Matthieu FARANDJIS, Julien LEEDER
E3 IMAC

PikaTrain

Présentation de notre projet de Synthèse d'Image.

Plan

I - Introduction

Dans le cadre de notre projet de synthèse d'image, nous avons décidé de partir sur le thème de Pokemon pour nos différentes productions. Les rails et le train sont donc assortis à Pikachu, et notre gare est la reproduction de la gare de la ville d'Illumis des jeux Pokémon XY et Pokémon ZA.

II - Détails

- a) Lien du dépôt GitHub

https://github.com/Farandjis/imac1_PikaStar
⚠ le dépôt est privé.

- b) Organisation du dépôt

  - **assets** : dossier contenant les shaders et les textures
    - **textures** : les textures de notre projet
    - **shaders** : les shaders de base du projet, contenant le flat et le phong shadding
  - **img_rapport** : les images pour ce document
  - **Projet** : le projet C++ avec chaque fichier nommé en fonction de sa fonction.
    - Le fichier principal du projet, à exécuter, est main.cpp
    - les fichiers .cpp contiennent le code
    - les fichiers .hpp et .h sont l'entête des éléments détaillés dans les .cpp, pour vulgariser.
  - **third_party** : les modules importés
  - **README.md** : ce rapport au format MarkDown

- c) Mode d'installation (compilation et exécution)

Ce programme a été développé sous Windows 11 et Ubuntu 24.04.4 LTS et fonctionne parfaitement sur ces deux systèmes.

- d) Commandes utilisateurs

  - Touche Escape : Permet de fermer la fenêtre
  - Touche C : Permet de mettre de désactiver la caméra et de faire balader la souris dans la fenêtre
  - Touches ZQSD : Permet de faire bouger la caméra (haut - gauche - bas - droite) + mouvement de la souris en vue FPS
  - Touche L : Affiche le mode maillage des objets
  - Touche P : Repasse les objets en forme pleine
  - Touches G : Afficher ou non la grille
  - Touches K : Activer ou non les lumières (mode flat shading ou phong shading)

- Barre espace : Nous permet de revenir à la position initiale du projet (plus zoomé sur le train, mais nous gardons les mêmes valeurs d'angle pour la vue)
- Touches T : Nous permet d'avoir un top shot de la scène (positionnement en vue du dessus)

❗Si la touche n'est pas inclue dans le pannel de touches, un message d'erreur est affiché : "Touche non geree"

- e) Résultats obtenus

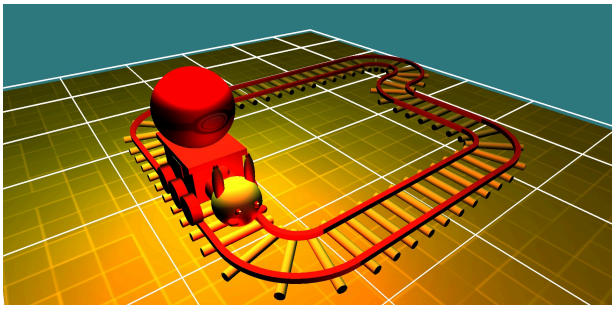


Image du train et des rails

Image de la gare

A titre de comparaison, voici à quoi ressemble la gare d'Illumis dans les jeux (la forme est reprises de Pokémon ZA, les textures de Pokémon XY (et la porte de Minecraft)) :

A gauche : Pokémon XY, à droite : Pokémon Legends ZA

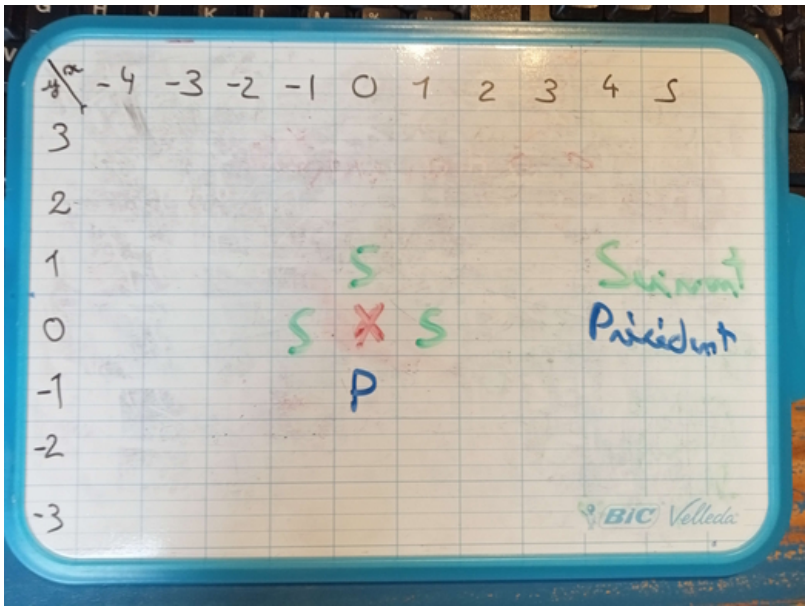
III – Détail d'une fonctionnalité

Dans cette partie, nous détaillons le fonctionnement de la fonction `determinerMarcheASuivre` se situant dans le fichier `deserialisation.cpp` .
Cette partie étant développé par moi, Matthieu, j'utilise "je" et j'évoque mes pensées.

L'objectif est de construire un circuit ferroviaire en fonction du circuit.json.
Nous sommes à l'étape où nous avons la liste des coordonnées complète des rails, il ne reste plus qu'à les interprétés afin de placer ces rails en OpenGL.
L'idée de placer des rails me faisait penser à Scratch, et je me suis dit que ça serait quand même super pratique d'avoir une fonction "avancer()", "tournerAGaucher()", "tournerADroite()" (comme dans Scratch) et exécuter chacune de ces fonctions les unes à la suite de l'autre.

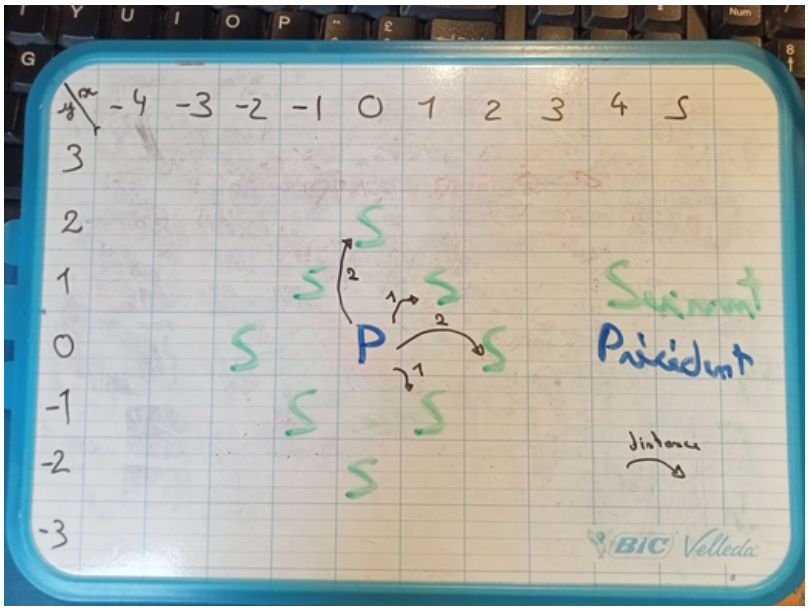
Alors j'ai développé cette partie dans l'optique de rendre la manipulation aussi simple que sur Scratch.

Pour déterminer quelle fonction parmi les trois nous devons exécuter, nous devons déterminer le type de rail à placer : droit, courbé vers la gauche, courbé vers la droite.
Pour cela, je compare le rail qui précède (noté P) notre rail X et le rail qui le succède (noté A). Il y a 3 possibilités : avant, gauche, droite/



ardoise_precedent_suivant.jpg

Peu importe le placement du rail X par rapport au rail précédent P, le rail droit aura TOUJOURS une distance de 2 (sinon, c'est un rail courbé).
Donc c'est facile, on calcule la distance sur les deux axes, si on obtient 2 pour l'un des deux, alors le rail X est un rail droit.



ardoise_distances.jpg

Remarque : ici le rail X n'est pas représenté, mais on devine son emplacement.

Ce qui donne le code suivant :

```
if (abs(distanceX) == 2 or abs(distanceY) == 2)
{
    directionsDesRails.push_back(TOUT_DROIT); // nous sommes un rail droit
}
```

Sinon, c'est un rail courbé, on va déterminer s'il est courbé vers la gauche ou vers la droite.
On remarque que sur l'axe X, quand le rail suis notre repère : côté Gauche < côté Droit
Sur ardoise_precedent_suivant.jpg, si on compare S à gauche avec P, on a S.x < P.x (car -1 < 0), si on regarde S à droite, on a S.x > P.x (car 1 > 0).

Ce qui donne le code suivant :

```
else if(combinaisonsDePositionsRailSuivant.x < combinaisonsDePositionsRailPrecedent.x)
{
    directionsDesRails.push_back(TOURNE_A_GAUCHE);
}
else if(combinaisonsDePositionsRailSuivant.x > combinaisonsDePositionsRailPrecedent.x)
{
    directionsDesRails.push_back(TOURNE_A_DROITE);
}
```

PROBLÈME

Là le code fonctionne parfaitement bien, car notre rail est dirigé vers le haut ↑, mais quand on va tourner à gauche ←, le repère va changer !

Cela donne les possibilités de repère suivant (soit i allant de 1 à 4) :

1. ↑ Gauche Droite (car Gauche < Droite)
2. ← Bas Haut (Bas < Haut)
3. ↓ Droite Gauche (Droite < Gauche) (car ici, la droite du train correspond au côté gauche de la map)
4. → Haut Bas (Haut < Bas) (car ici, le devant (haut) du train correspond au derrière (bas) de la map)

Lorsqu'on tourne à gauche, i augmente de 1 (on passe de 2. à 3. par exemple), quand on tourne à droite, i diminue de 1 (on passe de 2. à 1. par exemple).
Alors, j'ai conçu deux listes (pour les deux rails P et S) afin que nos else if fonctionnent dans toutes les directions.
La difficulté étant que lorsque nous sommes dirigé vers le bas ou vers la droite, notre droite est là où on décrémente et notre gauche là où on incrémente, donc pour ces deux cas, je fais -1* pour inverser la gauche et la droite du rail pour pouvoir les comparers avec no else if.

Et parce que dans nos deux listes, 5. (index 5 on va dire) n'existe pas, car il correspond à 1, et -1. n'existe pas, car il correspond à 4 (index 4), on effectue un modulo.
Ce modulo permet de nous assurer que lorsqu'on tourne, on lira bien le repère dont nous avons besoin, mais aussi que nous n'aurons jamais une erreur de type "out of range".

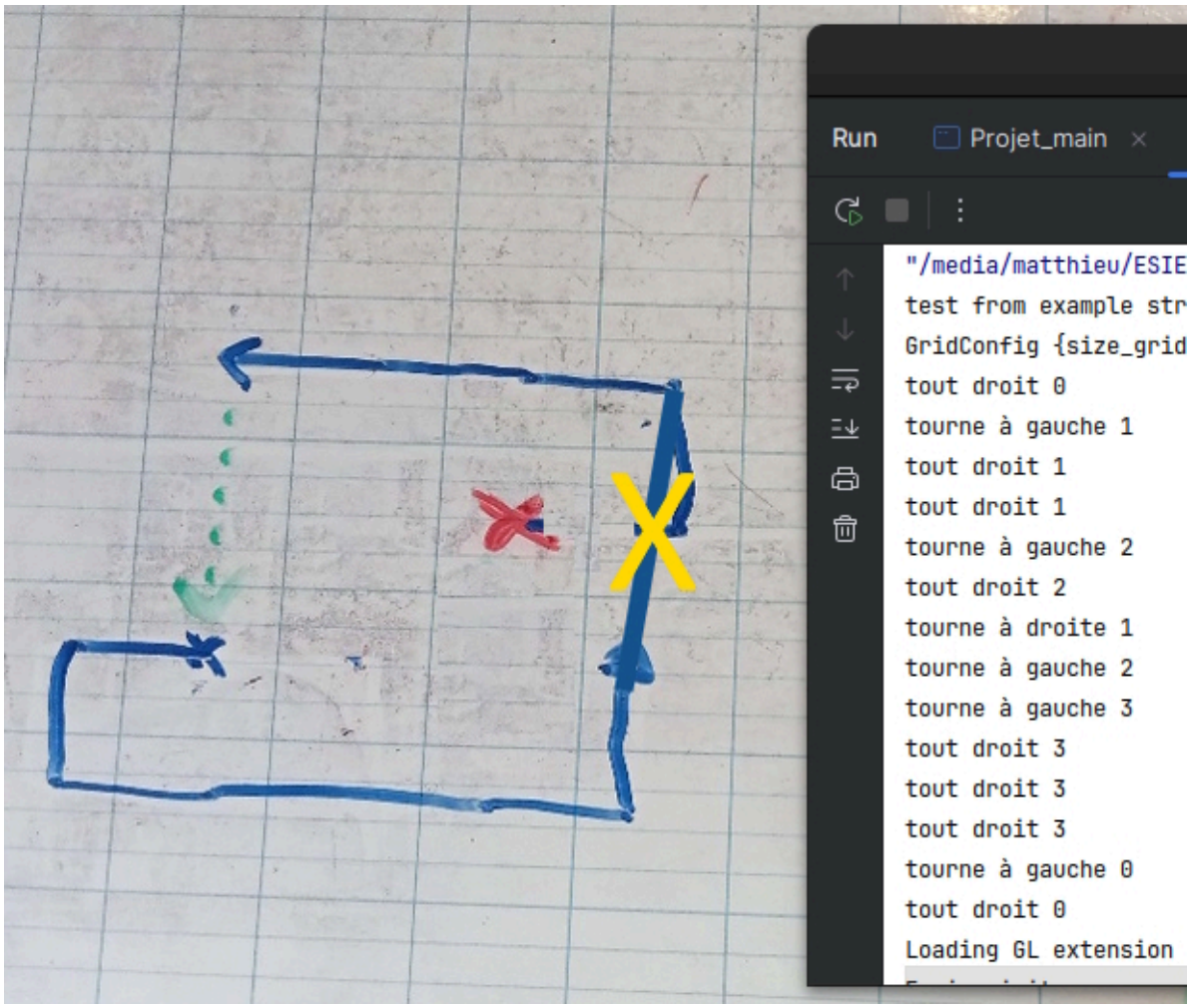
Ce qui donne le code suivant :

```
vector<int> combinaisonsDePositionsRailSuivant = {
            railSuivant.x,
            railSuivant.y,
            railSuivant.x * (-1),
            railSuivant.y * (-1),
        };

        vector<int> combinaisonsDePositionsRailPrecedent = {...};

        if (abs(distanceX) == 2 or abs(distanceY) == 2) {...}
        else if(combinaisonsDePositionsRailSuivant[(4 + base) % 4] < combinaisonsDePositionsRailPrecedent[(4 +
        else if(combinaisonsDePositionsRailSuivant[(4 + base) % 4] > combinaisonsDePositionsRailPrecedent[(4 +
```

On obtient ainsi une liste de Direction, suivant le parcours du circuit.



ardoise_circuit.jpg

Le rail de départ est le X jaune, c'est un rail qui va tout droit, vers le haut de l'image. Son rail suivant est un rail qui tourne à gauche.

On remarque également que la valeur juste à droite correspond à l'index de notre liste des repères (mais allant de 0 à 3 et non de 1 à 4 comme écrit dans ce document)

Pour le placement des rails en OpenGL, il suffira de :

1. voir quel type rail doit-on poser
2. préparer notre rail (s'assurer que nous aurons bien un rendu d'un rail qui tourne à droite par exemple), sans tenir compte dans quel sens on est nous sommes dirigés (vers le bas ? à gauche ?)
3. placer le rail, sans tenir compte du repère (car le repère a possiblement été modifié juste avant)
4. se positionner pour accueillir le prochain rail (c'est-à-dire, se déplacer et tourner et donc en quelque sorte, changer le repère)
5. revenir à l'étape 1, mais pour le rail suivant dans la liste des directions, jusqu'à ce que nous n'ayons plus de rail à poser.

À noter que dans le cas du premier rail de la liste des coordonnées (json), le rail précédent correspond au dernier rail de la liste des coordonnées (json).
Egalement, le rail suivant du dernier rail de la liste des coordonnées (json), correspond au premier rail de la liste des coordonnées (json).

IV – Éléments du code fourni modifié

Nous avons modifié les éléments suivants :

- Dans `Projet/deserialisation.h`
  - Ajout de `vector<Direction> marcheASuivreCircuit {};` dans la structure `GridConfig` (plus simple pour nous lors du placement des rails en OpenGL)
  - Ajout de l'énumération Direction
  - Ajout de la structure PositionBis (car à cause de la macro, je ne pouvais pas utiliser Position comme je le voulais donc j'ai créé PositionBis uniquement pour un cas dans le code)
- Dans `third_party/glbasimac/glbasimac/tools/basic_mesh.hpp`
  - Ajout de `inline IndexedMesh* basicCubeMaisPourLaMosaique(float width, float repeat)` (sinon, on ne pouvait pas répéter les textures)
  - Ajout de `inline IndexedMesh* basicTriangle(float width, float height, float depth)` (afin d'avoir des triangles en 3D)

Nous tenons à préciser que le fichier `circuit.json` contient l'exemple du json dans le sujet du projet tel qu'il est.
De ce fait :

- Nous lisons le json tel qu'il est, sans avoir modifié la logique du code partagé par Monsieur DE SMET (on a développé une fonction C++ de conversion)
- Nous avons géré la case manquante (le programme complète le tronçon manquant à l'aide d'une fonction C++)