Projet d'algorithmique et d'image

Programmation et algorithmique C++ - Synthèse d'Image I

- IMAC 1 - 2023-2024 -

IMAC TOWER DEFENSE

Nombre de personnes par projet : 2 ou 3

Date de rendu : 03 juin 2024 (au plus tard 23h59)

Introduction

Il est temps de mettre en pratique les connaissances acquises en programmation et en synthèse d'images.

Dans ce projet, vous allez créer un jeu de type **tower defense** en utilisant OpenGL. Le Tower Defense est une catégorie de jeu vidéo qui consiste à protéger un lieu, un objet ou une personne à l'aide de tours qui vont attaquer des vagues d'ennemies. Vous trouverez quelques clés pour comprendre cette catégorie de jeu dans cette vidéo Youtube.

Exemple de Tower Defense; source: pinterest

Cahier des charges

Voici les indications que vous devez respecter pour ce projet. Il s'agit des consignes du client. Néanmoins, si vous souhaitez modifier certains points, vous devez au préalable nous demander et nous convaincre des changements. Certains points seront explicités dans les parties suivantes.

Fonctions/Gameplay

Carte: Le jeu affiche une carte ou l'on peut identifier clairement le ou les départ.s des ennemies, leur destination (votre élément à protéger) et plusieurs chemins qu'ils peuvent emprunter.

Tours: L'utilisateur peut poser des tours en dehors des éléments cités au-dessus. Ces tours vont agir lorsqu'un ennemie passe à proximité, par défaut, il va attaquer pour l'empêcher d'atteindre l'objectif, mais vous pouvez imaginer d'autres comportements.

Vague Ennemies: L'utilisateur peut déclencher les vagues d'attaques des ennemies. Un certain nombre d'ennemies vont se succéder pour atteindre l'objectif. Ils apparaissent dans les différents départs et avancent en utilisant les chemins les plus pertinents.

Fin: Lorsque l'utilisateur réussit à repousser les vagues d'attaques ou si un certain nombres d'ennemis atteint leur destination, le jeu affiche un écran de fin (victoire ou défaite)

Projet

Structure du projet : Vous devez organiser votre code en fichiers et dossiers séparés. Nous recommandons l'utilisation d'un système de compilation CMake pour la compilation.

Initialisation de la carte : Vous devez être capable de charger une carte à partir d'un fichier image et un fichier (texte) de configuration.

Algorithmique : Vous devez mettre en place un graphe pour représenter les chemins de la carte. Vous devez également implémenter un algorithme de plus court chemin pour que les ennemies puissent trouver leur chemin vers la sortie.

Synthèse d'image : Vous devez afficher les éléments du jeu (carte, tours, ennemies) en utilisant des sprites (textures) et la bibliothèque OpenGL.

Gestion de Projet: Vous devez utiliser Git pour stocker/partager votre code et nous le rendre.

Compilation: Un système de compilation Cmake devra être intégré à votre projet, d'autant que vous aurez à utiliser des bibliothèques (opengl, lecture d'images, ...). votre projet devra contenir tout ce qui permet de le compiler et fonctionner sur Linux ou Windows (plateforme de développement à préciser dans le rapport). Si développement sur macOS, pensez à tester votre programme sur une autre machine afin que l'on puisse le compiler sur Linux ou Windows afin de le tester.

Le projet est à faire par **binôme** ou **trinômes**. Les **trinômes** devront obligatoirement réaliser deux fonctionnalités supplémentaires par rapport aux binômes. Si vous choisissez une amélioration qui n'est pas dans la liste des améliorations suggérées, vous devrez nous en faire part pour validation.

Rapport

Il contiendra une description des fonctionnalités implémentées de l'application (règles du jeu, etc), un guide succinct d'utilisation, et des captures d'écran.

Éventuellement, si vous souhaitez mettre en avant Un bout de code pour sa performance ou parce qu'il s'agit d'une idée intéressante, vous pouvez l'intégrer dans le rapport.

Post mortem: analyser le travail fourni, qu'est ce qui a bien fonctionné, quels ont été les problèmes rencontrés, comment vous les avez surmontés, auriez vous fait différemment ? Avec plus de temps, qu'est ce que vous pourriez ajouter ?

Ne nous faites pas des romans, vous pouvez faire court, par exemple 2 à 4 pages sans les illustrations.

Notation

Ce projet est commun aux matières : Programmation et algorithmique C++ et Synthèse d'images I. Le projet en lui-même possède deux aspects : une partie algorithmique et une partie infographie. Il y aura une base commune puis une note distinctive pour chaque matière.

Exemple: note commune: 8, note algo: 6, note SI: 5 = note finale algo 14, note finale SI 13

Structure de votre programme

Dans ce projet complexe, il est hors de question de n'utiliser qu'un seul fichier. Il vous faut donc séparer l'application en différents fichiers ***.cpp*** et ***.hpp***. La découpe des fichiers est laissée à votre appréciation mais doit être logique. Globalement, le projet étant scindé en deux parties, il serait logique que la partition des fichiers en tienne compte.

Un système de compilation **Cmake** devra être intégré à votre projet, d'autant que vous aurez à utiliser des bibliothèques (opengl, lecture d'images, ...).

Note importante : Tout rendu de projet sans possibilité de le compiler sur **Linux ou windows** entraînera un 0 ! (sauf si vous avez une raison valable et que vous avez prévenu vos enseignants).

Votre projet devra être organisé à minima dans un répertoire suivant la structure suivante :

```
NomDuProjet/
  \-- src/
  \-- lib/
  \-- data/
  \-- images/
  \-- doc/
  \-- CMakeLists.txt
```

- Le répertoire `src` contient les fichiers sources ***.cpp*** et ***.hpp*** (Vous avez la liberté de placer plutôt les fichiers d'entête ***.hpp*** dans un répertoire `include` si vous préférez ce type de structure de projet mais vous devrez alors adapter votre ***CMakeLists.txt***).
- (Optionnel) Le répertoire `lib` contiendra les fichiers de vos bibliothèques ainsi que tout le nécessaire pour les compiler ou les inclure dans votre projet (Vous pouvez également utiliser des fetch Cmake pour inclure des librairies).
- Le répertoire `data` contient les fichiers ***.itd*** des niveaux (expliqués plus loin dans le sujet) et autres fichiers de données utiles pour votre jeu.
- Le répertoire `images` contient toutes les images du projet (sprite, niveaux, ...).
- Le répertoire `doc` contiendra toute la documentation, dont votre rapport.
- Enfin, un ***CMakeLists.txt*** permettant de compiler le projet.
- (Optionnel) un répertoire `bin` dans lequel sera exporté l'exécutable compilé du projet (le fichier exécutable ne doit pas être inclus dans le dépôt git).
- Tout dossier temporaire `build` (utilisé par cmake par exemple) ne **devra pas être inclus** sur git (**.gitignore**)

Carte et Fichier de description

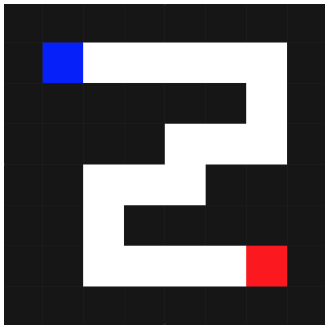
Pour ce projet nous allons utiliser une carte à base de cases (**tile-based**). C'est à dire que la carte est segmentée de cases de même taille. Il y a différents types de cases: **chemin, zone d'entrée, zone de sortie et case vide.**

Données de la carte sous forme d'image

Pour représenter la carte, nous utilisons une image en couleur au format ***.png*** afin d'indiquer visuellement où sont situés les chemins, la/les zones d'entrées ainsi que la zone de sortie.

Attention il ne s'agit pas là de l'illustration de la carte, cans cette image, chaque **pixel** représente une **case** de la carte et la couleur du pixel indique une certaine information (le type entre autre). Ainsi, une image de 10x10 pixels représente une carte de 10x10 cases.

Voilà un exemple d'une carte de 8x8 pixels :



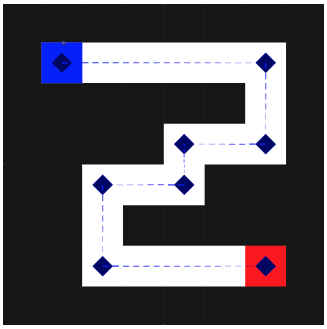
Chaque case devra être d'une couleur précise permettant d'associer à cette case le type de zone. Ici les couleurs utilisées sont :

- Chemin : Blanc (255, 255, 255)
- Zone d'entrée : Bleu (0, 0, 255)
- Zone de sortie : Rouge (255, 0, 0)

Ces couleurs sont à préciser dans le fichier de description de la carte (voir section suivante) et permettront de déterminer de lire l'image et de déterminer le type de chaque case.

Chemin sous forme de graphe

Pour permettre aux ennemies de se déplacer sur la carte, nous allons utiliser une représentation des chemins sous forme de **graphe**. Chaque **nœud** du graphe représente une intersection, un coude. La zone de sortie et d'entrée seront aussi des nœuds. Les **arrêtes** du graphe représentent les chemins reliant ces nœuds. Par soucis de facilité, nous allons considérer des lignes purement **verticales** ou **horizontales**.



Nous utiliserons ce graphe pour calculer le chemin le plus court entre la zone d'entrée et la zone de sortie pour les ennemies.

Fichier de description .itd

Les éléments constitutifs d'une carte sont regroupés dans un fichier texte de configuration simple d'extension ***.itd***.

> ℹ️ Les extensions : L'extension ne change pas la nature d'un fichier, il s'agit surtout d'un prefix qui permet au système d'exploitation (grossièrement parlant) de rapidement savoir si un fichier est de tel ou tel type sans regarder le contenu. Ici on choisit un prefix ".itd" (Imac Tower Defense), on aurait pu choisir ".toto"

Ce fichier sera en fait un simple fichier texte. Il indiquera les différents paramètres nécessaires pour charger les données d'un niveau, notamment les informations relatives à la carte.

```
ITD
# ligne de commentaire
map fichier.png
# Blanc
path 255 255 255
# Bleu
in 0 0 255
# Rouge
out 255 0 0
graph 8
node 0 1 1 1
node 1 6 1 2
node 2 6 3 3
node 3 4 3 4
node 4 4 4 5
node 5 2 4 6
node 6 2 6 7
node 7 6 6
```

La première ligne est constitué d'un mot clé (pour ce projet, ce sera toujours **"ITD"**) indiquant qu'il s'agit d'un fichier de description de carte ITD.

Les lignes d'après seront **ordonnées** selon la position dans le tableau suivant (et donc lu dans un ordre donné) et constituées de deux éléments : un **mot clé** et une **valeur**.

Mot clé	Description	Type de valeur
map	Nom du fichier image représentant la carte	chaîne de caractères
path	Couleur représentant les cases de chemin	Triplet R,V,B (0-255)
in	Couleur représentant la/les zone(s) d'entrée(s)	Triplet R,V,B (0-255)
out	Couleur représentant la zone de sortie	Triplet R,V,B (0-255)
graph	Nombre de noeuds du graphe	entier
node	Description d'un noeud du graphe	Voir ci-dessous

Les mots clés permettent de se **repérer**, de **vérifier** que les lignes sont dans le bon ordre lors de la lecture du fichier et donc d'indiquer si le fichier est mal formaté lors de sa lecture.

De plus, chaque ligne commençant par un **"#"** sera considéré comme une ligne de commentaire et ignoré pour la lecture du fichier (uniquement là pour des annotations si besoin comme dans l'exemple ci-dessus).

Comme vous pouvez le constater, le fichier contient la description du graphe représentant les chemins de la carte.

Un nœud est décrit par un mot clé **node** suivi de plusieurs valeurs :

- Un entier positif ou nul indiquant l'indice de ce nœud.
- Deux coordonnées x, y indiquant la position (en cases) de ce nœud dans l'image/carte (1 pixel = 1 case).
- Enfin, une liste d'indices représentant les nœuds connectés.

Exemple `node 1 6 1 2` corresponds au nœud d'indice 1 de position en cases (6, 1) et connecté au nœud d'indice 2.

Ces nœuds sont de natures différentes (Zone d'entrée des ennemies, zone de sortie, chemin) et cette nature est représentée dans l'image de la carte par un **pixel** d'une couleur particulière. On pourra donc en déduire le type de nœud en fonction de la couleur du pixel central de la case correspondante.

Validité du fichier

Plusieurs conditions doivent être remplies pour qu'une carte soit valide pour le jeu. Votre application devra être capable, lors du chargement, de vérifier si le fichier et donc la carte est valide ou non et d'indiquer à l'utilisateur si ce n'est pas le cas (par un message d'erreur par exemple).

Vous devez vérifier les éléments suivants :

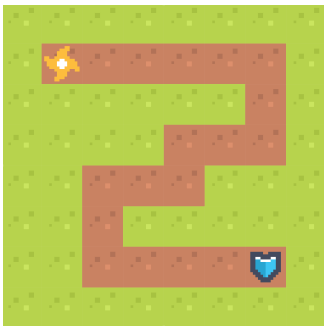
- toutes les lignes nécessaires sont présentes et dans le bon ordre.
- Triplet RGB valide pour les couleurs (comprises entre 0 et 255).
- Fichier image existant.
- Les coordonnées des noeuds sont valides (dans l'image).
- Existence d'au moins une zone d'entrée et de sortie (cette vérification pourra se faire implicitement lors de la recherche du chemin des ennemies).
- Existence d'au moins un chemin entre la zone d'entrée et de sortie (cette vérification pourra se faire implicitement lors de la recherche du chemin des ennemies).

Vous n'avez pas à vérifier que les arrêtes du graphe passent bien par des cases de chemin, Cela pourrait être une sécurité supplémentaire pour assurer une **cohérence visuelle** entre la **carte** et le **graphe** précisé dans le fichier mais ce n'est pas obligatoire (cf améliorations suggérés). Vous devez par contre faire attention lors de la création de votre fichier pour que cela soit le cas auquel cas vous observerez des comportements inattendus (ennemis ne trouvant pas de chemin par exemple).

Affichage de la carte

Pour la partie graphique, vous devez afficher la carte à l'écran.

Mais au lieu de simplement afficher les pixels de l'image, vous devez afficher des cases texturées en fonction du type de case (chemin, zone d'entrée, zone de sortie). Ces textures/images sont généralement appelées des **sprites**. Vous devez donc charger des sprites pour chaque type de case et les afficher à la place des pixels de l'image.



Exemple de rendu en remplaçant les cases par des sprites

Il faudra tenir compte de la taille de vos sprites pour placer correctement les sprites et afficher la carte correctement.

Éléments du jeu

Déroulement d'une partie

Au début de la partie, le joueur dispose d'une certaine somme d'**argent** initiale pour construire des tours de défense. Ces tours sont là pour détruire des ennemis, arrivant par **vagues**, et qui cherchent à atteindre la zone de sortie. Si un ennemi parvient à atteindre la zone de sortie la partie est perdue. Autrement, au bout d'un certain nombre de vagues, si aucun ennemie n'a atteint la zone de sortie, le joueur a gagné.

Chaque ennemie détruit par les tours rapporte de l'argent au joueur. Cet argent permet au joueur de construire des nouvelles tours.

Les chemins

Comme évoqué précédemment, les chemins sont représentés par des pixels d'une certaine couleur dans l'image de la carte mais surtout par un graphe dans le fichier ***.itd***.

Ce graphe est utilisé pour calculer le chemin le plus court entre la zone d'entrée et la zone de sortie pour les ennemies.

Vous devez donc implémenter une structure de **graphe pondéré** pour représenter les chemins de la carte.

Pondéré car les arrêtes du graphe ont une longueur (en nombre de cases) qui correspond à la distance entre deux nœuds. Le **poids** d'une arrête est la **distance** entre les deux nœuds en nombre de cases.

Les ennemies se déplacent en utilisant le plus court chemin entre la zone d'entrée et leur destination. Vous devez implémenter l'algorithme de **Dijkstra**(**recherche de chemin**) pour trouver ce chemin.

Les ennemis

Un ennemi a 3 caractéristiques:

- Des **points de vie**, ils se réduisent lorsqu'il est attaqué et lorsque ces points tombent à 0, l'ennemie meurt et se retire de la carte.
- Une **vitesse** indiquant combien de cases il parcourt en un temps donné.
- Une **récompense** indiquant combien d'argent le joueur gagne en l'éliminant.

Les ennemis arrivent par **vagues**. Idéalement, le niveau de difficulté du jeu devrait augmenter au fur et à mesure que le joueur progresse (nombre d'ennemies par vague, points de vie des ennemies, argent gagné, ...).

S'il y a plusieurs zones d'entrée, vous êtes libre de choisir comment les ennemies arrivent sur la carte (aléatoirement, en alternance, ...).

Les ennemis doivent se déplacer le long des chemins de la carte en continu. Par continuité, on entend que les ennemies ne se déplacent pas de case en case mais de manière "**fluide**". Vous devez implémenter un algorithme de **déplacement** pour les ennemis pour faire en sorte qu'ils se déplacent de manière fluide le long du chemin trouvé par l'algorithme de recherche de chemin.

Astuce: Vous pouvez tout simplement utiliser des **interpolations linéaires** pour faire se déplacer un ennemi d'arrête en arrête.

Graphiquement, vous devez également représenter les ennemies par des **sprites**.

Les tours

Le joueur peut créer des tours de défense qui tireront sur les ennemies. Les tours ont trois caractéristiques :

- La **puissance** indique les **dégâts** effectués par la tour.
- La **portée** indique en cases la **distance** (discrète) à laquelle la tour peut tirer (Cette distance sera mesurée avec la méthode de **Chebyshev**).

> La **distance de Chebyshev** correspond à ce que l'on appelle la **8 connexités** en traitement d'image. C'est une distance qui tient compte des déplacements en **diagonale**.

- La **cadence** indique en nombre de **dixième de secondes** l'intervalle entre deux tirs.

Les différents paramètres de ces différentes tours (puissance, portée, cadence) sont laissés à votre appréciation. Chaque tour a également un coût d'achat pour pouvoir être placée dans la carte.

Voilà des exemples de types de tours :

- Des tours avec une bonne portée, une cadence de tir élevée mais une puissance de tir faible.
- Des tours avec une puissance de tir élevée mais une cadence très lente et une distance de tir moyenne.
- Des tours avec une portée très courte mais avec une cadence de tir élevée et une puissance moyenne.
- ...

Les tours sont également représentées graphiquement par des **sprites**.

Les tours ne peuvent pas être placées sur un chemin, ni sur une autre tour. Il faut donc vérifier que le placement de la tour est valide en regardant le type de case et son contenu.

Interface graphique (IHM)

Votre application contiendra au minimum une fenêtre d'affichage contenant la carte et une fenêtre contenant au moins un bouton permettant de quitter l'application.

Lors du lancement de l'application, une fois le fichier ***.itd*** chargé, le jeu peut commencer après une action de l'utilisateur (bouton, touche, ...).

Le jeu fonctionne en "continu" c'est-à-dire que c'est le **temps** qui rythme la succession des événements (tirs des tours, déplacement des ennemies, ...).

L'utilisateur peut par contre construire une tour quand il le souhaite. L'application doit indiquer visuellement si la case peut recevoir une tour.

Le joueur doit pouvoir mettre en pause l'application ou quitter le jeu à tout moment (via un bouton ou une touche).

Votre interface devra permettre de visualiser les éléments suivants :

- la carte, les ennemies et les tours
- Indiquer l'argent restant disponible
- Indiquer le prix des tours
- Sélectionner une tour à construire (cela peut se faire par des boutons, des raccourcis clavier, ...)
- Placer cette tour dans la carte

Résumé

Pour résumer, voici les éléments que vous devez implémenter dans votre jeu :

- **ITD** : Chargement d'un fichier (texte) de description de carte.
- **Sprites** : Des images pour représenter les éléments du jeu (carte, tours, ennemies).
- **Graphes** : Un graphe pondéré pour représenter les chemins de la carte.
- **Ennemis** : Des ennemis qui se déplacent le long des chemins de la carte pour atteindre la zone de sortie. Minimum 2 types.
- **Déplacement des ennemies** : Un algorithme pour faire se déplacer les ennemies de manière "fluide".
- **Dijkstra** : Un algorithme de recherche de chemin pour les ennemies.
- **Tours** : Des tours de défense qui tirent sur les ennemies. Minimum 2 types.
- **Tirs des tours** : Les tours tirent sur les ennemies à intervalle régulier (la cadence de tir). Les tours tirent sur l'ennemi le plus proche d'elle.
- **Placement des tours** : Un système pour placer des tours sur une case valide.
- **Argent** : Un système d'argent pour acheter des tours. Les ennemies tués rapportent de l'argent.
- **Fin** : Un écran de fin indiquant une défaite ou une victoire.

- **IHM** : Une interface graphique pour visualiser les éléments du jeu (argent, points de vie, ...). Le joueur doit pouvoir mettre en pause le jeu ou quitter à tout moment.

Bonus et améliorations suggérés

Tout outils, spécifications ou fonctionnalités supplémentaires seront récompensés. Nous vous suggérons les améliorations suivantes qui nous semblent intéressantes pour ce projet.

Zones constructibles pour les tours (algorithmique)

Mettons en place des zones constructibles pour les tours. Ces zones sont des cases de la carte où les tours peuvent être placées. Les zones constructibles sont représentées par des pixels d'une certaine couleur dans l'image de la carte.

Dans le fichier ***.itd***, vous devrez ajouter une nouvelle ligne pour indiquer la couleur des zones constructibles.

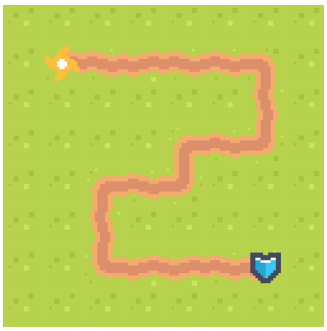
Vous devrez également vérifier que le placement de la tour est valide en regardant si la case sous la tour est bien constructible lors de la construction de la tour.

A* pour la recherche de chemin (algorithmique)

Vous pouvez remplacer l'algorithme de **Dijkstra** par l'algorithme **A*** pour la recherche de chemin. L'algorithme **A*** est un algorithme de recherche de chemin plus efficace que Dijkstra qui utilise une **heuristique** pour guider la recherche.

Placement intelligent des sprites de chemin (algorithmique et synthèse d'image)

Pour l'affichage de la carte, vous devez utiliser des sprites pour représenter les éléments de la carte. Vous pouvez améliorer l'affichage en utilisant des **auto-tiles**. L'idée est d'afficher des sprites différents en fonction des cases adjacentes pour représenter les chemins de manière plus esthétique (au lieu de répéter la même sprite pour chaque case de chemin).

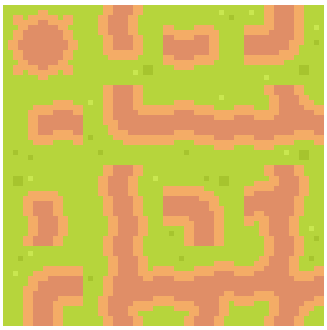


Exemple de rendu avec des auto-tiles

On peut remarquer qu'il a **4** adjacences (cas simple sans tenir compte des diagonales) pour une case (haut, bas, gauche, droite) et donc 16 possibilités de sprites différents pour une case de chemin.

Vous pouvez utiliser ce qu'on appel un masque binaire (**bitmask**) pour représenter les adjacences et donc les différentes possibilités de sprites et choisir le bon sprite à afficher en fonction de ce bitmask (dans un tableau de sprites, dans un tileset, ...).

> Un bitmask est un nombre dont chaque bit représente une information. Par exemple, pour les auto-tiles, on peut utiliser un bitmask de 4 bits pour représenter les 4 adjacences d'une case de chemin. le premier bit représente l'adjacence du haut, le deuxième bit l'adjacence de gauche, le troisième bit l'adjacence du droite et le quatrième bit l'adjacence du bas.



Exemple de tileset utilisé pour générer l'exemple précédent

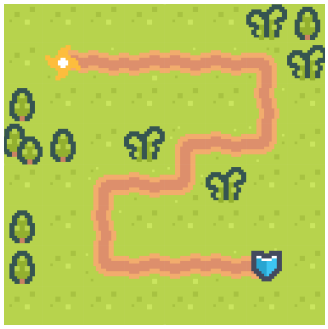
Plus d'explications sur cet article: https://code.tutsplus.com/how-to-use-tile-bitmasking-to-auto-tile-your-level-layouts--cms-25673t

---

Il est même possible d'aller plus loin en utilisant des algorithmes d'auto-tiling plus complexes qui tiennent compte des diagonales ou d'ajouter d'autres règles pour afficher des sprites supplémentaires en fonction de valeurs aléatoires ou de conditions particulières.

Cela peut se faire à l'aide de règles de placement de tiles successives (au lieu d'utiliser la technique du bitmask). Le bitmak est une technique simple et efficace pour commencer mais pourrait se remplacer par une succession de 16 règles de placement de tiles (si voisin haut et gauche alors sprite 1, si voisin haut et droite alors sprite 2, ...).

un exemple avec l'ajout d'**herbes** et **arbres** sur les cases qui ne sont pas des chemins de manière aléatoire :



Sprite animé (synthèse d'image)

Pour les ennemies, vous pouvez ajouter une animation pour les ennemies. Par exemple, vous pouvez faire en sorte que les ennemies aient une animation de marche lorsqu'ils se déplacent.

Cela peut se faire en utilisant plusieurs sprites pour représenter les différentes étapes de l'animation. Ces sprites sont affichés les uns après les autres pour donner l'illusion du mouvement.

Les différentes sprites sont généralement regroupées dans une seule image appelée **sprite sheet**.

Exemple de sprite sheet pour un effet de feu :

Generation du graphe à partir de l'image (algorithmique)

On peut se rendre compte que détailler le graphe dans un fichier texte peut être fastidieux et source d'erreur. Cela fait donc sens de générer ce graphe à partir de l'image de la carte. Cela rendrait la création de niveaux plus simple et plus rapide et éviterait les erreurs de saisie.

L'image étant relativement simple, il est possible de parcourir l'image pixel par pixel (et donc case par par case) pour trouver les noeuds et les arrêtes du graphe.

L'idée est de parcourir l'image et de trouver dans un premier temps un noeud évident (zone d'entrée, zone de sortie) puis à partir de ce noeud itérer dans les 4 directions pour trouver les autres noeuds et les arrêtes du graphe (chemins) de proche en proche. Cela peut se faire de manière récursive ou itérative c'est à vous de voir.

Si vous décidiez d'implémenter cette amélioration, vous pourrez donc omettre la partie **graph** et **node** du fichier ***.itd***. Il faudra le gérer dans votre code (détecter l'absence de ces lignes et générer le graphe à partir de l'image ou introduire un nouveau mot clé dans le fichier ***.itd*** pour indiquer que le graphe doit être généré à partir de l'image).

Si vous avez des questions sur cette amélioration, n'hésitez pas à demander à vos enseignants d'algorithmie pour plus de détails.

Autres améliorations possibles

On peut penser notamment à :

- Différents types de ennemies avec des caractéristiques différentes.
- Différents types de tours avec des caractéristiques différentes.
- Déplacement stochastique des ennemies : déplacement aléatoire à chaque noeud rencontré.
- Différents types de terrain pour les chemins ralentissant ou accélérant les ennemies.
- Visualisation des tirs des tours sur les ennemies.
- Sélection des ennemies et affichage de leurs propriétés.

- Dans le cas de différents types de tours, avoir une résistance propre à chacun des types de tours pour les ennemies.
- Créer une zone de sortie ayant des points de vie, encaissant les dégâts des ennemies avant de perdre la partie.
- Stratégies différentes pour les tours (priorité sur les ennemies les plus proches, les plus faibles, les plus forts, ...).
- Fonctionnalité de sauvegarde et de chargement de partie.
- ...

Néanmoins, si une nouvelle fonctionnalité modifiait même de manière légère les spécifications, notamment le format des fichiers ***.itd***, présentes dans cette description de projet alors cette nouvelle fonctionnalité devra être validée par vos enseignants. Elle devra aussi être indiquée clairement dans le rapport.

Conseils et remarques

- Ce sujet de projet constitue un cahier des charges de l'application. Tout changement à propos des spécifications du projet doit être validée par vos enseignants.

  Le temps qui vous est imparti n'est pas de trop pour réaliser ce projet. N'attendez pas le dernier mois pour commencer à coder.

- Il est très important que vous réfléchissiez **avant de commencer à coder** aux principaux modules, algorithmes et aux principales structures de données que vous utiliserez pour votre application. Il faut également que vous vous répartissiez le travail et que vous déterminiez les tâches à réaliser en priorité.

- Ne rédigez pas le rapport à la dernière minute sinon il sera bâclé.

- Il est **impératif** que chacun d'entre vous travaille sur une partie et non pas tous "en même temps" (plusieurs qui regardent un travailler). Sinon, vous n'aurez pas le temps de tout faire. C'est encore plus vrai pour les trinômes.

- Rappel: Les **trinômes** devront obligatoirement réaliser des fonctionnalités supplémentaires (expliquées dans la section **projet**) par rapport aux binômes.

- Utilisez la bibliothèque standard C++ pour les structures du type listes (**std::vector**), piles (**std::stack**), files (**std::queue**).

- N'oubliez pas de **tester** votre application à chaque spécification implémentée. Il est impensable de tout coder puis de tout vérifier après. Pour les tests, confectionnez-vous tout d'abord de petites cartes (taille 5 par 5 par exemple) avec un chemin extrêmement simple. Si cela marche, vous pouvez passer à plus gros ou plus complexe.

- Vos chargés de TD et CM sont là pour vous aider. Si vous ne comprenez pas un algorithme ou avez des difficultés sur un point (problème technique, compréhension du sujet, etc), **n'attendez pas la soutenance pour nous en parler !**

- Vous ne devez utiliser que des assets libres de droit pour votre projet. Vous devez citer les sources des assets utilisés dans votre rapport. Vous pouvez bien sûr créer vos propres assets.

Sources

Assets utilisés pour les illustrations :

- kenney: pixel-shmup
- brullov: fire-animation

Logiciel utilisé pour créer les images :

- Ldtk

> Written with StackEdit.