QGL – RAPPORT FINAL

Équipe Soyouz

SI3 | 2020–2021

ÉQUIPE FISÆ SOYOUZ

ESCHIMESE Alexis - KEWE François - NIGET Tom - VOURIOT Emmeline

Table des matières

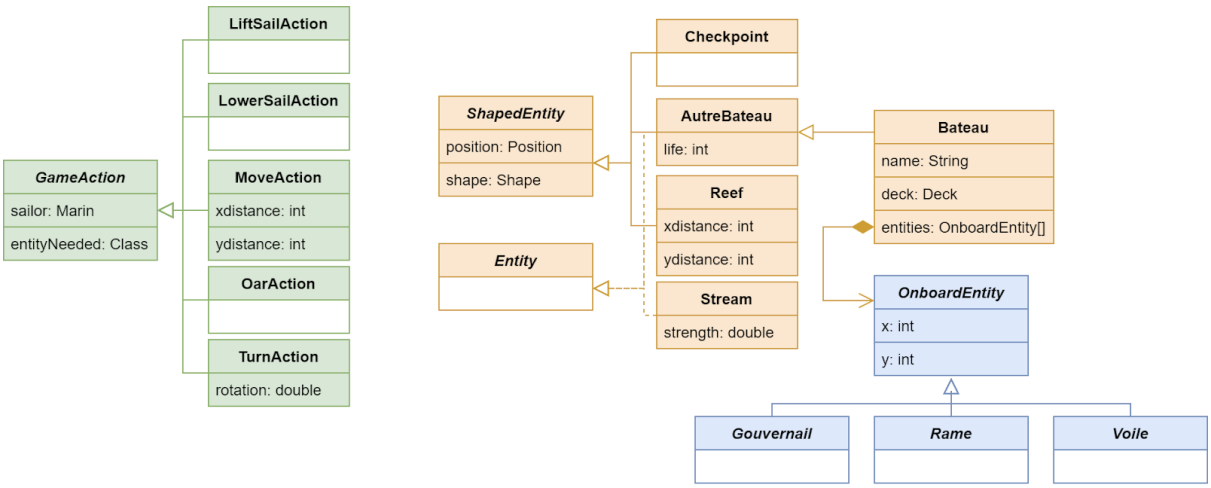
Description technique

Introduction

Lors de ce projet, nous devions faire parcourir un océan capricieux à notre navire, pour lui permettre, au choix, de venir à bout d'une course composée de plusieurs étapes, ou bien d'être le dernier survivant d'une longue et sanguinaire bataille navale. L'environnement et la composition de notre navire nous étant donnés via des fichiers JSON que nous devions parser pour en extraire les composants, nous n'avions alors plus qu'à réfléchir à la question suivante : comment atteindre notre objectif le plus rapidement possible ?

Approche « composants »

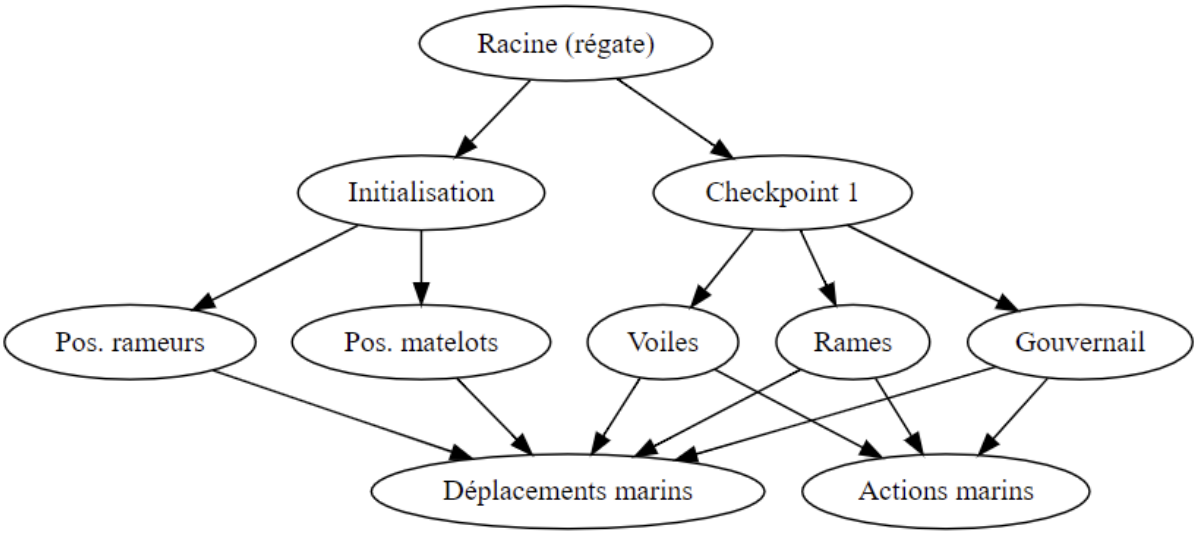
Pour commencer, et avant d'aborder l'architecture de notre phase de prise de décision, nous avons décidé d'appliquer les connaissances assimilées lors du PS5 en mettant en place un maximum d'abstractions sur nos composants. Ces dernières s'articulant, pour la plupart, autour d'un même noyau, nous avons alors mis en place des interfaces et classes abstraites nous assurant une bonne extensibilité du projet, sans avoir à apporter de modification aux composants existants. Pour ce faire, nous avons appliqué une séparation claire entre notre couche modèle et notre logique métier, appliquant un faible couplage entre ceux-ci. Ainsi, une fois les tests sur ces classes effectués, nous avons pu les mettre de côté pour tout le reste du projet, et ne plus nous en soucier, pour nous concentrer pleinement sur le développement de nos objectifs.



Approche « objectifs »

Nous avons mûrement réfléchi à l'approche à envisager pour la prise de décision du bateau. Pour terminer une course, devons-nous passer par l'ensemble des objectifs ? Pour atteindre un objectif, devons-nous ramer ? Tourner ? Et pour cela, devons-nous déplacer un marin sur l'entité correspondante ? Il nous parut alors de circonstance d'adopter un modèle de

graphe dirigé acyclique, permettant parfaitement de modéliser nos besoins. Le principe est simple : l'objectif racine détermine le type d'objectifs qui en découle ; s'adaptant aux différents buts : course et bataille navale. Les besoins pour résoudre cet objectif déterminent alors comment initialiser la position des marins en fonction des entités nécessaires. Puis, une fois ces marins placés, l'objectif racine est découpé en plusieurs sous-objectifs à atteindre. Ces derniers sont communiqués à l'étage inférieur ; les objectifs internes au bateau ; permettant alors aux différents marins de se déplacer sur les entités requises et d'effectuer les actions qui leur ont été attribuées. Tout cela dans le but final de s'approcher au maximum de l'objectif initial.



Cette manœuvre permet à l'arbre d'être totalement ouvert à l'extension. Il est tout à fait possible d'ajouter de nouveaux objectifs racines ou bien de nouveaux objectifs internes au bateau en conservant intégralement l'implémentation des classes déjà existantes. Bien que nous ayons mis du temps à démarrer, c'est cette même configuration qui nous a permis, sur les dernières semaines, de pouvoir ajouter des fonctionnalités toutes plus complexes les unes que les autres, sans devoir nous soucier d'adapter le code présent. Chaque objectif est clair, précis et possède une responsabilité unique. Une fois installé et correctement testé, il n'est alors plus nécessaire de s'en soucier. Ainsi, en plus de nous offrir un précieux gain de temps lors d'ajout de fonctionnalités, cet arbre respecte parfaitement les principes *SOLID*, point essentiel de ce projet.

Nous avons donc un arbre d'objectifs totalement extensible, mais ce n'est pas son seul point fort. La présence de nombreuses classes *Helper*, facilitant la prise de décisions dans nos objectifs, permet d'obtenir un très faible couplage entre chacune des classes. Les objectifs ne dépendant alors que d'eux-mêmes et éventuellement d'un *Helper* ; les déconnecter de l'arbre pour retirer certaines fonctionnalités de notre bateau est alors un jeu d'enfant, ce qui est particulièrement utile lors d'un retour à une configuration de bateau bien plus archaïque. Notre projet est alors bien rétrocompatible.

Prise de recul

Au sein du groupe nous sommes tous fiers de la puissance et de la flexibilité de notre arbre d'objectifs. Nous sommes satisfaits de ce choix et n'hésiterons pas à l'utiliser de nouveau dans de futurs projets si cela se prête au sujet (par ailleurs, l'idée de cette architecture venait initialement du projet Takenoko de PS5 où elle avait fait ses preuves). L'architecture nous aura permis d'ajouter des fonctionnalités rapidement et aisément tout au long du projet, indépendamment de leur complexité d'implémentation. Si nous avions cependant un point à reprocher à cette approche, il s'agirait sans aucun doute des efforts que cela demande au démarrage. L'arbre étant quelque chose de très abstrait, il nous aura fallu de longs moments de réflexions afin de correctement démarrer. Il a fallu concevoir les interfaces que les objectifs implémenteraient, ainsi que les objectifs racines et les premiers nœuds alors que nous ne disposions pas de l'ensemble des consignes. L'objectif étant d'ajouter de nouveaux nœuds à la suite de ces derniers, il est alors primordial de correctement poser les bases, tâche qui s'est révélée assez coûteuse en temps et en réflexions. Mais une fois ce travail achevé, les difficultés à surmonter pour ajouter de nouveaux nœuds sont devenues nulles voire inexistantes. À défaut d'avoir réussi à implémenter cette approche dès le début, nous nous sommes contentés d'une approche beaucoup moins propre et longue à adapter durant les premières semaines. De fait, quelques *refactoring*s globaux ont été entamés en vain. Les efforts réclamés pour ajouter de nouvelles fonctionnalités étaient importants, pressant notre besoin d'implémenter l'architecture d'arbre d'objectifs.

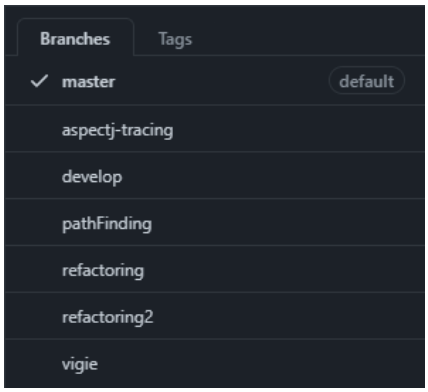
Les composants et les objectifs ayant été implémentés en suivant au mieux les concepts *SOLID*, nous n'avons pas eu de réels soucis à atteindre toutes les métriques et à assumer les rushs de fin de projet, comme avec le *pathfinding* (recherche de chemin) par exemple. Nous avons alors entièrement pu nous concentrer sur les finitions et sur l'application des autres concepts étudiés lors de ce module et dont nous allons vous parler sous peu.

Application des concepts vu en cours

Branching strategy

Dès le début de ce projet, nous avons commencé à utiliser une version très simpliste de la stratégie Git Flow. Une branche “develop” sur laquelle nous développions tous ensemble puis une branche master sur laquelle nous effectuions les fusions hebdomadaires pour valider tous les changements. Avec le temps, nous avons utilisé un peu plus en profondeur cette stratégie en créant des branches temporaires autour de la branche “develop”. Par exemple, tout gros refactoring est développé sur une nouvelle branche “refactoringX” qui une fois terminée et validée, est fusionnée sur “develop”.

Ensuite, notre groupe étant constitué de FISE (Alexis, Tom et Emmeline) et FISA (François), nous avons dû adapter notre flow de développement. En effet, François a développé sur une branche à part (*feature branch*) afin de travailler sur les éléments de la week suivante ; pour ne pas être retarder l’équipe étant donné son emploi du temps. Par exemple, la recherche du *pathfinding* a été effectuée une semaine à l’avance sur une autre branche ; grâce à cela nous avons déterminé que l’algorithme *A** était un bon choix. Cette stratégie permettait de pallier le décalage entre nos horaires de travail mais aussi de prendre de l’avance lors du développement. Ainsi chaque fonctionnalité à implémenter pour la semaine suivante donnait lieu à une *feature branch*.

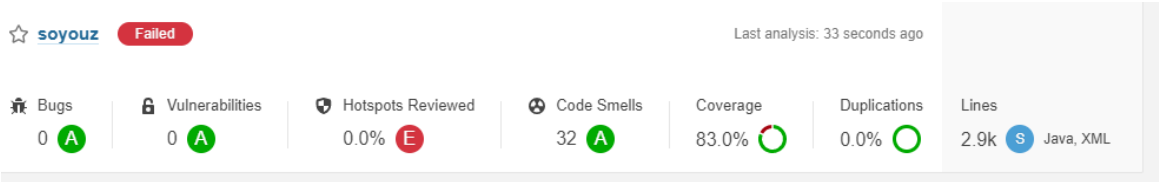


Qualité du code

En dépit de notre flow de développement, nos métriques qualité n’ont pas toujours été au rendez-vous. Nous mettant la pression pour terminer les courses chaque semaine, nous en avons même oublié l'intitulé de ce cours sur les premières semaines. Cette négligence sur la qualité du code s’est aussitôt retournée contre nous, avec principalement l’omniprésence d’échecs sur les premières courses. La capture d’écran suivante n’est qu’un reflet insatisfaisant de la qualité du code de notre projet à ce moment-là.

Analyse du 25/02/2021

C'est au cours des semaines suivantes que nous avons décidé de nous reprendre en main et de pallier ce problème de qualité. Nous avions alors entamé de conséquents refactorings, à l'issue desquels, nous avons pour la première fois réussi une course avec un bon classement.



Analyse du 28/03/2021

Ce résultat a aussi pu être atteint grâce à un refactoring majeur sur notre architecture de déplacement des marins. En effet, notre premier fonctionnement reposait sur une seule classe comportant énormément de lignes et de méthodes à forte complexité. Si l'algorithme choisi nous permettait d'anticiper absolument tous les cas possibles, il n'était pas pour autant viable, étant peu extensible car bien trop complexe. Nous avons alors décidé de complètement recommencer en nous basant sur une nouvelle architecture beaucoup plus réfléchie, notre fonctionnement précédent étant beaucoup trop difficile à tester et à maintenir tandis que le besoin d'évolution nous amenait à devoir constamment l'adapter, une tâche bien trop chronophage. Nous sommes alors repartis sur notre stratégie des objectifs pour y baser le placement des marins. Cette solution étant bien plus *SOLID* et simple à maintenir.

C'est en repartant de cette base de code robuste que nous avons implémenté les fonctionnalités supplémentaires de chaque semaine. Une fois cette optimisation faite, la suite du développement était intuitive et nous n'avons rencontré aucune difficulté menaçant nos métriques. Les fonctionnalités sont arrivées semaines après semaines, naturellement suivies de tests unitaires, et tout cela sans que l'on observe de quelconque dégradation en termes d'efficacité de nos algorithmes. Ainsi nous avons pu maintenir notre code et les concepts *SOLID* jusqu'à la fin du projet.

SonarQube nous aura alors été d'une grande aide pour analyser notre code avant chaque rendu hebdomadaire afin de drastiquement réduire la quantité de bugs ou encore de vérifier la bonne couverture des tests. À chaque analyse nous prenions le temps de réduire au maximum les *code smells* pour avoir un projet le plus compréhensible possible.

Les tests unitaires étant le premier rempart face aux bugs et comportements inattendus, nous avons alors mis un point d'honneur à couvrir le maximum de nos lignes avec des tests unitaires. Cependant il est difficile de juger leur pertinence. *Pitest*, avec ses tests de mutation, nous aura permis, tout au long du projet de garantir la qualité de nos tests en vérifiant notamment leur utilité. Si un test est déclaré comme insuffisant, nous venons alors le renforcer pour couvrir d'avantages de cas possibles. Nous avons effectué des analyses *Pitest* à la même fréquence que *SonarQube* et avons réussi à augmenter la couverture globale. Une méthode correctement testée n'étant plus à tester, *Pitest* nous aura permis d'économiser beaucoup de temps de débogage de méthodes mal testées.

Enfin, comme nous avions l'habitude de le faire sur nos anciens projets, nous avons mis en place un serveur d'intégration continue dès le démarrage du projet, afin de nous assurer que le projet compilait correctement après chaque push ; le but étant d'avoir toujours un projet fonctionnel.

Étude fonctionnelle et outillages additionnels

Stratégies implémentées

Pendant les premières semaines, notre logique de gestion de cap était très simple, voire simpliste : faire avancer le bateau dans la direction du prochain checkpoint, coûte que coûte. Bien sûr, l'ajout des récifs sur le chemin du bateau à la semaine 8 a changé la donne, et nous avons dû mettre en place une véritable logique de *pathfinding*. Bien qu'une approche basée sur la discrétisation (modélisation de la carte par une grille de cases) nous fût présentée en cours, nous nous sommes vite rendu compte que cette méthode était suboptimale du fait de la nature des cartes (grand « espace vide » avec quelques polygones). À la place, nous avons simplement appliqué l'algorithme *A** standard sur un graphe constitué des sommets des polygones présents sur la carte (voir capture d'écran du simulateur). En plus de beaucoup mieux s'adapter à cet algorithme (qui est, avant tout, un algorithme de *parcours de graphe quelconque*), cette méthode nous permet d'obtenir des chemins passant très proche des formes (donc plus courts), avec un impact très faible sur les performances. En outre, nous gérions d'emblée les collisions avec les bateaux et les courants, car ils ne représentaient pour notre code qu'une forme de plus dans l'océan (nous ne gérions toutefois pas le mouvement potentiel des formes, voir stratégies non implémentées).

Une fois le prochain cap déterminé, il faut réussir à l'atteindre. Voici la deuxième grande étape de ce projet. La stratégie adoptée pour la gestion des marins nous a permis d'être en fonctionnels dès le premier tour et de manière optimale. Nous récupérions simplement chaque entité présente sur le pont du bateau et les triions en fonction de leurs coordonnées. Nous faisions de même sur les marins et réalisions les associations les plus proches, de sorte que chaque marin puisse atteindre une entité dès son premier tour de jeu. À présent, les marins sont ordonnés en fonction de leur rôle et appelés lorsque cela était nécessaire. Il existe deux types de marins, les marins muables et immuables. Les marins immuables, comme leur nom l'indique, ne sont pas destinés à bouger de leur poste. Ces derniers se trouvent sur des entités spéciales comme le gouvernail, les voiles ou encore la vigie. Les marins muables sont positionnés sur les rames et sont amenés à se déplacer d'un côté à l'autre du bateau afin d'actionner la bonne combinaison de rames permettant de s'approcher au mieux de notre cap. Pour déterminer quelles actions sont à effectuer lors du tour de jeu, nous nous servons de helpers réalisant des calculs sur la distance et l'angle de rotation nous séparant de notre cap. Ces derniers nous renvoient une liste d'entités et une combinaison de rames à actionner, en adéquation avec les ressources disponibles.

Stratégies non implémentées

Parmi les stratégies que nous aurions pu implémenter, il y avait la prédiction de trajectoire des autres navires, au moins partielle. Nous aurions pu vérifier si les bateaux alentours auraient eu la possibilité de rencontrer notre trajectoire et donc dévier la nôtre (voire

nous arrêter totalement) afin de ne pas provoquer de collision. Cette prévision aurait cependant été très consommatrice de ressources, principalement car les bateaux concurrents n'ont pas forcément des calculs optimaux pour leur trajectoire, et donc choisir de ne pas traiter certains cas qui semblent peu probables car peu réalistes n'aurait pas été une solution envisageable.

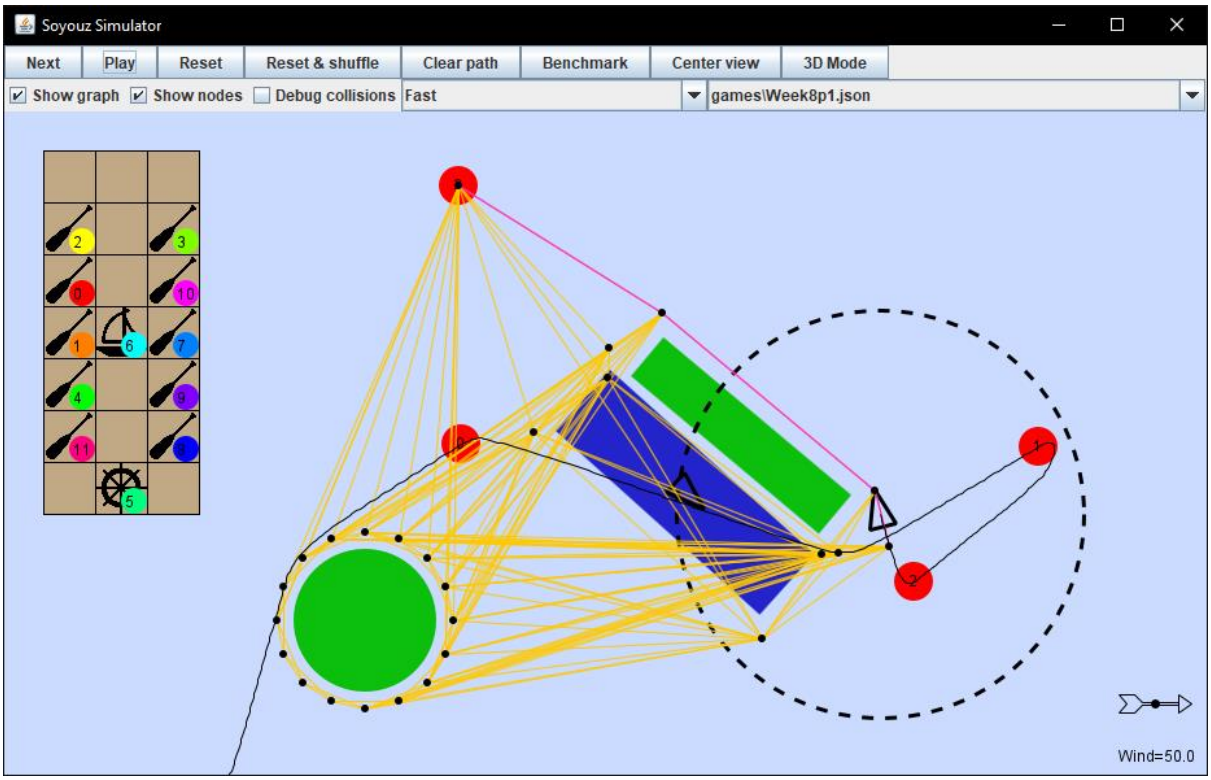
Nous aurions également pu utiliser les canons afin de faire de la place sur le parcours.

Autres outils utilisés

En raison de problèmes techniques, GitHub Actions n'était pas disponible au début du semestre, nous avons donc mis en place un serveur d'intégration continue *TeamCity* nous permettant de tester de manière automatisée le code après chaque mise à jour.

Outil créé : simulateur

Dès la première semaine, nous avons réalisé qu'il était essentiel d'avoir la possibilité de tester notre code métier en local ; à la fois pour ne pas épuiser nos crédits *WebRunner*, et pour pouvoir déboguer autrement que via les logs fournis une fois par semaine. Nous avons donc mis au point un simulateur en temps-réel, entièrement séparé du code métier, et permettant de visualiser toutes les informations essentielles à une course.

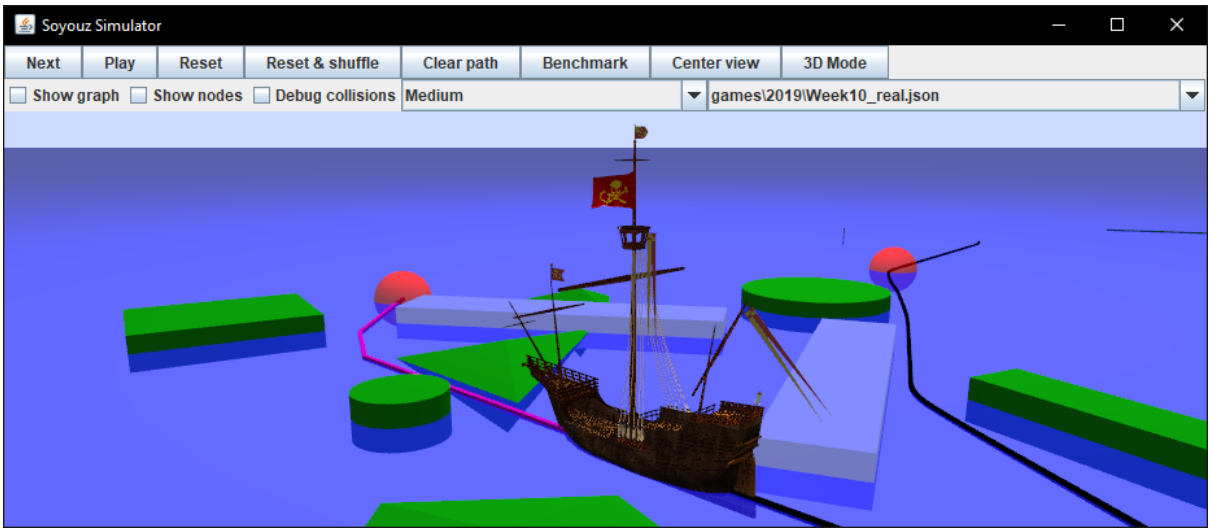


Supportant le format de configuration spécifique du *runner*, il permet de lancer directement toutes les courses, y compris celles de l'an dernier, qui nous ont permis de déceler des bugs durant le développement, de par leur complexité.

Pour un plus grand confort lors de l'utilisation, il permet de simuler la partie soit tour par tour, soit en continu, et d'accéder aux informations internes telles que la position exacte du bateau, les données des marins, du vent, et même les données internes de l'algorithme de recherche de chemin.

Ainsi, vous pouvez voir sur l'image ci-dessus les points noirs et les traits orange représentant le graphe utilisé pour la recherche ; les traits roses représentant, eux, le chemin choisi pour arriver au prochain objectif.

Notre simulateur supporte également le mode multijoueur, et peut simuler un nombre quelconque de bateaux, ce qui nous a aidé pour les dernières courses.



Le mode 3D, ajouté à la fin du projet, permet une meilleure visualisation des courses et une immersion aidant grandement aux tests.

Enfin, nous avons en parallèle de l'interface graphique, développé un outil en ligne de commande permettant de tester automatiquement et d'un seul coup toutes les courses du *runner*, en guise de tests de non-régression.

Conclusion

Pour conclure sur ce projet, nous pouvons dire que nous avons tous pu enrichir nos compétences. Notre premier objectif lors de ce module a été d'apprendre à utiliser de nouvelles technologies telles que la manipulation de JSON, ce qui a été un vrai challenge lors des premières semaines. Nous avons pu apprendre à utiliser de nouveaux outils nous permettant de nous assister dans la conquête de la qualité. *SonarQube*, *Pitest*, *Maven*, utilisés dans le monde professionnel, nous ont permis d'avoir un avant-goût de notre future vie professionnelle. Des concepts comme les principes *SOLID* nous ont permis de réfléchir constamment sur notre manière de programmer. Ainsi, nous avons effectué de nombreux refactorings tout au long de ce projet, des petits, mais aussi de plus conséquents où nous avons revu entièrement notre architecture pour la rendre encore plus *SOLID*. Toute cette réflexion nous a vraiment permis d'améliorer notre manière d'aborder le développement logiciel.

L'absence d'outil de visualisation globale en début de projet nous a également poussés à prendre l'initiative, à être plus autonomes en choisissant de créer notre propre simulateur. Les tests unitaires restent bien entendu nécessaires, mais le simulateur permettait d'avoir une approche très différente et apportait un grand gain de temps lors du débogage, afin d'identifier beaucoup plus rapidement les situations critiques ; ainsi qu'une meilleure vue d'ensemble, permettant d'aborder plus facilement de nouvelles stratégies.

De plus, notre expérience accumulée lors des précédents projets nous a grandement aidés lors de ce projet. Par exemple, nous avons, dès le départ, mis en place de l'intégration continue avec un serveur *TeamCity* afin de gagner du temps lors de notre développement.

Nous en retenons pour la suite que lors du démarrage d'un projet, il est important de prendre le temps de concevoir une bonne architecture, la plus extensible possible, le tout en mettant en place des outils d'intégration continue afin d'être le plus efficace possible ; quitte à ne pas avoir de code déployable dans les premières semaines.