Projet Tetris

Structure Générale du code :

Avant d'expliquer le fonctionnement détaillé du Tetris et toutes les fonctionnalités implémentées, voici comment se structure notre code.

1. **La Grille de Jeu**
   Notre Tetris repose sur une grille dans laquelle nous plaçons et déplaçons nos tétriminos. Il s'agit d'un tableau à deux dimensions dans notre code, représentant les cases de la grille. Côté front-end, une véritable grille CSS permet d'afficher visuellement le plateau de jeu.

2. **Les Tétriminos**
   Les tétriminos sont les différentes formes que le joueur déplace et fait pivoter sur la grille. Chaque forme est définie par une configuration spécifique de cases dans un tableau, ce qui facilite leur manipulation dans le code.

3. **Fonctions Principales**
   Nos fonctions principales contiennent la logique du jeu. Elles gèrent les mouvements des tétriminos, la détection des collisions, la rotation des pièces, ainsi que la suppression des lignes complètes. Ces fonctions assurent le bon déroulement du jeu et la réactivité des actions du joueur.

4. **Gestion du Score et de la Progression**
   Le système de score suit la progression du joueur en augmentant les points à chaque ligne supprimée. La difficulté du jeu augmente progressivement, avec une accélération de la descente des tétriminos à mesure que le score s'élève.

5. **Gestion des entrées claviers**

   Cette partie du code gère les interactions du joueur avec le jeu. Les touches du clavier permettent de déplacer les tétriminos à gauche, à droite, vers le bas, ainsi que de les faire pivoter. Chaque entrée est associée à des actions spécifiques qui modifient l'état du jeu en conséquence, tout en vérifiant les conditions de validité des mouvements (collisions, dépassement des limites de la grille, etc.)

6. **Interface Utilisateur**
   L'interface utilisateur permet au joueur d'interagir facilement avec le jeu. Elle inclut la grille de jeu, l'affichage du score, et des éléments de contrôle pour démarrer, mettre en pause ou réinitialiser la partie.

Voici maintenant comment le code fonctionne de manière un peu plus détaillée.

1) La grille :

Pour la conception de notre Tetris, nous avons décidé de créer 2 grilles dans notre javascripts. Une grille qui servira pour le tetrominos actuelle, et une autre sur lequel apparaitront les terminos précédent qui sont déjà entrer en collision.

```
const backgroundGrid = Array.from({ length: ROWS }, () => Array(COLS).fill(0));
const foregroundGrid = Array.from({ length: ROWS }, () => Array(COLS).fill(0));
```

Ces deux lignes de code permettent de déclarer nos grilles. La fonction Array.from permet de créer un tableau de taille ROWS (correspondant au nombre de lignes), et utilise une fonction fléchée pour générer, à chaque ligne, un tableau de taille COLS rempli de zéros grâce à la méthode fill(0). Ainsi, chaque ligne du tableau principal contient un sous-tableau de COLS éléments. Cela nous permet de créer une grille à deux dimensions.

Pour dessiner la grille dans notre front-end, nous avons créé une fonction :

```
function drawbackgroundGrid() {
  gameContainer.innerHTML = ""; // Réinitialiser la grille

  for (let row = 1; row < ROWS; row++) {
    for (let col = 0; col < COLS; col++) {
      const cell = document.createElement("div");
      cell.classList.add("cell");

      // Vérifier d'abord la foregroundGrid
      if (foregroundGrid[row][col] !== 0) {
        cell.classList.add(`active${foregroundGrid[row][col]}`);
      }
      // Si foregroundGrid est vide, vérifier la backgroundGrid
      else if (backgroundGrid[row][col] !== 0) {
        cell.classList.add(`active${backgroundGrid[row][col]}`);
      }

      gameContainer.appendChild(cell);
    }
  }
}
```

Cette fonction fait le lien entre la grille de notre code JavaScript et l'affichage dans le front-end. gameContainer contient l'élément du DOM qui va accueillir la grille du jeu. La fonction parcourt les grilles JavaScript (foregroundGrid et backgroundGrid) et, pour chaque cellule, crée un élément avec la classe cell. Si la valeur de la cellule est différente de 0, elle reçoit en plus la classe active suivie de cette valeur (par exemple

active1, active2). Cette valeur détermine la couleur de la case, correspondant à un type de tétrimino. A chaque utilisation la grille est réinitialisée au début de la fonction, nous nous serviront de cette fonction à chaque mise à jour de la grille.

```css
#game-container {
  display: grid;
  grid-template-rows: repeat(20, 40px); /* 20 lignes */
  grid-template-columns: repeat(10, 40px); /* 10 colonnes */
  gap: 1px;
  background-color: #444;
  margin: 0 auto;
  width: 410px; /* (10 * 25) + (9 * 1px gap) */
  /*border: 2px solid white;*/
}
```

Voici le CSS de notre gameContainer. Il s'agit de la grille ôté front end que notre fonction va remplir en créant des éléments à l'intérieur.

2) Les tetrominos :

Chaque tetrominos est représenté par un tableau à 2 dimensions également (une matrice). Nous avons créé une constante « tetrominos », qui est un dictionnaire qui vas contenir toute ces matrices.

```javascript
const tetrominos = {
  I: [
    [0, 0, 0, 0],
    [1, 1, 1, 1],
    [0, 0, 0, 0],
    [0, 0, 0, 0],
  ],
  J: [
    [2, 0, 0],
    [2, 2, 2],
    [0, 0, 0],
  ],
  L: [
    [0, 0, 3],
    [3, 3, 3],
    [0, 0, 0],
  ],
  O: [
    [4, 4],
    [4, 4],
  ],
```

Les valeurs contenues dans ces matrices permettent simplement d'associé une valeur pour chaque tetromino.

```
function addTetrominoTobackgroundGrid(tetromino) {
  for (let row = 0; row < tetromino.length; row++) {
    for (let col = 0; col < tetromino[row].length; col++) {
      if (tetromino[row][col] > 0) {
        if (
          coordonneeTetrominos[0] + row >= ROWS ||
          coordonneeTetrominos[1] + col >= COLS
        ) {
          console.log("outOfBoundException");
          throw new Error("outOfBoundException");
        }

        if (foregroundGrid[coordonneeTetrominos[0] + row]
                          [coordonneeTetrominos[1] + col] >0 ) {
        console.log("colissionWithTetrominoException");
        throw new Error("colissionWithTetrominoException");
        }

        backgroundGrid[coordonneeTetrominos[0] + row][
          coordonneeTetrominos[1] + col
        ] = tetromino[row][col];
      }
    }
  }
}
```

La fonction **addTetrominoTobackgroundGrid** permet d'ajouter le tétrimino actuel à la grille de fond en tenant compte de sa position. Elle parcourt chaque cellule du tétrimino et vérifie d'abord s'il dépasse les limites de la grille. Si c'est le cas, une exception est levée pour éviter qu'il ne sorte du cadre. Ensuite, elle contrôle si la cellule correspondante dans la grille de premier plan est déjà occupée. Si une collision est détectée avec un autre tétrimino déjà en place, une autre exception est levée pour annuler l'action. Si aucune erreur n'est rencontrée, les cellules du tétrimino sont copiées dans la grille de fond, permettant de fixer la pièce à sa position actuelle

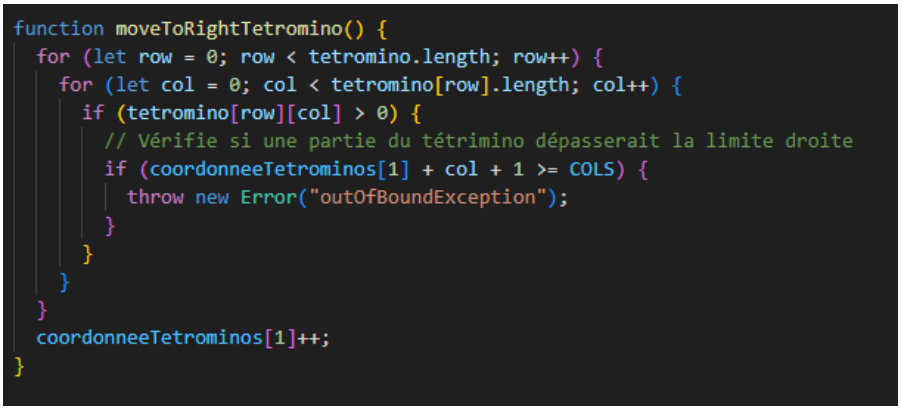
```
//genere un tetromino aleatoire
function getRandomTetromino() {
  const tetrominoKeys = Object.keys(tetrominos); // Récupère les clés (I, J, L, O, S, Z, T)
  const randomKey = tetrominoKeys[Math.floor(Math.random() * tetrominoKeys.length)]; // Sélectionne une clé au hasard
  return tetrominos[randomKey]; // Retourne le nom du tétriminos sous forme de chaîne
}
```

Nous avons également cette fonction permettant de générer un tetromino aléatoire.

Object.keys nous permet de récupérer toutes les clés du dictionnaire contenant no tetrominos, puis on génère une clé aléatoire, avant de retourner la bonne matrice à l'aide de cette clé aléatoire.

3) Fonctions principales :

Ensuite, pour pourvoir faire bouger ces tetrominos dans notre grille, nous avons créé plusieurs fonctions.



```
function moveToRightTetromino() {
  for (let row = 0; row < tetromino.length; row++) {
    for (let col = 0; col < tetromino[row].length; col++) {
      if (tetromino[row][col] > 0) {
        // Vérifie si une partie du tétrimino dépasserait la limite droite
        if (coordonneeTetrominos[1] + col + 1 >= COLS) {
          throw new Error("outOfBoundException");
        }
      }
    }
  }
  coordonneeTetrominos[1]++;
}
```

Cette fonction permet de déplacer le tetromino à droite. Pour effectuer ce déplacement, il suffit d'ajouter 1 à colonne dans les coordonnés du tétromino. Mais avant cela la fonction vérifie que le tétrimino ne dépasse pas la limite droite de la grille lors d'un déplacement. Pour cela, elle parcourt chaque cellule active du tétrimino et calcule sa nouvelle position après le déplacement. Si cette position dépasse la limite de la grille (COLS), une erreur est générée. Si aucune cellule ne dépasse, le tétrimino est déplacé d'une case vers la droite. Il y a également une fonction moveToLeftTetromino qui marche de la même manière.

Ensuite il y a cette fonction qui permet de faire tourner le tétromino :

```
function rotateTetromino() {
  let n = tetromino.length;
  for (let i = 0; i < n / 2; i++) {
    for (let j = i; j < n - i - 1; j++) {
      let temp = tetromino[i][j];
      tetromino[i][j] = tetromino[n - j - 1][i];
      tetromino[n - j - 1][i] = tetromino[n - i - 1][n - j - 1];
      tetromino[n - i - 1][n - j - 1] = tetromino[j][n - i - 1];
      tetromino[j][n - i - 1] = temp;
    }
  }
}
```

Puisque nos tetrominos sont codé sous forme de matrice carré, on peut utiliser une formule de rotation de matrice a 90° et la mettre sous forme d'algorithme pour générer la rotation au sein de la matrice (et non de la grille)

```
function moveToDownTetromino() {

  if (isCollision()) {
    console.log("Collision détectée");

    lockTetromino();

    tetromino = nextTetromino;
    nextTetromino = getRandomTetromino(); // Générer un nouveau tétrimino
    drawPreviewTetromino(nextTetromino); // Afficher le prochain tétrimino

    // Réinitialiser les coordonnées du tétrimino
    coordonneeTetrominos[0] = 0;
    coordonneeTetrominos[1] = 3;

  }
  coordonneeTetrominos[0]++;
}
```

Cette fonction permet de déplacer le tétromino vers le bas en enlevant 1 à la coordonnée de la ligne. Mais avant de la déplacer vers le bas, on vérifie si il y a collision ou non. Pour cela on utilise la fonction isCollision :

```
function isCollision() {
  for (let row = 0; row < tetromino.length; row++) {
    for (let col = 0; col < tetromino[row].length; col++) {
      if (tetromino[row][col] > 0) {
        const newRow = coordonneeTetrominos[0] + row + 1;
        const newCol = coordonneeTetrominos[1] + col;

        // Vérifie si le bas de la grille est atteint
        if (newRow >= ROWS) {
          return true;
        }

        // Vérifie s'il y a une autre pièce en dessous
        if (foregroundGrid[newRow][newCol] > 0) {
          return true;
        }
      }
    }
  }
  return false;
}
```

Cette fonction vérifie si le tétrimino entre en collision lorsqu'il descend. Pour chaque cellule occupée du tétrimino, elle vérifie deux cas : si la cellule dépasse le bas de la grille ou si elle entre en contact avec un autre tétrimino déjà placé. Si l'un de ces cas est vrai, la fonction signale une collision en renvoyant True. Sinon, elle retourne False, indiquant qu'aucun obstacle n'est présent. Puis on fait apparaitre on change de tetromino et reset les coordonnés de celui-ci.

Puis dans la fonction moveDownTetromino, s'il y a en effet collision, alors on lock le tetromino. Cette fonction a pour effet de faire passer le tetromino dans la 2 grilles, la « foreGroundGrid » qui nous sert justement pour les pièces arrivées en bas.

On peut voir également dans moveDownTetromino, que le nouveau tetromino prend la valeur de nextTetromino, il y a cela car nous avons fait en sorte d'avoir un preview du prochain tetromino dans notre conception. Nous avons donc besoin au moment de la collision remplacer le tetromino actuel par le prochain et de faire un randomTetromino pour le prochain tetromino à chaque collision. Puis d'utiliser notre fonction drawPreviewTetromino, qui dessine cette preview dans notre front-end dans une petite grille contenue dans un .

```
function drawPreviewTetromino(tetromino) {
  const previewContainer = document.getElementById("preview");
  previewContainer.innerHTML = ""; // Réinitialiser la prévisualisation

  // Créer une grille de prévisualisation pour afficher le tétrimino
  for (let row = 0; row < 4; row++) { // La preview sera une grille 4x4
    for (let col = 0; col < 4; col++) {
      const cell = document.createElement("div");
      cell.classList.add("cell");

      // Dessiner le tétrimino dans la preview si la cellule est active
      if (tetromino[row] && tetromino[row][col] > 0) {
        // Ajout d'une classe active en fonction de la couleur
        cell.classList.add(`active${tetromino[row][col]}`);
      }

      previewContainer.appendChild(cell);
    }
  }
}
```

Cette fonction fonctionne de manière similaire à notre fonction drawBackGroundGrid qui dessine la grille de base.

4) Gestion du score et de la progression :

La gestion du score et de la progression se fait en grande partie dans une seule fonction, qui est notre fonction checkLine. Il s'agit de la fonction qui vérifie si le joueur à réalisé une ou plusieurs ligne complète. Cette fonction est appelée également à chaque modification de la grille.

```
function checkLine() {

  let nbligneComplete = 0; //on compte le nombre de ligne complète pour le score

  for (let row = foregroundGrid.length - 1; row >= 0; row--) {
    let isComplete = true;
    //parcour la grille et verifie si une ligne est complete
    for (let col = 0; col < foregroundGrid[row].length; col++) {
      if (foregroundGrid[row][col] === 0) {
        isComplete = false;
        break;
      }
    }
    // Si la ligne est complète
    if (isComplete) {
      // Supprime la ligne
      foregroundGrid.splice(row, 1);
      // Ajoute une nouvelle ligne vide en haut de la grille
      foregroundGrid.unshift(new Array(foregroundGrid[0].length).fill(0));
      // Vérifier la même ligne à nouveau car elle a été remplacée
      row++;
      nbligneComplete++;
    }
  }
  nbLignecompleteTotal += nbligneComplete;
```

Voici la première partie de la fonction.

Cette fonction détecte et supprime les lignes complètes de la grille (la foreGroundGrid). Elle commence par parcourir la grille du bas vers le haut afin de ne pas manquer de lignes après suppression.

Pour chaque ligne, elle vérifie si toutes les cellules sont remplies. Si une seule case est vide, la ligne est considérée comme incomplète et la vérification passe à la suivante. En revanche, si une ligne est entièrement remplie, elle est supprimée de la grille à l'aide de la fonction splice (fonction que l'on peut appliquer au Array et permet de . Cela permet aux autres lignes de descendre naturellement, comme dans le jeu Tetris classique.

Chaque suppression de ligne est comptabilisée dans la variable nbligneComplete, qui sert ensuite à ajuster le score et le niveau du joueur.

```
let newLevel = Math.floor(nbLignecompleteTotal / 10) + 1;

if (newLevel > level) {
  level = newLevel;
  gameSpeed = initialSpeed - (level) * 50;
  levelElement.innerHTML = level;
}

score += nbligneComplete * (level + 1) * 100;
scoreElement.innerHTML = score;
```

Ensuite, voici la 2ème partie de la fonction. Le score et le niveau du joueur sont mis à jour en fonction du nombre de lignes supprimées.

Le score et le niveau du joueur sont mis à jour en fonction du nombre de lignes supprimées.

D'abord, le total de lignes complètes (nbLignecompleteTotal) est incrémenté. Ensuite, un nouveau niveau est calculé en divisant ce total par 10, puis en ajoutant 1. Cela signifie qu'un joueur monte de niveau toutes les 10 lignes supprimées. (Il serait intéressant de mettre une constante à la place de 10, puis potentiellement de laisser l'utilisateur gérer la progression de la difficulté)

Si le niveau augmente, la vitesse du jeu est ajustée pour rendre la partie plus difficile. La vitesse de base est réduite de 50 millisecondes par niveau, ce qui accélère la chute des tétriminos.

Enfin, le score est mis à jour en fonction du nombre de lignes supprimées et du niveau actuel. Plus le joueur atteint un niveau élevé, plus le score attribué est important. Le nouveau score est ensuite affiché à l'écran, tandis que la progression du niveau et la vitesse du jeu sont mises à jour en conséquence.

```
setInterval(updateGame, gameSpeed);
```

La vitesse du jeu est utilisée dans notre setInterval, qui est simplement la boucle de notre jeu. La fonction gameUpdate va être appelé toute les « gameSpeed » ms.

```
function updateGame(){
  resetBackgroundGrid();
  moveToDownTetromino();
  addTetrominoTobackgroundGrid(tetromino); // Ajouter un tétromino "T" à la grille
  drawbackgroundGrid();
  checkLine();
}
```

La fonction gameUpdate. Contient simplement un reset de la grille, suivis du déplacement vertical du tetromino, son ajout à la grille, le dessin de la grille et la vérification des lignes complètes.

5) Gestion des entrer clavier :

```
function recordKey(e) {
  switch (e.key) {
    case "Up":
    case "ArrowUp":
      try {
        resetBackgroundGrid();
        rotateTetromino();
        addTetrominoTobackgroundGrid(tetromino); // Ajouter un tétromino "T" à la grille
        drawbackgroundGrid();
        checkLine();

      } catch (error) {
        console.log("Error");
        //Si on a depasser donc on annule l'action
        if (error.message="colissionWithTetrominoException"){
          rotateTetromino();
          rotateTetromino();
          rotateTetromino();
        }
      }
      console.log(tetromino);
      break;
    case " ":
    case "Spacebar":
      break;
    case "ArrowRight":
      try {
        resetBackgroundGrid();
        moveToRightTetromino();
        addTetrominoTobackgroundGrid(tetromino); // Ajouter un tétromino "T" à la grille
        drawbackgroundGrid();
        checkLine();
      } catch (error) {
        //Si on a depasser donc on annule l'action
        if (error.message="colissionWithTetrominoException"){
          moveToLeftTetromino();
```

Cette fonction permet de contrôler les mouvements du tétrimino en fonction des touches pressées par le joueur. Elle écoute les événements clavier et exécute l'action correspondante selon la touche enfoncée.

Les touches fléchées (ArrowUp, ArrowDown, ArrowLeft, ArrowRight) permettent respectivement de faire pivoter le tétrimino, de le faire descendre plus rapidement, de le déplacer vers la gauche ou vers la droite. Avant chaque mouvement, la grille est réinitialisée pour éviter les doublons d'affichage, puis mise à jour après l'action.

Gestion des erreurs et corrections

Lorsque le joueur tente de déplacer ou de faire pivoter un tétrimino, une exception peut être levée si le mouvement est impossible (par exemple, en cas de collision avec un autre tétrimino ou avec les bords de la grille). Dans ces cas, l'action est annulée en effectuant l'opération inverse (comme annuler une rotation ou un déplacement).

Cela permet d’éviter que le tétrimino se retrouve bloqué hors de la grille ou empilé de manière incorrecte.

6) Interface utilisateur :

Pour ce qui est de notre interface utilisateur, celle-ci est assez légère. Notre html contient uniquement un conteneur pour la grille, et un autre qui contient le score, le niveau et la preview de la future pièce.

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Tetris</title>
  <link rel="stylesheet" href="tetris.css">
</head>
<body>
  <h1>Tetris</h1>
  <div id="game-container"></div>
  <div id="container">
    <h1 id = scoreLabel> score </h1>
    <h1 id="score">0</h1>
    <h1 id = levelLabel> level </h1>
    <h1 id="level"> 0 </h1>
    <div id="preview"> </div>
  </div>
  <script src="tetris.js"></script>
</body>
</html>
```

Repository github :

https://github.com/MohamedLaidaoui/Tetris