### Problématique

A ce stade de la conception, nous souhaitons faire en sorte que la gestion des différents états du jeu soit bien structurée au niveau de l’implémentation, afin d’augmenter la lisibilité et la maintenabilité du code source.

Cette stratégie, qui doit être adoptée pour toutes les composantes du système, augmentera l’efficacité et la facilité du travail des développeurs.

#### Fonctionnalité à implémenter

Tout jeu vidéo s’exécute en défilant devant le gamer des *«écrans »* clés du jeu, chacun représentant un état particulier, à travers lequel le jeu permet au gamer d’accéder à certains services.

Par exemple, tout jeu vidéo possède un état d’introduction (avec souvent une image caractéristique du jeu, ou même, dans les jeux les plus aboutis, une cinématique), puis un état de menu, qui donne à son tour la possibilité de démarrer la partie, et accessoirement aussi à un état d’options … etc. Lorsque la partie est terminée, le jeu affiche un état de *game over*, qui donne ensuite souvent la possibilité de retour au menu initial. Plus encore : un jeu peut même être simultanément dans plus d’un état à la fois, ainsi lorsque le joueur accède au menu des options pendant le déroulement de la partie.

Nous désirons mettre en place cette fonctionnalité d’états multiples pour notre jeu *Game Foot+ 2D*, en nous basant sur une conception robuste, qui permette la lisibilité et la réutilisabilité de notre code source.

### La mauvaise méthode

Une méthode commune, immédiate mais maladroite, consisterait à gérer les différents états à travers une grande boucle, qui met en place un aiguillage[[1]](#footnote-1) sur le type de l’état en cours.

Ainsi le programme démarre sur l’état d’introduction, puis boucle sur les entrées utilisateurs jusqu’à ce que le gamer réalise la manœuvre demandée (généralement, *« Appuyer sur une touche »*), puis l’écran de l’état de menu est affichée, jusqu’à ce qu’une sélection soit faite (par exemple, *« Commencer partie »*). Ensuite la partie du jeu commence, et boucle jusqu’à la fin de partie.

La boucle principale du jeu en devient très vite illisible et vulnérable aux erreurs de codage. En conséquence, le noyau du jeu, dans lequel cette boucle est en principe implémentée, et qui doit gérer entre autres les entrées utilisateurs, risque d’être une composante ambigüe de notre système, ce qui provoquera l’instabilité de l’application.

### La bonne méthode

#### Définition formelle d’un état

Un état est une composante de l’application, capable de prendre soin d’elle-même, ce qui déchargera le noyau et les moteurs de la gestion tous de tous les évènements destinés à l’état en cours.

Ainsi, un état configurera sa propre scène (puis l’envoie au noyau, qui se charge de l’aiguiller au moteur graphique, qui la dessine).

#### Automate de gestion d’états du jeu

Au sein de notre programme de jeu, les transitions entre les différents états peuvent être modélisées par un automate à états finis déterministe.

N.B. : comme spécifié dans le GDD, dans un premier temps, l’application n’implémentera que l’état *PlayingGame.* L’implémentation des autres états est optionnelle.

#### Utilisation du patron de conception *Etat*

Sur la base de l’analyse précédente, et en nous référant aux travaux connus sur les patrons de conception en informatique, nous avons choisi d’implémenter, pour les états multiples du jeu, le patron de conception *« Etat »* (*pattern design « State »)*, afin d’en optimiser la gestion, et ainsi augmenter la robustesse de notre système.

#### Conception

#### Considérations techniques

* La classe *GameState* étant abstraite, on ne peut pas la déclarer *Singleton* (du fait que notre template *Singleton* instancie les classes qui l’implémentent). Ainsi tout objet héritant de GameState (donc un état concret) sera chacun déclaré individuellement *Singleton*.

1. Qui peut être soit une séquence d’instructions if, soit une instruction switch. [↑](#footnote-ref-1)