Le Mans Université

Institut d'Informatique Claude Chappe
Le Mans Université

Le Mans Université

Licence Informatique 2ème année
Module 17UF02 Rapport de Projet

Titre du projet

Noms des auteurs

3 avril 2025

Table des matières

Résumé

Ceci est le texte de mon résumé...

1 Introduction

Cette introduction présentera le sujet qui sera traité et le travail avec une présentation du plan adopté
Dans le cadre d'un projet de notre fin d'année de L2 informatique, le type de jeu vidéo créer est un jeu de rôle qui se joue en tour par tour en faisant affronter des créatures fantastiques a l'instar du jeu phare de Nintendo nommé « Pokemon ».
Ce projet a été réaliser en langage C avec la librairie SDL et contrairement a son inspiration , le jeu ne se passe pas dans un pays tous droit sorti de l'imaginaire d'une personne mais bel et bien au sein du bâtiment de l'Institut Claude Chappe, Vous incarnerez un nouvel étudiant rentrant en première année et affronterez d'autres étudiants et enseignants chercheurs déjà installés dans le bâtiment a travers des duels de ICmon.

En première partie, nous allons expliquer comment nous avons conçu le jeu, une présentation général et détaillé de l'univers dans lequel il se situe, son histoire ainsi que celui du joueur communément appeler dans le jargon son « lore » . Ensuite sa direction artistique c'est a dire tous l'aspect graphique seront expliciter , avec quels logiciels les sprites des Icmons, du joueur et des niveaux ont été esquisser et enfin toutes les possibilités offertes au joueur pendant son aventure.

Puis nous parlerons de l'organisation, quelles missions a été confié pour chaque personne du groupe et sous quels deadline devaient-elles être rendu ,tous ceci a été planifié grâce a un diagramme de Gantt que nous avons réalisé.

En troisième partie nous expliquerons la gestion des Icmons , comment ils ont été générés et comment la gestion des duels a été réalisé, puis nous parlerons de la gestion des bases de données, comment les données des Icmons et le jeu ont été sauvegardé puis chargé et ensuite réutiliser de nouveau.

Puis nous aborderons la gestion de la carte, comment le joueur se déplace dans l'espace et comment la caméra le suit. et lorsqu'il rencontre un obstacle, une collision doit se produire.

Et enfin nous mettre en évidence la rendu du jeu , les sprites que nous avons créer doivent être implémenté directement dans le code, le système de combat c'est a dire les points de vie décrémenter lorsqu'une attaque est subie ,devrait être cohérent avec ce que le joueur voit sur l'écran et pour finir l'interface par quel moyen l'interface du jeu a été conçu, de quelle façon le joueur change de fenêtre lorsque tel bouton est appuyé

*Pokemon :le jeu sera présenter dans la suite du rapport
Sprite : image 2D utilisé dans les jeux vidéos
ICmon : contraction de l'expression "Institut Claude chappe " et "Monstre"
Lore : histoire d'un jeu vidéo
deadline : date limite

2 Conception

2.1 Présentation du jeu

ICPocket est un jeu de rôle basé sur l'univers des jeux vidéos "Pokémon" développés par GAME FREAK et publiés par Nintendo. Le jeu met le joueur dans la peau d'un étudiant de la faculté d'informatique du Mans et d'un dresseur de petites créatures nommées les *ICMons*. Cet étudiant, qui vient tout juste d'arriver au sein de l'institut Claude Chappe pour sa première année de license, choisit son premier partenaire parmi quatre ICmons offerts par Louis, Nathan, Alban et William.

L'objectif du joueur est de devenir le maître de l'IC2 en remportant le tournoi annuel de la faculté : "le tournoi de l'IC2", puis en faisant tombé son conseil des quatres et son champion. Pour cela, il devra se confronter à ses camarades et ses professeurs dans des duels d'ICMons : des combats de type joueur contre ordinateur en tour par tour dans lequelles chaque partie envoit un ICMons parmi un maximum de six se battre contre l'autre, jusqu'à ce que l'un des combattants n'est plus aucun ICMons apte à se battre. Ces affrontements tactiques et imprévisibles pousseront le joueur à découvrir les forces et faiblesses de son équipe et à mettre en place la meilleure stratégie pour parvenir à la victoire. Toutefois, cela ne sera pas une tâche aisée, puisque la moindre défaite signifie que le joueur sera éliminé du tournoi et devra donc recommencer du début.

2.2 Direction artistique

En hommage aux jeux vidéos originaux de la série "Pokémon", nous avons fait le choix de nous inspirer de la direction artistique de la troisième génération de ces jeux : "Pokémon Rubis et Saphir". Cette direction artistique est basée sur la 2D, avec des sprites pixelisées.

Pour la réalisation de la carte, nous avons utilisé les assets libre de droit disponible sur le site kenney.nl. La carte a ensuite été assemblé par Alban et Louis. La plupart des sprites des ICMons ont été dessiné par William, Loup Picault (le goat), et Nathan. Les sprites restants ont été dessiné par intelligence artificielle.

Afin de limiter le nombre de sprites que nous avions à réaliser pour ce projet, nous nous sommes inspirés de la méthode d'affichage des sprites du fangame "Pokemon Infinite Fusion", dans lequelle durant un combat, les sprites de l'équipe du joueur sont retournés par symétrie axiale verticale, ce qui donne l'illusion d'un sprite *de dos* tout en ayant qu'un seul sprite à utiliser.

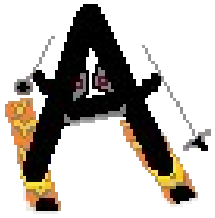


FIGURE 1 – Exemple de sprite pour un ICMon

2.3 Fonctionnalités

Dans cette partie, nous présenterons toutes les fonctionnalités disponibles dans le jeu

Au lancement du jeu, le joueur peut personnaliser ses paramètres selon ses souhaits : les dimensions de la fenêtre d'écran, la vitesse du texte en jeu et le volume sonore. Après cela, il peut charger une partie déjà entamée, à condition qu'une telle partie existe dans les fichiers du jeu, ou bien débuter une nouvelle aventure.

Dans le cas d'une nouvelle partie, le joueur se vera alors proposé quatre ICMons et choisira celui qu'il préfère, puis arrivera sur la carte, à l'entrée du bâtiment IC2. Le joueur peut alors explorer la carte comme il le souhaite : hall d'accueil, salle de TD et amphitéâtre. Le joueur trouvera également en bas de l'escalier principal un drôle de personnage nommé <Insérer nom ici>. Le joueur peut discuter avec lui : <Insérer nom ici> proposera notamment au joueur de changer d'endroit pour commencer son prochain match du tournoi de l'IC2.

Puis l'affrontement commence. Le joueur et son opposant, controlé par l'ordinateur, envoient au combat l'ICMons se trouvant en tête de leurs équipes respectives. Chaque tour, les deux protagonistes choisissent une action parmi les deux suivantes : utiliser l'une des quatres attaques connues de leur ICMon, ou bien remplacer le combattant actuel par un autre encore apte dans leur équipe. Durant ce duel, différentes afflictions peuvent modifiées le cours de la partie. En effet, certaines attaques possèdent une chance plus ou moins grande d'empoisonner, de brûler, de rendre confus, d'apeurer ou d'altérer les forces du lanceur de la capacité et de l'adversaire.

Le combat continue jusqu'à ce que les ICMons d'un des dresseurs soient tous hors d'état de se battre. Si le joueur perd son duel, sa sauvegarde est supprimée et il doit recommencer une nouvelle partie pour tenter de battre son meilleur score, c'est-à-dire son nombre de victoire. Dans le cas contraire, le joueur peut, si il le souhaite, récupérer l'un des ICMons que son adversaire avait dans son équipe et l'ajouter à la sienne. Enfin, le joueur retourne dans le hall du bâtiment IC2 pour se préparer au prochain affrontement. La partie en cours est sauvegardée à la fin de chaque duel, en cas de victoire du joueur, afin qu'il puisse mettre en pause son aventure et la reprendre plus tard.

Au bout de quinze victoires consécutives, les cinqs derniers combats proposés au joueur seront contre le conseil des quatres, composé des quatres créateurs du jeu : Louis, Nathan, Alban et William, et l'affrontement ultime contre Mme Py, professeur à l'université du Mans en informatique et dans le cadre du jeu mâitresse de l'IC2. Si le joueur parvient à faire chûter Mme Py de son trône, il remporte la partie.

3 Organisation

Notre projet de jeu 2D a été développé selon une architecture modulaire rigoureuse, permettant une séparation claire des responsabilités entre les différents composants. Nous avons adopté un modèle de développement itératif, en commençant par implémenter les fonctionnalités essentielles du moteur de jeu avant d'ajouter progressivement des mécanismes plus complexes. L'organisation s'est articulée autour de trois axes principaux : la gestion du rendu graphique avec SDL, la logique de jeu et les structures de données, et enfin la synchronisation entre ces éléments via un gestionnaire de threads. Cette approche nous a permis de maintenir un code propre et maintenable, tout en facilitant le débogage et l'extension des fonctionnalités au cours du développement. Notre système de maps interconnectées avec mémorisation des positions du joueur illustre parfaitement cette approche modulaire, où chaque composant (Map, Player, Camera) remplit un rôle spécifique mais communique efficacement avec les autres.

3.1 Planning Prévisionnel

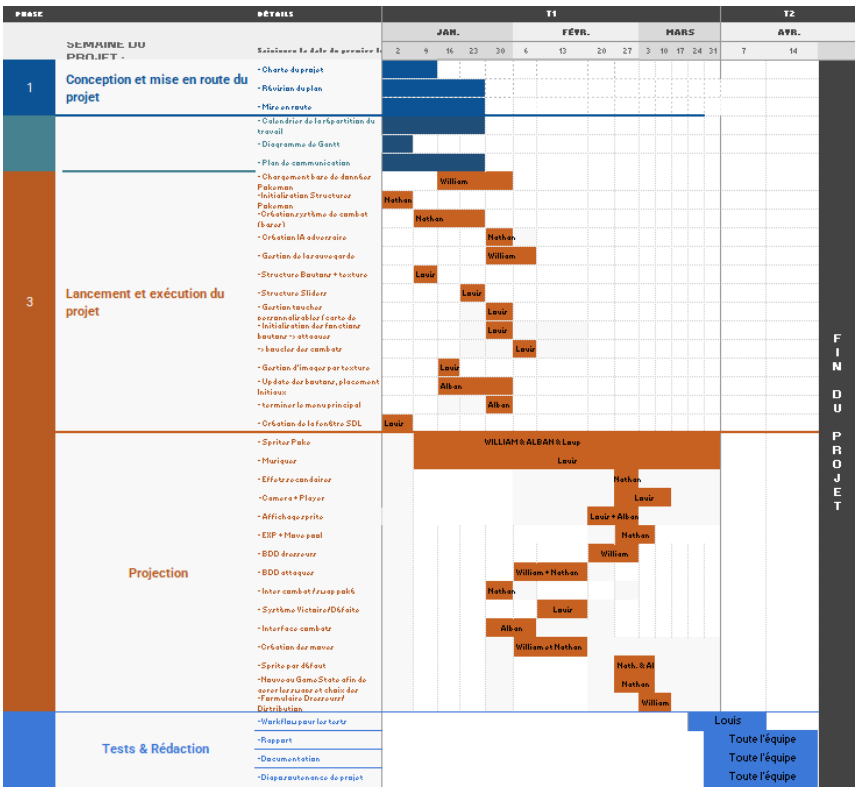


FIGURE 2 – Diagramme de Gantt

Notre planning prévisionnel a été établi dès le début du projet pour garantir une progression cohérente et le respect des délais. Nous avons divisé le développement en plusieurs phases distinctes, chacune avec des objectifs spécifiques et des livrables clairement définis. Cette approche structurée nous a permis d'identifier rapidement les tâches critiques et d'allouer les ressources nécessaires en conséquence. Grâce à des points de contrôle réguliers, nous avons pu ajuster notre planification en fonction des difficultés rencontrées et des avancées réalisées, assurant ainsi une gestion efficace du temps disponible jusqu'à l'échéance finale.

L'élaboration de notre planning prévisionnel s'est appuyée sur une méthode d'estimation collaborative où chaque membre de l'équipe a évalué le temps nécessaire pour les tâches relevant de son domaine d'expertise. Pour les composants interdépendants comme le système de transition entre maps et la gestion des déplacements du joueur, nous avons organisé des sessions de planification poker où les estimations individuelles étaient discutées pour atteindre un consensus. Cette méthode nous a permis d'identifier dès le départ les zones de risque potentiel, comme la synchronisation des threads ou la gestion des collisions, qui ont bénéficié d'une marge temporelle supplémentaire dans notre planning.

Notre diagramme de Gantt identifie clairement quatre jalons principaux : la finalisation de l'architecture de base (semaine 4), l'implémentation du système de combat (semaine 8), l'intégration complète des maps et des transitions (semaine 12), et les tests d'intégration finaux (semaine 15). Ces jalons ont servi de points d'ancrage pour nos réunions d'avancement et nous ont permis de mesurer objectivement notre progression par rapport aux objectifs initiaux. Une caractéristique importante de notre approche a été l'inclusion délibérée de périodes tampons après chaque jalon majeur, offrant la flexibilité nécessaire pour gérer les imprévus techniques inévitables dans le développement d'un jeu.

3.2 Répartition des tâches

La répartition des tâches s'est effectuée en fonction des compétences et des intérêts de chaque membre de l'équipe. Pour structurer notre travail et définir clairement les responsabilités, nous avons mis en place une matrice RACI complète. Cette matrice a permis d'identifier pour chaque tâche du projet qui était Responsable de son exécution, qui était tenu de rendre des comptes (Accountable), qui devait être Consulté et qui devait être simplement Informé des avancées. Par exemple, Louis a été désigné responsable des composants techniques comme la gestion de la caméra, le multi-threading et l'affichage des sprites, tandis que Nathan s'est chargé principalement de la logique de jeu avec les systèmes de combat et les effets secondaires. William a pris en charge les aspects liés à la gestion des données, notamment les bases de données des dresseurs et des attaques, tandis qu'Alban s'est concentré sur les interfaces utilisateur et les éléments graphiques.

Notre planification temporelle s'est appuyée sur deux diagrammes de GANTT complémentaires. Le diagramme prévisionnel, établi en janvier, décomposait le projet en quatre phases principales : conception et mise en route, définition et planification, lancement et exécution, et enfin tests et rédaction. Nous avions initialement alloué deux semaines pour la mise en place du système de transition entre maps et l'implémentation de la caméra, mais ce calendrier s'est révélé optimiste. Le diagramme de GANTT actualisé, que nous avons maintenu tout au long du projet, montre que ces tâches ont finalement nécessité près de trois semaines, principalement en raison des problèmes de gestion de mémoire lors des changements de map qui ont causé des erreurs de segmentation.

En revanche, certaines fonctionnalités ont été implémentées plus rapidement que prévu : l'interface des combats et le système de textes dynamiques, initialement prévus pour mi-mars, ont été terminés fin février. Cette avance nous a permis d'allouer plus de temps aux tests et à l'optimisation du système multi-thread, élément critique pour la fluidité du jeu. La comparaison entre nos deux diagrammes met en évidence l'importance d'une planification flexible et d'une communication constante entre les membres de l'équipe. Nos réunions hebdomadaires, documentées dans notre matrice RACI, nous ont

permis d'ajuster en permanence nos priorités et d'optimiser l'allocation des ressources humaines en fonction des difficultés rencontrées et des avancées réalisées.

3.3 Outils

Pour mener à bien notre projet, nous avons utilisé plusieurs outils essentiels qui ont considérablement facilité notre développement. GitHub [1] a constitué le pilier central de notre collaboration, nous permettant de gérer efficacement le versionnage du code source, de suivre les modifications et de travailler simultanément sur différentes fonctionnalités sans conflits majeurs. Grâce aux pull requests et aux issues, nous avons pu organiser notre développement de manière transparente et documenter nos choix techniques. Stack Overflow [2] s'est révélé être une ressource inestimable pour résoudre les problèmes techniques spécifiques à SDL et à la gestion de mémoire en C, notamment pour l'implémentation de notre système de transition entre maps.

Pour la conception des mécaniques de jeu inspirées de Pokémon, nous nous sommes appuyés sur plusieurs ressources spécialisées. Bulbapedia [3] a servi de référence encyclopédique pour comprendre les systèmes de combat et l'équilibrage des créatures, tandis que PokemonShowdown Calc [4] nous a permis d'affiner les formules de calcul des dégâts et statistiques pour nos ICMons. Le forum Developpez.net [5] a également été consulté régulièrement pour des questions spécifiques liées au développement en C et à l'optimisation des performances.

En complément de ces ressources en ligne, nous avons tiré parti des outils d'intelligence artificielle comme GitHub Copilot [6] pour accélérer certaines phases de débogage et pour suggérer des optimisations dans notre code. Cette assistance a été particulièrement utile pour résoudre les problèmes complexes de gestion de mémoire qui causaient des erreurs de segmentation lors des transitions entre maps. L'ensemble de ces outils complémentaires nous a permis de maintenir une progression constante tout en assurant la qualité technique du projet final.

1. Voir Annexe 6.1
2. Voir Annexe 6.1
3. Voir Annexe 6.1
4. Voir Annexe 6.1
5. Voir Annexe 6.1
6. Voir Annexe 6.1

2 - 1	On met un grand texte qui sera sur plusieurs lignes

4 Developpement

4.1 Gestions des ICMons

4.1.1 structures de données

4.1.2 systèmes de combats

4.2 Gestion des bases de données

4.2.1 sauvegardes

Dans un jeu vidéo, la gestion des données est primordiale, il faut que le joueur puisse sauvegarder sa progression et la reprendre ultérieurement sans perte d'informations. Pour cela, nous avons utilisé un fichier de sauvegarde au format CSV, contenant toutes les informations nécessaires au joueur et au bon déroulement de la partie.
Ces fichiers contiennent une liste de tous les ICmons du joueur, ainsi que leurs caractéristiques : points de vie, attaques, défenses, etc. Si un fichier de sauvegarde est invalide, c'est-à-dire inexistant, corrompu, vide ou contenant un mauvais format, une nouvelle partie se lance et un nouveau fichier de sauvegarde est créé.

4.2.2 Menus de sauvegarde

FIGURE 3 – Menu principal du jeu ICPocket

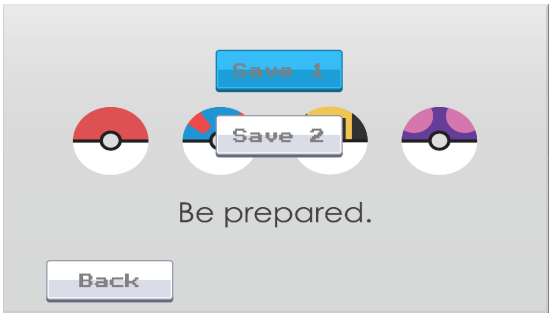


FIGURE 4 – Menu de chargement du jeu ICPocket

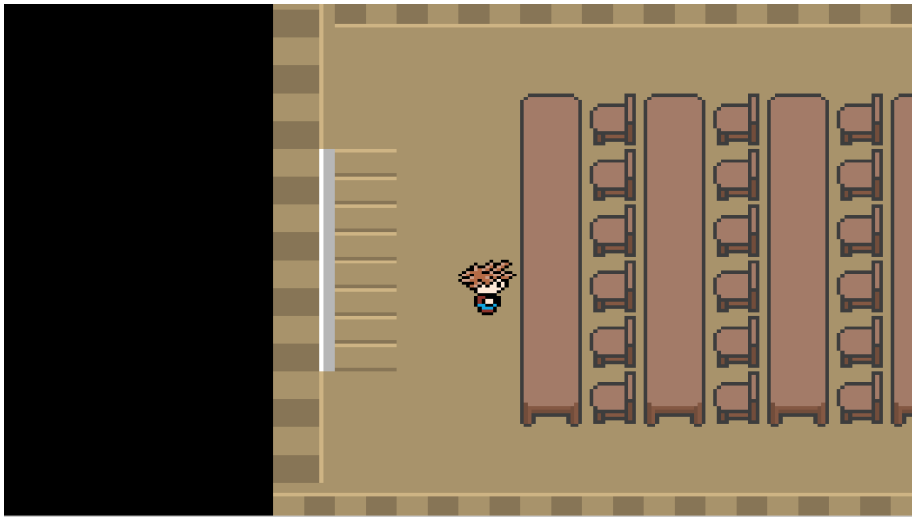
Dans le code, les fonctions de sauvegarde et de chargement se situent dans le fichier `save.c`. La fonction `sauvegarder` prend comme paramètres des structures d'équipes , respectivements celui du joueur et celui de l'adversaire, et cherche le fichier correspondant au numéro du bouton de sauvegarde appuyé. Ensuite , grave seulement les numéros d'identification des ICMons et leurs charactéristiques générés aléatoirement ou choisis par le joueur, tel que les IVs,niveaux de ICmon , le nombre et type d'attaque, et finalement les données propres a la progression comme le nombre et le pseudo des joueurs battus. La fonction `charger`

4.2.3 base de données des icmons

Toutes la gestion de base de données ont été fais dans des fichiers CSV situés dans le dossier `data/` Les ICMons sont générés grâce a leurs ID leurs numéro d'identification, avec la fonction `generate_poke` et `generate_poke_ennemie` qui prennent donc en paramètres les ID.

4.3 Gestion de la map

4.3.1 Chargement de la map



Analyse : Pour améliorer la qualité de notre gameplay et la qualité visuelle de notre jeu, nous avons décidé de créer une carte en 2D qui reconstitue le rez-de-chaussée de l'IC2, où nous pouvons retrouver des zones existantes du bâtiment telles que l'amphithéâtre, comme l'image ci-dessus. L'utilisation de fichiers CSV a permis de gérer les collisions. Dans les fichiers, les numéros 1 correspondent aux collisions, les 0 aux zones libres où le personnage peut marcher, et d'autres numéros servent à passer d'une carte à l'autre, comme du hall à l'amphithéâtre, grâce à une lecture de fichier qui permet de reconnaître l'endroit où l'on va. La carte apporte une plus grande liberté dans le jeu puisqu'il y a la possibilité de se déplacer partout dans le bâtiment. Il y a également l'ajout d'un PNJ (personnage non joueur) qui permet à l'utilisateur de lancer les combats dans l'ordre donné.

Réalisation : Pour finir sur le chargement de la carte, il y a eu la création de points de spawn qui sont accordés au chiffre -1, les collisions aux chiffres 1 et les zones libres aux 0 dans nos fichiers CSV. Voici un exemple de CSV correspondant à l'amphithéâtre :

```
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1
1,0,0,0,1,0,1,0,1,0,1,0,1,0,0,1
1,1,0,0,1,0,1,0,1,0,1,0,1,0,0,1
1,1,0,0,1,0,1,0,1,0,1,0,1,0,0,1
1,1,0,0,1,0,1,0,1,0,1,0,1,0,0,1
1,1,0,0,1,0,1,0,1,0,1,0,1,0,0,1
1,0,0,0,1,0,1,0,1,0,1,0,1,0,0,1
1,0,0,0,0,0,0,0,0,0,0,0,0,-1,-1,1
1,1,1,1,1,1,1,1,1,1,1,1,1,9,9,1
```

-1=spawn 0=Air 1=collision 9=retour dans le hall.

Les fonctions implémentées afin de rendre le rendu de la carte possible sont 'initMap', qui permet d'initialiser la carte en utilisant les autres fonctions du

fichier ‘Map.c‘, telles que ‘initCollisionFromCSV‘, qui permet de gérer les collisions grâce aux CSV. Nous avons créé la fonction ‘checkAndLoadNewMap‘, qui permet de vérifier si, dans un fichier CSV, le personnage se situe sur une case qui correspond au chargement d'une nouvelle zone. Dans ce cas, la fonction 'loadNewMap' est appelée pour mettre à jour le point d'apparition du joueur par rapport à la nouvelle carte et lorsque le joueur revient sur la carte précédente, il est renvoyé à l'endroit où il était avant de changer de zone. Pour ce faire, le programme regarde dans ses plus proches voisins s'il y a une case 0, si oui le joueur est directement placé à cet endroit.

Pour rendre cela possible, nous avons utilisé le logiciel de création de carte Tiled, puis Aseprite pour rendre la carte plus jolie en retirant les impuretés, afin de gérer tout cela avec des fichiers CSV. Tiled permet de créer des cartes facilement et de les exporter au format CSV, ce qui simplifie leur intégration dans le jeu.

4.3.2 Caméra

La gestion de la caméra est un élément essentiel pour offrir une expérience utilisateur fluide et immersive dans notre jeu. La caméra permet de suivre les déplacements du joueur tout en affichant une partie de la map grâce au zoom, ce qui est particulièrement utile pour les grandes cartes en 2D. Cela évite d'afficher l'intégralité de la map à l'écran, ce qui pourrait surcharger visuellement le joueur et réduire les performances.

Utilité de la caméra : La caméra a été implémentée pour plusieurs raisons : Suivi du joueur : La caméra suit dynamiquement la position du joueur, garantissant que ce dernier reste toujours visible à l'écran. La map est rendue en entier puis fait un zoom sur le joueur afin de le suivre uniquement et cacher l'affichage des endroits inutiles.

Immersion : En limitant la vue du joueur à une zone spécifique, la caméra renforce l'immersion en simulant une perspective réaliste.

Pourquoi l'avoir implémentée ? L'implémentation de la caméra était nécessaire pour gérer efficacement les grandes maps et offrir une meilleure expérience utilisateur. Sans caméra, il aurait été impossible de naviguer correctement sur des cartes de grande taille, car tout serait affiché en même temps, rendant le jeu confus et peu ergonomique. De plus, la caméra permet de centrer l'attention du joueur sur son personnage et les zones importantes, tout en masquant les parties inutiles de la map. Nous avons donc utilisé le système d'interpolation linéaire pour avoir un rendu pixel par pixel. La caméra a été conçue pour être flexible et s'adapter à différentes tailles de fenêtres.

En résumé, la caméra joue un rôle crucial dans la navigation et le rendu du jeu, en améliorant à la fois les performances et l'expérience utilisateur. Tout cela a été possible grâce aux différents outils utilisés tels que Stack Overflow, Developpez.net, Wikipedia afin d'utiliser l'interpolation linéaire.

Réalisation : L'utilisation de fonctions était obligatoire pour bien gérer la caméra. Pour cela, la fonction ‘getWorldToScreenRect‘ permet de convertir les coordonnées du monde en coordonnées écran. ‘updateCameraViewport‘ permet de mettre à jour la taille visible de la carte par rapport à la caméra et ‘updateCamera‘ fait en sorte d'actualiser la position de la caméra grâce à l'utilisation de l'interpolation linéaire, ce qui permet un rendu plus fluide des déplacements du joueur.

4.3.3 Déplacement

Dans un jeu, les déplacements sont les choses les plus importantes puisque sans ça, on ne peut rien faire sur une carte.

Implémentation des fonctions : Pour gérer les déplacements du joueur, nous utilisons la fonction ‘createPlayer‘, qui a pour but de créer le joueur et son sprite (image) ainsi que de définir comment il va se déplacer. Dans cette fonction, les déplacements se font par des calculs de coordonnées afin de déplacer le joueur du point de spawn jusqu'à la prochaine case, que ce soit en haut, en bas, à gauche ou à droite, à condition de ne pas rencontrer de collision.

Pour déplacer le joueur ainsi que gérer les animations du sprite, il faut utiliser les fonctions ‘updatePlayerPosition‘ et ‘updatePlayerAnimation‘. La fonction ‘updatePlayerPosition‘ met à jour la position du joueur sur la carte en utilisant un système d'interpolation linéaire pour rendre les déplacements fluides. Elle calcule la position intermédiaire entre la position actuelle et la position cible en fonction du temps écoulé (‘deltaTime‘).

La fonction ‘updatePlayerAnimation‘ permet de changer le sprite du joueur en fonction de son état (marche vers le haut, le bas, la gauche ou la droite). Elle utilise un système de frames pour donner un style d'animation réaliste, en alternant entre différentes images de la spritesheet.

L'ajout des déplacements donne une sorte de liberté au jeu, où le personnage peut se déplacer où il veut sur la carte. Grâce aux animations du personnage, cela rend les déplacements encore plus immersifs et réalistes.

De plus, la fonction ‘renderPlayerWithCamera‘ est utilisée pour afficher le joueur à l'écran en tenant compte de la position de la caméra. Cela permet de s'assurer que le joueur reste visible même lorsque la carte est plus grande que la fenêtre d'affichage.

Enfin, les déplacements sont également liés à la gestion des collisions. Le joueur ne peut se déplacer que sur des cases libres (valeurs spécifiques dans la matrice de la carte). Cela est vérifié dans les fonctions de gestion des événements, comme ‘handlePlayerEvent‘, qui détecte les entrées clavier et met à jour l'état du joueur en conséquence.

4.4 rendu du jeu

Le système de rendu graphique constitue l'un des piliers fondamentaux de notre jeu, devant concilier performances optimales et cohérence visuelle.

Architecture de base Notre implémentation repose sur la bibliothèque SDL (Simple DirectMedia Layer) qui offre une couche d'abstraction multi-plateforme pour l'accès au matériel graphique. Pour garantir une expérience fluide malgré la richesse visuelle du jeu, nous avons développé une architecture de rendu multi-niveaux qui sépare les différentes composantes graphiques.

Gestion des threads Une caractéristique notable de notre moteur de rendu est son intégration avec un système multi-thread. Notre gestionnaire de threads initialise plusieurs verrous mutex dédiés aux ressources partagées : l'audioMutex pour les opérations sonores, le physicsMutex pour les calculs de

déplacement et collisions, et initialement un renderMutex pour les opérations graphiques (désactivé dans la version finale pour des raisons d'optimisation).

Cette approche multi-thread permet d'exécuter simultanément la logique du jeu et le rendu graphique, optimisant ainsi l'utilisation des ressources système. Le rendu spécifique de la carte du monde est effectué dans une fonction dédiée (renderMap) qui synchronise l'accès aux données partagées à l'aide de mutex. Cette fonction verrouille d'abord le mutex physique, effectue les opérations de rendu de la carte et du joueur avec la caméra, puis déverrouille le mutex avant de présenter le résultat à l'écran.

La gestion des images par seconde (FPS) est assurée par un mécanisme sophistiqué (manageFrameRate) qui ajuste dynamiquement le timing du rendu pour maintenir une cadence constante. Notre système vise un objectif de 60 FPS en calculant le temps écoulé depuis le début de la frame et en introduisant un délai approprié si nécessaire. Ce mécanisme garantit une expérience fluide tout en calculant précisément le deltaTime, une variable critique pour la synchronisation des animations et des mouvements indépendamment de la puissance de calcul disponible.

4.4.1 gestion de l'interface

Système d'états Notre interface utilisateur est gérée par un système d'états (définis dans l'énumération AppState) qui détermine quels éléments doivent être affichés à chaque instant. La fonction render orchestre l'affichage des différents composants en fonction de l'état actuel du jeu. Cette fonction conditionne le rendu selon des valeurs telles que MENU, NEWGAME, SWAP, LEARNMOVE ou STARTERS, appelant les fonctions de rendu appropriées pour chaque situation.

Transitions et mises à jour Les transitions entre ces différents états sont gérées par une fonction changeState qui assure la cohérence des données et prépare les ressources nécessaires au nouvel état. Cette fonction prend en compte à la fois l'état actuel et l'état cible pour effectuer les actions de transition appropriées, comme l'initialisation d'un combat lorsque l'on passe de l'exploration à un affrontement.

Mise à jour des textes des boutons La fonction updateICMonsButtonText parcourt l'équipe spécifiée et met à jour les textes des boutons correspondants dans l'interface utilisateur. Si un emplacement est vide, le bouton affiche un espace, sinon il affiche le nom de l'ICMon. Ce système permet de maintenir une cohérence visuelle même lorsque les données sous-jacentes changent, comme lors d'un échange d'ICMons ou de l'apprentissage de nouvelles capacités.

Gestion des événements La gestion des événements utilisateur est assurée par un ensemble de fonctions spécialisées pour chaque état du jeu, comme handleLoadGameEvent ou handlePauseEvent, qui interprètent les interactions et déclenchent les actions appropriées. Par exemple, la fonction de gestion des pauses vérifie si la touche Échap a été pressée et change l'état du jeu en conséquence.

4.4.2 gestion des combats avec SDL

Machine à états Le système de combat repose sur une machine à états finie représentée par l'énumération BattleTurnState, qui définit les différentes

phases d'un tour : initialisation (TURN_INIT), attente des instructions du joueur (TURN_WAIT_TEXT), exécution des actions (TURN_ACTION1 et TURN_ACTION2), et finalisation (TURN_FINISHED).

Rendu des sprites Le rendu des sprites des ICMons est géré par des fonctions spécialisées comme renderICMonsSprite, qui s'occupe de positionner et d'animer correctement les créatures à l'écran. L'initialisation des sprites est effectuée lors du chargement d'une partie par la fonction initTeamSprites, qui prend en compte le positionnement relatif des sprites sur l'écran, avec des ratios différents selon qu'il s'agit de l'équipe du joueur (BLUE SPRITE X RATIO, BLUE SPRITE Y RATIO) ou de l'adversaire (RED SPRITE X RATIO, RED SPRITE Y RATIO).

4.4.3 Optimisation des performances

Gestion des ressources Notre moteur de rendu inclut plusieurs mécanismes d'optimisation pour garantir des performances fluides. L'un des plus importants est la gestion des ressources lors des transitions entre états. Lorsqu'un joueur passe de l'exploration à un combat, ou lors d'un changement de map, nous devons libérer certaines ressources et en charger d'autres. Cette gestion est centralisée dans la fonction cleanupResources qui libère méthodiquement les textures, les textes, les contrôleurs et les structures de données comme la carte, le joueur et la caméra.

Orchestration des threads Un autre aspect critique de l'optimisation est l'orchestration des threads. En séparant la logique audio, la physique et le rendu dans des threads distincts, nous exploitons efficacement les architectures multi-cœurs modernes.

La fonction principale du jeu (mainLoop) coordonne l'ensemble du processus de rendu en gérant les événements, en appelant les fonctions de rendu appropriées selon l'état du jeu, et en maintenant une cadence constante grâce à manageFrameRate. Cette boucle distingue le cas spécial de l'état MAP, où elle utilise renderMap, des autres états où elle effectue un rendu standard avec gestion des boutons et mise à jour du tour de combat si nécessaire.

En conclusion, notre système de rendu combine une architecture multithread sophistiquée, une gestion d'états flexible et des mécanismes d'optimisation ciblés pour offrir une expérience visuelle riche tout en maintenant des performances fluides sur une variété de configurations matérielles. Ce système démontre l'efficacité de notre approche modulaire et la pertinence de nos choix techniques, notamment l'utilisation de SDL et la parallélisation des traitements.

4.4.4 Technologies graphiques et défis techniques

Synchronisation L'implémentation de notre système de rendu a nécessité de relever plusieurs défis techniques spécifiques. L'un des plus complexes concernait la synchronisation entre le système de rendu et le système de physique, particulièrement critique lors des transitions entre maps. Nous avons conçu un mécanisme de verrouillage sélectif qui permet au thread physique de continuer à calculer les collisions et les déplacements pendant que le thread principal s'occupe du rendu, tout en évitant les problèmes de concurrence.

Effets visuels Le rendu des effets visuels dynamiques, comme les animations d'attaque ou les transitions entre états du jeu, a également posé des défis

techniques significatifs. Nous avons implémenté un système de timers et de séquences qui permet de coordonner précisément les animations avec la logique du jeu. Ce système utilise le deltaTime calculé par notre gestionnaire de fréquence d'images pour garantir une vitesse d'animation cohérente quelle que soit la puissance de calcul disponible.

Adaptabilité Un autre aspect innovant de notre système graphique est la gestion adaptative de la qualité visuelle. Sur les configurations matérielles moins puissantes, notre moteur peut automatiquement réduire certains effets visuels pour maintenir une fréquence d'images acceptable. Cette adaptation dynamique est particulièrement importante pour notre jeu qui vise une large compatibilité matérielle.

Le rendu des textes a également bénéficié d'une attention particulière, avec un système de mise en cache des textures de texte pour éviter de régénérer constamment les mêmes chaînes de caractères. Cette optimisation s'est révélée particulièrement efficace dans les écrans de combat où les mêmes messages peuvent apparaître à plusieurs reprises.

Notre approche du rendu multi-couches, particulièrement visible dans les écrans de combat, permet de composer des scènes visuellement riches en combinant différents éléments (arrière-plans, sprites des créatures, interface utilisateur) avec un contrôle précis sur l'ordre de dessin. Cette technique garantit que les éléments sont toujours affichés dans le bon ordre, évitant les problèmes de superposition incorrecte.

Pour conclure, l'architecture de notre système de rendu, bien que développée avec des technologies relativement simples comme SDL, démontre qu'une conception réfléchie et une implémentation soignée peuvent conduire à des résultats visuellement impressionnants et techniquement efficaces. Notre approche modulaire facilite également l'extension future du moteur avec de nouvelles fonctionnalités visuelles sans nécessiter de refonte majeure du système existant.

5 Bilan et résultats

5.1 Bilan

Le projet a été un succès, tant sur le plan technique que sur le plan de l'expérience utilisateur. Nous avons réussi à créer un jeu 2D fonctionnel et agréable à jouer, avec une interface intuitive et des mécaniques de jeu inspirées de l'univers Pokémon. La gestion des ICMons, des combats et des déplacements a été bien intégrée, offrant une expérience fluide et immersive. La répartition des tâches a été efficace, chaque membre de l'équipe ayant pu contribuer selon ses compétences et ses intérêts. La communication constante entre les membres a permis d'identifier rapidement les problèmes et de trouver des solutions adaptées. Nous avons également appris à travailler avec des outils de développement modernes, tels que GitHub pour la gestion de version, et à utiliser des ressources en ligne pour résoudre des problèmes techniques. L'utilisation de GitHub Copilot a été un atout précieux pour accélérer le développement et améliorer la qualité du code.

5.2 Résultats

Les résultats obtenus sont plus que satisfaisants grâce aux différents ajouts et améliorations apportés au projet. Nous avons ajouté de la liberté grâce à la carte et aux PNJ. Toutes les modifications apportées et l'avancement plus rapide du projet ont donc modifié notre Gantt prévisionnel pour en arriver au Gantt final ci-dessous.

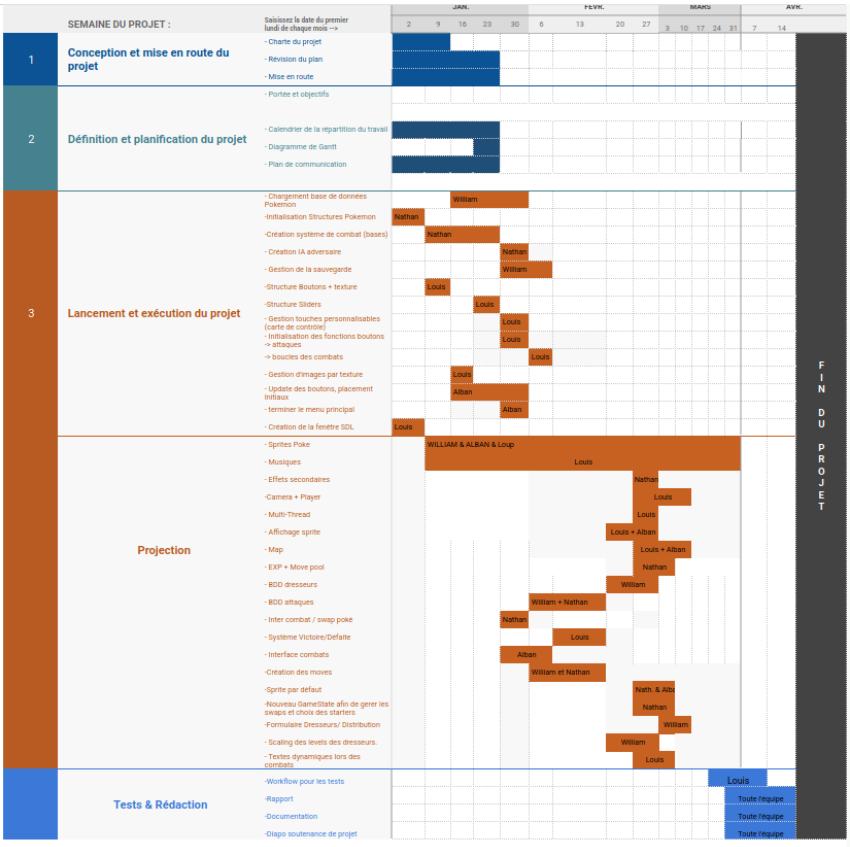


FIGURE 5 – Diagramme de Gantt

5.3 Avis des membres de l'équipe sur le projet

Avis des membres de l'équipe :

— **Alban :** «J'ai énormément apprécié réaliser ce projet avec mon groupe car nous avons eu une très bonne entente directement, que ce soit la répartition des tâches ou la communication entre nous. Nous avons tous su nous entraider et nous conseiller sur les différentes parties du projet. De plus, j'ai beaucoup appris sur le développement de jeux vidéo, notamment sur le fait d'approfondir mes capacités en C et de découvrir la bibliothèque SDL. J'ai également appris à utiliser GitHub. Je suis très satisfait du résultat final et j'espère que nous pourrons continuer à développer d'autres projets ensemble.»

— **Louis :**

— **Nathan :**

— **William :**

6 Références

6.1 Outils de développement

GitHub : github.com
Stack Overflow : stackoverflow.com
Bulbapedia : bulbapedia.bulbagarden.net
PokemonShowdown Calc : calc.pokemonshowdown.com
Developpez.net : forum.developpez.com
GitHub Copilot : copilot.github.com
Aseprite : aseprite.org
Tiled : mapeditor.org
Poképédia : pokepedia.fr

7 Annexes