du tuto

Ce tutoriel s'adresse à tout le monde, débutants comme développeurs aguerris : j'ai donc fait en sorte d'être le plus explicite possible. Si vous n'y connaissez rien en développement de jeux vidéos, ne vous inquiétez pas, je n'en avais moi-même jamais développé avant de découvrir Quintus !

I.1. Prérequis

Si vous vous lancez dans le développement d'un jeu, je suppose que vous avez quelques connaissances de bases pour suivre le tuto, notamment dans les domaines suivants :

— JavaScript

Et c'est tout. Si vous êtes bons en graphisme, en game design ou autres, c'est encore mieux ! Mais pour le tuto en lui-même ce n'est pas ce qui compte le plus. Donc révisez juste votre JS, respirez un bon coup et tout ira bien…

I.2. Un petit mot sur `Canvas`

Si vous n'avez encore jamais utilisé la balise `<canvas>`, je vous invite à lire quelques tutos pour comprendre le principe, la majorité de la librairie utilisant cette balise, il est intéressant de comprendre comment le dessin fonctionne même si ce n'est pas fondamental.

Le MDN propose de très bons exemples pour s'imprégner de l'API propre à cette balise :

— Dessiner avec Canvas
— Tutoriel Canvas

I.3. Et le debug dans tout ça ?!

Je pars aussi du principe que vous savez vous débrouiller pour déboguer un script. En gros vous êtes capable de vous servir des outils de développement du navigateur pour voir ce qui se passe dans la console et éventuellement au niveau du réseau.

Si vous rencontrez des soucis lors de votre développement, n'hésitez pas à demander de l'aide sur les forums ZdS, mais n'oubliez pas de fournir les informations nécessaires : erreurs dans la console, extrait du code concerné, requêtes effectuées par le navigateur si besoin, etc.

Deuxième partie

Démo

Comme l'objectif du tuto est de vous permettre de créer un jeu complet, autant vous permettre de tester les différentes étapes. Pour cela j'ai mis en ligne une démo pour que vous puissiez tester le code relatif à chaque partie et le comparer au votre. Vous pouvez aussi retrouver l'intégralité des sources de cette démo sur GitHub .

Troisième partie

Découverte de la librairie

Avant de commencer j'aimerais vous rappeler que pour l'instant Quintus est encore en développement actif. Cela signifie qu'aucune version n'est officiellement stable, même si ça tient plutôt bien la route en général.

Donc, si vous voulez passer en production, je vous conseille de faire attention à ce que vous faites : vous avez le droit, mais ne tapez pas les développeurs en cas de pépin.

Maintenant que c'est dit, en selle les enfants !

III.1. Se familiariser avec Quintus

Avant d'importer Quintus, il faut savoir quelle source vous souhaitez utiliser, étant donné qu'il s'agit d'une librairie en développement…

III.1.1. À propos de Quintus

Quintus est une librairie JavaScript, au même titre que jQuery ou Mootools (qui servent plutôt à manipuler le DOM), spécialisée dans le développement de jeux vidéos (principalement 2D, même si la 3D est prévue dans la to-do) en HTML5, via une balise `<canvas>`.

Son intérêt repose dans le fait qu'elle est modulable : vous pouvez utiliser les modules nécessaires et supprimer les autres de vos sources pour garder un code léger.

III.1.1.1. Quelques exemples

Voici quelques exemples de jeux développés avec Quintus :

— Un jeu de plateforme utilisé en démo sur le site officiel
— Pixel Game : du flat design et du pixel art en même temps

III.1.2. Différentes sources

Pour inclure Quintus dans votre script, il existe différentes sources principales :

— Utiliser le CDN fourni (au risque d'avoir rapidement une version ancienne et des fonctionnalités manquantes, voire des bugs non résolus)
— La jouer ninja en téléchargeant la dernière version du CDN, puis en déboguant à la main les éléments qui vous manquent ou qui posent problème
— Télécharger le code depuis le repo GitHub (pour avoir du code bien à jour) : c'est ce qui vous permettra de maîtriser votre code, pour éviter les mauvaises surprises

Pour éviter de donner un lien vers une version trop ancienne, je n'inclue volontairement pas d'URL pour le CDN : vous la trouverez sur le site officiel si besoin.

Rien de bien compliqué au final, il s'agit surtout de savoir ce que vous voulez utiliser, sachant que la librairie n'est pas garantie 100% stable (mais en est vraiment pas loin, je vous rassure).

III.2. Créer une page HTML basique

Bon ça, rien de bien compliqué, vous savez déjà faire je pense…

III.2.1. Un peu de HTML

Et aussi un peu de CSS, histoire d'avoir une page un peu jolie quand même.

```
```

On peut même ajouter du CSS pour que ce soit plus joli :

```
```

Pensez bien à l'appeler dans votre `<head>` : `<link rel="stylesheet" href="style.css" />`

On ajoute un canvas, auquel on applique les dimensions souhaitées (j'ai choisi 600x800, en mode portrait donc) :

```
```

Et on n'oublie pas d'insérer la librairie et notre code en fin de document (juste avant `</body>`, quoi) :

```
```

Alors, c'était dur ?

III.3. Lancer Quintus

Jusque-là ça reste simple, rien de bien compliqué, on ne touche qu'à du HTML.

Maintenant, on va passer aux choses sérieuses et préparer la librairie pour pouvoir s'en servir dans la partie suivante.

III.3.1. Initialiser la librairie et ses composants

On va donc commencer par créer notre objet Quintus, que l'on nommera `Q` et qui servira de base tout au long du tuto.

```
```

Ensuite on charge les composants dont on aura besoin. Attention, il y en a beaucoup (la plupart ne seront utiles que plus tard dans le tutoriel, mais autant les charger une bonne fois pour toute) :

```
```

À noter que vous pouvez aussi faire tout ça en une seule fois :

```
```

Vous remarquerez aussi que le plugin 2D n'est pas chargé tout à fait comme les autres. En réalité, il s'agit juste d'une syntaxe différente en JavaScript pour accéder aux propriétés d'un objet, étant donné que `2D` commence par un chiffre. Rien de bien grave, juste une particularité du langage (c'est d'ailleurs quelque chose d'assez commun dans plusieurs langages : les noms de variable commencent rarement par des chiffres)…

III.3.2. Paramétrer la librairie

Maintenant que Quintus est chargé, on va s'occuper d'appliquer quelques paramètres pour le lancer :

```
```

On va aussi charger le plugin de gestion de contrôles, qui s'occupe de gérer le clavier pour lancer des actions automatiquement (comme des mouvements) :

```
```

Et si vous souhaitez développer un jeu compatible avec les terminaux tactiles, il suffit de lancer le plugin adéquat :

```
```

Quintus est maintenant prêt à fonctionner. Alors, vous attendez quoi pour passer à la partie suivante

---

Voir le résultat

!(https://jsfiddle.net/jcqynbh5/3/)

N'hésitez pas à jeter un œil au code de la démo pour vérifier que votre code est bon.

Quatrième partie

Quelques manipulations de base

Maintenant que l'on a un beau canvas vierge et que la librairie est paramétrée, il va être temps de s'en servir, non ?

IV.1. Vocabulaire de base

Si vous développez un jeu pour la première fois, un peu de vocabulaire peut aider à ne pas vous perdre.

Je vous rassure, il ne s'agit pas d'avaler un dico : seulement d'assimiler quelques principes pour mieux comprendre ce qui se passe et ce que l'on manipule.

IV.1.1. Stage : les calques

Pour faire simple, un objet *Stage* est une sorte de calque, dans lequel vous pouvez caser à peu près tous les éléments graphiques que vous voulez (personnages, décors, textes, boutons...). Vous pouvez aussi superposer plusieurs stages pour gérer de façon indépendante plusieurs éléments (par exemple en affichant le score au-dessus du jeu).

IV.1.2. Sprite : les éléments graphiques

En gros, tous vos éléments graphiques ou presque seront des sprites. Vous pouvez donc agir sur les dimensions et la position de ces derniers. Mais vous pouvez aussi les animer (de plusieurs façons), les faire tourner, les rendre transparents, etc.

Les sprites sont insérés dans des scènes (*scene* en vocabulaire Quintus). Vous pouvez en insérer autant que vous voulez à chaque fois.

IV.1.3. UI : le composant d'interface

Ce n'est pas vraiment du vocabulaire, mais c'est un composant assez utile dans Quintus qui vous permet de créer pas mal d'éléments (descendants des sprites) comme du texte, des boutons ou des conteneurs (qui permettent d'y insérer d'autres éléments — textes et boutons entre autres).

IV.1.4. Tile : un petit parallélépipède rectangle (ou pas)

Pour rester simple et pour gérer facilement les collisions, votre espace de jeu sera composé de *tiles* (généralement des carrés, mais vous pouvez utiliser n'importe quelle forme, du moment que vous les définissez avec un peu de code). Cela permet via des fichiers externes (Quintus gère les fichiers JSON et TMX) de définir rapidement le décor et l'implantation des éléments statiques du niveau.

IV.1.5. Asset : les éléments externes

Comme en développement Web classique, les *assets* sont des éléments externes utilisés par votre script.

En l'occurrence, il peut ici s'agir de plusieurs choses :

— des images (des *tilesets*, des éléments graphiques pour les écrans intermédiaires…)
— des sons (eh oui, un jeu avec du son, c'est mieux !)
— des fichiers descriptifs (JSON ou TMX actuellement pour Quintus), pour décrire les niveaux par exemple

Quintus propose une architecture de base pour ranger ces assets :

— `images/` (vos images)
— `audio/` (vos sons)
— `data/` (vos fichiers JSON ou TMW)

Vous pouvez aussi décider de ranger vos fichiers autrement, auquel cas il suffit de passer votre configuration à Quintus lors de la création :

```
```

Et voilà, maintenant vous êtes un pro du jeu vidéo ! Comment ça non ?

Bon… au moins vous avez assimilé les bases et vous êtes capables de comprendre ce qui sera manipulé avec le code qui va arriver.

IV.2. Dessine-moi un mouton !

D'accord les enfants, j'ai compris, on passe à la pratique !

Bon, le mouton attendra, mais on va commencer par quelque chose de simple : un fond coloré (avec un dégradé, quand même) et un bouton (qui permettra de lancer le jeu en lui-même).

On va donc commencer par une page d'intro assez classique, sur laquelle on va dessiner doucement des petites choses basiques.

IV.2.1. Créer une scène

Le point de départ du jeu, c'est bien évidemment les scènes : vous pouvez les afficher et les cacher à volonté, mais surtout vous pouvez les créer comme bon vous semble et même les superposer !

On va donc commencer par afficher une scène qui nous servira d'écran de lancement, sur laquelle on va travailler tout au long du chapitre :

```
```

Ensuite on va demander à Quintus de l'afficher dès que possible :

```
```

Voilà, vous avez donc une scène (vierge pour l'instant) d'affiché sur votre canvas. Maintenant on va voir comment la remplir, histoire de lui donner des couleurs…

IV.2.2. Dessiner un fond coloré

Pour dessiner un simple rectangle jaune, on va commencer par créer un *sprite* de la taille que l'on veut (ici la taille du canvas) et lui attribuer un type *UI* (plus d'infos là-dessus dans la gestion des collisions) :

```
```

Maintenant que notre sprite est créé, il faut indiquer au canvas ce que l'on veut y dessiner (un rectangle jaune pour l'instant) :

```

```

Le contexte 2D du canvas est un objet qui nous permet de travailler sur le canvas en y dessinant des éléments. Comme il s'agit d'un contexte 2D, les éléments dessinés ne peuvent donc pas être en 3D (qui ne sera pas abordée dans ce tuto car elle n'est pas encore gérée par Quintus).

Mais jusqu'ici rien n'indique ce que l'on veut faire avec ce sprite. Il faut donc l'ajouter à notre scène :

```

```

Et voilà ! Vous avez maintenant un beau fond jaune !

IV.2.3. Ajouter un dégradé

Maintenant que l'on a un fond uni, ce serait peut-être pas mal d'y dessiner un dégradé, non ?

Et comme un veut que notre dégradé soit par-dessus notre fond jaune, il suffit de le dessiner juste après, en reprenant notre méthode `.draw()` :

```

```

En ajoutant ce code à la fin de la méthode vue un peu plus tôt, vous devriez voir apparaître un dégradé qui part du blanc (au centre) vers le jaune (ou plutôt du transparent, vu qu'on utilise notre fond jaune).

IV.2.4. Un peu de texte

Pour ajouter du texte, c'est très simple, il existe un élément `Text` que l'on peut placer automatiquement en fonction de son centre et qui peut être personnalisé (couleur, taille…).

Comme il s'agit d'un descendant de l'élément `Sprite` et que vous savez déjà ajouter un sprite à la scène, on va faire d'une pierre deux coups :

```

```

Vous voyez donc que `stage.insert()` nous renvoie l'élément ajouté : ici notre objet texte.

IV.2.5. Ajouter des boutons

Parce que c'est assez pratique d'avoir des boutons dans un jeu (surtout sur un écran de lancement), on va voir comment en créer.

Mais, plus intéressant encore, on va apprendre comment grouper des éléments ensemble (et ça ne fonctionne pas qu'avec les boutons) au moyen d'un `Container` :

```

```

Et voilà, vous savez grouper des éléments ! Non, je rigole, pour ça il manque un détail : le contenu.

On va donc se dépêcher de créer un bouton :

```

```

Avoir un bouton, c'est bien gentil, mais il faut peut-être qu'il soit utile ! Donc, on va faire en sorte de pouvoir cliquer dessus. Vous en dites quoi ?

```

```

Et maintenant, il manque un seul petit détail pour que notre conteneur soit à la bonne taille :

```

```

Et voilà, vous avez maintenant un beau bouton au centre de votre scène !

Je vous laisse le plaisir d'en ajouter d'autres (avec la même méthode) pour le fun…

> ! Attention tout de même : utilisez la méthode `container.fit()` une fois tous les boutons ajoutés, ce serait dommage de faire des calculs pour rien ou d'obtenir un résultat étrange.

IV.2.6. Et maintenant… le mouton !

Maintenant que vous avez compris comment on ajoute des éléments de base sur la scène, on va apprendre à ajouter des images.

On va donc commencer par un mouton. Et pas n'importe quel mouton : **Raymond le Super Mouton** !

FIGURE IV.2.1. – Dites bonjour à Raymond

Pour commencer, on va demander à la librairie de charger l'image dont on a besoin (celle affichée juste au-dessus, que j'ai nommée `raymond.png` et qui est rangée dans le dossier des images).

Et comme on ne veut pas afficher notre écran avant que l'image soit chargée, on va remplacer la ligne qui servait à afficher notre scène au passage :

J'en profite pour vous informer que vous pouvez suivre l'avancement du changement : Quintus calcule automatiquement le pourcentage.

Maintenant que notre image est en mémoire et que notre scène est affichée au bon moment, il n'y a plus qu'à afficher l'image. Je vous conseille de l'afficher juste après le dégradé de fond et avant le titre (même si l'afficher après ne changera pas grand chose, je vous l'accorde, mais au moins avant le conteneur si vous voulez pouvoir accéder au bouton).

IV.2.7. Bonus : un mouton qui bouge !

Comme vous avez été sages, je vais vous apprendre à animer un *Sprite* en utilisant la librairie `Tween` incluse dans Quintus !

Pour commencer, on va ajouter le composant à notre `Sprite` du mouton :

Ensuite, on va créer une fonction pour animer notre élément, qui sera appelée en boucle :

Pour mieux comprendre, voilà ce que l'on fait : on récupère notre sprite (`this` — vous devriez être familier avec ce mot-clé si vous avez fait un minimum d'objet), puis on l'anime pour le faire monter (en utilisant la position verticale de son centre) de 50 pixels en `1.5` seconde avec un effet d'*easing* (transition avec un rendu plus naturel car utilisant quelques principes de physique).

Ensuite, avec la méthode `chain()` on fait l'inverse pour qu'il retrouve sa position initiale une fois la première animation terminée.

Le dernier paramètre permet de définir une méthode de *callback* qui sera appelée une fois l'animation terminée. On va donc réutiliser notre fonction `moveSheep`…

Il ne nous reste plus qu'à appeler notre fonction pour que notre Raymond national soit animé :

```
```

Et voilà, votre mouton lévite tout seul !

Voir le résultat

Alors, il est pas beau ce mouton

Vous avez donc maintenant un bel écran d'accueil avec un bouton pour lancer le jeu (qui n'existe pas encore, mais ça viendra).

Pour avoir un aperçu du résultat, rendez-vous sur la démo .

!(https://jsfiddle.net/9uertj1r/1/)

Cinquième partie

Mettre en place un niveau

Maintenant que vous savez mettre vos éléments sur la scène, on va apprendre à créer un niveau, avec un décor et tout le tintouin !

V.1. Le décor d'abord

Pour bien commencer, on va d'abord appliquer une image de fond à notre niveau, qui va pouvoir défiler en même temps que l'on se déplace.

V.1.1. Préparer la scène

Avant de s'attaquer au décor, il ne faut pas oublier de créer la scène correspondant à notre niveau. Pour cela, rien de plus simple, c'est du déjà vu :

```
```

V.1.2. Un Repeater en fond

Le décor étant souvent (dans les jeux de plateforme) quelque chose qui se répète, on va faire simple et utiliser un composant inclus dans Quintus : un `Repeater`.

Pour cela il suffit de préciser quel asset on veut utiliser (ici une image, qu'il faudra penser à pré charger) et éventuellement la vitesse à laquelle il doit se déplacer dans la direction voulue (les deux, si besoin) :

```
```

J'ai donc utilisé une image de nuages, que vous pouvez trouver ici :

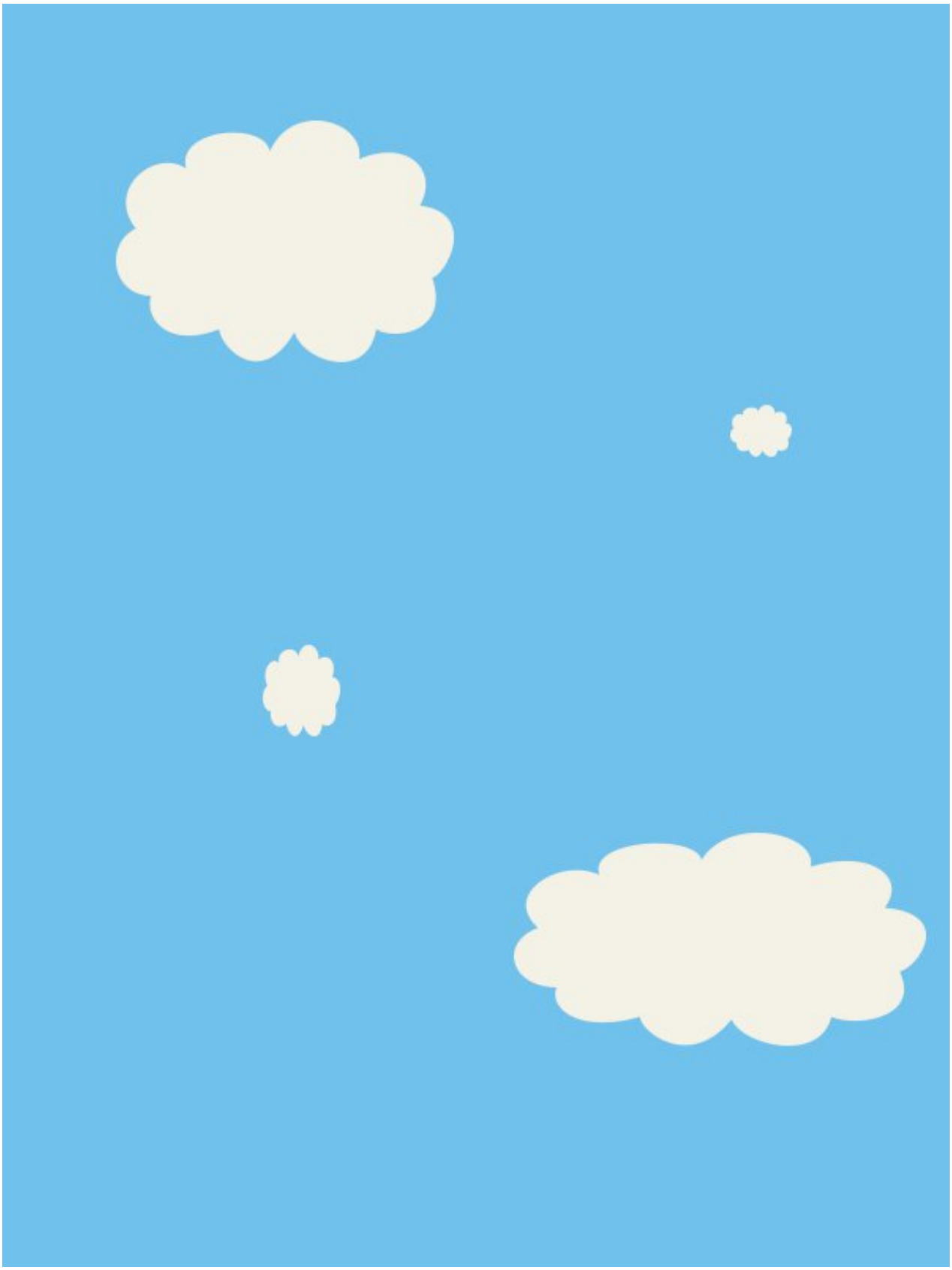

FIGURE V.1.1. – Nuages de fond

Et voilà, vous avez une jolie scène avec une image de fond qui se répète (verticalement, surtout, dans ce cas) !

V.2. Un tileset pour se déplacer

Pour se déplacer sur le niveau, il faut commencer par mettre en place les éléments importants comme les murs, le sol, etc.

Pour cela on va utiliser un tileset et configurer le jeu pour obtenir un niveau sur lequel on peut se déplacer, sauter, se cacher…

V.2.1. Qu'est-ce qu'un tileset ?

Pour faire commencer, un tileset est un ensemble de tiles. Ça, ce n'était pas trop dur à deviner.

Maintenant, le but d'un tileset, c'est quoi ? Eh bien il s'agit de définir, au moyen d'un fichier (JSON, TMX ou autre) à quoi ressemble le niveau. Ou, pour être plus précis, comment doivent être placées les tiles du niveau.

Pour faire simple (et parce qu'il n'y a pas besoin d'un logiciel spécifique), on va travailler avec un fichier JSON, qui nous permettra de placer les tiles sur une grille. Celle-ci fera 20 tiles de large (soit de tiles de 30 pixels) et 40 en hauteur (mais vous pouvez définir les dimensions que vous voulez, il suffit d'ajuster quelques nombres). Ce qui nous fait environ 800 tiles au total.

? Mais ! Ça veut dire que je dois dessiner 800 carrés à la main

Mais non, voyons ! C'est justement à ça que servent les **tilesheets** ! Eh oui, encore un mot nouveau !

En gros, une *tilesheet* correspond au mélange de votre *tileset* (défini par un fichier JSON ici) et de vos tiles (vos petits dessins qui représentent les différentes *tiles* dont vous avez besoin, regroupés sur une seule image).

Cela vous permet de ne dessiner que quelques tiles de façon individuelle : celles dont vous avez besoin au moins une fois. Par exemple, dans Super Mario, l'herbe est la même tout au long du parcours, il n'y a donc qu'une seule tile qui est utilisée plusieurs fois.

V.2.2. Créons notre tileset

On va déjà commencer par regarder à quoi ressemblent nos tiles (fichier `game-tiles.png` — pensez à le pré charger) :

FIGURE V.2.1. – Tiles du jeu

Et notre fichier JSON (nommé `game.json`) ressemblera à ça :

Contenu masqué n°1

Les nombres utilisés correspondent aux numéros des tiles (cf. paragraphe suivant), de `0` (tile vide) à `N` suivant le nombre de tiles que vous utilisez (ici `N=2`).

Une fois que vous aurez ajouté ces deux fichiers à votre fonction de pré chargement, il faudra les préparer les tiles (pour découper l'image en blocs de la bonne taille) :

Et quand on accède au niveau, il ne nous reste plus qu'à insérer un TileLayer à la scène :

Et là on va prendre un tout petit peu d'avance sur le chapitre des collisions, mais on va quand même indiquer à notre scène que les tiles que l'on utilise sont source de collision, tout en les insérant dans cette même scène :

Vous devriez maintenant avoir un aperçu du niveau. N'hésitez pas à le personnaliser (en modifiant le fichier JSON ou les tiles du PNG par exemple) pour voir ce qu'il est possible de faire.

Contenu masqué

Contenu masqué n°1

Retourner au texte.

V.3. Un joueur dans un jeu HTML5

Maintenant que l'on peut jouer, ce serait pas mal de pouvoir gérer le joueur qui sera affiché, pourra se déplacer (suivant des contrôles) et tout un tas d'autres choses…

V.3.1. Créer un joueur basique

Pour créer un joueur, il va falloir créer des sprites personnalisés. En gros, étendre la classe `Sprite`.

Heureusement, Quintus a tout prévu !

```
```

Vous aurez sans doute remarqué que l'on utilise une feuille `my_player` qui n'est pas encore définie. Plus pour très longtemps :

```
```

> *i* N'oubliez pas d'ajouter le fichier `player-sprite.png` à la liste des fichiers à pré-charger !

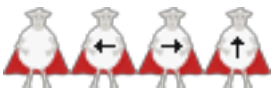

FIGURE V.3.1. – Sprite joueur

(*Oui, bon, je sais, ce n'est pas très original comme sprite, mais je fais de mon mieux, promis !*)

On peut maintenant créer notre joueur et l'ajouter à la scène :

```
```

V.3.2. Suivre le joueur grâce au viewport

Une fonctionnalité sympa dans Quintus, c'est le *viewport*. Il vous permet, pour faire simple, de centrer la vue sur une partie de la scène, sans avoir à faire de calculs par vous-même.

Un détail encore mieux : il peut être configuré pour suivre le joueur automatiquement !

On va donc faire les deux d'un coup :

```
```

Voilà, on a donc appliqué un viewport à notre scène, puis on l'a configuré pour suivre le joueur.

Le deuxième paramètre de la méthode `follow()` permet de configurer dans quelles dimensions il peut bouger : ici peu importe, mais vous pouvez remplacer `{ x: false, y :true }` par `null` si vous préférez (attention, des traitements supplémentaires risquent d'avoir lieu).

Le troisième paramètre sert à configurer les valeurs minimales et maximales de suivi. Ici on limite donc à la taille de notre niveau : pas besoin de dépasser, sauf en haut (d'où l'absence de `minY`) où on peut sauter et donc dépasser. Essayez de modifier ces valeurs ou les modifier si vous voulez vous amuser un peu…

V.3.3. Animer le joueur

Ce serait pas mal de changer le style de notre joueur quand il bouge, non ?

On va donc voir comment gérer les états d'un sprite et comment l'animer simplement.

Pour cela, on va utiliser `Q.animations()` pour définir les états d'une feuille (un sprite découpé). Mais avant il faut penser à ajouter le composant d'animation lorsque l'on initialise notre joueur :

```
```

Une fois notre feuille créée (une fois que l'on a utilisé `Q.sheet('my_player')`, donc), il suffit d'agir comme suit :

```
```

Pour l'instant, il n'y a qu'une seule frame par état, mais la logique ne change pas beaucoup pour en rajouter : on verra cela en fin de chapitre, promis.

Ensuite, il faut gérer la direction de notre joueur (et les sauts, tant qu'à faire). On va donc ajouter une méthode à notre objet (après `init()`) :

```
```

Et voilà, Raymond peut maintenant se déplacer comme il le souhaite et en plus on voit dans quelle direction il bouge !

Et voilà, vous pouvez dès à présent vous balader dans le niveau !

V.4. La fin du niveau

Pour finir le niveau, il faut bien définir ce qui va nous servir de repère. Il y a plusieurs possibilités bien sûr :

— Le score
— Un repère géographique
— La collision avec un objet déterminé

Et c'est ce dernier élément que l'on va choisir. Et comme Raymond est un mouton, quoi de plus logique que de le faire rentrer à la bergerie ?

V.4.1. Créer la bergerie

Comme pour notre joueur, il faut commencer par créer un Sprite personnalisé. Et comme j'aime bien tout nommer en anglais, on va l'appeler *Sheepfold* (bergerie, donc) :

```
```

Mais avant, il ne faut pas oublier de charger l'image correspondante et de préparer la feuille correspondante (oui, on va créer un sprite avec une seule tile) :

```
```

Et maintenant, mesdames et messieurs… la bergerie !

FIGURE V.4.1. – La bergerie

> N'oubliez pas d'ajouter l'image au pré chargement, comme toujours !

Il ne faudrait pas non plus oublier de l'ajouter à notre niveau :

```
```

V.4.2. Gérer la collision

Pour détecter si le joueur est arrivé au bon endroit, il suffit de savoir s'il entre en collision avec notre bergerie.

Pour cela, on va ajouter un écouteur dans le constructeur du joueur (l'objet `Player` dans le code) :

```

```

Notez que si vous pouvez obtenir pas mal d'infos sur les collisions. N'hésitez pas à tester !

V.4.3. L'écran de fin

Une fois notre joueur arrivé à la fin du niveau, il ne nous reste plus qu'à le féliciter, vous ne croyez pas ?

Pour cela, on va donc lancer une nouvelle scène, à laquelle on pourra passer des paramètres (vous verrez, ça peut parfois être très pratique), lors de la collision

```

```

On va aussi s'empresser de créer cette fameuse scène, avec un conteneur et un bouton pour relancer le jeu :

```

```

Bah… et notre paramètre alors

Justement, j'y venais !

Ce serait quand même sympa d'afficher un message personnalisé à notre joueur quand il finit le niveau, non ?

On va donc insérer un élément texte juste avant notre bouton, pour afficher notre fameux paramètre, qui peut être récupéré grâce à `stage.options` :

```

```

Le problème maintenant c'est que notre bouton et notre texte sont superposés. Il va donc falloir recalculer la position verticale du bouton.

Pour cela, un va créer une variable avant d'ajouter quoi que ce soit au conteneur :

```
```

Ensuite, après avoir ajouté notre texte, il va falloir recalculer tout ça :

```
```

Explication : On récupère la hauteur du texte (qui se trouve être le dernier enfant du conteneur jusqu'à présent — le bouton n'étant ajouté qu'après), on arrondit à l'entier supérieur et on ajoute une marge de 10 pixels.

Il donc n'y a plus qu'à utiliser cette variable pour positionner notre bouton en remplaçant `y: 0` par `y: button_y`. Et le tour est joué !

Et voilà, notre Raymond adoré peut maintenant rentrer tranquillement rejoindre ses copains !

Mais pour combien de temps encore… ?

V.5. Des ennemis, parce qu'il en faut

Après tout, quel jeu digne de ce nom ne comporte pas d'ennemis pour nous empêcher d'arriver au but ?

V.5.1. Un Sprite custom

On va donc commencer, comme pour les éléments personnalisés, par mettre en place les éléments graphiques :

```
```

Roulement de tambour

Mesdames et messieurs… le loup !

FIGURE V.5.1. – Sprite du loup

Oui, bon, d'accord, un tas de gribouillis ça ne ressemble pas vraiment à un loup, mais je ne sais pas dessiner un loup au repos…

On n'oublie pas de gérer les images des mouvements (attention, il y a quelques petites nouveautés) :

```
```

Puis on va pouvoir créer un Sprite customisé :

```
```

Vous pouvez aussi étendre vos propres Sprites : essayez donc `Q.Wolf.extend('Baby Wolf')`, pour voir !

V.5.2. Gérer les déplacements

Rien de bien compliqué, je vous rassure, puisque l'IA s'occupe du plus gros du travail. Il s'agit seulement d'éviter que notre loup se retrouve hors de la grille et de gérer ses animations (états).

On va donc toucher à notre objet `Wolf` pour définir la méthode `step` :

```
```

Et voilà. Pas trop perdu ? Bon, maintenant il va falloir voir ce que ça donne en vrai, quand même !

V.5.3. Les ajouter sur la grille

Pour simplifier le tout, j'ai pris des valeurs plus ou moins aléatoires, histoire d'éviter de faire trop de calculs. Mais rien ne vous empêche de faire quelques calculs chez vous, hein !

Bon, retour dans notre scène, où l'on va venir ajouter nos beaux petits loups à différents endroits et avec différentes propriétés :

```
```

Bon, bah ce n'était pas si compliqué, si ?

Maintenant, il va falloir s'occuper du plus important…

V.5.4. Gérer les collisions

Et voilà, on y est : le cœur du sujet !

On va enfin pouvoir comprendre à quoi servent les fameux `collisionMask` que l'on a utilisés, jusqu'ici sans rien dire (ni comprendre) !

Eh bien pour faire simple, ça permet de définir plusieurs choses, pour le moteur sache quoi faire :

— Si un sprite bute contre un autre (`hit`) ou lui tombe dessus (`bump.top`)
— Ce qu'il faut faire en cas de collision : empêcher le mouvement ou autoriser la superposition

Pour l'instant, c'est à peu près tout ce qu'il faut comprendre. Pour gérer tout cela plus facilement, Quintus propose quelques masques par défaut :

— `SPRITE_NONE` qui sera, je pense rarement utilisé pour vos projets
— `SPRITE_DEFAULT` qui correspond aux objets courants qui peuvent entrer en collision et bouger
— `SPRITE_PARTICLE` qui, comme son nom l'indique, sert à gérer des particules
— `SPRITE_ACTIVE`
— `SPRITE_FRIENDLY`
— `SPRITE_ENEMY` qui sert surtout pour les ennemis (comment ça c'était évident ?)
— `SPRITE_POWERUP`, pratique pour les champignons
— `SPRITE_UI`, pour tout ce qui sert à l'interface
— `SPRITE_ALL` qui regroupe tous les types précédents

Tous correspondent à de simples entiers, vous pouvez donc définir vos propres masques, par exemple en mettant dans votre code `Q.SPRITE_DOOR = 8;` pour définir le masque d'une porte.

! Passons à la pratique !

Pour gérer les collisions de notre ennemi, il va falloir rajouter quelques lignes dans le constructeur :

```
```

Et donc, la méthode `die()` de notre loup :

```
```

Et voilà, vous savez maintenant gérer des collisions et agir en conséquence !

Et voilà ! Maintenant vous avez donc un niveau sur lequel vous déplacer et des ennemis qu'il vaut mieux éviter.

Vous n'avez plus qu'à vous amuser un peu, tester tout ça et voir ce que vous pouvez en tirer.

Et voilà, notre niveau est mis en place !

!(http://jsfiddle.net/viki53/v571h9yr/3/)

Et si vous voulez gérer le score de votre joueur et afficher une UI digne de ce nom, rendez-vous dans la prochaine partie du tuto !

Et voilà ! Vous savez maintenant créer un jeu en HTML5 grâce à Quintus !

Si vous en voulez plus, n'hésitez pas à lire la doc officielle de quintus (en anglais) ou à parcourir la communauté Google+ .

Et si vous voulez aller encore plus loin, jetez donc un œil au repo sur GitHub !

Au passage, vous pouvez aussi ~~insulter~~ remercier Arius pour sa relecture attentive et consciencieuse

Liste des abréviations

IA Intelligence Artificielle. 40

MDN Mozilla Developer Network. 6

UI User Interface (Interface Utilisateur, en anglais). 1, 18, 20, 41