Documentation Technique Approfondie - Générateur de Donjons 2D

Table des matières détaillée

---

1. Introduction et Concepts Fondamentaux

1.1 Qu'est-ce qu'un donjon procédural ?

Un donjon procédural est une carte générée algorithmiquement plutôt que dessinée à la main. L'objectif est de créer des environnements qui sont :

- **Uniques** : Chaque génération produit un résultat différent
- **Cohérents** : La structure suit une logique architecturale
- **Jouables** : Tous les espaces sont accessibles et intéressants
- **Équilibrés** : Ni trop faciles ni trop difficiles à naviguer

1.2 Représentation en grille 2D

Concept de base :

- Une matrice (tableau de tableaux) représente l'espace

- Chaque cellule a une valeur : 1 (mur) ou 0 (sol)

- Les coordonnées sont notées [y][x] (ligne puis colonne)

Exemple 5x5 :

[1,1,1,1,1] ← Ligne 0 (bordure haute)

[1,0,0,0,1] ← Ligne 1

[1,0,1,0,1] ← Ligne 2

[1,0,0,0,1] ← Ligne 3

[1,1,1,1,1] ← Ligne 4 (bordure basse)

↑ ↑

Col 0 Col 4

1.3 Contraintes du moteur de jeu

Votre moteur impose que :

- Les bordures (x=0, x=31, y=0, y=31) soient toujours des murs
- La matrice utilise 1 pour bloquer et 0 pour le passage
- La taille est fixe pendant la génération

---

2. Architecture et Structure de Données

2.1 Variables globales expliquées

javascript

let GRID_SIZE = 32; *// Taille de la grille (modifiable selon le preset)*

Pourquoi variable ? La taille change selon le choix de l'utilisateur (24, 32, 48 ou 64).

javascript

const CELL_SIZE = 20; *// Taille en pixels pour l'affichage*

Pourquoi 20 ? C'est un bon compromis entre visibilité et taille totale du canvas.

javascript

let dungeon = []; *// La matrice principale*

let cellTypes = []; *// Types de cellules pour la coloration*

Deux matrices ?

- dungeon contient juste 0 ou 1 (pour le moteur de jeu)
- cellTypes contient des informations détaillées pour l'affichage (salle principale, corridor, labyrinthe, etc.)

2.2 La classe Random détaillée

javascript

```
class Random {
  constructor(seed) {
    this.seed = seed || Date.now();
  }
```

Concept de seed (graine) :

- Un nombre qui initialise le générateur
- Même seed = même séquence de nombres "aléatoires"
- Permet de reproduire exactement un donjon

javascript

```
  next() {
    this.seed = (this.seed * 9301 + 49297) % 233280;
    return this.seed / 233280;
  }
```

Linear Congruential Generator (LCG) :

- Formule : nouvelle_valeur = (ancienne * A + C) % M
- A=9301, C=49297, M=233280 sont des constantes choisies
- Division par M pour obtenir un nombre entre 0 et 1
- C'est déterministe : même seed → même séquence

2.3 La classe Room en détail

javascript

```
class Room {
  constructor(x, y, width, height, type = 'main') {
    this.x = x;               // Position coin haut-gauche
    this.y = y;
    this.width = width;         // Largeur en cellules
    this.height = height;       // Hauteur en cellules
    this.type = type;           // 'main', 'secondary', 'secret'
    this.centerX = Math.floor(x + width / 2);  // Centre pour les corridors
    this.centerY = Math.floor(y + height / 2);
    this.connections = [];      // Liste des indices de salles connectées
  }
```

Méthode intersects :

javascript

```
  intersects(other, padding = 1) {
    return !(this.x + this.width + padding <= other.x ||
```

```
        other.x + other.width + padding <= this.x ||
        this.y + this.height + padding <= other.y ||
        other.y + other.height + padding <= this.y);
  }
```

Logique d'intersection :

- Vérifie si deux rectangles se chevauchent
- padding ajoute une marge de sécurité
- Utilise la négation de "ne se touchent PAS"
- Si aucune de ces conditions n'est vraie, ils se chevauchent :
    - Ma droite est à gauche de son gauche
    - Sa droite est à gauche de ma gauche
    - Mon bas est au-dessus de son haut
    - Son bas est au-dessus de mon haut

---

3. Algorithme de Génération Aléatoire avec Seed

3.1 Pourquoi un générateur custom ?

JavaScript's Math.random() :

- Ne peut pas être initialisé avec une seed
- Résultats non reproductibles
- Problématique pour déboguer ou partager des donjons

Notre générateur :

- Reproductible avec la même seed
- Portable entre navigateurs
- Permet de sauvegarder/partager des donjons

3.2 Utilisation pratique

javascript

```
const rng = new Random(12345);  // Seed fixe

// Génère toujours la même séquence :
rng.nextInt(1, 10);  // Ex: 7
rng.nextInt(1, 10);  // Ex: 3
```

```
rng.nextInt(1, 10);  // Ex: 9

// Avec une autre seed :
const rng2 = new Random(54321);
rng2.nextInt(1, 10);  // Ex: 2 (différent)
```

---

4. Génération et Placement des Salles

4.1 Algorithme de placement aléatoire avec rejet

javascript

```
function placeMainRooms(rng, mode, density, zones = null) {
  const mainRooms = [];

  // Étape 1 : Calculer le nombre de salles souhaité
  const gridRatio = (GRID_SIZE * GRID_SIZE) / (32 * 32);
  const baseRoomCount = Math.floor((4 + Math.sqrt(gridRatio) * 3) * (density / 50));
```

Explication du calcul :

- gridRatio : Rapport entre la taille actuelle et la taille "standard" (32x32)
- 4 + Math.sqrt(gridRatio) * 3 : Formule empirique qui donne :
    - Petite carte (24x24) : ~3.5 salles de base
    - Moyenne (32x32) : 7 salles de base
    - Grande (48x48) : ~10 salles de base
- Multiplié par density/50 : Si densité=100%, on double le nombre

javascript

```
  // Étape 2 : Boucle de placement
  let attempts = 0;
  const maxAttempts = 1000 * gridRatio;

  while (mainRooms.length < baseRoomCount && attempts < maxAttempts) {
    attempts++;

    // Génère une salle aléatoire
```

```
    const width = rng.nextInt(minSize, maxSize);
    const height = rng.nextInt(minSize, maxSize);
    const x = rng.nextInt(2, GRID_SIZE - width - 2);
    const y = rng.nextInt(2, GRID_SIZE - height - 2);
```

Pourquoi commencer à 2 et finir à GRID_SIZE-2 ?

- Laisse une marge d'au moins 2 cellules avec le bord
- Espace pour les corridors et labyrinthes
- Évite que les salles touchent directement les murs extérieurs

javascript

```
    // Vérifie les intersections
    let valid = true;
    for (let room of mainRooms) {
      if (newRoom.intersects(room, padding)) {
        valid = false;
        break;
      }
    }

    if (valid) {
      mainRooms.push(newRoom);
      carveRoom(newRoom);
    }
  }
```

Algorithme de rejet :

1. Génère une position aléatoire
2. Vérifie si elle est valide (pas d'intersection)
3. Si oui : accepte et place
4. Si non : rejette et réessaie
5. Limite le nombre d'essais pour éviter boucle infinie

4.2 Ajustements selon les modes

javascript

```
switch (mode.name) {
  case 'Crypte':
    minSize = Math.max(2, minSize - 1);
    maxSize = Math.max(3, maxSize - 2);
    break;
  case 'Temple':
    minSize = Math.min(minSize + 1, maxSize);
    maxSize = Math.min(maxSize + 2, GRID_SIZE - 4);
    break;
}
```

Logique des ajustements :

- **Crypte** : Salles plus petites pour créer une ambiance confinée
- **Temple** : Salles plus grandes pour l'aspect monumental
- Math.max/min : Évite les valeurs impossibles

---

5. Algorithme de Kruskal pour la Connexion

5.1 Concept de l'arbre couvrant minimum

Problème : Connecter toutes les salles avec le minimum de corridors

Solution de Kruskal :

1. Considère toutes les connexions possibles
2. Trie par distance (plus court en premier)
3. Ajoute les connexions sans créer de cycle
4. Continue jusqu'à ce que tout soit connecté

5.2 Implémentation détaillée

javascript

```
function connectRooms(rooms, rng, mode) {
  // Étape 1 : Créer toutes les arêtes possibles
  const edges = [];
  for (let i = 0; i < rooms.length; i++) {
    for (let j = i + 1; j < rooms.length; j++) {
      edges.push({
```

```
          from: i,
          to: j,
          distance: rooms[i].distanceTo(rooms[j])
        });
      }
    }
```

Pourquoi toutes les arêtes ?

- N salles = N×(N-1)/2 connexions possibles
- On les génère toutes pour pouvoir choisir les meilleures
- Distance Manhattan : |x1-x2| + |y1-y2| (déplacement en grille)

javascript

```
  // Étape 2 : Trier par distance
  edges.sort((a, b) => a.distance - b.distance);
```

Tri croissant : Les connexions courtes sont privilégiées

javascript

```
  // Étape 3 : Union-Find pour éviter les cycles
  const parent = Array(rooms.length).fill().map((_, i) => i);

  function find(x) {
    if (parent[x] !== x) parent[x] = find(parent[x]);
    return parent[x];
  }

  function union(x, y) {
    parent[find(x)] = find(y);
  }
```

Union-Find (Disjoint Set) :

- Structure pour tracker les composantes connexes
- find(x) : Trouve le représentant du groupe de x
- union(x,y) : Fusionne les groupes de x et y
- Évite de créer des cycles (connexions redondantes)

javascript

```
// Étape 4 : Construire l'arbre couvrant
const connections = [];
for (let edge of edges) {
  if (find(edge.from) !== find(edge.to)) {
    union(edge.from, edge.to);
    connections.push(edge);
    // Creuse le corridor
    carveCorridorL(
      rooms[edge.from].centerX, rooms[edge.from].centerY,
      rooms[edge.to].centerX, rooms[edge.to].centerY, rng
    );
  }
}
```

5.3 Ajout de connexions supplémentaires

javascript

```
// Ajoute 30% de connexions en plus pour créer des boucles
const extraConnections = Math.floor(rooms.length * 0.3);
```

Pourquoi des boucles ?

- Évite les culs-de-sac frustrants
- Permet plusieurs chemins (gameplay intéressant)
- Stratégie : contourner des ennemis, exploration

---

6. Génération de Labyrinthes - Recursive Backtracking

6.1 Concept du labyrinthe parfait

Un labyrinthe "parfait" :

- Exactement un chemin entre deux points
- Pas de boucles
- Pas de zones inaccessibles
- Forme un arbre si on le visualise comme un graphe

6.2 Pattern de grille pour labyrinthes

Grille conceptuelle :

= Mur potentiel

. # . # . = Cellule du labyrinthe

# # # # #

. # .

# # # # #

Pourquoi des coordonnées impaires ?

- Les cellules sont sur (1,1), (1,3), (3,1), (3,3)...
- Les murs entre elles sont sur les coordonnées paires
- Garantit des passages d'une cellule de large

6.3 Algorithme Recursive Backtracking détaillé

javascript

```
function generateMazeInRegion(region, rng, complexity) {
  // Filtre pour ne garder que les cellules impaires
  const filteredRegion = region.filter(cell =>
    cell.x % 2 === 1 && cell.y % 2 === 1
  );
```

Étapes de l'algorithme :

1. **Initialisation** :

javascript

```
const stack = [];
const visited = new Set();
const start = filteredRegion[rng.nextInt(0, filteredRegion.length - 1)];
visited.add(`${start.x},${start.y}`);
stack.push(start);
```

2. **Boucle principale** :

javascript

```
while (stack.length > 0) {
  const current = stack[stack.length - 1];

  // Trouve les voisins non visités
```

```
  const neighbors = [];
  const directions = [
    {dx: 0, dy: -2, wallX: 0, wallY: -1},  // Nord
    {dx: 2, dy: 0, wallX: 1, wallY: 0},    // Est
    {dx: 0, dy: 2, wallX: 0, wallY: 1},    // Sud
    {dx: -2, dy: 0, wallX: -1, wallY: 0}   // Ouest
  ];
```

Pourquoi distance de 2 ?

- On saute par-dessus le mur entre deux cellules
- wallX/wallY : Position du mur à creuser

3. **Creuser ou backtracker** :

javascript

```
  if (neighbors.length > 0) {
    // Choisit un voisin aléatoire
    const next = neighbors[rng.nextInt(0, neighbors.length - 1)];

    // Creuse le passage
    dungeon[next.wallY][next.wallX] = FLOOR;
    dungeon[next.y][next.x] = FLOOR;

    visited.add(`${next.x},${next.y}`);
    stack.push(next);
  } else {
    // Pas de voisin : backtrack
    stack.pop();
  }
```

Mécanisme de backtracking :

- Avance tant que possible
- Quand bloqué, revient en arrière
- Garantit de visiter toutes les cellules
- Crée un labyrinthe sans boucles

6.4 Réduction de complexité

javascript

```
if (complexity < 1) {
  const loopsToAdd = Math.floor(visited.size * (1 - complexity) * 0.2);
  // Ajoute des boucles en supprimant des murs
}
```

Complexité variable :

- 100% = Labyrinthe parfait (difficile)
- 50% = Quelques boucles (équilibré)
- 0% = Beaucoup de boucles (facile)

---

7. Pathfinding A* et Analyse de Distance

7.1 Algorithme A* expliqué

A* trouve le chemin le plus court entre deux points :

javascript

```
function findPath(start, end) {
  const openSet = [{...start, f: 0, g: 0, h: 0, parent: null}];
  const closedSet = new Set();
```

Composants clés :

- g : Coût depuis le départ
- h : Estimation jusqu'à l'arrivée (heuristique)
- f = g + h : Score total
- openSet : Nœuds à explorer
- closedSet : Nœuds déjà explorés

7.2 Heuristique Manhattan

javascript

```
const h = Math.abs(nx - end.x) + Math.abs(ny - end.y);
```

Distance Manhattan :

- Adaptée aux grilles (pas de diagonales)
- Toujours optimiste (ne surestime jamais)
- Garantit de trouver le chemin optimal

7.3 Utilisation pour la qualité

javascript

```
function calculatePathDistance() {
  const path = findPath(startPos, endPos);
  return path ? path.length : 0;
}
```

Pourquoi mesurer la distance ?

- Trop court = Donjon trop facile
- Assure une exploration minimale
- Critère objectif de qualité

---

8. Système de Scoring et Métriques de Qualité

8.1 Philosophie du scoring

Le score reflète la "jouabilité" du donjon :

- **Exploration** : Distance suffisante S→E
- **Densité** : Équilibre espaces/murs
- **Choix** : Chemins multiples
- **Récompenses** : Culs-de-sac pour le loot

8.2 Calcul détaillé

javascript

```
function analyzeDungeonQuality() {
  let score = 100;  // On part du maximum

  // Distance minimum adaptée à la taille
  const sizeMultiplier = GRID_SIZE / 32;
  const minDistance = Math.floor(15 * sizeMultiplier);
```

Adaptation à la taille :

- Petite carte (24x24) : Distance min ~11
- Moyenne (32x32) : Distance min 15
- Grande (48x48) : Distance min ~22

8.3 Critères individuels

Ratio d'espaces libres :

javascript

```
const openRatio = (floorCount / (GRID_SIZE * GRID_SIZE)) * 100;
if (openRatio < 25) warnings.push("Trop dense");
if (openRatio > 60) warnings.push("Trop vide");
```

Logique :

- < 25% : Claustrophobe, peu de marge de manœuvre
- 60% : Trop ouvert, perd l'aspect "donjon"
- 25-60% : Équilibre exploration/contrainte

---

9. Optimisations et Patterns de Code

9.1 Utilisation de Sets pour performance

javascript

```
const regionSet = new Set(region.map(c => `${c.x},${c.y}`));
// Test rapide : O(1) au lieu de O(n)
if (regionSet.has(`${nx},${ny}`)) { ... }
```

Avantages :

- Recherche en temps constant
- Évite les boucles imbriquées
- Clé string pour coordonnées 2D

9.2 Early exit patterns

javascript

```
for (let room of rooms) {
  if (newRoom.intersects(room, padding)) {
    valid = false;
    break;  // Sort dès qu'on trouve une intersection
  }
}
```

Principe : Arrêter dès qu'on a la réponse

9.3 Copie d'état pour la génération en masse

javascript

```
function saveDungeonState() {
  return {
    dungeon: dungeon.map(row => [...row]),  // Copie profonde
    rooms: rooms.map(r => ({...r})),       // Copie des objets
    // ...
  };
}
```

Pourquoi copier ?

- Évite les références partagées
- Permet de naviguer entre états
- Isolation des données

---

10. Guide Pratique d'Extension

10.1 Ajouter un nouveau type de salle

javascript

```
// 1. Ajouter le type dans CELL_TYPES
const CELL_TYPES = {
  // ...
  TREASURE: 7
};

// 2. Créer une fonction de placement
function placeTreasureRooms(rng, mainRooms) {
  // Sélectionner des culs-de-sac éloignés
  const candidates = deadEndsList.filter(de => {
    const distToStart = Math.abs(de.x - startPos.x) + Math.abs(de.y - startPos.y);
    return distToStart > GRID_SIZE / 2;
  });

  // Placer 1-3 salles au trésor
  const count = rng.nextInt(1, 3);
```

```
  for (let i = 0; i < count && candidates.length > 0; i++) {
    const idx = rng.nextInt(0, candidates.length - 1);
    const pos = candidates.splice(idx, 1)[0];

    // Agrandir le cul-de-sac en salle
    for (let dy = -1; dy <= 1; dy++) {
      for (let dx = -1; dx <= 1; dx++) {
        const x = pos.x + dx;
        const y = pos.y + dy;
        if (x > 0 && x < GRID_SIZE-1 && y > 0 && y < GRID_SIZE-1) {
          dungeon[y][x] = FLOOR;
          cellTypes[y][x] = CELL_TYPES.TREASURE;
        }
      }
    }
  }
}

// 3. Ajouter la couleur dans drawDungeon
case CELL_TYPES.TREASURE:
  color = '#ffd700';  // Or
  break;
```

10.2 Implémenter un nouveau mode architectural

```
javascript
// Exemple : Mode "Ruines"
DUNGEON_MODES.ruins = {
  name: "Ruines",
  description: "Structures partiellement effondrées, beaucoup d'obstacles",
  roomShape: "irregular",
  corridorStyle: "broken",
  roomSizeVariance: 0.9,
```

```
  mazeComplexity: 0.3,
  symmetry: 0.2,

  // Paramètres custom
  wallDensity: 0.3,  // 30% de murs supplémentaires
  rubbleChance: 0.4  // 40% de chance de débris
};

// Dans generateDungeon, après la génération normale :
if (mode.name === "Ruines") {
  addRubbleAndDebris(rng, mode.wallDensity, mode.rubbleChance);
}

function addRubbleAndDebris(rng, density, chance) {
  // Ajoute des murs isolés dans les grandes salles
  for (let room of rooms.filter(r => r.type === 'main')) {
    if (rng.chance(chance)) {
      const rubbleCount = Math.floor(room.width * room.height * density);
      for (let i = 0; i < rubbleCount; i++) {
        const x = rng.nextInt(room.x + 1, room.x + room.width - 2);
        const y = rng.nextInt(room.y + 1, room.y + room.height - 2);

        // Évite de bloquer complètement
        if (hasMultiplePaths(x, y)) {
          dungeon[y][x] = WALL;
          cellTypes[y][x] = CELL_TYPES.RUBBLE;
        }
      }
    }
  }
}
```

10.3 Système de portes

javascript

```
// Trouver les intersections salle/corridor
function findDoorPositions() {
  const doors = [];

  for (let y = 1; y < GRID_SIZE - 1; y++) {
    for (let x = 1; x < GRID_SIZE - 1; x++) {
      if (dungeon[y][x] === FLOOR) {
        // Vérifie si c'est une transition
        const isRoomToCorridor =
          (cellTypes[y][x] === CELL_TYPES.ROOM &&
           hasAdjacentType(x, y, CELL_TYPES.CORRIDOR)) ||
          (cellTypes[y][x] === CELL_TYPES.CORRIDOR &&
           hasAdjacentType(x, y, CELL_TYPES.ROOM));

        if (isRoomToCorridor && isValidDoorPosition(x, y)) {
          doors.push({x, y, type: 'normal'});
        }
      }
    }
  }

  return doors;
}

function isValidDoorPosition(x, y) {
  // Une porte doit avoir des murs sur 2 côtés opposés
  const horizontal = dungeon[y][x-1] === WALL && dungeon[y][x+1] === WALL;
  const vertical = dungeon[y-1][x] === WALL && dungeon[y+1][x] === WALL;
```

```
    return (horizontal && !vertical) || (!horizontal && vertical);
}
```

Conclusion

Cette documentation approfondie couvre tous les aspects techniques du générateur de donjons. Chaque algorithme est expliqué avec :

- Le problème qu'il résout
- Son fonctionnement détaillé
- Les raisons des choix d'implémentation
- Des exemples concrets d'utilisation

Le code est conçu pour être :

- **Modulaire** : Chaque partie peut être modifiée indépendamment
- **Extensible** : Facile d'ajouter de nouvelles fonctionnalités
- **Compréhensible** : Avec cette documentation, vous pouvez maintenir et faire évoluer le projet