



Trajectory detection for game scoring

Reconnaissance de trajectoire pour le jeu vidéo

PFA - Rapport final

Louis Cesaro, Timothée Corsini, Jean Farines,
Arno Galvez, Adrien Lavancier, Anatole Martin, Félix Pierre

Clients : Martial Bossard, Pierre-Marie Plans

Responsable Pédagogique : David Renault

8 avril 2019, ENSEIRB-MATMECA

Table des matières

Introduction	4
I Organisation du projet	4
1 Technologies utilisées	4
1.1 Moteur de jeu	4
1.2 Assets	4
1.3 Contrôle de sources	5
2 Répartition des tâches	5
2.1 Support de communication	5
2.2 Sessions de travail et réunions	5
II Environnement de jeu	5
3 Physique de l'avion	6
3.1 Système de forces	6
3.2 Première approche : physique du point	6
3.3 Limites du premier modèle	7
3.4 Passage à la physique du solide	7
4 Interface de jeu	8
4.1 Menu	8
4.2 Niveau de jeu	8
4.3 Retours pour l'utilisateur	9
4.4 Gestion de la caméra	10
III Détection des figures	10
5 Communication entre le jeu et la détection	10
5.1 Structure de données	10
5.2 Communication par FigureManager	11
6 Méthode naïve : Automates	12
6.1 Idée de base	12
6.2 Première implémentation	13
6.3 Les machines à états finis	13
6.3.1 Algorithme principal	13
6.3.2 Premières versions	15
6.3.3 Modifications des coordonnées et nouvelles implémentations	15
6.4 Interface pour faciliter la création de nouvelles figures	16
7 Reconnaissance de tracé : algorithmes \$1 / \$P	17
7.1 Idée de base	17
7.2 Description de l'algorithme	17
7.3 Courbes de référence	18
7.4 Implémentation du code	19
7.4.1 De Coordinate aux courbes	19
7.4.2 Des courbes aux figures	19
7.4.3 Passage à \$P pour une meilleure détection	19

8 Comparaison des algorithmes	20
8.1 Qualité de la détection	20
8.2 Complexité des algorithmes	20
8.3 Facilité d'utilisation	21
IV Travail réalisé et améliorations possibles	21
9 Chronologie du projet	21
10 Structure du code	23
11 Améliorations envisageables	24
11.1 Niveau de jeu	24
11.2 Physique de l'avion	24
11.3 Détection par automates	24
11.4 Reconnaissance de tracé (\$1/\$P)	24
Conclusion	25
Annexes	26
A Figures de références pour \$P	26
B Document de spécification des besoins	29
C Manuel d'utilisation et de maintenance	40

Introduction

Dans le cadre du PFA, nous avons été mis en lien avec Asobo Studio, une société bordelaise créatrice de jeux vidéos. Ainsi, le projet suivant nous a été confié : *Trajectory detection for game scoring*. Il consiste en un prototype de jeu de course de style Micromachines™ avec obstacles capable de reconnaître des figures de voltige aérienne réalisées par le joueur. Les jeux Micromachines™ sont des jeux de course mettant en scène des voitures miniatures dans des circuits à échelle humaine (salon, table de billard, cuisine...). Le but est donc d'avoir le même type de parcours avec des avions miniatures. Néanmoins, le cœur du projet est l'algorithme de reconnaissance des figures, qui pourra être réutilisé par l'entreprise dans le cadre de futurs projets. Le document qui suit décrit notre avancement et notre organisation au cours des 6 mois de réalisation.



FIGURE 1 – Course de voitures
Micromachines™ issue du jeu vidéo *Micro Machines World Series*



FIGURE 2 – Avion effectuant une figure lors d'un spectacle de voltige aérienne

Première partie Organisation du projet

1 Technologies utilisées

Dans le cadre de ce projet, nous étions libre d'utiliser les technologies et les moteurs de jeu que nous voulions afin de développer notre jeu avec la détection de trajectoire.

1.1 Moteur de jeu

Nous nous sommes posés la question de savoir quel moteur nous utiliserions entre Unity, que nous connaissons tous, et Unreal Engine que nous connaissons moins. Malgré le fait que Unreal permet de coder en C++ et d'avoir un contrôle sur la mémoire, ce que ne permet pas Unity qui n'autorise que la programmation en C# ou JavaScript, nous avons choisi le moteur de jeu Unity pour ce projet. Nous avons démarré le projet sur la version 2018.2.15f, qui était la dernière version stable à ce moment là. Nous avons travaillé sur cette version durant la quasi-totalité du projet.

Par la suite, nous avons décidé de changer et de migrer vers la version 2018.3.8f afin d'avoir une meilleure gestion de nos *Prefab* ainsi que pour un soucis de compatibilité de librairie C# (notamment la librairie System.Numerics). En effet, nous avons utilisé les structures de données de la librairie System.Numerics pour communiquer entre les algorithmes de détection et Unity.

1.2 Assets

Dans le but de nous assurer de la liberté des droits des objets 3D que nous utilisions au cours de notre projet, nous avons décidé de les faire par nous-mêmes. Pour ce faire, nous avons utilisé le logiciel Blender qui nous a permis de réaliser rapidement le modèle de l'avion ainsi que de l'indicateur de rotation dans les anneaux du jeu. L'un des avantages majeurs de ce logiciel est que

le format de fichier des objets 3D créés dans Blender sont directement utilisable dans Unity et ne nécessitent pas de changement de format.

Cependant, par un souci de gain de temps, le modèle des anneaux a été trouvé sur Internet afin de pouvoir passer plus de temps sur la programmation de la physique et de la détection plutôt que sur la création des objets 3D.

1.3 Contrôle de sources

Afin de pouvoir travailler tous ensemble sur le projet sans conflits, nous avons créé un dépôt Git sur la plate-forme d'hébergement GitHub. Les dépôts SVN supportant mal le développement sur différentes branches, l'utilisation de Git nous est parue évidente. Nous n'avons pas créé le dépôt sur la Forge de l'école du fait de l'importance de la taille des fichiers utilisés pour les modèles 3D notamment (15 Mo pour l'anneau). En prévention de la taille que pouvait prendre le projet sur la Forge, nous avons préféré le créer sur GitHub. Aujourd'hui, en comptant la taille prise par les différences entre les branches du projet, le projet prend entre 500 Mo et 1 Go de mémoire sur le dépôt.

Cette expérience avec Git a été très enrichissante. Tout d'abord, elle nous a permis de travailler avec un outil de gestion de projet utilisé en entreprise, mais également, cela nous a imposé une rigueur de travail afin d'éviter les conflits lors de notre travail, qui étaient principalement dûs aux fichiers .META, créés par Unity.

2 Répartition des tâches

2.1 Support de communication

Afin de s'organiser et se répartir les tâches efficacement au cours du projet, nous avons tenté d'utiliser la plate-forme Trello. Nous rentrions la découpe des tâches à réaliser sur les premières semaines de recherche et de codage, ainsi que la ou les personnes en charge de la tâche. Nous avons finalement très peu utilisé ce Trello, préférant la communication via une application de messagerie. La principale difficulté que nous avons eu dans l'utilisation de ce Trello a été la découpe fine des tâches à réaliser sur le sprint. Ne prenant pas le temps de définir clairement les tâches à faire, nous avons peu à peu arrêté d'utiliser le Trello pour la répartition des tâches. Nous avons cependant continué à l'utiliser pour archiver les comptes rendus des rendez-vous avec notre client comme avec notre encadrant.

2.2 Sessions de travail et réunions

Les réunions avec le professeur responsable, M. David RENAULT, et le consultant technique d'Asobo Studio, M. Pierre-Marie PLANS, étaient fréquentes : environ une fois toutes les 1 ou 2 semaines. Cela nous a permis de mieux synchroniser les tâches de groupe et d'avoir un retour rapide sur notre implémentation. Plus rarement étaient organisés des réunion avec le client, M. Martial BOSSARD, où nous montrions l'avancement de notre projet et ce que nous envisagions de réaliser jusqu'à la prochaine réunion. C'est lors de ces rencontres que des objectifs clairs étaient fixés, nous permettant de nous concentrer sur un aspect en particulier.

Parallèlement à cela, nous travaillions en groupe plusieurs fois par semaine, notamment lors des séances PFA aménagées dans notre emploi du temps.

Deuxième partie

Environnement de jeu

3 Physique de l'avion

La première étape de réalisation du projet était de disposer d'un avion capable de voler dans l'espace de jeu et ainsi réaliser des figures pour gagner des points. Cela implique donc que nous définissons un modèle de physique à appliquer à l'avion pour le faire se déplacer selon les entrées de l'utilisateur.

3.1 Système de forces

Le modèle physique que nous avons développé se base sur le système simplifié à 4 forces "Poussée Portance Traînée Poids". Comme son nom l'indique, on identifie dans ce modèle 4 forces uniques (illustrées en Figure 3) ayant une influence sur l'avion :

- Poussée (Thrust) : Force appliquée par le moteur faisant avancer l'avion.
- Portante (Lift) : Force subie par l'avion en mouvement dans l'air, appliquée perpendiculairement à la direction du mouvement.
- Traînée (Drag) : Force de frottement augmentant avec la vitesse de l'avion et opposée au mouvement.
- Poids : Force subie par l'avion du fait de sa masse et la gravité.

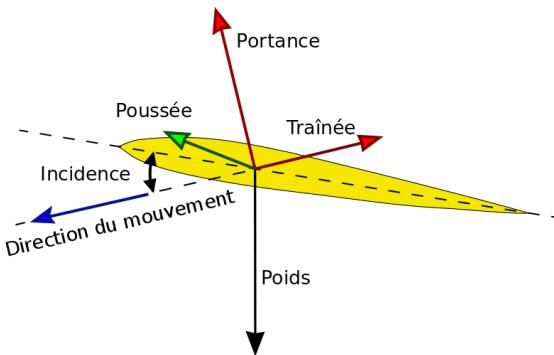


FIGURE 3 – Forces auquel l'avion est soumis

En se restreignant à ce système, nous avons fait le choix d'ignorer un ensemble de forces annexes qui nous semblait peu pertinent d'ajouter : par exemple, le vent peut avoir un impact sur la trajectoire de l'avion, mais celui-ci peut être ignoré sans altérer le réalisme du modèle physique.

Une fois les forces à considérer définies, il faut établir comment les calculer. On dispose ainsi des formules suivantes :

$$Lift = \frac{1}{2}C_L S \rho v^2 \quad Drag = \frac{1}{2}C_D A \rho v^2 \quad Poids = mg$$

où C_L est le coefficient de portance, C_D le coefficient de frottement, S la surface des ailes, A l'aire de la surface de frottements, ρ la masse volumique de l'air, m la masse de l'avion et v sa vitesse.

Pour le choix des valeurs de telles constantes, nous les avons fixées de façon à retranscrire une physique suffisamment réaliste, tout en permettant de faire des figures et en restant maniable.

3.2 Première approche : physique du point

Un ensemble de forces ne suffit pas à faire voler un avion. Il faut ensuite déterminer comment appliquer les forces. En première approche, nous avons choisi d'utiliser un modèle de physique du point : l'avion est représenté comme un point unique sur lequel toutes les forces sont appliquées. Le moteur de physique d'Unity permet facilement d'appliquer des vecteurs force à un objet, nous avons donc pu rédiger des scripts qui calculent à chaque *frame* les nouvelles forces adaptées à l'avion.

En choisissant des valeurs adaptées pour les constantes, nous avons alors obtenu un premier modèle fonctionnel : en appuyant sur une touche adéquate, l'avion parvient à s'élever dans le ciel. Cependant, il était en l'état impossible de tourner, toutes les forces s'appliquant au même point.

Pour permettre à l'avion de s'orienter, notre idée fut alors d'imposer des rotations d'angle selon les 3 axes de l'avion (nommés *roll*, *pitch* et *yaw*, illustrés en Figure 4) en fonction des commandes de l'utilisateur.

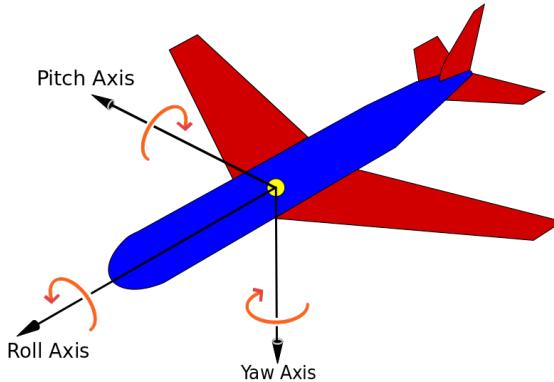


FIGURE 4 – Axes de rotation de l'avion

Avec cela, nous avons alors obtenu un premier modèle physique fonctionnel, dans lequel l'avion pouvait se déplacer dans l'aire de jeu.

3.3 Limites du premier modèle

Rapidement, il est apparu que notre modèle physique n'était pas satisfaisant sur bien des points.

Tout d'abord, il était extrêmement compliqué de réaliser une figure correcte. De plus, les sensations en jeu n'étaient pas du tout réalistes. Ces deux problèmes ont une source commune : l'utilisation de rotations fixes imposées par l'utilisateur. En effet, ces rotations sont effectuées dans le jeu indépendamment du moteur physique : ainsi, les mouvements de l'avion apparaissent secs. De même, les déplacements de ce dernier semblaient en décalage par rapport à l'orientation de l'avion, comme si l'aéronef "dérapait" dans les airs.

Il est donc apparu qu'il nous fallait une autre méthode pour traduire les rotations de l'avion.

3.4 Passage à la physique du solide

L'utilisation de rotations fixes venait du constat que, dans un modèle de physique du point, tout s'applique au même endroit. Or, si on veut des rotations générées par la physique, il faut imposer des moments de force.

Nous avons donc choisi de changer de modèle, en utilisant dorénavant la physique du solide. Dans ce nouveau modèle, les forces ne sont plus toutes appliquées au centre de gravité, mais plutôt à des points d'application spécifiques à chaque force. Ainsi, la *Poussée* est appliquée au niveau du moteur et des forces de *Portance* sont appliquées sur chaque aile. Seuls le poids et la traînée sont toujours appliqués au centre de gravité. Ces points d'application sont illustrées sur la figure 5.

A partir de cela, on peut alors faire apparaître des moments de forces pour faire tourner l'avion. Ainsi, nous avons fait en sorte que, lorsque le joueur veut faire tourner l'avion selon son axe *roll*, on applique des forces de *Portance* opposées sur les ailes, ce qui entraîne une rotation. Pour la variation du *pitch*, nous avons utilisé le fait que les ailes soient excentrées avec le centre de gravité pour faire apparaître un moment entre le *Poids* et la *Portance*. Enfin, les rotations selon *yaw* sont faites par application d'une légère force de *Portance* au niveau de la queue de l'avion.

Avec ces modifications, l'avion est devenu plus réactif et plus facile à contrôler. La sensation de "dérapage" a disparu, et il est possible de réaliser des figures de bonne qualité avec une bonne prise en main.

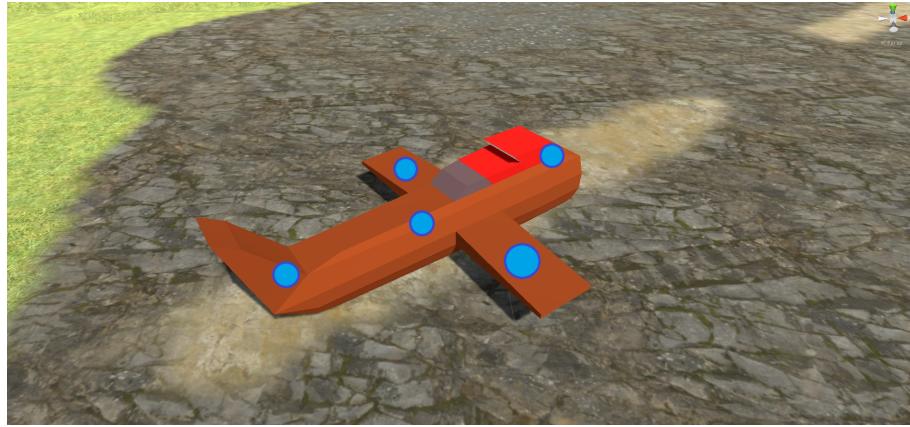


FIGURE 5 – Points d’application des forces

4 Interface de jeu

4.1 Menu

Afin de permettre à l’utilisateur de modifier les paramètres de l’avion à sa guise, nous avons implémenté un menu principal de jeu qui permet de lancer le jeu, modifier quelques paramètres de base (volume, résolution), ainsi que les différents coefficients des forces de l’avion évoqués précédemment. De plus, appuyer sur la touche P permet de changer de méthode de reconnaissance de figures (Automate ou \\$). Afin de recommencer une partie rapidement, appuyer sur la touche R entraîne la réinitialisation de la scène.



FIGURE 6 – Menu d’options

4.2 Niveau de jeu

Il a vite été nécessaire de créer un terrain où évoluer avec notre avion afin de tester les différentes fonctionnalités de notre programme. Ainsi, nous avons modélisé un paysage basique avec des reliefs afin d’avoir une réelle impression d’altitude et de vitesse. Nous voulions aussi ajouter un élément qui interagirait avec le joueur. Nous avons donc placé des anneaux flottants dans la scène pour que le joueur essaie de passer dedans. Plus tard, nous les avons améliorés en incitant l’utilisateur à y rentrer avec une certaine inclinaison (qui peut-être paramétrée aléatoirement au début de chaque partie), le familiarisant ainsi avec les commandes du jeu, comme on peut le voir dans la figure 7.

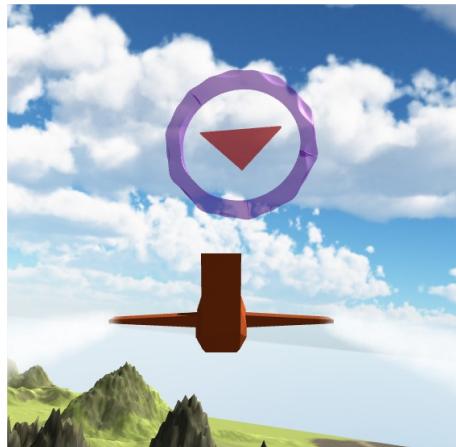


FIGURE 7 – Anneau dans lequel le joueur doit passer avec une inclinaison vers la droite de 180°

4.3 Retours pour l'utilisateur

De nombreux affichages permettent un retour utilisateur optimal. Un indicateur de vitesse et d'altitude permet au joueur d'avoir une réelle idée de son évolution et de sa position. De plus, des indicateurs d'orientation et d'inclinaison lui permet de mieux se repérer dans l'espace (figure 8).

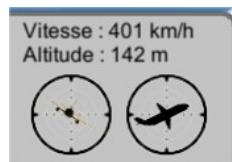


FIGURE 8 – Indicateur de vitesse, d'altitude et d'orientation

Il est important que le joueur puisse savoir facilement quand il a réussi à réaliser une figure, et qu'il ait un sentiment de réussite suite à cela. C'est pourquoi nous avons implémenté un système de score : quand le joueur réussit une figure, une inscription s'affiche à l'écran (figure 10) et le score augmente en fonction de la difficulté de la figure. Comme le montre la figure 9, une traînée de fumée derrière l'avion permet de se rendre compte de la trajectoire empruntée en plus d'ajouter un aspect esthétique.

Enfin, une musique d'ambiance permet d'améliorer l'immersion du jeu, et un bruitage sonore informe le joueur qu'il est passé dans un anneau avec la bonne orientation.



FIGURE 9 – Fumée représentant la trajectoire



FIGURE 10 – Affichage de la figure réalisée

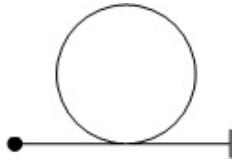
4.4 Gestion de la caméra

Le premier système de caméra que nous utilisions était très simple : la caméra suivait exactement l'avion. Cela rendait l'expérience de jeu très peu plaisante : il n'y avait aucune impression de vitesse, et la perception de la physique de l'avion était grandement diminuée. En effet, le joueur avait l'impression que l'avion était immobile et que le paysage défilait autour de lui. Un simple effet de latence appliqué à la rotation de la caméra a permis de corriger cela : l'utilisateur peut maintenant voir la direction que prend l'avion lorsqu'il tourne. Nous avons ajouté un peu plus tard un léger effet de recul sur la caméra lié à la vitesse de l'avion, permettant de mieux se rendre compte de l'accélération. Ces ajouts ont permis une utilisation plus agréable des commandes de l'avion.

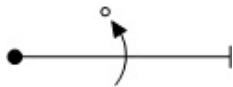
Troisième partie Détection des figures

Les figures aéronautiques sont un sujet à part entière dans le domaine de la reconnaissance de figures. En effet, contrairement à une moto ou à un skater, un avion peut se déplacer librement dans un espace en 3 dimensions, et pivoter librement selon les 3 axes de rotation.

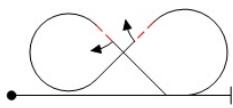
Nous nous sommes fixés pour but de reconnaître les 3 figures suivantes :



Loop : L'avion s'oriente vers le haut jusqu'à revenir dans sa position initiale. Il doit garder ses ailes horizontales lors de la manœuvre et terminer à la même altitude que celle à laquelle il commence.



Aileron Roll : L'avion vole en ligne droite et effectue une rotation sur lui-même, en s'orientant vers la gauche ou vers la droite.



Cuban Eight : L'avion effectue 5/8 d'un *Loop* vers le haut puis fait la même chose vers le bas, jusqu'à revenir à sa position initiale. Les deux cercles doivent avoir le même rayon et la même altitude. L'avion doit aussi avoir la même altitude au début et à la fin de la manœuvre.

Pour cela, plusieurs méthodes se sont présentées à nous. Nous avons décidé d'en retenir deux, afin de pouvoir les comparer côté à côté. La première est une méthode dite naïve : la trajectoire de l'avion active successivement les états d'un automate fini qui valide la figure s'il arrive dans un état terminal. La deuxième méthode est plus complexe, et est basé sur un algorithme de reconnaissance de tracé, le tracé étant ici la trajectoire de l'avion projeté sur un plan 2D.

5 Communication entre le jeu et la détection

5.1 Structure de données

Une des premières choses qu'il a fallu faire lorsque nous nous sommes attaqués à la détection des figures fut de définir ce qu'était une figure et comment la représenter. Nous avons décidé, dans un premier temps, de considérer qu'une figure était un ensemble de points avec des rotations particulières. Ainsi, il suffit que comparer des ensembles de points (celui du jeu avec celui de référence) pour détecter une figure sur une fenêtre de temps donnée.

Ainsi, nous passions le `transform` de l'avion, à savoir ses vecteurs de position et de rotation dans le monde de jeu, aux algorithmes de détection. Rapidement, nous avons décidé de changer cela en créant deux classes : *Coordinate* et *Figure*.

La classe *Coordinate* correspond à l'ensemble des coordonnées d'un point de vol de l'avion. Elle est utilisée pour transmettre les coordonnées depuis Unity aux algorithmes de détection. Elle

possède les attributs suivants :

```

1   float posX; //position suivant l'axe X du monde
2   float posY; //position suivant l'axe Y du monde
3   float posZ; //position suivant l'axe Z du monde
4   float rotX; //rotation suivant la valeur X du quaternion
5   float rotY; //rotation suivant la valeur Y du quaternion
6   float rotZ; //rotation suivant la valeur Z du quaternion
7   float rotW; //rotation suivant la valeur W du quaternion
8   float time; //temps d'enregistrement du point

```

La classe *Figure* correspond au retour des algorithmes de détection. Cette classe indique le pourcentage de détection d'une figure parmi l'ensemble des figures détectables. Elle possède les attributs suivants :

```

1   enum id; //le type de figure reconnu
2   float quality; //le pourcentage de reconnaissance de cette figure

```

Rapidement, nous nous sommes rendus compte que l'utilisation des valeurs du quaternion pour la rotation n'était pas très claire et pratique. Nous avons donc changé et sommes passés sur les angles d'Euler pour faciliter la manipulation des données ainsi que le débogage de notre code. Ce changement modifia les attributs concernant la rotation de la classe *Coordinate* (l'attribut *rotW* a été supprimé). Néanmoins, cela entraîna de nouveaux problèmes.

En effet, les angles d'Euler varient de -180° à 180° pour les angles α et γ et de -90° à 90° pour l'angle β (c.f. figure 11), au lieu de 0° à 360° comme nous le pensions. Cela induit un problème de continuité des valeurs des rotations suivant les différents axes observés.

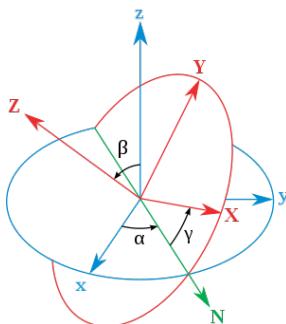


FIGURE 11 – Définition géométrique des angles d'Euler

Finalement, pour tenter de résoudre ce problème d'angle, nous nous sommes orientés vers les produits scalaires. L'idée est de calculer les produits scalaires entre les vecteurs de l'avion et les vecteurs du monde afin d'avoir une idée de la direction prise par l'avion à un instant t. Les données de l'avion sont nécessaires aux calculs des produits scalaires, nous avons donc décidé de ne plus utiliser la classe *Coordinate* mais de passer l'instance de la classe *Plane* aux algorithmes de détection. Ce changement impacte notamment la détection avec les automates qui se servait des angles pour passer d'un état à un autre. Maintenant les conditions de transition dépendent des produits scalaires.

5.2 Communication par FigureManager

L'objectif de ce projet est d'obtenir des algorithmes de détection de figures 3D qui soient indépendants du moteur de jeu qui les utilise. Sachant cela, nous avons implémenté la classe *FigureManager* qui sert de lien entre Unity et les algorithmes de détection. Reliant un certain nombre d'éléments de notre jeu, *FigureManager* gère l'aspect UI du jeu et récupère toutes les données à

transmettre pour la détection.

Pour pouvoir remplir son rôle, la classe *FigureManager* possède les attributs suivants :

```
1  public enum Detector; //definition des methodes de detection
2  public static string []; DetectorName //tableau des noms des
   méthodes de detection
3  public GameObject plane; //objet d'Unity correspondant a l'avion
4  public Text textScore; //UI affichant le score du joueur
5  public Text textFigure; //UI affichant les figures reconnues
6  public Text textAlgo; //UI affichant l'algorithme utilise
7  private Plane _plane; //instance contenant les donnees pour la
   detection
8  private int _score; //score du joueur
9  private float _timeToDisplay; //temps d'affichage de textFigure
10 private string [] _figureName; //tableau des nom des figures
11 private int [] _figurePoint; //tableau des points de chaque figure
12 private IFigureDetection; _figureDetection //Interface reliant
   algorithmes et FigureManager
```

Ces attributs ont été plusieurs fois modifiés au cours du projet. Notamment, `_figureName` et `_figurePoint` n'étaient pas envisagés au début du projet mais ont été rajoutés dans le but de rendre l'affichage et le calcul du score plus générique. En effet, grâce à l'ajout de ces attributs, lors de l'ajout de figures à détecter, il suffit simplement d'ajouter les figures et leur score associé dans les tableaux correspondants. Les fonctions d'affichage et de calcul du score n'ont pas besoin d'être modifiées contrairement aux premières implémentations qui correspondaient à des "switch/case" et qui impliquaient le rajout de conditions par figure.

Par ailleurs, la question s'est posée en fin de projet de supprimer l'attribut `plane` du fait que l'ensemble des données nécessaires à la détection des figures (position, orientation par rapport au monde, etc...) se trouvait dans l'attribut `_plane`. Finalement, nous avons conservé `plane` du fait qu'il permet l'instanciation de l'attribut `_plane` au lancement du jeu. Néanmoins, il est possible de changer cela en mettant un tag sur le préfab de l'avion afin de l'identifier depuis le code, de le récupérer dans le constructeur de la classe *Plane* afin d'initialiser les attributs de la classe *Plane* et de ne plus avoir à passer le *GameObject* en paramètre du constructeur de *Plane*.

Dans le but de s'assurer du bon fonctionnement de la communication entre *FigureManager* et les algorithmes de détection, nous avons mis en place une *FigureFaussaire* qui n'utilise pas les structures de données ni les classes de Unity. En effet, la détection étant implementée en C# standard, la classe *FigureFaussaire* ne fait pas exception afin de modéliser parfaitement la communication des algorithmes et de *FigureManager*. Le but de ce faussaire est d'utiliser les fonctions de la classe *IFigureDetection* afin de s'assurer que les retours sont corrects et qu'il y ait bien un retour UI. À chaque frame, on appelle les fonctions de reconnaissance qui, respectivement, renvoient toutes les 3, 7 et 29 secondes le signal qu'un Roll, un Loop et un CubanEight ont été reconnus. Par ailleurs, nous nous assurons que le score est bien augmenté du montant de points correspondant à chaque figure grâce à l'affichage via `textScore`. Ce faussaire a été utilisé le temps que les algorithmes de détection, détaillés dans la partie suivante, soient implémentés.

6 Méthode naïve : Automates

6.1 Idée de base

Une première idée d'algorithme de reconnaissance est celle de vérificateurs : on atteste que l'avion passe par certaines "étapes" avant de faire une figure.

Par exemple, dans le cas d'un looping (voir figure 12), l'avion sera tout d'abord en position horizontale (1) avant de se diriger vers le haut (2), être renversé (3), tourné vers le bas (4) et enfin revenir en position horizontale (5).

Avec ce modèle, on peut imaginer des automates qui permettent d'attester de ces étapes :

- quand l'avion respecte les conditions de l'état suivant, on avance dans notre algorithme
- quand l'avion respecte les conditions de l'état actuel, rien ne se passe
- quand l'avion s'éloigne trop des propriétés de l'état actuel et de l'état suivant, l'automate revient à l'état initial

Enfin, quand une machine atteint son état final, on considère la figure associée comme réalisée. La figure 13 illustre un automate qui représente le comportement attendu.

6.2 Première implémentation

Afin de réaliser un prototype de cette méthode, nous avons utilisé le module Unity *Playmaker*. Ce module permet de créer des machines à états finis au sein même d'Unity. Nous avons donc réalisé l'automate du looping, en regardant les valeurs de l'angle de l'avion, comme le montre la figure 14.

Ce modèle permettait de reconnaître un looping de manière systématique, ce qui nous a confirmé qu'un automate serait une bonne manière de reconnaître des figures. C'est notamment lors de la réalisation d'un automate pour reconnaître le Aileron Roll avec *Playmaker* que nous avons réalisé qu'utiliser les angles d'Euler ne serait pas aussi simple que nous le pensions.

Toutefois, *Playmaker* est lié à Unity, ce qui rentre en conflit avec notre désir d'avoir une méthode de reconnaissance indépendante au moteur de jeu. C'est pourquoi, on a décidé d'utiliser un autre système d'automates avec un algorithme principal qui communique avec *FigureManager*.

6.3 Les machines à états finis

Nous avons donc cherché à mettre en place un algorithme en C#, indépendant de Unity, fonctionnant avec *FigureManager*, et qui soit simple à utiliser.

6.3.1 Algorithme principal

On commence par mettre en place l'algorithme de reconnaissance principal à partir de l'interface perçue par *FigureManager*, *IFigureDetection* dont l'implémentation est *AutomataDetector*.

AutomataDetector a pour objectif de calculer les états des automates et de renvoyer éventuellement les figures détectées, ici, une figure est soit détectée, soit réfutée, le pourcentage de reconnaissance est donc forcément 0 ou 100.

AutomataDetector dispose d'une liste d'automates (un par figure) sur lesquels il calcule un nouvel état à partir du point donné en entrée. Afin de synthétiser le code et de permettre l'ajout de nouvelles figures, nous avons créé une interface *IFigureAutomata*, la figure 15 permet de montrer les liens entre les différentes classes.

Interface des automates

On cherche à ce qu'un automate puisse :

- calculer un état étant donné une nouvelle position
- indiquer s'il est situé ou arrive sur un état final
- indiquer quelle figure il représente

Il peut aussi être intéressant de savoir si un automate retourne à son état initial dans le cas d'un échec.

Notre interface décrit les méthodes suivantes :

- **resetStates()** qui replace l'automate à son état initial, elle est appelée par l'automate lui-même en cas d'échec de la détection ou bien par *AutomataDetector* quand la figure arrive sur son état final.
- **getFigureId()** permet de récupérer l'identifiant de la figure que l'automate est censé détecter.
- **isValid()** permet de savoir si l'automate est sur son état final ou pas.
- **calculateState()** détermine le nouvel état de l'automate quand on lui passe une position, c'est cette fonction qui va convertir les positions en transitions et effectuer le changement

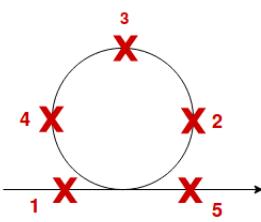


FIGURE 12 – Étapes basiques d'un looping

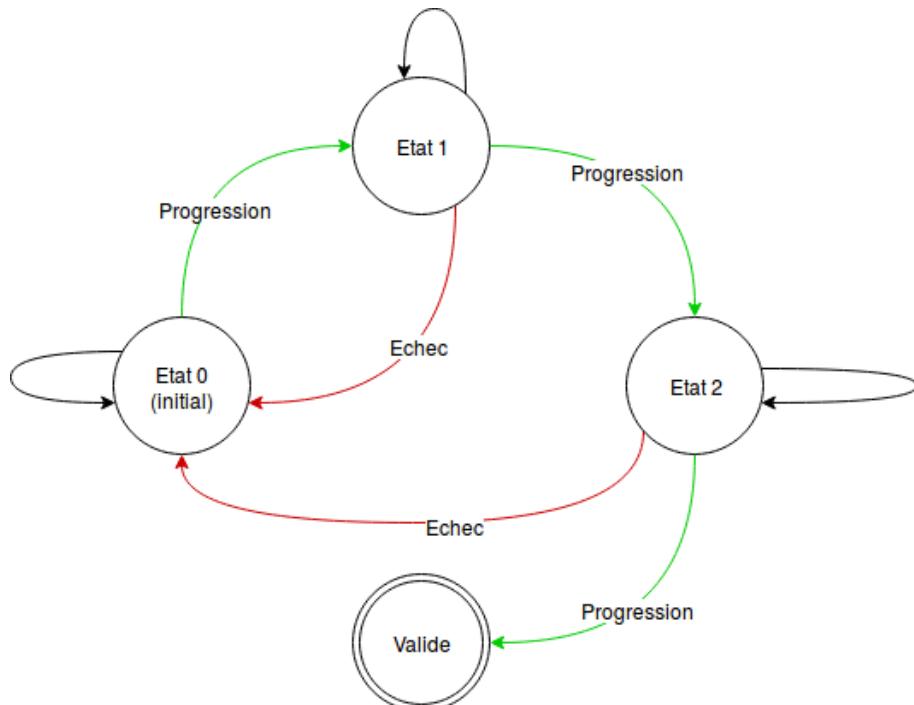


FIGURE 13 – Nos automates en général

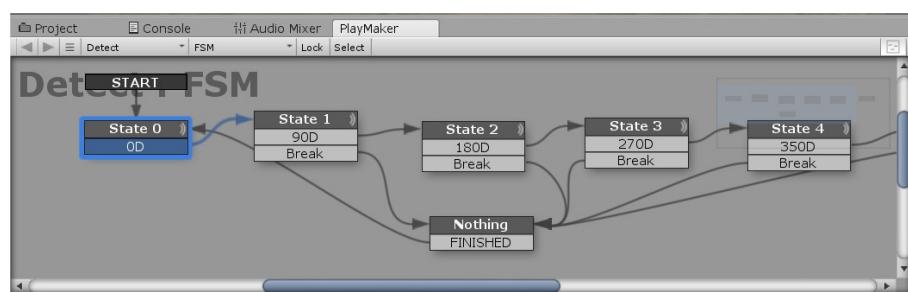


FIGURE 14 – Le looping avec Playmaker

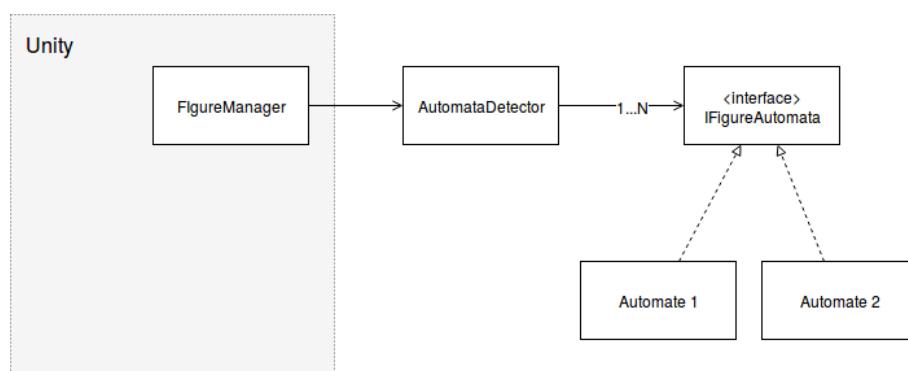


FIGURE 15 – Liens entre les automates et FigureManager

d'état. Elle va renvoyer 1 si l'automate arrive sur l'état final, -1 si l'automate rencontre un échec (et revient à l'état 0) et 0 dans un cas intermédiaire.

Quand un automate arrive sur son état final, il n'est pas censé recommencer à l'état 0, ce choix est laissé à *AutomataDetector* ou à des automates qui le réutilisent.

L'interface ajoute aussi des méthodes qui donnent le nom de la figure (ou l'automate) sous forme de chaînes de caractères, le nombre d'états de la machine ou l'identifiant de l'état actuel, ces méthodes ne sont pas utilisées mais peuvent servir de vérification.

Des automates qui réutilisent les autres

Avec ce type qui décrit un automate (ou une figure) de façon abstraite, il nous est venu à l'esprit de faire des automates de figures complexes, on a donc proposé deux classes abstraites d'automates :

- *AbstractSequentialFigureAutomata* correspond à une liste de figures à réaliser dans un ordre, étant donné N automates, cet "automate" va exécuter les automates dans l'ordre : quand l'automate 1 est considéré valide, on commence l'exécution du deuxième et ainsi de suite, une fois les N automates valides, on considère la figure réalisée.
Si l'un des automates échoue (retour -1 par `calculateState()`), tous les automates sont réinitialisés et on recommence le calcul depuis le premier. Cette classe permet de décrire une série de figures à réaliser, par exemple un double Looping ou un Looping suivi d'un Aileron Roll.
- *AbstractComposedFigureAutomata* correspond à une figure principale et un ensemble de figures secondaires. Ici, la figure est considérée comme "terminée" quand la figure principale arrive sur son état final, si toutes les figures secondaires sont aussi sur leur état final, alors la figure est validée, sinon on recommence depuis l'état initial. Si la figure principale recommence depuis l'état initial, toutes les autres sous-figures sont réinitialisées. Quand une figure secondaire échoue, cela n'a pas d'impact sur les autres figures.
Cette classe permettrait de décrire un regroupement de figures, par exemple une Looping avec un Aileron Roll pendant l'exécution du looping mais à n'importe quel moment.

À l'heure actuelle, ces deux classes d'automates ne sont pas utilisées, il s'agit ici de proposition pour des figures plus complexes.

6.3.2 Premières versions

Nos premières versions se sont basées sur la valeur des différents angles de l'avion : l'utilisateur choisissait le nombre d'états présents dans l'automate, définissant ainsi en même temps la précision de celui-ci, et ensuite l'automate ramenait les angles donnés par la structure Coordinate à un intervalle d'angles [0,360]. L'automate divisait ensuite cette courbe par le nombre d'états.

Cependant, cette représentation n'était pas pratique et ne marchait pas dans certains cas. En effet, la conversion des angles était difficile à gérer de part leur comportement dans Unity, comme mentionné plus tôt. Par exemple, la courbe représentant l'évolution du Pitch au cours d'un looping effectue un saut discontinu d'une valeur de 180 degrés, et est constante à une valeur dépendant de la rotation de l'avion au début du looping sinon. Ce saut est peu pratique à détecter, car selon comment le joueur effectue le looping, le saut peut s'effectuer en une ou multiples frames.

Nous avons alors décidé de ne plus regarder les angles, mais d'utiliser la valeur des produits scalaires des axes de l'avion avec l'axe Y du monde.

6.3.3 Modifications des coordonnées et nouvelles implémentations

L'utilisation des scalaires permet de traduire la trajectoire de l'avion par des sinusoïdes, ce qui est beaucoup plus compréhensible et manipulable que les courbes obtenues avec les angles. Nous avons simplifié la représentation de l'automate : il n'est désormais plus possible de choisir le nombre d'états, nous laissant ainsi plus de liberté pour l'écriture du programme au détriment d'une fonctionnalité peu importante.

Chaque figure "simple" (Looping et Aileron Roll) possède un automate à 5 états : 1 pour chaque "quart de figure". En effet, pour effectuer un Aileron Roll, l'avion doit pivoter vers la droite ou la gauche de 0 à 90, puis de 90 à 180, 180 à 270, 270 à 0, et enfin revenir dans l'état entre 0 et 90, qui correspond à l'état final. Parallèlement à cela, nous avons rajouté des conditions supplémentaires selon la figure : durée limite pour réaliser une figure, contrôle de l'altitude, ...

Pour les figures plus compliquées, on peut appliquer la même méthode, et on peut se faciliter la tâche en observant que la plupart sont une combinaison des états de l'automate Loop et ARoll. C'est notamment le cas du Cuban eight : deux quarts de Loop, deux quarts d'ARoll, deux quarts de Loop et deux quarts de Loop. Ainsi, l'implémentation de l'automate reconnaissant le Cuban eight fut très rapide.

6.4 Interface pour faciliter la création de nouvelles figures

A la demande du client, nous avons facilité au maximum l'utilisation du code afin que la construction de nouveaux automates soit plus intuitive et facile pour le développeur. Cette fonctionnalité a fait resurgir certains problèmes : là où nous n'avions qu'à rajouter quelques lignes de code pour éliminer des cas particuliers, il nous est maintenant impossible de demander à l'utilisateur de rajouter telle ou telle commande s'il utilise telle ou telle combinaison pour sa figure.

Nous avons alors dû apporter une flexibilité à notre code : par exemple, dans notre précédente implémentation, dès que l'avion se situait dans une position correspondant à un état, cet état s'activait et si le précédent n'était pas l'état actif, l'automate effectuait un reset. Cela ne posait pas de soucis concernant les figures simples, car un avion ne se retrouvait jamais deux fois dans la même orientation au sein de la même figure. Cependant, rien n'empêche l'utilisateur de créer la figure Loop + un quart de Loop. L'automate serait alors perdu car il se réinitialiseraient en boucle dû au fait qu'il ait deux états (le premier et le dernier) actifs en même temps. Pour remédier à cela, l'automate ne surveille maintenant plus que deux états : l'état actif et le suivant.

Nous avons finalement réussi à obtenir un programme facilement utilisable : l'utilisateur a à sa disposition 8 fonctions correspondant aux 4 quarts du Loop et de l'ARoll. Il peut les assembler comme il le souhaite pour créer la figure de son choix. Un exemple est donné dans la figure 16, où un Cuban eight a été codé de cette manière.

```

checkAltitude(plane, 50, 2);
checkTime(5);

figure[0] = Q1Loop();
figure[1] = Q2Loop();
figure[2] = Q3Loop();
figure[3] = Q3ARoll();
figure[4] = Q4ARoll();
figure[5] = Q1ARoll();
figure[6] = Q2Loop();
figure[7] = Q3Loop();
figure[8] = Q3ARoll();
figure[9] = Q4ARoll();

```

FIGURE 16 – Cuban eight simplifié

De plus, l'utilisateur a aussi à sa disposition 3 fonctions de vérification : une pour vérifier l'altitude de l'avion lors de la figure, une autre pour vérifier si le nez de l'avion garde la même orientation pendant une figure, et enfin une pour vérifier que le joueur ne met pas trop de temps à réaliser chaque partie de la figure.

Afin de pouvoir tester librement des combinaisons, un fichier CustomAutomata a été créé, correspondant à la figure "Custom Figure".

7 Reconnaissance de tracé : algorithmes \$1 / \$P

7.1 Idée de base

Avant même de penser à utiliser un automate pour la reconnaissance de figures, nous avions eu l'idée d'exploiter directement la trajectoire au cours du temps de l'avion. Le principe général serait d'utiliser un algorithme qui surveillerait en permanence l'évolution de la trajectoire et des différents états de l'avion afin d'y détecter une éventuelle figure. Après quelques temps de réflexion, nous sommes arrivés à la conclusion que l'évolution de l'altitude ainsi que les 3 angles de rotation (que nous avons ensuite remplacés par 3 scalaires) suffisaient à attribuer à chaque figure une combinaison unique de ces 4 courbes. La deuxième étape consiste en l'exploitation de ces courbes. C'est ainsi que nous avons utilisé l'algorithme \$1 et \$P.

7.2 Description de l'algorithme

\$1 [2] est un algorithme de reconnaissance de figures à un trait, conçu pour les interfaces utilisateur basées sur la saisie de gestes. Il fait partie de la famille d'algorithmes \$, contenant notamment \$N, qui accepte les figures à plusieurs traits, et \$P, qui améliore \$N. Ces algorithmes ont été créés par Jacob O. Wobbrock, Andrew D. Wilson et Yang Li, de l'université de Washington.

L'algorithme \$1 prend en entrée une liste de points, représentés par 3 nombres : x , y et $time$, et renvoie le nom de la figure qu'il a reconnu, ainsi que son score, correspondant à un pourcentage de reconnaissance. L'algorithme fonctionne en 4 étapes :

Étape 1 : Rééchantillonner le chemin de points

Comme les listes de points que l'on compare ne sont pas toutes de la même taille, on commence par rééchantillonner les points en N points équidistants (Figure 17). En pratique, on utilise $N = 64$.

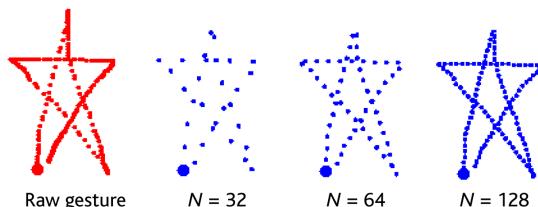


FIGURE 17 – Figure rééchantillonnée en $N = 32$, 64 et 128 points

Étape 2 : Tourner selon un certain angle

Maintenant que nous avons 2 figures de même taille, il faut en tourner une pour l'aligner avec l'autre. D'abord, on trouve l'*angle indicatif*, défini comme l'angle formé entre le centroïde de la figure (\bar{x}, \bar{y}) et le premier point de la figure. Ensuite, on tourne la figure pour que cet angle soit à 0° (Figure 18).

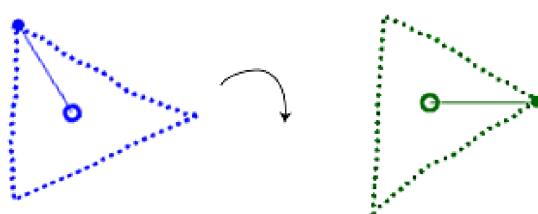


FIGURE 18 – Figure tournée pour que son *angle indicatif* soit égal à 0°

Étape 3 : Mise à l'échelle et translation

Après rotation, la figure est mise à l'échelle d'un rectangle de référence. Après cela, elle est translatée jusqu'à un point de référence. Ici, jusqu'à ce que son centroïde (\bar{x}, \bar{y}) soit à $(0,0)$.

Étape 4 : Trouver l'angle optimal pour le meilleur score

À ce moment, les figures candidates et les figures de référence ont subies les mêmes transformations. On peut maintenant procéder à la reconnaissance. Avec l'équation 1, le candidat C est comparé aux templates T_i pour trouver la distance moyenne d_i entre les points.

$$d_i = \frac{\sum_{k=1}^N \sqrt{(C[k]_x - T_i[k]_x)^2 + (C[k]_y - T_i[k]_y)^2}}{N} \quad (1)$$

Le template T_i correspondant au plus petit d_i est le résultat de la reconnaissance. Cette distance d est convertie en score avec l'équation 2 suivante :

$$score = 1 - \frac{d}{\frac{1}{2}\sqrt{size^2 + size^2}} \quad (2)$$

Ici, $size$ est la longueur d'un côté du carré de référence de l'étape 3. Le dénominateur correspond donc à la moitié de la diagonale et sert de limite à d .

L'algorithme a une complexité de $O(n.T)$, avec n le nombre de points choisis à l'étape 1, T le nombre de figures de référence.

7.3 Courbes de référence

Notre but était donc d'identifier chaque figure par différentes courbes d'évolution au cours du temps, passer à \$1 les courbes d'évolution de l'avion en tant que candidat, et si l'algorithme reconnaît ces courbes comme étant les courbes d'une figure de référence, cette figure est reconnue.

Pour utiliser l'algorithme, nous devons donc avoir des courbes de références pour chaque figure. Pour cela, nous avons utilisé *DummyPlayer* afin d'enregistrer et rejouer des trajectoires parfaites, et *PlaneTracker* afin de stocker les valeurs de l'avion dans un fichier. Après analyse de ces fichiers, nous avons remarqué qu'il était possible de distinguer chaque figure grâce à l'évolution sur le temps de 4 valeurs significatives : la hauteur (Y) et les produits scalaires entre les vecteurs *forward*, *up* et *right* de l'avion et le vecteur *up* du monde (c.f. figure 19).

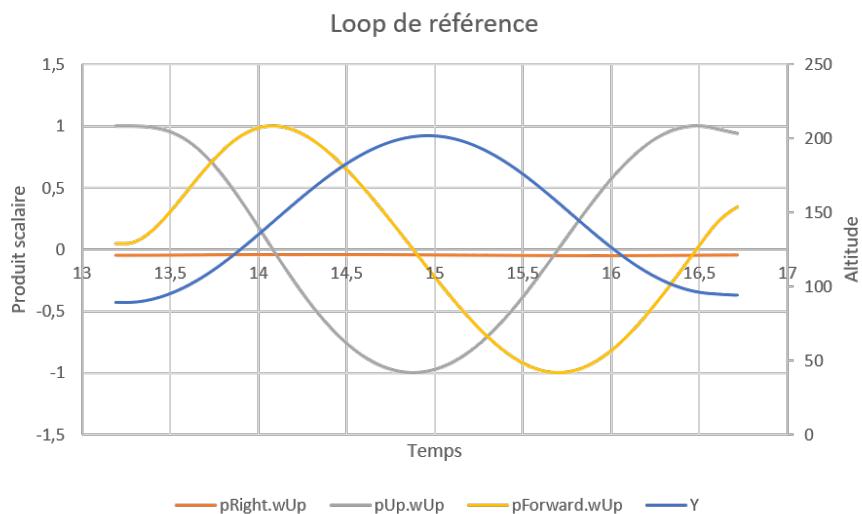


FIGURE 19 – Courbes utilisées pour reconnaître un Loop

7.4 Implémentation du code

7.4.1 De Coordinate aux courbes

Pour construire les courbes à analyser, il nous faut récupérer diverses données de l'avion. Pour cela le *FigureManager* appelle la fonction `setPoint()` avec en paramètre un *Coordinate* qui contient diverses valeurs de l'avion potentiellement utiles. Plus tard, cette structure sera remplacée par un *IFlyingObject*, interface implémentée par l'avion qui permet de récupérer directement les valeurs sur l'avion.

Dans la fonction `setPoint()`, l'algorithme récupère les valeurs utilisées pour les courbes et met à jour plusieurs buffers de points. Lorsque ces buffers sont pleins, l'élément le plus ancien est supprimé pour faire de la place.

Ces buffers sont ensuite utilisés dans la fonction `detection()`, qui passe cette liste de points à l'algorithme de reconnaissance de \$1 (\$P par la suite) qui la traite comme une courbe.

7.4.2 Des courbes aux figures

Lors de l'appel à la fonction `detection()`, on lance l'analyse des courbes définies précédemment avec l'algorithme \$. On lance pour cela autant d'analyses que de courbes (soit 4 dans notre cas).

Pour valider une figure, il faut que l'analyse reconnaissasse toutes les courbes composant cette figure. Lorsque c'est le cas, on renvoie la figure associée.

Exemple Pour valider un *Loop*, il faut que les 4 courbes analysées ressemblent chacune à une courbe précise de la figure de référence (c.f. figure 19).

- la courbe du produit scalaire entre *plane.right* et *world.up* doit ressembler à une ligne droite
- la courbe du produit scalaire entre *plane.up* et *world.up* doit ressembler à une bosse vers le bas
- la courbe du produit scalaire entre *plane.forward* et *world.up* doit ressembler à un zigzag
- la courbe de l'altitude doit ressembler à une bosse vers le haut

Si ces 4 courbes sont reconnues, cela signifie que le joueur a effectué un *Loop*.

Pour minimiser les erreurs de l'algorithme et le temps de calcul, chaque courbe n'est analysée qu'avec les courbes associées des figures de référence : l'altitude avec l'altitude de chaque figure, le produit scalaire entre *plane.right* et *world.up* avec le produit scalaire entre *plane.right* et *world.up* de chaque figure, ... Toutes les figures sont consultables en Annexe A.

7.4.3 Passage à \$P pour une meilleure détection

Pendant le projet, nous avons découvert que la détection effectuée par \$1 n'est pas très précise dans notre cas. En effet, l'algorithme a tendance à toujours reconnaître quelque chose comme étant une des figures qu'il connaît à plus de 70%, alors qu'ici, la plupart des courbes qu'il reçoit ne correspondent à rien. Cela posait des problèmes de reconnaissance, notamment de faux positifs.

Nous avons donc décidé de migrer sur l'algorithme \$P [1], version plus avancée de \$1, qui avait l'air plus précis, comme le montre la figure 20.

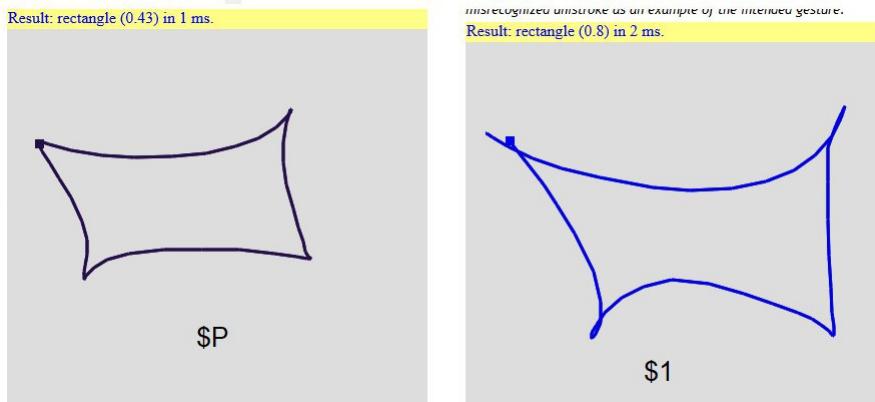


FIGURE 20 – Comparaison des algorithmes \$P et \$1 avec un rectangle mal dessiné

\$P commence par rééchantillonner, mettre à l'échelle et translater la figure, de la même manière que \$1. Cependant la reconnaissance se fait différemment, car \$P travaille sur des nuages de points. Le but est de relier chaque point d'un nuage avec un point de l'autre.

Pour chaque point du premier nuage (C_i), il trouve le point le plus proche qui n'a pas encore été relié. Une fois que C_i est relié, il continue avec C_{i+1} jusqu'à ce que tous les points soient reliés. La complexité est en $O(n^2)$ car chaque point de C doit faire une recherche linéaire sur T .

Cependant, le résultat est différent selon le C_k qu'on utilise pour commencer. Il faut donc recommencer avec plusieurs points de démarrage et renvoyer le plus petit résultat. Si C_k est le point de démarrage, alors :

$$\sum_i \omega_i \|C_i - T_j\| = \sum_{i=k}^n \|C_i - T_j\| + \sum_{i=1}^{k-1} \|C_i - T_j\| \quad (3)$$

Le poids ω_i correspond à la confiance que l'on a en la paire (C_i, T_j) . Le premier point a un poids de $\omega_1 = 1$ car il a toutes les données nécessaires pour prendre une décision. Au fur et à mesure de l'algorithme, cette confiance baisse car le nombre de points disponibles devient plus restreint. Ainsi, on a :

$$w_i = 1 - \frac{i-1}{n} \quad (4)$$

On note ε le paramètre contrôlant le nombre d'exécutions, tel que $\varepsilon = 0$ corresponde à 1 exécution et que $\varepsilon = 1$ corresponde à n exécutions. Sachant cela, \$P a une complexité de $O(n^{2+\varepsilon})$, soit dans le meilleur des cas $O(n^2)$ et dans le pire des cas $O(n^3)$.

8 Comparaison des algorithmes

8.1 Qualité de la détection

L'automate reconnaît les figures simples de manière quasi-systématique. Le Cuban eight est parfois reconnu sans que le joueur ne le veuille, cela est dû au fait qu'elle est composée de rotations simples de l'avion, et donc un joueur voulant faire deux looping en s'inclinant peut déclencher la reconnaissance du Cuban eight.

\$P reconnaît le Loop et le Roll mais cette détection se fait souvent trop tôt, au milieu de la figure. On peut donc faire une moitié de Roll qui est reconnue comme un Roll complet. Un autre problème est que le Cuban eight n'est pas reconnu, car cette figure nécessite un grand nombre de points (> 1000) qui ne rentre pas dans les tableaux de points que nous donnons en entrée à l'algorithme.

Plus généralement, le problème majeur des automates est qu'il n'y a que deux issues possibles : si le joueur passe exactement par tous les états de la figure, il la réussit, sinon elle est ratée. Cela amène à des situations où, par exemple, le Loop n'est pas détecté car le joueur ne s'est pas suffisamment repositionné à la verticale. Les algorithmes \$ n'ont pas ce problème : comme on cherche à identifier la figure dans sa globalité, une anomalie locale aura peu d'impact sur la détection finale.

8.2 Complexité des algorithmes

La complexité des algorithmes est un point crucial dans notre cas, car la détection est censé se faire en temps réel : un algorithme trop coûteux entraînera une chute du nombre d'images par seconde, rendant le jeu difficilement jouable.

La détection par automate est ici très efficace : sachant que la vérification d'une transition se fait en temps constant, la détection par automate dans sa complexité est de complexité linéaire en le nombre de figures détectables.

Les algorithmes \$ sont, quant à eux, beaucoup moins performants. La complexité d'au pire $O(n^2m)$, avec n le nombre de points d'échantillonnage et m le nombre de figures, de \$P en fait ainsi un algorithme extrêmement lourd en terme d'utilisation processeur.

La figure 21 permet ainsi de visualiser l'utilisation CPU de nos deux méthodes.

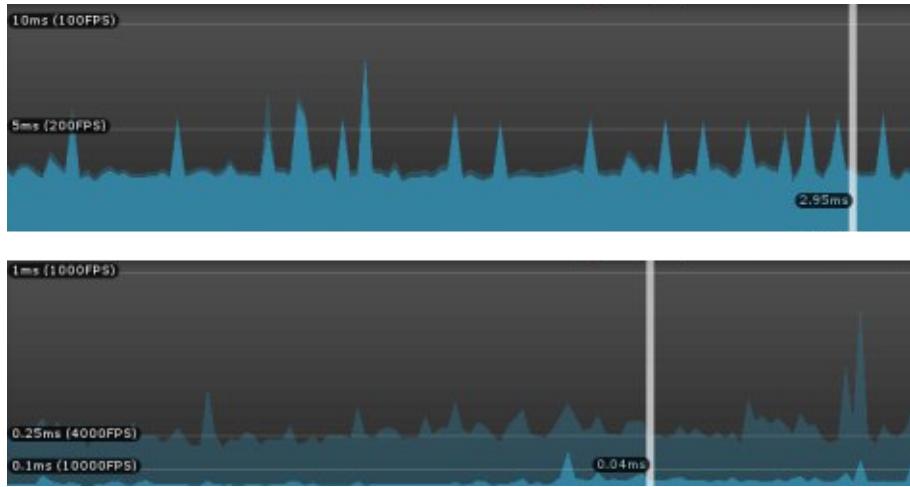


FIGURE 21 – Comparaison de l'utilisation processeur des 2 méthodes
(\$P en haut et Automate en bas)

8.3 Facilité d'utilisation

Le gros avantage de la méthode automate est sa facilité d'évolution. Comme vu précédemment, grâce à l'interface designer, il est très simple de rajouter et créer un nouvel automate pour une nouvelle figure.

Pour \$P, la démarche est un peu plus complexe, mais reste aisée. Si l'on met de côté le problème de taille de fenêtre, ajouter une figure est très simple puisqu'il suffit de l'enregistrer en jeu, ce qui crée un fichier CSV que pourra utiliser l'algorithme en tant que figure de référence, et cela ne dépend pas de la complexité de la figure. Il faudra tout de même penser à ajouter, nommer chaque courbe composant la figure, pour que celles-ci puissent être identifiées.

Dans les deux cas, il faut en tout dernier lieu indiquer au *FigureManager* l'existence de la nouvelle figure.

Quatrième partie

Travail réalisé et améliorations possibles

9 Chronologie du projet

Pendant les deux premiers mois, nous nous sommes surtout concentrés sur tout ce qui touchait à la simulation pour le jeu : physique, monde, menu principal... C'est aussi pendant cette période que nous avons réfléchi sur les différentes possibilités de méthodes pour la reconnaissance de figures. Ainsi, après s'être décidés sur l'implémentation de \$ et Automate, nous nous sommes concentrés sur l'ajout de ces méthodes lors des 3 derniers mois du projet, en plus de l'ajout d'une UI pour le retour utilisateur.

Dans le document de spécification, nous présentions un diagramme de Gantt prévisionnel du projet, visible en Figure 22. La Figure 23 montre le diagramme de Gantt du projet tel qu'il a été réalisé en pratique. On peut constater que les tranches de temps que nous avions fixé avec le diagramme n'ont pas vraiment été respectées, et que les dépendances entre les tâches sont peu visibles du fait de la fragmentation de leur réalisation. Cela nous laisse penser que le diagramme de Gantt tel que nous l'avions établi ne correspond pas à un développement d'un jeu vidéo : nous avons été amenés à modifier tout au long du projet des fonctionnalités codées auparavant, notamment basées sur le ressenti lors de l'expérience de jeu. De plus, c'est la première fois que nous nous lancions dans un projet de cette envergure, nous manquions donc de recul sur comment nous allions procéder, et il nous était difficile de correctement estimer le temps nécessaire pour réaliser chaque tâche.



FIGURE 22 – Diagramme de Gantt prévisionnel.
L'unité de durée utilisée ici est fictive, et ne sert qu'à différencier la durée des tâches entre elles.

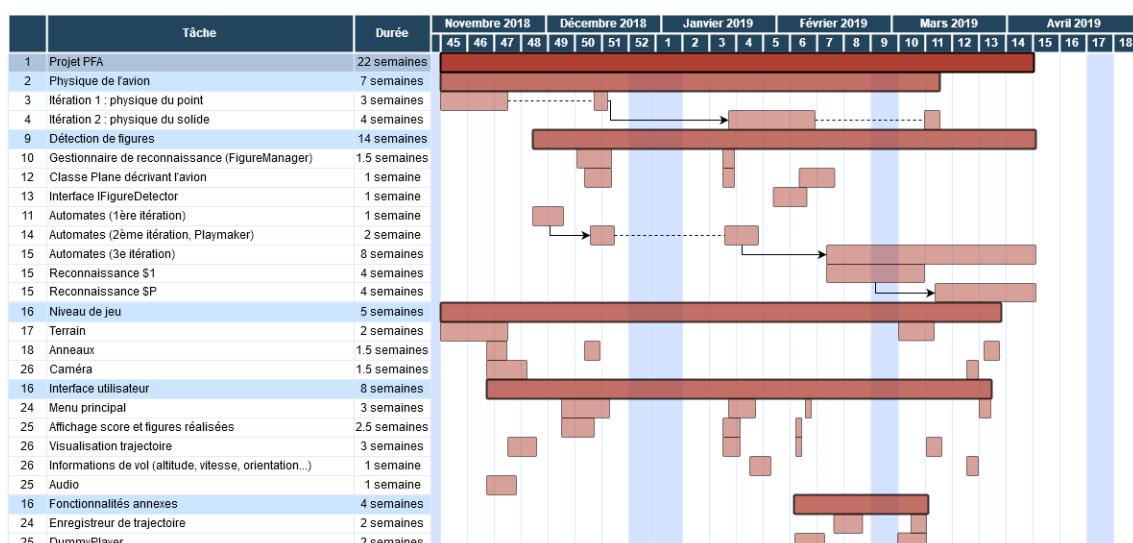


FIGURE 23 – Diagramme de Gantt réel.
La durée indique ici le nombre total de semaines qui a été dédié à chaque tâche.

10 Structure du code

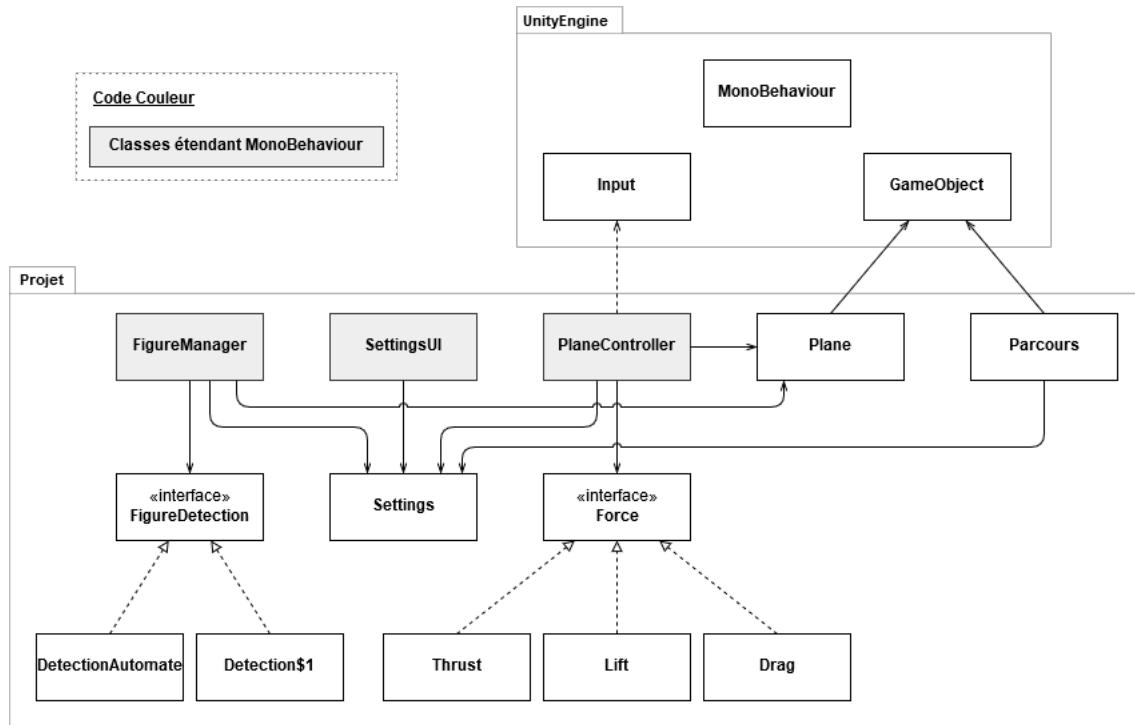


FIGURE 24 – Diagramme de classes prévisionnel

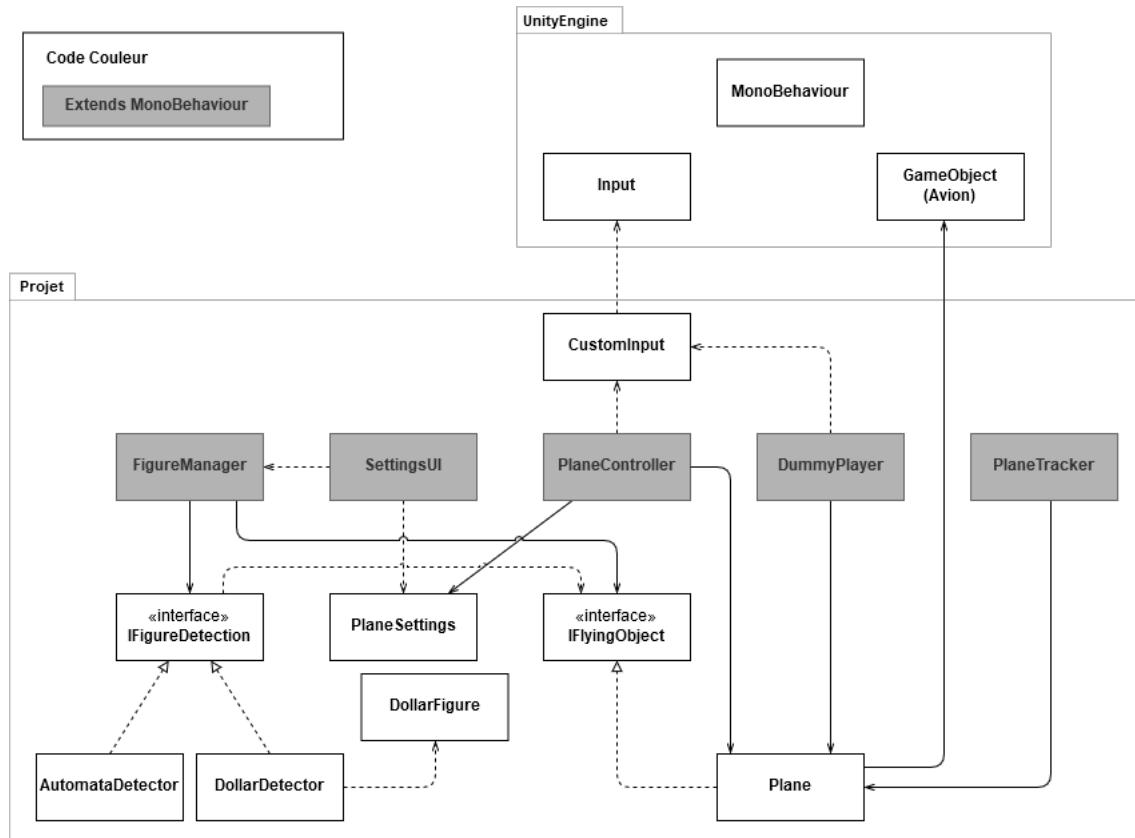


FIGURE 25 – Diagramme de classes réel

Les figures 24 et 25 présentent les diagrammes de classes prévisionnels et réel du projet. On peut remarquer que si la structure dans sa globalité est restée la même, quelques modifications ont été apportées au cours du développement.

Le changement notable est l'abandon de l'interface `Force`. En effet, nous nous sommes rendu compte qu'une telle interface rajoutait une couche de complexité peu utile. Actuellement, les forces sont ainsi directement calculées dans la classe `PlaneController`. De plus, nous avons rajouté une interface `IFlyingObject` permettant de standardiser la notion d'objet volant, et ainsi s'affranchir de la dépendance de `Plane` à Unity.

On peut aussi remarquer des changements plus mineurs : dorénavant, `SettingsUI` n'as plus de lien "a-un" avec la classe `Settings`, mais va plutôt par elle même modifier les valeurs statiques dans les classes correspondantes. Nous avons aussi une nouvelle classe `DollarFigure` standardisant certains aspects de la détection \$.

Enfin, deux nouvelles classes sont apparues, `DummyPlayer` et `PlaneTracker`. Ce sont elles qui se chargent d'enregistrer et rejouer des séquences.

11 Améliorations envisageables

11.1 Niveau de jeu

Un point notable du niveau de jeu actuel est qu'il n'est pas sans limite : le terrain est une surface carrée dont on peut atteindre les extrémités. Le problème est qu'actuellement, rien n'empêche d'aller au delà de ces limites : il serait donc intéressant de proposer, au choix, un terrain se répétant à l'infini, un terrain généré automatiquement au cours du jeu, ou tout simplement instaurer un système empêchant le joueur de quitter la zone de jeu.

11.2 Physique de l'avion

Comme mentionné précédemment dans ce document, nous avons effectué un certain nombre d'approximations pour le modèle physique de l'avion, les plus importants étant la valeur des constantes physiques et les points d'applications des forces. Il est en effet remarquable que la chute de l'avion paraît très lente, et la décélération très rapide. Cela est en fait dû à une constante de frottement choisie arbitrairement grande, qui a ensuite impacté l'ensemble des valeurs choisies, trop importantes elles aussi. De plus, les points d'applications de forces n'étant pas exactement ceux d'un avion réel, il se pourrait que la physique s'en retrouve légèrement changée.

11.3 Détection par automates

Il reste certains cas où la détection de figure est défectueuse (exemple : faire un Aileron Roll juste après un looping), nous aurions aimé pouvoir gérer ces cas.

On peut imaginer plusieurs améliorations à apporter au niveau de l'interface designer et la facilité de création de figures. Une idée serait d'implémenter la même fonctionnalité qu'avec l'algorithme \$, c'est-à-dire la possibilité d'enregistrer des figures à la main dans l'éditeur, et le programme se chargerait de traduire la trajectoire en automate. Une interface graphique pourrait aussi être ajouté, automatisant ainsi la création de figure et du fichier associé.

11.4 Reconnaissance de tracé (\$1/\$P)

La reconnaissance de tracé avec l'algorithme \$ reconnaît les figures un peu trop tôt, certaines sont détectées au milieu de la figure. Il faudrait voir à ajuster la précision minimale requise des figures.

Il faudrait également modifier la manière de construire les courbes car pour l'instant, lors du redimensionnement des courbes au sein de \$, la variation en y des points est beaucoup plus grande que la variation en x , ce qui rend les courbes plutôt plates et très similaires dans l'ensemble, puisqu'en ligne droite, toutes les figures sont reconnues avec plus de 90% de précision.

Il a également quelques cas où une figure n'est pas détectée alors qu'elle est correctement réalisée. Il faudrait donc améliorer la précision de la reconnaissance pour ces cas là.

Une autre chose à améliorer sur cet algorithme est la vitesse de calcul qui est un peu lente. Une manière de l'améliorer serait, par exemple de faire les calculs en parallèle sur plusieurs processeurs.

De plus, il faut noter que les algorithmes $\$$ sont prévus pour reconnaître tout type de tracé, alors que nous utilisons ici uniquement des courbes, l'algorithme pourrait donc être optimisé.

Conclusion

Nous avons obtenu des résultats concluants avec les deux méthodes Automate et $\$$. La reconnaissance ne fonctionne malheureusement pas à 100%, mais nous avons quand même pu saisir le principe et la complexité qui se cache derrière le sujet à l'apparence simple de la reconnaissance de figures aériennes. Ce projet nous a permis de nous mettre face aux besoins d'un client, tout en devant travailler avec un groupe de taille non négligeable. Le suivi au cours de l'année par le professeur encadrant et le client nous a guidé vers la bonne démarche à suivre pour réaliser un projet de cette envergure. Même si nous ne sommes pas allés aussi loin que nous l'aurions voulu, nous envisageons la possibilité de transmettre notre code aux élèves l'année prochaine dans le cadre du PFA afin qu'ils étudient, encore plus en détail et avec une base de code, ce vaste sujet.

Références

- [1] Yang Li JACOB O. WOBBROCK Andrew D. Wilson. *Gestures as point clouds : A \$P recognizer for user interface prototypes*. 2012. URL : <http://faculty.washington.edu/wobbrock/pubs/icmi-12.pdf>.
- [2] Yang Li JACOB O. WOBBROCK Andrew D. Wilson. *Gestures without libraries, toolkits or training : A \$I recognizer for user interface prototypes*. 2007. URL : <http://faculty.washington.edu/wobbrock/pubs/uist-07.01.pdf>.

A Figures de références pour \$P

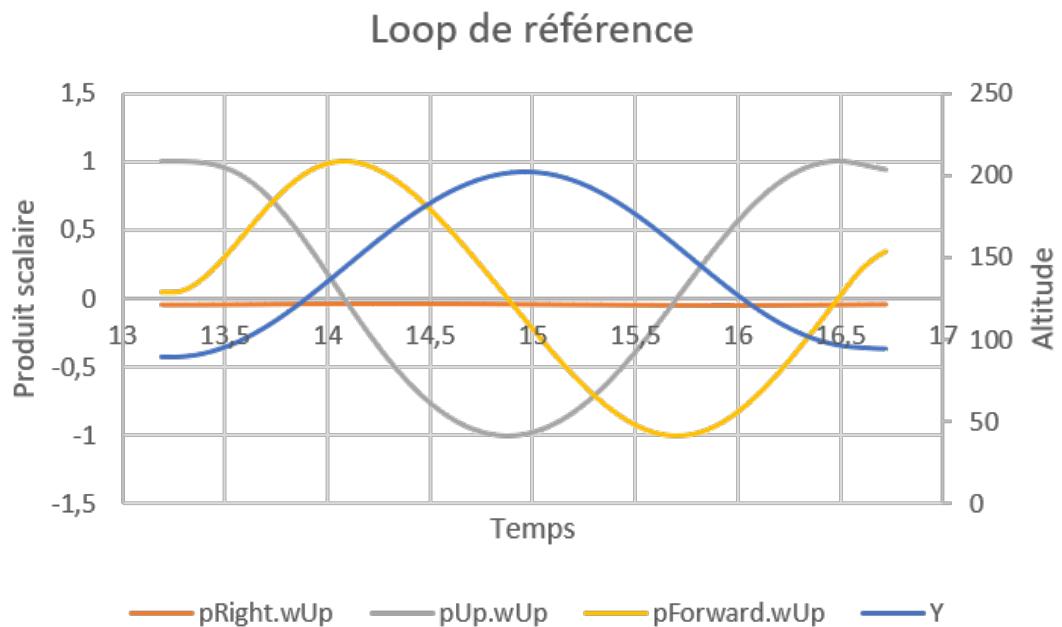


FIGURE 26 – Loop de référence

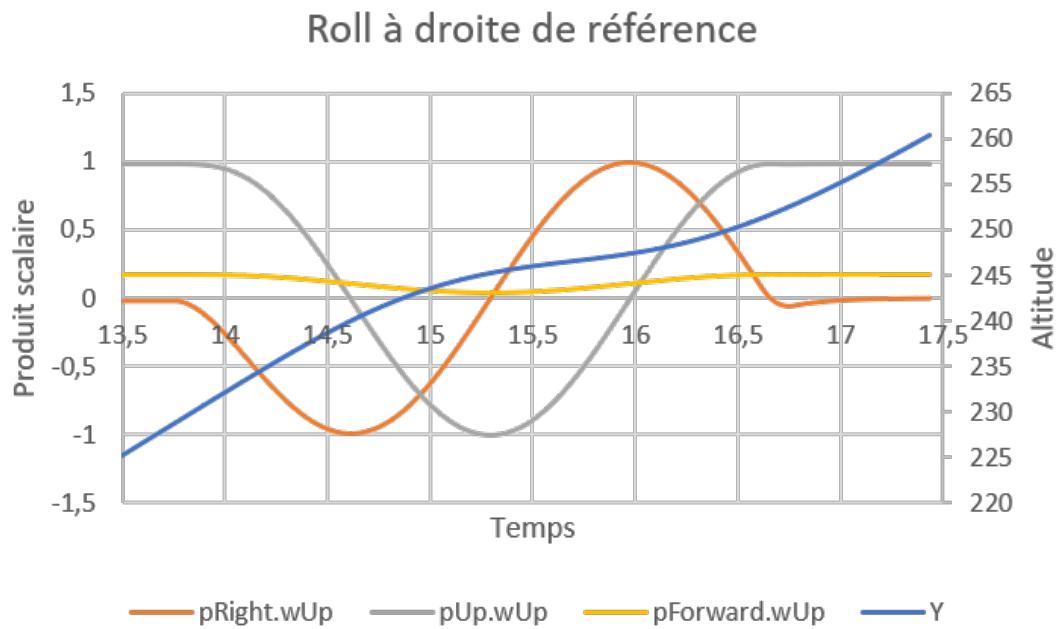


FIGURE 27 – Roll à droite de référence

Roll à gauche de référence

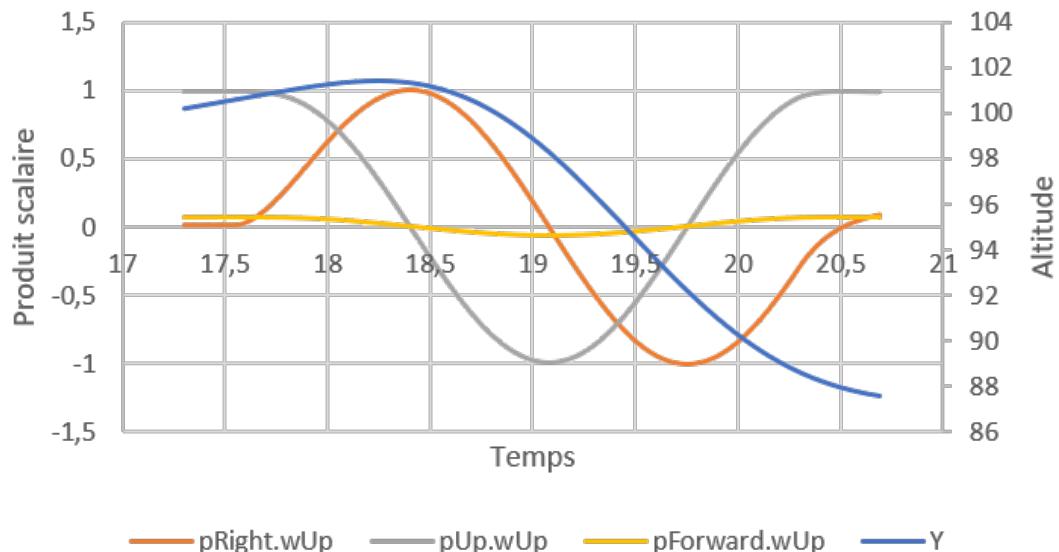


FIGURE 28 – Roll à gauche de référence

Cuban-eight de référence

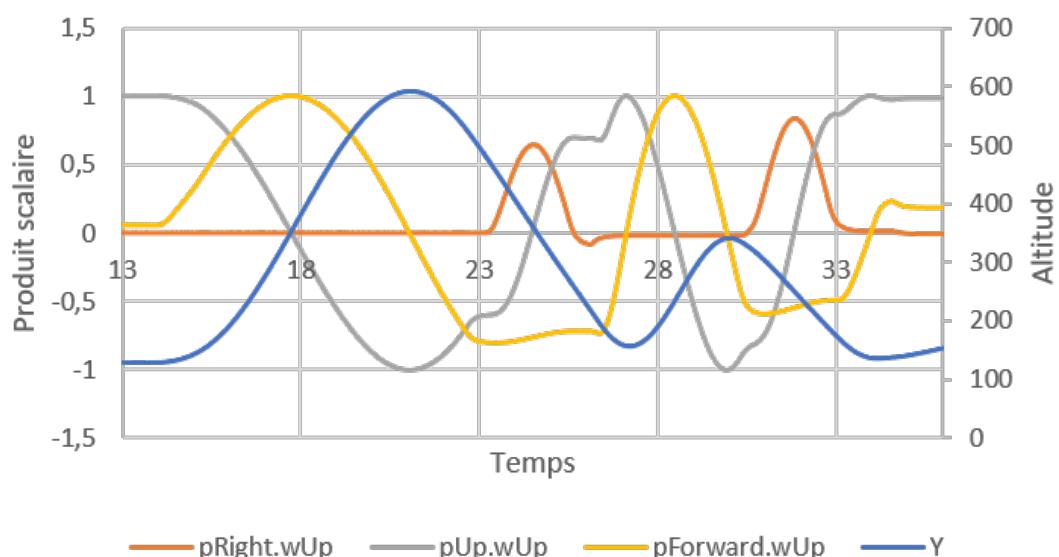


FIGURE 29 – Cuban-eight de référence

Trajectoire quelconque

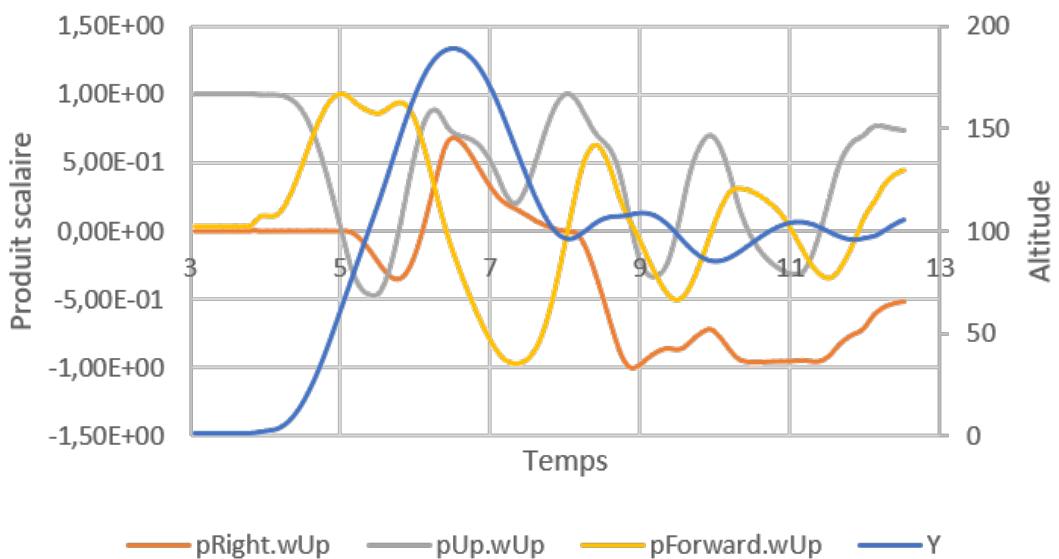


FIGURE 30 – Trajectoire quelconque

B Document de spécification des besoins



Trajectory detection for game scoring

PFA - Document de spécification des besoins

Louis Cesaro, Timothée Corsini, Jean Farines,
Arno Galvez, Adrien Lavancier, Anatole Martin, Félix Pierre

Client : Martial Bossard

Responsable Pédagogique : David Renault

11 décembre 2018, ENSEIRB-MATMECA

Table des matières

1	Introduction et motivation	3
1.1	Définition du projet	3
1.2	Objectifs du projet	3
1.3	Figures	3
2	Besoins fonctionnels	4
2.1	Contenu visible	4
2.2	Fonctionnalités	4
2.3	Physique de vol	4
2.4	Détecteur de figures	4
2.4.1	Figures détectées	4
2.4.2	Algorithmes de détection	6
3	Besoins non fonctionnels	7
4	Architecture et conception	7
4.1	Gestion des objets dans le moteur de jeu	7
4.2	Physique de l'avion et détection des figures	8
5	Évolution du système	8
6	Plan de tests de validation	9
6.1	Tests sur les algorithmes de reconnaissance	9
6.2	Tests sur les déplacements de l'avion	9
6.3	Tests d'intégration	9
7	Planning prévisionnel	10

1 Introduction et motivation

1.1 Définition du projet

Ce dossier de spécification renferme la description complète du projet effectué dans le cadre du PFA à l'ENSEIRB-MATMECA, en collaboration avec Asobo Studio. Ce document est une vue d'ensemble du projet, et comprend la description des besoins, des algorithmes de reconnaissance de figures et de calcul de physique, la spécification de l'architecture de développement, les éventuelles améliorations à apporter et l'environnement de test.

Le projet décrit dans ce document est un prototype de jeu de course de style MicromachinesTM avec obstacles capable de reconnaître des figures de voltige aérienne réalisées par le joueur (cf. partie 1.3). Les jeux MicromachinesTM sont des jeux de course mettant en scène des voitures miniatures dans des circuits à échelle humaine (salon, table de billard, cuisine...). Le but serait donc d'avoir le même type de parcours avec des avions miniatures. Néanmoins, le cœur du projet sera l'algorithme de reconnaissance des figures, qui pourra être réutilisé par l'entreprise dans le cadre de futurs projets.



FIGURE 1 – Course de voitures MicromachinesTM issue du jeu vidéo *Micro Machines World Series*



FIGURE 2 – Avion effectuant une figure lors d'un spectacle de voltige aérienne

1.2 Objectifs du projet

Le jeu final devra comporter au moins un parcours d'obstacle, où évoluera l'avion du joueur. Cet avion aura une physique similaire à un avion de voltige, avec les forces adaptées (portance, poids, accélération...), tout en restant dans un cadre ludique.

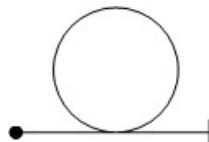
Lorsque le joueur effectue une figure, le logiciel la reconnaît et attribue des points selon la précision de la figure. Le logiciel de reconnaissance devra également prendre en compte les combinaisons de figures.

Le parcours consistera en un petit monde que le joueur devra traverser d'un bout à l'autre. Il comprendra des obstacles (statiques ou mobiles par la suite) qui inciteront le joueur à effectuer des figures pour les éviter. Le niveau pourra éventuellement être généré de manière procédurale.

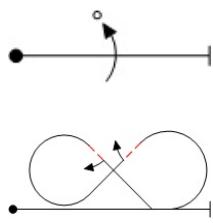
1.3 Figures

Une figure est une succession de manœuvres vérifiant un ensemble de conditions telles que la durée, l'orientation et l'altitude.

Le système de reconnaissance de figures gérera les figures reconnues officiellement par la FAI (Fédération Aéronautique Internationale) dans les documents de la CIVA (FAI Aerobatics Commission) [1]. Voici quelques exemples :



Loop : L'avion s'oriente vers le haut jusqu'à revenir dans sa position initiale. Il doit garder ses ailes horizontales lors de la manœuvre et terminer à la même altitude que celle à laquelle il commence.



Barrel Roll : L'avion effectue un *Loop* tout en effectuant une rotation sur lui-même, en s'orientant vers la gauche ou vers la droite. Sa trajectoire a la forme d'un tire-bouchon.

Cuban Eight : L'avion effectue 5/8 d'un *Loop* vers le haut puis fait la même chose vers le bas, jusqu'à revenir à sa position initiale. Les deux cercles doivent avoir le même rayon et la même altitude. L'avion doit aussi avoir la même altitude au début et à la fin de la manœuvre.

2 Besoins fonctionnels

2.1 Contenu visible

Ce jeu sera développé avec l'aide du moteur Unity. Puisque le projet est un jeu vidéo en 3 dimensions, il devra posséder un environnement visible. Dans cet environnement, les éléments suivants devront être présents et visibles par le joueur :

- **Avion** : Un modèle d'avion, créé ou pris d'un modèle libre de droits, qui sera contrôlé par le joueur.
- **Environnement** : Un environnement, constitué d'un sol et d'un ciel qui servira de décor pour le jeu et dans lequel le joueur pourra évoluer.
- **Parcours** : Un parcours constitué d'anneaux indépendants de l'environnement, dans lesquels le joueur devra passer pour gagner. Le jeu devra également comporter des parcours d'entraînement pour apprendre les figures. Ces parcours comporteront des anneaux pour guider le joueur lors de la réalisation de la figure.

2.2 Fonctionnalités

Le jeu devra également intégrer différents algorithmes permettant de réaliser les éléments suivants :

- **Vol** : Il doit être possible de faire voler un avion et de faire des figures. Pour cela, une physique simplifiée sera utilisée (cf. partie 2.3).
- **Détection de figures** : Le jeu doit pouvoir détecter les figures réalisées par le joueur (cf. partie 2.4)
- **Score** : Le jeu doit attribuer des points au joueur selon la figure réalisée et la qualité d'exécution de cette figure, selon les critères de notation de la CIVA.

2.3 Physique de vol

La physique sera proche du modèle défini sur la page wikipedia de la dynamique de vol [4] :

- **Forces** : Seules les forces relatives au poids, à la puissance du moteur (traction), à la portance et à la traînée seront considérées (cf. figure 3). La force du vent sur l'avion ne sera pas considéré.
- **Rotations** : Les trois rotations devront être considérées (lacet (*yaw*), tangage (*pitch*) et roulis (*roll*)) (cf. figure 4).

2.4 Détecteur de figures

2.4.1 Figures détectées

Dans un premier temps, le détecteur devra reconnaître les figures de base telles que les *Loops* et les *Barrels Roll* selon une méthode probabiliste (chaque figure aura un taux de correspondance associé, le taux le plus haut sera la figure reconnue). Des figures plus avancées telles que le *Cuban Eight* et des figures officielles de compétition présentées dans les documents de la CIVA [1] seront ajoutées au fur et à mesure à l'algorithme de reconnaissance. Les figures devront être reconnues si

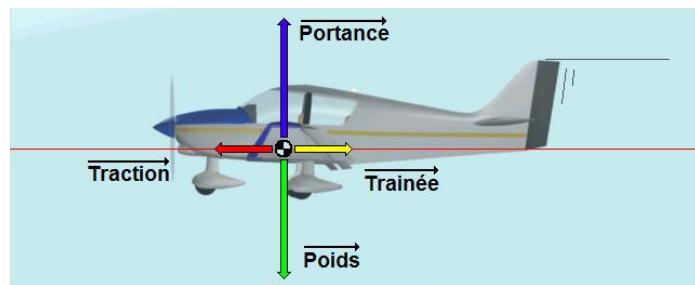


FIGURE 3 – Physique simplifiée de l'avion

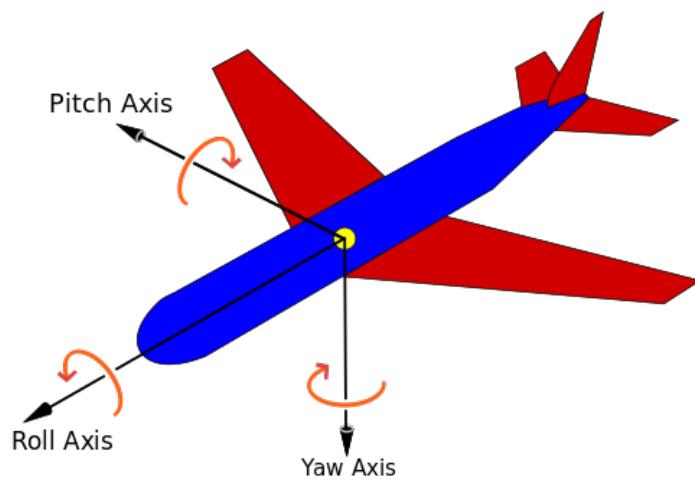


FIGURE 4 – Physique simplifiée de l'avion

elles possèdent un pourcentage de ressemblance suffisamment élevé par rapport au modèle parfait, et les faux positifs¹ devront être évités au maximum.

2.4.2 Algorithmes de détection

Le détecteur intégrera plusieurs méthodes de détection dont :

- une méthode "naïve", basée sur un automate dont les états suivent l'inclinaison de l'avion (cf. figures 5 et 6).
- une méthode plus avancée, basée sur du *pattern matching*² s'appuyant sur l'algorithme \$1 [2]. Le *pattern matching* sera effectué sur l'évolution de la hauteur et des rotations de l'avion au fil du temps (cf. figure 7).

Ces méthodes devront pouvoir être comparées en termes de temps de calcul et de précision de la reconnaissance. Il faut de plus que ces méthodes puissent analyser les figures en continu afin de n'en manquer aucune, ainsi que prendre en compte la réalisation de plusieurs figures à la suite.

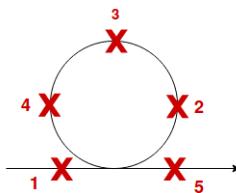


FIGURE 5 – Exemple de grammaire pour reconnaître un *loop*

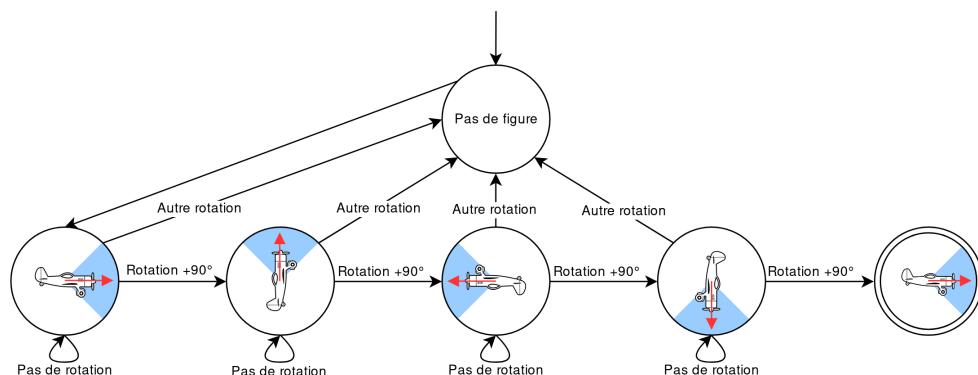


FIGURE 6 – Exemple d'automate capable de reconnaître un *loop*

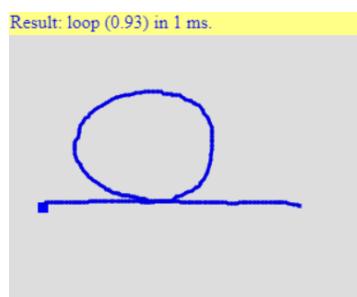


FIGURE 7 – Exemple de pattern matching de l'algorithme \$1 appliquée à un *loop*

1. Trajectoires reconnues comme des figures valides alors qu'elles n'en sont pas
 2. Reconnaissance de motifs

3 Besoins non fonctionnels

Le projet se présentant sous la forme d'un jeu vidéo, certains besoins doivent être remplis afin que l'utilisateur puisse tester les fonctionnalités dans le meilleur environnement possible. Le jeu se doit donc d'être :

- **Fluide** : Le *framerate* minimal sera de 30 images par secondes, afin que l'expérience de jeu soit fluide et non saccadée. En étant optimiste, cette barre pourra être mise par la suite à 60 IPS³ minimum.
- **Ergonomique** : Piloter un avion est très complexe, notre but est de transcrire cela le plus intuitivement possible, permettant au joueur de comprendre rapidement comment évoluer dans l'environnement. Le joueur pourra choisir entre un clavier et une manette : il est beaucoup plus intuitif de commander un avion avec des joysticks, mais tout le monde n'en possède pas, une implantation des deux est donc nécessaire.
- **Réaliste** par rapport à la physique d'un avion : Afin de renforcer l'immersion, la physique de l'avion devra être ludique et réaliste : le joueur devra avoir le sentiment de piloter un vrai avion, sans pour autant devoir faire face aux contraintes auxquelles un vrai pilote ferait face. La physique devra être telle que le joueur puisse intuitivement exécuter des figures complexes, comme celles décrites plus haut, ainsi qu'interagir avec son environnement (passer sous un pont, dans un anneau....).
- **Rejouable** : l'ajout d'un score augmentant en fonction des figures effectuées par le joueur lui permettra d'avoir un retour direct sur ses performances. Une figure complexe et bien réalisée rapportera plus de points qu'une figure simple ou pauvrement exécutée. Un chronomètre lors des parcours permet au joueur d'avoir envie de le refaire mieux et plus rapidement.
- **Portable** : Il faut que le produit final soit compatible sur Windows 64-bits, et qu'il ne soit nécessaire d'installer aucun logiciel particulier tel qu'Unity ou Visual Studio pour pouvoir lancer le jeu.

4 Architecture et conception

Le projet sera réalisé avec l'aide du moteur de jeu Unity dans sa version 2018.2.15f1. Le langage utilisé pour le développement avec Unity est le C#, nous adopterons donc une architecture orientée objet dans l'ensemble de notre code.

L'utilisation du moteur se fait par l'utilisation des classes et méthodes du package `UnityEngine`[3]. En particulier, nous utiliserons principalement la classe `MonoBehaviour`, englobant les méthodes et classes appelées à chaque *frame*, `Input`, qui permet de récupérer les commandes de l'utilisateur, ainsi que `GameObject`, permettant de contrôler les objets présents dans le jeu (affichés ou non).

Le code du projet va être amené à gérer différents éléments dans le jeu, nous prévoyons donc de découper nos classes selon le modèle suivant, illustré en Figure 8 et détaillé ci-après.

4.1 Gestion des objets dans le moteur de jeu

Le moteur de jeu Unity dispose par défaut d'un grand nombre de classes permettant d'interagir avec les objets présents dans le jeu. Il nous faut donc de notre côté des classes avec lesquels nous récupérons des informations sur les objets et les contrôlerons :

- **Plane** : Cette classe correspond à l'avion que contrôle le joueur. Elle fait le lien entre notre code et le modèle visible en jeu, et permet de regrouper les informations importantes (position, rotations, vitesse...).
- **PlaneController** : Classe gérant les calculs sur la physique de l'avion. Elle fait notamment le lien entre les *inputs* du joueur et leur influence sur les forces à appliquer.

3. Images Par Seconde

- **FigureManager** : Cette classe coordonne la reconnaissance périodique de figures. C'est elle qui communique à l'algorithme les positions de l'avion, traite le résultat, attribue les points, et communique les résultats au joueur.
- **SettingsUI** : Elle définit une interface utilisateur (*UI*) permettant la consultation et/ou la modification des paramètres de jeu.

4.2 Physique de l'avion et détection des figures

La modélisation physique de notre avion passe par le calcul de différentes forces. La reconnaissance de figures, quant à elle, se fait via plusieurs méthodes. Nous disposons donc des classes suivantes :

- **Force** : Ce sont les classes caractérisant chacune des forces s'appliquant sur notre avion. Elles permettent, à partir de l'état de l'avion à un instant donné, d'obtenir un vecteur force utilisable par le moteur de jeu.
- **FigureDetection** : Cette interface et ses implémentations sont les différentes méthodes de détection de figures que nous utiliserons dans notre projet.
- **Settings** : Cette classe rassemble la majorité des paramètres de jeu : propriétés de l'avion, figures à détecter, etc.

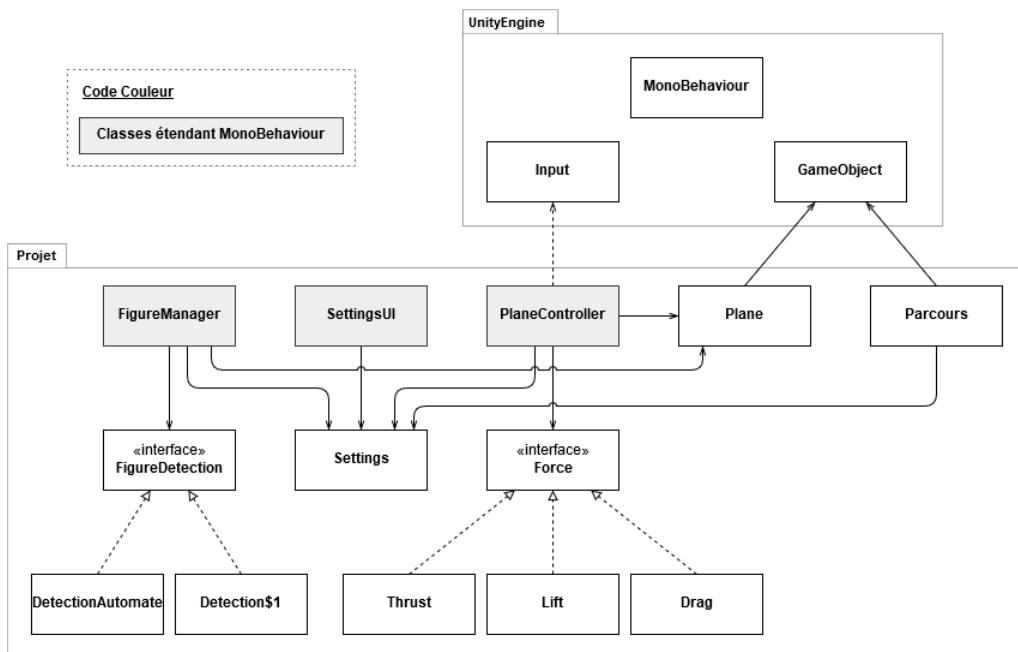


FIGURE 8 – Diagramme de classe prévisionnel du projet

5 Évolution du système

Par la suite, le projet pourra voir l'apparition de nouvelles fonctionnalités telles que :

- **Une génération procédurale** de parcours : le lieu du parcours et le parcours doivent pouvoir être générés de façon procédurale afin d'avoir une nouvelle expérience de jeu à chaque fois.
- **Une interface "d'entraînement"** avec mise en place d'anneaux requérant une rotation particulière : afin de pouvoir s'entraîner, il devra être possible de générer des parcours à suivre via des anneaux ou autres nécessitant un contact particulier afin de réussir à réaliser des figures plus ou moins complexes.

- **Différents modes de difficulté** : le joueur pourra être en mesure de modifier la difficulté du mode. Cela se traduira notamment par la baisse ou l'augmentation de la précision de la reconnaissance de figure.
- **La variété des modes de jeu** : divers modes de jeux pourront être implémentés et proposées au joueur, afin de pouvoir tester les différentes fonctionnalités : course contre la montre, freestyle...
- **L'ajout d'obstacles mobiles** : il sera possible de rendre les parcours plus difficiles en ajoutant, lors de la génération desdits parcours, des obstacles non plus statiques mais mobiles. Ce point, indiqué par le joueur avant de lancer un parcours, sera pris en compte lors de la création du parcours.

6 Plan de tests de validation

Pour vérifier le bon fonctionnement du projet, des tests à différentes échelles seront réalisés.

6.1 Tests sur les algorithmes de reconnaissance

Les tests sur les algorithmes de reconnaissance seront à faire de préférence sans Unity pour qu'ils soient indépendants du logiciel. On fera une série de tests pour chaque méthode de reconnaissance et pour chaque figure à reconnaître. Les figures seront enregistrées sous la forme de série de points, on commence par vérifier que les figures parfaites (en proportions) sont toujours reconnues. On prendra ensuite des figures jugées reconnaissables pour regarder leur validité. Enfin, on vérifiera que les séries de points qui ne représentent pas la forme attendue ne sont pas reconnues, ce qui permettra de vérifier qu'on distingue bien les figures différentes mais aussi qu'on ne reconnaît pas les déplacements qui ne représentent aucune figure. Un algorithme de test simple serait de la forme :

```
void testReconnaissance(methode, forme) {
    figure = formeParfaite(forme);
    assert(correspondance(methode(figure), forme));
    Pour figure dans listeFiguresValides {
        assert(correspondance(methode(figure), forme));
    }
    Pour figure dans listeFiguresInvalides {
        assert(!correspondance(methode(figure), forme));
    }
}
```

La liste de figures reconnues pourra être agrandie au fur et à mesure du projet. Des tests plus poussés pourraient évaluer des cas plus ambigus, par exemple la détection de deux figures avec un taux de reconnaissance similaire. Ces tests afficheront la vitesse de calcul des méthodes de reconnaissance.

6.2 Tests sur les déplacements de l'avion

Des tests sur le déplacement de l'avion permettront de vérifier que les forces s'appliquent correctement. On observe les déplacements de l'avion pour voir s'ils sont cohérents, notamment vis-à-vis de la vitesse de déplacement ou encore des virages. Ces tests seront réalisés sous Unity.

Une partie de ces tests pourrait fonctionner avec une série de commandes donnée en entrée, et vérifierait que le déplacement est cohérent.

Des tests de prise en main et de jouabilité lors du déplacement de l'avion pourront être réalisés au travers du "didacticiel", avec des éléments de décors comme repères tels que des cercles pour observer les déplacements de l'avion.

6.3 Tests d'intégration

Enfin, les tests d'intégration auront pour but de vérifier que le système de déplacement de l'avion fonctionne correctement avec le système de reconnaissance de figures. Nous vérifierons en temps

réel qu'une figure est correctement reconnue et de manière fluide, que l'avion bouge correctement selon les commandes, sans bugs d'affichage. Ce test sera fait sous la forme d'un didacticiel de prise en main avec pour objectif la réalisation de figures demandées, le déplacement dans les bonnes zones, avec un retour indiquant la réussite ou non des épreuves.

7 Planning prévisionnel

A la demande du client, un aperçu du projet sous la forme d'exécutable sera livré au client tous les troisièmes mardis de chaque mois, afin d'avoir une vue sur l'avancement du projet et de pouvoir corriger si besoin est.

Une découpe des tâches du projet comme indiqué sur la figure 9 sera à prévoir lors de la suite du projet.

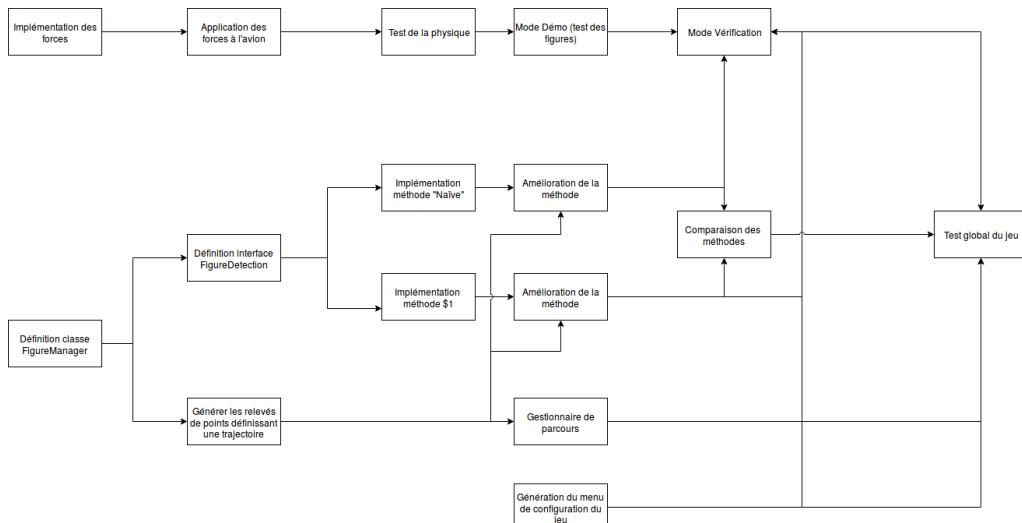


FIGURE 9 – Diagramme de découpe des tâches du projet



FIGURE 10 – Diagramme de Gantt du projet (durées relatives)

Références

- [1] CIVA. *Free Known Programme 1 Power Intermediate / Yak-52, Advanced an Unlimited*. 2018. URL : https://www.fai.org/sites/default/files/documents/civa_free_known_programme_guidance_-_power_aircraft_2018_v3a.pdf.

- [2] Yang Li JACOB O. WOBBROCK Andrew D. Wilson. *Gestures without Libraries, Toolkits or Training :A \$1 Recognizer for User Interface Prototypes*. 2007. URL : <http://faculty.washington.edu/wobbrock/pubs/uist-07.01.pdf>.
- [3] UNITY. *Unity Scripting Reference*. 21 Novembre 2018. URL : <https://docs.unity3d.com/ScriptReference/>.
- [4] WIKIPEDIA. *Flight dynamics (fixed-wing aircraft)*. 12 Juillet 2018. URL : [https://en.wikipedia.org/wiki/Flight_dynamics_\(fixed-wing_aircraft\)](https://en.wikipedia.org/wiki/Flight_dynamics_(fixed-wing_aircraft)).

C Manuel d'utilisation et de maintenance



Trajectory detection for game scoring

Reconnaissance de trajectoire pour le jeu vidéo

PFA - Manuel d'utilisation et de maintenance du code

Louis Cesaro, Timothée Corsini, Jean Farines,
Arno Galvez, Adrien Lavancier, Anatole Martin, Félix Pierre

Clients : Martial Bossard, Pierre-Marie Plans

Responsable Pédagogique : David Renault

8 avril 2019, ENSEIRB-MATMECA

Table des matières

1 Ouverture du projet	3
1.1 Logiciels requis	3
1.2 Compilation	3
2 Description des fonctionnalités du jeu	3
2.1 Launcher Unity	3
2.2 Écrans de jeu	3
2.3 Contrôles de jeu	5
2.4 Séquences enregistrées	5
3 Structure générale du projet	5
3.1 Contenu d'une scène de jeu	5
3.2 Diagramme de classes	6
4 Ajout d'une figure	6
4.1 Ajout de figure à FigureManager	7
4.2 Ajout de figure : automate	7
4.2.1 Création de nouvel Automate	7
4.2.2 Ajout à AutomataDetector	8
4.3 Ajout d'une figure : \$P	8
5 Comparaison des algorithmes de détection	8
5.1 Qualité de détection	8
5.2 Complexité des algorithmes	8
5.3 Facilité d'utilisation	9

Introduction

Le présent document a un double objectif. Tout d'abord, il s'agit de décrire comment utiliser le projet et son code. De plus, il est décrit ici la structure globale du code, et la façon dont il peut être maintenu / réutilisé.

Ce document n'est **pas** une documentation détaillé du code. Cette dernière est fournie avec le code source, dans le répertoire *Documentation*, elle se génère avec l'utilitaire Doxygen.

1 Ouverture du projet

1.1 Logiciels requis

Le présent projet a été développé à l'aide du moteur de jeu Unity dans sa version 2018.3.8f1. Par conséquent, il est nécessaire de disposer d'une version égale ou supérieur à celle-ci pour pouvoir ouvrir le projet sans problème. Pour les assets 3D, ceux-ci sont éditables avec le logiciel Blender.

Il est important de noter que le projet dans sa version exécutable ne nécessite aucune dépendance particulière.

1.2 Compilation

Pour créer un exécutable du projet, il est nécessaire de passer par l'interface de *Build* d'Unity. L'ensemble des paramètres de *build* ayant déjà été configurés au préalable, il n'est normalement pas nécessaire d'effectuer de manipulations supplémentaires.

Soyez néanmoins conscient du fait que ce projet utilise un certain nombre de fonctionnalités fournies par les *runtimes* dotNet4, assurez vous donc que celles-ci sont bien sélectionnées dans *Edit -> Project Settings* (cf. Figure 1).

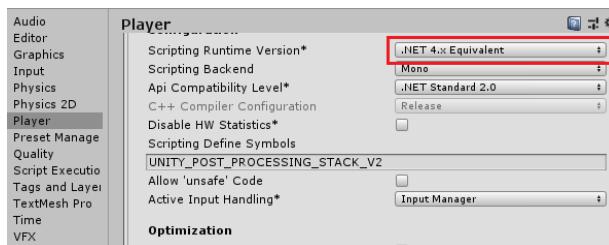


FIGURE 1 – Option de sélection des runtimes dotNet

A noter également que la phase de compilation n'est pas nécessaire pour essayer le projet : celui-ci peut être directement testé depuis l'éditeur d'Unity.

2 Description des fonctionnalités du jeu

2.1 Launcher Unity

Lors du lancement du jeu s'affiche tout d'abord le launcher Unity. Celui-ci permet de régler d'une part la qualité graphique du jeu, mais surtout de personnaliser les commandes selon la convenance du joueur, comme illustré en Figure 2. À noter que les commandes ne sont pas modifiables une fois le jeu lancé : il faudra quitter et relancer le jeu.

2.2 Écrans de jeu

Le jeu dispose de 3 écrans principaux : un menu principal, un menu d'options, et un niveau de jeu. Le menu principal est le premier écran s'affichant en jeu, et permet d'accéder au niveau, comme au menu d'options.

Le menu d'options permet de modifier divers coefficients physique de l'avion : puissance du moteur, intensité de la portance, coefficient de frottement... Il est également possible de choisir depuis ce menu l'algorithme de détection utilisé en jeu, et de changer la résolution du jeu.

Enfin, en cliquant sur "Jouer" depuis le menu principal, on accède au jeu en lui même.

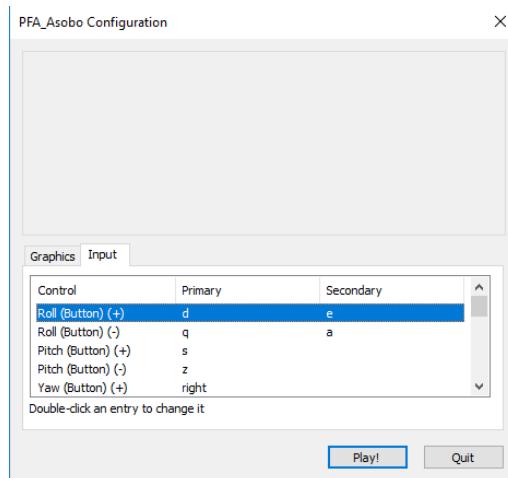


FIGURE 2 – Launcher Unity avec éditeur de touches



FIGURE 3 – Menu d'options en jeu

2.3 Contrôles de jeu

La liste des contrôles est aussi accessible depuis le menu principal :

- **ECHAP** fait apparaître le menu principal qui permet de relancer la scène de jeu et d'accéder aux options de jeu
- **R** relance la scène
- **P** permet d'alterner entre les algorithmes de reconnaissance (\$P et automates)
- **touches directionnelles** et **Z,Q,S,D** contrôles de l'avion.

Certains contrôleurs de jeux, tels que les manettes Xbox, sont également supportés.

2.4 Séquences enregistrées

Il est possible de rejouer des séquences pré-enregistrées en cours de jeu : le joueur n'a ainsi plus besoin de toucher les commandes, la séquence choisie se répétant en boucle. Actuellement, trois figures sont disponibles en tant que séquences enregistrées : le *Loop*, le *Roll* et le *Cuban Eight*.

Il est également possible d'enregistrer ses séquences de jeu. Celles-ci sont placées dans des fichiers .csv dans le dossier parent du jeu. Ces fichiers décrivent l'entièreté des positions de l'avion au cours de l'enregistrement, ainsi que les *inputs* utilisateur. La procédure pour ajouter ses propres séquences rejouables sera décrite plus loin.

Les commandes associées à ces fonctionnalités sont :

- **Numpad** – Sélectionne la séquence à rejouer
- **Numpad Entrée** Lance la séquence sélectionnée
- **Numpad + (appui long)** Démarré/Interrompt l'enregistrement

Ajouter ses propres séquences La classe responsable des séquences enregistrées est **DummyPlayer**.

Pour ajouter une nouvelle séquence enregistrer, il faut donc :

- Disposer d'un fichier .csv à 5 colonnes décrivant les inputs successifs à simuler (dans l'ordre : *Temps, Accélération, Roll, Pitch et Yaw*).
- Ajouter le nom de la figure dans `private string[] nameModes = "Aucun", "Loop", "Roll", "Cuban8" ;`
- Rajouter dans la méthode *Update* de la classe la lecture du fichier correspondant, par un appel à *replayFromFile*

La séquence sera dès lors sélectionnable depuis le jeu pour être rejouée.

3 Structure générale du projet

3.1 Contenu d'une scène de jeu

Une scène de jeu est constitué d'au moins 3 objets principaux : - un terrain de jeu - l'avion contrôlé par le joueur - la caméra suivant l'avion

Des *prefabs* ont été créés pour faciliter la création d'un niveau, et sont disponibles dans *Assets/Prefs*. Ainsi, le prefab **StartLinePrefab** regroupe l'avion, la caméra, et différents scripts permettant notamment de redémarrer le niveau : il suffit donc de le glisser dans une scène et de rajouter un simple terrain pour obtenir une base de niveau fonctionnel.

Voici la liste des prefabs disponibles :

- **DummyPlayer** rejoue des séquences enregistrées
- **Explosion** prefab instancié par l'avion lors d'un impact avec le sol
- **PlaneCameraPrefab** caméra suivant sa cible avec un retard
- **PlanePrefab** avion contrôlé par le joueur
- **PlaneTracker** enregistre la trajectoire de l'avion dans un fichier csv
- **Rampe** simple rampe de décollage
- **StartLinePrefab** contient les bases d'un niveau
- **TrailSmokePrefab** particules traçant la trajectoire de l'avion
- **UI Dial** affichage d'informations sur l'avion

Il est important de noter que certains GameObjects doivent être liés entre eux. Ainsi, la caméra, le DummyPlayer et le PlanerTracker doivent être liés à l'avion, et l'avion doit posséder un lien vers des objets Text pour afficher les figures réalisées et l'algorithme utilisé.

3.2 Diagramme de classes

La Figure 4 présente les classes importantes du projet et leurs relations.

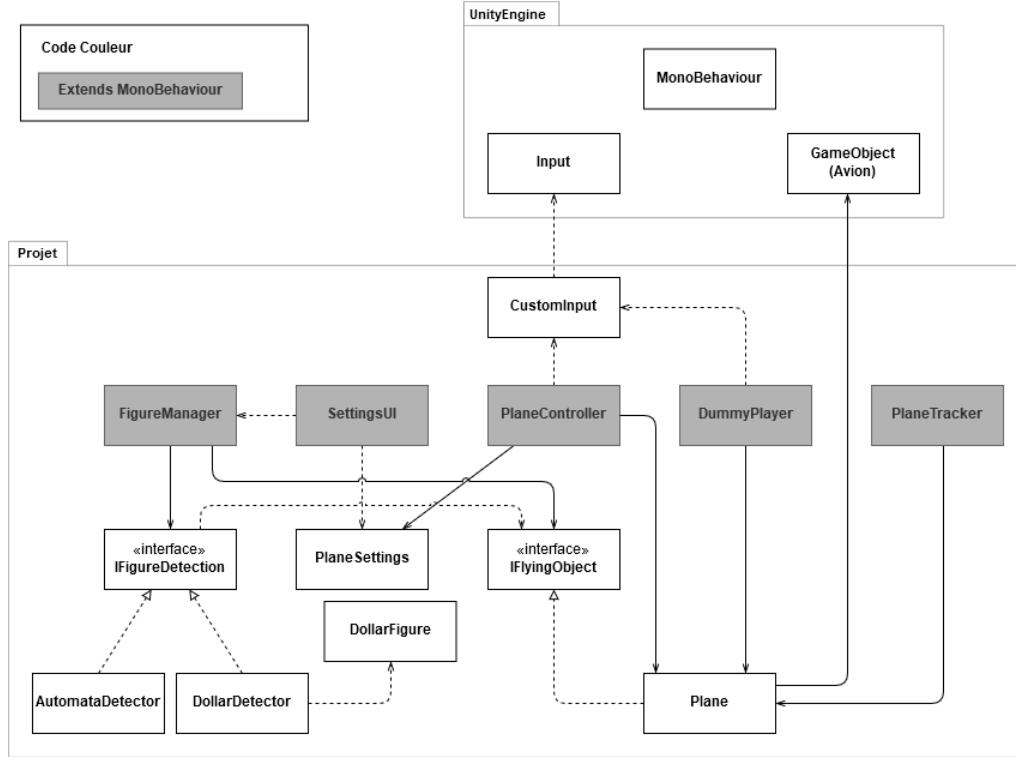


FIGURE 4 – Diagramme de classes du projet

Ce diagramme montre en particulier que les algorithmes de détection sont indépendants d'Unity : ils n'utilisent qu'un *IFlyingObject*, qui dans notre cas a été implémenté à l'aide du Gameobjet de l'avion dans le jeu.

4 Ajout d'une figure

Toutes les classes sont dans le dossier Assets/Scripts.

Le principe du projet est l'implémentation de détecteurs de figures, il est possible d'ajouter de nouvelles figures au jeu, il faut en revanche respecter certaines étapes pour qu'elles soient prises en compte par les détections :

1. ajouter un nouveau type de figure à la classe **FigureManager**
2. dans le cas d'un automate :
 - (a) créer une nouvelle classe d'automate
 - (b) ajouter l'automate au gestionnaire **AutomataDetector**
3. dans le cas de l'algorithme \$P :
 - (a) créer un fichier csv contenant la trajectoire de référence (trajectoire parfaite)
 - (b) charger la trajectoire à l'initialisation du détecteur
 - (c) ajouter l'analyse des résultats pour cette figure

Il n'est pas nécessaire d'ajouter la nouvelle figure aux deux algorithmes pour que le programme fonctionne.

4.1 Ajout de figure à FigureManager

Création du type de figure Il faut dans un premier temps déclarer le type de la figure s'il n'existe pas déjà, pour cela, on modifie l'énumération `figure_id` (fichier `TrajectoryStruct`) :

```
public enum figure_id {LOOP, ROLL, CUBANEIGHT, CUSTOMFIGURE};
```

Modification de FigureManager on doit modifier les attributs suivants de FigureManager :

```
private string[] _figureName = { "LOOP", "ROLL", "CUBANEIGHT", "CUSTOM" };  
private int[] _figurePoint = { 20, 10, 50, 1 };
```

Le premier associe un nom à la figure pour l'afficher, le second paramètre correspond au score associé. Ici, le constructeur redéfinit les attributs, il faut donc les modifier dans le constructeur de la classe FigureManager.

Une fois la figure définie, on peut s'intéresser à sa détection, qui va différer selon l'algorithme utilisé.

4.2 Ajout de figure : automate

Comme dit précédemment, la création d'un automate détecteur de figure passe par deux étapes, la création d'une nouvelle classe d'automate puis l'ajout de cette nouvelle classe au gestionnaire des automates.

4.2.1 Crédation de nouvel Automate

Les automates doivent implémenter l'interface `IFigureAutomata`, il est possible de créer son propre automate indépendant ou bien utiliser les classes abstraites préexistantes :

- **SimpleAutomata** (recommandé) : dispose d'états simples qui représentent les orientations de l'avion, à utiliser pour construire des figures par morceaux, exemple : un looping suivi d'un quart de Aileron Roll.
- **AbstractComposedFigureAutomata** (non testé) : utile si on veut que des sous-figures s'exécutent pendant une figure principale, exemple : Barrel Roll composé d'un Aileron Roll dans un Looping. Nécessite des automates déjà construits.
- **AbstractSequentialFigureAutomata** (non testé) : utile pour effectuer des figures à la suite, exemple : deux Looping à la suite. Nécessite des automates déjà construits.

Utilisation de SimpleAutomata :

1. Créer une classe héritant de SimpleAutomata
2. Implémenter le constructeur, `getFigureId()`, `getName()` et `calculateState()` de la même manière que dans LoopingAutomata (par exemple).
3. Détails sur CalculateState :
 - Faire appel à `init` pour tout initialiser, puis faire un appel à `isValid()` pour vérifier que nous ne sommes pas dans un état final
 - Ajouter les différentes fonctions de vérifications si voulu(`checkTime`, `checkForward`, `checkAltitude`).
 - Créer l'automate : ajouter dans la tableau `figure[]` une séquence de quart de figures (Il faut modifier le nombre d'états dans le constructeur, paramètre `n`)
 - Faire appel à `process()` pour faire les changements éventuels d'états
 - Revérifier si l'on est dans un état final

Le fichier `CustomAutomata` est présent pour essayer directement une figure personnalisable. Pour créer un nouvel état à détecter (état Q1Chandelle par exemple), il faut le faire dans la classe abstraite `SimpleAutomata`.

4.2.2 Ajout à AutomataDetector

Une fois le nouvel automate créé, il faut l'ajouter au constructeur d'AutomataDetector, la ligne sera de la forme :

```
_myAutomatas.Add(new MyAutomata());
```

Si l'automate est bien construit et que sa figure est reconnue par FigureManager, le jeu devrait afficher le résultat quand on exécute la figure correctement.

4.3 Ajout d'une figure : \$P

Il est possible d'ajouter une figure à la liste des reconnaissances de \$P. Pour cela, il est nécessaire de disposer d'un fichier contenant la trajectoire de référence au format csv. Ce fichier peut être enregistré directement depuis le jeu (c.f. partie 2.4). Il faut ensuite modifier quelques fichiers :

- **DollarFigure.cs** : Il faut rajouter un tableau de chaînes de caractère donnant le nom des courbes de la nouvelle figure. Ce nom peut être commun à plusieurs figures et apparaître plusieurs fois dans le tableau. Ce nom sert à distinguer les courbes lors de la reconnaissance. Si deux courbes au même indice ont une allure similaire, il vaut mieux leur donner le même nom pour éviter les confusions de l'algorithme.
- **FigureLoaderP.cs** : Il faut rajouter, dans la fonction `LoadFigures()`, un appel à la fonction `Load()` avec, en paramètre, le fichier à charger et le tableau contenant le nom des courbes précédemment rajouté.
- **DollarDetector.cs** : Il faut écrire le code suivant dans la fonction `detection` en remplaçant `<tableau de nom des figures>` et `<id figure>` par les bonnes valeurs :

```
if (AnalyseResults(results, <tableau de nom des courbes>)) {  
    figures.Add(new Figure(<id figure>, 1f));  
    ClearLists();  
}
```

5 Comparaison des algorithmes de détection

5.1 Qualité de détection

L'automate reconnaît les figures simples de manière quasi-systématique. Le Cuban eight est parfois reconnu sans que le joueur ne le veuille, cela est dû au fait qu'elle est composée de rotations simples de l'avion, et donc un joueur voulant faire deux looping en s'inclinant peut déclencher la reconnaissance du Cuban eight. \$P reconnaît le Loop et le Roll mais cette détection se fait souvent trop tôt, au milieu de la figure. On peut donc faire une moitié de Roll qui est reconnue comme un Roll complet. Un autre problème est que le Cuban eight n'est pas reconnu, car cette figure nécessite un grand nombre de points (> 1000) qui ne rentre pas dans les tableaux de points que nous donnons en entrée à l'algorithme. Plus généralement, le problème majeur des automates est qu'il n'y a que deux issues possibles : si le joueur passe exactement par tous les états de la figure, il la réussit, sinon elle est ratée. Cela amène à des situations où, par exemple, le Loop n'est pas détecté car le joueur ne s'est pas suffisamment repositionné à la verticale. Les algorithmes \$ n'ont pas ce problème : comme on cherche à identifier la figure dans sa globalité, une anomalie locale aura peu d'impact sur la détection finale.

5.2 Complexité des algorithmes

La complexité des algorithmes est un point crucial dans notre cas, car la détection est censé se faire en temps réel : un algorithme trop coûteux entraînera une chute du nombre d'images par seconde, rendant le jeu difficilement jouable. La détection par automate est ici très efficace : sachant que la vérification d'une transition se fait en temps constant, la détection par automate dans sa complexité est de complexité linéaire en le nombre de figures détectables. Les algorithmes \$ sont, quant à eux, beaucoup moins performants. La complexité d'au pire $O(n^{3m})$, avec n le nombre de points d'échantillonnage et m le nombre de figures, de \$P en fait ainsi un algorithme extrêmement lourd en terme d'utilisation processeur.

5.3 Facilité d'utilisation

Le gros avantage de la méthode automate est sa facilité d'évolution. Comme vu précédemment, grâce à l'interface designer, il est très simple de rajouter et créer un nouvel automate pour une nouvelle figure. Pour \$P, la démarche est un peu plus complexe, mais reste aisée. Si l'on met de côté le problème de taille de fenêtre, ajouter une figure est très simple puisqu'il suffit de l'enregistrer en jeu, ce qui crée un fichier CSV que pourra utiliser l'algorithme en tant que figure de référence, et cela ne dépend pas de la complexité de la figure. Il faudra tout de même penser à ajouter, nommer chaque courbe composant la figure, pour que celles-ci puissent être identifiées.