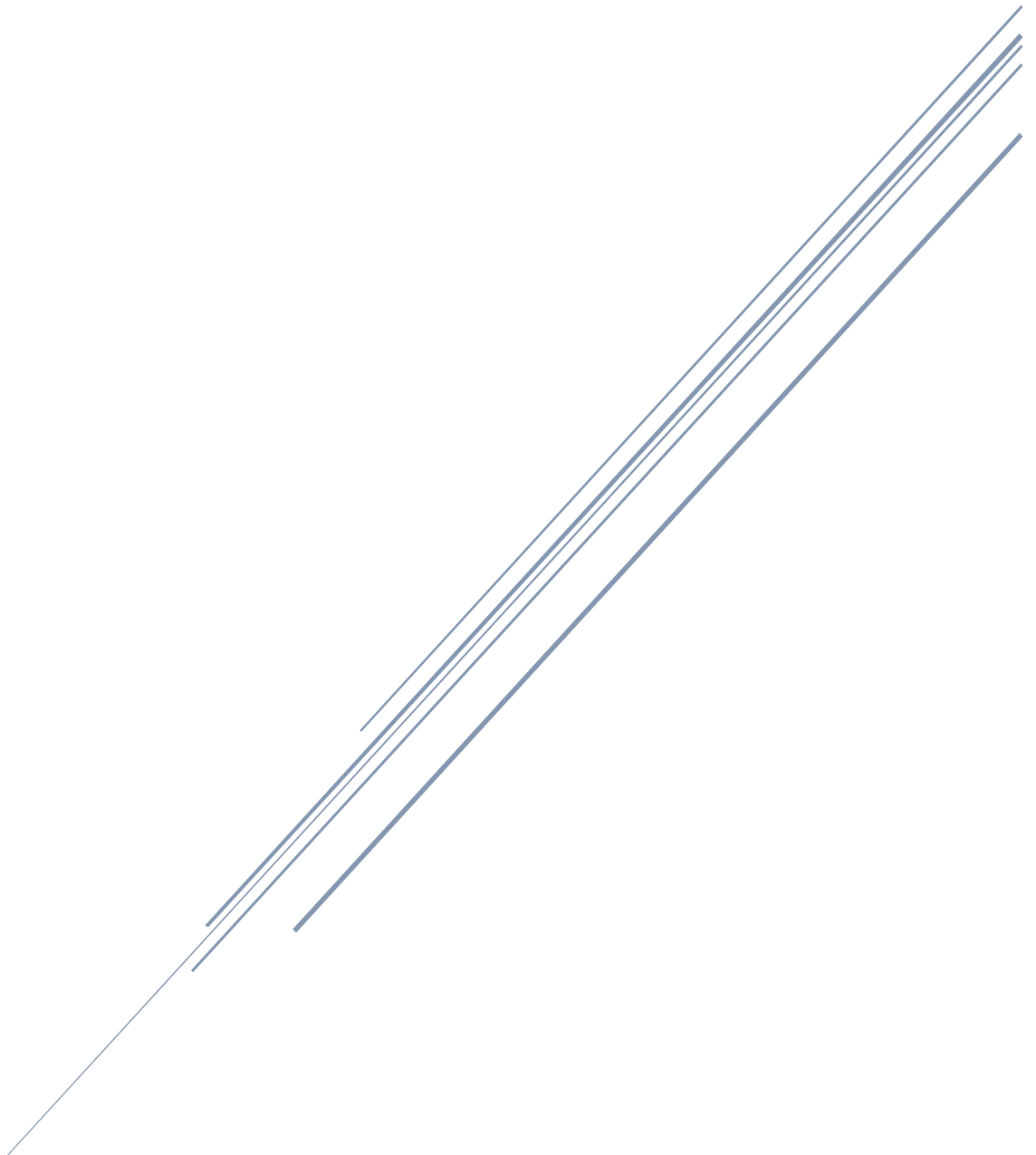


P_BULLES-SNAKE

Ethan Schafstall



CID2B, ETML
07.11.2023-09.01.2024

Table des matières

1	Introduction.....	1
2	Analyse	2
2.1	Déplacement du serpent.....	2
2.2	Interaction pomme/serpent.....	3
2.3	Collision serpent/bordure	4
2.4	Interaction serpent tête/corps.....	5
3	Implémentation.....	7
3.1	Classes	7
3.2	Constants.....	7
3.3	Variables var et let.....	8
3.4	Fonctions fléchées.....	8
3.5	Fonctions de Manipulation de Tableau.....	9
3.6	Mapping.....	10
3.7	Rest.....	11
4	Conclusion	12
4.1	Tests.....	12
4.2	Conclusion	13
4.2.1	Amélioration.....	13
4.2.2	Avis	14
5	Annexe et Références.....	15
5.1	Annexes	15
5.2	Figures	15

1 Introduction

Le projet P_Bulles a comme bût la familiarisation, et l'apprentissage de javascript. Pour attendre ces objectives il est demandé de créer le célèbre jeux Snake en application web en utilisant l'**HTML**, le **CSS**, et le **javascript** avec l'environnement **node.js**

Quelques objectives à apprendre :

- Utiliser const et let mais jamais var
- Créer des classes (Snake, Apple, etc)
- Utiliser les fonctions fléchées (sauf dans le cas des méthodes d'une classe)
- Les modules : import / export
- Utiliser l'opérateur rest pour décomposer un tableau
- Diverses structures de données
- Gestion mémoire
- Génération aléatoire

Durant le projet c'est aussi fortement recommandé de créer une aide-mémoire (cheetsheet) personnel avec des infos/astuces qui nous servirais pour de futurs modules durant le long de notre formation.

2 Analyse

Pour la réalisation de notre jeu Snake, nous devons effectuer une analyse du fonctionnement du jeu afin de le recréer. Le jeu Snake peut être divisé en quatre éléments principaux :

1. **La grille de jeu** : La grille de jeu sert d'environnement sur lequel le serpent et la pomme vont être positionnés. Elle délimite la zone dans laquelle les deux autres éléments ont le droit d'exister.
2. **La pomme** : La pomme représente l'objectif du jeu, le but que le joueur doit atteindre. Dès que le serpent mange la pomme, une nouvelle pomme apparaît de manière aléatoire sur la grille du jeu, prête à être consommée à nouveau.
3. **Le serpent** : Le serpent est l'entité qui représente l'utilisateur et constitue le véhicule avec lequel l'utilisateur peut interagir dans le monde du jeu. Le serpent se déplace de manière automatique, mais l'utilisateur a l'influence sur la direction de son déplacement.

2.1 Déplacement du serpent

La première chose à analyser est le déplacement du serpent. À chaque frame (mise à jour du visuel), le serpent se déplace d'un carré selon la direction de sa tête. Pendant ce déplacement, deux événements se produisent :

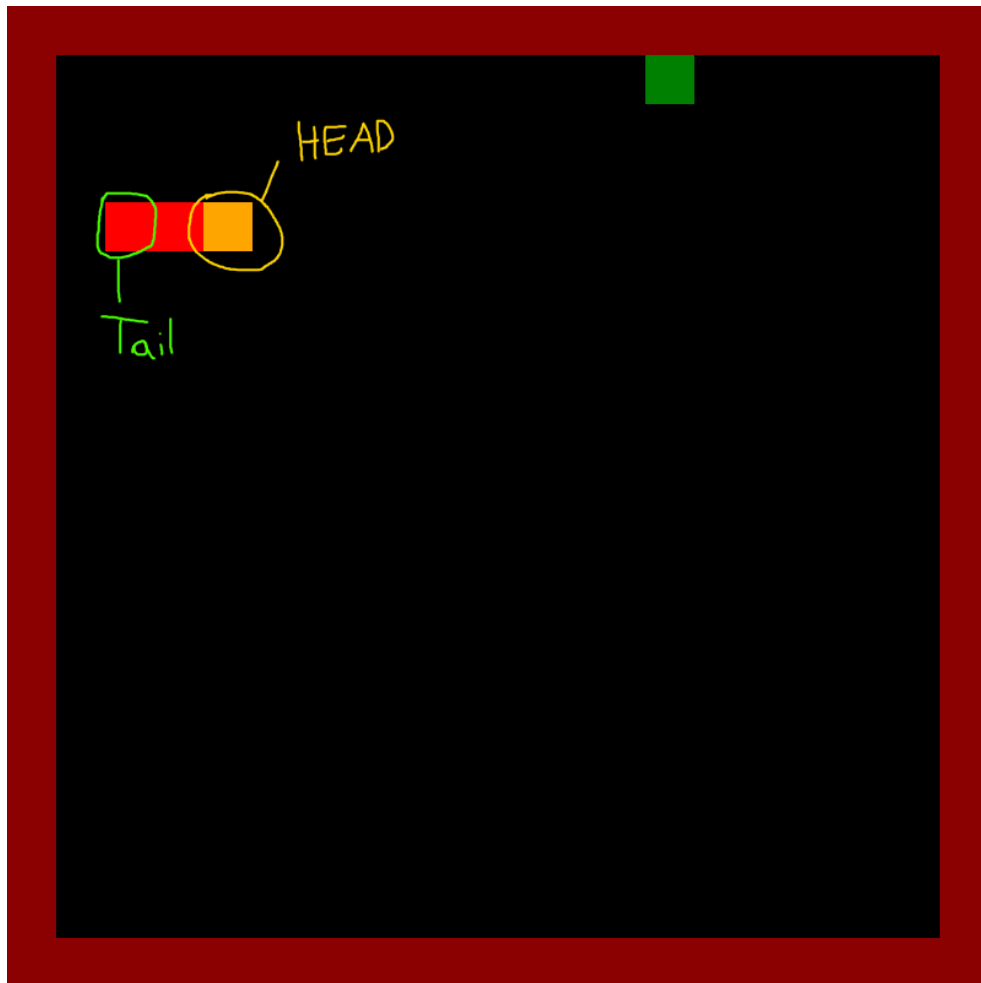
1. Un nouveau carré remplace la tête.
2. Le carré qui représente sa queue disparaît.

Avec ces informations, on peut comprendre que seuls le premier et le dernier carré du serpent changent à chaque frame, tandis que toute la partie du milieu reste statique. Le serpent ne se déplace pas réellement, mais c'est plutôt une illusion de déplacement. À chaque frame, seuls deux carrés changent de couleur, mais chaque carré du serpent a sa propre position sur la grille de jeu qui ne change pas, excepté lors du changement de couleur.

Ainsi, deux informations sont essentielles pour la réalisation du jeu :

1. Les coordonnées de chaque carré qui représente le serpent.
2. La direction de la tête.

La suppression de la queue est automatiquement effectuée à chaque frame et n'est pas influencée directement par la direction. Il est simplement utile de connaître sa position.



Annexe 1 : Déplacement du serpent – GIF

2.2 Interaction pomme/serpent

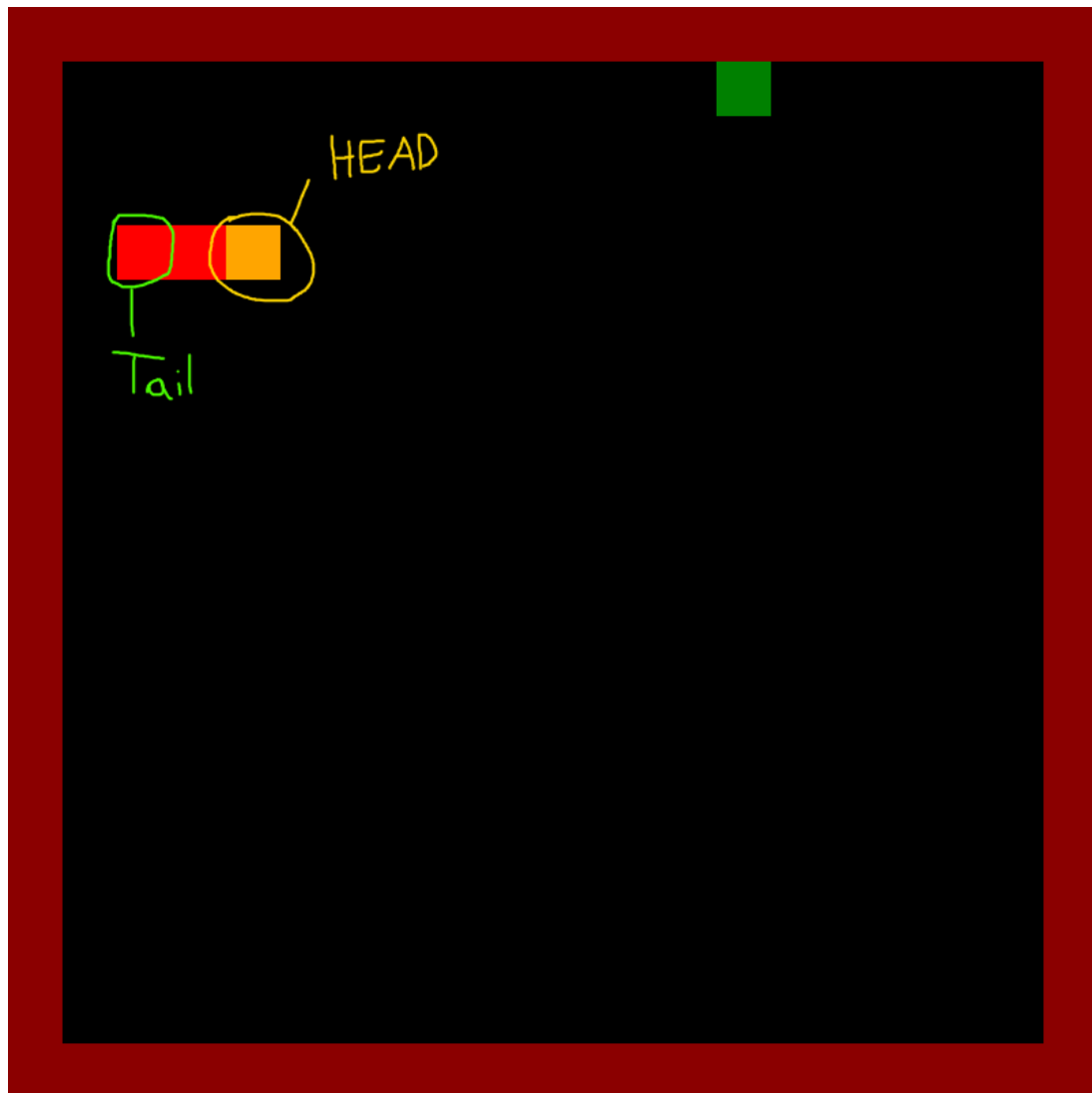
Après que la tête du serpent entre en collision, ce qui signifie que la pomme a été mangée, trois événements se déroulent dans le frame suivant.

Dans le frame où la collision entre la pomme et le serpent a lieu, la pomme change de position sur la grille de jeu et le score augmente de 1. Il n'y a pas de transition, cela se produit immédiatement dès que la tête du serpent couvre la pomme.

Dans le deuxième frame, après qu'il y a eu cette collision, on observe que la tête a avancé d'un carré dans sa direction, mais que sa queue n'a pas disparu. Ainsi, le serpent a grandi d'un carré.

Donc pendant chaque frame de jeu il faut vérifier les positions de la tête du serpent, et celle de la pomme. Quand il y a une collision :

- Le score devrait augmenter de 1
- Un nouveau segment de serpent devrait être ajouté
- La pomme doit changer de position, sans être en conflit avec les positions du serpent



Annexe 2 : Interaction pomme et serpent – GIF

2.3 Collision serpent/bordure

Quand il y a une collision entre le serpent et la bordure de la grille de jeu, il jeu se termine et le joueur a perdu. Le serpent étant un Object dont les positions sont en constant changement à chaque frame de jeu, la bordure quand a elle reste statique. Il n'y a pas forcément de complexité dans les calculs qui doivent être faite. Le zone de jeu, ou le serpent et pomme on le droit d'exister est défini par un largeur et une hauteur. Toute parti du serpent qui ne se trouve pas cette zone est considéré une bordure.

E.G. la zone est 50x50, et commence à (0 ; 0), les quatre coins de la zone seraient :

Nom Coin	Position X	Position Y
Haut Droit	0	0
Haut Gauche	0	50
Bas Droit	50	0
Bas Gauche	50	50

Avec c'est coordonné on sait que si le serpent se déplace à droite et a comme position Y 51, il est sorti de la zone et a donc touché une bordure.

C'est aussi avec ces données que les calculs pour la position d'une pomme, la position de départ du serpent va être généré



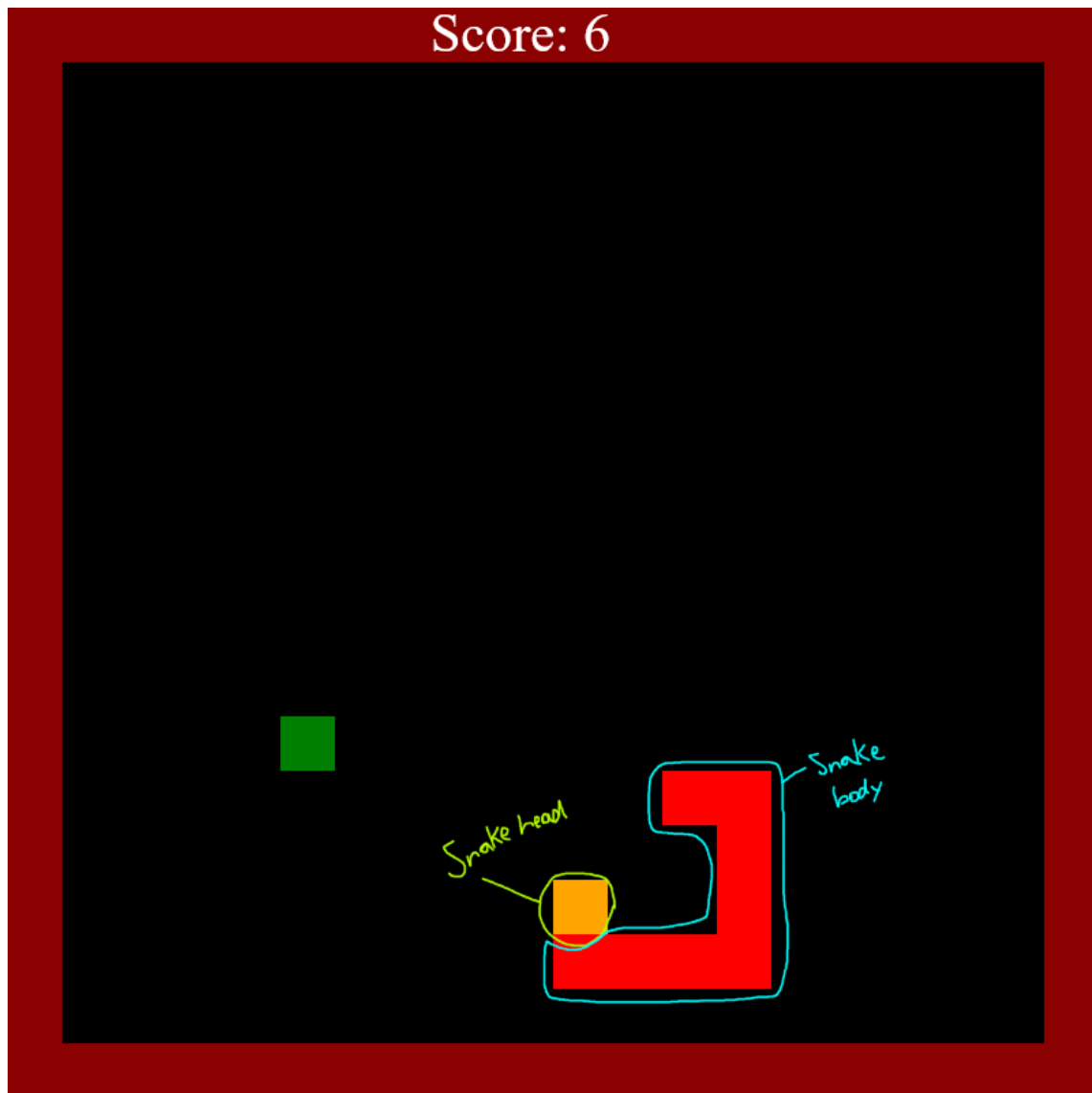
Annexe 3 : Interaction serpent et bordure de jeu - GIF

2.4 Interaction serpent tête/corps

Quand le serpent rentre en contact avec lui-même (tête et corps), ce qui est seulement possible à partir de 5 segments en comptant sa tête, le jeu fini.

Parce que c'est obligé de connaître toutes positions de chaque segment du s'éprend, à chaque frame, c'est seulement nécessaire de faire une simple comparaison des positions.

Si la tête du serpent partage les mêmes positions x et y avec n'importe qu'elle autre segment, c'est qu'il est rentré en lui-même.



Annexe 4 : Interaction serpent tête et corps – GIF


3 Implémentation

3.1 Classes

Quatre classes ont été utilisé dans ce jeu.

- **Main.js** : Class contenant toutes fonctionnalité du jeu, la boucle de jeu, ainsi que toutes code en rapport de la visuelle dans le navigateur avec l'utilisation d'un canvas.
- **Apple.js** : Class objet qui repésent la pomme, elle a que comme propriété qu'une position x et y, des intergers.
- **Segment.js** : Class objet qui est utiliser comme bloc de construction pour créer le serpent, elle a que comme propriétés qu'une position x et y, des intégrrer.
- **Snake.js** : Class objet qui représente le serpent. Comme propriétés il y a "segments", un tableau pour toutes objets segments qui le compose. Et "direction" qui est la direction dans le quelle il se déplace.

C'est classes sont leurs propres fichiers et vont être utiliser avec les mot clés **export** et **import**, qui serre a exporté et importé une classe a une autre qui se trouve dans un autre fichier.



```
export class Snake
```

Figure 1 Class à être exporté




```
import { Snake } from './Snake.js';
```

Figure 2 Importer la class depuis l'emplacement fichier

3.2 Constants

Toutes variables qui sont utilisé plusieurs et dois rester statique sont déclarer comme **const**

Avec des noms pertinents pour que la relecture, et compréhension du code soit le plus facile.



```
// general settings.
const PIXEL_SIZE = 40;
const REFRESH_RATE = 100;
```

Figure 3 Paramètres général du jeu en const. Taille des pixels, fréquence de rafraîchissement


3.3 Variables var et let

La différence entre déclarer une variable avec var et let est leur portée et levage.

Un let est seulement accessible dans le porté (scope), les accolade. Et le let est accessible dès qu'elle est déclaré, partout dans le code.


Les variables déclarées avec var sont hissées (hoisted) en haut de leur portée de fonction ou globale pendant la phase de compilation

Donc en cas pratique ça veut dire qu'avec les vars il y a plus de mémoire utiliser pendant que l'application tourne, parce que chaque var qui se trouve dans le code existe et est accessible (même sans avoir de valeur) au lancement du code.



```
function exampleScope() {
  if (true) {
    var varVariable = "I am a var variable";
    let letVariable = "I am a let variable";
  }
  console.log(varVariable); // Accessible
  console.log(letVariable); // ReferenceError: letVariable is not defined
}
```

Figure 4 Exemple scope - var vs let



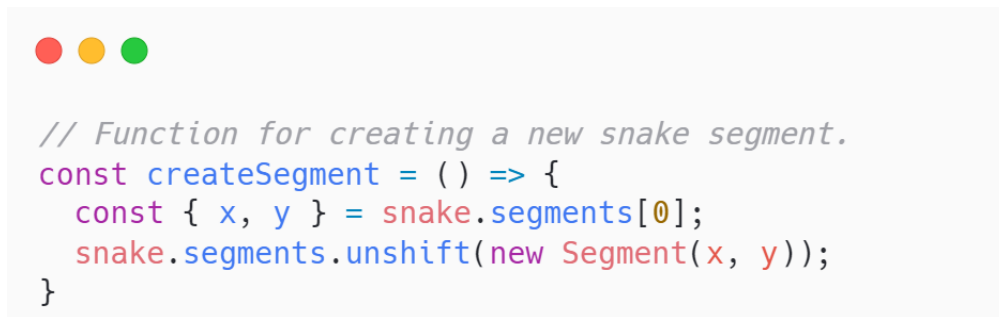
```
console.log(varVariable); // Undefined, but no ReferenceError
console.log(letVariable); // ReferenceError: letVariable is not defined

var varVariable = "I am a var variable";
let letVariable = "I am a let variable";
```

Figure 5 Exemple hissé - var vs let

3.4 Fonctions fléchées

Les fonctions fléchées se comporte comme toutes autre fonctions mais est déclaré comme un **const** et avec une syntaxe différente.



```
// Function for creating a new snake segment.
const createSegment = () => {
  const { x, y } = snake.segments[0];
  snake.segments.unshift(new Segment(x, y));
}
```

Figure 6 Fonction fléché sans paramètres



```
// Function to get a random position within a specified range.
const getRandomPosition = (min, max) => {
  return Math.floor(Math.random() * (max - min + 1)) + min;
}
```

Figure 7 Fonction fléché avec paramètres

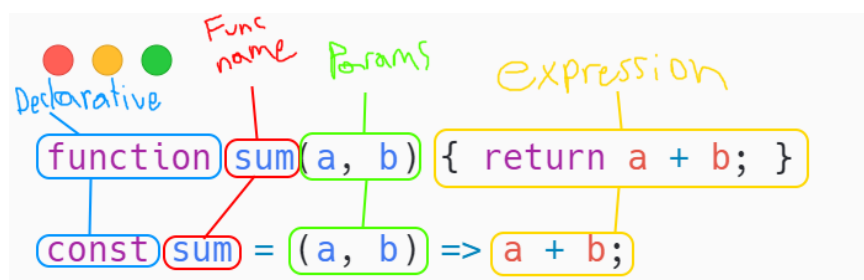


Figure 8 Comparaison entre fonction normale et fléché

3.5 Fonctions de Manipulation de Tableau

JavaScript étant est un langage de code optimiser pour les références et les tableaux.

Par default il y a un nombre de fonctions à disposition pour la manipulation de ces tableaux.

Pour ce je quatre ont étai nécessaire pour gérer le serpent et son déplacement :

- **Shift** : Enlevé le premier élément du tableau (index 0)
- **Pop** : Enlevé le dernier élément du tableau (index length-1)
- **Unshift** : Ajoute un nouvel élément en premier place du tableau (index 0)
- **Push** : Ajoute un nouvel élément a la dernière place du tableau (index length-1)
- **Slice** : Crée un nouvel tableau d'éléments découper d'un tableau originaire
- **Some** : Compare une condition pour toutes éléments d'un tableau

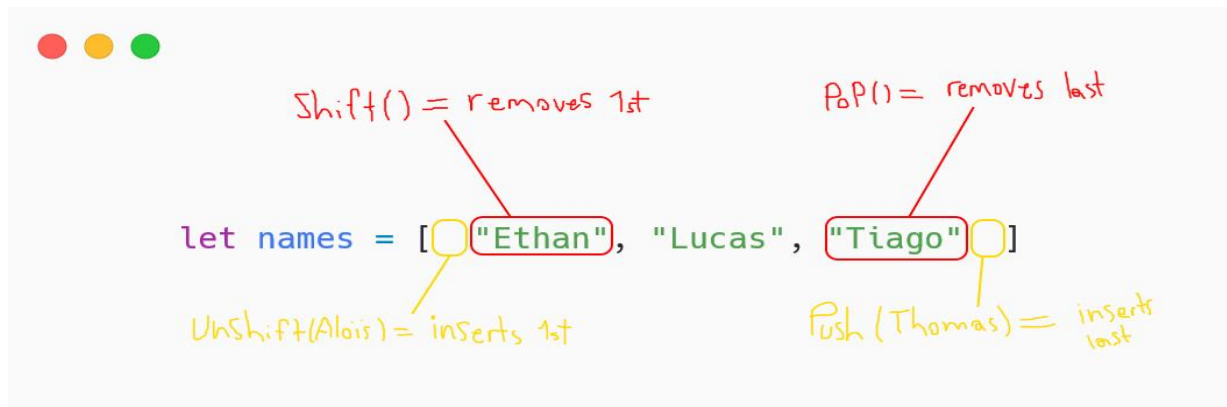


Figure 9 Exemple Shift, Pop, Unshift, Push



Figure 10 Exemple Slice

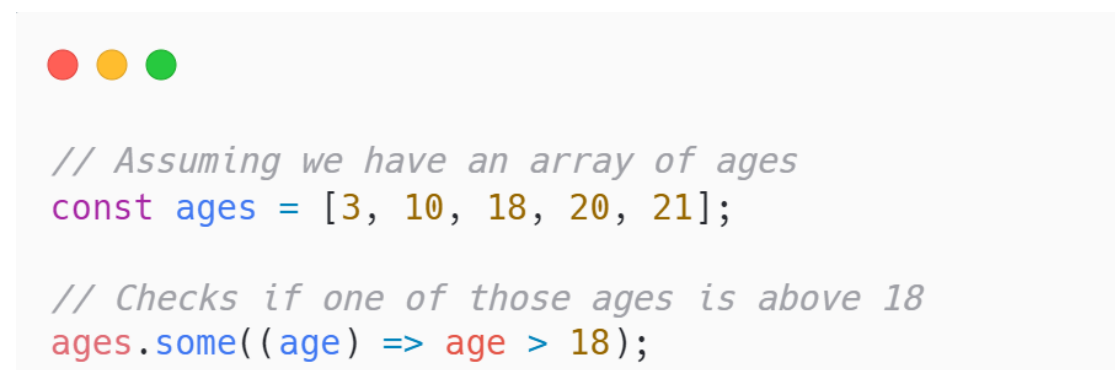



Figure 11 Exemple Some

3.6 Mapping

La fonction **Map()** est une fonction pour les tableaux.

Map() va retourner un tableau peuplé les résultats de l'appel d'une fonction fournie sur chaque élément du tableau appelant.




```
// Assuming we have an array of numbers
const numbers = [1, 2, 3, 4, 5];

// Creating a new array with each number squared
const squaredNumbers = numbers.map(num => num * num);

// Now squaredNumbers will be [1, 4, 9, 16, 25]
```

Figure 12 Fonction Map avec un tableau

Dans le cas d'un tableau d'objets il est utile pour extraire des propriétés, et d'en faire un list.



```
// Extracts all of the x and y values from each snake segment objects and saves into an array.

const X_VALUES = snake.segments.map(a => a.x);
const Y_VALUES = snake.segments.map(a => a.y);
```

Figure 13 Fonction Map avec un tableau d'objets

3.7 Rest

La syntaxe du paramètre rest permet à une fonction d'accepter un nombre indéfini d'arguments sous forme d'un tableau, offrant ainsi une manière de représenter des fonctions [variadiques](#) en JavaScript.



```
const person = (name, ...info) => console.log(info);

person(Ethan, Age, Address, Height, BirthPlace); // [Age, Address, Height, BirthPlace]
```

Figure 14 Exemple Rest

4 Conclusion


4.1 Tests

7 simple tests unitaire ont été faite pour s'assurer que les différentes fonctionnalités du jeu marchent correctement. Parmi ces test 6 sur 7 sont passé.

Nom	Attendu	Résultat
Test de Déplacement avec WASD	Le serpent change de position conformément à la touche appuyée. (W:Haut ; A:Gauche ; S:Bas ; D:Droit).	OK
Test de Collision Serpent avec la Bordure	La collision avec la bordure déclenche la fin du jeu.	OK
Test de Croissance du Serpent après avoir Mangé une Pomme	La taille du serpent est augmentée après la consommation de la pomme.	OK
Test Spawn Aléatoire du Serpent	Le serpent spawn dans une position aléatoire à chaque démarrage de jeu.	OK
Test de Génération Aléatoire de la Position de la Pomme	Une nouvelle pomme est générée à une position qui n'est pas occupée par le serpent.	NOK
Test de Collision avec Soi-même	La collision avec la queue du serpent déclenche la fin du jeu	OK
Test d'Augmentation du Score après avoir Mangé une Pomme	Le score est augmenté après la consommation de la pomme.	OK

Le test de "Génération Aléatoire de la Position de la Pomme" n'a pas fonctionné. La fonction dans la classe main.js qui est responsable de généré une pomme avec des positions aléatoires, qui ne sont pas déjà occupée par la pomme a bien été implémentation mais il arrive quand même que certains pommes spawn directement sur le serpent.

Une solution à cette bug n'a pas pu être trouver avant la fin du projet.



```

// Function that returns an apple object which x & y positions that
// aren't equal to any of the snake segments x & y pos.
const spawnApple = () => {

  const MAX_COORDINATE = GAMEBOARD_LIMIT / PIXEL_SIZE;
  const MIN_COORDINATE = GAMEBOARD_START / PIXEL_SIZE;

  let newX;
  let newY;

  let randomX;
  let randomY;

  // Extracts all of the x and y values from each snake segment objects
  // and saves into an array.
  const X_VALUES = snake.segments.map(a => a.x);
  const Y_VALUES = snake.segments.map(a => a.y);

  // Generates random numbers between 1-18 (gameboard limits), while the
  // numbers don't equate to any of the segment x & y values.
  do {
    randomX = Math.floor(Math.random() * MAX_COORDINATE) +
MIN_COORDINATE;
    randomY = Math.floor(Math.random() * MAX_COORDINATE) +
MIN_COORDINATE;
  } while (X_VALUES.includes(randomX) && Y_VALUES.includes(randomY));

  // new x and y values become the randomly generated values, multiplied
  // by pixelsize so the position fits within gameboard grid.
  newX = randomX * PIXEL_SIZE;
  newY = randomY * PIXEL_SIZE;

  return new Apple(newX, newY);
}

```

Figure 15 Fonction spawnApple

4.2 Conclusion

4.2.1 Amélioration

Durant le projet j'ai beaucoup appris en **JavaScript** qui a donné à des constants changements et optimisations du code. Et toujours maintenant je remarque qu'il y a des choses que j'aurais fait autrement.

Par exemple une réduction du nombre de classes.

Les classes **Apple** et **Segment** sont identique sauf le nom, et où ils sont utilisés dans le main. Il est possible d'en faire qu'un class pour les deux. Un class "bloc" avec que comme propriétés "x" et "y".

La class **Snake** peut aussi être enlever. Le tableau "segments" et sa "direction" peut simplement être remplacer par dans variables qui se trouve dans la main.

Une fonctionnalité sympa que j'aurais pu ajouter pour justifier c'est classes, est de généré une ton couleur de vert aléatoire pour chaque pomme ou de rouge pour chaque segment du serpent.

4.2.2 Avis

Globalement le projet c'est bien dérouler. J'ai peu attendre les caractéristiques du langage JavaScript et mieux comprendre son positionnement dans le monde professionnel et son importance à connaitre.

Ça me motive à continuer de faire du JavaScript dans les mois à venir.

Peut-être que le jeu en lui-même étais un peu facile au niveau de l'algorithme à craquer. Je pense qu'avec un autre langage que je maitrise plus (c#) j'aurais pu le finir en un tiers du temps. Mais c'était une sympa introduction quand même, même si debugger du JavaScript n'est pas claire ou facile.

5 Annexe et Références

5.1 Annexes

1. [Déplacement du serpent – GIF](#)
2. [Interaction pomme et serpent – GIF](#)
3. [Interaction serpent et bordure de jeu - GIF](#)
4. [Interaction serpent tête et corps - GIF](#)

5.2 Figures

1. [export class](#)
2. [import class](#)
3. [constant](#)
4. [scope](#)
5. [hoist](#)
6. [arrow function w/o param](#)
7. [arrow function w/ param](#)
8. [function vs arrow function](#)
9. [shift, pop, unshift, push](#)
10. [slice](#)
11. [some](#)
12. [map array](#)
13. [map object array](#)
14. [spawnApple function](#)
15. [rest](#)