Chapitre IV

Projet Jeu Vidéo

NB : Tous les documents nécessaires à la réalisation de ce projet vous sont fournis sous forme de documents téléchargeables sur la plate-forme.

1 Présentation générale du projet

1.1 Les exigences du projet

Vous allez développer un jeu vidéo en C# en utilisant le framework de jeu vidéo `MonoGame`. Le développement du jeu sera libre. Autrement dit, ce sera a vous de proposer un jeu et d'imaginer un gameplay.

Soyons clairs, vous n'allez pas concevoir et développer le nouveau *Call of Duty*, le nouveau *Skyrim*, ni le nouveau *Mario Kart*. Vous n'en n'aurez pas le temps et ce n'est clairement pas le but.

Le but, je le rappelle est que vous mettiez en pratique des savoir-faire incluant

- la conception d'un cahier des charges
- la structuration de votre code, de vos données
- l'usage de concepts et technologies dont XML/XSD/XSLR, les parsers, la sérialisation, mais aussi la programmation objet, les concepts issus de la programmation fonctionnelle ...

Vous serez notés sur ces critères en très grande partie ; votre implication dans ce projet, le soin que vous apporterez à votre code feront également partie de la notation. Ce projet vous donne donc une grande liberté d'action, sous contrainte d'inclure des éléments technologiques et conceptuels qui seront requis, mais attention, c'est aussi une difficulté car cela va vous demander de vous impliquer fortement dans ce projet et de vous dépasser concernant la programmation. Ce que vous allez réaliser, démarche incluse, préfigure le projet intégrateur qui démarre en décembre.

1.2 Organisation du quadrinome

Vous travaillerez en quadrinomes. L'objectif est que vous participiez tous au code, à la conception ... mais aussi que vous appreniez à vous répartir les tâches (telle partie du code / telle autre partie, ...). Dans tous les cas, vous devez nommer un chef d'équipe qui synchronisera les tâches de votre projet et organisera les points réguliers que vous devez avoir.

1.3 La conception

La conception de votre jeu (son concept) fait partie de l'exercice et le soin qui y sera apporté fera d'une part partie de la notation, d'autre part constituera une étape essentielle de votre travail puisqu'elle vous permettra de guider le développement du programme correspondant.

Vous aurez à :

- Sélectionner ou imaginer un concept de jeu (jeu de rôle, jeu de plate-forme, shooter, puzzle game ...)

- identifier (lister) les composants de base (ex: un joueur, des créatures, une carte positionnant des éléments de décors, ...)
- identifier (lister) les interactions de base (mouvement du joueur, blocages sur la carte (collisions), actions simples du joueur, détection par des créatures, action des créatures ...)
- structurer le code et son fonctionnement via des diagrammes UML. Ces diagrammes sont essentiels. Ils vous permettront d'analyser *a priori* et concevoir l'architecture logicielle de votre jeu. Les deux diagrammes attendus sont :
  - un diagramme de classe : les composants attendus de votre jeu (exemple : créatures, joueur, décors, écran de jeu ...) devront être organisés et devront utiliser intelligemment les notions d'héritage et de composition.
  - le fonctionnement du jeu (déroulement d'une partie, changements de niveaux, actions de création de joueur, de chargement, de sauvergarde ... devront être décrits dans un diagramme fonctionnel.
- apprendre à afficher un objet interactif (ex: player) avec Monogame
- ecrire, par étapes, un programme sans graphiques (sans MonoGame) dans lequel les composants et actions de base son implémentées (sorties console)
- permettre à ces composants de charger des données ou de les sauvegarder (sérialisation, utilisation de parsers)
- ajouter ces éléments (composants et actions) au programme graphique

Il sera demandé de créer une **fiche signalétique du jeu** identique à celles présentées sur ce site dont la lecture détaillée vous donnera tous les éléments pour créer votre jeu : Tutoriel **Ecrire un Game concept**

Mes recommandations personnelles sont :

- Imaginez un jeu simple :
  - simple dans le concept : un jeu compliqué d'un point de vue conceptuel sera difficile à concevoir (à formaliser) puis à implémenter. Un jeu simple peut aboutir à une expérience vidéoludique tout a fait amusante, agréable, intéressante.
  - simple dans la réalisation : ne partez pas dans quelque chose que vous ne pourriez pas réaliser. Par exemple, si vous faites un jeu de rôle, une simple carte limitée à l'écran de jeu, un seul joueur, et quelques interactions suffiront pour prouver le concept de jeu.
- Testez régulièrement des versions fonctionnelles et sauvegardez les.
- Utilisez un dépot git pour ces sauvegardes.

Un jeu simple et un peu moche mais fonctionnel et respectant les exigences de l'UE (les miennes) aura de la valeur. Un jeu compliqué et/ou magnifique mais non fonctionnel et ne respectant pas mes exigences n'aura **aucune** valeur.

1.4 A propos des IA génératrices (chatGPT et ses copains)

Certains d'entre vous vont être très tentés de passer par ces IA génératrices de texte et contenus graphiques. C'est une très mauvaise idée et je vais tout de suite vous mettre en garde :

- d'abord ce ne sera en conséquence pas votre projet mais celui produit par un ChatGPT (ou autre), sachant que ces moteurs fonctionnent en produisant un contenu moyen à partir de ce qui l'a alimenté (d'autres humains). J'ajoute dans ce point que vous auriez peu de fierté à valider un projet / une UE en trichant ... en premier avec vous même, vu que passer par ces systèmes pointera clairement un manque de savoir-faire et/ou d'autonomie.
- et la conséquence du point précédent, c'est que de nos jours on peut sans peine détecter l'usage d'un tel outil (par exemple avec des outils comme ZeroGPT qui ont été entrainés à reconnaitre l'usage de ces IA génératives et qui sont très efficaces).

Autant vous dire que je ne me prive pas de me servir d'outils de détection. Sans compter que nous passerons voir vos projets et verrons dans quelle mesure *vous* vous impliquerez dans vos projets.

2 Quelques concepts de jeux

Ces quelques pitchs pourront vous inspirer. Il est recommandé mais pas obligatoire de suivre un concept voisin d'un de ces 3 jeux.

2.1 Un puzzle d'aventure avec des feux follets

a) Pitch Vous êtes Flamèche, un feu follet. Vous vous êtes égaré dans le bayou. Votre objectif est, de plateau en plateau, d'atteindre à la fin le cimetière dont vous êtes issu pour retrouver votre famille de feux follets. Chaque plateau constitue un niveau avec une entrée (là où vous êtes) et une sortie (un point à atteindre).

Au cours de vos aventures, vous visitez divers biomes (des marais, des forêts humides, des forêts sèches, des prairies, des cours d'eau ...). Plusieurs éléments dans ces milieux sont très combustibles (les branches de bois mort, les arbres morts, les bulles de gaz sortant du marais, les herbes sèches ...), d'autres sont peu combustibles (les branches et arbres verts, les herbes vertes), d'autres sont non combustibles (pierres, sable, terre), d'autres enfin éteignent les feux (chutes et cours d'eau, foret humide ...).

Vous vous déplacez donc de case en case (avec les flèches de direction) mais pouvez rarement vous arrêter autrement que sur les zones non combustibles. Les arrêts même assez courts sur les zones très combustibles mettent le feu. Les arrêts prolongés sur les zones même peu combustibles finissent par déclencher un feu. Les arrêts plus ou moins prolongés sur les zones humides provoquent votre extinction (= votre mort).

Le feu a des effets sur les cases voisines : il se propage. Il a aussi un effet sur vous, s'il vous fait grossir (et vous rend plus dangereux pour l'environnement), il peut être mortel en vous consumant complètement si vous vous y exposez trop longtemps.

Vous comprenez ainsi que les éléments du décors ont leur propre vie : ils ont un état (un niveau de combustibilité positif ou négatif), ils ont une durée d'exposition à votre présence ou a la présence de flammes voisines (augmentation de chaleur = une accumulation de chaleur) ou d'humidité voisine (baisse de chaleur = accumulation de rafraichissement), et ils peuvent changer d'état (passer d'élément naturel à flamme, puis de flamme à tas de cendres non combustible).
Vous même, en tant que feu follet, présentez une durée d'exposition au décors à votre contact : exposition au feu (vous grossissez et augmentez votre action de feu sur les éléments voisins, mais réduisez vos points de vie) ; exposition à l'humidité (vous devenez plus petit et diminuez ainsi votre action calorifique sur les éléments du décors, mais diminuez vos points de vie). Votre vie remonte toute seule progressivement mais lentement.

Le gameplay sera pensé comme un puzzle avec des parties à résoudre pour atteindre vos objectifs à chaque plateau (par exemple traverser une forêt). L'idée est de constituer le gameplay (et de jouer) en exploitant le fait que le joueur puisse volontairement mettre le feu pour provoquer des actions, pour passer, ... ou au contraire puisse volontairement s'humidifier pour éviter de déclencher des incendies.

b) Difficulté : * à **

La réalisation de ce jeu est relativement peu difficile. Les cartes sont simples et sur 1 plateau à chaque fois, les sprites (voir définition plus bas) n'ont pas besoin d'être animés, les actions sont simples à implémenter.

2.2 Le jeu Crazy Classroom : micro-management d'une classe

Sachez que le jeu Crazy Classroom a été réalisé par 2 étudiants de BTS sous ma direction récemment ... et qu'il est plutôt amusant.

a) Pitch Le professeur Krab porte bien son nom puisqu'il passe son temps à se déplacer entre les chaise et les tables comme un crabe pour répondre au questions des étudiants. Pas seulement pour leur répondre d'ailleurs, mais aussi pour leur demander de se taire et de se remettre au travail. Son objectif,

c'est que sa salle de classe reste disciplinée et que les étudiants finissent leur projet.

Les élèves ont un avancement progressif de leur travail (par exemple 1% par seconde de temps réel de jeu), mais cette progression est ponctuée d'arrêts qui mettent en pause l'avancement du travail. Les arrêts sont de 3 nature :

- (A) les questions symbolisées dans une bulle de dialogue verte (un affichage graphique) par un point d'interrogation (**?**).
- (B) les "monsieur/madame, ça marche pas !", des *non-questions* symbolisées dans une bulle de dialogue bleue par un point d'exclamation (**!**).
- (C) le bavardage symbolisé dans une bulle de dialogue rouge par le texte "blablabla".

Ces évènements se produisent de façon probabiliste (tirages aléatoires) avec une fréquence configurable dans un fichier de configuration du jeu. Dans une première version du jeu les évènements sont exclusifs pour chaque élève : si un évènement A, B ou C se déclenche pour un élève, tant qu'il n'est pas débloqué, il ne peut se produire d'autres évènements pour cet élève. Une autre version du jeu peut rendre non exclusif le bavardage par exemple.

Les bulles de dialogue montrant les évènements A, B ou C sont des boutons permettant par un click une ou plusieurs actions de la part du professeur. Dans les 3 situations, le professeur doit se déplacer jusqu'à l'élève pour débloquer la situation.
Parmi les actions possibles, le professeur peut :

- **(A) Question de l'élève** :
  - soit répondre à la question -> l'élève est débloqué immédiatement mais son travail ne vaut pas 100% de la note maximale
  - soit donner un indice à l'élève -> dans ce cas l'élève n'est pas débloqué immédiatement (quelques secondes sont nécessaires), mais il ne perd pas de points
- **(B) Exclamation de l'élève** :
  - soit débloquer l'élève en lui donnant la solution -> dans ce cas l'élève perd des points, plus que dans le cas A correspondant (car l'élève n'a pas fait d'efforts)
  - soit lui donner un indice -> l'élève n'est pas débloqué immédiatement (quelques secondes sont nécessaires), et il perd quelques points, plus que dans le cas A correspondant, mais moins que si on lui donne la réponse
  - soit dire à l'élève que c'est un feignant et qu'il lise la doc et se débrouille -> dans ce cas le déblocage est possible mais pas systématique ; cette possibilité ainsi que le temps auquel se produit l'éventuel déblocage sont obtenus par tirage aléatoire
- **(C) Bavardage** :
  - soit dire à l'élève de se taire et de se remettre au travail -> dans ce cas l'élève perd des points ; de plus le déblocage est possible mais pas systématique ; cette possibilité ainsi que le temps auquel se produit l'éventuel déblocage sont obtenus par tirage aléatoire
  - soit dire à l'élève de sortir de classe -> l'élève ne gagne plus de points. Le score de jeu sera affecté d'abord parce que l'élève n'a pas une bonne note, d'autre part parce que l'enseignant n'a pas réussi a garder cet élève en cours ; il est pénalisé de la différence entre la note maximale (ex: 100%) et la note atteinte par l'élève au moment de son exclusion.

Lorsque le prof Krab tarde à s'occuper des élèves, les élèves s'énervent. Le taux d'énervement augmente donc avec le temps à partir du moment où l'élève est bloqué. Ce taux rediminue progressivement à mesure que l'élève travaille. Dans une version évoluée du jeu, vous pouvez également faire en sorte que les élèves voisins d'un élève qui bavarde et/ou qui s'énerve, s'énervent aussi.
L'énervement d'un élève (même lorsqu'il travaille et que son niveau d'énervement n'est pas redescendu à 0) augmente la probabilité qu'il bavarde. De plus, lorsqu'un élève est bloqué, s'il atteint son niveau d'énervement maximal, l'élève réalise une action inconsidérée :

- il sort de cours (il s'exclut lui-même) -> l'élève ne gagne plus de points. Le score de jeu sera affecté d'abord parce que l'élève n'a pas une bonne note, d'autre part parce que l'enseignant n'a pas réussi a garder cet élève en cours ; il est pénalisé de la différence entre la note maximale (ex: 100%) et la note atteinte par l'élève au moment de son exclusion.

- il provoque le bavardage de tous ses voisins

- ...

Un niveau de jeu est fini quand tous les élèves ont terminé leur travail ou sont sortis ou que le temps imparti pour un TP est terminé (vous pouvez par exemple fixer à 150 secondes le temps maximal, soit 1.5 fois le temps d'avancement normal de chaque élève (100 secondes)).

Les niveaux de jeu augmentent le nombre d'élèves, complexifient la disposition des bureaux dans les salles, rajoutent des obstacles, rajoutent des salles (impliquant que vous pouvez sélectionner la salle), rajoutent des professeurs, font réaliser un travail aux profs en parallèle (le prof doit finir son travail aussi avant la fin), transforment les élèves énervés en zombies agressifs qui poursuivent le prof, un dinosaure entre dans la salle, les PCs de la fac tombent en panne aléatoirement impliquant de nouvelles actions (déplacement de l'élève), le prof doit vraiment débugger du code qui s'affiche à l'écran pour débloquer la situation, ... Tout est possible, mais soyons clairs, le fonctionnement de la version "de base" sera déjà très bien !

b) Difficulté : ** à ***

Le jeu contient plus de composants (les bulles de dialogue émises par et appartenant aux élèves, les bureaux qui représentent l'avancement du travail des élèves, ...) et d'actions dont certaines sont ordonnées (suite d'actions de réponses à mener auprès des élèves ...). Les événements se produisant doivent être surveillés par les profs, mais aussi par les élèves voisins ...

2.3 Un shooter horizontal / un astéroids

a) Pitch Vous êtes dans un vaisseai spatial qui peut tirer des missiles ou tout autre projectile. Dans le cas d'un jeu de type Asteroids, tout se passe sur un seul écran de jeu. Le vaisseau peut se déplacer dans toutes les directions dans l'écran. De partout arrivent des objets, en particulier des astéroides, qu'il faut détruire.
Dans le cas d'un shooter (par exemple un shooter dit 'horizontal'), le vaisseau est aligné à l'horizontale (dirigé de gauche à droite) et semble se déplacer vers la droite. En réalité, c'est tout le décors qui se déplace vers la gauche. Des enemis apparaissent depuis la droite, soit immobiles (des éléments agressifs du décors, par exemple des tourelles canon), soit mobiles comme d'autres vaisseaux oud es créatures. Le but est d'arriver au bout du niveau évidemment.

Vous remarquerez que ce concept de jeu peut se transposer à d'autres situations, par exemple un jeu dans lequel le vaisseau est un globule blanc dans votre sang. Les énemis sont les virus et autres bactéries et parasites qui infectent votre hôte. Vous les détruises soit en les phagocytant (vous chercherez la définition en ligne), soit en envoyant dessus des immunoglobulines (là aussi, cherchez la définition). Vous pouvez vous faire aider par d'autres cellules immunitaires en communiquant avec elles grâce à des messagers chimiques, les cytokines. De plus, de nombreux éléments normaux mais gênants vous arrivent dessus comme des globules rouges, éléments qu'on peut détruire mais qu'il vaut mieux éviter de détruire pour ne pas affecter la survie de votre hôte.

b) Difficulté : asteroids : * / shooter horizontal : ** à ***

Dans le cas d'un asteroids, le jeu est assez simple a programmer car tout se passe sur un seul écran de jeu. Les difficultés viennent des mouvements d'une part (pas case à case), et des actions médiées par agents (le vaisseau du joueur emet des projectiles qui agissent sur les objets à détruire / modifier).

Dans le cas d'un shooter, la difficulté rajoutée est le changement continu d'écrans (scrolling) et la carte de grande taille ou la génération procédurale de carte.

3 Réalisation

3.1 Définitions préalables

Au cours de la conception du jeu, vous allez rencontrer des mots consacrés. J'en donne la définition ici :

- **Sprite** : c'est une marque de soda, mais c'est aussi un élément graphique que l'on peut éventuellement déplacer à l'écran. Un sprite peut être animé (ex: créature) ou non (ex: texte affiché avec une fonte graphique). C'est sa gestion en tant qu'objet graphique affichage qui le désigne comme sprite.
- **Décor** : à la différence du sprite le décor est souvent constitué d'une image de fond souvent immobile. Il est possible de rendre dynamique le décor avec l'usage de sprites (qui deviennent des éléments de décors mobiles ou animés).
- **Périphérique** : les périphériques désignent les éléments de l'ordinateur reliés à l'unité centrale. Il y a les périphériques d'entrée et les périphériques de sortie. Dans le cadre du jeu vidéo, nous aurons affaire aux périphériques d'entrées, à savoir le clavier et la souris, dont nous capturerons les états (touches pressées, relachées, maintenues, mouvements ...). Nous aurons aussi affaire au périphérique de sortie graphique (et éventuellement sonore).
- **Scrolling** : le scrolling est une opération graphique qui consiste à faire se déplacer le décor de façon fluide dans l'écran. Du nouveau décor apparaît d'un côté de l'écran vers lequel le sujet principal du jeu (le joueur) se déplace, tandis qu'à l'endroit opposé le décor disparaît.
- **Effet** : un effet graphique est une opération permettant de réaliser une transition d'une image vers une autre image, par exemple le fondu entre 2 images, le clignotement d'une image, le rétrecissement ou l'agrandissement d'une image ...
- **Gameplay** : le gameplay, c'est le coeur du jeu, à savoir ce qui fait du programme un jeu. Il intègre bien entendu les relations entre les composants du jeu (les créatures, le décor, ...), les contrôles que le joueur humain a sur la partie, les règles de jeu. Le soin qu'on y apporte contribue, avec le level design, au succès de l'expérience vidéoludique.
- **Level design** : le level design confère au jeu son intérêt ludique. Il s'agit de l'architecture des niveaux, de la façon dont le jeu va affecter la progression du joueur.

3.2 Composants essentiels du jeu et leur fonctionnement

Dans les 3 concepts présentés ci-dessus, il y a des points communs. Ils comportent tous au minimum de :

- un plateau de jeu intégrant éventuellement des décors passifs et positionnant les éléments actifs de jeu
- un joueur (élément actif) qui est habituellement représenté par un avatar (un personnage dans le jeu). Il peut ne pas y avoir d'avatar.
- un ensemble d'éléments actifs à vie propre (agents) qui peuvent influencer les états du joueur et qui peuvent s'influencer entre eux.
- des évenements soit scriptés, soit aléatoires
- des interactions directes (le prof clique sur la bulle de bavardage de l'élève pour lui dire de se taire) ou indirectes (le joueur envoie un objet sur un autre objet, par exemple un missile sur un astéroïde), les deux déclenchées par des actions clavier et/ou souris.

3.3 Ressources

Vous trouverez ici un certain nombre de ressources utiles pour la conception de votre jeu.

- Tutoriel **Ecrire un Game concept**
- La documentation de **Monogame**

- **Pixilart**, une app en ligne pour éditer des sprites
- **Piskel**, une autre app en ligne (et hors ligne) pour éditer des sprites
- **Universal LPC Spritesheet Generator**, une app en ligne pour générer des tuiles de sprites de personnages
- **Open Game Art**, un site vous donnant accès à de très nombreux objets graphiques (sprites, décors, ...)

4 Un premier projet Monogame - Jeu “MyGame”

La documentation de Monogame se trouve à ce lien.

4.1 Installation de Monogame

Nous allons installer la bibliothèque du framework Monogame. Pour ça, ouvrez un terminal et tapez la commande :
`dotnet new install MonoGame.Templates.CSharp`
Cela va installer la lib et les modèles de projets.

4.2 Création d’un projet Monogame (cross-platform)

Ce que nous voulons créer, c’est un projet MonoGame multi-plateformes, à savoir dont les graphismes seront gérés par la bibliothèque OpenGL.

Vous avez à votre disposition 2 façons de créer un projet dotnet, et en particulier ici un projet Monogame.

- **Depuis un terminal :** Dans un terminal, tapez la commande
`dotnet new mgdesktopgl -o NomDuProjet` où `NomDuProjet` est le nom que vous donnez à votre projet, par exemple `BasicMonoGame`.
- **Depuis Jetbrains Rider [Recommandé] :** Depuis le menu, faites :`New Solution -> Monogame Cross-Platform Desktop Application`. Cela ouvrira une fenêtre de dialogue dans laquelle vous indiquez le nom de votre Solution / Projet (mettez le même nom, par exemple `BasicMonoGame`), le chemin dans lequel vous créez la solution. Cochez les cases “Put solution and project in the same directory” et “Create Git repository”.

NB : Pour connaitre tous les types de projets pouvant être créés depuis dotnet (étant donné les libs installées) vous pouvez taper dans un terminal la commande
`dotnet new list`

Le projet créé contient une classe `Program` et une classe `Game1` qui hérite de la classe `Game` de la lib `Monogame`. Vous pouvez renommer cette classe (refactoring) bien entendu, par exemple `Game1` devient `MyGame`. Dans la suite des descriptifs ci-dessous, nous utiliserons ce nom là pour la désigner (`MyGame`).

Une classe (comme `MyGame`) héritant de `Game` permet de gérer un jeu. Elle contient (obligatoirement):

- Un attribut statique `Content`. Cet attribut de classe se réfère à un gestionnaire de contenu de type `ContentManager`. C’est à cette instance statique `Content` que vous transmettrez vos contenus graphiques (sprites, fontes ...).
- Les attributs dynamiques :
  - `_graphics : GraphicsDeviceManager` qui permet de lier le jeu (classe `Game` et dérivées) au gestionnaire de périphériques graphiques.
  - `_spriteBatch : SpriteBatch` qui permet de gérer les contenus graphiques (les sprites) compilés par mgcb (voir plus bas) : il met en interaction le périphérique graphique et les contenus (`Content`). Les dessins sont réalisés dans le spritebatch

- un constructeur qui instancie notamment un nouveau `GraphicsDeviceManager`
- 4 méthodes protégées surchargées (overrided) :
  - `void Initialize()` qui permet d'initialiser toutes les valeurs nécessaires à votre jeu. Elle est appelée au lancement du jeu.
  - `void LoadContent()` qui permet de charger les contenus dans l'instance `Content`. Elle est appelée au lancement du jeu.
  - `void Update(GameTime gameTime)` est une callback qui permet d'actualiser les évènements d'entrée (clavier, souris).
  - `void Draw(GameTime gameTime)` est une callback qui permet de dessiner.

4.3 Test du projet Monogame

Attendez que le projet s'ouvre complètement puis compilez (bouton Run). Si tout va bien, une fenêtre graphique bleue s'ouvrira. Pour sortir, pressez `Echap / Escape`.

4.4 Installation de l'outil mgcb (MonoGame Content Builder)

Une fois le projet Monogame créé, nous allons tester la présence de l'éditeur de ressources pour monogame et sinon l'installer.

- Ouvrez le terminal de Rider (petit icone `Terminal` en bas à gauche
- Dans le terminal tapez la commande `dotnet mgcb-editor`. Normalement un outil s'ouvre.
- Si ce n'est pas le cas, installez mgcb et mgcb-editor depuis le terminal de Jetbrains Rider en tapant `dotnet tool install -g dotnet-mgcb`.
  Testez à nouveau (point au dessus).

4.5 Ajout de ressources de jeu avec mgcb-editor

Nous allons maintenant ajouter des ressources qui seront prises en charges par MonoGame. Notez que dans votre jeu, toutes les ressources ne seront pas prises en charge par MonoGame, par exemple les fichiers XML... Ce que l'on désigne donc là par "ressources gérées par MonoGame", ce sont des images, sons, fontes, *etc* que les méthodes de la bibliothèque utilisent pour afficher des images (sprites) ou du texte et jouer des sons.

Pour être prises en compte, ces ressources doivent être intégrées dans le projet MonoGame et compilées par mgcb (le MonoGame Content Builder). Cet outil génère des fichiers à l'extension `xna` que l'on appelle contenu (`Content`). L'outil `mgcb` gère un projet nommé par défaut `Content.mgcb`. Ce projet peut être ouvert par `dotnet mgcb-editor Content.mgcb` ; il contient les ressources associées et permet de les compiler en contenus. Pour tout savoir sur les contenus MonoGame, référez vous à ce lien.

Dans le terminal de Jetbrains Rider, executez la commande `dotnet mgcb-editor &`. Cela ouvrira un editeur. De là, vous pouvez rajouter des items existants puis sauvegarder et compiler votre projet mgcb.

4.6 Ajout d'un sprite controlé par les périphériques d'entrée

Avec mgcb, ajoutez un sprite (montrant un personnage, un vaisseau spatial ou un objet quelconque pour l'instant), ce sprite vous servant à matérialiser le joueur (donc son avatar). Dans cet exemple, nous désignerons l'image du sprite par un nom générique (disons `sprite.png`) mais vous pouvez utiliser bien entendu un nom différent.

Créez une classe `Sprite`. Cette classe sera utilisée pour afficher le sprite. Nous ferons en sorte qu'un sprite soit instancié et mis en oeuvre par le jeu (classe `MyGame`). Voici la classe `Sprite` (fournie dans le TP) :

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;

namespace BasicMonoGame;

public class Sprite {

    private Texture2D _texture;
    protected Vector2 _position;
    private int _size = 100;
    private static readonly int _sizeMin = 10;
    private Color _color = Color.White;

    public Texture2D _Texture { get => _texture; init => _texture = value; }
    public int _Size { get => _size; set => _size = value>=_sizeMin?value:
        _sizeMin; }
    public Rectangle _Rect { get => new Rectangle((int) _position.X, (int)
        _position.Y, _size, _size); }

    public Sprite(Texture2D texture, Vector2 position, int size) {
        _Texture = texture;
        _position = position;
        _Size = size;
    }

    public void Update(GameTime gameTime){
        if (Keyboard.GetState().IsKeyDown(Keys.Up)) { /*...*/ }
        //...
    }

    public void Draw(SpriteBatch spriteBatch) {
        var origin = new Vector2(_texture.Width / 2f, _texture.Height / 2f);
        spriteBatch.Draw(   _texture, // Texture2D,
            _Rect, // Rectangle destinationRectangle,
            null, // Nullable<Rectangle> sourceRectangle,
            _color, //  Color,
            0.0f, //  float rotation,
            origin,  // Vector2 origin,
            SpriteEffects.None, // SpriteEffects effects,
            0f ); // float layerDepth
    }

}
```

Cette classe contient les attributs suivants :

- `_texture : Texture2D` : permet de charger une texture faisant partie des contenus compilés par mgcb
- `_position : Vector2` : permet de gérer la position de l’objet (coin supérieur gauche du sprite)
- `_size : int` : donne une taille au sprite
- `_sizeMin : int` : cet attribut statique permet de fixer une taille minimale qu’un sprite peut avoir
- `_color : Color` : permet de gérer la couleur du sprite. Par défaut cette couleur est blanche ; ainsi le sprite affiché est celui de l’image ; sinon l’image prend la couleur indiquée par cet attribut.

Vous remarquerez le rôle des propriétés `_Texture`, `_Size` et `_Rect` pour gérer l’accès de certains attributs ou rajouter un attribut calculé. La propriété `_Rect` permet de désigner un rectangle qui constitue la bounding box de l’objet.

La classe implémente les méthodes :

- `void Update(GameTime gameTime` : permet de gérer les modifications d'attributs comme la position, l'angle ou la taille en fonction de l'usage de touches du clavier ou de la souris.
- `void Draw(SpriteBatch spriteBatch` : permet d'afficher le sprite dans le spritebatch.

Pour mettre en oeuvre le sprite, vous devez l'instancier dans la classe `MyGame` et appeler ses méthodes `Update` (en transmettant le `gameTime` de `MyGame`) et `Draw` (en transmettant le `spriteBatch` de `MyGame`), comme suit :

```
1  using Microsoft.Xna.Framework;
2  using Microsoft.Xna.Framework.Graphics;
3  using Microsoft.Xna.Framework.Input;
4
5  namespace BasicMonoGame;
6
7  public class MyGame : Game {
8      private GraphicsDeviceManager _graphics;
9      private SpriteBatch _spriteBatch;
10     private Sprite _ship; // instance de Sprite
11
12     public MyGame() {
13         _graphics = new GraphicsDeviceManager(this);
14         Content.RootDirectory = "Content";
15         IsMouseVisible = true;
16     }
17
18     protected override void Initialize() {
19         base.Initialize();
20     }
21
22     protected override void LoadContent() {
23         _spriteBatch = new SpriteBatch(GraphicsDevice);
24         Texture2D shipTexture = Content.Load<Texture2D>("ship2");
25         _ship = new Sprite(shipTexture, new Vector2(150, 150));
26     }
27
28     protected override void Update(GameTime gameTime) {
29         if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.
               Pressed ||
30             Keyboard.GetState().IsKeyDown(Keys.Escape))
31             Exit();
32         _ship.Update(gameTime);
33         base.Update(gameTime);
34     }
35
36     protected override void Draw(GameTime gameTime) {
37         GraphicsDevice.Clear(Color.CornflowerBlue);
38         _spriteBatch.Begin(samplerState: SamplerState.PointClamp);
39             _ship.Draw(_spriteBatch);
40         _spriteBatch.End();
41         base.Draw(gameTime);
42     }
43 }
```

Modifiez la méthode `Update` de manière à faire bouger en avant, en arrière et sur les côtés votre vaisseau lorsqu'une touche du clavier est pressée.

Modifiez la méthode `Update` de manière à réduire la taille du vaisseau ou au contraire à la faire grossir. A cet effet, profitez en pour ajouter un attribut statique `_sizeMax` (de valeur 100 par exemple) et modifiez la propriété `_Size` de façon à ce que la taille `_size` ne puisse pas dépasser cette taille maximale.

4.7 Un mouvement plus réaliste

Ajoutez un attribut privé `_speed : Vector2` qui donnera une vitesse à notre sprite. Nous allons nous en servir pour réaliser un mouvement réaliste ayant les caractéristiques suivantes :

- l'appui de la flèche du haut augmente la vitesse de la valeur 1. Cela fait avancer le vaisseau dans la direction vers laquelle il pointe
- l'appui de la flèche du bas augmente la vitesse (arrière) de la valeur -1. Cela fait reculer le vaisseau dans la direction opposée.
- l'appui de la flèche droite augmente la vitesse de la valeur 0.5. Cela fait bouger le vaisseau latéralement à droite (strafe). Le mouvement est moins rapide et doit diminuer plus vite.
- l'appui de la flèche gauche augmente la vitesse de la valeur 0.5. Cela fait bouger le vaisseau latéralement à gauche (strafe). Le mouvement est moins rapide et doit diminuer plus vite.
- la vitesse revient à la valeur 0 alors que le temps de jeu passe. Pour ça, si la vitesse est positive, on fait diminuer celle-ci de la valeur 0.05 à chaque update. Réciproquement, si elle est négative, on la fait augmenter de 0.05.

Modifiez la fonction update de votre Sprite comme suit :

```
1   public void Update(GameTime gameTime) {
2
3       if (Keyboard.GetState().IsKeyDown(Keys.Up)) {
4           _speed.X +=1.1f;
5       }
6       if (Keyboard.GetState().IsKeyDown(Keys.Down)) {
7           _speed.X -=1.1f;
8       }
9       if (Keyboard.GetState().IsKeyDown(Keys.Right)) {
10          _speed.Y +=0.5f;
11      }
12      if (Keyboard.GetState().IsKeyDown(Keys.Left)) {
13          _speed.Y -=0.5f;
14      }
15
16      _position.X = _position.X + _speed.X;
17      _position.Y = _position.Y + _speed.Y;
18      if (_speed.X > 0) _speed.X -= 0.05f;
19      if (_speed.X < 0) _speed.X += 0.1f;
20      if (_speed.Y > 0) _speed.Y -= 0.1f;
21      if (_speed.Y < 0) _speed.Y += 0.1f;
22  }
```

Une amélioration consiste à stocker la décélération (décroissance de la vitesse) dans une variable `_speedDec`. De même pour l'accélération avec une variable `_speedAcc`. Cela permet de régler les incréments décréments de vitesse avec des valeurs qu'on peut modifier, contrairement aux valeurs fixes proposées dans le listing ci-dessus. Ces variables doivent être des propriétés ou être des attributs privés contrôlés par des propriétés de façon à imposer des limites minimale et maximale à l'accélération et à la décélération, par exemple des valeurs comprises entre $0.0f$ et $1.0f$. Elles devront bien entendu être instanciées et initialisées dans le constructeur.

4.8 Modification : création d'une classe GameObject et d'une classe Player qui héritent du Sprite

Nous allons maintenant utiliser cette base pour réaliser un principe de jeu plus ambitieux.

Vous allez :

- Créer une classe `GameObject` qui hérite de la classe `Sprite`. Dans ce contexte, un `Sprite` n'est plus qu'un objet affichage mais ne dispose plus de controles, de vitesse, de caractéristiques autre que celles de son affichage. Un `GameObject` est un objet de jeu pouvant se déplacer ; c'est lui qui dispose d'une vitesse, d'une accélération, d'une décélération. La classe `Sprite` ne doit donc plus

réaliser d'update de mouvement ; c'est le `GameObject` qui met à jour la position en fonction de la vitesse.

- Créer une classe `Player` qui hérite de la classe `GameObject`. Seul le le `Player` est doté du contrôle des mouvements. C'est aussi sa fonction d'update qui fait décélérer la vitesse. Cette fois, c'est à ce joueur qu'est associées la texture de vaisseau `ship2` qui est fournie.

- A ce stade votre vaisseau est contrôlable mais il peut traverser les bords de l'écran (et être perdu de vue). A vous de faire le nécessaire pour ne pas perdre votre vaisseau. Il y a plein de solutions, par exemple :

  - faire chuter la vitesse à 0 (empêcher le déplacement) lorsqu'on arrive aux abords de l'écran et qu'on s'y dirige (méthode la plus simple)
  - recentrer l'ensemble des objets, dont le vaisseau, avec un mouvement inverse lorsque le vaisseau s'arrête
  - idem, mais l'action de recentrage est déclenchée par l'appui d'une touche
  - afficher un sprite de flèche indiquant la direction où se trouve le vaisseau en dehors de l'écran
  - afficher une minimap qui localise le vaisseau hors de l'écran mais dans le jeu (donc le jeu à d'autres limites que l'écran)

- Ajouter une classe `Creature`. Cette classe se comporte de façon semblable sauf que les créatures ont leur mouvement propre. A vous de voir quel mouvement vous souhaitez implémenter. Vous pouvez utiliser la texture `Virus` fournie pour cette créature. Eventuellement, rajoutez un type énuméré qui décrit différentes créatures (ex: virus, parasite, bactérie ...) et donnez un type à la construction de votre créature. Selon son type, la créature aura des mouvements différents et une texture différente.

- Créez une classe `Projectile`. A vous de voir comment elle doit se comporter, de qui elle hérite ... Evidemment, l'idée est que l'appui d'une touche du clavier déclenche la production d'un projectile (ex: missile ...) par le vaisseau, ce missile pouvant avoir (selon son type) divers comportements comme se déplacer tout droit jusqu'à entrer en collision avec une créature, ou chercher les créatures, ou cibler une créature ... Cela implique que vous devrez gérer les collisions.

5 Réalisation du jeu principal

5.1 Conceptualisation

Réfléchissez au concept de jeu que vous souhaitez créer. Ecrivez sa fiche signalétique. Vous devrez produire une fiche signalétique complète comme montré dans le site Tutoriel **Ecrire un Game concept**. Créeé une première version complète que vous affinerez par la suite.

5.2 Modélisation UML

Ecrivez le diagramme de classes et le diagramme de séquence de votre jeu. Cette modélisation pourra se faire par incréments au fur et à mesure que votre réalisation évoluera. Néanmoins, vous devez, avant de commencer le code proprement dit, avoir modélisé dans les grandes lignes les composants de votre jeu et son fonctionnement.

5.3 Le reste ...

Le reste, c'est la programmation de votre jeu bien entendu. Nous avons vu ensemble à quel point vous êtes libres de réaliser le jeu de votre choix. Il y a cependant des contraintes fortes à respecter impérativement.

En plus de :

- la conceptualisation sous la forme de la fiche signalétique
- les diagrammes UML (séquence et classe)

votre jeu devra impérativement exploiter l'ensemble des technologies XML vues en cours, précisément (au minimum) :

- Fichiers XML pour
  - l'initialisation du jeu
  - le profil de joueur (l'humain)
  - la sauvegarde des parties
- Les Schémas XML correspondants
- Un programme C# (éventuellement séparé) de test des conformité et validité des documents XML
- Des feuilles de transformation XSLT pour
  - générer une page HTML présentant un joueur et ses parties
  - générer un fichier XML listant toutes les parties jouées dans un fichier ne stockant que le nom du joueur, la date de la partie et le score. Ce fichier XML pourra être utilisé pour retrouver une partie et accéder au fichier du joueur.
  - générer une page HTML présentant la liste des Hi-Scores (ce qui sous-entend que vous devez pouvoir enregistrer toutes les parties et stocker les meilleures)
- Une utilisation (une de chaque au moins) :
  - DOM + requêtes XPath
  - XmlReader
  - la sérialisation : votre sérialisation devra se faire sur des objets contenant au moins : beginitemize
  - des listes ou tableaux
  - des énumérations
  - des données en lecture seule et d'autres en lecture-écriture

Vous ne devez pas limiter leur usage à 1 ou 2 fichiers. Votre jeu doit exploiter à fond ces technologies. Vous pouvez les exploiter dans le jeu ou dans des programmes (ou projets) annexes comme par exemple un éditeur de niveaux.

6 Rendus

Les rendus seront obligatoirement :

- La fiche signalétique de votre jeu au format PDF exclusivement (tout autre format ne sera pas accepté). La fiche signalétique deva inclure le nom des auteurs.
- Un document expliquant où et dans quel cadre (pourquoi / quel usage) vous avez employé chacune des technologies demandées ci-dessus.
- Le projet C# incluant :
  - Le code source nettoyé
  - La documentation technique (documentation des classes et méthodes)
  - Les ressources graphiques, textuelles (dont fonts) et sonores
  - Les diagrammes UML de classe et de séquence du jeu
  - Une vidéo courte (moins de 2 minutes) de présentation de votre jeu
- le lien vers le dépot Github ouvert.

L'ensemble sera compressé dans **un seul fichier au format ZIP** (exclusivement ! pas de `.rar`, pas de `tar.gz` ...) dont le nom sera composé des noms (première lettre du prénom + nom) des auteurs, par exemple : `CFouard_ADemeure_ALagoutte_NGlade_.zip`

7 Conclusion

Vous êtes fier-e de votre jeu et de ce que vous avez entrepris pour y arriver ?!

C'est bien et ça peut vous donner envie d'aller plus loin comme *Lou Lubie* avec le projet *Rose in the woods* et sa *Petite Fabrique du Jeu Vidéo*.

En attendant, pensez à utiliser ce que vous aurez appris (méthodologie, organisation, formalisation, technos ...) dans votre projet intégrateur.