Chicken Slayer

SDi

HTML Game – Task4

Une image contenant texte

Description générée automatiquementUne image contenant texte, lumière

Description générée automatiquementUne image contenant texte, lumière

Description générée automatiquement

Romain Sierro

Table des matières

[1. Introduction 2](#_Toc70115695)

[2. Développement 2](#_Toc70115696)

[2.1 Architecture 2](#_Toc70115697)

[2.2 Instructions 3](#_Toc70115698)

[2.3 Implémentations & Fonctionnalités 4](#_Toc70115699)

[2.3.1 asset 4](#_Toc70115700)

[2.3.2 Client.html: 4](#_Toc70115701)

[2.3.3 Client.js 5](#_Toc70115702)

[2.3.4 Phaser.min.js 5](#_Toc70115703)

[2.3.5 Game.js 5](#_Toc70115704)

[3. Conclusion 7](#_Toc70115705)

[3.1 Problèmes rencontrés 7](#_Toc70115706)

[3.2 Améliorations 7](#_Toc70115707)

[3.3 Conclusion 7](#_Toc70115708)

[7](#_Toc70115709)

# Introduction

Le jeu que j’ai choisit de réaliser est un « endless runner », le but est de sauter de plateforme en plateforme et éviter de tomber dans le vide. Le joueur contrôle un humanoïde d’origine inconnue qui a visiblement un problème contre les poulets. Il peut exécuter des sauts simples ou des doubles sauts et tirer sur les poules pour augmenter le score.

Le jeu a été développé sur WebStorm et utilise Phaser pour toute la gestion de la 2D du jeu. Phaser est gratuit et contient beaucoup d’exemple ce qui m’a grandement aidé dans le projet.

# Développement

## Architecture

Button state changed

Button pressed



MQTT Server



Bu

Thingy 1 & 2

On peut voir que le fonctionnement de notre code n’est pas des plus complexe. Les thingies communiquent avec le raspberry via le bluetooth. Toute la communication raspberry-Thingy a été reprise de la task2. Lorsque le bouton d’un thingy sera pressé ou relâché, le raspberry indiquera au serveur qu’un état a été modifié. Une variable globale sera ensuite modifiée pour indique au programme quel action réaliser.

## Instructions



Figure 1: Ecran d'accueil du jeu¨

Une fois sur la page du jeu, on peut voir les instructions de jeu (quel bouton fait quoi). Pour pouvoir continuer, un simple appui sur l’un des Thingies suffit.

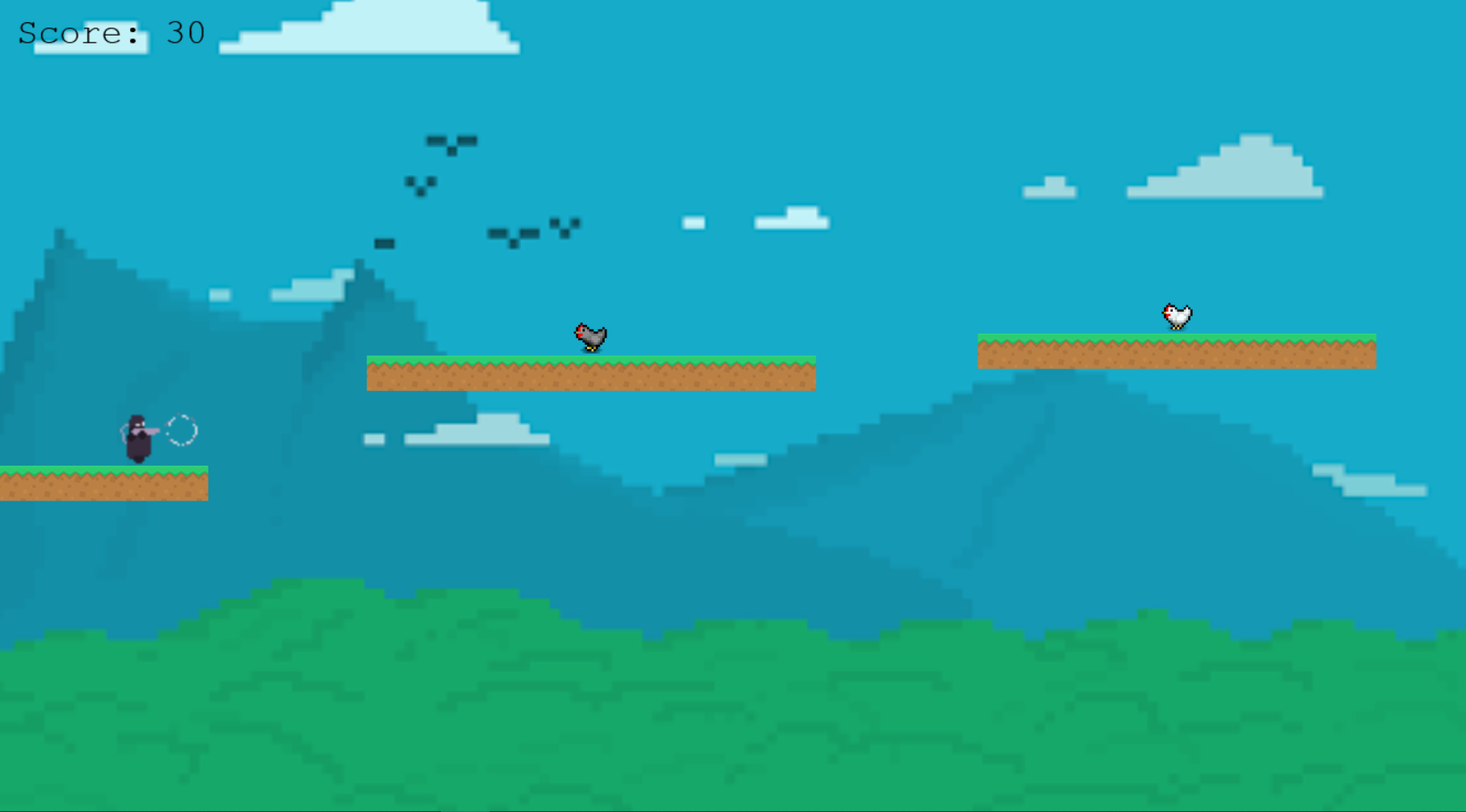


Figure 2: Capture du jeu

Dans cette capture, on peut apercevoir le joueur en fin d’animation de tir, il reste un résidu de tir. Un poulet blanc a été généré sur une plateforme. On peut aussi voir le score du joueur en haut à gauche de l’écran. En pressant le Thingy violet, le joueur va sauter. On a la possibilité de réaliser un double saut, mais attention à ne pas sauter trop loin. Le Thingy vert permet de tirer. Le but est de tirer à proximité d’un poulet afin de remporter 10 points.

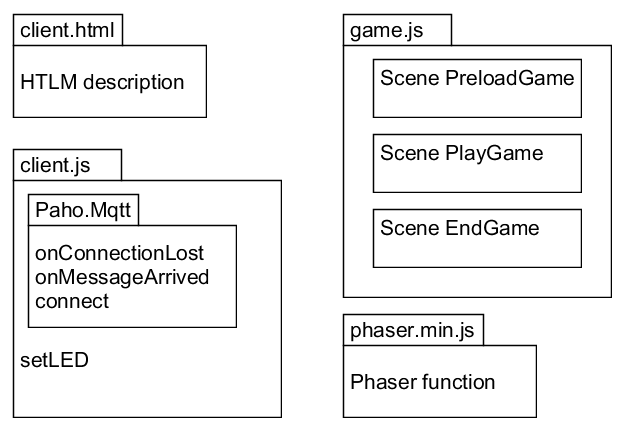
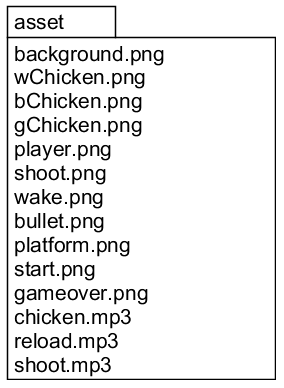
Si le joueur tombe dans le vide, la partie se termine.



Figure 3: Ecran de fin

Dans le cas où le joueur est tombé dans le vide, l’écran de game over apparait et affiche le score du joueur. Si l’on veut relancer une partie, il suffit de presser le bouton d’un Thingy.

## Implémentations & Fonctionnalités



### asset

Ce dossier contient tous les sprites, images et fichiers audios utilisés dans notre application.

### Client.html:

Client.html est le fichier qui inclus tous les autres au jeu. Afin de pouvoir lancer une partie, il faut chercher dans la barre d’adresse (L’adresse peut changer selon l’emplacement du jeu sur votre machine !) :

« *localhost :8080/task4/sdi-task4-TheGame/****client.html****»*

### Client.js

*Client.js* instancie un « client = new Paho.MQTT.Client ». Nous pourrons ainsi accéder aux méthodes *client.onConnectionLost, client.onMessageArrived* et *client.connect.* Ce code nous a été donné en début de projet, mais il a été modifié. Afin d’atténuer la latence entre le moment où l’on appuie sur le bouton d’un thingy et le moment où le personnage du jeu réalise l’action*,* il faut modifier le port de l’instance du client qui est de base à 80 et mettre le 443. Il faut aussi activer la connexion SSL dans la fonction *connect*:

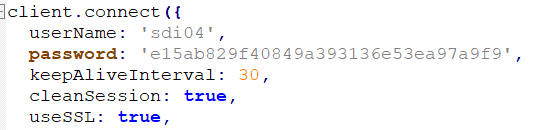


Figure 4: Partie du code de la fonction connect

Ce fichier déclare en global 2 variables qui passent à *true* quand un bouton est pressé et *false* quand il est relâché. La valeur des variables est ensuite lue dans le fichier *game.js.*

### Phaser.min.js

*Phaser.min.js* comporte toutes les fonctions et la physique de phaser. Il a été téléchargé depuis le site phaser.

### Game.js

*Game.js* est le fichier qui comporte tout l’affichage et la gestion du jeu. Il est composé de 3 classes/scènes : *PreloadGame, PlayGame et EndGame.*

Chaque classe hérite de Phaser.Scene et contient les fonctions :

* *preload() :* Précharge les images, les sprites ainsi que les fichiers audio utilisés. Pour les sprites, c’est dans cette fonction que nous donnons les dimensions afin de les découper.
* *create() :* Créer les animations et affiche les sprites, ajoute les images et différents textes à l’écran. C’est aussi dans cette fonction que sont déclarés les différents objets utilisés pour le jeu (joueur, plateforme) et que la physique ainsi que les collisions sont créées. L’ajout des sons et des input (souris, clavier) se font aussi dans cette fonction.
* *update()*: Cette fonction est appelée par défaut 60 fois par seconde. C’est dans cette fonction que l’on déplace les différents éléments du jeu.

**PreloadGame :**

Cette scène s’occupe principalement de charger tous les sprites, images et sons qui seront utilisés dans notre jeux. Elle crée les animations des sprites et affiche l’écran d’accueil ainsi que les commandes du jeu. Elle attend ensuite qu’un bouton soit appuyé pour lancer le jeu

**PlayGame :**

C’est la scène principale dans laquelle se déroule le jeu à proprement parler. Il comporte :

* Un joueur « player ». Il possède différentes animations : la course, le saut ainsi que le tir. De la gravité lui est appliquée pour le faire retomber après un saut. Il a les collisions activées seulement avec les plateformes.
* Le tir du pistolet « bullet ». Lors d’un tir, je ne voulais pas agrandir la hitbox du joueur afin d’éviter qu’il puisse se rattraper sur le bord d’une plateforme en tirant et ainsi éviter la fin de partie. J’ai donc placer le tir en permanence devant le joueur et le rend visible au moment ou le joueur tir. Il possède la même physique que le joueur, mais il a en plus les collisions avec les poulets. C
* Les plateformes. Elles sont réparties dans 2 groupes. *platformGroup* qui contient les plateformes actives du jeu, c’est-à-dire les plateformes utilisées à l’écran. Une fois utilisées, elles sont recyclées en étant misent dans la *platformPool.* Cette technique nous permet de ne générer que quelques types de plateformes et de les réutiliser à l’infini.
* Les poulets fonctionnent comme les plateformes. Il y a le *chickenGroup* dans lequel les poulets actifs sont placés, et la *chickenPool* qui recycle les poulets sortis de l’écran. A chaque création de plateforme, il y a 40% de chance qu’un poulet y soit placé dessus. Il existe 3 types de poulets différenciables par leur couleur et leur animation mais ils apportent tous le même nombre de points (10 points).
* Le score en haut à gauche de l’écran. Il s’incrémente à chaque fois qu’un poulet est touché par un tir, mais seulement si celui-ci est visible et donc que le joueur a tiré.

Dans sa fonction *update,* on contrôle si un poulet ou une plateforme est sorti de l’écran pour le recycler dans leur pool. On ajoute une nouvelle plateforme au jeu si la distance minimale avec la dernière plateforme ajoutée a été respectée. On contrôle si le bouton d’un thingy a été pressé.

Dans le cas où le bouton du thingy violet a été activé, la fonction *jump()* est appelée. Cette fonction test si le joueur a déjà fait un double-saut et si se n’est pas le cas, joue l’animation de saut et applique un vecteur vertical au *player* ainsi qu’au *bullet*.

Si c’est le thingy vert qui est activé, la fonction de tir *attack()* est appelée. On va donc rendre visible *bullet* et jouer son animation de tir ainsi que celle du *player*. Une fois que l’animation est terminée, une fonction callback est appelée afin de rendre à nouveau *bullet* invisible et de remettre l’animation de course pour notre *player*.

Durant la partie, des sons sont joués. Lorsque le joueur tir un son de fusil retentit. Si un poulet est touché par le tir, on peut l’entendre piauler. Si le joueur ne tire pas, on peut aussi entendre le son du rechargement du fusil.

**EndGame :**

Une fois que le joueur est tombé, on détecte que sa position n’est plus dans les limites de la scène et donc on appelle cette scène *EndGame.* Ici on affiche le game over et le score du joueur. Pour recommencer la partie, il faut presser un bouton du thingy et cela rappellera la scène *PlayGame*.

# Conclusion

## Problèmes rencontrés

La création des plateformes m’a fait perdre beaucoup de temps. Les premières plateformes étaient bien mais par la suite leur hitbox était minuscule. J’utilisais la physique de phaser donnée dans un exemple trouvé sur leur site, puis en fouillant j’ai pu trouver un autre fichier *Phaser.min.js* à inclure. En utilisant le nouveau fichier, les plateformes apparaissent correctement. Ce problème est totalement dû au manque de connaissance en phaser et en javascript. Je pensais que mon code n’était pas correct alors qu’il fallait utiliser une autre version de la physique de phaser.

L’adaptation des hitbox de mon personnage a été compliquée. Finalement, j’ai opté pour la solution de laisser le tir en invisible en permanence devant le personnage et de le rendre visible seulement quand je tir. Cela a aussi faciliter la détection de collision avec un poulet, car je ne veux l’activer que lorsque le personnage tir. Ensuite la collision se générait plusieurs fois. Donc au lieu de donner 10pts, le joueur recevait entre 100 et 140 points. Il a fallu contrôler si le poulet en collision était déjà en collision précédemment.

La prise en main de phaser et de javascript : Etant donné le temps pour réaliser un jeu et le fait que ce soit notre premier programme dans ce langage, il a fallu choisir un jeu assez simple. De plus, phaser change entre les versions et la plus grosse partie de la tâche a été la recherche d’information et l’étude de la documentation phaser.

## Améliorations

Le jeu étant beaucoup trop simple, il faudrait pouvoir accélérer le jeu en fonction du temps de la partie. Plus vous allez loin, plus le jeu accélère. Cela donnerait plus de challenge et rendrait le jeu plus addictif.

Les poulets apparaissent au centre d’une plateforme. Ce serait bien de pouvoir aussi en faire tomber depuis le haut de l’écran. Il faudrait donc être plus attentif pour faire des points, ce qui augmenterai la complexité du jeu.

L’ajout de bonus pourrait rendre le jeu beaucoup plus fun. Un jetpack pourr voler et éviter de tomber pendant un moment, un item qui déclencherai un spawn excessif de poulet qui tomberait de partout.

L’amélioration du HUD ainsi que des menus donnerait au jeu un aspect plus finit et plus professionnel.

## Conclusion

Pour finir je dirai que ce projet m’a surtout appris à étudier la documentation d’un nouveau langage. La partie communication avec les thingy ayant été rapidement terminée, tout mon temps s’est écoulé sur la compréhension de phaser. Mais réaliser un jeu reste très ludique et amusant. De plus, incorporer la communication avec les thingies nous permet d’utiliser plusieurs tâches réalisées précédemment ce qui est plutôt satisfaisant.