IPROG

JurENSiC World :

Rapport de projet

Réalisé par : Léa-Ava MAIRE et Jade LARIVAILLE, Noémie TCHOUA

Promo 2027 - 1A - Groupe 2

Décembre 2024

Sommaire :

I. Présentation du jeu

JurENSiC World est un jeu de plateau inspiré du célèbre film Jurassic World. On retrouve dans ce jeu les personnages clés du film comme Owen, Maisie, Blue et l'Indominus. Ils ont chacun leurs caractéristiques et ont donc tous un impact sur l'issue du jeu.

Le joueur incarne 2 personnages : Owen et Blue. Owen doit construire un enclos autour de l'IR en jetant des grenades au sol qui créent des crevasses de 2 cases que l'IR ne peut pas franchir. Cependant, Owen possède un nombre limité de grenades qui, lorsqu'elles explosent, la case qu'elles touchent ainsi qu'une case qui se trouve juste à côté qui est tirée aléatoirement disparaissent. Owen ne peut se déplacer que d'une case à la fois. Blue protège Owen et Maisie en faisant reculer l'IR de 3 cases (ou jusqu'à un obstacle) si elles se retrouvent sur la même case. Tout comme Owen, Blue ne peut se déplacer que d'une case à la fois. Maisie, paniquée, se déplace aléatoirement d'une case à la fois et ne sautera jamais sur l'IR. L'IR veut croquer tous les personnages et se déplace donc aléatoirement. Elle peut manger Owen et Maisie mais pas Blue. Elle ne craint pas les grenades, cela la blesse légèrement et l'énerve encore davantage. Le nombre de grenades que Owen possède est égal à la longueur du plateau que le joueur a choisi en début de partie.

Concernant la partie créative, nous avons ajouté plusieurs règles au jeu :

- Tous les personnages, sauf Owen, ont des points de vie qui sont initialisés au début de chaque partie. Maisie et Blue commencent avec 100 pdv. L'IR a un nombre de points de vie égale à 10 fois le nombre de grenades.
- Owen a la possibilité de récupérer deux grenades spéciales qui sont posées sur le plateau et que seul lui peut utiliser. Ces grenades créent des crevasses de 3 cases.
- Si Blue ou Maisie sont touchées par une grenade, leur espérance de vie est divisée par 2.
- Si l'IR est touchée par une grenade, elle perd 10 points de vie. Elle s'énerve et se déplace alors de 2 cases à la fois.
- Le joueur choisit entre 4 directions (Nord, Sud, Est, Ouest) où il veut que Blue fasse reculer l'IR.

La partie est donc gagnée si l'IR est enfermée seule dans l'enclos ou qu'elle n'a plus de points de vie. Cependant, le joueur perd la partie si Owen ou Maisie est mangé par l'IR, si Blue ou Maisie est touché 2 fois par une grenade et qu'elle n'a donc plus de points de vie, si Owen n'a plus de grenades alors que l'IR est encore en liberté.

I. Structure du code et choix de modélisation

Notre code se décompose en plusieurs sous-programmes que nous appelons dans le programme principal, chaque sous-programme ayant une fonction précise.

1. Les sous-programmes

Nous avons codé plusieurs sous-programmes selon les caractéristiques propres à chaque personnage et les différents états possibles du plateau de jeu.

- Déplacements

Nous avons deux sous-programmes pour les déplacements :

DeplacementAleatoire :

A chaque tour de jeu, **Maisie** et l'**IR** effectuent un déplacement aléatoire verticalement ou horizontalement. Ceci est exécuté par le sous-programme "DeplacementAleatoire" qui est une procédure prenant en paramètres le symbole du personnage à déplacer et ses coordonnées (passées par référence avec ref pour les mettre à jour après le déplacement). Nous utilisons deux variables locales dans ce sous-programme : "deplacementValide" (indicateur booléen permettant de valider le déplacement) et "rng" (générateur de nombres aléatoires pour choisir la direction).

La boucle principale est la boucle "do while" qui cherche une case accessible tant que le déplacement n'est pas valide (!deplacementValide). On commence par générer la direction aléatoire grâce à la méthode "Next". Une direction est aléatoirement choisie parmi : Ouest (0), Est (1), Nord (2), Sud (3). Les variables "deltaX" et "deltaY" définissent le déplacement relatif selon la direction obtenue. On utilise la boucle "switch case" pour traiter des quatre différents cas. Par exemple, si on obtient 1, le déplacement se fera vers l'Est donc la coordonnée x sera augmentée d'où deltaX = 1.

Les variables "tentativeX" et "tentativeY" correspondent aux potentielles nouvelles coordonnées du personnage. On les assigne puis on leur fait passer les conditions de validité. **Maisie** ne peut se déplacer que d'une case à la fois, d'où tentativeX += deltaX (parallèlement pour tentativeY). Le déplacement est validé uniquement si les nouvelles coordonnées restent dans les limites du plateau et que la case cible est vide (⬜).

Pour l'**IR** il y a deux cas possibles : soit elle n'est pas énervée dans ce cas elle ne peut se déplacer que d'une case à la fois, soit elle est énervée (si elle a été touchée par une grenade) dans ce cas elle peut se déplacer plus vite (deux cases à la fois). Si elle n'est pas énervée, la procédure est similaire à celle de **Maisie**. La seule différence est que l'**IR** peut tomber sur la case d'autres personnages pour les croquer (sauf **Blue** car elle est trop rapide). Le déplacement ne sera donc pas valide si la case cible est une crevasse (💥) ou **Blue** (🟦). Lorsque l'**IR** est énervée, elle tente de se déplacer jusqu'à 2 cases. La boucle "for" vérifie

chaque case traversée pour détecter des obstacles ou les limites du plateau. Si un obstacle est rencontré à la première ou à la deuxième case, l'**IR** s'arrête à la dernière case valide. Le déplacement est validé seulement si l'**IR** a bougé afin d'éviter un déplacement nul.

DeplacementClavier :

Blue et **Owen** effectuent des déplacements verticaux ou horizontaux de 1 case à la fois selon le choix du joueur. Pour cela, il peut utiliser les flèches du clavier (haut, bas, gauche, droite). Ceci est géré par le sous-programme "DeplacementClavier" qui est une procédure prenant en paramètres le personnage à déplacer, ses coordonnées et son nom. Comme précédemment, afin de gérer les différents cas d'erreurs, nous avons utilisé le booléen "deplacementValide" et la boucle "do while".

On commence par gérer les nouvelles coordonnées du personnage selon la flèche du clavier pressée grâce à "ControleKey". Si les nouvelles coordonnées dépassent les dimensions du plateau, alors le déplacement est impossible et on demande à l'utilisateur de presser une autre flèche.

Sinon, on regarde si la nouvelle case contient une grenade spéciale. Si le personnage est **Owen**, alors il récupère la grenade spéciale et son déplacement est validé. Si le personnage est **Blue**, le déplacement est validé mais la grenade spéciale sera écrasée (irrécupérable par Owen au prochain tour).

Sinon, si la nouvelle case est différente d'une case vide et différente de l'**IR** alors le déplacement n'est pas valide quel que soit le personnage à déplacer Sinon, le déplacement est valide : la position du joueur est mise à jour et on sort de la boucle "do while".

En effet, **Owen** et **Blue** ne peuvent pas aller sur une case où il y a déjà un autre joueur sauf si c'est l'IR. Si **Owen** tombe sur l'**IR** par accident alors la partie sera finie. Par contre si **Blue** va sur l'**IR** alors elle pourra utiliser son pouvoir (en appelant le sous-programme "PouvoirBlue"). Ils ne peuvent également pas aller sur une crevasse.

- Grenades

Grenade :

La procédure "Grenade" permet de gérer tout ce qui concerne le lancer de grenades par **Owen** à chaque tour. Les variables "coorXGrenade" et "coorYGrenade" correspondent aux coordonnées de l'impact de la grenade lancée. Elles sont initialisées à des valeurs bien supérieures aux dimensions du plateau pour éviter tout problème d'assignement des valeurs. Ainsi, le programme commence par demander au joueur s'il souhaite lancer une grenade. De la gestion de cas d'erreur est implantée ici afin de ne pouvoir répondre que par oui ou non grâce à la méthode "Enum.TryParse" qui vérifie que ce qui est entré dans la console correspond bien aux 4 choix possibles : "oui", "Oui", "Non, et "non". Si la réponse est Non, on affiche sur la console qu'**Owen** n'a effectivement pas lancé de Grenade. Si la

réponse est oui, on demande alors s'il souhaite lancer une grenade normale ou spéciale. Ici aussi la méthode "Enum.TryParse" est utilisée pour n'autoriser que "s", "S", "n", ou "N" (S pour spéciale, N pour normale).

Peu importe le choix, on vérifie qu'il reste effectivement des grenades de ce type, et si oui on enlève 1 au compteur. On fait alors appel au sous-programme "SelectionCoordonneesGrenade" pour obtenir les coordonnées auxquelles le joueur souhaite lancer la grenade en vérifiant à chaque fois que ce n'est ni en dehors des limites du plateau, ni au-delà de la portée d'**Owen** qui est de 3 cases. Si les coordonnées remplissent ces conditions, on regarde si elle est tombée sur l'**IR**. Si c'est le cas, on enlève 10 PV à l'IR et on vérifie que ceux-ci ne sont pas à 0, auquel cas la partie serait terminée. On affiche alors les PV restants et on passe le booléen "enervement" à *true*, l'**IR** se déplacera de 2 cases à partir du prochain tour. Si la grenade tombe sur **Blue** ou **Maisie**, cela leur enlève la moitié de leurs PV, grâce à la procédure "SystemePV".

Si jamais la grenade ne touche aucun autre personnage, on crée une crevasse. La création de crevasses utilise de l'aléatoire pour générer un nombre aléatoire compris entre -1 et 1 pour chaque paramètre de coordonnée (x et y). Tout ceci est dans une boucle "while" qui vérifie que ces nombres aléatoires ne sont pas tous les deux nuls pour assurer le fait de créer un impact, on vérifie également que cela ne sorte pas des limites du plateau. On crée **2** crevasses pour une **grenade normale** et **3** crevasses pour une **grenade spéciale**. Ainsi dans le cas de la grenade spéciale, on reprend le même fonctionnement en partant du second impact pour déterminer la position du 3ème. Enfin, on regarde à nouveau si **Blue** et **Maisie** ont été touchées et on fait appel à SystemePV si c'est le cas, ce qui ne créera pas de crevasse à l'endroit de l'impact si c'est le cas.
Pour terminer, on réaffiche le plateau et on informe le joueur du nombre de grenades de chaque type qu'il lui reste.

SelectionCoordonneesGrenade :

La procédure "SelectionCoordonneesGrenade" sert à demander sur quelle case le joueur souhaite lancer une grenade. Il doit donc entrer le numéro de la ligne puis la colonne qui sont sauvegardés dans les variables "coorYGrenade" et "coorXGrenade". Cette procédure est utilisée dans le sous-programme "Grenade". Ce sous-programme inclut également de la gestion de cas d'erreur, l'utilisateur n'a ainsi que la possibilité d'entrer des entiers.

- ## Les pouvoirs

Croquer :

L'**IR** mange quiconque se trouve sur son passage, pour cela nous avons créé la procédure "Croquer" qui prend comme paramètres les coordonnés de l'**IR**, de **Maisie** et d'**Owen**. Cette procédure se divise en deux parties : une pour manger Maisie et une pour

manger Owen. Dans chacune d'elles on vérifie si les coordonnés Y et X des personnages sont les mêmes que celles de l'**IR** avec la condition "if". Si cette condition est vérifiée la forme représentant le personnage est remplacée par celle de l'**IR** (plateau[positionYMaisie, positionXMaisie] = "🟥"). Le joueur est alors informé que le personnage a été mangé. Enfin, le booléen "finCroc" prend la valeur *true* pour sortir de la boucle "while" du sous-programme "Jeu" (décrit plus tard).

PouvoirBlue :

Blue dispose d'un pouvoir stratégique qui permet de protéger **Owen** et **Maisie** en éloignant l'**IR** jusqu'à 3 cases si ce dernier se trouve sur la même position qu'elle. Pour implémenter cette mécanique, nous avons créé la procédure "PouvoirBlue", qui prend en paramètres les coordonnées X et Y de l'**IR** (positionXIR et positionYIR) et qui permet de déplacer l'**IR** dans l'une des directions disponibles : Nord, Sud, Est ou Ouest.

Premièrement, la case actuelle où se trouve l'**IR** est réinitialisée à "⬜", qui symbolise une case vide. On déclare ensuite quelques variables locales au sous-programme : "directionValide" qui est un indicateur pour vérifier si l'utilisateur a saisi une direction valide, "deltaX" et "deltaY" qui définissent les déplacements relatifs sur les axes X (colonnes) et Y (lignes), et enfin "tentativeX" et "tentativeY" qui sont des variables temporaires pour calculer les nouvelles positions de l'**IR** avant validation.

Ensuite, on passe dans une boucle "do while" afin de vérifier la validité de la direction choisie par l'utilisateur et d'assigner le déplacement relatif (deltaX ou deltaY). Pour chacune des quatres directions possibles, on utilise la méthode "Equals" pour vérifier si la direction entrée par l'utilisateur est égale à l'une des directions possibles (Ouest, Est, Nord ou Sud). Le paramètre "StringComparison.OrdinalIgnoreCase" indique que la comparaison doit être **insensible à la casse**, ce qui signifie que "ouest", "OUEST" ou "OuEsT" seront considérés comme égaux à "Ouest". On vérifie également que le déplacement ne soit pas nul. Par exemple, pour pouvoir être déplacé vers l'Ouest (vers la gauche), il faut diminuer la coordonnée X (d'où deltaX = -1), donc l'**IR** ne doit pas être initialement positionné sur la première colonne du plateau (d'où la condition : positionXIR != 0). Si la saisie est invalide ou que le déplacement est impossible (bordure du plateau), un message d'erreur est affiché, et l'utilisateur doit ressaisir une direction.

Une fois la direction validée, on passe au calcul du déplacement. Une boucle "for" est utilisée car le nombre d'itérations est connu à l'avance (3 ici). En effet, on teste les positions successives en calculant tentativeX et tentativeY jusqu'à un maximum de 3 cases. Les positions sont multipliées par "i" pour simuler un **déplacement progressif** (1, 2, 3 cases). Le déplacement de l'**IR** est conditionné par les limites du plateau et les crevasses "💥". Si une limite ou un obstacle est rencontré, les coordonnées sont ajustées à la **dernière position valide** (i - 1). La boucle est forcée de se terminer en définissant i = 4.

Enfin, les coordonnées finales de l'**IR** sont mises à jour après validation puis elle est placée sur sa nouvelle position.

- Le plateau

Taille du plateau :

La taille du plateau est demandée à l'utilisateur à chaque début de partie via les fonctions "DemanderHauteurPlateau" et "DemanderLongueurPlateau". Elles sont construites de la même manière, mais l'une renvoie un entier correspondant à la **hauteur** du plateau et l'autre renvoie un entier correspondant à la **longueur** du plateau. A l'intérieur de chacune de ces fonctions, la méthode "TryParse" est utilisée afin de vérifier que l'utilisateur entre un entier. Cette méthode retourne un **booléen** que l'on stocke dans la variable "saisieValide". Tant que la saisie n'est pas valide (!saisieValide), on redemande à l'utilisateur d'entrer un entier.

Création du plateau :

A chaque début de partie, le plateau doit être initialisé selon :

- la **taille** (nombre de lignes et de colonnes)
- les **cases vides**
- le **placement aléatoire** des personnages et des grenades

Ces fonctionnalités sont regroupées dans le sous-programme "CreerPlateau" qui est une fonction renvoyant une matrice de chaînes de caractères (string) et qui prend en paramètre deux entiers : "dim1" et "dim2" correspondant aux dimensions du tableau. Les arguments sont demandés à l'utilisateur en amont dans les fonctions "DemanderHauteurPlateau" et "DemanderLongueurPlateau".

Au début de ce sous-programme, la matrice de string "plateau" est déclarée, c'est la matrice qui sera renvoyée à la fin du sous-programme. Puis il est initialisé avec des carrés blancs ("⬜") représentant les cases vides. Au départ, l'idée était de manipuler une matrice contenant des variables de type char, cependant les caractères spéciaux utilisés pour représenter nos personnages et nos grenades sont en fait composés de plusieurs éléments d'affichage indépendants (plusieurs char), d'où l'utilisation d'une matrice de string.

Ensuite, on fait appel à un autre sous-programme nommé "PlacerAleatoire" créé pour placer le personnage de son choix dans un plateau, les deux étant passés en paramètre. De plus, dans ce sous-programme on fait aussi appel à un autre sous-programme nommé "TirerNbAleatoire" qui prend en paramètre la borne supérieure (max) de l'intervalle dans lequel le nombre entier aléatoire est tiré grâce à la méthode Next(0,max). Si on revient au sous-programme "PlacerAleatoire", on fait appel à "TirerNbAleatoire" qui tire une coordonnée x (numéro de colonne) et y (numéro de ligne) au hasard entre **0** et respectivement le **nombre de colonnes du plateau - 1** et **le nombre de lignes du plateau - 1**. Ceci est répété tant que la case sur laquelle on tombe n'est pas vide (afin d'éviter que plusieurs personnages se retrouvent sur la même case). Enfin la fonction renvoie le plateau modifié avec la nouvelle position du personnage passé en paramètre.

Revenons-en au sous-programme “CréerPlateau”, le sous-programme “PlacerAléatoire” est appelé pour positionner aléatoirement chacun des personnages sur notre plateau. **Owen** est représenté par le carré vert "🟩", **Blue** par le carré bleu "🟦", **Maisie** par le carré violet "🟪" et l’**Indominus Rex** par un carré rouge "🟥". Puis deux grenades spéciales sont placées aléatoirement et sont représentées par “🧨”.

Récupérer les coordonnées :

Au début de chaque partie et après la création du plateau, il est nécessaire de récupérer les coordonnées des personnages. Pour cela, nous avons créé la procédure “RecupererCoord” pour mettre à jour les coordonnées des personnages en parcourant le plateau.

Nous utilisons deux boucles “for” imbriquées : la première parcourt les **lignes** du plateau (i) et la deuxième parcours les **colonnes** (j). Pour chaque case du plateau, le programme compare le contenu de la case (plateau[i,j]) avec les symboles des personnages. Lorsqu’un personnage est identifié, ses coordonnées sont mises à jour dans les variables correspondantes.

Le but de cette procédure est d’examiner l’état actuel du plateau et de récupérer les coordonnées X (colonnes) et Y (lignes) de chaque personnage en fonction de leurs symboles respectifs. Ces coordonnées sont ensuite stockées dans des variables passées par **référence**, afin qu’elles puissent être utilisées directement dans d’autres parties du programme.

Affichage du plateau :

La procédure “AfficherPlateau” est responsable de l’affichage graphique du plateau de jeu dans la console. Le sous-programme prend en paramètre la matrice de string “plateau”, elle n’a pas besoin d'être passée en référence pour que ses modifications apparaissent dans l'entièreté du programme.

L’objectif est de mettre à jour les positions des personnages et d’afficher le plateau dans un format **lisible** avec des indices de ligne et de colonne. Premièrement, les coordonnées des personnages (passées par référence et gérées ailleurs dans le programme) sont utilisées pour assigner leurs symboles aux bonnes positions dans le tableau. Cette étape garantit que l’affichage du plateau reflète les **positions les plus récentes** des personnages.

Ensuite, pour l’affichage des indices de **colonnes**, on commence par ajouter un espace initial pour aligner correctement les indices des colonnes avec les cases du plateau. La boucle “for” parcourt les colonnes du plateau (de 0 à plateau.GetLength(1) - 1) et affiche les indices (t) en les séparant par des espaces. Un retour à la ligne (Console.WriteLine()) est ajouté pour séparer les indices des colonnes de l’affichage du contenu du plateau.

L’affichage des numéros de **lignes** est effectué en même temps que l’affichage du contenu du tableau. En effet, à chaque itération de la boucle extérieure (par ligne), l’indice

de la ligne actuelle (i) est affiché. La boucle intérieure parcourt les colonnes de chaque ligne et affiche le contenu de chaque case (Console.Write(plateau[i, j])). Après avoir parcouru toutes les colonnes d'une ligne, on retourne à la ligne (Console.WriteLine()) puis on passe à l'affichage de la ligne suivante du plateau.

- **L'enclos**

Pour gagner le jeu, un des moyens est d'enfermer l'**IR** toute seule dans un enclos constitué de crevasses ou de crevasses mêlées aux limites du plateau. Pour ce faire, deux sous programmes sont utilisés.

TesterEnclos :

Le sous-programme "TesterEnclos" permet de vérifier la présence d'un enclos, et s'il y en a un, de regarder si un autre personnage se trouve à l'intérieur, ce qui conditionne l'issue de la partie. En effet, si l'**IR** est enfermée toute seule, la partie est remportée, sinon c'est perdu. Ainsi pour ce sous-programme on crée une matrice "CasesVisitees" qui va permettre de stocker à l'aide de **booléens** les cases explorées ou non par le sous-programme récursif "RechercherAutour" que l'on détaillera plus bas. Ce qui est important de noter pour l'instant est que les **cases visitées** sont représentées par le booléen *true* dans cette matrice, sinon leur statut est *false*. Ainsi, une fois la récursivité terminée, on regarde si parmi les cases visitées et donc du potentiel enclos, il y a la case où se trouve **Owen**. Si c'est le cas, alors on considère qu'il n'y a pas d'enclos et le programme s'arrête là grâce à return.

A l'inverse, si la case d'**Owen** n'est pas dans les cases visitées, le programme ne rentre pas dans la conditionnelle et celui-ci continue. On initialise un booléen "IRSeule" comme vrai, on part donc du principe qu'elle est seule. Puis, comme pour Owen, on regarde si **Maisie** ou **Blue** se trouvent dans l'enclos (les CasesVisitees = *true*), si c'est le cas, on écrit un message pour en informer le joueur et lui indiquer que la partie est perdue et on passe les booléens "IRSeule" à faux et "finEnclos" à vrai. Cela permet respectivement d'empêcher d'écrire que l'IR est seule et que la partie est gagnée ensuite, et de terminer la partie dans la boucle principale.

Si aucun des deux personnages n'est enfermé, alors "IRSeule" reste faux et on prévient le joueur qu'il a gagné la partie, on passe également le booléen "finEnclos" à vrai pour que la partie se termine.

RechercherAutour :

Le sous-programme "RechercherAutour" est une procédure **récursive** qui permet d'explorer une à une les cases du plateau autour d'une case donnée. On commence à la position de l'**IR** et on évolue à partir de là.

Ce programme regarde si la case qu'il vérifie est hors des limites du plateau, déjà visitée ou une crevasse, auquel cas il l'ignore grâce à "return" et ne recherche pas parmi les cases autour de celle-ci. Si aucune des conditions ci-dessus n'est remplie, alors on marque la case comme visitée dans la matrice "CasesVisitees" en la passant comme vraie. On appelle ensuite la procédure pour les 4 directions autour de cette case (haut, bas, gauche et droite).

Le sous programme s'arrête lorsque toutes les cases autour d'une case remplissent au moins une des conditions au-dessus. C'est-à-dire si elles sont toutes visitées ou des crevasses ou hors des limites du plateau. On a ainsi une matrice des cases contenues dans l'enclos s'il y en a un. Sinon, elle contient toutes les cases du plateau, dont celle d'**Owen**, et ainsi aucun enclos ne sera détecté.

- Le système de points de vie (PV)

Tous les personnages du jeu sauf **Owen** sont soumis à un système de points de vie. Le nombre de points de vie de l'**IR** est déterminé par le nombre de grenades. Ainsi, il faut que le joueur fasse atterrir toutes ses grenades normales sur l'**IR** pour que celle-ci soit tuée et remporter la partie autrement qu'en créant un enclos.

Concernant **Blue** et **Maisie**, elles peuvent survivre à un seul impact de grenade, si cela arrive une seconde fois, elles sont tuées par **Owen** et la partie se termine. Leur PV sont gérés par la procédure "SystemePV".

2. Le sous-programme principal

Le sous-programme principal "Jeu" fait appel aux différents sous-programmes dans un ordre précis et avec des conditions spécifiques.

- La partie

Une partie se compose d'une succession de tours générées par une boucle "while" qui s'arrête dès qu'une des conditions de fin de partie est vérifiée (l'**IR** croque **Owen** ou **Maisie**, l'**IR** n'a plus de points de vie, l'**IR** est enfermé seul dans l'enclos, **Maise** est tuée par une grenade).

Tout d'abord, ces conditions sont initialisées par un **booléen** (finCroc = false, finGrenade = false, finEnclos = false) puis sont vérifiées tout au long de la partie à l'intérieur des sous-programmes (Croquer, Grenade, Enclos).

- Le tour de jeu (appel des fonctions/procédures)

Chaque tour de jeu (procédure "Jeu") se déroule dans l'ordre suivant :

Maisie est le premier personnage à se déplacer. Elle se déplace de manière aléatoire (appel de la procédure DeplacementAleatoire). Si elle atterrit sur la même case que l'IR elle se fait manger (appel procédure Croquer) et la partie s'arrête : if (finCroc) return.

Ensuite, l'**IR** se déplace aléatoirement sur le plateau mais s'il a été touché par une grenade dans le tour précédent, il sera alors énervé et pourra se déplacer plus vite (appel de la procédure DeplacementAleatoire). De plus, si l'IR se trouve sur la même position qu'un personnage (Owen ou Maisie) il le croque automatiquement (appel de la procédure Croquer) et la partie s'arrête : if (finCroc) return.

Ensuite le joueur déplace **Blue** (appel de la procédure DeplacementClavier). Blue peut protéger Owen et Maisie, en effet, si elle se retrouve sur la même case que l'IR elle le fait reculer (appel de la procédure PouvoirBlue). Cependant, si en reculant l'IR se retrouve sur la même case que Owen ou Maisie il peut les manger (appel de la procédure Croquer) et la partie s'arrête : if (finCroc) return.

Le joueur déplace ensuite **Owen** (appel de la procédure DeplacementClavier). S'il arrive sur la même case que l'IR alors il se fait croquer (appel de la procédure Croquer) et la partie s'arrête. S'il le veut, Owen peut lancer une grenade normale ou spéciale (appel de la procédure Grenade). Si l'IR est touchée par une grenade il perd des points de vie et arrivé à zéro il meurt. Cependant, Maisie ou Blue peuvent subir le même sort et donc mourir si elles n'ont plus de points de vie, la partie s'arrête alors : if (finGrenade || finPv) return.

Enfin, à la fin de chaque tour nous vérifions l'existence d'un enclos grâce au sous-programme TesterEnclos.

3. Le programme principal

Le programme principal commence avec l'affichage de la légende et l'initialisation des variables. Il permet de faire plusieurs parties à la suite grâce à une boucle "do while".

Le jeu commence si le joueur clique sur la touche "**Entrer**" du clavier (appel procédure Jeu), on utilise la condition "if" avec la comparaison "key.key == ConsoleKey.Enter" qui vérifie que la touche cliquée est bien "Entrer".

Quand la partie est finie, si le joueur le souhaite, il peut rejouer en cliquant sur la touche "Enter". S'il clique sur cette touche, le plateau est alors réinitialisé et le joueur peut changer la taille du plateau : plateau = CréerPlateau(HauteurPlateau(), LongueurPlateau()). En cliquant de nouveau sur cette touche il peut commencer la nouvelle partie. Il est important de noter qu'à chaque début de partie les conditions de fin de partie, les pdv des personnages et le nombre de grenades spéciales sont réinitialisés.

II. Organisation de l'équipe

1. Planning de réalisation et répartition des tâches

Pour mener à bien ce projet nous avons tout d'abord établi un cahier des charges pour définir les fonctionnalités que nous voulions intégrer au jeu et identifier tous les sous-programmes dont nous aurions besoin. Ensuite, nous nous sommes répartis les tâches, chacune d'entre nous travaillait sur un sous-programme précis et en cas de blocage nous mettions en commun pour résoudre le problème. Cette méthode nous a permis d'avancer rapidement et d'éviter de stagner sur une partie du code en particulier. Une fois les sous-programmes quasiment fonctionnels nous avons pu commencer à coder le sous-programme principal (tours de jeu) et enfin le programme principal (parties). Néanmoins, des modifications dans les sous-programmes étaient constamment réalisées afin d'améliorer notre jeu.

La majorité du travail était effectué en dehors des créneaux de TD qui nous servaient principalement à poser des questions au professeur et faire un point sur notre avancée. Nous codions donc sur notre temps libre, sans suivre de planning prédéfini, mais de manière régulière.

2. Matrice d'implication

Voir le fichier Excel.

III. Bilan

1. Difficultés rencontrées

Au cours de ce projet nous avons rencontré trois difficultés majeures : la gestion des sorties du plateau, des cas d'erreur et de l'enclos.

En effet, pour les sous-programmes de déplacement nous devions vérifier que les personnages ne puissent ni sortir du plateau ni aller sur une crevasse ou sur la même case qu'un autre personnage (sauf pour l'IR et Blue). Nous avions également un problème similaire avec les grenades car Owen ne pouvait pas en jeter une qui sorte du plateau. Il fallait donc que nous vérifions les coordonnées des crevasses créées par ces grenades. Nous devions aussi gérer les fautes de frappe lorsque le joueur est amené à saisir des coordonnées ou entrer une réponse.

Concernant l'enclos, nous avions réfléchi à plusieurs méthodes avant de trouver celle que nous avons utilisé dans le jeu. La première a vite été abandonnée puisqu'elle ne

fonctionnait qu'en cas d'enclos carré ou rectangulaire. La deuxième, s'appuyait sur la technique de la "main droite" utilisée à la base pour sortir d'un labyrinthe, l'objectif étant donc de longer les parois de l'enclos jusqu'à revenir à la position de départ. Finalement, nous avons opté pour une autre méthode utilisant une procédure récursive. Elle consiste à explorer une à une les cases autour de la position de l'IR.

2. Compétences acquises

Grâce à la réalisation de ce projet, nous avons progressé sur plusieurs points essentiels.

Nous avons amélioré notre maîtrise de la manipulation de tableaux, notamment les matrices, utilisées pour gérer le plateau de jeu. En effet, il a fallu à de nombreuses reprises parcourir le plateau afin de suivre les positions des personnages et effectuer les vérifications nécessaires.

L'utilisation de plusieurs sous-programmes dans le jeu nous a permis de bien différencier les procédures, qui exécutent des actions comme les déplacements, et les fonctions, qui retournent des résultats.

Nous avons renforcé notre compréhension des structures itératives, en utilisant des boucles comme "while", "do...while" et "for" pour parcourir le plateau, répéter des actions et vérifier les conditions jusqu'à ce qu'elles soient valides.

La gestion des erreurs de saisie nous a permis de découvrir et de manipuler de nouveaux outils, comme "TryParse" pour valider les entrées numériques, et "Equals" pour comparer des chaînes de caractères ; tout en rendant notre programme plus robuste.

En travaillant avec GitHub, nous avons appris à collaborer efficacement en équipe sur un même code, à gérer les versions et à résoudre les conflits éventuels. Cela a renforcé nos compétences en gestion de projet, notamment en organisant le travail et en respectant les étapes nécessaires à la finalisation du jeu dans le temps imparti.

Ainsi, ce projet nous a permis de consolider nos bases tout en acquérant des compétences directement applicables à des projets futurs.