Projet de Programmation C : Tower Defense

L'objectif de ce projet est de développer une application graphique d'un jeu de *tower defense*. Il s'agit typiquement d'un jeu d'action-stratégie, dans lequel le joueur doit maintenir un système de tours pour protéger sa base contre des vagues d'ennemis.

Figure 1: Capture d'écran de *Gemcraft Chapter One: The Forgotten*, qui inspire les mécanismes décrits dans ce sujet.

1 Description et déroulement du programme

Le jeu se déroule sur une grille de taille 28×22, où il y a un chemin composé des cases qui part du nid des monstres au camp du joueur. Il y aura des vagues de monstres qui sortent de leur nid pour attaquer le camp. Le but du joueur est de se défendre contre ces vagues de monstres avec des tours, où le joueur peut mettre une gemme magique qui tire sur les monstres automatiquement.

Pour générer les gemmes magique, le joueur se dispose d'une certaine quantité de *mana* au début, et on peut aussi récupérer de mana à chaque défaite de monstre. Cependant, quand un monstre rentre dans le camp, une quantité de mana est nécessaire pour le bannir. Quand le joueur n'a plus de mana, il est défait et le jeu termine. Le score final affiché sera constitué du nombre de vagues endurées et de la quantité de dégât généré.

2 Mécanisme du jeu

Le jeu se passe sur un terrain d'une grille de taille 28×22. Sur la grille, le chemin est représenté par un ensemble de cases qui relie la case représentant le nid des monstres à celle représentant le camp du joueur. Les autres cases sont considérées comme vide. Les monstres se déplacent au centre du chemin de façon **continue**. Dans la suite, l'unité de distance sera la longueur d'un côté d'une case de la grille. Dans la description du mécanisme, les valeurs calculées par des formules doivent être reconverties en entier.

2.1 Terrain et vagues de monstres

Les monstres arrivent en vagues, qui aura lieu tous les 35 secondes, marchant de leur nid au camp du joueur. La première vague est toujours déclenchée par le joueur. Le joueur peut aussi déclencher la prochaine vague à tout moment. Dans ce cas là, le joueur gagnera un quantité de mana qui s'élève à t% de la capacité maximale courante de la réserve, où t est le temps restant avant la prochaine vague. Dès qu'une vague est déclenchée, la prochaine vague arrivera dans 35 secondes.

Il y a quatre types de vagues de monstres :

- Vague normale : 12 monstres, vitesse d'une case par seconde ;
- Vague de foule : 24 monstres, vitesse d'une case par seconde ;
- Vague agile : 12 monstres, vitesse de deux cases par seconde ;
- Vague de boss : 2 monstres, vitesse d'une case par seconde.

Un type est tiré au hasard pour chaque vague déclenchée, avec la probabilité 50% pour une vague normale, 20% pour une vague de foule, 20% pour une vague agile, et 10% pour une vague de boss. Pourtant une vague boss n'arrive jamais parmi les cinq premières vagues. Pour les monstres dans la i-ième vague, leur HP initial est donné par $h \times 1.2^i$, où h est une constante **dont la valeur est laissée à votre choix**. Cette valeur influence fortement la dynamique du jeu, donc à calibrer soigneusement. Cependant, les monstres dans une vague de boss font une exception, possédant un HP initial $12 \times h \times 1.2^i$.

2.2 Gestion de mana

Le joueur se défend contre les vagues de monstres avec les gemmes dans les tours. Leur création et d'autres actions se fassent en consumant de mana. La gestion de mana est alors crucial pour la survie.

Le joueur possède initialement d'une réserve de mana de niveau 0 de maximum 2000, avec le niveau initial de mana à 150. Le maximum de la réserve augmente avec son niveau, et sera 2000×1.4^n au niveau n. Le joueur peut augmenter le niveau de la réserve en sacrifiant une quantité de mana de 25% du maximum courant, donc 500×1.4^n pour passer du niveau n au niveau $n+1$.

Le joueur peut récupérer de mana en battant les monstres. Pour chaque démise d'un monstre, 10% de son HP initial multiplié par 1.3^n (avec n le niveau de la réserve) est converti en mana ajouté à la réserve, mais tout mana dépassant le maximum sera gaspillé. Cependant, si le monstre arrive au camp, pour protéger le joueur, le monstre sera banni et retourné au nid automatiquement, mais au prix de consommer une quantité de mana égale à 15% de son HP initial multiplié par la même facteur 1.3^n. S'il n'y a pas assez de mana, le monstre pénètre dans le camp, et la partie termine.

2.3 Gemmes, teintes et éléments

Le joueur peut bâtir des tours sur les cases vides. Cependant, il est impossible de détruire une tour déjà bâtie. Pour le coût de construction, trois tours sont offertes, la première tour payant coûtera 100 mana, et ce coût double pour chaque nouvelle tour.

Une tour ne tire pas aux monstres toute seule. C'est les *gemmes* qui tirent. Une gemme doit être placée dans une tour pour tirer aux monstres. Cependant, pour chaque placement ou remplacement, la gemme prendra 2 secondes pour se charger. Après cette période de chargement, la gemme tire avec une fréquence maximale de 2 tirs par secondes. La gemme tire quand il y a au moins un monstre dans un rayon de 3 cases, et elle tire toujours au monstre avec le plus d'HP dans ce rayon.

Chaque gemme et chaque monstre a une *teinte* (couleur), qui est un entier entre 0 et 359 (compris). Pour obtenir la représentation RGB d'une teinte, vous pouvez se référer à ce page Wikipédia. On peut aussi voir une teinte comme un angle. Ce point de vue sera utile dans la suite. La teinte d'un monstre est tirée aléatoirement, mais la teinte d'une gemme est plus compliquée.

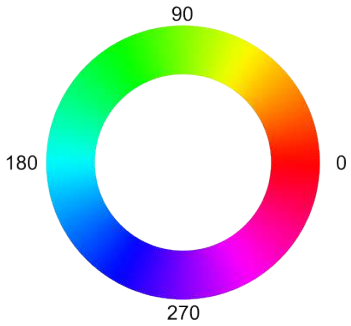


Figure 2: Les teintes et leur couleurs associées (Wikipedia)

Il y a deux types de gemmes : les *gemmes pures*, qui possèdent des effets spéciaux, et les *gemmes mixtes*, qui peuvent générer plus de dégâts. Chaque gemme possède un *niveau*, qui influence le dégât généré par un tir. Supposons qu'on a une gemme de niveau n et de teinte t_g qui a fait un tir à un monstre de teinte t_m. Le dégât de base généré à l'atterrissage par ce tir est

$$d \times 2^n \times (1 - \cos(t_g - t_m)/2).$$

Ici, t_g et t_m sont pris comme degré (entre 0 et 360). On observe que le dégât est exponentiel au niveau de la gemme, et plus les teintes de la gemme et du monstre est

proche, moins le tir va générer de dégât. Ici, d est une constante **dont la valeur est laissée à votre choix**. Cette valeur influence fortement la dynamique du jeu, donc à calibrer soigneusement.

Il y a 3 types de gemmes pures : rouge, verte et bleue. Le type d'une gemme pure est déterminé par sa teinte, et ce type détermine l'effet d'élément associé à leur tirs. Voici une liste précise de l'intervalle de teinte et l'élément associé de chaque type :

- *Rouge*, teinte 0–30 et 330–359, élément *Pyro* ;
- *Verte*, teinte 90–150, élément *Dendro* ;
- *Bleue*, teinte 210–270, élément *Hydro*.

En un mot, modulo 360, les gemmes rouges (vertes et bleues, respectivement) ont leur teinte dans une zone de taille 60 centrée à la teinte 0 (120 et 240 respectivement).

Le joueur ne peut créer que les gemmes pures, avec un coût 100×2^n pour une gemme au niveau n. Lors de la création d'une gemme pure, on tire d'abord son type, puis une teinte uniformément dans l'intervalle de teinte associé au type.

Autre que créer les gemmes pures, le joueur peut aussi fusionner deux gemmes **du même niveau** n, au prix de 100 mana. La nouvelle gemme sera de niveau $n + 1$, avec la teinte au point médian des teintes des deux gemmes fusionnées. Si les gemmes fusionnées sont pures et de même type, alors la nouvelle gemme sera aussi pure. Sinon, elle sera une gemme mixte, qui ne possède pas d'effet d'élément, mais ses tirs génèrent un dégât doublé, avec un chance de 10% de doubler encore ce dégât.

2.4 Effets des éléments

Les tirs d'une gemme pure associée à un élément possèdent des effets spéciaux.

- Élément *Pyro* pour les gemmes rouges : effet d'éclaboussure (générer un dégât de 15% du dégât de base aux monstres dans un rayon de 2 cases du point d'atterrissage d'un tir, en prenant la teinte en compte).
- Élément *Dendro* pour les gemmes vertes : effet de parasite (générer un dégât de 2.5% du dégât de base tous les 0.5 secondes du monstre pendant 10 secondes).
- Élément *Hydro* pour les gemmes bleues : effet de ralentissement (diviser la vitesse par 1.5 pendant 10 secondes).

Un tir d'une gemme pure donne aussi au monstre frappé un *résidu élémentaire*, qui peut réagir avec un autre tir pour créer des effets supplémentaires. Quand un tir d'une gemme pure frappe un monstre sans résidu, l'élément associé s'attache au monstre comme résidu élémentaire. Cependant, quand le monstre vient avec un résidu élémentaire du même type que le tir, alors l'effet spécial associé sera appliqué, avec le chronomètre réinitialisé (pour *Hydro* et *Dendro*). Si le tir possède un autre élément que le résidu élémentaire, le résidu élémentaire rentrera en réaction élémentaire et sera éliminé, produisant un effet de réaction qui remplace l'effet spécial du tir :

- *Pyro* + *Hydro* (vaporisation) : générer un dégât de 5% aux monstres dans un rayon de 3.5 cases du point d'atterrissage, et diviser aussi leur vitesse par 1.25 pendant 5 secondes.
- *Pyro* + *Dendro* (combustion) : tripler le dégât du tir courant.
- *Dendro* + *Hydro* (enracinement) : immobiliser le monstre pendant 3 secondes.

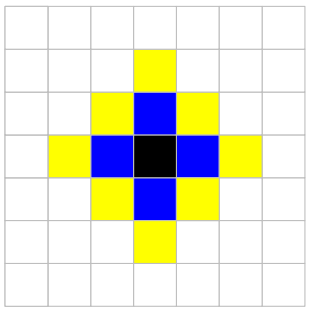
On conclut avec une liste des actions possibles du joueur et leurs coûts en mana :

- Augmenter le niveau de la réserve de mana de n à $n+1$: 500×1.4^n.
- Bâtir une tour sur une case vide : 0 pour les trois premières, puis $100 \times 2^{k-4}$ pour la k-ième tour.
- Générer une gemme pure de niveau n : 100×2^n.
- Fusionner deux gemmes : 100.

3 Génération de chemin

Le chemin sera généré aléatoirement pour chaque partie. Pour que le chemin soit intéressant pour le jeu, dans cette section on propose un algorithme de génération. Vous pouvez implanter votre propre algorithme, mais il faut absolument argumenter et le documenter précisément dans le rapport et dans les commentaires. Sinon, **toute déviation sera perçue comme une faute d'implantation**.

Pour commencer, on définit la *distance de Manhattan* de deux cases (i_1, j_1) et (i_2, j_2) par $|i_1 - i_2| + |j_1 - j_2|$. Dans la figure suivante, les cases bleues sont à distance 1 de la case noire, tandis que les cases jaunes sont à distance 2.



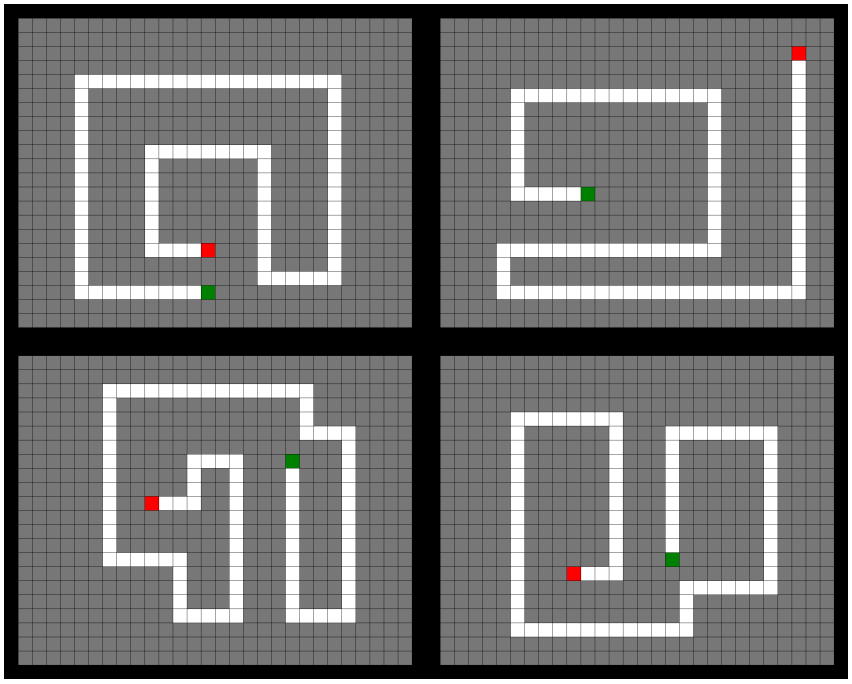
L'idée est d'étendre le chemin sans trop rapprocher une autre partie du chemin. Partant d'une case au bout de chemin en cours de construction, on dit que son *étendu* dans une direction cardinale donnée est le nombre maximal de cases consécutives dans cette direction sans rentrer en distance au plus 2 aux autres cases de chemin ou à l'extérieur de la grille, sauf la case courante et celle qui le précède.

Commençant par le terrain vide, l'algorithme se déroule de manière suivante :

1. Initialiser la grille à vide.

2. Choisir une case aléatoire avec une distance au moins 3 aux bords comme le nid des monstres.
3. Calculer les étendus pour toute direction cardinale, puis choisir aléatoirement la direction initiale avec une probabilité proportionnelle à l'étendu de chacune.
4. Si l'étendu de la direction courante est plus petit ou égal à 2, alors aller à l'étape 7. Sinon, supposons que l'étendu est n. Tirer n valeurs aléatoires indépendamment, chacune vaut 1 avec probabilité $3/4$, et 0 sinon. Soit s la somme de ces valeurs aléatoires. Le nombre de case à ajouter au chemin dans la direction courante sera $\max(s, 3)$.
5. Avancer jusqu'à la case au bout du nouveau segment de chemin, et calculer ses étendus dans les deux directions en tournant 90 degrés à gauche et à droite. La nouvelle direction courante sera choisie parmi les deux avec une probabilité proportionnelle à leurs étendus.
6. Revenir à l'étape 4.
7. Vérifier si le chemin obtenu a fait au moins 7 virages et est de longueur au moins 75. Si oui, l'algorithme termine avec succès. Sinon, revenir à l'étape 1.

Les valeurs de 7 virages et longueur minimale 75 peuvent être modifiées pour adapter à votre besoin. Pourtant on observe que les valeurs données ici donnent des résultats raisonnables. Typiquement un chemin valide sous ces paramètres est généré après une dizaine d'itérations. Voici un échantillonage des chemins ainsi générés.



4 Mouvement des monstres et des tirs

Pour l'animation des mouvements de monstres et de tirs, l'affichage se fera au taux de rafraîchissement (ou *framerate*) de 60, c'est-à-dire on met à jours les positions des monstres et des tirs tous les $1/60$ secondes. Comme tous les mouvements sont continues, **tous les calculs reliés aux mouvements doivent être effectués avec les flottants**.

Les monstres d'une vague apparaissent de leur nid avec un intervalle d'une case (donc un intervalle de 1 seconde, sauf pour les vagues agiles et les vagues de foule, où c'est 0,5 secondes). Quand les monstres se déplacent, ils voyagent de façon continue du centre de la case courante au centre de la prochaine case du chemin. Pour donner un peu d'aléas, pour chaque mise à jour, au lieu de faire avancer chaque monstre à la même vitesse v, on met une fluctuation de 10%, c'est-à-dire que la distance parcourue sera une valeur tirée uniformément entre $0.9 \times v/60$ et $1.1 \times v/60$. Attention aux virages du chemin, où un monstre pourrait faire le virage entre deux mises à jour.

Quand une gemme tire, le tir apparaît au centre de la gemme et se déplace à une vitesse de 3 cases par secondes. Cependant, le monstre visé se déplace aussi. Pour simuler un guidage automatique, lors de chaque mise à jour, on fait avancer le tir vers la position courante du monstre visé. Comme le tir se déplace plus vite que le monstre, il le rattrape toujours. Si la distance entre le tir et le monstre est inférieure à ce que le tir peut parcourir entre deux mises à jour, alors le tir est considéré comme atterri sur le monstre lors de la prochaine mise à jour. Quand un monstre est battu, tous les tirs qui le visent disparaissent.

Pour faciliter la programmation, on vous donne les formules nécessaires pour calculer le mouvement d'un tir. Supposons que le tir est à position $T = (x_T, y_T)$ et le monstre se situe à $M = (x_M, y_M)$ lors de la mise à jour précédente. Le vecteur unitaire $\vec{v} = (x_v, y_v)$ de la direction du tir vers le monstre sera

$$\vec{v} = (x_v, y_v) = \left(\frac{x_M - x_T}{\|MT\|}, \frac{y_M - y_T}{\|MT\|} \right).$$

Ici, $\|MT\| = \sqrt{(x_M - x_T)^2 + (y_M - y_T)^2}$ est la distance entre le tir à T et le monstre à M. Soit d la distance que le tir parcourt entre deux mises à jour. Si $\|MT\| \leq d$, alors le tir atterrit à cette mise à jour. Sinon, la nouvelle position du tir à cette mise à jour sera $(x_T + x_v d, y_T + y_v d)$.

5 Gestion de framerate

Pour gérer le taux de mise à jour de l'affichage, deux choses sont fondamentales :

- Votre programme ne demande pas trop de ressources.
- Vous limitez le nombre d'appels à la fonction `MLV_update_window`.

Votre fonction main ou bien une autre fonction gérant globalement votre jeu devra comporter une boucle maîtresse qui gère le jeu frame par frame. Tant que le joueur

ne veut pas quitter, on met l'affichage à jours, on résout tous les événements sur la frame, puis on calcule le temps passé et on ajuste la vitesse du jeu en attendant quelques millisecondes.

```
/* Main loop over the frames ... */
while(!quit){
    /* Some declaration of variables */
    ...

    /* Get the time in nanoscond at the start of the frame */
    clock_gettime(CLOCK_REALTIME, &end_time);
    /* Display of the current frame, sample function */
    /* THIS FUNCTION CALLS ONCE AND ONLY ONCE MLV_update_window */
    draw_window(&param, &grid);

    /* We get here some keyboard events */
    event = MLV_get_event(...);

    /* Dealing with the events */
    ...
    /* Move the entities on the grid */
    ...
    /* Collision detection and other game mechanisms */
    ...

    /* Get the time in nanosecond at the end of the frame */
    clock_gettime(CLOCK_REALTIME, &new_time);
    /* Compute the time spent for the current frame */
    frametime = new_time.tv_sec - end_time.tv_sec;
    frametime += (new_time.tv_sec - end_time.tv_sec) / 1.0E9;

    /* Force the program to spend at least 1/60 second in total */
    extratime = 1.0 / 60 - frametime;
    if(extratime > 0){
        MLV_wait_milliseconds((int)(extratime * 1000));
    }
}
```

6 Interface graphique

Pendant une partie, tout le terrain (grille 28×22) doit être affiché, avec le numéro de vague courante et l'état de la réserve de mana. Il doit aussi avoir une interface à côté qui permet tous les actions possibles du joueur, avec un espace pour stocker les gemmes non utilisées. Les déplacements des monstres et des tirs doivent être fluides.

Le contrôle doit être réactif. **La fluidité et la réactivité ont un impact important sur la jouabilité du jeu, et seront prises en compte dans l'évaluation finale.** La zone de tirage de chaque gemme doit être dessinée avec un cercle quand elle est sélectionnée par le joueur pour des actions éventuelle. Les monstres doivent être accompagné d'une barre d'HP. Les effets élémentaires doivent être clairement représenté, par exemple avec une couleur.

Pour réaliser l'interface graphique, vous utiliserez obligatoirement la bibliothèque graphique C de l'université : `libMLV`. Une documentation de cette bibliothèque est accessible à l'adresse `http://www-igm.univ-mlv.fr/~boussica/mlv/api/French/html/index.html`. Elle est déjà installée sur toutes les machines de l'université. Pour l'installer sur votre propre machine sur Ubuntu (et d'autres distributions avec `apt`), il suffit d'installer `libmlv3` et `libmlv3-dev` avec `apt`. Pour les autres distributions, il y a normalement les mêmes paquets dans le système de gestion de paquets. Si vous êtes sur Windows, il est conseillé d'installer Ubuntu avec WSL (*Windows Subsystem of Linux*), ou bien d'utiliser une machine virtuelle. Si vous avez un Mac, alors la seule solution sera d'installer une machine virtuelle.

Ce choix de la libMLV est une obligation[1]. Il y a mieux, il y a aussi moins bien... On attend de vous que vous vous familiarisiez avec une bibliothèque écrite par un autre, et que sa documentation vous suffise à construire une véritable application graphique.

7 Modularité

Votre projet doit organiser ses sources sémantiquement. Votre code devra être organisé en modules rassemblant les fonctionnalités traitant d'un même domaine. Un module sans entêtes pour le main, un module pour le moteur du jeu, un module pour la gestion du terrain et un module graphique sont un minimum. Le moteur du jeu sera vraisemblablement très volumineux et on peut même imaginer le subdiviser au besoin (module pour les gemmes, module pour les tirs, module pour les monstres...).

Votre projet devra être compilable via un Makefile qui exploite la compilation séparée. Chaque module sera compilé indépendamment et seulement à la fin, une dernière règle de compilation devra assembler votre exécutable à partir des fichiers objets en faisant appel au linker. Le flag -lMLV est normalement réservé aux modules graphiques ainsi qu'à l'assemblage de l'exécutable (sinon, c'est que votre interface graphique dégouline de partout et que vos sources sont mal modulées).

[1]Certains cas particuliers justifient d'utiliser les bibliothèques `libsdl2` ou `raylib`, dont la prise en main est plus difficile. Cette permission ne sera accordée qu'au cas par cas, après discussion avec le responsable du cours. Toute utilisation de `libsdl2` ou de `raylib` sans autorisation sera sanctionnée par une note de 0/20 à l'ensemble du projet.

8 Pour aller plus loin

Pour les plus récalcitrants d'entre vous, toute amélioration sera considérée comme un plus pour l'évaluation. Pour les valoriser, toutes améliorations doivent être documentées soigneusement dans le rapport. Voici quelques suggestions d'améliorations possibles pour ce projet. Ces propositions sont gradués avec un indice de difficulté entre parenthèse.

- (blob) Ajouter des décorations aléatoires sur les cases vides.
- (blob) Faire augmenter le rayon de détection et la vitesse de tirage d'une gemme selon son niveau. Les détails sont à calibrer.
- (gobelin) Rendre le mouvement des monstres plus naturel.
- (gobelin) Ajouter une animation pour chaque effet élémentaire et chaque réaction élémentaire.
- (gobelin) Ajouter un tableau de classement, enregistrant les meilleurs scores selon le nombre de vagues endurées et la quantité de dégât généré. Pour empêcher les modifications malveillantes, la sauvegarde devra être en **binaire**.
- (gobelin) Ajouter une fonctionnalité pour accélérer l'écoulement du temps dans le jeu.
- (gobelin) Ajouter un nouveau type de vague avec les monstres qui volent. Dans ce cas là, il faut s'assurer que la distance euclidienne entre le nid et le camp soit assez grande lors de la génération de terrain. Les spécificités sont à calibrer.
- (paladin) Ajouter une action avec laquelle le joueur peut frapper tous les monstres présents sur le terrain en sacrifiant une gemme et une certaine quantité de mana (à calibrer). Le dégât de base de cette frappe doit être calculé selon le type et le niveau de la gemme. Si la gemme sacrifiée est pure, il faut aussi que l'effet élémentaire associé soit appliqué.
- (paladin) Ajouter de nouveaux éléments et leur effet. Ajuster l'association entre les types de gemmes pures et les teintes si nécessaire.
- (paladin) Ajouter une fonctionnalité qui permet le réglage de cible pour chaque gemme. Les options naturelles seront de tirer au monstre avec le moins de HP courant, la plus grande vitesse, la plus petite vitesse, le plus de HP initial, etc. Il peut aussi être souhaitable de cibler toujours le même monstre.
- (dragon) Ajouter une fonctionnalité où on peut jeter les gemmes pour générer un dégât à chaque monstre dans un certain rayon. Le dégât (qui doit varier selon le niveau de la gemme jetée) et le rayon sont à calibrer. Le dégât généré sur chaque monstre doit aussi dépendre de la même façon des teintes du monstre et de la gemme.

- (dragon) Ajouter un système de compétences, où le joueur recevoir des points tous les 3 ou 5 vagues à investir dans les compétences auxiliaires (à inventer) qui augmentent les dégâts générés par les gemmes.
- (Jörmungandr) Ajouter une action du joueur qui permet de construire des pièges sur le chemin, auxquels on peut charger des gemmes. Les pièges non chargés ralentissent simplement le monstre, et les pièges chargés avec des gemmes génèrent des dégâts aux monstres qui passent. Les effets sont à inventer et à calibrer.

Attention, **tout changement déteriorant la jouabilité sera lourdement pénalisé** ! Faites attention à bien calibrer les paramètres et les nouveaux mécanismes !

9 Conditions de développement

Le but de ce projet est moins de pondre du code que de développer le plus proprement possible. C'est pourquoi vous développerez ce projet en utilisant un système de gestion de versions Git. Nous utilisons l'outil *Github Classroom* cette année. Il est obligatoire de créer votre dépôt git sur Github à travers du lien suivant :

`https://classroom.github.com/a/_VfQrnD-`

Le nom de l'équipe pour chaque binôme doit prendre la forme "NOM1_NOM2", où "NOM1" et "NOM2" sont les noms de famille des deux étudiants en binôme (en remplaçant les espaces éventuel par un tiret "-".

Comme nous attendons de vous que vous le fassiez sérieusement, votre rendu devra contenir un dump du fichier de logs des opérations effectuées sur votre projet, extrait depuis le serveur de gestion de versions ; afin que nous puissions nous assurer que vous avez bien développé par petites touches successives et propres (commits bien commentés), et non pas avec un seul commit du résultat la veille du rendu. L'évaluation tient compte de la capacité du groupe à se diviser équitablement le travail.

Voici quelques **remarques importantes** :

- L'intégralité de votre application doit être développée exclusivement en langage C (la documentation technique dynamique fait évidemment exception). Toute utilisation de code dans un autre langage (y compris C++) vaudra 0 pour l'intégralité du projet concerné.
- En dehors des bibliothèques standards du langage C et de `libMLV`, il est interdit d'utiliser du code externe : vous devrez tout coder vous-même. Toute utilisation de code non développé par vous-même vaudra 0 pour l'intégralité du projet concerné.
- Tout code commun à plusieurs projets vaudra 0 pour l'intégralité des projets concernés.

10 Conditions de rendu

Vous travaillerez en **binôme** et vous lirez avec attention la Charte des Projets. Il faudra rendre au final une archive `tar.gz` de tout votre projet (tout le contenu de votre projet `git`), les sources de votre application et ses moyens de compilation. Il sera alors crucial de lire des recommandations et conseils d'utilisation de `git` sur la plate-forme `e-learning`.

Un exécutable devra alors être produit à partir de vos sources à l'aide d'un `Makefile`. Naturellement, toutes les options que vous proposerez (ne serait-ce que `--help`) devront être gérées avec `getopt` et `getopt_long`.

La cible `clean` doit fonctionner correctement. Les sources doivent être propres et bien commentées. C'est bien de se mettre un peu à l'anglais si possible.

Votre archive devra aussi contenir :

- Un fichier `log_dev` correspondant au dump des logs de votre projet (nom des commits, qui? et quand?), extrait depuis le serveur de gestion de versions que vous aurez utilisé.
- Un fichier `Makefile` contenant les règles de compilation pour votre application ainsi que tout autre petit bout de code nécessitant compilation (comme les tests par exemple).
- Un dossier `doc` contenant la documentation technique de votre projet ainsi qu'un fichier `rapport.pdf` contenant votre rapport qui devra décrire votre travail. Si votre projet ne fonctionne pas complètement, vous devrez en décrire les bugs.
- Un dossier `src` contenant les sources de votre application.
- Un dossier `include` contenant tous les headers de vos différents modules.
- Un dossier `bin` contenant les fichiers objets générés par la compilation séparée.
- Aucun fichier polluant du type `bla.c~` ou `.%smurf.h%` généré par les éditeurs. Votre dossier doit être propre !
- Si possible, la partie `html` générée par l'utilitaire `doxygen` à partir de votre application de visualisation. Ceci est optionnel mais tellement plus propre.

Sachant qu'un script automatique scrupuleux va analyser et corriger votre rendu avant l'oeil humain, le non-respect des précédentes règles peut rapidement avorter la correction de votre projet. Le respect du format des données est une des choses des plus critiques.