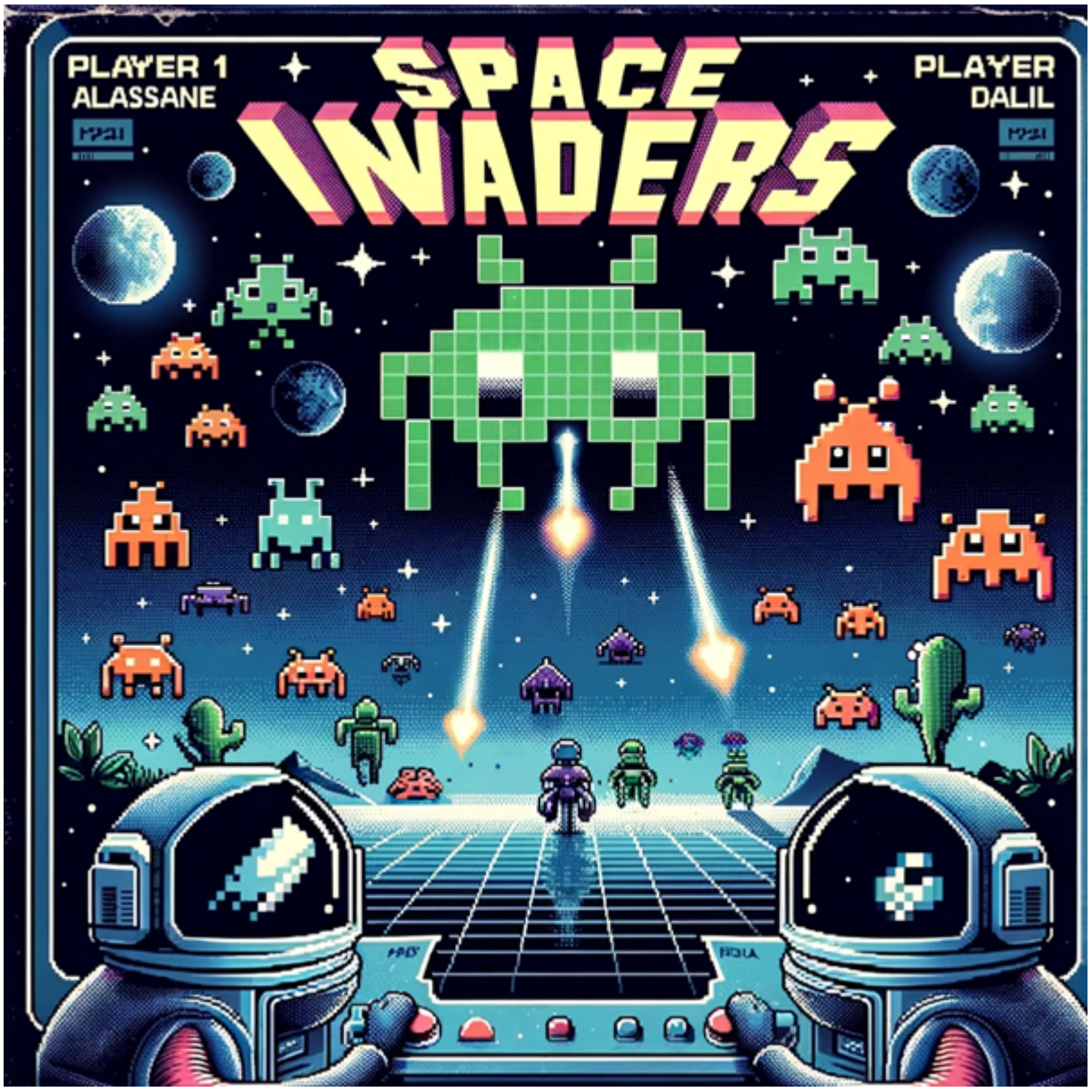
RAPPORT



Introduction

Les jeux vidéo occupent une place prépondérante dans l'univers du divertissement interactif, offrant aux joueurs une expérience immersive et captivante. Dans le cadre de notre projet de programmation, nous avons entrepris de donner vie à l'un des classiques intemporels du monde des jeux d'arcade : Space Invaders. Ce projet, développé en utilisant le langage de programmation C#, vise à non seulement recréer l'essence nostalgique du jeu original, mais surtout à explorer les aspects modernes de la programmation orientée objet.

Comme nous commençons la programmation orientée objet, nous avons fait le choix de passer par la piste verte et nous pensons qu'il faut d'abord passer par la base pour mieux apprendre de manière générale.

L'objectif principal de ce rapport est de présenter en détail le processus de conception, de développement et d'optimisation de notre version de Space Invaders. Nous aborderons les choix architecturaux, les structures de données, les algorithmes et les défis spécifiques rencontrés lors de la programmation du jeu. En outre, nous examinerons les fonctionnalités ajoutées pour améliorer l'expérience utilisateur et les enseignements tirés de ce projet en termes de développement logiciel.

(Dans le projet lorsqu'on parlera de d'instance d'une classe, ce sera de ce type : "*SpaceShip*")

Les collisions

Dans ce projet, la plupart des éléments du jeu sont des objets qui sont ajoutés dans une liste de l'instance unique du jeu. Dans le projet, les "*SpaceShip*" sont des vaisseaux qui ont la possibilité de lancer des "*Missile*". Et chaque "*Missile*" vérifie s'il est en collision avec un autre objet du jeu. L'objectif pour les collision était de d'abord vérifier si le "*Missile*" était dans le rectangle englobant de l'autre objet et d'ensuite vérifier si chaque pixel du "*Missiles*" était en collision avec un des pixels de l'autre objet. Et ce n'est qu'à partir de ce moment-là qu'il y a une "vraie" collision.

Pour vérifier que chaque pixel d'un "*Missile*" était en collision, il fallait faire un changement de repère. Il fallait passer du repère de la fenêtre graphique, à celui de l'objet qui est vérifié s'il est en collision avec le "*Missile*".

On avait eu un souci pour le changement de repère. Ce qu' avait oublié de prendre en compte, c'est que c'est le "*Missile*" qui passe du repère de la fenêtre à celui de l'objet. Et comme on teste chaque pixel du "*Missile*", il peut être à moitié en dehors de l'objet. Du coup on avait des valeurs négatives et pour la suite du programme certaines

méthodes ne prenait pas de valeurs négatives et le jeu crashait. Pour régler ce problème il fallait juste vérifier qu'on le prenne que les pixels qui sont présents dans l'objet en question. Voici ci-dessous la fonction qui vérifie ceci :

```
1 référence
private bool ChangementRepere(int i, int j, double missileX, double missileY, double objectX, double objectY)
{
    int missilePixelScreenX = (int)(missileX + i);
    int missilePixelScreenY = (int)(missileY + j);
    int missilePixelOtherX = (int)(missilePixelScreenX - objectX);
    int missilePixelOtherY = (int)(missilePixelScreenY - objectY);
    this.pixelOnObject = new Vecteur2D(missilePixelOtherX, missilePixelOtherY);
    if (missilePixelOtherX >= 0 && missilePixelOtherX < this.image.Width && missilePixelOtherY >= 0 && missilePixelOtherY < this.image.Height)
        return true;
    return false;
}
```

DLC : Boss

Nous avons créé une classe Boss qui dérive de la classe SpaceShip.
Pour l'implémentation d'un "*Boss*", nous avons choisi de le faire apparaître lorsque tous les ennemis de bases meurent. En ce qui concerne la programmation en C#, un dès problèmes que nous avons eu concernait l'apparition du boss. En fait, comme je l'ai dit précédemment, le boss apparaît dès qu'il y a plus aucun ennemis dans la liste d'ennemies du jeu et pour cela nous avons fait en sorte que le "*Boss*" s'ajoute à la liste d'ennemies. Mais lorsque le "*Boss*" meurt, la liste est à nouveau vide donc un nouveau "*Boss*" réapparaît. Pour régler le soucis, nous avons créé une variable de type booléen qui est à "true" si le "*Boss*" est déjà apparue sinon "false". Donc grâce à cela nous avons pu faire apparaître le "*Boss*" qu'une seule fois par partie.

Voici le code ci-dessous :

```
if (this.state == GameState.Play)
{
    if (this.enemies.IsAlive() && bossAlleadyAdded)
        PlayMusic(SpaceInvaders.Properties.Resources.bossMusic, 4);
    else PlayMusic(SpaceInvaders.Properties.Resources.playSong1, 1);
    foreach (GameObject gameObject in gameObjects) {
        gameObject.Update(this, deltaT);
    }
}
```

DLC : Couleur

Pour l'implémentation des couleurs, nous avons créé une fonction dans la classe SimpleObjet. La voici ci-dessous :

```
protected ColorMatrix TheColorObject(Color couleur)
{
    float r = couleur.R / 255f;
    float g = couleur.G / 255f;
    float b = couleur.B / 255f;

    ColorMatrix colorMatrix = new ColorMatrix(new float[][]
    {
        new float[] {0, 0, 0, 0, 0},
        new float[] {0, 0, 0, 0, 0},
        new float[] {0, 0, 0, 0, 0},
        new float[] {0, 0, 0, 1, 0},
        new float[] {r, g, b, 0, 1}
    });
    return colorMatrix;
}
```

Nous avons choisi ceci car nous voulions nous donner la possibilité de créer des objets avec ou sans une couleur directement lors de sa construction. C'est pourquoi nous avons aussi dû créer plusieurs constructeurs. Les voici :

```
1 référence
public SimpleObject(Side side, ColorMatrix colorMatrix) :base(side)
{
    this.colorMatrix = colorMatrix;
}
2 références
public SimpleObject(Side side) : base(side)
{
    colorMatrix = null;
}
```

Nous devions donc créer des constructeurs chinés dans les classes dérivés. Notamment dans la classe SpaceShip. Voici le code :

```
public SpaceShip(int lives, double viewWidth, double viewHeight, Bitmap image, Side side) : base(side)
{
    this.position = new Vecteur2D(viewWidth, viewHeight);
    this.lives = lives;
    this.image = image;
    Side = side;
}
3 références
public SpaceShip(int lives, double viewWidth, double viewHeight, Bitmap image, Side side, ColorMatrix colorMatrix) : this(lives, viewWidth, viewHeight, image, side)
{
    this.colorMatrix = colorMatrix;
}
```

Ces constructeurs chaînes nous évitent une duplications du code et nous laissent plus de choix lors de l'instanciation des objets.

DLC : Sound Effect

Pour l'implémentation de sound effect dans le projet, nous avons décidé de créer un classe AudioSfx qui utilise un package externe "NAudio". En fait, la problématique

que nous avons eu est que lorsque le jeu doit jouer plusieurs sons en même temps, cela ne marche pas. En effet, l'utilisation simple de soundplayer ne permettait pas de jouer plusieurs sons en même temps, ce qui faisait que chaque nouveau son joué écrasait le précédent. Et ce n'était pas ce qu'on voulait. Nous avons donc dû trouver une solution et c'est l'utilisation du package NAudio qui est la plus revenue dans nos recherches.

Qualité du code

Concernant la qualité du code, nous avons fait très attention à la duplication de même "bout de code" et à la fragmentation d'une grande fonction en plusieurs petites car c'est ce qui permet une meilleurs maintenabilité du code.
Voici un exemple :

```
public override void Update(Game gameInstance, double deltaT)
{
    CheckPlayerCollision(gameInstance);
    List<SpaceShip> shipsToRemove = new List<SpaceShip>();

    if (goingRight)
    {
        MoveShips(deltaT, 1, gameInstance);
        if (CheckEdgeCollision(gameInstance))
        {
            HandleDirectionChange();
            MoveDown();
        }
    }
    else if (!goingRight)
    {
        MoveShips(deltaT, -1, gameInstance);
        if (CheckEdgeCollision(gameInstance))
        {
            HandleDirectionChange();
            MoveDown();
        }
    }

    RemoveDeadShips(shipsToRemove);
    UpdateSize();
}
```

La méthode Update à l'origine faisait plus de 100 lignes de code. Et ce n'est pas du tout facile de maintenir ce code. Dans l'image ci-dessus, toutes ces petites méthodes permettent, s'il on débug par exemple, de mieux savoir d'où viennent les soucis ou tout simplement permettent au programmeur ce qu'il se passé.

Conclusion

En conclusion de ce projet captivant de programmation de Space Invaders en C#, nous pouvons réfléchir aux nombreux défis relevés, aux leçons apprises et aux réalisations accomplies au cours de cette aventure logicielle. La recréation de ce classique du jeu d'arcade a été bien plus qu'une simple expérience de codage ; elle a été une plongée profonde dans le monde complexe du développement de jeux vidéo, mettant en avant notre capacité à résoudre des problèmes, à concevoir des solutions efficaces et à repousser les limites de notre compréhension de la programmation orientée objet.