Projet: DeepDiep

Le but de ce projet est de réaliser un petit jeu de combat entre robots, inspiré de `diep.io` et `arras.io`. Il y aura de la programmation orientée objet, un peu d'algorithmique, un peu de graphisme et de manière optionnelle, un peu d'intelligence artificielle (mais pas profonde).

Ce projet se découpe en plusieurs parties plus ou moins distinctes. Il est à faire en binôme. Les parties comportent quelques interdépendances. Vous devrez donc échanger entre vous pour vous assurer que tout fonctionne correctement, et vous aurez probablement besoin de tests unitaires pour simplifier cette collaboration au point de vue technique. Il est obligatoire d'utiliser git : nous inspecterons l'historique de vos commits lors de l'évaluation.

1 Prise en main

Vous trouverez sur le GitLab de la faculté un template de code ainsi qu'un mini tutorial git. Le code fourni constitue un mini-diepio jouable : vous contrôlez un petit tank, et vous pouvez tirer sur des petits carrés jaunes (utilisez les touches pour vous déplacer, et pointez à la souris pour viser [1]). La bibliothèque SFML 2.5 est nécessaire pour le compiler. Sous Debian ou assimilé, installer le paquet `libsfml-dev` suffit. Le site web de SFML donne des instructions pour les autres systèmes. Sous Mac, utiliser homebrew est peut-être plus simple que d'utiliser XCode.

Exploration du source. Ouvrez maintenant le code source après l'avoir importé dans votre éditeur. On trouve 6 classes, avec à chaque fois un fichier d'entête et un fichier d'implémentation :

Game : C'est la classe principale du jeu. Elle ouvre une fenêtre graphique, crée un monde de jeu, et entre dans une boucle où le monde est animé puis affiché 33 fois par secondes, jusqu'à la fin de partie. La fonction `main()` du programme se trouve en bas de ce fichier.

World : Cette classe s'occupe de la logique du jeu, sans se préoccuper de l'affichage. La méthode principale est `update(GameCmd)`, qui anime le jeu pendant une frame. Son code est finalement assez simple : on anime toutes les entités du jeu, on détecte les collisions puis on retire de la liste les entités qui sont mortes à cette étape. Les autres méthodes de cette classe sont de simples getters.

Entity : Tous les éléments du jeu (carré, tank ou balles) sont des entités. Cette classe compte beaucoup de champs, mais seulement cinq méthodes intéressantes (à part les getters/setters) :
- `update()` : fait bouger l'entité pendant une frame.
- `collides()` : teste si l'entité est en collision avec une autre entité.
- `hitBy()` : réagit au fait que l'entité s'est fait rentré dedans par une autre.
- `makeSquare()` et `makeBullet()` : méthodes statiques permettant de créer un carré jaune ou une balle.

1. S'il est impossible d'utiliser la souris et le clavier en même temps, c'est probablement parce que le touchpad est réglé pour se désactiver quand le clavier est utilisé. Essayez avec une vraie souris externe.

Tank : Une classe dérivée de `Entity` définissant un tank. La méthode `update()` est encore plus compliquée.

View : Cette classe est chargée d'afficher l'état du monde à chaque étape. Elle contient également un catalogue de formes, assez proche des costumes utilisés en Scratch.

GameCmd : Cette petite classe encode l'état du clavier et de la souris. Elle n'a pas de méthode à part les getters. On l'utilise pour passer cet état entre Game et World, afin de mieux séparer le monde de la logique d'affichage.

Au final, ce projet fait déjà presque 700 lignes, dont 150 lignes très répétitives dans la classe `View`. Ce code reste relativement lisible malgré tout, sauf peut-être la classe `Entity`, encore un peu fouillis pour l'instant.

2 Rendu et attendus du projet

Le projet est à faire en binôme. Le rapport et les sources du programme doivent être placés dans un **projet git privé** sur le gitlab de l'istic (ou un autre hébergeur comme gitlab.com ou framagit). Vous ajouterez les deux enseignants du module (`mquinson` et `matlaurent`) à la liste des développeurs de votre projet. Le rapport en pdf doit être dans le git (ou en artefact du CI), et l'intégralité de votre programme doit être écrit en C++.

Code. Vous porterez un soin particulier à l'écriture du code. *Pensez à nettoyer pour ne pas rendre un brouillon.* Le code doit être commenté et bien écrit pour être facilement lisible. L'écriture de tests unitaires vous permettra de refactorer ce qui doit être remis au propre, sans risquer de casser les fonctionnalités. Il est rappelé que l'on écrit un programme pour que d'autres humains puissent le lire, et (accidentellement seulement) pour que les machines puissent l'exécuter. Nous passerons plus de temps à lire votre code qu'à exécuter votre programme.

Rapport. Vous devez apporter une réponse claire et détaillée à chaque question du sujet, sans reprendre trop de code. Pour les questions optionnelles, vous pouvez donner une idée de solution même si vous ne l'avez pas implémenté. Le rapport (pdf de 5 pages maximum) doit contenir une courte introduction générale, une réponse pour chaque question explicitant et justifiant vos choix d'implémentation, une synthèse générale, et une bibliographie précise de toutes les sources (sites, livres ou individus) qui vous ont aidé, avec quelques mots de ce que vous en avez retiré. Si vous avez des propositions d'amélioration pour ce projet, ajoutez-les en annexe.

Extensions. Les questions optionnelles de chaque partie sont ... optionnelles, et il est tout à fait déraisonnable de toutes les implémenter. Vous pouvez implémenter certaines questions optionnelles, mais souvenez vous qu'un projet sans option mais bien architecturé, bien documenté et particulièrement lisible sera mieux évalué qu'un projet moins bien architecturé ou moins bien écrit mais avec plus d'options. La notation ne tiendra absolument pas compte du gameplay ou des animations graphiques. On trouve à la fin du sujet quelques idées et pistes de lecture pour aiguiser votre curiosité si vous souhaitez maintenir ce code après le projet. Il n'est pas demandé d'implémenter ces idées, encore moins dans le cadre du projet. Le faire quand même ne donne pas de bonus.

Triche. La frontière est mince entre *l'oubli* de sources et le plagiat. N'oubliez rien, ne trichez pas. Voler des idées, maquiller des résultats ou mal citer ses sources sont des péchés mortels en recherche. Tricher, c'est reconnaître qu'on va devoir changer de métier, car ceux pris à tricher en science n'ont pas de seconde chance.

En revanche, il est conseillé de discuter du projet et d'échanger des idées avec tous vos collègues. Pour ne pas franchir la ligne jaune, **ne lisez jamais de code écrit par un autre groupe, et ne permettez pas aux autres groupes de lire votre code**. Vous ne pouvez rendre que du code écrit par vous-même, et vous devez détailler brièvement vos sources d'inspiration sur internet dans la partie bibliographie de votre rapport. Soyez spécifique : si par exemple, vous avez trouvé des réponses sur Stack Overflow, pointez les pages utilisées.

IA générative. Ce projet a pour but de vous permettre d'améliorer vos capacités de programmation en C++ et vous simplifier les TP suivants, qui vont accélérer. Prenez l'occasion de vous entraîner sérieusement au lieu de gâcher les ressources mondiales. Recopier du code généré par des perroquets stochastiques au lieu de faire les exercices attendus nuit à votre développement personnel. C'est aussi bête que de porter un exo-squelette pour éviter de transpirer à la salle de sport. Autant changer de cursus.

Collaboration. La science moderne étant un sport d'équipe, nous évaluerons votre capacité à collaborer. Un projet composé de parties individuelles mal assemblées recevra une évaluation sévère, et nous attendons que tous les membres du projet commitent du code dans l'historique du projet. Savoir aider les autres et demander de l'aide efficacement sont des compétences fondamentales pour être scientifique de haut niveau. Quel que soit le contexte, collaborer efficacement est indispensable pour parvenir à produire des logiciels intéressants.

Nous ferons un "checkout" de vos projets git le **vendredi 20 février à 19h** heure de Paris.

Il n'y aura aucune extension. Attention au commit de dernière minute qui casse tout.

3 Meilleure copie de l'original

Les éléments de cette section visent à ajouter des choses qui manquent à notre version par rapport à l'original. Les extensions proposées ne sont pas simples, et il est nécessaire de réfléchir avant de programmer. Souvenez-vous que la clarté de votre code est bien plus importante que le gameplay offert par votre jeu. La première section a pour but de vous faire rentrer dans le code et est difficile à répartir, tandis que les deux suivantes sont plus indépendantes et peuvent être réalisées en parallèle si votre code est bien organisé.

3.1 Prise en main et premières modifications

▷ **Question 1:** Lorsqu'elles sortent de l'écran, les balles reviennent de l'autre côté. Le monde de diep.io n'étant pas torique, modifiez ce qui doit l'être afin que les balles soient détruites lorsqu'elles sortent de l'écran.

▷ **Question 2:** De même, faites en sorte que le joueur ne puisse pas sortir du niveau, mais qu'il soit plutôt stoppé sur les bords.

▷ **Question 3: Base sûre.** Il arrive pour l'instant que l'on meure dès le début si le hasard a fait qu'un carré soit créé à l'emplacement initial du tank. Faites en sorte que le tank commence dans une zone délimitée visuellement en haut à gauche de l'écran, et qu'aucun carré ne puisse être créé dans cette zone.

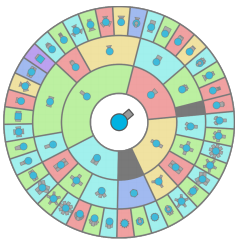
▷ **Question 4: Cooldown.** Le tank envoie actuellement une balle par frame, ce rend le jeu trop facile. Introduisez un délai minimal entre les tirs pour améliorer la jouabilité. La méthode `World::getTick()` permet de compter les frames. Le nombre de ticks à attendre avant d'autoriser le tir suivant devrait être donné par un champ `reload_` de votre tank.

▷ **Question 5: Recul des armes.** C'est un élément de gameplay important du jeu original : à l'extrême (comme avec l'*Annihilator*), on peut tirer derrière soi pour se propulser plus vite. Ajoutez un champ `recoil_` à la classe `Tank` pour déterminer de combien chaque tir doit faire reculer le tank, et modifiez ce qui doit l'être pour que le recul soit effectif.

▷ **Question 6: Score.** Ajoutez un décompte du score, qui sera affiché à l'écran. Casser un carré jaune rapporte 10 points [2].

▷ **Question 7: Accélérations et frottement (optionnel).** Les mouvements actuels du jeu ne sont pas très souples, car il manque la toute petite accélération au début du déplacement et le freinage à l'arrêt. Au lieu de mettre à jour directement la position, il faut modifier un champ `move_`, qui est appliqué automatiquement à chaque tour. Attention à ne pas modifier `speed_`, qui est la vitesse maximale du tank, une caractéristique intrinsèque.

3.2 Plus de tanks (mais proprement)



Le jeu original offre de nombreux tanks, aux caractéristiques diverses. On peut séparer les caractéristiques quantitatives (les balles d'un destroyer sont plus grosses, infligent plus de dégât,s mais rechargent plus lentement) des caractéristiques qualitatives (un smasher ne tire pas de balle du tout).
On pourrait tout implémenter dans la classe `Tank`, mais il faut trouver une organisation (un découpage) pour assurer que cela reste lisible. On peut aussi faire une classe C++ par classe de Tank. Une troisième solution pour le quantitatif est d'agir par *composition* [WP] comme nous avons déjà fait pour la forme (le costume) des entités. Chaque forme est définie une bonne fois pour toutes (dans `View`), et les entités font référence à la forme qui les concerne. À vous de choisir entre ces trois possibilités les différents cas.

▷ **Question 8:** Implémentez deux ou trois tanks différents parmi ceux existants, en vous limitant pour l'instant aux changements quantitatifs. On pourra proposer au joueur de choisir au début, ou bien lorsqu'il a accumulé suffisamment de points d'expérience. Vous expliquerez dans votre rapport comment vous avez organisé votre code pour améliorer sa lisibilité. Votre explication devra être brève mais compréhensible, et argumentée.

▷ **Question 9:** Implémentez un ou deux autres tanks introduisant des changements qualitatifs, et indiquez clairement comment vous avez organisé votre code. Par exemple, le tri-angle tire dans trois directions à la fois tandis que le trappeur tire des projectiles triangulaires qui ne se déplacent pas. Les tanks auto5 et auto3 visent automatiquement les cibles proches ; c'est donc un défi intéressant au découpage du code. Le nécromancien est probablement trop difficile à implémenter puisqu'il convertit les carrés qu'il touche en objets qui suivent son pointeur. Il ne faut donc pas l'implémenter.

▷ **Question 10:** Implémentez d'autres formes de polygones, c'est-à-dire d'objets tournant sur place en attendant d'être détruits par les joueurs, ainsi que des objets gris incassables et rebondissants. Indiquez dans votre rapport comment vous avez organisé votre code.

2. `https://diepio.fandom.com/wiki/Polygons`

3.3 Améliorations graphiques

▷ **Question 11:** Implémentez la possibilité de zoomer la vue, ainsi que de ne voir qu'une partie du monde (centrée autour du tank du joueur). Cette fonctionnalité semble impressionnante, mais c'est en fait très simple avec la notion de vue dans SFML. Voyez le tutoriel officiel afférent, ou celui du wiki de SFML.

▷ **Question 12: Sans souris (optionnelle).** Si vous avez une manette USB (PS4, XBox ou autre), il suffit de la brancher en USB pour qu'elle soit utilisable avec SFML. On ne peut plus utiliser le pointeur de la souris pour savoir vers où pointe le tank du joueur. À la place, on sauvegardera les coordonnées (x, y) d'un pointeur supplémentaire, qui bougeront en réaction aux événements `sf::Event::JoystickMoved` dans la boucle `pollEvent` de la classe `Game`. Si vous n'avez pas de joystick à votre disposition, le second joueur peut utiliser 4 touches du clavier pour déplacer son pointeur de visée.

▷ **Question 13: Multi-joueurs en écran partagé.** Implémentez la possibilité de jouer à plusieurs sur le même ordinateur. Cette fonctionnalité est un peu plus difficile. Il faut bien réfléchir à l'avance sur l'organisation du code, afin que les bons événements SFML soient traités par le bon objet `GameCmd`, lui-même passé au bon tank. Décrivez l'architecture choisie dans votre rapport.

3.4 Gestion des collisions

Dans un jeu comme asteroids dont est inspiré `diep.io`, la détection des collisions pose un défi algorithmique fondamental. La double boucle testant tous les couples d'entités, proposée dans le code fourni, n'est absolument pas suffisante pour gérer des centaines de tanks et des dizaines de milliers de projectiles et polygones.

Pour optimiser ce problème, `diep.io` considère que toutes les entités sont rondes lors des collisions : on peut facilement faire passer son canon au dessus d'un carré ou même toucher la pointe d'un carré, sans prendre de dégâts. Le problème est aussi simplifié par le fait que les polygones ne se déplacent pas, ce qui peut permettre de les trier par leurs coordonnées afin de ne tester la collision qu'avec un sous-ensemble des entités.

Dans le template de code proposé, les objets sont de plus catégorisés : l'équipe du joueur et l'équipe des polygones. On évite ainsi calculer la distance entre une balle et un joueur de la même équipe pour ensuite réaliser qu'il ne se passe rien en cas de contact.

▷ **Question 14: Découpage spatial en grille.** Il s'agit ici de découper le terrain de jeu en cases relativement grandes, et de ne tester la collision d'une entité donnée qu'avec les entités étant sur la même case qu'elle ou sur les cases immédiatement à côté.

On ajoutera donc une structure de données au monde, composée d'une grille de `std::list<Entity*>` et permettant de retrouver toutes les entités d'une case donnée.

Trouver la bonne taille de cases pose un autre défi. Si elle est trop grande, chaque case contiendra trop d'entités et la détection restera trop coûteuse. Si certaines entités sont plus grandes que les cases, il ne suffit plus d'explorer la case de l'entité considérée puis les huit cases voisines. Des cases trop petites peuvent demander trop d'opération de type `std::list::splice()` pour passer les entités en mouvement d'une liste dans une autre. Au final, déterminer la bonne taille de case nécessitera un peu de tâtonnement. Une fois la bonne taille trouvée, n'oubliez pas d'expliquer pourquoi et comment dans votre rapport.

▷ **Question 15: (optionnelle)** Étudiez les performances de votre algorithme de détection des collisions. On souhaite comprendre l'impact du nombre de polygones, de balles et de tanks dans la partie sur les performances. On pourra proposer une étude à la fois théorique et basée sur des mesures de performance pratique. Vous pourriez aussi implémenter un tank "rouleau compresseur" tirant un tapis de balles lentes pour stresser votre algorithme.

3.5 Robots et bêtise mécanique

Au final, ce qui manque le plus à notre version par rapport au jeu original, ce sont les autres joueurs. Nous allons maintenant développer quelques IA vraiment rudimentaires afin de pouvoir jouer contre l'ordinateur.

▷ **Question 16: Algorithmes naïfs.** Implémentez les trois "algorithmes" suivants : tir continu en tournant sur place (comme quand on appuie sur C et E dans le vrai jeu) ; déplacements aléatoires ; trajectoire fixe en rond ou en spirale.

▷ **Question 17: Algorithmes simples (optionnelle).** Implémentez les algorithmes suivants : visée automatique, évitement des objets proches, exploration.

▷ **Question 18: Championnat de robots (optionnelle).** Comparez les performances de plusieurs algorithmes en organisant une compétition entre eux.

4 Procrastination hors sujet

Cette section regroupe quelques idées de procrastination autour du sujet, mais elle n'est donnée que pour aiguiser votre curiosité et vous proposer quelques lectures supplémentaires. Vous n'avez pas le temps d'aborder ces sujets dans le cadre de ce projet scolaire, et c'est très bien ainsi. Faire des extensions n'apportera pas de point supplémentaire puisque c'est la propreté du code écrit qui compte, pas la quantité.

4.1 Algorithmique des collisions (procrastination inutile)

Au lieu de pré-déterminer la taille des cases, on peut utiliser une structure de données groupant dynamiquement les entités par proximité spatiale, par exemple avec un QuadTree. Il existe de nombreux tutoriels sur la mise en place des quadtree dans les jeux pour optimiser la détection de collision. On trouve également de nombreuses implémentations sur Internet dont vous pourriez vous inspirer. N'hésitez pas à citer vos sources en commentaire de votre code.

De nombreuses idées sont possibles pour pousser la limite du nombre d'entités que notre jeu autorise. On pourrait utiliser une approche mixte, plaçant les polygones dans un quadtree tandis que les autres entités sont dans une grille, ou bien le contraire. On pourrait aussi éviter de recalculer le quadtree à chaque frame, comme semble faire le vrai jeu. Enfin, on pourrait expérimenter avec d'autres structures de données pour le découpage spatial, et même évaluer les différentes solutions et optimisations pour tester le nombre maximal d'entités que l'on parvient à animer à 33 frames par seconde. Mais bon.

4.2 Jeu en réseau (procrastination inutile)

Plutôt que de jouer contre un autre joueur humain sur la même machine, on peut vouloir jouer contre un ou plusieurs adversaires externe via le réseau. D'un point de vue génie logicielle, cette option n'est pas très différente de ce que vous avez dû mettre en place pour permettre à de petites IA de jouer en local.

SFML propose une interface pour le réseau, mais elle reste relativement inintéressante par rapport à l'interface BSD de base vue dans le cours de réseau au premier semestre. En général, il n'est pas trivial de réaliser un tel jeu en temps réel, et il faut optimiser convenablement le protocole applicatif (même à seulement 33 frames par seconde). Déporter de l'affichage dans un processus séparé peut également permettre l'écriture de tests automatisés de votre code, ce qui est toujours une bonne chose.

Une bonne première étape pourrait être de permettre la connexion de spectateurs : les clients n'envoient pas de commandes, mais ils reçoivent le nouvel état du monde à chaque frame. Le plus simple est de recréer un nouveau monde à chaque frame coté client, en utilisant un protocole texte où l'on envoie les valeurs en ASCII. Pour cela, il faut implémenter des opérateurs de sérialisation et désérialisation `<<` et `>>` dans toutes les classes envoyées sur le réseau (au moins `GameCmd`, `World` et `Entity`). C'est aussi simple qu'inefficace, mais ça marche (pour de petits mondes). Une fois les spectateurs implémentés, il est plus facile d'étendre à des tanks controllés par des joueurs en rajoutant des messages du client vers le serveur.

Il est ensuite possible d'optimiser ce protocole pour n'envoyer que les éléments visibles par le tank destinataire ou encore pour binariser les protocoles (avec `memcpy`) plutôt que d'utiliser tout l'ASCII. Une bibliothèque comme `FlatBuffer` peut s'avérer utile pour cela.

Pour aller encore plus loin, on peut s'amuser selon deux axes : rendre le protocole tolérant aux défauts du réseau (pertes, gigue) et réduire à l'extrême le volume de données échangées.

Le premier défi demanderait de faire notre propre protocole tolérant aux fautes au-dessus d'UDP. En effet, si les paquets sont retardés, on n'a plus besoin d'envoyer toutes les frames intermédiaires : on peut directement sauter à la frame courante. Suivre cette idée mènerait à une sorte de TCP sans réémission.

Une solution classique au second défi est de n'échanger que les commandes, et laisser chaque ordinateur recalculer les mises à jour du monde. Cela impose soit des contraintes de synchronisation très forte (personne ne passe à la frame suivante avant d'être sûr d'avoir toutes les `GameCmd`), soit d'avoir la possibilité de revenir dans le passé quand on reçoit un `GameCmd` après coup. Dans tous les cas, c'est très difficile. Les curieux consulteront ce cours de *Parallel Discrete-Event Simulation*, qui est un sujet de recherche actif depuis un demi-siècle. Une autre approche serait d'utiliser un canal causalement consistant [WP], mais ce n'est pas plus simple.

Une autre idée folle serait de vouloir jouer au vrai jeu avec notre client graphique. Cela demande de percer les secrets du protocole utilisé entre le navigateur web et le serveur `diep.io`. Ce retro-engineering est volontairement rendu difficile par l'auteur pour éviter la création de scripts permettant de tricher. Cela afin de ne pas gâcher le plaisir des vrais joueurs (qui rémunèrent l'auteur par le biais des publicités). Avec une première description du protocole, vous pourriez malgré tout y parvenir...

4.3 Intelligence artificielle et apprentissage (procrastination inutile)

Plutôt que de compliquer les algorithmes écrits spécifiquement pour ce jeu, il pourrait être être amusant d'implémenter une AI apprenant toute seule à jouer à ce jeu. Pour cela, l'approche NEAT présentée dans cet article pourrait être une alternative plus simple que le deep learning. NEAT a déjà été utilisé avec succès dans différents jeux ici, là, là ou encore là.

Cette profusion montre que la méthode n'est pas affreusement compliquée à mettre en œuvre. Dans le premier cas ci-dessus (MarI/O) est assez pédagogique, et le code est même disponible. La moitié des 1200 lignes de lua gère les interactions avec l'émulateur SNES tandis que l'algorithme d'évolution génétique de réseaux de neurones occupe l'autre moitié du code. On trouve de nombreuses implémentations de cet algorithme en ligne, mais je ne sais pas si elles sont vraiment utilisables. Je pense malgré tout que NEAT peut s'appliquer à DeepDiep (même si je ne l'ai pas encore testé moi-même), car tous les ingrédients sont présents. Un quadrillage de la zone autour du robot fournira les entrées comme dans MarI/O, tandis que les sorties sont très simples : une `GameCmd` ne demande que neuf booléens : les quatre touches de déplacement (haut/bas/gauche/droite), la touche pour tirer, et les quatre touches de déplacement du pointeur de visée.

Cette idée ressemble à une procrastination particulièrement chronophage, mais l'imaginer suffit à m'amuser.