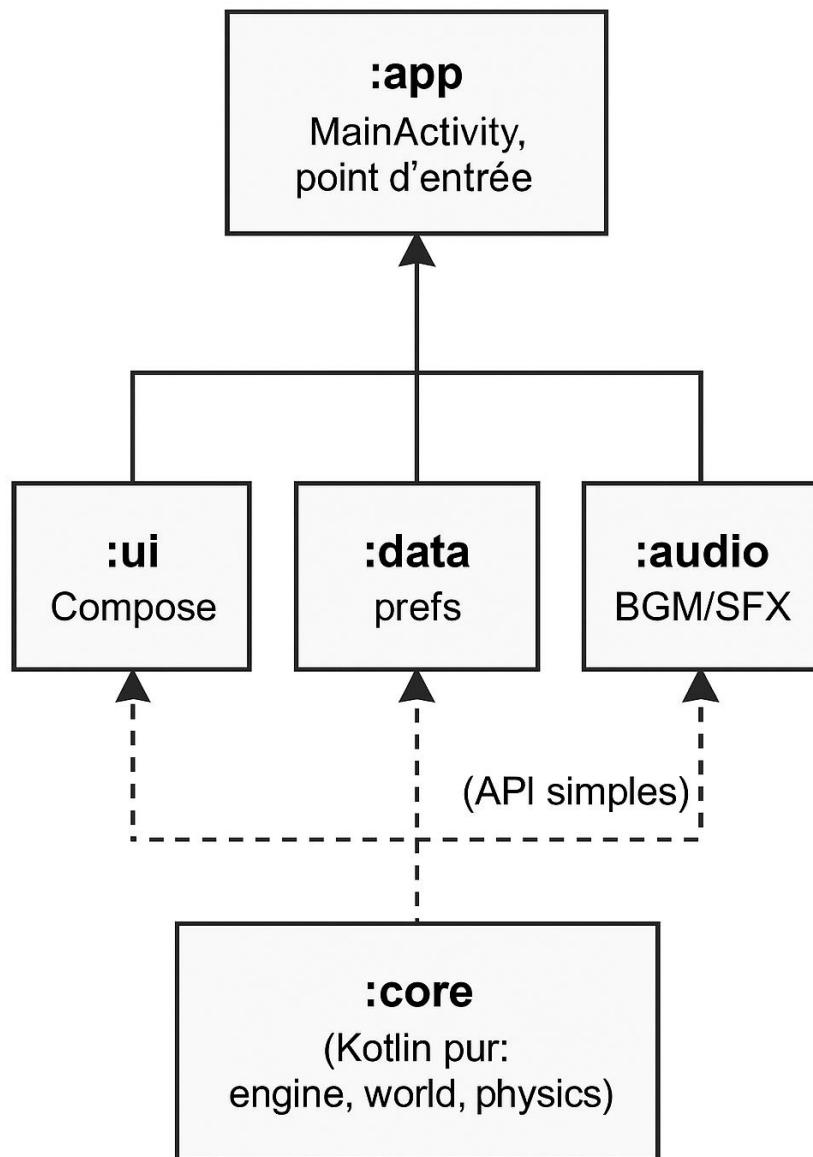


## 1. Diagramme de composants (modules & dépendances)



Contraintes : **:core** ne dépend de personne (pur Kotlin) ; **:ui** dépend de **:core** et **:data** ; **:app** assemble.

### Détail des composants

#### 1 :app

- Type : module Android Application

- **Rôle** : point d'entrée du jeu (MainActivity).
- **Responsabilités** :
  - Lance l'UI via RootNav().
  - Gère le cycle de vie Android.
  - Assemble tous les modules.
- **Dépend de** : :ui, :data, :audio.

## 2 :ui

- **Type** : module Android Library (Compose).
- **Rôle** : interface visuelle (menus, écrans, HUD).
- **Responsabilités** :
  - Écrans (GameScreen, HomeScreen...).
  - Thèmes, composants, ViewModels (MVVM).
  - Navigation entre les écrans.
- **Dépend de** : :core (moteur logique) et :data (sauvegarde).

## 3 :core

- **Type** : module Kotlin pur (pas d'Android).
- **Rôle** : moteur du jeu.
- **Responsabilités** :
  - Boucle principale (GameLoop).
  - Monde (World), entités (Dragon, obstacles...).
  - Physique, collisions, logique pure.
- **Ne dépend de personne** → il est totalement isolé.

## 4 :data

- **Type** : module Android Library.
- **Rôle** : gestion des données et préférences.
- **Responsabilités** :
  - Stockage avec DataStore.
  - Sauvegarde des paramètres et scores.

- Fournir un GameRepository accessible au ViewModel.
- **Peut dépendre de rien** (il expose juste des services).

## 5 :audio

- **Type** : module Android Library.
- **Rôle** : gestion des sons et musiques.
- **Responsabilités** :
  - AudioManager, Sfx (effets sonores).
  - Contrôle volume, lecture, crossfade, stop.
- **Indépendant** du moteur ou des données.

## ⌚ Avantages de cette structure

- **Modularité** → tu peux tester ou compiler chaque partie séparément.
- **Réutilisabilité** → core pourrait servir à un autre jeu.
- **Performances** → compilation parallèle (Gradle) accélérée.
- **Isolation** → le moteur (core) ne dépend pas d'Android.
- **Évolutivité** → ajouter un module (ex. :network) sera simple.

## 2) Diagramme de classes — cœur minimal

package core.engine

- + GameLoop(scope, tickMs, onUpdate)
- + start()
- + stop()

package core.world

- + World
- + update(dtMs)
- + entities: List<Entity>

package core.entity

<<interface>> Entity

- + update(dtMs)
- + pos: Vec2

+ Dragon : Entity

- + light: Float
- + update(dtMs)

package core.physics

- + Collider
- + bounds(): AABB
- + CollisionSystem
- + step(entities)

 **Explication classe par classe**

 **GameLoop (core.engine)**

**Rôle :** cœur du moteur, cadence le jeu (~60 FPS).

**Fonctionnement :**

- Lance une coroutine qui appelle onUpdate(dt) toutes les tickMs (par défaut 16 ms).
- C'est ce callback qui fait "vivre" le monde.

**Pseudo-code :**

```
while (running) {  
    val dt = now - last  
  
    onUpdate(dt)    // demande une mise à jour du monde  
  
    delay(16)       // 60 FPS  
}
```

**Responsabilités :**

- Gérer le temps (dtMs).
- Appeler la logique (World.update()).
- S'arrêter proprement (stop()).

---

## World (core.world)

**Rôle :** représente l'univers courant du jeu.

**Contient :**

- Une liste d'**entités** (Entity).
- Une référence au système de collisions.

**Méthodes :**

- update(dt) : met à jour toutes les entités (mouvement, physique, etc.).
- Appelle ensuite CollisionSystem.step(entities).

**Responsabilités :**

- Centraliser la logique globale (gravité, obstacles, etc.).
- Coordonner entités et physique.

---

## Entity (core.entity)

**Type** : interface / classe abstraite.

**Rôle** : contrat que toutes les entités du jeu doivent respecter.

#### Méthodes typiques :

- update(dtMs) : met à jour sa position, son état interne.
- pos (vecteur position), vel (vitesse), etc.

Cela permet à World d'itérer sur tous les objets sans connaître leurs types.

### Dragon (core.entity)

**Rôle** : entité principale, contrôlée par le joueur.

#### Propriétés :

- pos: Vec2 : position dans le monde.
- light: Float : intensité lumineuse (effet de brume).

#### Méthodes :

- update(dt) : applique les entrées du joueur, met à jour light, détecte collisions.

#### Particularité :

C'est la **seule entité interactive** du jeu (les autres sont statiques ou automatisées).

### Collider & CollisionSystem (core.physics)

**Rôle** : gestion de la physique légère et des collisions.

#### Collider

- Interface pour les objets “solides”.
- Fournit bounds(): AABB (rectangle englobant pour tests rapides).

#### CollisionSystem

- Gère la détection et la réponse des collisions.
- Méthode step(entities) : vérifie les intersections et notifie les entités concernées.

#### Responsabilités :

- Calculer les collisions sans bloquer le reste du jeu.
- Maintenir une complexité faible ( $O(n^2)$  ou grille spatiale simple).

---

## Comment tout interagit

- 1 GameLoop.start() s'exécute toutes les 16 ms.
  - 2 Appelle World.update(dt).
  - 3 World parcourt entities et appelle entity.update(dt) sur chacune.
  - 4 Après les mises à jour, World appelle CollisionSystem.step(entities) pour vérifier les interactions.
  - 5 Les entités réagissent (ex. le Dragon rebondit ou perd de la lumière).
  - 6 Le résultat (DragonState, score, etc.) est envoyé au ViewModel du module :ui.
- 

## Intégration dans le MVVM

- **GameViewModel (dans :ui)** possède une instance de World et de GameLoop.
  - Quand l'utilisateur appuie sur "Start" :
  - viewModel.start() → gameLoop.start()
  - À chaque tick, le ViewModel met à jour un StateFlow<GameUiState> que le GameScreen observe pour se recomposer (dessin du dragon, brume, etc.).
- 

## En résumé

Élément	Rôle	Dépendances
GameLoop	Cadence la mise à jour du jeu	Kotlin Coroutines
World	Regroupe les entités et gère la physique	Entity, CollisionSystem
Entity	Contrat pour tout objet du monde	–
Dragon	Entité contrôlée par le joueur	Entity
Collider	Interface pour objets solides	–
CollisionSystem	Détection de collisions	Collider, Entity

### 3) Diagramme de classes — MVVM côté UI

#### ⌚ Objectif du diagramme

Ce diagramme représente **comment le jeu s'organise côté interface**, c'est-à-dire :

- comment les **entrées du joueur** (touches, gestes) deviennent des **intentions (Intent)**,
  - comment le **ViewModel** gère la logique,
  - et comment l'**UI Compose** observe l'état et se met à jour.
- 

#### ✳️ Vue du diagramme

```
package ui.viewmodel

+ GameViewModel

  + state: StateFlow<GameUiState>

  + event(intent: GameIntent)

  + start(), pause(), resume(), stop()

+ GameUiState

  + phase: GamePhase

  + score: Int

  + dragon: DragonVM

  + fps: Int

+ GameIntent (sealed)

  + Start, Pause, Resume, Quit

  + Touch(x: Float, y: Float)

  + Drag(dx: Float, dy: Float)

package ui.screens
```

+ GameScreen(vm: GameViewModel)

---

## Comment tout s'articule

### GameViewModel

C'est le **chef d'orchestre** de la partie *jeu côté UI*.

Il relie la logique (moteur core) et le rendu (Compose).

#### Responsabilités :

- Démarrer / arrêter la boucle (GameLoop) du core.
- Maintenir un flux d'état réactif (StateFlow<GameUiState>).
- Réagir aux entrées (GameIntent) venant de l'UI.

#### Méthodes clés :

```
fun start()    // Lancer la partie  
fun pause()   // Met en pause la boucle  
fun resume()  // Reprendre  
fun stop()    // Arrêter la boucle  
fun event(intent: GameIntent) // Traiter une action du joueur
```

#### Propriété principale :

val state: StateFlow<GameUiState>

 Le GameScreen observe ce flux pour se recomposer à chaque frame.

---

## GameUiState

Un **snapshot immuable** de tout ce que l'UI doit afficher à un instant donné.

#### Exemple :

```
data class GameUiState(  
    val phase: GamePhase,  // Running, Paused, GameOver...  
    val score: Int,  
    val dragon: DragonVM,  
    val fps: Int
```

)

### Rôle :

- Fournir une version “UI friendly” des données.
  - Garantir que la vue est **une projection** de l’état, jamais une source de vérité.
- 

## GameIntent

C'est la **traduction des actions utilisateur** en événements métiers.

### Défini comme une sealed class :

```
sealed class GameIntent {  
  
    object Start : GameIntent()  
  
    object Pause : GameIntent()  
  
    object Resume : GameIntent()  
  
    object Quit : GameIntent()  
  
    data class Touch(val x: Float, val y: Float) : GameIntent()  
  
    data class Drag(val dx: Float, val dy: Float) : GameIntent()  
  
}
```

### But :

- Centraliser toutes les interactions possibles.
  - Simplifier le test et la lecture (une seule fonction event(intent) les gère toutes).
- 

## GameScreen (UI Compose)

C'est la **vue réactive** : elle affiche les données et envoie des intentions.

@Composable

```
fun GameScreen(vm: GameViewModel) {  
  
    val state by vm.state.collectAsState()  
  
    Canvas(Modifier.fillMaxSize()) {  
  
        drawDragon(state.dragon)  
  
        drawScore(state.score)  
    }  
}
```

```
    }  
}  
}
```

### Responsabilités :

- Observer vm.state.
  - Afficher le rendu (Canvas, sprites, score).
  - Envoyer des GameIntent selon les gestes utilisateur.
- 

### DragonVM

Une version “vue” du Dragon du core :

```
data class DragonVM(  
    val x: Float,  
    val y: Float,  
    val light: Float  
)
```

→ Le ViewModel la construit à chaque tick à partir du Dragon logique (core.entity).

---

### Cycle complet MVVM

**1 Entrée utilisateur** → GameIntent

(ex : Touch(x, y))

**2 GameViewModel.event(intent)** reçoit cette intention.

Il agit sur le World du core (mouvement du dragon, collision, etc.).

**3 World.update(dt)** → met à jour l'état logique.

**4 GameViewModel** reconstruit un GameUiState à partir du modèle.

**5 stateFlow.emit(GameUiState)**

**6 GameScreen** observe ce flux et se **recompose** automatiquement (Compose).

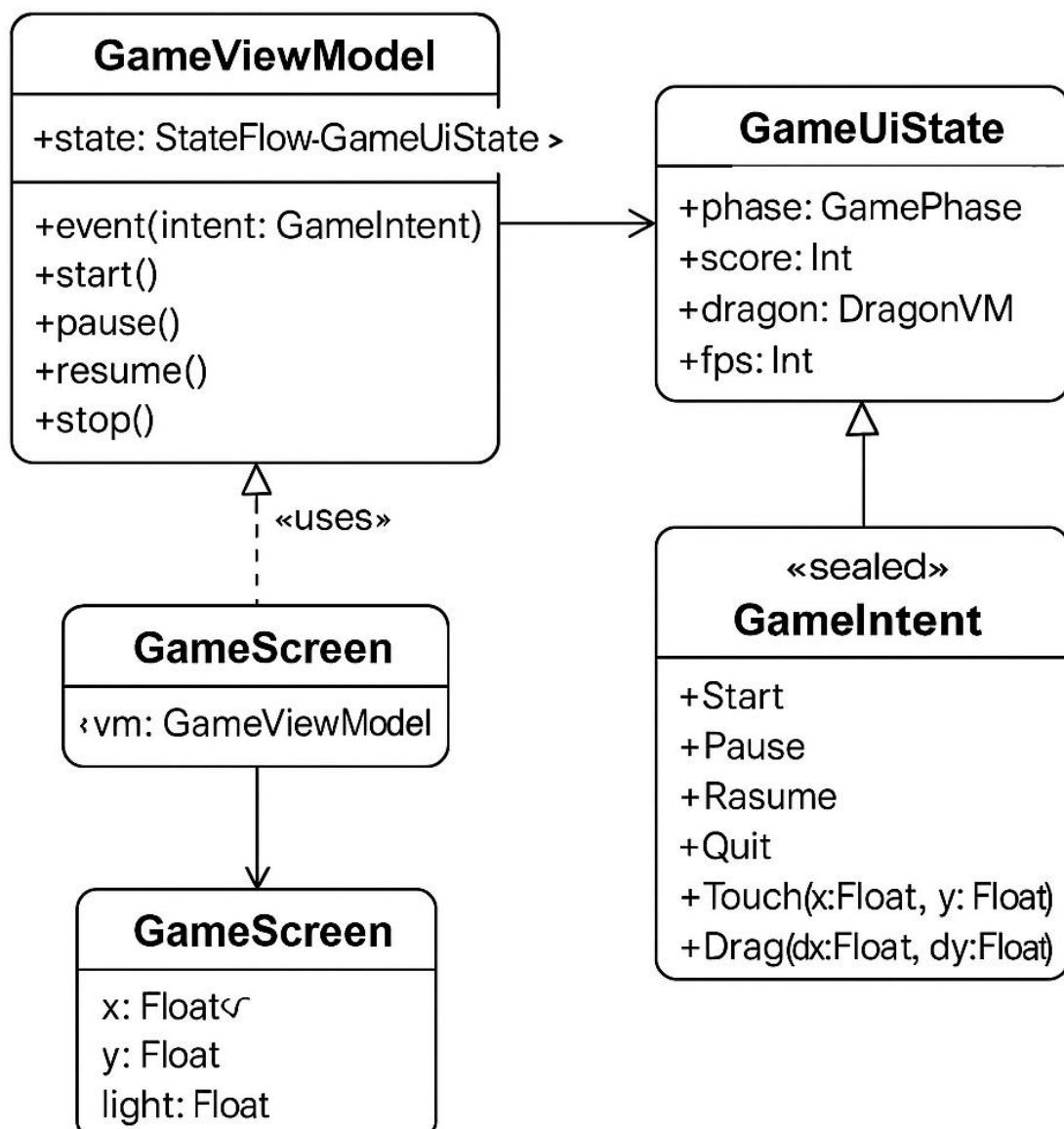
---

### Bénéfices de ce découplage

Couche	Responsabilité	Ne dépend que de...
<b>View (GameScreen)</b>	Afficher l'état, capter les entrées	GameViewModel
<b>ViewModel</b>	Logique, états, orchestration	core, data, audio
<b>Model (core)</b>	Simulation pure, physique, monde	Rien d'Android

## Résultat :

- Tu peux tester le moteur (core) **sans Android**.
  - Tu peux tester les ViewModels avec des **fakes** de World.
  - Le rendu Compose reste simple et réactif.



 En résumé

Élément	Rôle principal
<b>GameViewModel</b>	Orchestration, gestion des intentions et des états
<b>GameUiState</b>	Données prêtes à afficher
<b>GameIntent</b>	Actions utilisateur
<b>GameScreen</b>	Vue réactive basée sur Compose
<b>DragonVM</b>	Vue simplifiée du modèle Dragon
<b>Flux</b>	Intent → ViewModel → State → UI

## 4) Diagramme de séquence — un frame “tick 60 FPS”

### Vue simplifiée du diagramme

GameLoop → GameViewModel : onUpdate(dtMs)

GameViewModel → World : update(dtMs)

World → CollisionSystem : step(entities)

World --> GameViewModel : résultats (positions, collisions, score)

GameViewModel → state : emit(GameUiState)

UI(GameScreen) ← state : recomposition Compose (Canvas draw)

---

### Déroulement détaillé (16 ms en ~6 étapes)

#### 1 GameLoop → GameViewModel

Toutes les 16 millisecondes environ, le GameLoop déclenche son callback :

onUpdate(dtMs: Long)

Il appelle donc :

gameViewModel.onUpdate(dtMs)

→ C'est le signal qui indique “une frame de jeu doit être calculée”.

---

#### 2 GameViewModel → World

Le GameViewModel transmet le pas de temps (dtMs) à la logique du moteur :

world.update(dtMs)

Le World est responsable d'appliquer les mouvements, de faire progresser la physique et de gérer les entités (comme le Dragon).

---

#### 3 World → CollisionSystem

Pendant cette mise à jour, le World délègue la détection de collisions :

collisionSystem.step(entities)

Le CollisionSystem détecte :

- les intersections (ex : Dragon touche un mur, un obstacle ou un bonus),
  - et renvoie des événements (contacts, dégâts, rebonds...).
- 

#### 4 World → GameViewModel

Une fois la simulation terminée :

- le World renvoie les informations mises à jour (positions, collisions détectées, score, etc.) au ViewModel.
- le ViewModel convertit ces données internes (Dragon, World) en **objets de présentation** (DragonVM, GameUiState).

Exemple :

```
state.value = GameUiState(  
    score = world.score,  
    dragon = DragonVM(x = dragon.pos.x, y = dragon.pos.y, light = dragon.light),  
    phase = GamePhase.Running,  
    fps = 60  
)
```

---

#### 5 GameViewModel → state

Le ViewModel émet ce nouvel état dans un flux réactif :

```
_state.emit(newState)  
→ C'est ici que Compose va détecter un changement.
```

---

#### 6 GameScreen ← state

GameScreen, côté UI Compose, observe ce flux :

```
val state by viewModel.state.collectAsState()
```

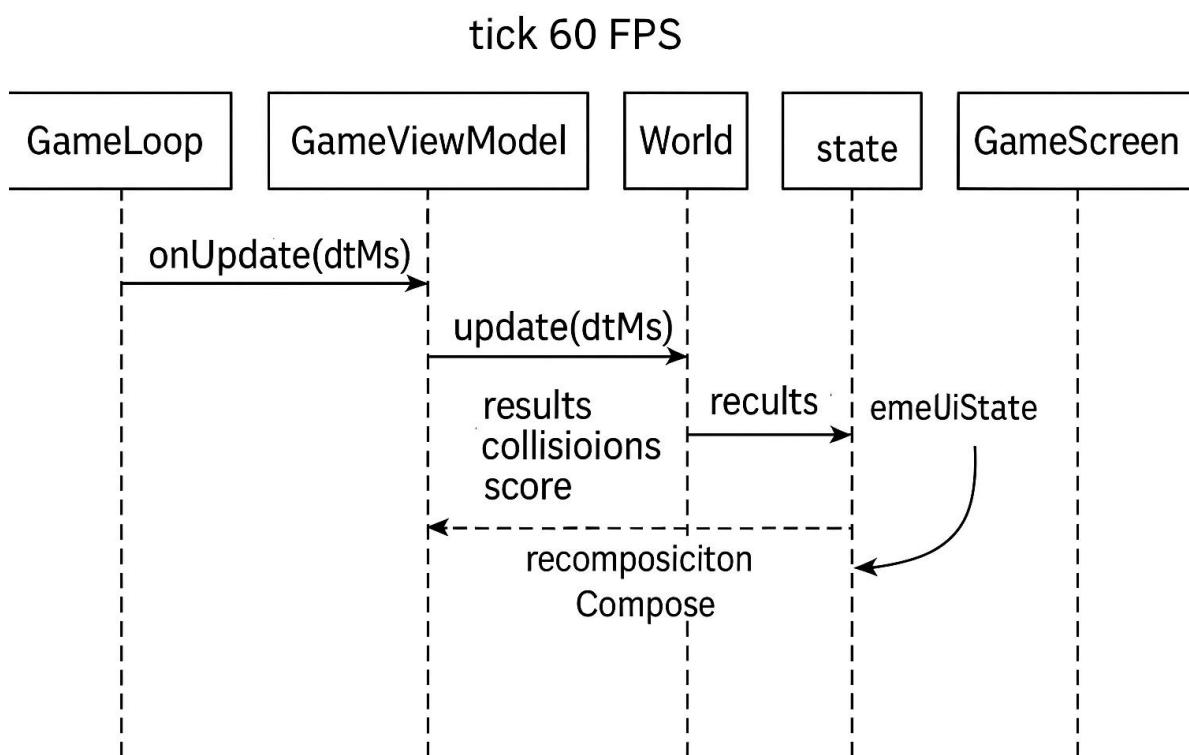
À chaque nouvelle émission :

- la fonction GameScreen() se **recompose automatiquement**,
- et redessine le Canvas (drawDragon, drawScore, etc.) avec les nouvelles valeurs.

C'est ainsi que **le moteur et l'interface restent synchronisés** sans que tu aies besoin de forcer les rafraîchissements manuellement.

### ⌚ En résumé : le “pipeline 60 FPS”

Étape	Acteur	Action	Résultat
1	GameLoop	Appelle onUpdate(dt)	Une frame démarre
2	GameViewModel	Met à jour le monde	Simulation
3	CollisionSystem	Vérifie les collisions	Résultats physiques
4	World	Retourne le nouvel état logique Score, position	
5	GameViewModel	Émet un GameUiState	Données prêtes pour l'UI
6	GameScreen	Observe le flux state	Recomposition visuelle



### 💡 Points clés à retenir

- **Le GameLoop ne connaît jamais l'UI.**  
Il ne fait qu'appeler une fonction générique `onUpdate(dt)`.

- **Le ViewModel fait la passerelle** : il écoute la boucle et émet des états immuables.
- **Le GameScreen est purement réactif** : il n'a pas besoin de timer ni de rafraîchissement manuel.
- **Chaque frame est indépendante** : si une frame lag, la suivante reprend naturellement le bon dtMs.

### 🎮 Illustration imagée

On peut l'imaginer comme un flux continu :

[ GameLoop tick ] → [ World simulation ] → [ New State ] → [ Compose redraw ]

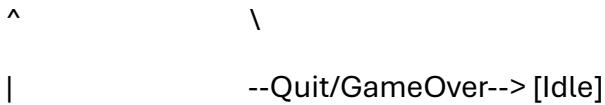
16 ms            2 ms            1 ms            13 ms

⌚ En tout : ≈ 16 ms → 60 images par seconde.

## 5) Diagramme d'états — cycle de jeu

### ⌚ Vue globale du diagramme

[Idle] --Start--> [Running] --Pause--> [Paused] --Resume--> [Running]



### ✳️ Description des états

#### ⚡ Idle

- **Signification :** le jeu est “au repos” — menu principal, aucun GameLoop actif.
- **Acteurs actifs :**
  - HomeScreen ou MenuScreen.
  - Le GameViewModel est créé mais inactif.
- **Actions possibles :**
  - Start → Lancer une nouvelle partie.  
(Initialisation du World, du score, de la boucle, etc.)

#### 🏃 Running

- **Signification :** la partie est en cours.
- **Acteurs actifs :**
  - GameLoop fonctionne (tick toutes les 16 ms).
  - World met à jour les entités, collisions, score.
  - GameViewModel émet les nouveaux états (GameUiState).
- **Actions possibles :**
  - Pause → mettre le jeu en attente.
  - GameOver → fin de la partie (dragon mort, score final).
  - Quit → retour au menu principal.

## Paused

- **Signification :** la partie est suspendue temporairement (menu pause, interruption Android...).
- **Acteurs actifs :**
  - GameLoop stoppé.
  - L'état du World est conservé (positions, score, etc.).
- **Actions possibles :**
  - Resume → reprendre la partie.
  - Quit → retour à Idle sans reprise.
  - (Parfois Restart → nouvelle partie depuis zéro.)

## GameOver

- **Signification :** la partie est terminée, mais le jeu reste affiché pour montrer le score final.
- **Acteurs actifs :**
  - GameLoop arrêté.
  - GameViewModel envoie un état GamePhase.GameOver.
  - GameScreen affiche le résultat et propose des options.
- **Actions possibles :**
  - Quit → retour au menu (Idle).
  - Restart → nouvelle partie (Running).

## Transitions principales

Transition	Déclencheur	Action côté code
Idle → Running	Bouton Start	viewModel.start() lance GameLoop
Running → Paused	Bouton Pause ou perte de focus	viewModel.pause() stoppe la boucle

Transition	Déclencheur	Action côté code
Paused → Running	Bouton <i>Resume</i>	viewModel.resume() relance GameLoop
Running → GameOver	Collision, vie=0	viewModel.stop() et phase = GameOver
GameOver → Idle	Bouton <i>Quit</i>	Retour au menu principal
Paused → Idle	Bouton <i>Quit</i>	Retour au menu principal

### ⚙️ Implémentation dans GameViewModel

Souvent, on encapsule ces états dans un **enum** ou une **sealed class** :

```
enum class GamePhase {
    Idle,
    Running,
    Paused,
    GameOver
}
```

Et dans le GameViewModel, on contrôle les transitions :

```
fun start() {
    if (state.value.phase == GamePhase.Idle) {
        world.reset()
        loop.start()
        updatePhase(GamePhase.Running)
    }
}
```

```
fun pause() {
    loop.stop()
    updatePhase(GamePhase.Paused)
```

```
}
```

```
fun resume() {  
    loop.start()  
    updatePhase(GamePhase.Running)  
}
```

```
fun stop() {  
    loop.stop()  
    updatePhase(GamePhase.GameOver)  
}
```



### Intérêt du diagramme d'états

- Clarifie **les scénarios possibles** du jeu.
- Évite les incohérences (ex. relancer un jeu déjà en pause).
- Sert de base à la gestion du **cyclde de vie Android** (pause/reprise).
- Simplifie les tests (on peut vérifier que chaque transition produit le bon comportement).



### En résumé

État	Description	Actions possibles
Idle	Jeu inactif (menu principal)	Start
Running	Partie en cours	Pause, GameOver, Quit
Paused	Partie suspendue	Resume, Quit
GameOver	Partie terminée	Quit, Restart

## 6) Schéma de navigation (Compose)

Home —Play—> Game —Pause—> PauseDialog

\ \-> Settings

\—> Credits

Routes minimales : home, game, settings (pause peut être un dialog).