Gestion de ressources

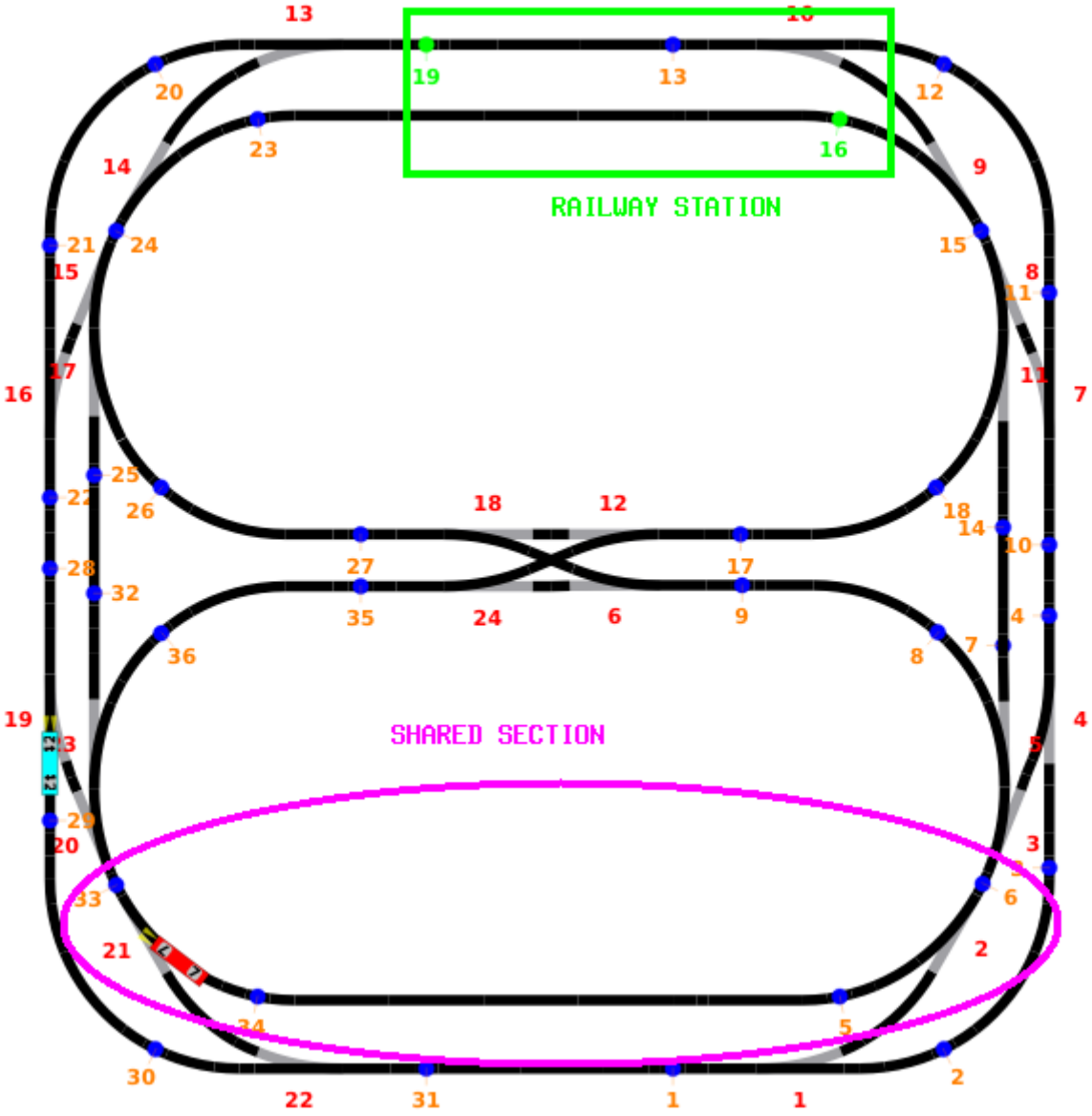
Auteurs: Arthur Junod et Edwin Haeffner

Description des fonctionnalités du logiciel

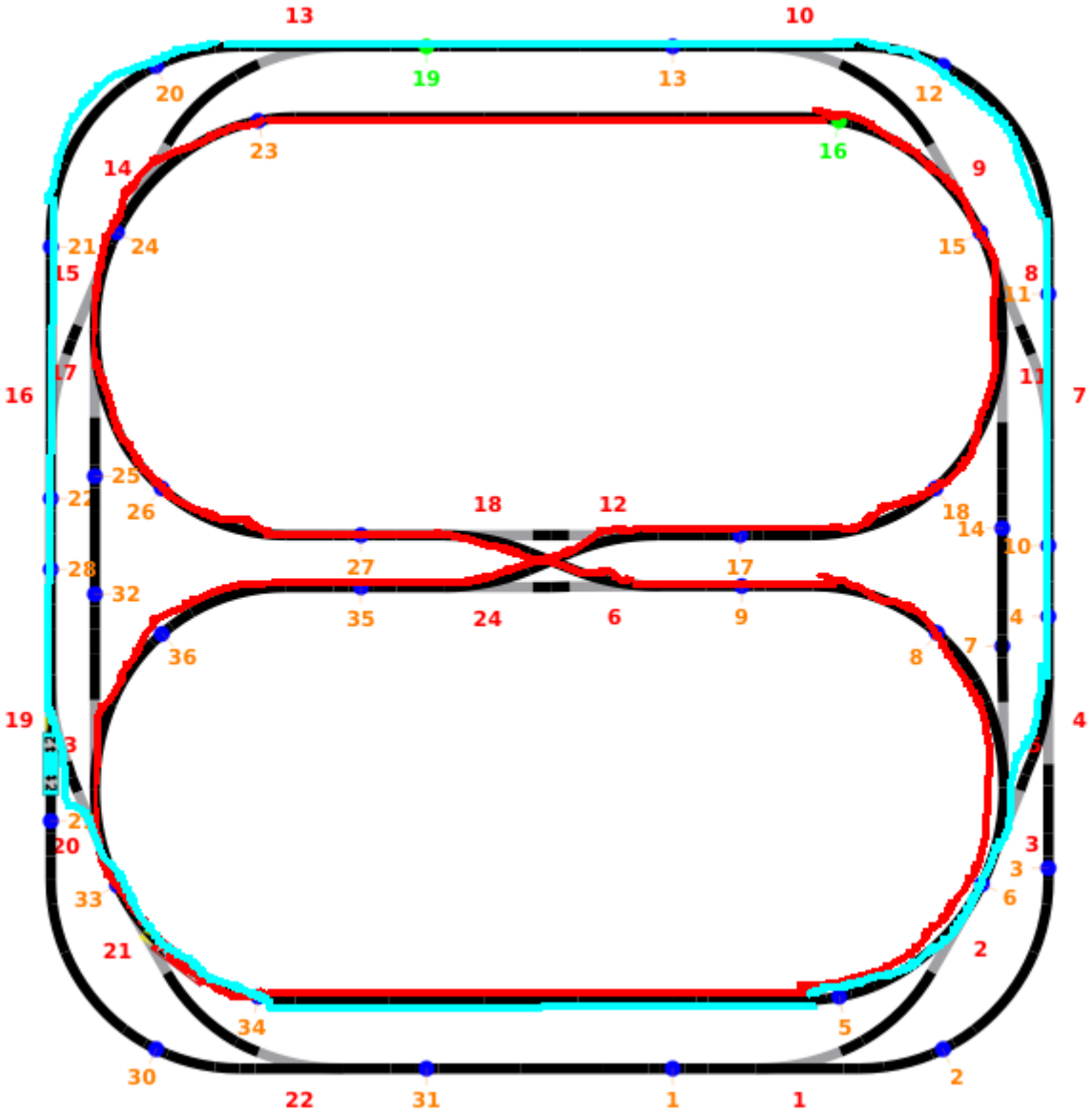
Ce logiciel simule une maquette de petits trains. Le but est d'avoir un système d'attente à une gare et une gestion d'accès concurrent d'une voie ferrée entre les deux trains (ou plus).

Au final, on veut que deux trains qui se partagent une voie ferrée puissent se croiser sans se percuter.

Gare et section partagée:



Parcours:



Choix d'implémentation

cppmain.cpp/emergency_stop()

Pour l'emergency stop nous avons décidé de juste fixer la vitesse des locomotives à 0, ainsi elle ne pourront plus redémarrer et seront bloquées à l'endroit où l'ES s'est activé.

Nous voulions, au début, arrêter également les threads grâce à `requestStop()` mais comme il nous est impossible d'accéder à ceux-ci depuis la fonction `emergency_stop()` (car ils sont initialisés dans une autre fonctions `cmain()`) nous en sommes restés à simplement arrêter les locomotives (nous ne savions pas si nous pouvions modifier la fonction `cmain()` comme nous le souhaitions). Si nous devions arrêter les threads nous aurions donc appelés `requestStop()` sur chacun d'eux et dans *locomotiveBehaviour.cpp* vérifier dans la boucle while que le flag avait été levé. Ce qui veut dire que actuellement la boucle while du locomotiveBehaviour ne peut pas s'arrêter.

Nous n'avons également pas trouvé de meilleur moyen pour arrêter chaque locomotive que de faire l'appel manuellement sur chacune d'entre elles. Cela aurait pu être simplifié s'il existait un vector de locomotive.

cppmain.cpp

Nous avons juste rajouté un paramètre `nbLoco` qui nous sera utile dans *Synchro.h* afin de savoir la locomotive qui est arrivée en dernier à la gare.

locomotiveBehaviour.h

Nous avons ajouté des paramètres quand nous faisons appel au constructeur. Ces paramètres nous seront utiles ensuite pour définir le parcours que fera chaque locomotive.

param	**description**
`gare`	Le contact qui défini la gare pour notre locomotive.
`debutSharedSection`	Le contact de départ de notre section partagée.
`finSharedSection`	Le contact qui défini la fin de la section partagée.
`numSwitchSharedSection`	L'aiguillage qui sépare les trajet de nos 2 locomotives à la fin de la section partagée.
`deviation`	Le sens que doit prendre l'aiguillage de fin.

locomotiveBehaviour.cpp/run()

Nous n'avons que modifié l'intérieur de la boucle while en définissant un parcours qui serait identique au niveau des appels de fonctions entre les locomotives mais différents dans les paramètres que prendraient ces fonctions.

1. La locomotive attend le contact qui définit la gare.
2. Une fois atteint elle fait appel à la fonction `Synchro->stopAtStation()`.
3. Ensuite elle attend le contact du début de la section partagée.
4. Une fois atteint `Synchro->access()` est appelé.
5. Finalement elle attend le contact avec la fin de la section partagée.
6. Elle bouge l'aiguillage de la fin de la section partagée dans la direction de son parcours.
7. Elle fait appel à `Synchro->leave()`.

Synchro.h

Variables

name	**description**
mutex	Utilisé pour protéger les variables partagées dans les fonctions.
waitingSection	Semaphore utilisé pour bloquer les locomotive qui doivent donner la priorité dans la section partagée.
waitingStation	Semaphore utilisé pour faire attendre les locomotives déjà à la gare toutes les autres.
fifo	Mutex utilisé pour s'assurer que la locomotive qui attend depuis le plus longtemps à la section partagées soit la première à partir.
isInSharedSection	Booléen indiquant s'il y a une locomotive dans la section partagée.

name	description
nbWaiting	Nombre de locomotive qui attendent avant la section partagée.
nbLoco	Nombre de locomotive qui se trouvent en tout sur la maquette.
lastArrivedNum	Numéro de la locomotive qui est arrivée en dernier à la gare.
nbWaitingAtStation	Nombre de locomotive qui attendent à la gare la dernière à arriver.

Synchro()

Le constructeur prend désormais un paramètre `nbLoco` qui permet de savoir combien de locomotive il y a sur la maquette.

stopAtStation()

Elle prend paramètre la locomotive qui doit s'arrêter à la gare.

La première chose que fait la fonction est d'arrêter la locomotive concernée puis de lock `mutex` afin de ne faire qu'une vérification pour chaque locomotive à la fois. Elle vérifie ensuite si la locomotive est la dernière que l'on attend à la gare grâce à `nbLoco`:

- Si non la locomotive incrémente `nbWaitingAtStation` et unlock `mutex` afin de se faire bloquer par `waitingStation` qui fait office de queue.
- Si oui le numéro de la locomotive va être mis dans `lastArrivedNum` afin de le réutiliser dans la section partagée et va faire appel à `releaseStation()`. Elle fini par unlock `mutex`.

Ensuite toutes les locomotives attendent 5 secondes à la gare afin que les voyageurs puisse faire le transit et redémarre en même temps.

On indique les locomotive qui doivent attendre, la dernière locomotive et le départ par des messages.

releaseStation()

Dans cette fonction privée de *Synchro.h* on release `waitingStation nbWaitingAtStation` fois (On libère toutes les locomotives attendant à la gare).

access()

Elle prend en paramètre la locomotive qui essaie d'accéder a la section partagée.

Pour cette fonction il y a deux cas possible:

1. La locomotive a la priorité car elle était la dernière arrivée à la station (`loco.numero() == lastArrivedNum`).
2. La locomotive doit céder la priorité au cas 1 ou a une autre locomotive qui se trouve deja dans la section partagée.

Pour le cas 1:

La locomotive lock `mutex` et met ensuite `isInSharedSection` à true. Vu qu'elle est la dernière locomotive arrivée en gare, elle ne rentre pas dans le `if()`, ni utilise `fifo`. Elle indique finalement qu'elle se trouve dans la section partagée par un message.

Pour le cas 2:

La locomotive lock `fifo` afin d'être sur qu'une fois la section libérée elle sera la première à rentrer dedans et ensuite lock `mutex`. Ensuite elle vérifie si la section partagée est occupée avec `isInSharedSection` si oui elle rentre dans le `if()` sinon elle continue comme dans le cas 1 car elle a la priorité (elle unlock juste `fifo` en plus avant de sortir de notre fonction). Si elle est rentrée dans le `if()` elle va incrémenter `nbWaiting`, unlock `mutex` et s'arrêter. Elle sera ensuite bloquée par `waitingSection` jusqu'a ce que la locomotive dans la section partagée sorte de celle-ci. Une fois libérée elle continue son chemin en relockant `mutx` et en décrémentant `nbWaiting` puis comme dans le cas 1 (en unlock `fifo`).

leave()

Elle prend en paramètre la locomotive concernée par la sortie de la section partagée.

La locomotive lock `mutex` afin de mettre `isInSharedSection` à false et de libérer une locomotive en attente (s'il y en a une). Finalement elle unlock `mutex`.

On indique également la libération de loco et la sortie de de section partagée par des messages.

Tests effectués

Nous avons vérifié grâce au messages envoyés dans la console générales et des locomotives et visuellement que:

1. L'emergency_stop empêche bien toutes les locomotives de redémarrer ou de bouger après son appui (Visuel).
2. Il n'y a pas de deadlock qui empêche les locomotive de rentrer/sortir de la section partagée et de la section (Visuel + messages).
3. Si une locomotive arrive à la section partagée au même moment qu'elle se libère qu'elle ne fasse que de ralentir brièvement sans s'arrêter(Visuel + messages).
4. La locomotive qui arrive en dernier à la station soit bien celle qui a la priorité à la section partagée (Visuel).
5. Une seule locomotive peut avoir accès à la section partagée à la fois (Visuel + messages).
6. Les deux locomotives s'attendent à la gare et redémarre 5 secondes après l'arrivée de la dernière (Visuel + messages).
7. Il n'y a pas de cas où une locomotive s'arrête/démarre brusquement (Visuel).