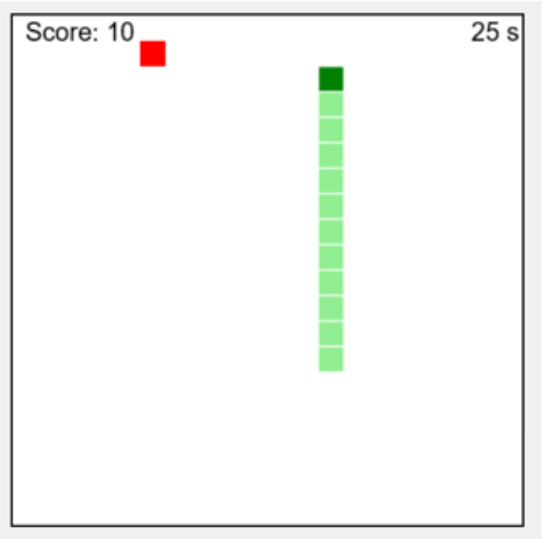
P_Bulles Dev



Meron Essayas – FID2
ETML, Lausanne
Durée 7 Semaines

Table des matières

1 INTRODUCTION

Ce projet consiste à réaliser un réplica du jeu classique du serpent en utilisant **JavaScript**.

Le jeu doit être dessiné sur un **canevas HTML** (<Canvas>)

Le code **JavaScript** utilise la syntaxe des modules **ES** (ECMAScript Modules, ESM) pour organiser le code en plusieurs fichiers avec des responsabilités distinctes.

Le serveur de développement est géré par **Vite**, un outil moderne qui permet un rechargement rapide et une configuration minimale.

2 OBJECTIF

- Contrôler un serpent qui se déplace sur une grille
- Manger de la nourriture (des pommes) pour grandir
- Eviter de se heurter aux murs ou à son propre corps.
- Le score augmente à chaque fois que le serpent mange de la nourriture.

3 INSTALLATION ET SERVEUR DE DÉVELOPPEMENT

On commence par télécharger le fichier p-bulle-snake-step1-start.zip depuis la marketplace. Ensuite, on extrait l'ensemble du contenu du fichier ZIP. Une fois l'extraction effectuée, on ouvre l'invite de commandes dans le dossier obtenu, puis on exécute successivement les commandes **npm install** et **npm run dev.** Enfin, on lance un navigateur web et on accède à l'adresse http://localhost:3000.

4 FONCTIONNALITÉS

4.1 Déplacement du serpent

Pour le déplacement du serpent on va utiliser la fonction **initSnake()** pour initialiser la position de départ du serpent définie sur la grille.

const snake = [{ x: 1, y: 1 }]; → crée une variable snake contenant un tableau.

Ce tableau a un seul objet : { x: 1, y: 1 }.

Cet objet représente la position du premier segment du serpent sur la grille (colonne x = 1, ligne y = 1).

return snake; → la fonction renvoie le tableau, c'est-à-dire le serpent initial prêt à être utilisé dans le jeu.

```
/**
 * Initialise le serpent au début du jeu.
 *
 * Cette fonction crée le serpent en tant que tableau contenant un seul segment,
 * positionné à une position de départ définie sur la grille.
 *
 * @returns {Array<{x: number, y: number}>} - Un tableau contenant un objet représentant la position du premier segment du serpent.
 */
function initSnake() {
  const snake = [{ x: 9 * 20, y: 6 * 20 }]; // Position initiale du serpent (9,6) sur une grille de 20x20 pixelsa
  return snake;
}
```

Figure 1 : Initialise le serpent au début du jeu.

const head = snake[0]; → récupère la tête actuelle du serpent (le premier élément du tableau).

let newHead = { x: head.x, y: head.y }; → crée une copie de la tête pour calculer la nouvelle position.

Les if vérifient la direction et ajustent x ou y selon la taille d'une case (box).

snake.unshift(newHead); → insère la nouvelle tête au début du tableau.

snake.pop(); → enlève le dernier segment du serpent pour donner l'illusion du mouvement.

La fonction retourne les nouvelles coordonnées de la tête.

```
/**
 * Déplace le serpent dans la direction actuelle.
 *
 * Cette fonction calcule la nouvelle position de la tête du serpent en fonction
 * de la direction actuelle (gauche, haut, droite, bas). Le reste du corps du serpent
 * suit la tête. La fonction retourne un objet représentant la nouvelle position de la tête du serpent.
 *
 * @param {Array<{x: number, y: number}>} snake - Le tableau représentant le serpent, où chaque élément est un segment avec des coordonnées `x` et `y`.
 * @param {string} direction - La direction actuelle du mouvement du serpent ("LEFT", "UP", "RIGHT", ou "DOWN").
 * @param {number} box - La taille d'une case de la grille en pixels, utilisée pour déterminer la distance de déplacement du serpent.
 * @returns {{x: number, y: number}} - Un objet représentant les nouvelles coordonnées `x` et `y` de la tête du serpent après le déplacement.
 */
function moveSnake(snake, direction, box) {
    // Récupère la tête actuelle
  const head = snake[0];
  let newHead = { x: head.x, y: head.y };

  // Met à jour les coordonnées en fonction de la direction
  if (direction === "LEFT") {
    newHead.x -= box;
  } else if (direction === "UP") {
    newHead.y -= box;
  } else if (direction === "RIGHT") {
    newHead.x += box;
  } else if (direction === "DOWN") {
    newHead.y += box;
  }

  // Ajoute la nouvelle tête au début du tableau
  snake.unshift(newHead);

  // Supprime le dernier segment pour simuler le déplacement
  snake.pop();

  return newHead;
}
```

Figure 2 : Déplace le serpent dans la direction actuelle.

La fonction drawSnake a pour objectif de dessiner le serpent sur le canvas du jeu. Elle prend trois paramètres :
ctx → le contexte 2D du canvas, qui permet de dessiner des formes.
snake → un tableau contenant tous les segments du serpent, chaque segment étant un objet {x, y} indiquant sa position sur la grille.
box → la taille d'une case de la grille en pixels, qui détermine la largeur et la hauteur de chaque segment.
La fonction utilise forEach pour parcourir chaque segment du serpent :
segment → représente le segment actuel à dessiner (objet {x, y}).
index → représente la position du segment dans le tableau.
Si index === 0, c'est la tête du serpent.
Les autres segments représentent le corps.
Pour chaque segment :
On définit la couleur avec ctx.fillStyle : darkgreen pour la tête, lightgreen pour le corps.
On dessine un rectangle avec ctx.fillRect(segment.x, segment.y, box, box), à l la position du segment et avec la taille box.

```
/**
 * Dessine le serpent sur le canvas.
 *
 * Cette fonction parcourt chaque segment du serpent et le dessine sur le canvas en utilisant
 * un rectangle coloré. La tête du serpent est dessinée dans une couleur différente des autres segments
 * pour la distinguer visuellement. Chaque segment est dessiné à sa position actuelle sur la grille,
 * avec une taille déterminée par la valeur de `box`.
 *
 * @param {CanvasRenderingContext2D} ctx - Le contexte de rendu 2D du canvas utilisé pour dessiner.
 * @param {Array<
 * @param {number
 */
function drawSnak
  snake.forEach((segment, index) => {
    ctx.fillStyle = index === 0 ? "darkgreen" : "lightgreen"; // Couleur différente pour la tête et le corps
    ctx.fillRect(segment.x, segment.y, box, box);
  });
}
```

```
(parameter) segment: {
    x: number;
    y: number;
}
```

Figure 3 : Dessine le serpent sur le canvas.

4.2 Gestion des collisions

La fonction modifie la direction du serpent lorsque l'utilisateur appuie sur une touche fléchée.

Paramètres :

event → l'événement clavier indiquant la touche enfoncée.

currentDirection → la direction actuelle du serpent.

Vérifie la touche enfoncée (Flèche Haut, Flèche Bas, Flèche Gauche, Flèche Droite).

S'assure que la nouvelle direction n'est pas opposée à la direction actuelle (pour empêcher le serpent de faire demi-tour).

Renvoie la nouvelle direction si elle est valide ; sinon, conserve la direction actuelle.

```
/**
 * Gère le changement de direction du serpent en fonction de l'entrée de l'utilisateur.
 *
 * Cette fonction est appelée chaque fois qu'une touche directionnelle est pressée.
 * Elle vérifie que la nouvelle direction n'est pas opposée à la direction actuelle
 * (pour éviter que le serpent se retourne sur lui-même) et retourne la nouvelle direction
 * si elle est valide.
 *
 * @param {KeyboardEvent} event - L'événement clavier qui contient les informations sur la touche pressée.
 * @param {string} currentDirection - La direction actuelle du serpent (peut être "UP", "DOWN", "LEFT", ou "RIGHT").
 * @returns {string} - La nouvelle direction du serpent après traitement, ou la direction actuelle si le changement n'est pas valide.
 */
function handleDirectionChange(event, currentDirection) {
    let newDirection = currentDirection;

  switch (event.key) {
    case "ArrowUp":
      if (currentDirection !== "DOWN") newDirection = "UP";
      break;
    case "ArrowDown":
      if (currentDirection !== "UP") newDirection = "DOWN";
      break;
    case "ArrowLeft":
      if (currentDirection !== "RIGHT") newDirection = "LEFT";
      break;
    case "ArrowRight":
      if (currentDirection !== "LEFT") newDirection = "RIGHT";
      break;
  }

  return newDirection;
}
```

Figure 4 : Gère le changement de direction du serpent en fonction de l'entrée de l'utilisateur.

4.3 Gestion du Score

La fonction drawScore a pour rôle d'afficher le score actuel du jeu sur le canvas.

Paramètres :

ctx → le contexte 2D du canvas, qui permet de dessiner du texte et des formes.

score → un nombre entier représentant le score actuel du joueur.

Principe de fonctionnement :

ctx.fillStyle = "black"; → définit la couleur du texte en noir.

ctx.font = "20px Arial"; → définit la taille et la police du texte.

ctx.textAlign = "left"; → aligne le texte sur la gauche du point de départ.

ctx.fillText(\Score: ${score}`, 10, 25);→ dessine le texte"Score: X"sur le canvas à la position(x=10, y=25)` (coin supérieur gauche).

```
/**
 * Dessine le score sur le canvas.
 *
 * Cette fonction affiche le score actuel du jeu dans le coin supérieur gauche du canvas.
 * Le score est affiché en noir avec une police Arial de 20px.
 *
 * @param {CanvasRenderingContext2D} ctx - Le contexte de rendu 2D du canvas utilisé pour dessiner.
 * @param {number} score - Le score à afficher, qui est un entier.
 */
function drawScore(ctx, score) {
  ctx.fillStyle = "black";       // text color
  ctx.font = "20px Arial";       // font size and family
  ctx.textAlign = "left";        // align text to the left
  ctx.fillText(`Score: ${score}`, 10, 25); // draw text at x=10, y=25
}
```

Figure 5 : Dessine le score sur le canvas.

4.4 Chargement Asynchrone de la Configuration

La fonction loadConfig est une fonction asynchrone qui permet de charger le fichier **config.json** au démarrage du jeu. Elle utilise fetch pour récupérer ce fichier contenant les paramètres essentiels comme la taille des cases (box) et la vitesse du jeu (gameSpeed). Grâce à await, l'exécution attend que les données soient disponibles avant de continuer, garantissant ainsi que le jeu ne démarre qu'une fois les configurations correctement chargées. Si le fichier est introuvable ou qu'une erreur survient lors du chargement, celle-ci est capturée et affichée dans la console. Cette approche assure une initialisation fiable et flexible du jeu.

```
async function loadConfig() {
  try {
    const response = await fetch('config.json');
    if (!response.ok) throw new Error('Failed to load config');
    const config = await response.json();
    box = config.box;
    gameSpeed = config.gameSpeed;
    startGame(); // démarrer uniquement après le chargement de la configuration
  } catch (error) {
    console.error('Error loading config:', error);
  }
}
```

Figure 6 : Chargement asynchrones

4.5 Fonctionnalité de Pause et Reprise du Jeu

La fonctionnalité de pause du jeu est gérée par la fonction **togglePause** qui bascule l'état du jeu entre pause et en cours. Lorsque le jeu est en pause, la boucle principale est interrompue grâce à la fonction **clearInterval**, et un élément visuel s'affiche pour indiquer cet état. Cet élément atténue légèrement l'arrière-plan et affiche un message « Pause » semi-transparent au centre de l'écran, en indiquant clairement l'état du jeu. Pour reprendre le jeu, la boucle principale est relancée par la fonction **setInterval**, et le jeu reprend là où il avait été interrompu.

```
function togglePause() {
  paused = !paused;
  if (paused) {
    clearInterval(gameInterval);
    drawPausedMessage();
  } else {
    gameInterval = setInterval(draw, gameSpeed);
  }
}

function drawPausedMessage() {
 // Atténuer légèrement l'arrière-plan
  ctx.fillStyle = "rgba(0, 0, 0, 0.3)"; // noir semi-transparent
  ctx.fillRect(0, 0, canvas.width, canvas.height);

  // Dessiner le texte « En pause » au centre
  ctx.font = "40px Arial";
  ctx.fillStyle = "rgba(255, 255, 255, 0.8)"; // blanc semi-transparent
  ctx.textAlign = "center";
  ctx.fillText("Paused", canvas.width / 2, canvas.height / 2);
}
```

Figure 7 : Fonctionnalité de Pause et Reprise du Jeu

5 CONCLUSION

Ce projet m'a permis de mettre en pratique mes connaissances en JavaScript et d'approfondir ma compréhension du développement web moderne. En réalisant un jeu complet et interactif comme le Snake, j'ai appris à structurer mon code avec les modules ES, à utiliser un serveur de développement avec Vite, et à documenter mes fonctions à l'aide de JSDoc. L'ajout d'un système de gestion du Top 5 des scores, persistant via une API, m'a également permis de découvrir les bases de la communication entre un client et un serveur Node.js. Ce travail m'a aidé à renforcer mes compétences en programmation, en organisation et en autonomie tout en réalisant un projet concret et fonctionnel.