Projet d'algorithmique et d'image

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

- IMAC 1 - 2024-2025 -

IMAC DIGGER

Nombre de personnes par projet : 2 ou 3

Date de rendu : 02 juin 2025 (au plus tard 23h59)

Soutenance : 03 juin 2025

Introduction

Pour mettre en pratique les connaissances acquises en programmation et en synthèse d'images vous allez devoir réaliser un projet sous la forme d'un jeu vidéo. Ce projet est commun aux matières **Programmation et algorithmique C++** et **Synthèse d'images I**.

Dans ce projet vous devez réaliser un jeu vidéo inspiré de **Mr. Do!**, **Digger** ou encore **Pacman** dans lequel votre but est de vous déplacer dans un niveau pour collecter des objets tout en évitant des ennemis.

Voilà deux exemples de jeux de ce type :

- Digger
- Diamond digger

Visuel du jeu diamond-digger; source: itch.io

Cahier des charges

Voici les indications que vous devez respecter pour ce projet. Néanmoins, si vous souhaitez modifier certains points, vous devez au préalable nous demander et faire valider ces changements.

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.

Synthèse d'images : Vous devez afficher les éléments du jeu (carte, ennemis) 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 **trinôme**. Les **trinômes** devront obligatoirement réaliser une fonctionnalité supplémentaire 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 la faire valider.

Thème du jeu

Vous n'êtes pas obligé de respecter le thème du jeu (un mineur qui cherche des gemmes) mais vous devez respecter les spécifications du projet. Vous êtes libres de choisir un autre thème, exemple :

- Un pirate qui cherche des trésors (au milieu des squelettes).
- Un explorateur qui cherche des artefacts (au milieu des momies).
- Un voleur qui cherche des objets de valeur (au milieu des gardes).
- Un astronaute qui cherche des cristaux (au milieu des aliens).
- Une sorcière qui cherche des ingrédients (au milieu des monstres).

Rapport

Il contiendra une description des fonctionnalités implémentées (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 (mais rester succinct).

Ajoutez enfin une partie "Post mortem" pour 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**. 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/
  \-- images/
  \-- CMakeLists.txt
  \-- README.md
  \-- .gitignore
```

- 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 `images` contient toutes les images du projet (sprite, ...).
- 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).
- Un fichier ***README.md*** contenant les instructions pour compiler et exécuter le projet. Votre rapport de projet peut être intégré à ce fichier ou dans un fichier séparé (ex: doc/rapport.pdf).
- Tout dossier temporaire `build` (utilisé par CMake par exemple) ou exécutable (**.exe**) ne **devra pas être inclus** sur git (**.gitignore**) sous peine de pénalité.

Carte

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 devra y avoir différents types de cases:

- **Bloc vide**: case vide, sans objet ni ennemi ou le joueur peut se déplacer librement.
- **Bloc plein**: case pleine, le joueur ne peut pas se déplacer dessus mais peut détruire cette case (ex: un bloc de terre).
- **Objet**: case contenant un objet à collecter (ex: une gemme, une pièce d'or, une pomme, ...).
- **Obstacle**: case pleine qui ne peut pas être détruite (ex: un mur, une pierre, ...).
- **Piège**: case pleine qui ne peut pas être détruite mais qui fait perdre le joueur (ex: un trou, une flamme, ...).

Ces cases sont représentées par des sprites (images) qui sont affichées à l'écran. Vous devez donc créer ou trouver des sprites pour représenter ces cases.

Génération de la carte

Vous devez générer la carte de manière procédurale (c'est à dire que la carte est générée aléatoirement selon des règles définies et n'est pas prédéfinie).

Il existe de nombreux algorithmes de génération procédurale mais pour ce projet nous allons utiliser un algorithme de génération procédurale de type **cellular automata**. Cet algorithme est relativement simple à mettre en place et s'adapte très bien à la génération de cartes de type **cave** (donc avec des cases pleines et vides).

Description de l'algorithme

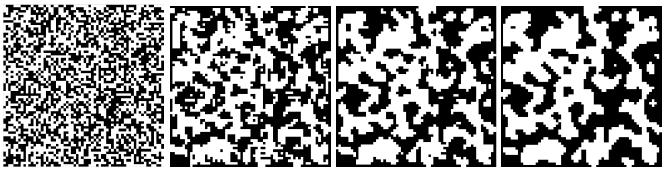
L'algorithme de génération procédurale de type **cellular automata** fonctionne de la manière suivante:

1. On initialise la carte avec des cases vides et pleines de manière aléatoire (ex: 50% de cases pleines et 50% de cases vides). (Vous pouvez faire varier la probabilité)
2. On met à jour les états des cases de la carte en fonction de l'état actuel selon les règles suivantes:
    - Si une case est pleine et a au moins 4 cases pleines voisines, elle reste pleine sinon elle devient vide.
    - Si une case est vide et a plus de 4 cases pleines voisines, elle devient pleine.

> Autrement dit: Une case sera pleine à l'étape suivante si le nombre de cases pleines voisines (en se comptant elle même) est supérieur ou égal à 5, sinon elle sera vide.

3. On répète l'étape 2 un certain nombre de fois (ex: 4 fois) pour affiner la carte.

Exemple de 4 itérations de l'algorithme:



Ensuite, vous pouvez ajouter des objets (ex: gemmes) et cases pièges sur la carte de manière aléatoire. Vous pouvez également ajouter des ennemis sur la carte en fonction de la taille de la carte (ex: 1 ennemi pour 20 cases vides).

Vous pouvez également placer le point d'apparition du joueur aléatoirement sur la carte (idéalement pas trop proche d'un ennemi par exemple).

Flow field pathfinding

Pour pouvoir faire se déplacer les ennemis sur la carte, il nous faut un algorithme de recherche de chemin. Nous allons utiliser un algorithme de type **flow field pathfinding**.

L'idée de cet algorithme est de créer un "champ de direction" (notre **Flow field**) qui indique la direction à prendre pour atteindre la destination. Chaque case contient la direction à prendre (sous la forme de la prochaine case sur laquelle se rendre) pour atteindre la destination.

Une fois ce flow field créé, il suffit de faire se déplacer les ennemis dans la direction indiquée par la case sur laquelle ils se trouvent (ou mieux une interpolation linéaire des directions des cases adjacentes si il se trouve entre plusieurs cases).

Pour arriver à générer ce champ de direction nous utiliserons un algorithme de type **Breadth First Search** (BFS) qui en partant d'une case de destination (le joueur) va remplir toutes les cases adjacentes avec la direction à prendre pour atteindre cette case.

Le principe de l'algorithme est le suivant:

- On initialise une file de cases à explorer (ex: la case du joueur) et on initialise toutes les cases avec une direction nulle.
- Tant qu'il nous reste des cases à explorer on prend la dernière et pour chaque case adjacente on assigne une direction (la case d'où l'on vient) et on les ajoute à la file des cases à explorer (si elles ne sont pas déjà explorées).
- On répète l'étape 2 jusqu'à ce que toutes les cases soient explorées et les directions assignées.

Vous trouverez des explications supplémentaires et des animations sur le site de RedBlobGames.

> Ce genre d'algorithmes s'applique généralement sur des graphes. On peut voir notre carte 2D comme un graphe où chaque case est un noeud et les cases adjacentes sont les arêtes du graphe. On peut donc utiliser les algorithmes de recherche de chemin sur ce "graphe" simplifié.

Il suffit de générer le flow field uniquement lorsque c'est nécessaire (ex: lorsque le **joueur se déplace** ou que la **carte change**). C'est pourquoi c'est très efficace quand le nombre d'ennemis est important et/ou que la carte ne change pas ou pas trop souvent car il n'est pas nécessaire de recalculer le meilleur chemin pour chaque ennemi. (type A* ou Dijkstra par exemple).

Le joueur

Le joueur doit être représenté par un sprite (image) et doit pouvoir se déplacer de manière fluide. (Il est recommandé d'utiliser une taille du joueur identique à celle des cases de la carte mais ce n'est pas obligatoire).

Les déplacements devront être fluides et non pas en "case par case". Bien que la carte soit composée de cases, les positions du joueur et des ennemis ne seront donc pas forcément alignées sur les positions des cases. (vous pouvez utiliser les flèches du clavier ou des touches **ZQSD**).

Le joueur doit pouvoir "miner" c'est à dire pouvoir se déplacer sur un **bloc plein** et le détruire (cette case deviendra alors un bloc vide). Il peut également se déplacer sur objet (ex: une gemme) pour le récupérer. Le joueur ne peut pas détruire un ennemi.

Les ennemis

Les ennemis doivent se déplacer en empruntant le chemin trouvé par l'algorithme de recherche de chemin. Il doivent se déplacer de manière fluide (pas case par case) en fonction de leur vitesse (ex: 1 case par seconde).

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

Interface graphique (IHM)

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

Vous devez faire un **menu de démarrage** qui permet de lancer le jeu. Ce menu doit contenir au moins un bouton pour démarrer le jeu et un bouton pour quitter l'application.

Vous devez aussi afficher un **écran de fin** (victoire ou défaite). Lorsque le joueur réussit à récupérer tous les objets de la carte, il a gagné. Si un ennemi atteint le joueur, il a perdu.

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

L'utilisateur peut par se contre déplacer quand il le souhaite. Il doit pouvoir se déplacer à l'aide des flèches (ou des touches **ZQSD**) du clavier.

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 ennemis et les objets
- Indiquer le nombre d'objets à collecter (ex: 3 gemmes restantes)

Résumé

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

- **Sprites** : Des images pour représenter les éléments du jeu (carte, ennemis, ...).
- **Carte** : Une carte à base de cases (tile-based) avec des cases de même taille.
- **Génération procédurale** : Un algorithme de génération procédurale pour générer la carte (type cellular automata).
- **Flow Field pathfinding** : Un algorithme de recherche de chemin utilisant une grille.
- **Ennemis** : Des ennemis qui se déplacent en suivant le chemin trouvé par l'algorithme de recherche de chemin. Minimum 2 ennemis sur la carte.
- **Minage** : Le joueur peut détruire des blocs pleins et collecter des objets (ex: gemmes).
- **Déplacements** : Déplacement fluide (pas case par case) du joueur et des ennemis.
- **Fin** : Un écran de fin indiquant une défaite ou une victoire.
- **IHM** : Une interface graphique pour visualiser les éléments du jeu (nombre d'objets à collecter, ...). Le joueur doit pouvoir mettre en pause le jeu ou quitter à tout moment.

Bonus et améliorations suggérés

Nous vous suggérons les améliorations suivantes qui nous semblent intéressantes pour ce projet.

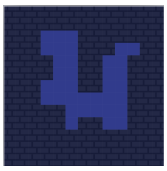
Types de terrain additionnels (algorithmique)

- Vous pouvez ajouter un ou plusieurs autres types de terrain:
    - **Ralentissement**: une case qui ralentit le joueur et les ennemis (ex: un marécage, une flaque d'eau, ...).
    - **Accélération**: une case qui accélère le joueur et les ennemis (ex: un chemin de terre, une route, ...).
    - **Téléportation**: une case qui téléporte le joueur ou les ennemis à un autre endroit de la carte (ex: un trou, une porte, ...).
    - **Bloc plein double**: une case pleine qui peut être détruite mais demande deux destructions avant de devenir vide (avec un état intermédiaire lors de la destruction).
    - ...

Placement intelligent des sprites (algorithmique et synthèse d'images)

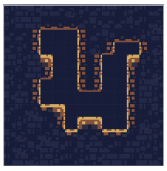
Pour l'affichage de la carte, vous devez utiliser des sprites pour représenter les éléments de la carte.

carte état plein/vide



Sprites sur les cases pleines

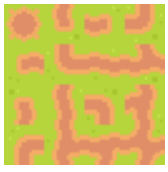
Vous pouvez améliorer l'affichage en utilisant des **auto-tiles**. L'idée est d'afficher des sprites différents en fonction des cases adjacentes pour représenter les cases de manière plus esthétique (au lieu de répéter la même sprite pour chaque case).



Exemple de rendu avec des auto-tiles

Vous pouvez utiliser ce qu'on appelle un masque binaire (**bitmask**) pour représenter les adjacences (cas simple sans tenir compte des diagonales: haut, bas, gauche, droite) 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. le premier bit représente l'adjacence du haut, le deuxième bit l'adjacence de gauche, le troisième bit l'adjacence de 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.

Sprite animé (synthèse d'images)

Vous pouvez ajouter une animation pour vos sprites. Par exemple, vous pouvez faire en sorte que les ennemis 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 :

Niveau de difficulté et ennemis

- Vous pouvez ajouter un niveau de difficulté au jeu en ajoutant des types ennemis supplémentaires ou en augmentant la vitesse de déplacement des ennemis.

Dans ce cas, il faudra un minimum de 3 types d'ennemis différents.

Vous pouvez également ajouter des niveaux de difficulté qui modifient la taille de la carte, le nombre de collectibles et ajouter des cases de type **obstacle**.

Dans ce cas, il faudra au minimum 3 niveaux de difficulté différents et que la carte soit générée différemment en fonction du niveau de difficulté.

Génération procédurale améliorée (algorithmique)

Vous pouvez améliorer la génération procédurale de la carte en ajoutant des algorithmes de génération procédurale plus avancés pour avoir des cartes plus variées et des "salles", "biomes", "zones" plus intéressantes.

Ou encore ajouter une étape de filtrage pour éviter les zones inaccessibles (flood fill filtering).

Il est aussi possible de faire varier les règles et regarder l'état de la génération 2 itérations en arrière au lieu de seulement 1 itération. Ou encore de répéter l'algorithme de génération plusieurs fois avec des règles différentes (pour filtrer les îlots constitués d'une seule case pleine par exemple).

Cette source mentionne une telle variation de l'algorithme.

Champ de vision limité (synthèse d'images)

Comme on peut le voir dans l'exemple de jeu **diamond-digger**, il est possible d'ajouter un effet qui masque une partie de l'écran pour rendre le jeu plus intéressant (généralement appelé **fog of war** dans les jeux vidéo). Dans cet exemple de jeu, le champ de vision s'agrandit au fur et à mesure que le joueur trouve des objets "torches" pour éclairer la carte. A vous de trouver comment implémenter cette fonctionnalité (shader, texture, ...).

Autres améliorations possibles

On peut penser notamment à :

- Utilisation de "pouvoir" pour le joueur (ex: ralentir les ennemis, détruire un ennemi, placer un bloc, ...).
- Fonctionnalité de sauvegarde et de chargement de partie.
- Ajout d'un système de score (ex: en fonction du temps, du nombre d'ennemis tués, ...).
- Ajout d'un système de vie (ex: le joueur a 3 vies et perd une vie lorsqu'il est touché par un ennemi).
- Ajout d'un système de progression (ex: le joueur gagne des niveaux et débloque des compétences ou des objets qu'il peut réutiliser dans de nouvelles parties).

Néanmoins, si une nouvelle fonctionnalité modifiait même de manière légère les spécifications 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 validé par vos enseignants.
- Le temps qui vous est imparti n'est pas de trop pour réaliser ce projet. N'attendez pas le dernier moment 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 au minimum une fonctionnalité supplémentaire (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. 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

Algorithmes de génération procédurale:

- roguebasin: Cellular Automata Method for Generating Random Cave-Like Levels
- generativelandscapes: cave cellular automaton algorithm

Recherche de chemin:

- RedBlobGames: Flow Field Pathfinding

Autres ressources:

- tilesetter: outil de création de tileset
- celeste exemple d'auto-tiling
- tilekit: outil d'auto-tiling avancé

Assets utilisés pour les illustrations :

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

Logiciel utilisé pour créer les images :

- Ldtk