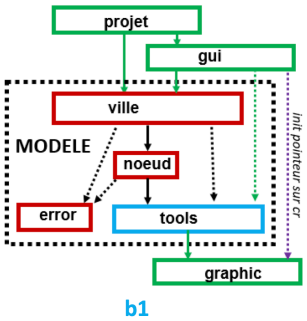
Rapport - BIENAIME Natacha #315948 & HANGARD Kélyan #312936

Architecture logicielle et description de l'implémentation :



Dessin de l'architecture logicielle choisie

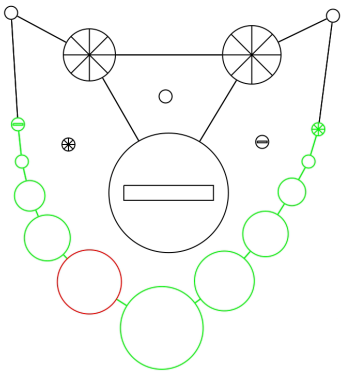
Structuration des données pour représenter le graphe de la ville: Dans le *unnamed space* du module *ville* nous avons un *vector* de pointeurs sur chaque instance de la classe *Quartier,* appelé *tab,* mémorisant l'ensemble des noeuds courants de la ville. Chaque noeud est crée avec le constructeur de *Quartier* et possède donc chacun des attributs de la classe *Quartier*.

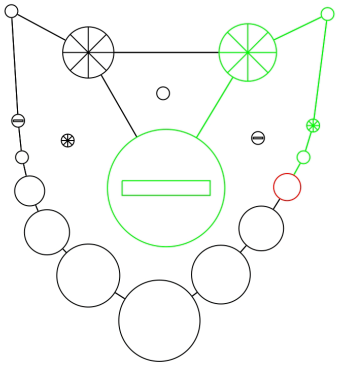
Modules du modèle	Tâches réalisées
ville	**Initialisation de la ville :** procède à la lecture puis au décodage du fichier **Dessin de la ville** : grâce à la fonction *draw city* qui parcourt le *tab* pour lancer le dessin de chaque quartier et chaque connexion. Permet aussi le dessin du *shortest path.* **Modification du modèle:** peut réinitialiser le noeud courant ou la ville entière (ajout ou suppression de quartiers) grâce aux modules *tools* et *noeud.* Il a également recours au module *noeud* lors de tout déplacement de quartier, changement de taille d'un quartier, modification du noeud courant, ajout et suppression de liens. **Calcul les critères:** chaque critère (CI, ENJ et MTA) est calculé grâce à une méthode de la classe *Ville*, ils sont réactualisés après chaque modification qui est apportée par l'utilisateur sur la ville. **Exportation des données du modèle:** notamment les informations importantes (valeurs des critères, le succès de lecture de la ville courante ou le fait qu'un noeud courant soit désigné) vers le module *gui*. **Etat courant de la ville:** Réactualise le *tab* après modification apportée sur la ville flottante par l'utilisateur et enregistre également l'état actuel de la ville avec sa fonction *save.*
noeud	**Initialisation de la ville:** Procède aux vérifications de tout quartier ou lien venant à être ajouté. Signale une erreur potentielle grâce aux fonctions du module *error*. Ajoute un pointeur vers le Quartier au *tab* si ce quartier passe les tests de vérification. Contient la **classe *Quartier*** comportant les caractéristiques de chaque noeud dont ses connexions. **Dessin de la ville**: Permet que chaque quartier soit dessiné en précisant au module *tools* ses attributs caractéristiques (type, taille, noeud courant, etc…). **Modification du modèle**: Les méthodes de vérification du module *noeud* assurent que les modifications apportées à la ville ne créent pas de nouvelles erreurs.
tools	**Initialisation de la ville:** Contient les formes et fonctions géométriques de base ainsi que les fonctions analysant s'il y a des collisions. **Dessin de la ville**: commande les détails du dessin (au module *graphic* situé hors du modèle) quand il y a besoin de dessiner un noeud ou une connexion.
error	Module fourni permettant d'afficher les potentiels messages d'erreurs détectés à la lecture d'un fichier.

Algorithme de Dijkstra:
Afin que notre algorithme de Dijkstra, correspondant à la fonction *algo_dijkstra* (méthode de la classe *Ville),* soit utilisable tant pour trouver le noeud production ou le noeud transport le plus proche du noeud de départ, il a été décidé de fournir en Input l'*uid* d du noeud de départ ainsi que le **caractère** indiquant le *type* du noeud final souhaité. Notre algorithme sort en *output* soit l'*uid* du noeud production soit celui du noeud

transport le plus proche. Notre tableau trié selon les valeurs *access* se nommant *uids_tries*, est au début initialisé avec tous les *uids* des noeuds pointés par le *tab.* Une fonction *tri_selon_access* appartenant au module *noeud* est appelé dès qu'il faut réordonnée *uids_tries* selon les valeurs *access.* On a interprété le « Tant que TN est non-vide » grâce à une boucle *while* vérifiant si *still_in* est à *true.* Ce booléen est en effet d'abord initialisé à *false* puis s'il s'avère qu'un des quartiers a son *in* à *true,* le *still-in* devient *true,* nous permettant ensuite d'entrer dans la boucle *while,* boucle correspondant en grand partie au pseudocode fourni mais adapté au *type* du noeud final. On a cependant décidé d'établir une fonction *check_liens_in* appartenant à la classe *Quartier* qui va pour chaque lien (ayant encore son *in* à *true*) du quartier d'*uid* n , mettre à jour la valeur *access* et la valeur *parent* si un chemin plus court que le précédent a été trouvé tout en vérifiant que l'on ne passe pas par un noeud production. Après, on vérifie si notre *tab* pointe encore sur un Quartier ayant un *in* à *true* et l'on adapte le booléen *still_in* en conséquence afin de continuer ou non la recherche du noeud final.

Dessins du plus court chemin





On retrouve ici les chemins les plus courts vers un noeud production ainsi que vers un noeud transport. Le noeud production n'est pas traversé ce qui est cohérent. De plus, ces chemins sont logiques car visibles à l'oeil nu.

Méthodologie et conclusion :

Après avoir entamé un **travail collaboratif** pour comprendre le fonctionnement du modèle, nous avons commencé par coder le module *tools* nécessaire pour l'abstraction géométrique. Kélyan a ensuite avancé celui-ci en travaillant sur les contraintes géométriques à respecter à l'aide d'un module de tests indépendant. Il a élaboré le module *projet* et entamé le module *ville* avec les fonctions *lecture* et *décodage* puis a effectué certaines méthodes de vérification du module *noeud.* Natacha s'est, quant à elle, concentrée sur les modules *noeud* et *ville*, a élaboré les caractéristiques communs à chaque quartier et a veillé à établir une **encapsulation** et à ce que l'on respecte les conventions de programmation. Les **fichiers tests fournis** nous ont particulièrement été utiles pour vérifier que notre programme marchait correctement.
Kélyan s'est par la suite chargé de l'**interface utilisateur** en élaborant le module *gui* et *graphic* tandis que Natacha s'est occupée du **calcul des critères**. Nous avons travaillé chacun de notre côté, Natacha a travaillé sur les critères grâce au modèle pour faciliter les tests. Kélyan a implémenté les caractéristiques propres à la fenêtre du projet (boutons, taille) en procédant à des tests via les fonctions *stubs* associées aux boutons. Le module *graphic* et la fonction de changement de coordonnées du module *gui* ont été testés via leur utilisation par des fonctions du *tools.* Avec du **recul** cependant il aurait été convenable de tester davantage notre programme avec des fichiers tests **vides**. En effet, on n'y a pas pensé tout de suite et cela aurait pu nous éviter bien des maux. Pour le rendu 3, chacun a réalisé quelques fonctions ensuite testées directement via un **affichage** sur la fenêtre pour vérifier la cohérence de notre code.
Notre **bug le plus fréquent** a été le fameux ***segmentation fault*** que l'on a souvent retrouvé lors de l'élaboration des rendus 2 et 3. On a su procéder à un **scaffolding** nous permettant d'en retrouver l'origine à chaque fois. Pour le rendu 2, on a passé beaucoup de temps à trouver l'origine de ce problème avant de remarquer que lors de l'appel de notre fonction *set_link*, nous allouions dynamiquement un nouvel emplacement pour un *Quartier* ayant déjà une place mémoire attribuée. Cela ne nous a posé aucun problème pour le rendu 1, cependant cela a compliqué le rendu 2.

✘ Avant: *void set_link(Quartier& q) { link.push_back(new Quartier(q)) ; }*
✔ Après: *void set_link(Quartier* pt_quartier) { link.push_back(pt_quartier) ; }*

On a aussi eu quelques **problèmes de coordination** dûs à un décalage horaire de six heures qui nous sépare. Cependant, on a su les surmonter en nous répartissant bien les diverses tâches à réaliser par les modules afin de pouvoir travailler indépendamment. D'après nous, un des **points forts** de notre programme est qu'il n'y a qu'un seul dessin de la ville effectué pour chaque fichier et ce, grâce à la fonction *draw_city* de *on_draw,* ce qui le rend plus rapide lors de l'exécution. En effet, dès que l'utilisateur interagit, ce sont seulement des données propres au modèle qui sont modifiées. En outre, on a développé la capacité à comprendre rapidement ce qui était demandé tout en ayant peu de fautes de compilation. Notre **point faible** principal était le fait de ne pas assez anticiper les futures actions à réaliser. En effet, nous rencontrions principalement des erreurs qui se trouvaient dans une fonction que l'on réutilisait. On passait ainsi beaucoup de temps à chercher souvent une erreur simple que l'on aurait pu anticiper, il faudra donc améliorer cela à l'avenir.