

---

# **Développement d'outils ou matériel d'enseignement de l'informatique**

## **Développement d'un outil informatique démontrant visuellement la perception d'une liste pour un ordinateur**

*Collège du sud, Travail de maturité Version intermédiaire*

**Grégoire Geinoz**

janv. 14, 2022



---

## Table des matières

---

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Présentation de l’outil . . . . .	1
1.2	Intêret de l’outil . . . . .	1
1.3	Technologies utilisées . . . . .	2
1.4	Configuration matérielle requise pour utiliser l’outil . . . . .	2
1.5	Connaissance requises pour utiliser l’outil . . . . .	2
1.6	Connaissances requises pour comprendre le fonctionnement de l’outil . . . . .	2
<b>2</b>	<b>Aspects pédagogiques</b>	<b>3</b>
<b>3</b>	<b>API</b>	<b>5</b>
<b>4</b>	<b>Scénarios d’utilisation</b>	<b>7</b>
<b>5</b>	<b>Fonctionnement du projet</b>	<b>9</b>
5.1	L’utilisation de Phaser . . . . .	9
5.2	Système d’animation . . . . .	9
<b>6</b>	<b>Indices and tables</b>	<b>15</b>



### 1.1 Présentation de l'outil

L'objet de ce travail de maturité consiste en la programmation d'un outil permettant d'accompagner un professeur souhaitant développer une compréhension intuitive de la notion de liste à ses élèves. Il permet d'avoir une vision claire de ce qu'est une liste en programmation et quelles sont les manipulations qu'un ordinateur peut effectuer sur celle-ci.

Concrètement, une liste est représentée comme étant un "jeu de cartes". Chaque élément de la liste compose une carte avec sa valeur affichée d'un côté de la carte. Ainsi, tous les éléments de la liste forment à eux, le jeu de cartes au complet. Certains algorithmes requièrent des variables externes à la liste, ces variables sont créées via des appels de fonctions sur une API (à expliquer) et sont représentées dans une zone de l'écran spécifique. Cependant, des variables "spéciales", qui n'ont comme seul rôle, d'être un index qui parcourt les éléments d'une liste (dans une "boucle for" par exemple), peuvent également être créées et sont représentées comme des flèches qui pointent sur l'élément de la liste qui correspond à l'index contenu dans la variable.

Le but étant de démontrer les actions opérées par un ordinateur sur une liste, les interactions entre les éléments de la liste entre eux ou avec une variable sont animées. Ainsi, pendant l'exécution d'un programme, il fera discrètement appel à l'API afin qu'une animation se déclenche et montre sur l'écran le comportement de la liste.

### 1.2 Intérêt de l'outil

Les listes et les algorithmes sont des sujets importants à comprendre et à maîtriser dans le cadre de l'apprentissage de la programmation. Actuellement, ces concepts sont amenés aux étudiants et expliqués de manière bien trop abstraite. Il est nécessaire de comprendre ce que représente une liste pour un ordinateur et comment il peut interagir avec ; ainsi que de comprendre le fonctionnement des algorithmes, étape par étape.

Des plateformes d'enseignement de l'informatique tel que "code.org" ont un concept similaire au projet de ce travail de maturité : Développer une compréhension intuitive de concepts par l'expérimentation. Cependant, leur projet cible un public jeune pour leur inculquer des bases de programmations en posant brique par brique des blocs de code afin d'amener une cible jusqu'à son objectif. Le projet envisagé dans le cadre de ce travail de maturité cible un public plus mature capable de pousser un raisonnement plus en profondeur, en effet les listes sont une notion plus abstraite car

elles sont des conteneurs de valeurs quelconques. De plus, ce projet vise à enseigner des notions très spécifiques de la programmation : les listes.

## **1.3 Technologies utilisées**

Le projet repose sur 2 technologies principales : Javascript et HTML5. Cependant pour alléger la quantité de travail, un framework de jeu 2D, Phaser, est utilisé pour gérer tout ce qui relève de l'affichage d'images, de la gestion de scènes et d'événements et autres qui impliquent la gestion d'objets (les cartes et les variables). Enfin, le projet étant écrit en Javascript principalement, un interpréteur de code peut être nécessaire si le langage étudié par l'outil est différent.

## **1.4 Configuration matérielle requise pour utiliser l'outil**

Afin que l'utilisation de l'outil soit la plus accessible possible, les langages de programmation du Web ont été utilisés pour le développer. Ce qui signifie que pour utiliser l'outil, seul un accès à un ordinateur opérationnel étant doté d'un navigateur internet, et d'une connexion, suffit.

## **1.5 Connaissance requises pour utiliser l'outil**

L'outil fait appel aux notions de variables ; éléments qui constituent la liste, ainsi qu'aux opérations de bases entre variables numérique (calculs, affectations de valeurs) et la notion de condition entre les variables utilisant des opérateurs de conditions.

## **1.6 Connaissances requises pour comprendre le fonctionnement de l'outil**

Afin de comprendre comment l'outil fonctionne, il est nécessaire de comprendre le code qui le compose. C'est à dire qu'il faut avoir un niveau de base qui couvre tous les fondamentaux du Javascript. Le code étant largement commenté, il n'est pas forcément nécessaire de savoir utiliser le framework Phaser pour comprendre le code source.

## CHAPITRE 2

---

Aspects pédagogiques

---





## CHAPITRE 3

---

API

---



## CHAPITRE 4

---

### Scénarios d'utilisation

---



# CHAPITRE 5

---

## Fonctionnement du projet

---

Le projet repose sur plusieurs grands aspects, qui permettent une répartition du code en différents systèmes.

### 5.1 L'utilisation de Phaser

#### 5.1.1 Système de gestion des cartes

#### 5.1.2 Système d'évènements

#### 5.1.3 Gestion de la scène et de la boucle principale

### 5.2 Système d'animation

Le système d'animation permet au développeur de créer des schémas d'animation. C'est à dire que, par exemple, le développeur peut aisément créer une animation qui engendre le déplacement simultané ou séquentiel d'une ou plusieurs cartes.

#### 5.2.1 Principe fondamental

Naïvement, on pourrait penser qu'il suffit que chaque carte possède un attribut « animation » qui contient les informations nécessaires à décrire une animation, par exemple, de déplacement :

```
const card = {...};

card.animation = {
  toAnimate: true,
  type: "movement",
```

(suite sur la page suivante)

(suite de la page précédente)

```
// Coordonnées où la carte doit se déplacer.
x: ...,
y: ...
};
```

Et qu'ainsi, dans la fonction « update » gérée par Phaser, une « boucle for » parcourt toutes les cartes et effectue l'animation qui lui est attachée, si celle-ci doit être animée :

```
update() // Exécutée 60 fois par seconde par Phaser.
{
    // Parcourt toutes les cartes.
    for (let i = 0; i < nbCards; i++)
    {
        // Stocke la carte courante dans une variable.
        const currentCard = lstCards[i];

        if (currentCard.toAnimate)
        {
            // Gérer l'animation.
        }
    }
}
```

Or, cette manière de procéder comporte un gros désavantage. En effet, elle ne laisse au programme que la possibilité de gérer toutes les animations en même temps, ce qui signifie que si le développeur souhaite jouer des animations dans un certain ordre, il doit attendre que l'animation précédente soit terminée avant de configurer l'animation suivante dans une ou plusieurs cartes. Ce n'est pas viable pour gérer une quantité importante d'animations qui s'exécutent à la suite.

C'est pour cela qu'un véritable système d'animation est nécessaire. Concrètement, la scène principale possède un attribut « animationQueue ». Il s'agit d'une liste initialement vide, qui stocke les animations les unes à la suite des autres. Ce principe simple permet de conserver l'ordre dans lequel les animations doivent être jouées ; selon l'ordre d'apparition dans la liste. Ce procédé nécessite donc également la création d'un objet « animation », qui sera l'objet stocké dans la liste animationQueue :

```
function moveCard(targetCard, ...)
{
    const animation = {
        // Type de l'animation.
        type: "movement",

        // Contient une référence à la carte qui doit subir l'animation.
        card: targetCard,

        // Coordonnées où la carte doit se déplacer.
        x: ...,
        y: ...,

        // Devient vrai quand l'animation est arrivée à son terme.
        isFinished: false
    };

    // La liste d'animations relative à la scène principale.
    animationQueue.push(animation);
}
```

(suite sur la page suivante)

(suite de la page précédente)

```
}
```

De cette manière, la fonction « update », gérée par Phaser, peut accéder à la liste d'animations et les jouer dans l'ordre. Ainsi, une fois que la première animation de la liste est arrivée à son terme, il suffit de la supprimer pour pouvoir traiter l'animation suivante.

```
update() // Exécutée 60 fois par seconde par Phaser.
{
    // Si la liste d'animations n'est pas vide.
    if (animationQueue.length > 0)
    {
        // Stocke la première animation de la liste.
        const currentAnimation = animationQueue[0];

        // Identifie la type de l'animation.
        switch (currentAnimation.type)
        {
            // Type: Déplacement
            case "movement":
                // Récupère la référence de la carte concernée par l
                ↪ 'animation.
                const card = currentAnimation.card;

                // Exécution de l'animation...
                // ...
                // ...

                if (/* Animation terminée */)
                {
                    currentAnimation.isFinished = true;
                }
                break;
            // ...
        }

        // Si l'animation courante est terminée.
        if (currentAnimation.isFinished)
        {
            // Retire la première animation de la liste.
            animationQueue.shift();
        }
    }
}
```

Cependant, comme chaque objet « animation » décrit le comportement d'une unique carte, il n'est pas possible d'exécuter plusieurs animations simultanément. Pour contourner ce problème, le système est programmé pour jouer toutes les animations à la suite qui ne sont pas interrompues par un objet « animation » de type « break » (« pause » ou « rupture » en anglais). Lorsque ce type est rencontré, le système s'assure que toutes les animations précédentes soient terminées avant de jouer le bloque d'animations suivant :

```
update() // Exécutée 60 fois par seconde par Phaser.
{
```

(suite sur la page suivante)

(suite de la page précédente)

```
// Parcours les animations de la liste (animationQueue).
for (let i = 0; i < animationQueue.length; i++)
{
    // Stocke l'animation courante de la liste.
    const currentAnimation = animationQueue[i];

    // Identifie le type de l'animation.
    if (currentAnimation.type == "movement")
    {
        // Récupère la référence de la carte concernée par l'animation.
        const card = currentAnimation.card;

        // Exécution de l'animation...
        // ...
        // ...

        if (/* Animation terminée */)
        {
            currentAnimation.isFinished = true;
        }
    }
    // ...
    else if (currentAnimation.type == "break")
    {
        // Si le type break est le premier élément de la liste.
        if (i == 0)
        {
            // Retire le premier élément de la liste.
            animationQueue.shift();
        }

        // Met fin à l'exécution de la boucle for.
        break;
    }

    // Si l'animation courante est terminée.
    if (currentAnimation.isFinished)
    {
        // Supprime l'élément courant de la liste.
        animationQueue.splice(i, 1);
    }
}
}
```

Bien que l'emploi du « switch » soit pratique pour identifier le type d'animation, il ne peut pas être utilisé dans le cas présent car l'instruction `break;` du type « break » doit mettre fin à la « boucle for » et non à l'instruction « switch ».



## 5.2.2 Déplacement des cartes

Le framework Phaser permet de déplacer ses objets par un procédé qui s'appelle le « tweening ». Ce procédé permet en effet de déplacer un objet d'un point A à un point B automatiquement et fluidement. Appliquer du tweening sur une carte exécute un code asynchrone modifiant les propriétés « x » et « y » de la carte concernée afin de la faire se diriger vers le point souhaité :

```
// "this" fait référence à la scène de Phaser.
this.tweens.add({
  // Objet concerné par l'animation.
  targets: image,

  // Coordonnées de la destination.
  x: ...,
  y: ...,

  // Durée de l'animation.
  duration: ...,

  // Type d'accélération.
  ease: 'Power1',

  // Fonction appelé à la fin de l'animation
  onComplete: function() {...}
});
```

Cependant, ce principe ne respecte pas le fondement du système d'animation développé précédemment, qui consiste à stocker dans une liste toutes les animations créées, afin de pouvoir les jouer dans un ordre défini, simultanément ou non. En effet, le tweening proposé par Phaser déclenche une animation au moment-même où le tweening est créé, ou éventuellement avec un délai mesuré en microsecondes. De plus, le système de « callback » (appel d'une fonction à la fin de l'animation) rajoute une complexité supplémentaire dans la gestion des animations. Par conséquent, il est préférable que les déplacements des cartes ne relèvent pas de la responsabilité de Phaser.

Il est donc nécessaire que le programme gère ce type d'animation lui-même. Pour cela, l'objet « animation » de type « movement » doit faire appel à des notions de trigonométrie élémentaire afin de calculer l'angle en radian entre le point de départ et le point d'arrivée du déplacement :

```
const animation = {
  // Type de l'animation.
  type: "movement",

  // Autres propriétés de l'animation.
  // ...

  // Angle entre le point de départ et d'arrivée.
  directionAngle: /* Angle en radian */
};
```

Pour obtenir l'angle de la direction, il suffit de calculer l'arc tangente du quotient de la différence d'ordonnée sur la différence d'abscisse entre le point de départ et de d'arrivée :

$$\alpha = \arctan\left(\frac{\Delta y}{\Delta x}\right)$$

Javascript possède nativement un objet « Math » qui contient une fonction « atan2(y, x) » qui prend en paramètres les coordonnées « x » et « y » du point d'arrivée relativement au point de départ (0; 0) et retourne l'arc tangente formé par

le quotient de « y » sur « x ». L'avantage de cette fonction est qu'elle gère elle-même les cas où « x » ou « y » serait égal à 0 :

```
const animation = {  
  // Type de l'animation.  
  type: "movement",  
  
  // Autres propriétés de l'animation.  
  // ...  
  
  // Angle entre le point de départ et d'arrivé.  
  directionAngle: Math.atan2((animation.y - card.futureY),  
                             (animation.x - card.futureX));
```

Comme l'angle de la direction dans laquelle la carte doit se déplacer n'est calculé qu'une seule fois au moment de la création de l'animation, la fonction « update » de Phaser n'a plus qu'à actualiser les coordonnées de la carte en fonction de l'angle. Les fonctions trigonométriques Sinus/Cosinus permettent d'obtenir

### 5.2.3 Retournement des cartes

## CHAPITRE 6

---

### Indices and tables

---

- `genindex`
- `search`