# Dossier complet – Créer un jeu *inspiré* de Money Cart 4 avec PixiJS

> **Note légale & éthique**
> Ce document explique comment réaliser **un jeu original inspiré** de mécaniques de type "Hold & Win/Respin" popularisées par divers éditeurs (dont Relax Gaming avec *Money Cart 4*). **N'utilise pas d'actifs visuels/sonores protégés** ni de noms déposés. Crée tes propres visuels, sons et intitulés. Ce guide est purement technique/éducatif et **ne propose ni jeu d'argent réel**.

---

## 0) Aperçu du gameplay ciblé (version inspirée)

- **Grille** : 6 colonnes × 4 rangées, extensible jusqu'à 6×8 (rangées verrouillées au départ).
- **Phase principale (base game, optionnelle)** : spins rapides ; des symboles Bonus peuvent déclencher la **phase Respin**.
- **Phase Respin (cœur du jeu)** : on commence avec 3 respins. Chaque symbole qui atterrit **réinitialise à 3**.
- **Cellules persistantes** : les symboles restent affichés pendant la phase Respin, additionnent/modifient les valeurs et déclenchent des **modificateurs spéciaux**.
- **Déblocage de rangées** : conditions (par ex. symbole Expander/Unlocker, ou N symboles posés dans la grille).
- **Fin de bonus** : plus de respins → on **score** en additionnant les valeurs des symboles.
- **Volatilité élevée** : rares gros gains.

---

## 1) Pré-requis & stack technique

- **PixiJS v7+** (rendu WebGL/Canvas 2D fallback).
- **Vite** (ou Webpack) pour dev/serveur local & bundling.
- **GSAP** (ou `@pixi/animate`, ou tweens maison) pour les animations.
- **Spritesheets** générés via TexturePacker/Free-Texture-Packer.
- **Howler.js** (ou WebAudio natif) pour l'audio.
- **TypeScript** recommandé (robustesse), mais JS pur possible.

---

## 2) Structure de projet recommandée

```
my-holdwin-game/
├─ assets/
│  ├─ fonts/
│  ├─ sfx/
│  ├─ music/
│  ├─ spritesheets/
│  │  ├─ symbols.json / symbols.png
│  │  ├─ ui.json / ui.png
```

```
│  │  └─ fx.json / fx.png
├─ src/
│  ├─ core/
│  │  ├─ App.ts           // boot PIXI, loader, resize
│  │  ├─ Assets.ts        // déclaration des packs d'assets
│  │  ├─ Audio.ts         // sons/musiques
│  │  ├─ RNG.ts           // RNG, seed, distributions
│  │  ├─ Signals.ts       // EventBus (mitt, tiny-emitter…)
│  │  └─ Tweens.ts        // helpers GSAP/tweens
│  ├─ game/
│  │  ├─ Game.ts          // orchestrateur, state machine
│  │  ├─ states/
│  │  │  ├─ BootState.ts
│  │  │  ├─ MenuState.ts
│  │  │  ├─ BaseSpinState.ts
│  │  │  └─ RespinState.ts
│  │  ├─ grid/
│  │  │  ├─ Grid.ts       // placement, unlock logic
│  │  │  ├─ Cell.ts
│  │  │  └─ Symbol.ts     // classe de symbole générique
│  │  ├─ symbols/
│  │  │  ├─ CoinSymbol.ts         // valeur fixe (x1, x2…)
│  │  │  ├─ CollectorSymbol.ts    // collecte des voisins
│  │  │  ├─ PayerSymbol.ts        // paye aux autres
│  │  │  ├─ SniperSymbol.ts       // multiplie des cibles
│  │  │  ├─ NecromancerSymbol.ts  // réactive des symboles
│  │  │  ├─ ExpanderSymbol.ts     // débloque une rangée
│  │  │  └─ Persistent* variants  // s'activent chaque tour
│  │  ├─ ui/
│  │  │  ├─ HUD.ts        // solde, mise, compteur respins
│  │  │  ├─ Buttons.ts
│  │  │  └─ WinPanel.ts
│  │  ├─ effects/
│  │  │  ├─ Particles.ts  // confettis, sparks, trails
│  │  │  └─ Screenshake.ts
│  │  └─ math/
│  │     ├─ Paytable.ts   // tables de poids, RTP target
│  │     └─ Simulator.ts  // sim Monte Carlo debug
│  └─ main.ts
├─ index.html
├─ package.json
└─ vite.config.ts
```

---

## 3) Installation rapide

**package.json (exemple)**

```
{
  "name": "holdwin-pixijs",
  "version": "1.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "preview": "vite preview"
  },
  "dependencies": {
    "pixi.js": "^7.4.0",
    "gsap": "^3.12.5",
    "howler": "^2.2.4",
    "mitt": "^3.0.1"
  },
  "devDependencies": {
    "typescript": "^5.6.2",
    "vite": "^5.3.3"
  }
}
```

**Boot minimal (src/core/App.ts)**

```
import { Application, Container, Assets } from 'pixi.js'

export class AppCore {
  app: Application
  root: Container

  constructor(parent: HTMLElement) {
    this.app = new Application({
      background: '#0b0f14',
      resolution: devicePixelRatio,
      antialias: true,
      autoDensity: true
    })
    parent.appendChild(this.app.view as HTMLCanvasElement)

    this.root = new Container()
    this.app.stage.addChild(this.root)

    window.addEventListener('resize', () => this.resize())
    this.resize()
  }

  async load(packs: any[]) {
    for (const pack of packs) await Assets.load(pack)
  }
```

```
  resize() {
    const w = window.innerWidth
    const h = window.innerHeight
    this.app.renderer.resize(w, h)
    // Adapter root scale/position ici si besoin
  }
}
```

---

## 4) Assets & spritesheets

**Conseils** - Conçois des **symboles lisibles** : pièces (Coin), modificateurs (Collector, Payer, Sniper, Necromancer...), verrous de rangée, multiplicateurs, etc.
- Exporte **@2x** pour les écrans haute densité.
- Prévois des **états** : idle, spawn, hit, persist, glow, disabled.
- Regroupe par familles : `symbols`, `ui`, `fx`.

**Spritesheets** (TexturePacker / Free Texture Packer)
- Génère un `symbols.png` + `symbols.json` (format Pixi).
- Nommer les frames par convention : `coin_idle_000.png`, `coin_spawn_000.png`, etc.
- Pour les FX, des cycles courts (6–12 frames) suffisent avec un léger motion blur.

---

## 5) Architecture & State Machine

**États typiques** 1. **BootState** : chargement assets, affichage logo.
2. **MenuState** : choix mise, bouton "Jouer".
3. **BaseSpinState** (optionnel) : spins rapides, détecte déclencheur Respin.
4. **RespinState** : cœur du jeu, boucle respins, gestion symboles persistants.
5. **WinPanel** : totalisation gains, effets, retour au menu.

**Pattern conseillé** : **Command/Action Queue** dans `RespinState` pour séquencer les animations (spawn → resolve modifs → count → ui update → next).
**Event Bus** (`mitt`) pour découpler : `SYMBOL_SPAWNED`, `RESPIN_RESET`, `ROW_UNLOCKED`, `TOTAL_UPDATED`, etc.

---

## 6) La grille

- **Coordonnées** : (col, row), `0 ≤ col < 6`, `0 ≤ row < currentRows`.
- **Cellule** : contient un `Symbol` (ou vide) + décor (cadre, lueur).
- **Rangées verrouillées** : représentées par un calque visuel semi-opaque avec cadenas.
- **Placement** : une `GridLayout` calcule les positions PIXEL de chaque cellule selon la taille d'écran (layout réactif).

**Grid.ts – idées clés**

```
class Grid extends Container {
  cols = 6
  rows = 4
  maxRows = 8
  cells: Cell[][] = []

  constructor() { super(); this.build(); }

  build() {
    this.cells = []
    for (let c=0; c<this.cols; c++) {
      const col: Cell[] = []
      for (let r=0; r<this.rows; r++) {
        const cell = new Cell(c, r)
        this.addChild(cell)
        col.push(cell)
      }
      this.cells.push(col)
    }
    this.layout()
  }

  layout() {
    // calcule x/y de chaque cell selon viewport
  }

  unlockRow(direction: 'top'|'bottom') {
    if (this.rows >= this.maxRows) return
    this.rows++
    // créer visuellement la nouvelle rangée, animation slide
  }
}
```

---

## 7) Système de symboles (modèle objet)

**Base**

```
export interface SymbolContext {
  grid: Grid
  rng: RNG
  signals: Emitter
}

export abstract class SymbolBase extends Container {
  type: string
  value: number = 0
  persistent: boolean = false
```

```
  ctx: SymbolContext

  constructor(type: string, ctx: SymbolContext) { super(); this.type = type;
this.ctx = ctx }

  /** Appelé quand le symbole apparaît */
  async onSpawn(cell: Cell) {}
  /** Appelé à la résolution du tour (tick de respin) */
  async onResolve() {}
}
```

**Exemples de symboles** - **CoinSymbol** : valeur fixe (ex: x1, x2, x5…).
- **CollectorSymbol** : additionne toutes les valeurs visibles et l'ajoute à **lui-même**.
- **PayerSymbol** : ajoute sa valeur à **tous les autres** symboles (ou à N aléatoires).
- **SniperSymbol** : choisit K cibles et **multiplie** leur valeur (×2/×3…).
- **NecromancerSymbol** : "réactive" X symboles utilisés (ou à 0) pour **relancer** leurs onResolve.
- **ExpanderSymbol / Unlocker** : débloque une rangée (haut/bas).
- **Persistent** *: mêmes effets mais à chaque tour de respin.*
- *Resetter** : ajoute +1 au compteur de respins (ou fixe à 3).

**Collector – esquisse**

```
export class CollectorSymbol extends SymbolBase {
  constructor(ctx: SymbolContext) { super('collector', ctx) }
  async onResolve() {
    const total = this.ctx.grid.sumValues()
    await animatePulse(this)
    this.value += total
    this.ctx.signals.emit('TOTAL_UPDATED')
  }
}
```

---

## 8) Boucle Respin (logique & séquencement)

**Variables** : `respinsLeft = 3`, `symbolsPlacedThisSpin = 0`.

**Cycle** 1. **Spawn Step** : on tente de faire apparaître 0..N nouveaux symboles dans des cellules vides selon une **distribution/poids** (probas par type).
2. Si ≥1 symbole spawné → `respinsLeft = 3`, else `respinsLeft--`.
3. **Resolve Step** : pour chaque symbole (ordre déterministe pour lisibilité), on exécute `onResolve()`.
4. **Row Unlock** : si condition atteinte → `grid.unlockRow()` avec animation.
5. **UI Update** : compteur, totaux, jingles.
6. **Fin** : si `respinsLeft === 0` → totalisation & sortie.

**Orchestrateur (pseudo)**

```
async function respinLoop() {
  while (respinsLeft > 0) {
    const spawned = await spawnPhase()
    if (spawned > 0) respinsLeft = 3
    else respinsLeft--

    await resolvePhase()  // séquenceActions()
    await maybeUnlockRow()
    await updateUI()
  }
  await showWinPanel()
}
```

---

# 9) Animations – explications détaillées

## 9.1 Spawn d'un symbole (pop-in)

- **But** : donner du *poids* au symbole qui arrive.
- **Étapes** : 1) Crée le conteneur du symbole hors-écran (ou en très petit scale).
  2) "*Anticipation*" : légère descente/écrasement (scaleY > 1, scaleX < 1).
  3) *Pop* : scale vers 1 avec *overshoot* (ease back), petit **screenshake doux** de la cellule.
  4) **Spark FX** : particules étoiles 100–200 ms.
- **GSAP** (exemple) : `gsap.fromTo(sym.scale, {x:0.3,y:0.3},{x:1,y:1,duration:0.35,ease:'back.out(1.8)'})`.

## 9.2 Highlight de cellule active

- **But** : indiquer le focus du tick.
- **Effet** : halo pulsant + anneau qui s'élargit et fade.
- **Loop** : faible intensité hors action, intensité forte au moment de `onResolve()`.

## 9.3 Effet Collector

- **Lignes de collecte** : des *beams* (traits fins) partent des autres symboles vers le Collector.
- **Somme** : compteur **floating text** qui s'additionne au-dessus du Collector.
- **Audio** : *sucking/woosh* crescendo + *coin add* à l'impact.

## 9.4 Effet Payer

- **Ondes circulaires** depuis le Payer, qui touchent les voisins et déclenchent un petit **bump** (scale 1→1.08→1).
- **Texte flottant** `+v` au-dessus de chaque cible.

## 9.5 Effet Sniper

- **Viseur** : reticule qui **verrouille K cibles** (laser fin).
- **Multiplication** : texte `×2` / `×3` qui éclate en éclats polygonaux.
- **Sound** : *click* + *laser ping*.

### 9.6 Effet Necromancer

- **Âmes** : particules fantomatiques bleu/vert qui ravivent **X symboles** (halo qui redevient coloré).
- **Relance** : rejoue l'animation `onResolve()` des cibles.

### 9.7 Déblocage de rangée (Unlocker/Expander)

- Le **layer verrou** glisse vers l'extérieur (haut/bas) en 300–450 ms.
- La **nouvelle rangée** "entre" avec un *slide & fade* + *snap* des cellules.
- Petit **camera nudge** (screenshake vertical) pour l'impact.

### 9.8 Compteur de respins

- Compteur mécanique **3 → 2 → 1**, à chaque décrément : *clack* + léger flash de l'UI.
- Quand réinitialisé à 3 : *ding* montant + particules bleues.

### 9.9 Totalisation finale

- **Count-up** (0 → total) sur 1–2 s, synchronisé à un jingle.
- Confettis légers si gros gain (> seuil).
- Bouton "Continuer" apparaît après 800 ms.

---

## 10) UI & ergonomie

- **HUD** : affichage mise, bankroll, multiplicateur total, respins restants.
- **Boutons** : Auto, Turbo, Sound, Menu.
- **Lisibilité mobile** : tailles min ($\geq$ 14 px texte), **touch areas** $\geq$ 40 px.
- **Accessibilité** : contraste suffisant, feedback audio désactivable, vibration (mobile) optionnelle.

---

## 11) Audio & feedback

- **SFX courts** ($\leq$ 300 ms) pour actions fréquentes.
- **Layers adaptatifs** : pendant Respins, musique en *ostinato* + couche d'intensité qui monte avec les rangées débloquées.
- **Sidechain** : diminuer légèrement la musique pendant les SFX importants (ducking).

---

## 12) Math, poids & RNG

- **RNG** : `Mulberry32` / `XorShift` ou `crypto.getRandomValues` (non seedable).
- **Seeding** pour replays/débug.
- **Tables de poids** : proba de chaque type de symbole par **niveau d'extension** (4→8 rangées), avec plafonds de valeur.
- **Cap de valeur** pour éviter l'infini lors de combos *Persistent Payer* × *Collector*, etc.
- **Simulateur Monte Carlo** (outil debug) pour approx. RTP/volatilité **en sandbox** (sans argent réel).

**Exemple de tables (simplifiées)**

```
const WEIGHTS = {
  rows4: { coin: 70, payer: 8, collector: 8, sniper: 6, necro: 3, expander:
5 },
  rows6: { coin: 62, payer: 10, collector: 10, sniper: 8, necro: 4,
expander: 6 },
  rows8: { coin: 55, payer: 12, collector: 12, sniper: 10, necro: 5,
expander: 6 }
}
```

---

## 13) Flux détaillé de la RespinState

1. `enter()` : prépare compteur respins, verrouille les entrées UI, focus caméra.
2. `spawnPhase()` : choisit 0..N cellules vides ; *pop-in* + SFX.
3. `resolvePhase()` : pour chaque symbole (ordre stable) :
4. jouer FX de focus,
5. exécuter logique,
6. mettre à jour `value` + UI + particules.
7. `rowUnlockCheck()` : si **X symboles** posés ou **Expander** vu → `grid.unlockRow()`.
8. `endTick()` : si `spawned>0` → `respins=3` else `respins--`.
9. `exit()` : calcul du **total** + `WinPanel`.

---

## 14) Performances & qualité

- **Batches** : évite les filtres lourds ; privilégie `cacheAsBitmap` pour les halos statiques.
- **Particles** : réutilise via **pools** (objet pool).
- **Textes** : `BitmapText` pour compteurs fréquents.
- **Resolution** : `app.renderer.resolution = devicePixelRatio`, mais limite à 2 sur mobile si besoin.
- **GC friendly** : recycle les Symbol/Cell.

---

## 15) Mobile & responsive

- **Layouts** portrait/paysage : grille centrée, HUD adaptatif.
- **Safe areas** (iPhone X+).
- **Touch input** : gestuelle simple, pas de drag long.
- **FPS cible** : 60 mais stable à 30 acceptable.

---

## 16) Persistance & réglages

- **LocalStorage** : volume SFX/musique, turbo, seed debug.
- **Sauvegarde de session (facultatif)** : restaurer une Respin en cours après rafraîchissement (journal d'actions à rejouer).

---

## 17) Debug & outils créateurs

- **Panneau debug** :
- Forcer `rows=8`,
- Forcer prochain symbole (payer/collector…),
- Multiplier la valeur d'une cellule,
- Lancer `Simulator.run(10000 spins)` et afficher histo gains.
- **Replay** : input d'un `seed` + `step index` pour reproduire un run.

---

## 18) Exemples de code clés

### 18.1 RNG seedable (Mulberry32)

```
export function mulberry32(a:number){return function(){let
t=a+=0x6D2B79F5;t=Math.imul(t^t>>>15,t|1);t^=t+Math.imul(t^t>>>7,t|
61);return((t^t>>>14)>>>0)/4294967296}}
```

### 18.2 Sélection pondérée

```
export function weightedPick(table:Record<string,number>, rnd:()=>number){
  const total = Object.values(table).reduce((a,b)=>a+b,0)
  let r = rnd()*total
  for (const [k,w] of Object.entries(table)) { if ((r-=w)<0) return k }
  return Object.keys(table)[0]
}
```

### 18.3 Spawn phase (simplifiée)

```
async function spawnPhase(){
  let spawned = 0
  const empties = grid.getEmptyCells()
  const slotsToFill = Math.min(empties.length, rng.int(0,2)) // 0..2
  for (let i=0;i<slotsToFill;i++){
    const type = weightedPick(WEIGHTS[rowsKey()], rng.next)
    const sym = createSymbol(type)
    await grid.placeSymbol(empties[i], sym)
    await sym.onSpawn(empties[i])
    spawned++
  }
  return spawned
}
```

### 18.4 Resolve phase (séquencée)

```
async function resolvePhase(){
  const symbols = grid.getAllSymbols()
  for (const s of symbols){
    await highlightCell(s.cell)
    await s.onResolve()
  }
}
```

---

## 19) Tables d'équilibrage – idées

- **Caps** : `value ≤ 10000x` par cellule.
- **Limites de cibles** (Sniper, Payer) par tick.
- **Rareté croissante** avec plus de rangées (mais gains potentiels ↑).
- **Éviter boucles** : ne pas laisser `Persistent Payer` buffer sur `Collector` de façon explosive sans plafonds.

---

## 20) Qualité visuelle – direction artistique

- **Style** : steampunk/faro-ouest futuriste (ou autre identité forte).
- **Palette** : fonds sombres, symboles chauds (or/cuivre) + accent cyan/émeraude pour modificateurs.
- **Lisibilité** > décor : désature l'arrière-plan, *bloom* léger sur symboles actifs.
- **FX** : préférer des **courtes** animations réactives à des longues cinématiques.

---

## 21) Audio design – détails pratiques

- **Layering** : *bed* musical discret en boucle + stems d'intensité.
- **Cues** : spawn, resolve, unlock, respin reset, big win.
- **Mix** : limiter les fréquences 2–5 kHz des SFX pour éviter la fatigue.

---

## 22) Test plan

- **Unitaires** : RNG, weighted pick, caps.
- **Intégration** : ordre de résolution, collisions d'effets (Payer→Collector).
- **UX** : compréhension du joueur sans tutoriel (tooltips facultatifs).
- **Perf** : 100 symboles + FX simultanés ≤ 8 ms/frame sur desktop milieu de gamme.

---

## 23) Feuille de route (itérations)

1) **MVP Respin** : grille 6×4, Coin+Collector+Payer, respin 3→0, totalisation.
2) **Déblocage de rangées** + Sniper, Necromancer.
3) **Persistent variants** + tuning poids.
4) **Audio complet** + polish FX.
5) **Panneau debug + simulateur**.
6) **Optim mobile + UI finale**.

---

## 24) Conseils de publication

- **Héberge** sur Netlify/Vercel/GitHub Pages.
- Fournis un **seed** pour rejouer une session à l'identique (utile pour QA).
- Évite toute mention ou ressemblance forte avec marques déposées.

---

## 25) Aller plus loin

- **Spine** (optionnel) pour des animations de symboles complexes (via `pixi-spine`).
- **Shaders** Pixi (filters custom) pour lueur, aberration chromatique légère, distorsion de chaleur.

---

### Annexes

- **Checklist d'assets** : idle/spawn/hit/persist pour chaque symbole, verrous de rangée, halo cell, beams, reticule, confettis, counters, boutons, panneaux.
- **Nommage** : `symbol_<type>_<state>_<frame>.png`.
- **Tailles** : base 256px (idle) → downscale runtime selon viewport.

---

Si tu veux, je peux **ajouter des fichiers de départ** (Vite + Pixi + classes vides) et un **spritesheet d'exemple** avec des formes géométriques pour tester immédiatement. Dis-moi si tu veux que je te génère un squelette de projet minimal 👇