Manuel Technique

Travail Pratique Individuel
MiFiSy

CFPT Informatique

Yoann Meier

8 mai 2024

Table des matières

Introduction

Ce document est un rapport présentant la conception du projet MiFiSy (MIDI Firework Symphony).
Ce projet est réalisé dans le cadre du projet de fin de formation au CFPT Informatique dans la formation *Développement d'applications*.
Il s'agit du *Travail Pratique individuel* (TPI).

MiFiSy est un projet monogame permettant de créer des feux d'artifice virtuel à l'aide d'une guitare MIDI.
Une musique de fond peut être ajoutée.
La séquence d'effet de feu d'artifice peut être sauvegardée au format XML et rejouée dans l'application.

Résumé du cahier des charges

Description de l'application

- Page d'accueil
  - Liste des musiques disponibles
  - Liste des séquences sauvegardées
  - Bouton pour aller dans le mode libre (musique optionnelle)
  - Bouton pour aller dans le mode replay (choix d'une séquence obligatoire)

- Page de jeu, mode libre
  - 5 mortiers sont disposés uniformément au bas de l'écran avec un angle entre -10 et 10 degrés
  - Bouton de retour à l'accueil
  - Bouton de sauvegarde de la séquence de feu d'artifice au format XML
  - Jouer une corde déclenche un effet de feu d'artifice, la vélocité influe sur la vitesse ou la taille du feu d'artifice
    - Effet de comète :
      Une traînée lumineuse propulsée par un des 5 mortiers choisis aléatoirement.
      La direction de la traînée est en fonction de l'angle du mortier.

    - Effet de pluie de particules :
      Des particules sont générés sur un point en haut de l'écran, celles-ci tombent avec l'effet de la gravité et disparaissent après un certain temps ou lorsqu'elles atteignent le bas de l'écran.

- Page de jeu, mode replay
  - La séquence jouée choisit dans l'accueil est jouée
  - Les mortiers possèdent le même angle que dans la séquence
  - Bouton de retour à l'accueil
  - Les informations de la séquence (auteur, date, nom de la séquence) sont affichées

Élément bonus

- Feu d'artifice
  - Étoiles simples (ou points)
    Un point lumineux qui apparaît et disparaît.

  - Feux d'artifice à explosion simple (pivoine)
    Une explosion basique qui se propage de manière uniforme dans toutes les directions.

  - Fontaines (pot au feu)
    Des particules jaillissent vers le haut avant de retomber, comme une fontaine.

Framework/librairie

Dans ce projet, j'utilise le framework Monogame pour développer mon application.
Monogame est un framework C# permettant de faire des jeux.

J'utilise également la librairie NAudio permettant de récupérer les cordes joué par une guitare MIDI dans le format MIDI.

Outils/logiciels utilisés

- Ordinateur Windows 10
- C# - Visual Studio 2022
- Framework Monogame
- GitHub
- Google Drive
- LaTeX
- Suite office

Livrables

- Manuel technique
- Manuel utilisateur
- Journal de bord
- Rapport TPI
- Le projet

Méthodologie

Méthodes en 6 étapes

Pour assurer le bon déroulement de mon projet, j'ai utilisé une méthodologie de travail afin d'être organisé et efficace.
Après avoir examiné différentes méthodes lors de ma formation, j'ai opté pour la méthode en 6 étapes.
Je l'ai choisie, car, parmi les différentes méthodes étudiées, elle s'est révélé être la plus adaptée pour un travail individuel à court terme.

- **S'informer**
  Au début du projet, j'ai analysé le cahier des charges afin de comprendre toutes les fonctionnalités à réaliser.
  J'ai également demandé des précisions à mon formateur sur certains points que je n'ai pas compris.
- **Planifier**
  Une fois le cahier des charges compris, j'ai préparé le planning de mon travail en découpant le travail par tâches avec une durée prévisionnelle.
  Voir : Planning prévisionnel
- **Décider**
  Dans la phase de décision, j'ai décidé l'ordre de réalisation de toutes les tâches en fonction de l'importance de celle-ci sur le projet.
- **Réaliser**
  Cette étape est cruciale dans mon projet, car elle implique la concrétisation de mon travail en suivant la planification établie précédemment. Grâce à une bonne compréhension des tâches à accomplir, j'ai pu effectuer mon travail dans des conditions optimales.
- **Contrôler**
  Cette étape est importante, car elle consiste à contrôler le bon avancement du projet en s'assurant que les objectifs du travail sont atteints. À chaque tâche terminée, j'ai effectué des tests pour vérifier le bon fonctionnement de ces tâches.
  Voir : Rapport de test
- **Évaluer**
  Pour finir, j'ai évalué tous les résultats obtenus et j'ai également réfléchi aux différents moyens d'améliorer mon travail, que ce soit pour ajouter de nouvelles fonctionnalités ou améliorer mon code.
  Voir : Amélioration possible

Analyse fonctionnelle

Vue Accueil

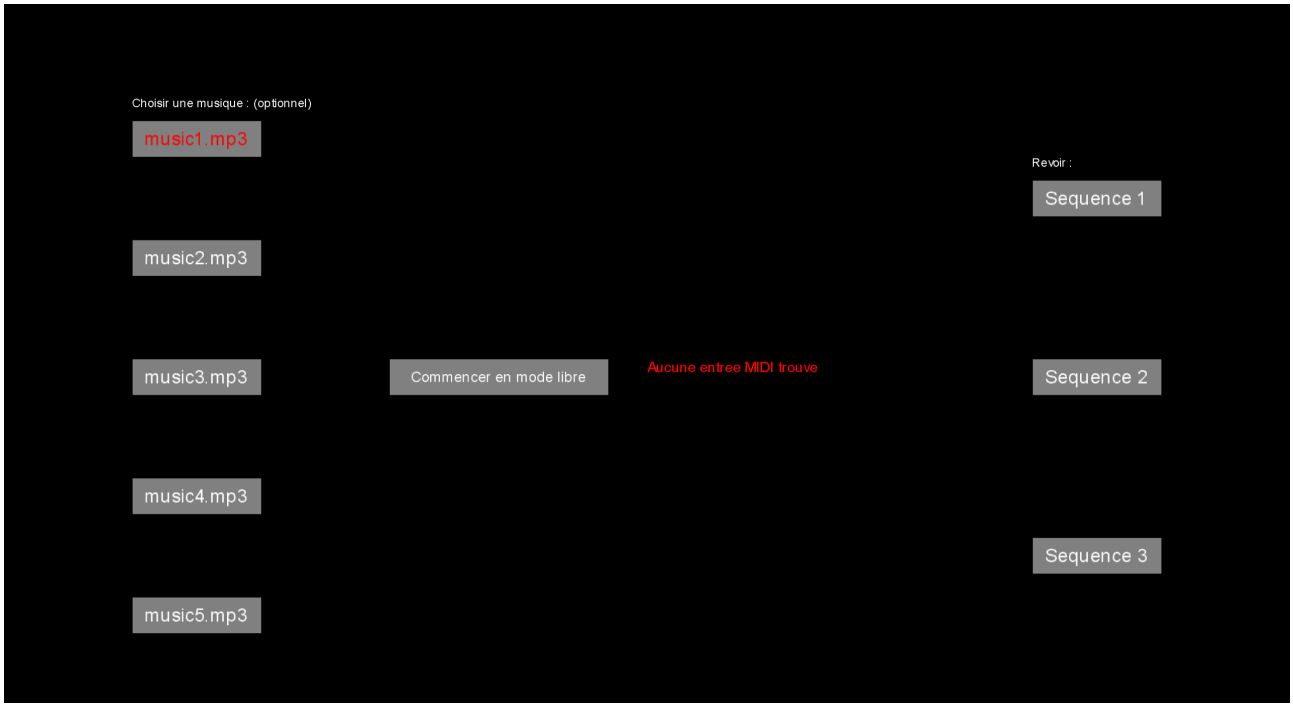


FIGURE 1 – page de démarrage de l'application

Dans cette page, un message d'erreur apparaît au milieu de l'écran si aucune entrée MIDI n'est détectée : "***Aucune entrée MIDI trouvée***".

Si l'utilisateur souhaite une musique de fond pendant le mode libre, il peut appuyer sur l'une des musiques affichée à gauche. Chaque musique provient d'un dossier qui peut être défini dans le fichier de configuration. Lorsqu'une musique est sélectionnée, sa couleur deviendra rouge (music1.mp3 sur l'exemple).

L'utilisateur peut appuyer sur le bouton "***Commencer en mode libre***", ce qui le redirigera vers la page de jeu en mode libre.

À droite de l'écran, les différentes séquences précédemment enregistrées sont affichées.
Lors d'un clique sur l'une d'entre elle, l'utilisateur est redirigé sur la page de jeu en mode replay.

Vue Jeu

Mode libre

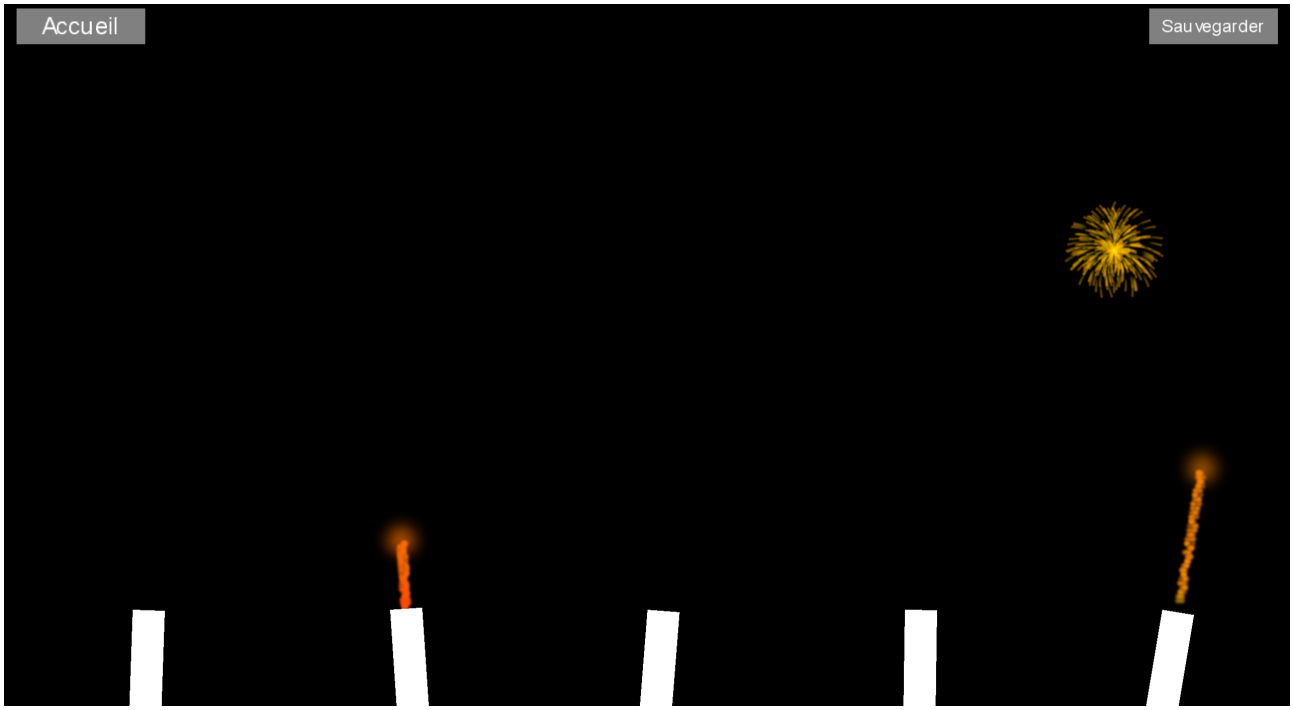


FIGURE 2 – Mode de jeu libre

Dans cette page, un bouton "***Accueil***" permet de retourner à l'accueil, ce qui remet à 0 la séquence de feu d'artifice.

Plusieurs mortiers, définis dans le fichier de configuration (position x et y, taille en hauteur et en largeur et l'angle d'inclinaison) sont affichés.
Si aucun mortier n'est défini dans le fichier de configuration, 5 mortiers sont disposés uniformément en bas de l'écran avec la même taille et un angle entre -10 et 10 degrés.

Si l'utilisateur (connecté à une guitare MIDI) joue la première corde (tout en bas), la comète sera lancée aléatoirement de l'un des mortier représenté par un rectangle blanc, avec la même direction.
La vitesse de déplacement dépend de la vélocité à laquelle la corde est jouée.

Si l'utilisateur joue la deuxième corde, la pluie de particules est créée sur un point aléatoire du haut de l'écran.
La durée de vie du feu d'artifice dépend de la vélocité à laquelle la corde est jouée.

Le bouton en haut à droite permet de sauvegarder la séquence de feu d'artifice crée au format XML.
Un message de confirmation de sauvegarde apparaît brièvement sur l'écran.

Mode replay

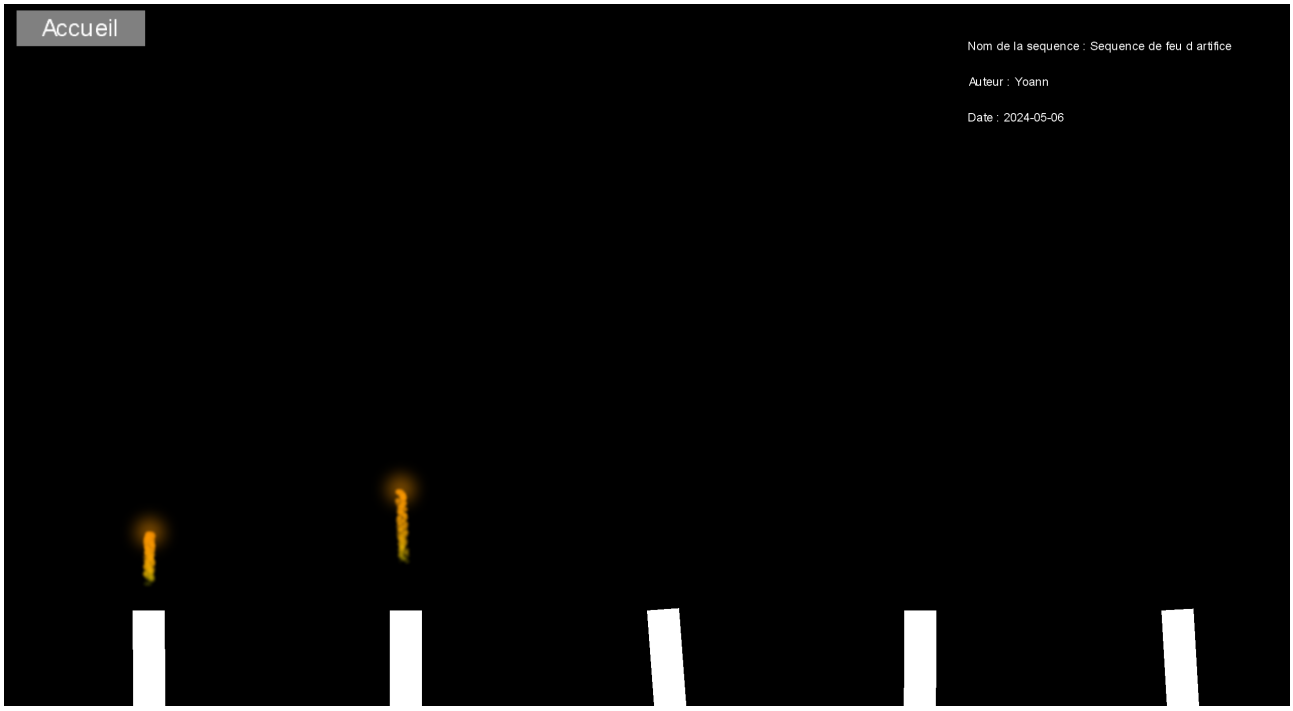
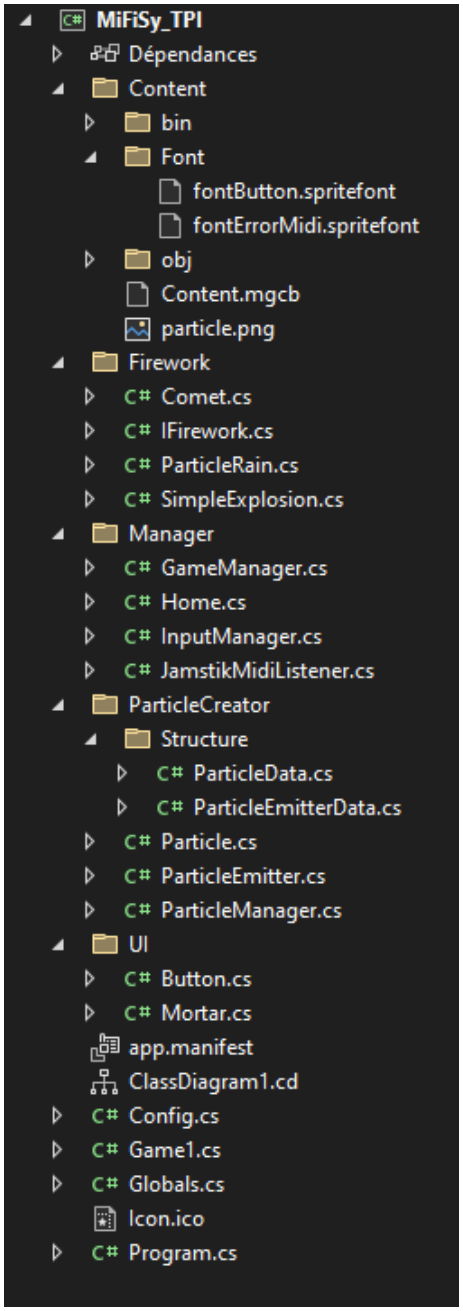


FIGURE 3 – Mode de jeu replay

Dans cette page, une séquence sauvegardée est jouée, la musique, l'image de fond et la position des mortiers est identique, les informations de la séquence sont écrits en haut de l'écran.
A la fin de la séquence, un message ("***Sauvegarde effectue***") apparaît pour notifier la fin de l'enregistrement à l'utilisateur.
Le bouton "***Accueil***" permet de revenir à l'accueil.

Analyse organique

Structure des fichiers



- **Content**
  Contient les polices de texte dans le fichier "**Font**" et l'image des particules : "**particle.png**"
- **Firework**
  Contient tous les éléments concernant les feux d'artifices.
- **Manager**
  Contient toutes les classes de gestion : des vues (Accueil et jeu) ainsi que les touches (Souris/clavier et guitare MIDI)
- **ParticleCreator**
  Contient toutes les classes et structures concernant la création, l'émission et la gestion des particules.
- **UI**
  Contient les classes concernant l'interface (boutons, mortier)

Diagramme de classe

Fichier de configuration

Dans ce projet, j'ai placé les constantes comme la vitesse par défaut d'une particule, le dossier des musiques, le dossier de sauvegarde des séquences de feux d'artifices dans un fichier xml qui se trouvent dans le même dossier que l'exécutable du projet : config.xml.

Voici un exemple du contenu du fichier :

```
<Config>
  <Author>Yoann</Author>
  <NameSequence>Sequence de feu d artifice</NameSequence>
  <PathMusic>music/</PathMusic>
  <PathImg></PathImg>
  <PathSaveSequence>save/</PathSaveSequence>
  <Mortar angle="10" height="0,15" width="0,025" positionX="0,1" positionY="0,85"/>
  <Mortar angle="10" height="0,15" width="0,025" positionX="0,3" positionY="0,85"/>
  <Mortar angle="10" height="0,15" width="0,025" positionX="0,5" positionY="0,85"/>
  <Mortar angle="10" height="0,15" width="0,025" positionX="0,7" positionY="0,85"/>
  <Mortar angle="10" height="0,15" width="0,025" positionX="0,9" positionY="0,85"/>
  <ColorStartParticleRain r="255" g="69" b="0"/>
  <ColorStartComet r="255" g="69" b="0"/>
  <ColorEndParticleRain r="255" g="255" b="0"/>
  <ColorEndComet r="255" g="255" b="0"/>
  <ParticleRain sizeParticle="10" nbParticle="360" lifeSpan="0,06" timeSpawn="0,04" ↩
      defaultSpeed="90"/>
  <Comet sizeMainParticle="120" sizeOtherParticle="20" defaultSpeed="7" ↩
      defaultLifespan="1,5" />
</Config>
```

Les explications de chaque valeurs se trouvent dans le ***manuel utilisateur***.

Description des classes et des méthodes importantes du projet

Config.cs

Cette classe sert à récupérer les données du fichier de configuration.
Elle contient des propriété static permettant seulement de récupérer une information du fichier, comme l'auteur des feux d'artifices par exemples.
Elle contient aussi un constructeur pour ouvrir et récupérer les données du fichier, ce qui évite d'ouvrir le fichier à chaque fois que l'on veut récupérer une de ses valeurs.

Globals.cs

Cette classe permet de récupérer des valeurs static qui sont nécessaires dans énormément de classes comme la hauteur et la largeur de l'écran, le SpriteBatch, une classe qui permet d'afficher des éléments, image ou texte, à l'écran et dont je ne possède qu'une seule instance (Détail sur le site de monogame : SpriteBatch)

Cette page contient aussi des propriétés static très peu utilisées, mais qui sont utilisées dans des classes très séparés et différentes comme par exemple de r instanciée la classe de l'accueil depuis la classe Bouton, lors du retour à l'accueil depuis le jeu.

Game1.cs

C'est la classe principale du projet qui est créé par défaut lorsqu'on crée un projet.
Elle hérite de la classe de Monogame Game et contient plusieurs méthodes essentielles au fonctionnement du projet comme **Initialize** et **LoadContent** au lancement du projet et surtout **Update**, une méthode qui est appelée de nombreuses fois chaque secondes et qui permet de mettre à jour les informations.
Finalement, il y a **Draw** qui permet d'afficher les éléments du jeu et qui lui aussi est appelé très souvent.
Plus d'informations ici.

Dans mon projet, toutes mes classes qui doivent mettre à jour ou afficher des éléments sont appelés dans l'Update et le Draw de Game1.cs.
Une enum permet de mettre à jour et d'afficher les éléments correspondants à la page actuelle : l'accueil ou le jeu.

ParticleData.cs

C'est une **struct** qui stock des informations concernant une particule : la texture, la couleur, la taille, la vitesse, l'angle...
Cela provient d'un tutoriel pour créer un système de particule.

ParticleEmitterData.cs

C'est une **struct** qui stock des informations permettant d'émettre des particules avec un ParticleData, l'interval d'émission, le nombre à émettre, un angle ou une vitesse aléatoire, etc.
Proviens également du tutoriel, mais que j'ai légèrement modifié.

Particle.cs

Cette classe permet de créer une particule, de lui donner une durée de vie, de se déplacer, de changer de couleur en fonction de la durée de vie.
Cette classe utilise le struct ParticleData.cs et vient du même tutoriel.

ParticleEmitter.cs

Cette classe émet des particules selon les informations du ParticleEmitterData.cs qu'il possède.
Les particules sont émises à une position défini dans les paramètres du constructeur.
Cette classe provient également du tutoriel de système de particules.

ParticleManager.cs

Cette classe gère les particules avec simplement une liste de particules et une liste de particuleEmitter, avec des méthodes static pour ajouter et supprimer ainsi qu'une méthode "Update" et "Draw" pour afficher et mettre à jour les particules et les émetteurs.
Viens également du tutoriel.

Button.cs

Cette classe permet simplement de créer un bouton un champ string ”action” permet de savoir quelle action le clique réalise avec par exemple ”goBack” pour revenir à l'accueil ou encore ”play” pour aller depuis l'accueil dans le jeu en mode libre.

La fonction ”SetTextPositionAndScale” permet de calculer la taille et la position du texte dans le rectangle en fonction du nombre de caractères récupérés avec la fonction ”MeasureString” de la classe SpriteFont pour récupérer la taille du texte en X et en Y.

SpriteFont est une classe pour charger une police de texte et l'utiliser pour afficher du texte (documentation).

```
72          public void SetTextPositionAndScale()
73          {
74              if (Globals.FontButton.MeasureString(_text).X != 0)
75              {
76                  // Calcul du facteur d'échelle pour le texte
77                  float scaleX = (Rectangle.Width * (1 - _padding)) / ↩
                        Globals.FontButton.MeasureString(_text).X;
78                  float scaleY = (Rectangle.Height * (1 - _padding)) / ↩
                        Globals.FontButton.MeasureString(_text).Y;
79                  _scale = Math.Min(scaleX, scaleY);
80              }
81              else
82              {
83                  _scale = 1;
84              }
85
86              _textPosition.X = Rectangle.X + (Rectangle.Width - ↩
                    Globals.FontButton.MeasureString(_text).X * _scale) / 2;
87              _textPosition.Y = Rectangle.Y + (Rectangle.Height - ↩
                    Globals.FontButton.MeasureString(_text).Y * _scale) / 2;
88          }
```

Mortar.cs

Cette classe permet simplement d'afficher un rectangle avec une rotation pour simuler un mortier qui tire des feux d'artifices.
J'ai créé cette classe pour la sauvegarde, car la position, la taille et l'angle des mortiers doit être identique dans le mode replay.

IFirework.cs

C'est une interface qui possède des propriétés obligatoires à chaque feu d'artifice : une position, une durée de vie, le moment dans le temps où il a été lancé (pour refaire la séquence au bon moment dans le replay) et une vitesse.
Mais le plus important est la fonction ”Update” qui permet de mettre à jour l'effet de particule.
Le but de cette interface est de n'avoir qu'une seule liste de feu d'artifice et non une seule liste par type de feu d'artifice, ce qui évite la redondance de code et simplifie le travail.

ParticleRain.cs

Cette classe permet de créer le feu d'artifice de la pluie de particules, elle implémente IFirework.cs.
Je vais expliquer comment cela fonctionne.

Tout d'abord, voici mon constructeur dans le mode libre (celui pour le mode replay est presque identique) :

```
41          public ParticleRain(float speed, float lifespan, float launchTime, float ↩
                distanceFromBorder = 100)
42          {
43              LaunchTime = launchTime;
44              Lifespan = lifespan;
45              StartSpeed = speed;
46              _nbParticle = Config.PARTICLE_RAIN_NB;
47              _colorStart = Config.COLOR_PARTICLE_RAIN_START;
48              _colorEnd = Config.COLOR_PARTICLE_RAIN_END;
49              _size = Config.PARTICLE_RAIN_SIZE;
50
51              _timerLife = 0;
52              _timerSpawn = 0;
53              // Position aléatoire du feu d'artifice sur la partie haute de l'écran
54              StartPosition = new Vector2(Globals.RandomFloat(distanceFromBorder, ↩
                    Globals.ScreenWidth - distanceFromBorder) / Globals.ScreenWidth, ↩
                    Globals.RandomFloat(distanceFromBorder, Globals.ScreenHeight / 2) / ↩
                    Globals.ScreenHeight);
55              _lstMainParticles = new List<Particle>();
56
57              for (int i = 0; i < _nbParticle; i++)
58              {
59                  float angle = 360 / _nbParticle * i;
60                  // Vitesse aléatoire entre 0 et le maximum
61                  float newSpeed = Globals.RandomFloat(0, speed);
62                  ParticleData particleData = new ParticleData()
63                  {
64                      angle = angle,
65                      speed = newSpeed,
66                      colorStart = _colorStart,
67                      colorEnd = _colorEnd,
68                      sizeStart = _size,
69                      sizeEnd = _size,
70                      lifespan = Lifespan,
71                  };
72                  Particle p = new Particle(StartPosition, particleData);
73                  _lstMainParticles.Add(p);
74                  ParticleManager.AddParticle(p);
75              }
76          }
```

Dans ce code, j'initialise les variables et choisis une position aléatoire.
Ensuite, je crée les particules qui seront en mouvement. Le nombre, la couleur et la taille sont définis dans le fichier de configuration.
La vitesse est aléatoire et les particules sont réparties sur 360 degrés.

Voici ma méthode ”Update” qui est appelée plusieurs fois par secondes :

```
126         public void Update()
127         {
128             _timerLife += Globals.TotalSeconds;
129             _timerSpawn += Globals.TotalSeconds;
130             // Supprime en fin de vie
131             if (_timerLife >= Lifespan)
132             {
133                 _lstMainParticles.Clear();
134             }
135 
136             if (_lstMainParticles.Count != 0)
137             {
138                 if (_timerSpawn >= Config.PARTICLE_RAIN_TIME_SPAWN)
139                 {
140                     // Ajoute une particule immobile sur chaque particule en mouvement
141                     for (int i = 0; i < _nbParticle; i++)
142                     {
143                         ParticleData particleData = new ParticleData()
144                         {
145                             angle = MathHelper.ToDegrees(_lstMainParticles[i].Data.angle),
146                             speed = 0,
147                             colorStart = _colorStart,
148                             colorEnd = _colorEnd,
149                             sizeStart = _size,
150                             sizeEnd = _size,
151                             lifespan = Lifespan - _timerLife,
152                         };
153                         Particle p = new Particle(_lstMainParticles[i].Position, ↩
                                particleData);
154                         ParticleManager.AddParticle(p);
155                     }
156                     _timerSpawn = 0;
157                 }
158 
159                 // Si un tiers du temps total est passé, les particules en movement tombent
160                 if (_timerLife >= Lifespan / 3)
161                 {
162                     foreach (Particle item in _lstMainParticles)
163                     {
164                         ParticleData data = item.Data;
165                         int angleAdd = MathHelper.ToDegrees(data.angle) < 180 ? 1 : -1;
166                         data.angle = MathHelper.ToDegrees(data.angle) + angleAdd;
167                         item.Data = data;
168                         item.SetAngleAndDirection();
169                     }
170                 }
171             }
172         }
```

Tout d’abord, j’augmente les compteurs et je vide les listes de particules si la durée de vie est terminée.
Ensuite, je vais faire apparaître une particule immobile à la même position que chaque particule en mouvement avec un délai défini dans le fichier de configuration et une durée de vie en fonction du temps passé pour que la fin du feu d’artifice soit en même temps.

Finalement, si le tiers de la durée de vie est atteint, je vais déplacer l’angle des particules en mouvement d’un degré pour simuler la gravité.
Les prochaines particules seront donc décalées d’un degré par rapport au précédent et cela crée donc un effet de chute des particules et de mouvements.

Comet.cs

Cette classe crée une comète.
Il y a une particule principale qui part et un émetteur qui suit la particule principale.
La position et l’angle viennent d’un des mortiers, le choix et l’aléatoire sont dans le GameManager.cs.

JamstikMidiListener.cs

Cette classe gère la connexion avec la guitare MIDI.
Dans le constructeur, on parcourt les connexions et se connecter à ”Jamstick”, le nom de la connexion de la guitare. Ensuite, à chaque évènement déclencher, une méthode reçois ces évènements, regarde si c'est une note, si elle vient de sonner, récupère la corde et appelle une fonction pour créer le feu d'artifice lié à cette corde.
Si aucune connexion n'est trouvée, une fonction affiche un message d'erreur dans l'accueil.

InputManager

Cette classe permet simplement de gérer les cliques avec la souris pour les boutons et pouvoir récupéré l'état du clique partout avec une propriété static.

Home.cs

Cette classe contient l'accueil de l'application.
La classe Home gère l'interface utilisateur de l'accueil de l'application, choisir une musique, aller en mode libre et charger des séquences enregistrées.
Les musiques sont dans une liste de boutons. Je peux ensuite récupérer la musique dans le jeu libre avec le nom du bouton.
Alors que pour les enregistrements, j'utilise un dictionnaire avec comme clé un string qui contient le nom du fichier (et qui est donc forcément unique) et comme valeur un bouton avec comme texte le nom de la séquence récupéré dans le fichier de la séquence.
J'utilise ce dictionnaire pour pouvoir transmettre le nom du fichier dans le GameManager.cs et récupérer le contenu.
En effet, le nom de la séquence peut être le même pour plusieurs séquences, car il est défini lors de la sauvegarde et récupéré du fichier de configuration.
Seules les 10 premières musiques et les 10 premières séquences sont chargées.
Un message d'erreur est affiché au milieu de l'écran si la guitare MIDI n'a pas été trouvée au lancement.

GameManager.cs

Cette classe gère le jeu, le mode libre comme le mode replay.
Cette classe fais donc l'affichage des deux modes et charge les musiques et l'image de fond si spécifiée dans le fichier de configuration pour le mode libre ou dans la sauvegarde de la séquence pour le mode replay.

Cette classe contient les méthodes de création de feu d'artifice utilisées dans JamstikMidiListener.cs : ”**CreateComete**” et ”**CreateParticleRain**” qui ajoute le nouveau feu d'artifice dans une liste d'IFirework en prenant en compte une vélocité transmise en paramètre.
La liste sert à la sauvegarde.

- **Méthode ”SaveSequence”**
  Cette méthode sauvegarde donc la séquence dans le mode libre.
  Tout d'abord, elle crée les informations de base de la séquence : son nom, l'auteur, le durée totale de la séquence, la date actuelle et possiblement une image ou une musique.
  Ensuite, on ajoute les mortiers avec leurs positions, leurs angles, leurs hauteurs et leurs largeurs.
  Finalement, je parcours la liste des feux d'artifices et les ajoute dans mon xml avec comme données principale le type (comète ou pluie de particules), le temps de lancement du feu d'artifice (un chronomètre se lance à l'arrivée dans le mode libre), leurs positions, etc.

- **Reproduction de la séquence**
  Pour reproduire la séquence, un chronomètre se lance et reproduis les feux d'artifice lorsque le chronomètre est identique à la valeur stockée dans chaque feu d'artifice du fichier xml.

Plan de test et tests

Périmètre de test

Pour MiFiSy, je vais créer un plan de test visant à garantir le bon fonctionnement de l'application du point de vue de l'utilisateur.
Ce plan inclura des tests fonctionnels pour évaluer à la fois la performance et la convivialité de l'interface utilisateur.
Ensuite, je consignerai tous les tests réalisés ainsi que leurs résultats dans un tableau pour assurer la qualité de l'application et suivre l'évolution du projet.

Plan de test

N°	Description du test	Résultat attendu
1	Lors d'un clique sur une musique d'ambiance	La couleur du nom de la musique change et la musique est sélectionnée
2	Lors d'un clique sur une musique d'ambiance puis sur le mode libre dans l'accueil	La musique sélectionnée est jouée en boucle dans le mode libre
3	Lors d'un clique sur le mode libre dans l'accueil sans cliquer sur une musique	Le mode libre se lance sans musique
4	Lors d'un clique sur un replay dans l'accueil	Le replay se lance, la musique et les effets sont identiques
5	Lorsque le replay est terminé	Un message l'indiquant apparaît à l'écran
6	Lors du clique sur 'Accueil' dans le mode libre ou replay	Retour à la page d'accueil
7	Lors du clique sur 'Sauvegarder' dans le mode libre	Toute la séquence crée est sauvegardé dans un fichier xml et un message "Sauvegarde réussi" apparaît brièvement au milieu de l'écran
8	Dans le mode libre, lorsque la première corde de la guitare est joué	L'effet de comète est crée sur un des mortiers aléatoirement
9	Dans le mode libre, lorsque la deuxième corde de la guitare est jouée	L'effet de pluie de particules est crée aléatoirement sur le haut de l'écran
10	Si la guitare n'est pas trouvée lors du démarrage de l'application	Un message d'erreur apparaît au milieu de l'écran
11	Dans l'accueil, s'il y a plus de 10 musiques	Seuls les 10 premiers s'affichent dans l'ordre alphabétique
12	Dans l'accueil, s'il y a plus de 10 fichiers a rejouer	Seuls les 10 premiers s'affichent dans l'ordre alphabétique
13	Lors du chargement d'un fichier extérieur au programme (image ou musique), si le fichier n'existe pas	L'application continue de fonctionner et aucun fichier n'est chargé
14	Lors du chargement d'un fichier extérieur au programme, si le fichier ne correspond pas au type demandé	L'application continue de fonctionner et aucun fichier n'est chargé

Evolution des tests

N Test	J1	J2	J3	J4	J5	J6	J7	J8	J9	J10	J11
1	✗	✗	✗	✗	✓	✓	✓	✓			
2	✗	✗	✗	✗	✓	✓	✓	✓			
3	✗	✗	✗	✗	✓	✓	✓	✓			
4	✗	✗	✗	✗	✗	✓	✓	✓			
5	✗	✗	✗	✗	✗	✓	✓	✓			
6	✗	✓	✓	✓	✓	✓	✓	✓			
7	✗	✗	✗	✓	✓	✓	✓	✓			
8	✗	✗	✗	✓	✓	✓	✓	✓			
9	✗	✗	✗	✓	✓	✓	✓	✓			
10	✗	✗	✗	✗	✓	✓	✓	✓			
11	✗	✗	✗	✗	✗	✗	✓	✓			
12	✗	✗	✗	✗	✗	✗	✓	✓			
13	✗	✗	✗	✗	✗	✗	✗	✓			
14	✗	✗	✗	✗	✗	✗	✗	✓			

Rapport de test

N°	Date du test	Résultat obtenu	OK/KO

Conclusion

Difficultés rencontrées

Amélioration possible

Bilan personnel

Annexes

Planning prévisionnel

Planning effectif