Projet long Technologie Objet
Rapport final

Groupe GH-2

Département Sciences du Numérique - Première année
2021-2022

Contents

1 Rappel du sujet

Notre projet est le développement d'un jeu 2D de type bac à sable et exploration.
Le jeu se compose d'une fenêtre graphique comportant une map composé de différents types de cubes (Pierre, bois etc..). Le joueur peut se déplacer dans le monde en cassant ou ajoutant des blocs dans le monde. Il possède possède une barre de vie qui diminue si il se fait attaquer par un monstre.

2 Principales fonctionnalités (user stories)

2.1 Fonctionnalités implémentées

2.1.1 Must

Fenêtre du jeu :

La fenêtre du jeu est la première fonctionnalité que nous avons développée et que nous avons améliorée au fur et à mesure de notre avancée. La fenêtre affiche l'interface du jeu avec le monde composé de cubes, les unités (joueur et monstre), la barre de vie du joueur et l'inventaire. L'affichage de la fenêtre a été réalisée lors de l'itération 1, les éléments qui la composent ont été rajoutés par la suite, lors des itérations 2 et 3.

Déplacement du joueur :

Le déplacement du joueur a été réalisé lors des itérations 1 et 2. Le joueur peut actuellement se déplacer de droite à gauche, sauter et tomber. Le déplacement est satisfaisant car il est robuste aux interactions avec les cubes. En effet, le joueur ne peut pas traverser de blocs sauf ceux qui sont traversable (blocs de feuille).
La caméra suit le joueur lors de son déplacement et a été implémentée lors de l'itération 3.

Possibilité pour le joueur d'altérer l'état du monde :

Le joueur a la possibilité d'altérer le monde, c'est-à-dire qu'il peut détruire des blocs et/ou en rajouter sur le monde à sa guise. La portée du joueur est limitée. En effet, il ne peut casser des blocs ou en ajouter seulement à une distance limitée de 4 blocs. Cette fonctionnalité a été ajoutée durant l'itération 2 et la portée du joueur a été implémentée lors de l'itération 3.

Inventaire :

L'inventaire a été réalisé lors de l'itération 2. Il permet au joueur de pouvoir récolter ce qu'il détruit afin de le stocker. Ce dernier peut alors utiliser les items présents dans l'inventaire (cubes, objets, ...). Cet inventaire est affiché en haut à gauche de la fenêtre du jeu. Le joueur peut déplacer les items présents dans l'inventaire pour les changer de position et sélectionner un item pour le placer dans sa main à l'aide de la molette de la souris.

2.1.2 Should

Monde généré aléatoirement :

Chaque monde généré sera différent du précédent. En effet, la génération du monde est aléatoire. Selon certaines profondeurs, la probabilité d'apparition des blocs est différente afin d'être cohérent avec la rareté des différents blocs. Des arbres sont aussi générés en couche supérieure de façon aléatoire. Cette fonctionnalité a été réalisée et modifiée lors des itérations 2,3 et 4 car nous avons

souvent modifié la génération.

Monstres :

Deux types de monstre ont été créés. Les monstres sont générés au fur et à mesure du jeu autour du joueur de façon aléatoire. Ils sont capables d'infliger des dégâts au joueur lorsqu'il s'en approche de trop près. Ils ont une barre de vie au-dessus d'eux et peuvent mourir à la suite de dégâts infligés par le joueur. Cette fonctionnalité a été commencée durant l'itération 2 et approfondie lors de la troisième.

Barre de vie :

La barre de vie du joueur est représentée par une barre de coeurs en haut à gauche de la fenêtre. Le nombre de coeurs à afficher est calculé selon le rapport vie actuelle/ vie max. Une barre de vie est aussi affichée sur les monstres sous la forme d'une barre verte. Cette fonctionnalité a été implémentée lors de l'itération 3. Depuis l'itération 4, des pommes peuvent être trouvées en cassant des blocs de feuille. Chaque pomme donne un coeur.

Sauvegarde :

Les différents chunks sont sauvegardés lors du jeu et en lors de la sortie du jeu. Cette fonctionnalité a été réalisée lors des itérations 3 et 4.

2.1.3 Could

Musique/bruitages :

Lors du lancement du jeu et tout au long de la partie, une musique de fond est audible afin d'ajouter une ambiance de fond. Nous avons également ajouté des buitages à chaque action remarquable du joueur (mort d'un monstre, cassage d'un bloc, placement d'un bloc, baisse de vie du joueur). Cette fonctionnalité a été implémentée lors de l'itération 3.

Possibilité d'avoir différents modes de jeu :

Deux modes de jeu sont disponibles : le mode découverte et le mode classique. Le mode de jeu découverte se différencie par la possibilité du joueur de pouvoir voler. Cette fonctionnalité a été implémentée lors de l'itération 4.

Personnalisation du joueur et de la texture des blocs :

Lors du lancement du jeu, il est possible de choisir la texture des blocs, des monstres et du joueur. Cette fonctionnalité a été implémentée lors de l'itération 4.

Accomplissements :

Afin de terminer le jeu, des accomplissements doivent être réalisés. Ces accomplissements tiennent compte des statistiques du joueur (nombre de blocs cassés, nombre de mobs tués, ...). A chaque accomplissement réalisé, une récompense particulière pour cet accomplissement est donné au joueur, jusqu'à ce qu'il gagne ou meurt. Cette fonctionnalité a été implémentée lors de l'itération 4.

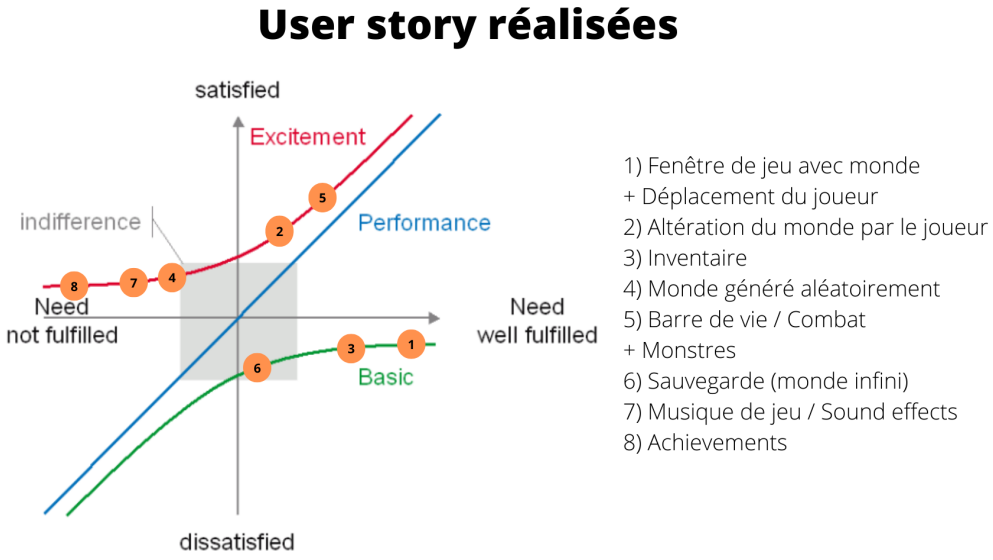

Figure 1: Fonctionnalitées réalisées

2.2 Fonctionnalités non implémentées

2.2.1 Must

Toutes les fonctionnalités prévues dans cette section ont été implémentées.

2.2.2 Should

Craft de nouveaux objets :

Nous n'avons pas pu développer le système de craft pour le joueur. En contrepartie, nous avons décidé, lors de l'itération 4, que le joueur pourrait obtenir des objets (pioche, épée, ...) au bout d'un certain nombre de ressources récoltées.

2.2.3 Could

Animaux :

La classe unité permettrait d'ajouter facilement des animaux qui ne sont rien de plus que des monstres qui n'attaquent pas le joueur et qui laissent tomber de la nourriture. Mais comme ils n'apportent rien pour le jeu, nous avons choisi de ne pas l'implémenter.

Barre de faim :

Sans les animaux qui donnent de la nourriture, il était inutile d'ajouter une mécanique de fin dans le jeu. Néanmoins, cette barre serait gérée de la même manière que la barre de vie, en perdant une nourriture dans un délai donné.

Possibilité de modifier les touches de commandes :

Il est possible de modifier les touches de commandes avec l'architecture utilisée pour les entrées clavier/souris mais cela n'a pas été réalisé car cela aurait demandé de faire de l'analyse d'un fichier de configuration, ce qui aurait ralenti le développement du jeu pour une fonctionnalité que nous estimons sans grand intérêt pour un joueur lambda.

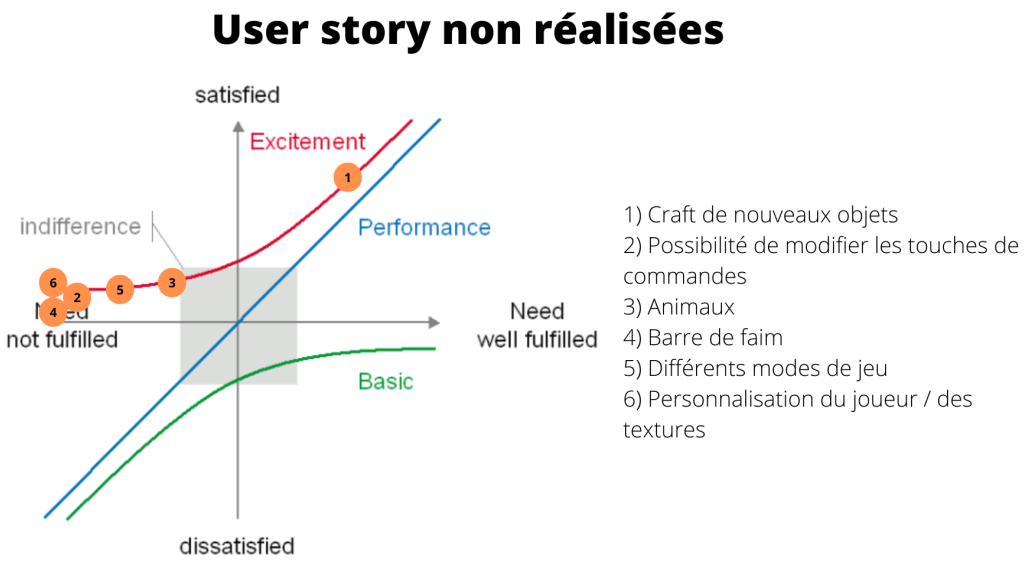

Figure 2: Fonctionnalitées non réalisées

3 Découpage de l'application en sous-systèmes

3.1 Main

Le package main permet le démarrage du jeu et gère son déroulement.

3.2 Blocks

Le package blocks implémente tous les blocs et la génération du monde. Il contient également le sprit des différents blocs.

3.3 Unites

Le package unites implante les différentes unités du jeu et permet leur gestion
(personnage, monstres, ...).

3.4 Affichage

Le package affichage contient les méthodes nécessaires à l'affichage des différents objets du jeu.

3.5 Utils

Le package utils contient les différents objets utiles pour l'implantation du reste de l'application.

3.6 Hitbox

Le package hitbox permet de délimiter les blocs et les unités pour les collisions.

3.7 Items

Le package items implante la représentation des différents objets de l'inventaire.

3.8 Save

Le package save implante la gestion de la sauvegarde des différents éléments du jeu.

3.9 Sound

Le package sound permet la gestion de la musique du jeu en arrière-plan et des sound effets qui peuvent arriver en jeu.

3.10 Achievements

Le package achievements défini et met en place les différents accomplissements du jeu.

4 Diagrammes de classe qui donnent l'architecture de l'application

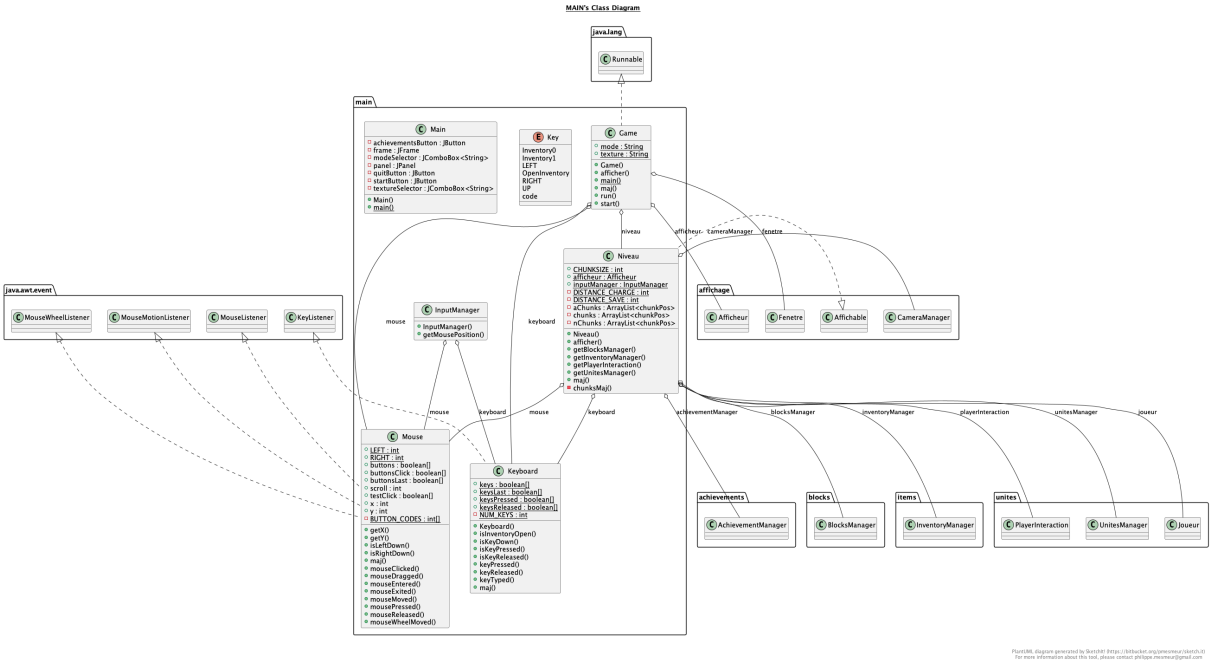

Figure 3: UML du package main

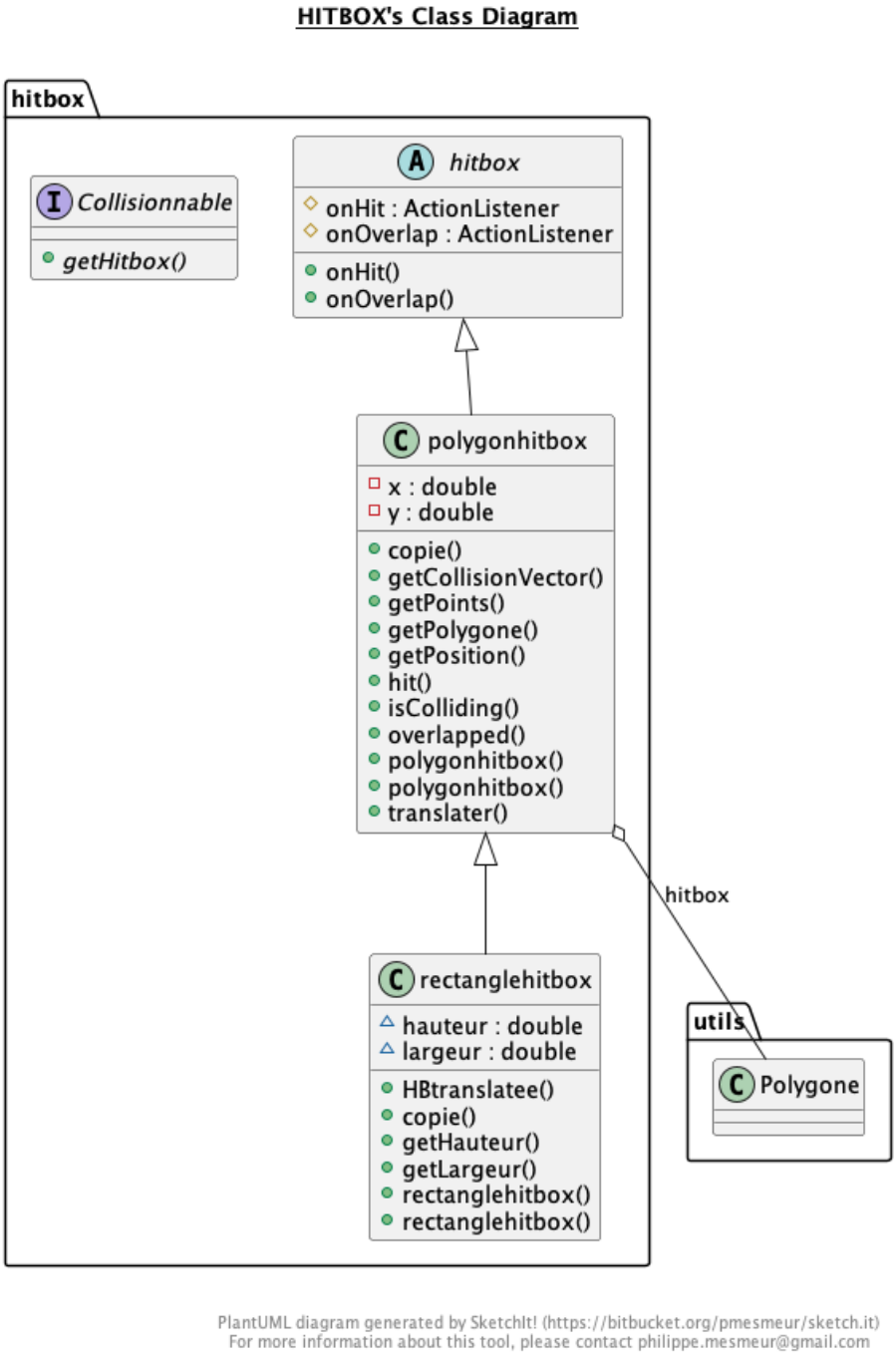

Figure 4: UML du package hitbox

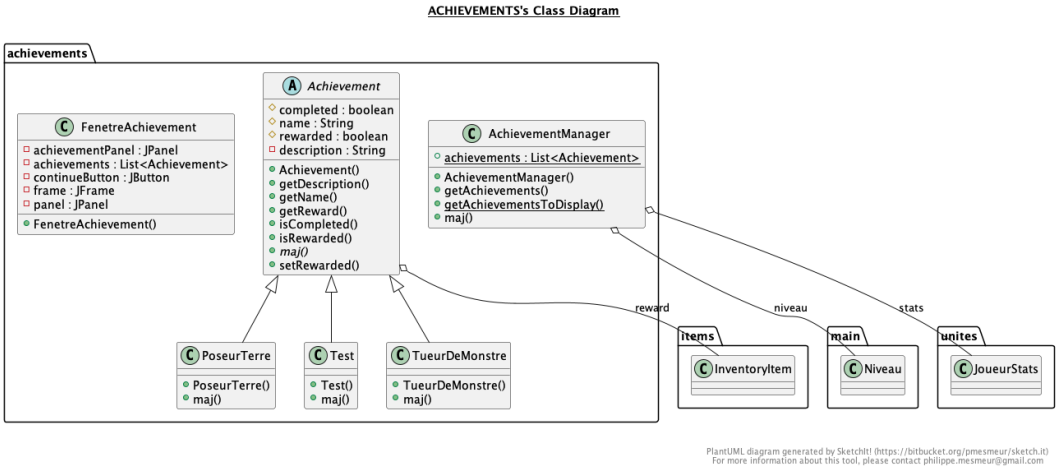

Figure 5: UML du package achievements

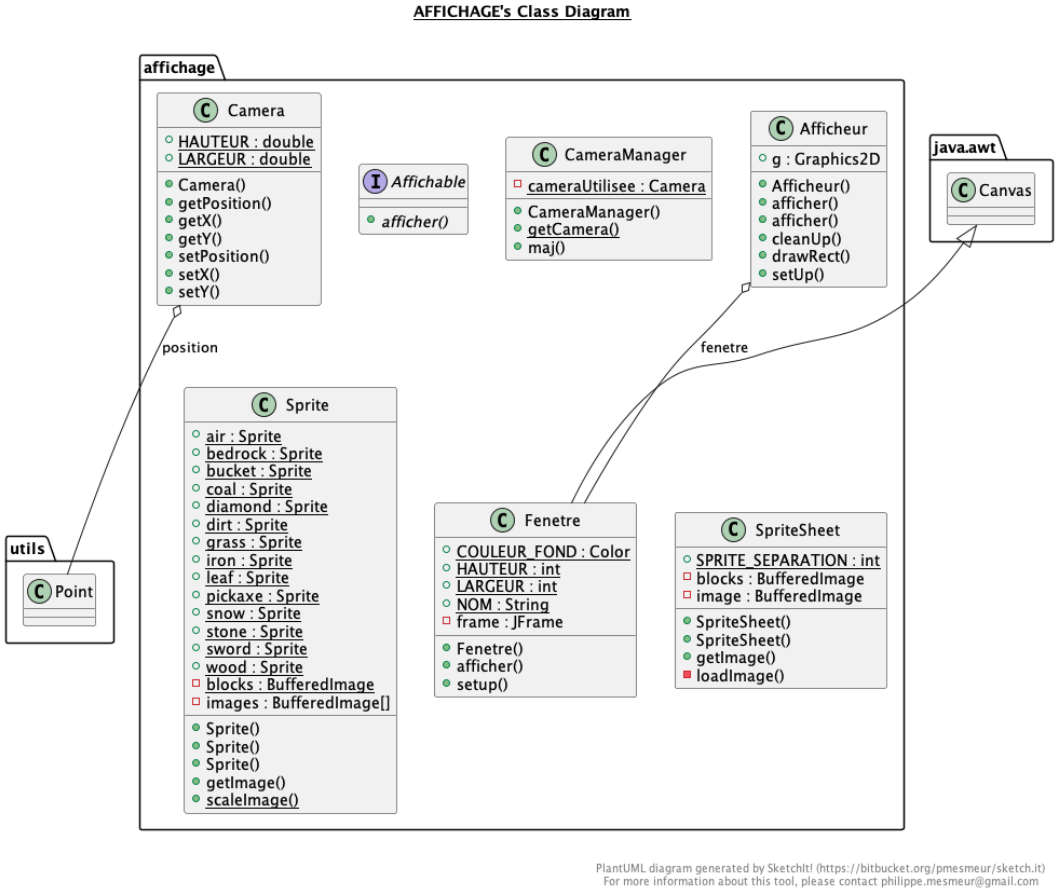

Figure 6: UML du package affichage

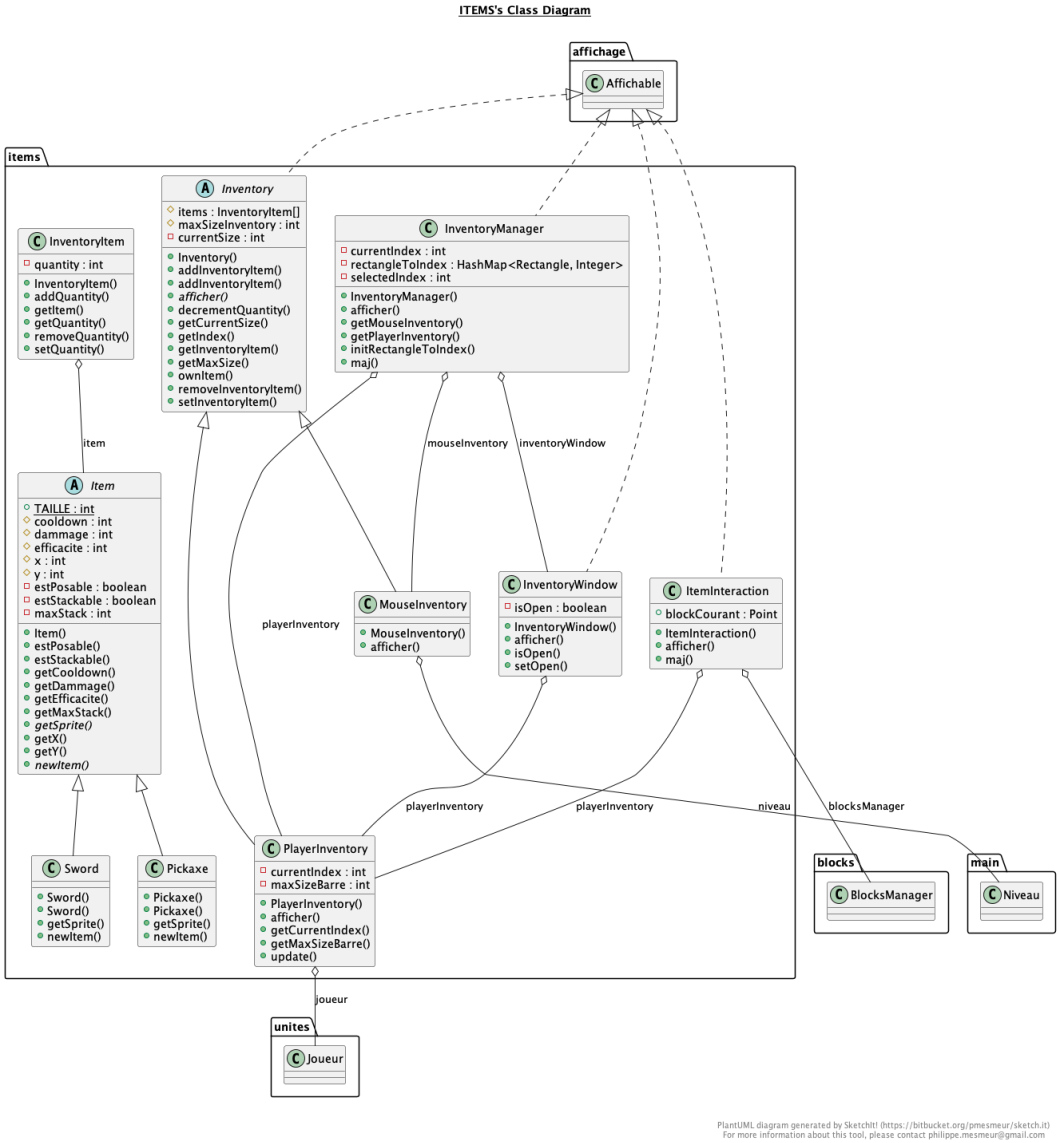

Figure 7: UML du package items

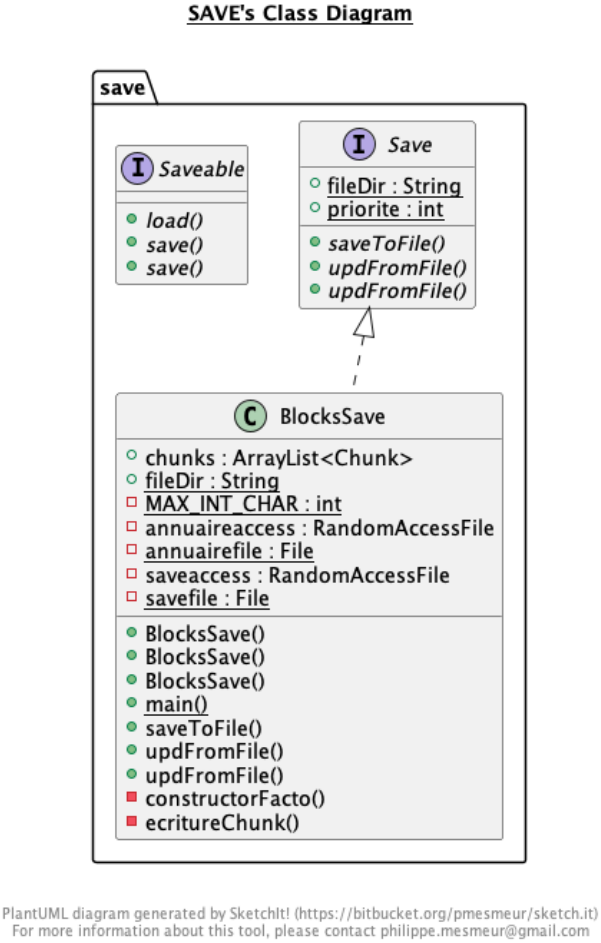

Figure 8: UML du package save

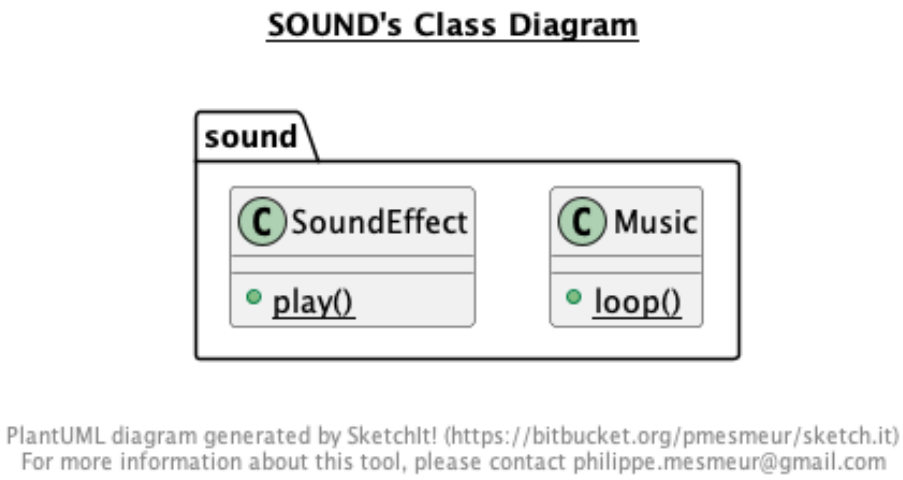

Figure 9: UML du package sound

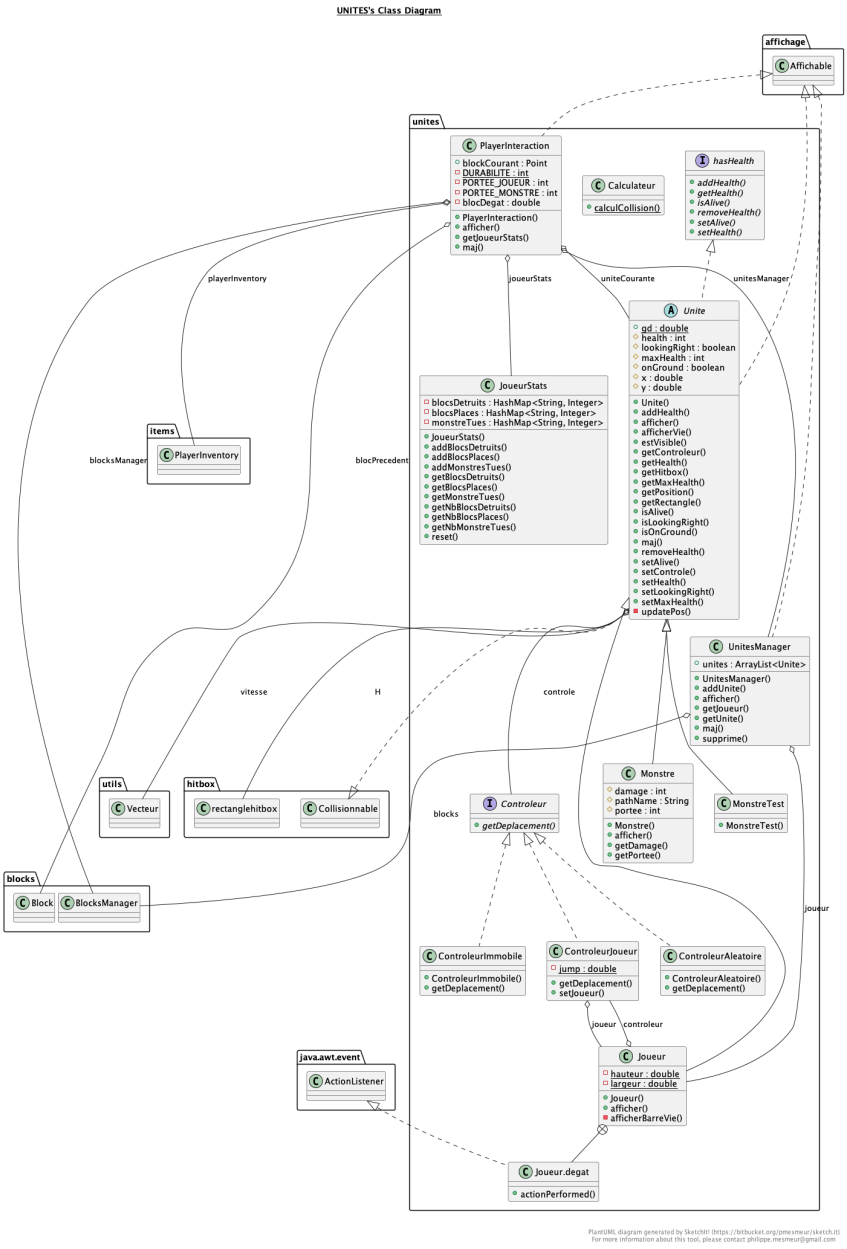

Figure 10: UML du package unites

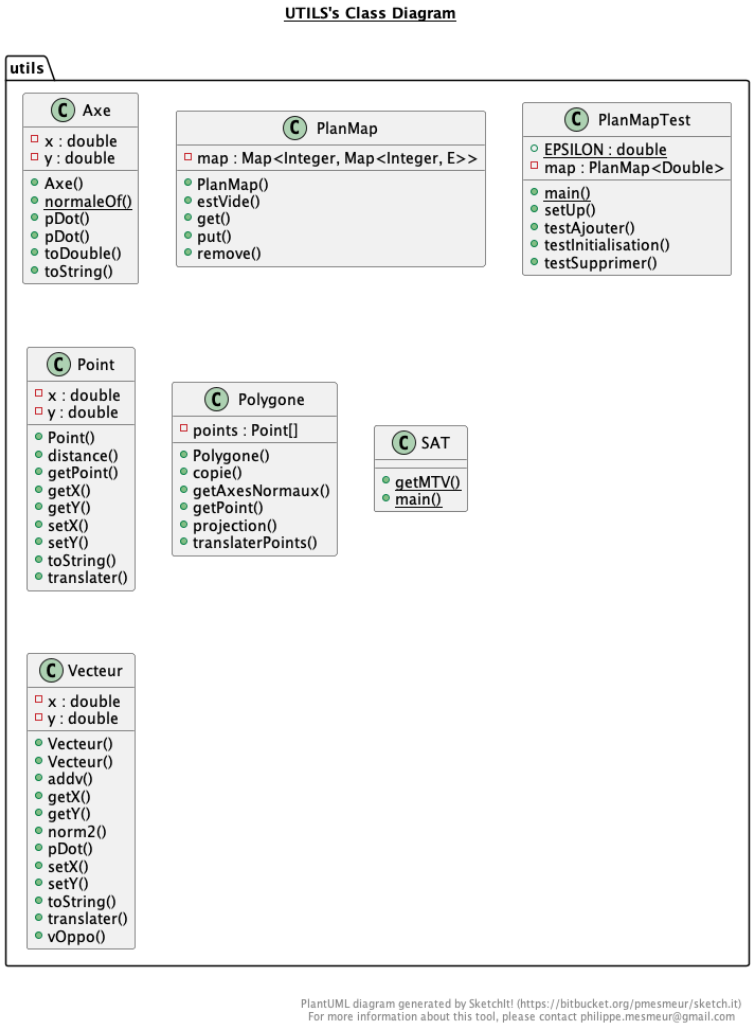

Figure 11: UML du package utils

5 Principaux choix de conception et de réalisation, problèmes rencontrés, solutions apportées

5.1 Principaux choix de conception et de réalisation

5.1.1 Main

Pour avoir une expérience conviviale pour l'utilisateur en lançant le jeu, une fenêtre permettant de configurer et comprendre le but du jeu a été ajoutée à l'aide de Swing.
Ainsi, lorsque le bouton Start est cliqué, on bascule sur le jeu en lui même qui est composé d'une boucle infinie qui met à jour la logique interne du jeu à un nombre constant de ticks par seconde puis affiche la logique interne sur une fenêtre. Un système de thread a été utilisé pour pouvoir utiliser d'autres fonctionnalités en parallèle comme la musique par exemple. La logique interne du jeu est contenu dans un objet de type Niveau qui contient notamment les entrées clavier/souris et les nombreux managers s'occupant par exemple des Unités, des Blocs, de l'inventaire... Cette boucle de jeu était l'architecture la plus utilisée pour créer un jeu et n'a donc pas beaucoup été remise en question lors du développement du jeu.

5.1.2 Inventaire

Pour implanter l'inventaire en suivant le modèle de minecraft, il fallait pouvoir déplacer des items dans l'inventaire tout en pouvant interchanger deux items.
L'architecture de l'inventaire comporte donc l'inventaire du joueur et l'inventaire de la souris dont le but est de stocker un item temporairement pour pouvoir le replacer dans l'inventaire plus tard. Afin de factoriser le code commun entre les deux types d'inventaire on a utilisé une classe abstraite Inventory. Il fallait ensuite pouvoir faire le lien entre les deux inventaires, ce qui a donné naissance à la classe InventoryManager et l'affichage de l'inventaire est géré à l'aide de InventoryWindow.
Enfin, l'inventaire permet de stocker des InventoryItem qui sont composés d'un Item et d'une quantité, tout objet du jeu étant un Item.

5.1.3 Sauvegarde

Pour rendre la sauvegarde versatile et indépendante du traitement matériel des objets, une interface Saveable a été réalisée dans le but d'en faire hériter les objets à sauvegarder. Elle renvoie et utilise également des objet "Save", une interface qui précise les fonctions à implémenter pour passer de la save à la sauvegarde disque et inversement. Ainsi, les informations à sauvegarder sont précisées par l'objet à sauvegarder uniquement. La save associée étant relative à une classe d'objet particulière, la structure de la sauvegarde sur le disque peut être adaptée et optimisée.

5.1.4 Affichage

Pour représenter les images utilisées pour le jeu on utilise une classe Sprite. La classe Fenêtre symbolise la fenêtre de jeu. La classe essentielle pour réaliser les affichages est la classe Afficheur. Cette classe propose des méthodes simples et adaptées à nos besoins pour utiliser la classe Graphics2D de la bibliothèque Java.
La deuxième partie du paquetage tourne autour de la caméra. On a réalisé une classe Camera et une classe CameraManager. Le Manager s'occupe de déplacer la ou les caméras et de transmettre à l'afficheur quelle caméra utiliser pour réaliser les affichages. Cette organisation permet d'adapter l'affichage à tout type de déplacement de caméra.

5.1.5 Hitbox

Pour les hitbox, le choix a été fait d'implémenter un attribut polygone, représentant la forme d'une hitbox. Ce choix a été fait afin de rendre la hitbox plus adaptative afin de pouvoir passer

d'une hitbox rectangulaire à une autre forme (exemple : escalier, triangle, ...). Ce choix présente en contre-partie le défaut de rendre l'implémentation de la hitbox fastidieuse.

5.1.6 Chunks

Pour stocker les blocs, on a décidé d'utiliser le principe de chunks de minecraft. On découpe le monde en zones (bandes, carrés) de taille constante appelées chunks. On sauvegarde et charge les blocs par chunks vers/depuis un fichier de sauvegarde. On a choisit pour notre projet d'utiliser des chunks carrés.
De plus, il faut déterminer une distance à partir de laquelle on sauvegarde ou charge un chunk. On a décidé de prendre une distance de sauvegarde supérieure à celle de charge car ainsi, si le joueur reste dans une zone limitée, on réalise très peu (si ce n'est aucune) d'opération mémoire.

5.1.7 Génération aléatoire

La génération aléatoire s'effectue à chaque génération de chunks. Pour générer, on a décidé de parcourir la map du haut vers le bas. En parcourant la Map de haut en bas, on prend blocs par blocs la décision de poser un bloc ou non avec une méthode qui renvoie un booléen (on doit placer un bloc, ou on ne doit pas placer un bloc). Cette décision est prise en fonction de la profondeur (Si la profondeur est basse on génère, si la profondeur est haut on génère selon une certaine probabilité).
Si un bloc doit être posé, on prend la décision de quel bloc doit être posé en fonction de la profondeur. En effet selon la profondeur la probabilité d'avoir tel ou tel bloc change. (exemple : plus de chance d'avoir des diamants en profondeur que dans les couches hautes). De plus, si il s'agit d'un bloc en surface, on génère des types de blocs particuliers (bloc de neige, bloc de terre ou arbre).

5.1.8 Achievements

Achievements est une classe abstraite dont la méthode maj a vocation à être définie selon l'objectif à remplir pour obtenir la récompense. On a ensuite rassemblé les différents Achievements qui sont centralisés dans AchievementManager. Pour indiquer au joueur quels achievements sont disponibles, on a ajouté un FenetreAchievement dans le menu de départ.

5.2 Problèmes rencontrés et solutions apportées

5.2.1 Rendre le jeu évolutif

Le premier problème auquel nous avons été confronté était de rendre le jeu voulu le plus évolutif possible. De plus, le jeu que nous voulions réaliser comportait beaucoup de fonctionnalités à implanter et le choix des structures de données et des classes conditionnent beaucoup de choses.
Ainsi, afin de conserver la possibilité de changer des choses dans le futur, le choix a été fait de découper le plus possible les différents composants de l'inventaire pour pouvoir faire des changements au niveau local de chaque fonctionnalité sans impacter le comportement de l'ensemble du module.

5.2.2 Sauvegarde

Lors de la sauvegarde des chunks, nous avons rencontré un problème vis-à-vis des indices négatifs. Il s'avère que plusieurs problèmes se produisaient. En conséquence, les chunks d'indices négatifs n'étaient pas chargés correctement dans le jeu.
Le premier problème est dû aux définitions des opérateurs mathématiques usuels en Java (/, % et int) sur les entiers négatifs. Ces opérateurs arrondissent à l'entier le plus proche de zéro et non pas à l'entier inférieur.
Le second problème était lié à la traduction d'entiers en caractères, les entiers négatifs sont vu

comme des entiers positifs modulo 2^{16}. En modifiant les opérateurs par ceux du paquetage Math et en appliquant un modulo sur les indices, les problèmes ont pu être corrigés.

5.2.3 Génération du monde

Lors de l'ajout de la sauvegarde en chunks, la génération du monde n'était plus adaptée. Cela est dû au fait que la génération était initialement pensée pour un monde fini généré en une fois. La solution apportée est l'ajout d'une "ligne de niveau" qui désigne la hauteur du premier bloc sur une colonne donnée. Quand un chunk est généré même en profondeur la ligne de niveau est mise à jour et le chunk se génère correctement.

5.2.4 Temps

Le problème principal auquel nous étions confronté pendant tout le long du projet a été le temps. En effet, nous souhaitions implémenter beaucoup de fonctionnalités et il était impossible de toutes les implanter. Nous avons alors, tout au long de notre projet, choisi les fonctionnalités essentielles à notre jeu et nous nous sommes concentré sur ces dernières. Nous avons également remplacé certaines fonctionnalités comme les différents crafts qui ont été remplacés par des accomplissements.

6 Organisation de l'équipe et mise en oeuvre des méthodes agiles

6.1 Organisation

Pour nous organiser dans ce projet, nous nous sommes inspirés des méthodes agiles présentées en cours.
Nous nous sommes alors organisés sur un discord commun avec différents channels pour échanger sur les différentes tâches, la progression et les potentielles difficultés de chacun tout au long du projet. En plus de cela, svn nous a permis de mettre en commun nos codes. Toutes les deux semaines, soit une fois par itération, nous organisions des réunions afin de définir les futures user-stories à implémenter.

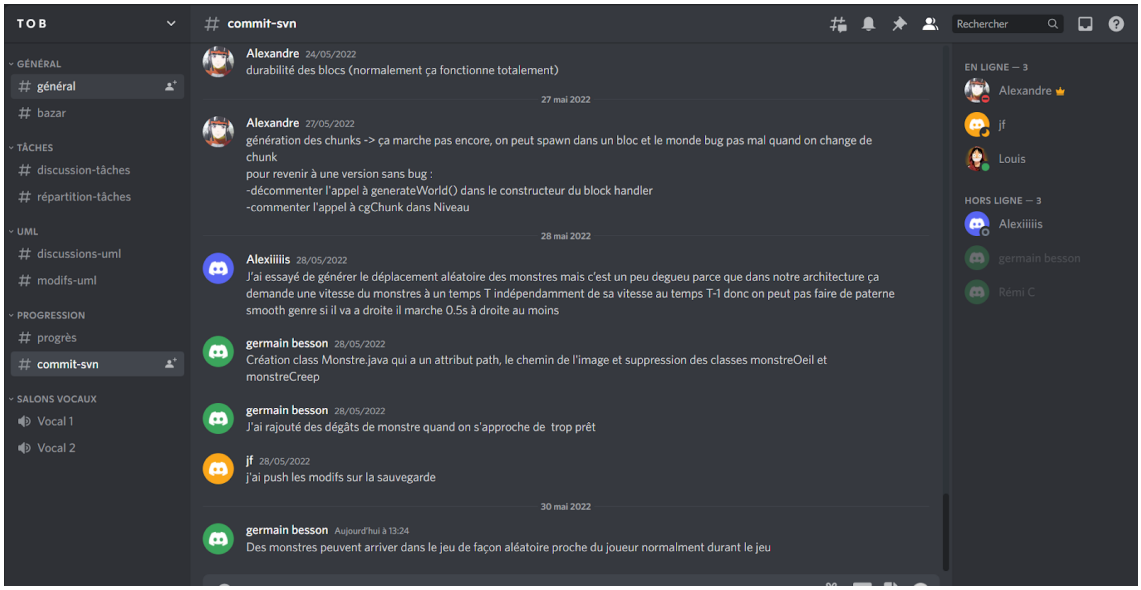

Figure 12: Organisation du serveur discord

6.2 Mise en oeuvre des méthodes agiles

Après une phase d'analyse du sujet, nous avons défini les fonctionnalités du jeu en leur attribuant différents niveaux de priorité (must, should, could) afin de savoir l'importance de ces dernières pour notre jeu. Nous avons ensuite reporté ces dernières sous la forme d'un diagramme de kano afin de choisir quelles fonctionnalités implémenter en premier lieu, lesquels sont les plus essentiels.
Nous avons également, afin de mettre en oeuvre les méthodes agiles, utilisé l'application scrum poker à chacune de nos réunions bimensuelles afin de définir ensemble les efforts à associer à chacune des user-stories de la future itération.

6.3 Effort/vélocité

Itération 1

Fonctionnalités	Story point évalué (effort)	Story point réel (effort)
- Fenêtre graphique - Génération déterministe du monde - Liaison entrée clavier/souris et méthodes du jeu - Architecture du jeu	? ? ? ?	2 2 2 5
Total	**?**	**11**

Figure 13: Efforts des user-stories de l'itération 1

Nous pouvons remarquer que nous avons assez bien évalué les efforts en fonction des user-stories pour l'itération 1. Cependant, la vélocité du groupe lors de cette dernière n'était pas très élevée et une erreur dans un effort représente alors une part importante des efforts globaux. Cette faible vélocité s'explique toute fois par la mise en place du projet lors de cette itération.

Itération 2

Fonctionnalités	Story point évalué (effort)	Story point réel (effort)
- Inventaire	8	9
- Alteration du monde et liaison avec l'inventaire	5	5
- Affichage d'un joueur pouvant se déplacer	3	3
- Détection de collision : hitbox	4	6
- Génération aléatoire du monde	5	5
- Restructuration de l'affichage	5	5
Total	**30**	**33**

Figure 14: Efforts des user-stories de l'itération 2

Nous avons, lors de l'itération 2, nettement augmenté la vélocité du groupe, qui a quasiment triplée. Cela est toute fois logique puisque lors de l'itération 1, il fallait mettre en place tout le projet et partir de 0. Les efforts étaient encore une fois assez bien évalués.

Itération 3

Fonctionnalités	Story point évalué (effort)	Story point réel (effort)
- Génération aléatoire d'arbre - Sauvegarde - Musique et Sound effects - Camera se déplaçant avec le joueur - Limites dans la pose de blocs - Barre de vie, Combat et unité ennemie	4 9 2 4 2 7	3 12 2 6 2 7
Total	**28**	**32**

Figure 15: Efforts des user-stories de l'itération 3

Lors l'itération 3, la vélocité est restée constante par rapport à l'itération 2 et nos efforts étaient très bien estimés, sauf pour la sauvegarde.

Itération 4 (sur une semaine)

Fonctionnalités	Story point évalué (effort)	Story point réel (effort)
- Correction de la sauvegarde	6	6
- Achievements	5	5
- Génération et déplacement aléatoire des monstres	3	3
- Adaptation de la génération du monde	4	3
- Menu du jeu avec 2 modes différents et choix de texture	5	5
Total	**23**	**22**

Figure 16: Efforts des user-stories de l'itération 4

Nous pouvons constater une baisse de vélocité lors de l'itération 4. Hors cette baisse est logique car cette itération a duré deux fois moins de temps que les autres. On peut néanmoins remarquer que lors de l'itération 4, les efforts pour les user-stories étaient bien évalués, ce qui présente une nette amélioration par rapport aux autres itérations.