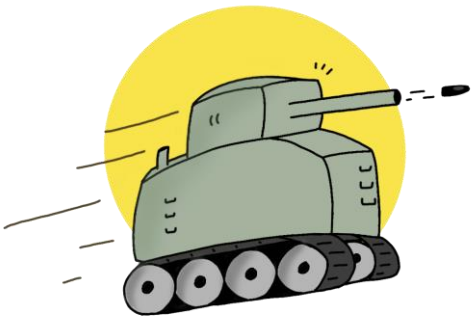
Projet De Programmation

Tiny Tanks



Janvier 2022 - Mai 2022

Rapport du projet

Sommaire

Qu’est-ce que Tiny Tanks ? 2
Version originale 2
Notre version 2

Comment s’y est-on pris ? 3
Spécificités techniques : 3
Répartition du travail : 3

Comment avons-nous codé ? 4
Zone de jeu 4
Menu 4
Plateau 4
Bonus 5
Niveaux personnaliser 5
Succès 5
La sauvegarde des fichiers 6
Tanks 6
Joueur (contrôle du tank) 7
L’IA 7
Missiles 8

Raids aériens 8
Crédits 9

A* 10

Détection de collisions entre un missile et un obstacle 13

Images 15

Rapport Hebdomadaire 17

Qu’est-ce que Tiny Tanks ?

Version originale

Il s’agit d’un jeu solo où le joueur incarne un tank en contrôlant ses déplacements et de la visée de son canon. L’objectif, éliminer le ou les tanks adverses afin de remporter la bataille. Le jeu est sous la forme d’une campagne se divisant en une vingtaine de niveaux, chacun avec une carte unique et une difficulté croissante. À cela s'ajoutent des combats contre des “boss” tous les 6 niveaux, des ennemis plus redoutables avec des capacités accrues.

Lien du jeu : https://www.crazygames.fr/jeu/tiny-tanks-b

Notre version

Notre objectif était de reproduire le jeu original spécifié ci-dessus, tout en lui apportant de nouvelles fonctionnalités :

- Un éditeur de niveau
- Apparition de bonus et de malus
- Une difficulté en plus avec les raids aériens
- Des succès
- Le support de la manette
- Une personnalisation des paramètres du jeu
- Système de sauvegarde
- Changement de langue
- Portabilité (.jar et .exe)

Comment s'y est-on pris ?

Spécificités techniques :

Nous avons décidé d'utiliser comme moteur de production "Gradle", sous couche Kotlin, avec Java 11 pour des raisons de stabilité et en se basant sur nos connaissances déjà acquises depuis le début de notre cursus informatique. Cela nous a permis d'utiliser des librairies telles que :

- Jackson Yaml qui nous permet d'analyser facilement des fichiers yaml qui sont utilisés pour stocker les informations des niveaux et aussi de les créer facilement
- log4j qui nous permet d'avoir des logs pour déboguer le jeu , mais aussi en cas de problème pour remonter facilement les problèmes des utilisateurs
- Spring Core qui permet de récupérer la liste des fichiers dans le dossier ressources pour l'utiliser après la compilation en .jar
- JamePad qui permet de récupérer des contrôleurs externes tel que des manettes et leurs mouvements
- sdl2gdx qui est une dépendance pour utiliser JamePad
- commons-io qui permet d'effacer plus rapidement des fichiers

Répartition du travail :

Nous avons créé un dépôt Gitlab pour héberger le projet afin de suivre une organisation de travail comme vu au semestre précédent.
Nous faisions également deux réunions hebdomadaires organisées autour des soutenances avec le professeur, une avant faisant office de rétrospective et une après servant à la planification des tâches pour la semaine suivante.

Notre équipe s'est partagé les tâches de la façon suivante :

Tom **RENÉ**	Tank (déplacement, modèle, affichage, responsive …..).Modèle du boss.Partie calcul du raid aérien.
Illia **GARGAUN**	Interface graphique Swing, Contrôleur pour chaque interface, Génération de bonus, Support de la manette, Les succès, Le système de musique, Sauvegarder et la sécurisation des fichiers. Éditeur de niveaux, Système de langue.
Nicolas **THEAU**	Partie technique du plateau (création, génération, sauvegarde, modifications, vérification de noms et détection). L'IA (Boss + bot). Direction des réunions
Isabelle **LIN**	Système de Missiles, Système de musique, Les crédits.
Yanis **ROZIER**	Affichage de la zone de jeu et son responsive, affichage du plateau, design du jeu, création du logo, affichage du raid aérien.

Comment avons-nous codé ?

Zone de jeu

Il s'agit de l'affichage réunissant toutes les fonctionnalités nécessaires à la jouabilité du jeu. Dirigé avec GameView.java, nous avons un panel qui réunit, de part un JLayeredPane, le plateau, les tanks (joueurs et robots), les missiles, les bonus et malus ainsi que des informations complémentaires pour le joueur tel que le nom du niveau, son nombre de vies restantes dans sa campagnes et le nombre de tanks ennemis au début du niveau. A noter que dans le cas d'un niveau comportant un boss, le thème de la zone de jeu change (affichage et son) et une jauge de santé correspondant à celle du grand ennemi s'affiche en haut du plateau.

Voir illustration ci-dessous.

Menu

La plupart des menus ont été créés avec WindowBuilder qui est un plugin pour Eclipse qui permet une création facile et rapide d'interface graphique sous Swing. Les menus sont des JPanel, chaque menu a son propre contrôleur qui contient le code d'interaction de l'utilisateur.

Le contrôleur est une classe qui contient plusieurs sous classe qui étend quelques événements de Swing, comme appuyer sur un bouton, une touche, etc.

Plateau

Le plateau de chaque map est stocké dans un objet PlateauModel. Nous avons décidé que les plateaux soit en fait un fichier en .yaml qui contient toutes les informations du jeu. Si vous générez une map depuis l'éditeur vous aurez toutes les instructions pour modifier la map depuis le fichier directement mais il y a certaine condition a respecter:

- la partie bg et fg doivent être entouré de # de largeur 25 et de hauteur 18.
- Dans bg toutes les cases à l'intérieur doivent être H(Herbe) ou S(Sable) qui va correspondre à la couleur du background.
- Dans fg les cases peuvent avoir la valeur :
  - " " pour mettre du vide
  - "M" : Mur
  - "N" : Mur cassable
  - "B" : Buisson Vert
  - "C" : Buisson Orange
  - "T" : Trou
  - "J" : Joueur
  - "R" : Bot
  - "Z" : Boss
- Dans fg il doit y avoir 1 seul joueur, au moins un bot (un boss est compté comme un bot) et au maximum 1 boss.
- Un champ raid qui est true ou false en fonction de si les raid sont activé sur la map.

Nous stockerons donc dans PlateauModel dans un tableau de tableau de case les background et le foreground ainsi que la position des bots et du joueur et un boolean raid. Au moment de l'accès à la liste des niveaux, toutes les maps dans le dossier maps sont lus mais seules celles qui remplissent les critères ci-dessus sont affichées.

Bonus

Le jeu contient 5 bonus, dont 4 positifs et 1 négatif. Chaque bonus a sa propre probabilité d'apparaître sur le plateau. Jusqu'à 2 bonus peuvent apparaître toutes les 20 secondes, l'apparition des bonus est avertie par un point d'exclamation qui clignote, une fois apparu sur le plateau, le joueur ou le bot peut le prendre en roulant sur le bonus, l'effet est appliqué directement. Les bonus ont une vie de 10 secondes, après ce temps, ils disparaissent du plateau, la disparition du bonus est avertie par son clignotement.

Liste de bonus :

- Speed : augmente la vitesse du tank pendant 1.5 seconde.
- Slow : diminue la vitesse du tank durant 1.5 seconde.
- Life: rajoute une vie supplémentaire.
- Shield: fait apparaître un bouclier en tour du tank durant 1.5 seconde.
- Invisibilité : rend invisible le tank durant 1 seconde.

Niveaux personnaliser

Chaque utilisateur peut créer/modifier et partager ces niveaux, personnaliser. La création se fait dans l'éditeur spécial où on peut "dessiner" ses niveaux en personnalisant le fond et le terrain de jeu. *(voir image : Éditeur)* L'utilisateur peut aussi créer/modifier une map à la main en modifiant le fichier "map.yaml" du niveau souhaité.
L'utilisateur devra choisir un nom de map qui sera vérifié pour savoir si il est valide ou bien disponible, dans le cas inverse un autre nom sera proposé.

Les niveaux sont sauvegardés dans un dossier qui porte le nom du niveau, le dossier contient un fichier "map.yaml" qui contient la map du niveau et une image "logo.png" qui est un aperçu du niveau. L'aperçu est généré à chaque changement de la map. L'aperçu peut être forcé, pour cela il suffit de renommer une image en "logo-ignore.png".

Succès

Le jeu comporte 12 succès, certaine demande de passer beaucoup de temps en jeu et d'autre de survivre à des événements spéciaux qui apparaisse sur différents niveaux ou bien qui demande une tactique spéciale.

- Nombre de tanks détruits : il faut détruire 50 tanks
- Nombre de morts : il faut mourir 50 fois.
- Temps passé en jeu : le joueur doit passer 100 minutes dans des niveaux
- Finir la campagne sans mourir : est un des succès les plus durs à réaliser
- Niveaux de campagne atteint: il faut gagner 6 niveaux
- Regardes les crédits en entier : fait partie des succès les plus faciles
- Créé une map custom : fait partie aussi des succès les plus faciles

- Survivre à un raid aérien: dans certains niveaux le joueur a une chance ou bien le malheur d'avoir un avion qui le jeté des bombes sur le joueur
- Bot qui tue un autre bot : est un niveau où il faut avoir une tactique spéciale
- Gagner une partie sans tirer : est un niveau très dur, en effet il faut calculer la direction des missiles.
- Finir le jeu : est un succès qui a une première vue parait très facile , mais en réalité est très compliqué.

La sauvegarde des fichiers

Les fichiers du jeu se trouvent dans le dossier ".tinyTank" de l'utilisateur de l'ordinateur, ce qui permet au jeu d'être lancé depuis n'importe quel endroit de l'ordinateur sans perdre son avancement dans le jeu.

Les fichiers qui sont sauvegardés sur la machine de l'utilisateur sont divisés en deux parties : fichier de système qui contient les fichiers de paramètres, succès et la progression de la campagne et fichier de jeu qui sont les niveaux et l'image qui représente les niveaux. Les fichiers de jeu sont sauvegardés sous le format yaml et png. Les fichiers système sont des classes qui héritent de la classe *"Serializable"* ce qui facilite l'enregistrement sur l'ordinateur.

Les paramètres sont enregistrés sans sécurité, ce qui permet aux utilisateurs de s'échanger ce fichier facilement.

La campagne et les succès sont sauvegardés sur l'ordinateur d'une manière sécurisé, c'est-à-dire le fichier est chiffré donc on ne peut pas le modifier à la main en plus de cela pendant l'ouverture du fichier , on vérifie l'UUID unique du pc avec celui qui se trouve dans les fichiers, si c'est le même UUID l'utilisateur peut continuer à jouer, mais si ce n'est pas le même UUID la campagne et les succès sont remis à zéro.

Tanks

Pour les tanks, nous avons décidé de diviser le code en une classe mère « TankModel » abstraite et plusieurs classes filles correspondant aux différents modèles des différents bots et de celui du joueur.

Cette classe contient ainsi les attributs classiques et communs à tous les tanks, comme la taille de leurs composants (canon et corps), la valeur du vecteur de vitesse, un controller (qui permet de décider les actions transmises au modèle), le nombre de munitions et la gestion des bonus entre autres.

Nous avons décidé de créer une classe « ObjetManipulable » qui serait commune à tous les objets pouvant se déplacer et où nous devrions gérer leurs collisions. Pour l'instant, ça se résume aux tanks et aux missiles.

Cette classe stocke les coordonnées du centre de l'objet, la valeur de longueur et de largeur de l'image qui lui est associée et une classe « Shape » qui correspond à sa zone de collision actuelle. C'est elle qui est mise à jour lors de chaque déplacement.

Au niveau des méthodes, la classe abstraite TankModel contient toutes les fonctionnalités basiques des tanks qui sont :

- L'application des déplacements et par conséquent la rotation des tanks.
- La fonction de tir
- Et la mise à jour des bonus

La fonction d'orientation des canons des bots (et du boss particulièrement) étant trop différentes de celles du joueur, nous n'avons pas pu la regrouper ici.

La classe ObjetManipulable, elle, contient donc les fonctions de déplacement des coordonnées de l'objet, donc avec interaction avec les éléments du plateau (buissons, bonus et collisions).

Les classes TankModel/ObjetManipulable n'étaient pas spécialement difficiles à coder, mais il fallait être rigoureux et penser à tous les cas possibles (surtout pour la rotation du corps du tank et de la détection de cases pour les interactions avec les éléments du plateau).

Un point négatif que j'aurais aimé me rendre compte avant la conception est que le fonctionnement des classes Missiles et Tank étaient trop différentes pour généraliser les déplacements dans une seule même fonction. Donc actuellement les fonctions move() d'ObjetManipulable sont utilisées que par les tanks.

Joueur (contrôle du tank)

On a décidé de faire une classe fille de TankModel qui elle s'occuperait de récupérer les entrées du controller pour les transformer en valeur utilisable par les fonctions génériques.

La classe fille Joueur, elle possède, en plus des attributs de la classe mère, des attributs de sauvegarde de position. On y sauvegarde la position du curseur sur l'écran, de la position de l'image du tank. On retient aussi les entrées de la manette lorsqu'elle est active.

En termes de fonction, la classe possède seulement des fonctions qui agissent à partir des entrées récupérées par le controller. Ces fonctions sont celles qui donnent une valeur au vecteur vitesse et à l'orientation du canon.

L'IA

Il y a 2 IA dans notre jeu :

- Le boss qui a une IA plutôt simple, mais qui est très puissante, notamment grâce à ses points de vie et le nombre de missiles qu'il tire. Le boss va se déplacer de manière aléatoire dans la map et va tirer à l'aide de ses 4 canons à intervalle régulier, ses balles ne peuvent pas le toucher.
- Les Bots qui elles ont une IA un peux plus pousser notamment grâce à de la détection d'obstacle qui lui permet de savoir s'il y a un obstacle entre lui et le joueur pour éviter que la balle rebondit sur lui-même. Il est aussi très agressif grâce à l'implémentation de l'algorithme A* qui lui permet de trouver la route la plus courte. entre lui et le joueur tout en évitant les obstacles. Voici en pseudo code son comportement :
    - Si le bot vois le joueur : Aller dans une direction aléatoire accessible

- Si le bot vois le joueur dans les cases autour accessible :
  Aller dans une de ces directions
- Sinon :
  Se diriger vers le joueur

Plus bas une explication de l'algorithme A* avec un schéma.

Missiles

La classe MissileModel contient les attributs communs à tous les missiles comme la vitesse de déplacement, le joueur ayant tiré le missile, le temps avant son autodestruction, etc… .

En ce qui concerne les méthodes, la classe possède deux méthodes, la première permet de mettre à jour les coordonnées du missile et la seconde vérifie les collisions entre le missile et les obstacles du plateau. Si il y a effectivement une collision alors sa direction est inversée. Le principe de détection de collisions est en grande partie basé sur le cercle trigonométrique. (cf. *Détection de collisions entre un missile et un obstacle* pour plus de détails)

Quant à la détection de collisions entre un missile et un tank, elle repose sur la mise en place de « hitbox » correspondant aux zones sensibles du missile et du tank. Si il y'a au moins un point d'intersection entre les deux « hitbox » alors le missile est en contact avec le tank.

Réussir à bien synchroniser la vue du missile et celle du plateau a été plutôt compliqué mais le plus difficile reste quand même la gestion des collisions. Il a fallu trouver un moyen pour gérer toutes directions, angles que peut avoir un missile et prendre en compte l'intersection entre deux obstacles.

Raids aériens

Nous avons également décidé d'implémenter une difficulté supplémentaire notamment pour les niveaux allant de 5 à 9, des raids aériens. Le joueur devra faire face à un ennemi supplémentaire venu du ciel, un avion qui larguera une traînée de bombes, plus précisément cinq bombes, sur le joueur afin de l'éliminer et cela à des moments aléatoires.

Le raid aérien fonctionne de la façon suivante :
Périodiquement, notre algorithme tire un nombre aléatoire entre 0 et 10000. L'opération se réitère jusqu'à ce que le chiffre 0 ressorte. A ce moment-là, le raid se déclenche.

L'algorithme choisit les coordonnées d'une case aléatoire sur le plateau, à l'exception de celle se trouvant dans un rayon de 10 cases du joueur (la case doit également être dénuée d'obstacle). Il s'agira des coordonnées sur lesquelles sera larguée la première bombe du raid aérien.

Quant aux suivantes, elles seront larguées suivant une trajectoire allant en direction du joueur et réparties sur cette même trajectoire par un vecteur vitesse plus ou moins élevé. Juste avant le largage des bombes, l'ombre d'un avion passe au-dessus du plateau annonçant l'attaque aérienne et la trajectoire que prendra cette dernière.

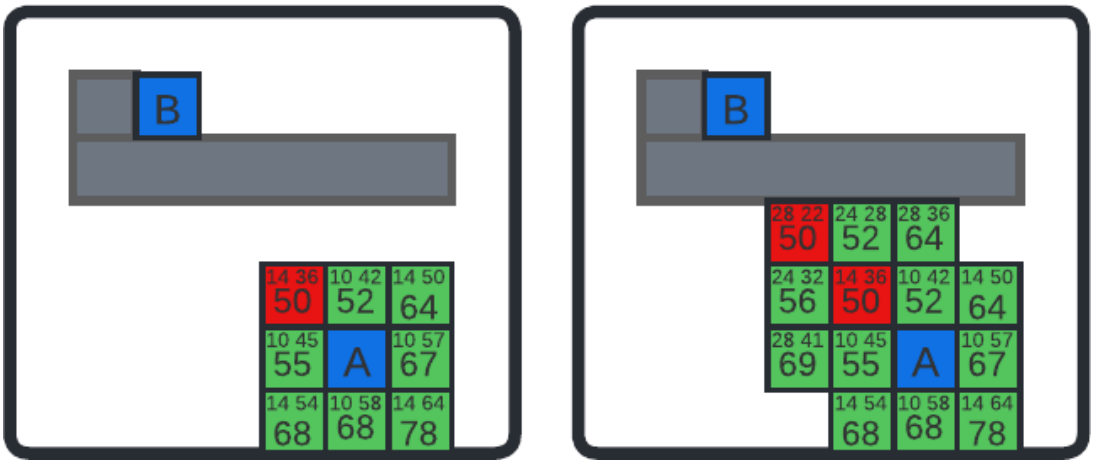
Le joueur voit alors les bombes exploser sur le terrain après avoir aperçu leurs ombres se rapprocher progressivement du sol, en espérant pour lui qu'il a eu le réflexe de les esquiver à temps.

Crédits

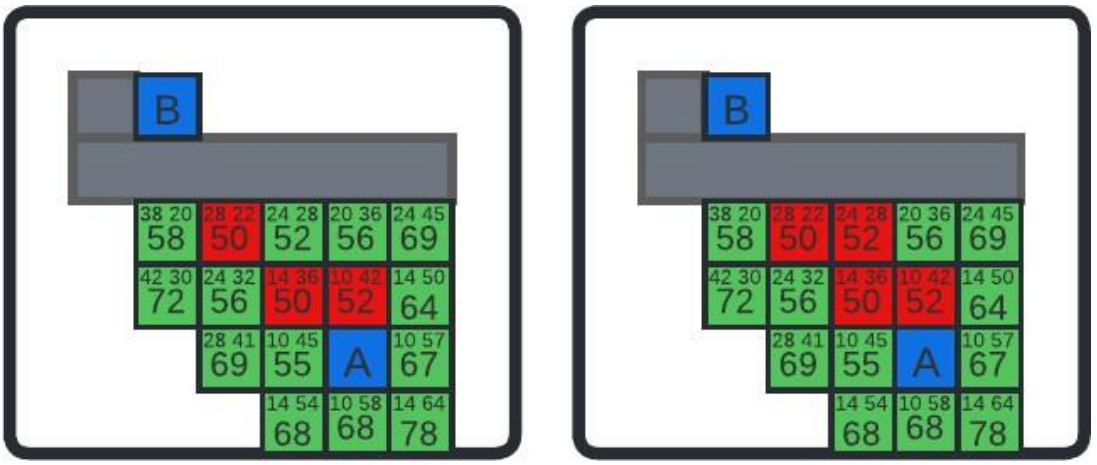
La classe contenant les crédits étend JPanel. Elle est composée d'un JTextePane qui contient le texte et d'un JPanel qui sert de fond sous le texte. Le défilement du texte est dû au déplacement du JTextePane vers le haut.

La seule difficulté a été de trouver comment faire défiler du texte.

A*

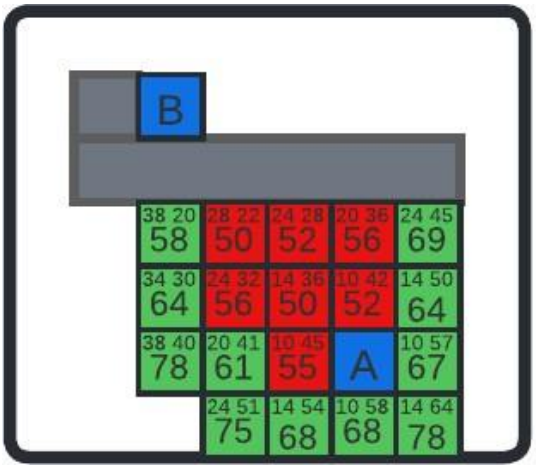


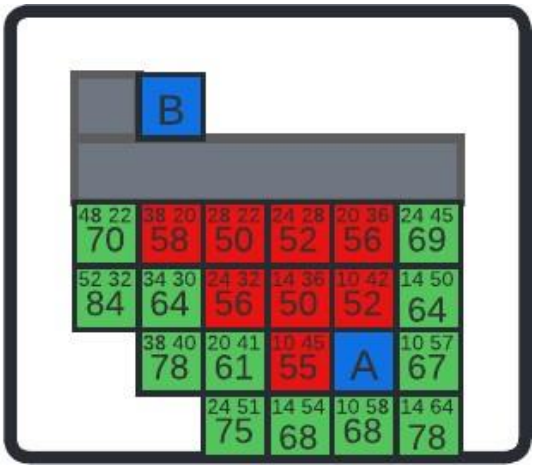
La case A représente la position d'un bot et la case B le joueur, le bot cherche le chemin le plus court sans calculer la distance avec des cases qui éloignerait le bot du joueur.
Nous générons des nœuds autour du bot qui représente une case, sa distance avec le bot,en haut à gauche, puis sa distance euclidienne (à vole d'oiseau), en haut à droite, enfin le nombre du milieu est la somme des deux. Après un premier passage nous pouvons voir que la case 50 en haut à gauche de A semble être la plus proche du joueur, nous allons donc continuer pour voir si nous trouvons un chemin en calculant la distance des cases autour,

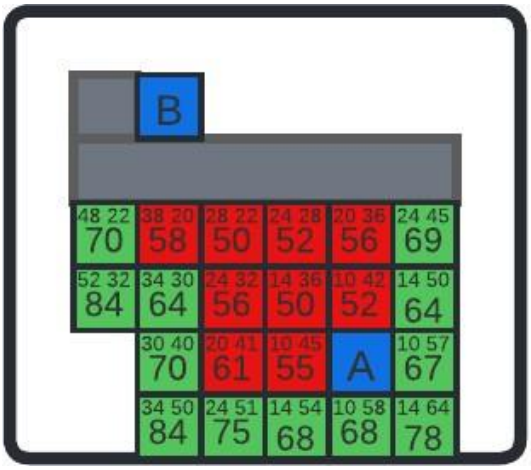


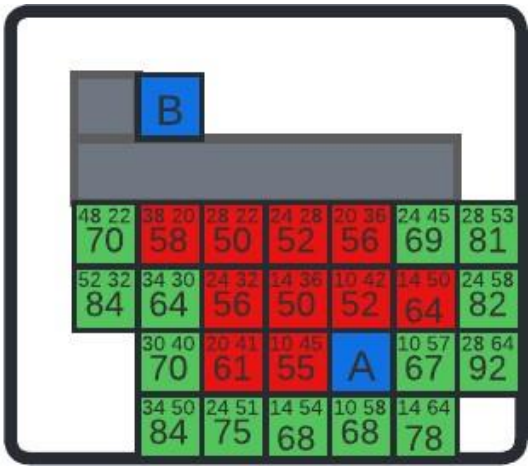
Sur la 3eme image nous pouvons remarquer que nous somme bloqué et que le chemin que l'on commence à prendre pouvait potentiellement ne pas être le plus optimisé, il y a 2 cases qui sont à 52 de distance comment savoir par laquelle continuer ? Il suffit juste de prendre la première qui a été découverte, lorsque l'on rencontre des cases qui ont déjà été évalué on regarde si il n'est pas possible d'obtenir une distance entre le bot et la case plus courte, c'est le cas pour la 3eme image ou la case 64 (celle la plus haute) devient 56.

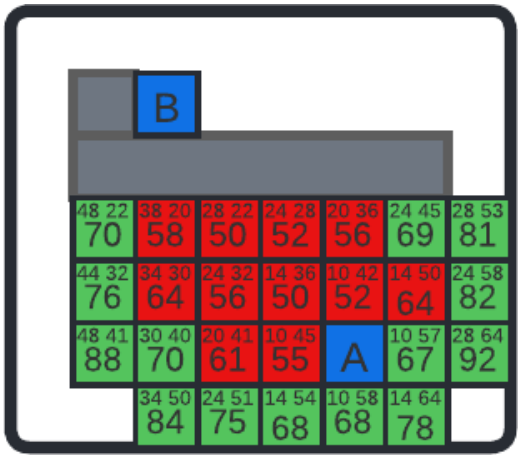
Nous pouvons continuer de dérouler l'algorithme.

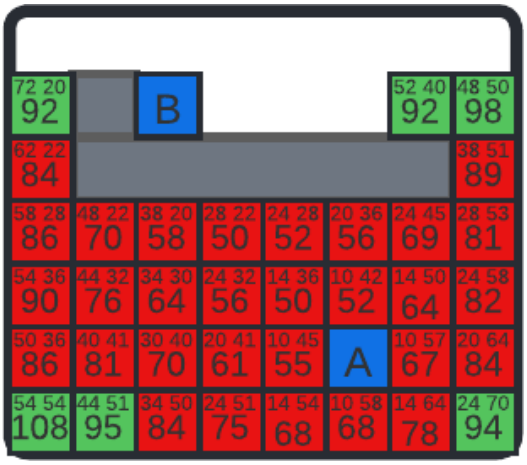


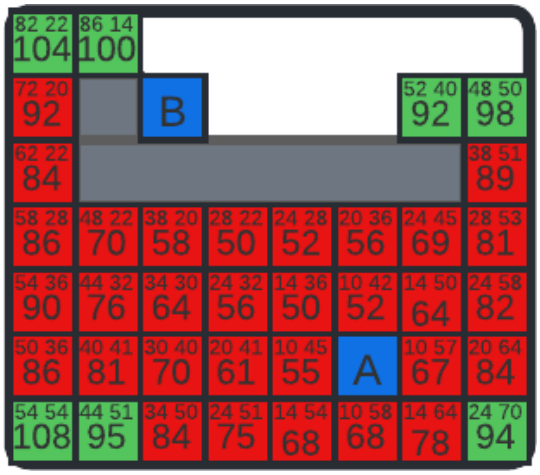


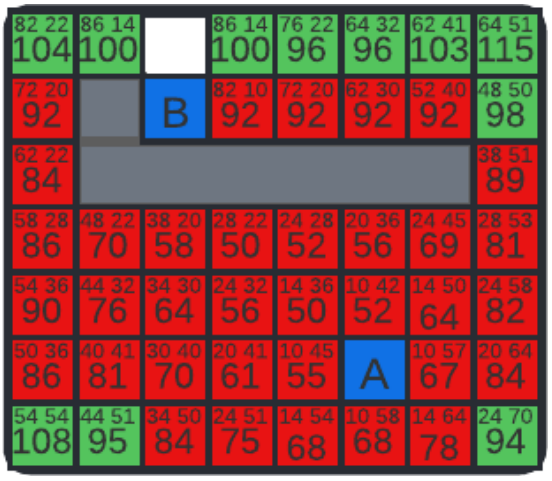


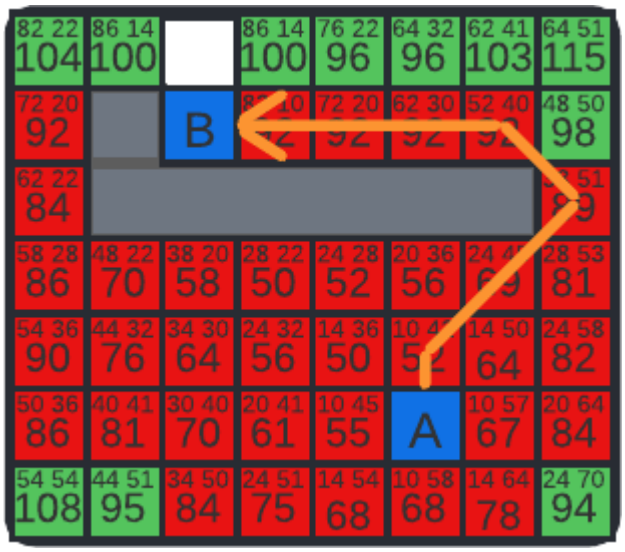










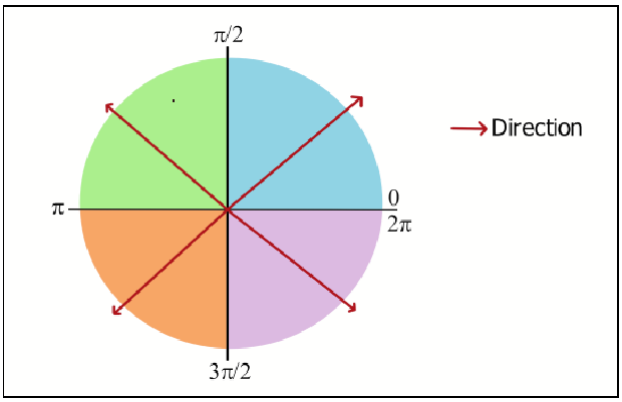


Lorsque que nous arrivons il suffit de remonter tout les pères de chaque noeud pour trouver le chemin le plus rapide

La taille du plateau a été réduit pour rendre le schéma plus lisible mais sur un plateau plus grand il n'y aurait pas eu beaucoup plus de de calcul supplémentaire pour trouver le chemin

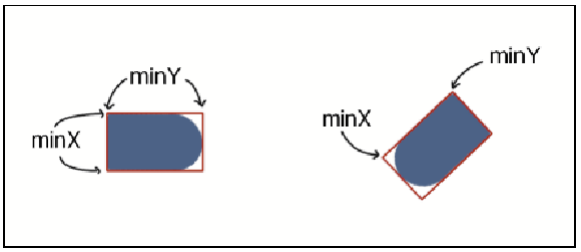
autre représentation de A * : https://github.com/LogicJake/A-star-search

Détection de collisions entre un missile et un obstacle



Avant de calculer les collisions, il faut d'abord connaître l'orientation du missile.
Pour cela, nous nous appuyons sur le cercle trigonométrique.

Sur le schéma ci-contre, les flèches rouges indiquent dans quelle direction se déplace le missile.

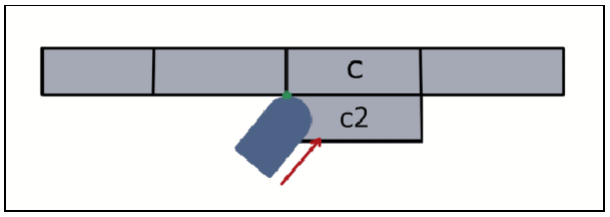


Le schéma ci-contre indique à quoi correspondent les points minX et minY pour un objet de type Shape.
Le rectangle rouge correspond à la « hitbox » du missile.

Pour chaque direction, il n'y a que deux possibilités, soit l'obstacle se situe au-dessus ou en-dessous du missile, soit il se situe à gauche ou à droite du missile.

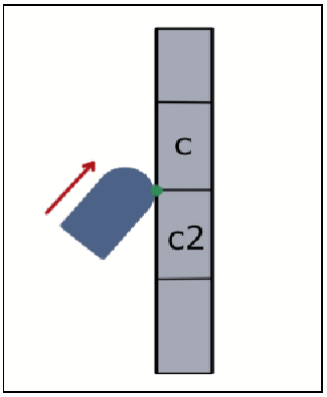
Considérons un missile se dirigeant vers le haut à droite.
c correspond à la case avec laquelle le missile est en collision et **c2** permet de gérer la collision à l'intersection de deux cases.

Cas 1 : l'obstacle se situe au-dessus du missile.



Lors de la collision avec un obstacle se situant au-dessus, le premier point appartenant à la « hitbox » du missile qui sera en intersection avec un point de l'obstacle sera le point ayant l'ordonnée la plus petite, soit minY.

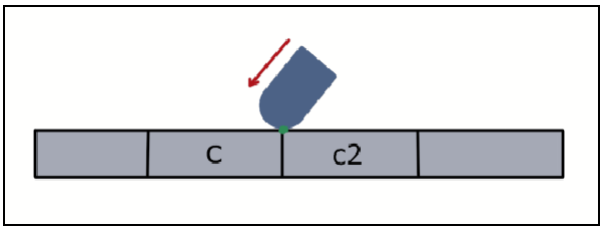
Cas 2 : l'obstacle se situe à droite du missile.



Dans le cas où l'obstacle se trouve à droite, le premier point appartenant à la « hitbox » du missile qui sera en intersection avec un point de l'obstacle sera le point ayant l'abscisse la plus grande, soit maxX.

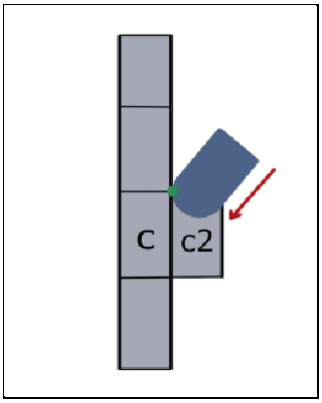
Considérons maintenant un missile se dirigeant vers le bas à gauche.

Cas 1 : l'obstacle se trouve en dessous du missile.



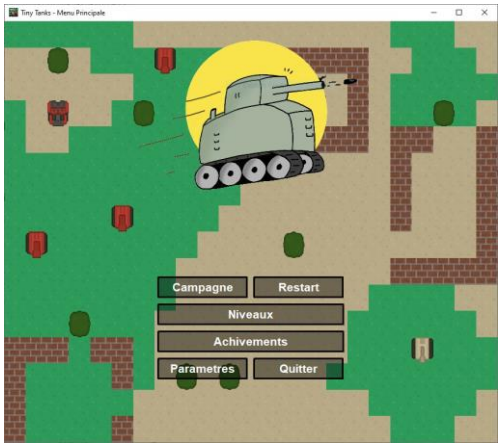
Si l'obstacle se trouve en-dessous, le premier point appartenant à la « hitbox » du missile qui sera en intersection avec un point de l'obstacle sera le point ayant l'ordonnée la plus grande, soit maxY.

Cas 2 : l'obstacle se trouve à gauche du missile.

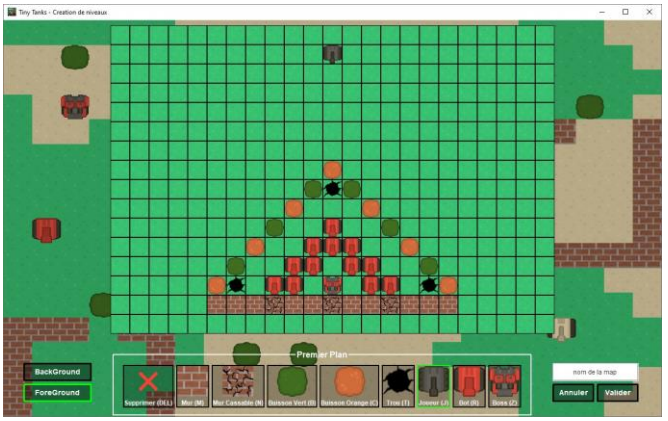


Lorsque l'obstacle se situe à gauche, le premier point appartenant à la « hitbox » du missile qui sera en intersection avec un point de l'obstacle sera le point ayant l'abscisse la plus petite, soit minX.

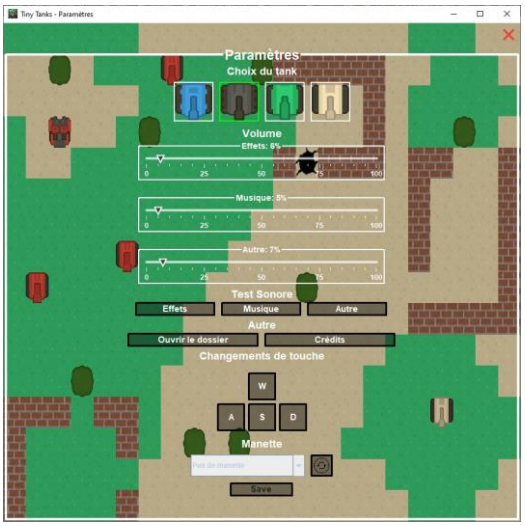
Images



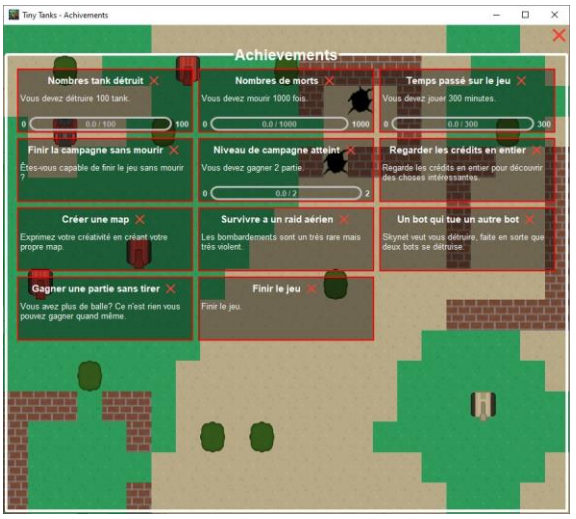
(Menu Principale)



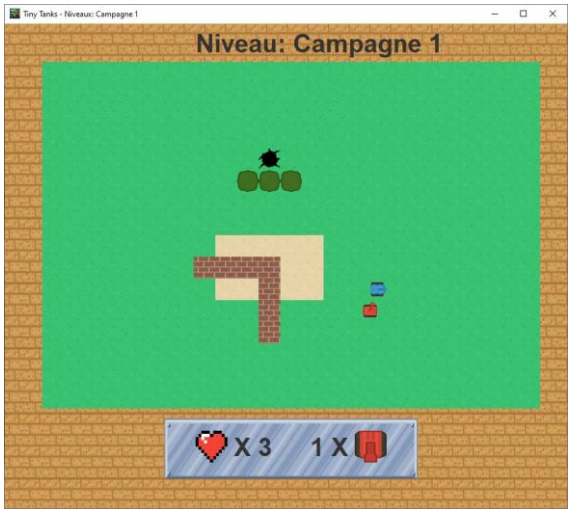
(Éditeur)



(Paramètres)



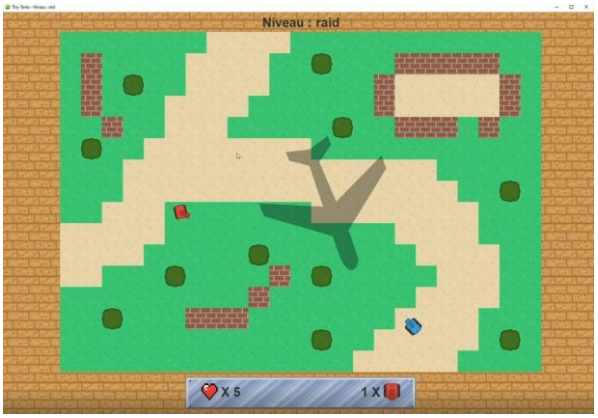
(Succès)



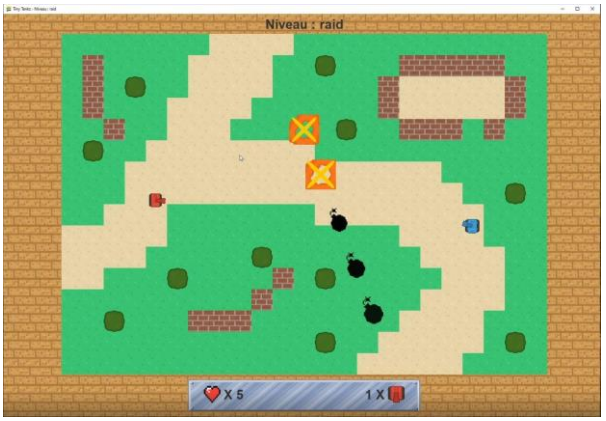
(Dans un niveaux)

(Les bombes tombent sur le terrain)

(liste de niveaux)



(L’avion annonce le raid)



(Les bombes tombent sur le terrain)