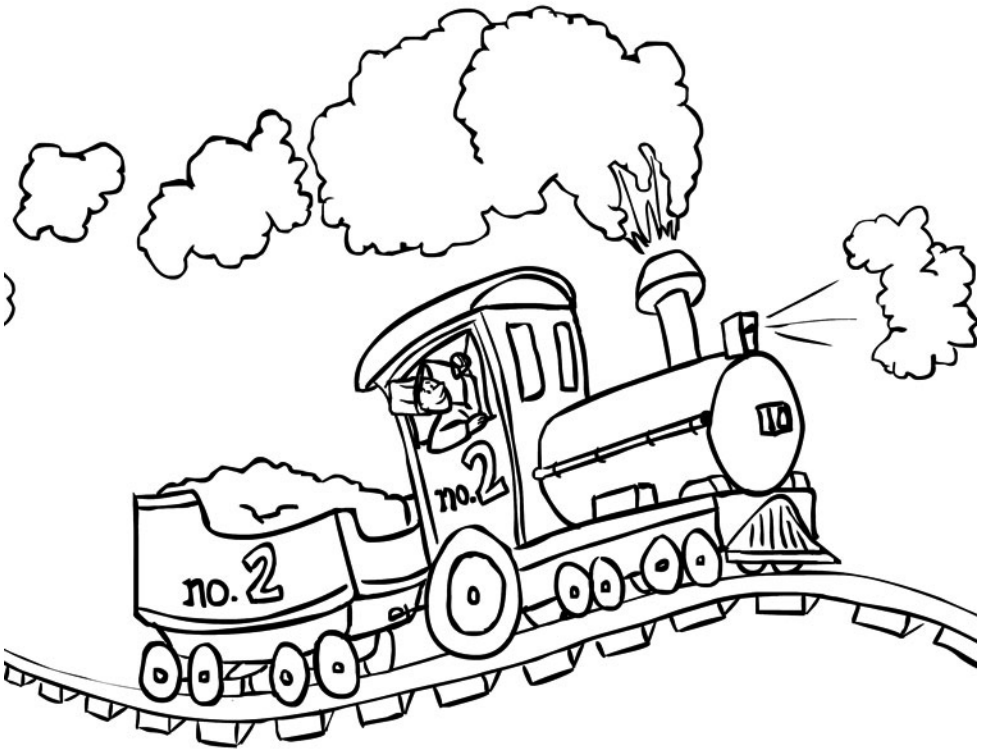
ECOLE NATIONALE
DES SCIENCES
GEOGRAPHIQUES

Rapport Projet OpenGL



IT2
Biadi Hamza
Briend Camille
Couppé Elias

I. Environnement de développement

L'ensemble du projet a été développé sous Linux en utilisant « Qt Creator » pour des raison de simplicité et surtout pour avoir la possibilité d'ajouter une IHM même si cela n'a pas été réalisé finalement par manque de temps. Pour ajouter des trains ou pour modifier leur vitesse, il suffit le fichier main.cpp à la ligne 70.

II. Graphe

Les documents associés au graphe étaient fournis avec le sujet du projet. Il s'agit de :

- une table SIF dans laquelle se trouve la liste des arcs ainsi que leur sommet initial et final correspondant
- une table PAXYZ contenant la liste des points annexes associés à chaque arc
- une table AXYZ contenant les coordonnées XYZ de tous les points annexes.
- une table SXYZ contenant les coordonnées XYZ de tous les sommets du graphe.

A l'aide de l'API fournie avec le projet, elles seront chargées puis lues afin de générer le graphe tridimensionnel sur lequel les mobiles se déplaceront.

II. Trains

a) Déplacement de la sphère

Nous avons choisi d'utiliser une sphère pour faciliter les premiers calculs de déplacement sur le graphe. Afin de placer cette sphère sur le graphe, il est nécessaire de définir des coordonnées, il s'agit de deux angles et d'une distance.

Connaissant les deux points constituant un tronçon, on peut calculer l'orientation du déplacement du mobile et la distance qu'il parcourra avec cette orientation, il s'agit de la longueur du tronçon. Pour déplacer le mobile, on va donc commencer par initialiser à 0 la distance qu'il a parcouru sur ce tronçon, ensuite on incrémentera cette distance d'un pas ΔD en fonction de la vitesse du mobile jusqu'à ce qu'il atteigne la distance maximale : la longueur du tronçon.

$D_n(t) = D_{n-1} + \Delta D$ avec $\Delta D = v*t$, v est la vitesse, t est le temps

Pour dessiner la sphère, on effectue une translation pour se mettre sur le point initial du tronçon, une rotation pour orienter le mobile et finalement une translation pour se déplacer le long du tronçon.

Ce processus est répété à chaque fois que le mobile atteint un point annexe. Au dernier point annexe d'un arc, il choisira aléatoirement le prochain arc qu'il va parcourir. Il s'agira d'un arc dont le sommet initial correspond au sommet final précédent.

b) De la sphère au train articulé

Il s'agit désormais de créer un train d'aspect le plus réaliste possible. On modifie la sphère en plusieurs wagons qui se suivent à courte distance. Le wagon est dessiné et on le modélise sous forme de parallélépipède rectangle.

Extrait du code correspondant :

```
void Wagon::drawWagon()
{
glPushMatrix();
glTranslatef(posIni->getX(),posIni->getY(),posIni->getZ());
glRotatef(90-H,0,0,1);
glRotatef(-V,0,1,0);
glTranslatef(D,0,0);
// wagon
glScalef(1.5,1.0,1.0);
glutSolidCube(0.5);
glPopMatrix();
}
```

Pour dessiner un train, il suffit donc de dessiner tous les wagons un par un.

Extrait du code correspondant :

```
void Train::drawTrain()
{
for(int i=0; i<wagons.size(); i++)
{
wagons[i].drawWagon();
}
}
```

La difficulté réside dans le fait que chaque wagon doit se suivre au niveau des changements d'arcs. On attribue un changement d'arc aléatoire au wagon de tête, tandis que les wagons qui suivent n'auront pour unique choix de prendre le même arc.

Les mobiles peuvent désormais être assimilés à des trains.

c. Gestion des collisions

Plusieurs trains se déplacent sur le graphe. Il s'agit désormais d'éviter qu'ils se touchent, en instaurant et codant une gestion des collisions.

Ainsi, on va créer des méthodes qui vont permettre d'arrêter, de ralentir et d'accélérer les trains. On gère les trains associés à leur vitesse sous la forme de listes.

Extrait du code correspondant :

```
vector<Train> listeTrain; // liste des trains
vector<int> liTarr; // liste des trains arrêtés
vector<float> liVarr; // liste des vitesses des trains arrêtés
vector<int> liTral; // liste des trains ralentis
vector<float> liVral; // liste des vitesses initiales des trains ralentis
```

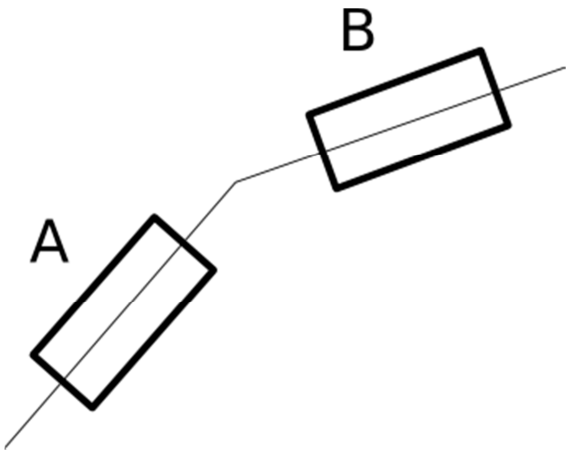
D'autres méthodes sont définies afin de gérer les variations de vitesse :

Extrait du code correspondant :

```
void stopTrain(int IDT);
void raleTrain(int IDT, float _speed);
void demarreTrain(int IDT);
void accelerTrain(int IDT);

void ralentir();
void arreter();

void accelerer();
void demarrer();

bool testSommet(int _IDT); // vrai si le sommet est libre et faux sinon
int testTroncon(int _IDT); // renvoie l'id du train devant et -1 si pas de
train
int testArc(int _IDT); // renvoie l'id du train sur l'autre arc et -1 si
pas de train
```

A l'aide des trois dernières méthodes, on a pu gérer les différents cas de collisions possibles (détaillés plus loin). Les quatre méthodes avant vont, elles, s'occuper la gestion d'arrêt, démarrage, accélération et ralentissement de tous les trains.

1- Deux trains qui se suivent : (le sens de déplacement est de A vers B)



Il existe plusieurs possibilités dans ce cas-là :

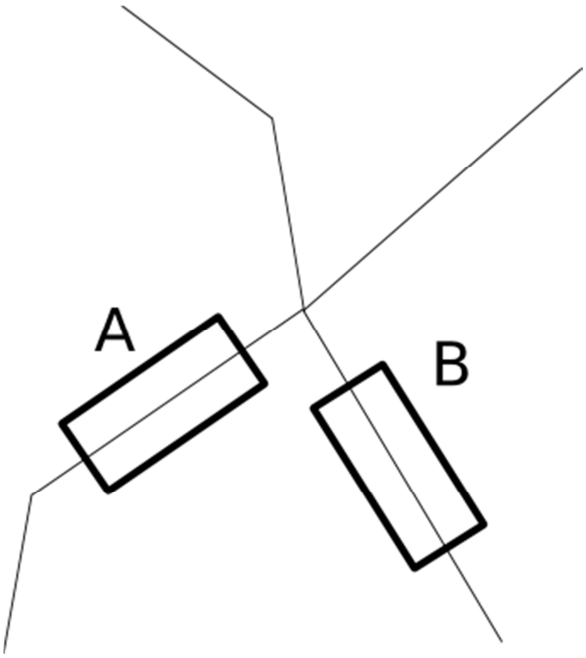
- Le train A est plus rapide que B : on met le train A à la même vitesse que B
- Le train B est plus rapide que A : on ne fait rien

- Le train A est plus rapide que B et les deux trains sont sur deux arcs différents : on vérifie si le train A est sur son dernier tronçon, si B est sur son premier tronçon et met A à la même vitesse que B si c'est le cas

On effectue le changement de vitesse seulement si la distance entre A et B est inférieure à une distance de sécurité car on estime que le train B pourrait, si les deux sont sur le dernier tronçon d'un arc, le temps que A le rattrape, prendre un arc différent que celui de A.

A chaque changement de vitesse, la vitesse initiale est stockée et on remet le train à la même vitesse quand il n'y a plus de train devant lui à vitesse inférieure.

2- Deux trains qui arrivent au même sommet :



Il y a deux possibilités dans ce cas :

- Le train B est toujours sur son dernier tronçon et aucun train ne traverse le sommet : dans ce cas, on compare les vitesses des trains A et B et on laisse passer le plus rapide. Si l'un des deux trains est moins rapide mais il a traversé une bonne partie du tronçon alors on arrête le train à côté même si il est plus rapide. Ceci permet d'éviter l'intersection des trains qui sont proches d'un sommet.
- Le train B traverse le sommet : dans ce cas, on arrête le train A et on attend que le train B passe.

Remarque : Si le train B est en train de traverser le sommet et qu'un autre train C arrive derrière B alors on arrête le train A, le train C sera dans le premier cas (deux trains se suivent). Une fois B traverse le sommet, on revient au cas où deux trains arrivent au même sommet avec aucun train qui le traverse.

On remet au train sa vitesse initiale quand il n'y a aucun train sur le sommet et que ce dernier est prioritaire sur le train, s'il y en a, de l'arc voisin.

Ces différents cas sont testés quand le train dépasse la distance de sécurité fixée.

III. Caméra

La caméra par défaut est une caméra dite « libre », semblable à celle de certains jeux vidéo (caméra libre du jeu « Counter Strike »). Il s'agit de pouvoir se déplacer dans l'espace par des translations à l'aide des touches QSDZ, de prendre de la hauteur en z par rapport au point visé par la caméra par les touches + et -, et de tourner l'axe de caméra à l'aide de la souris.

Pour gérer cette caméra, on définit tout d'abord les coordonnées du sommet de prise de vue par l'attribut « posCam ».

On définit ensuite un vecteur « Left », qui va prendre en charge les translations latérales de la caméra. Ainsi, le vecteur « Left » étant dirigé vers la gauche, on modifiera les coordonnées du sommet de prise de vue positivement à l'appui de la touche Q (déplacement vers la gauche), négativement à l'appui de la touche D (déplacement vers la droite).

Pour gérer l'avancement et le recul de la caméra, on définit le vecteur « Fwd » qui va correspondre à la direction de la vue de la caméra.

Ainsi, encore une fois, on modifiera les coordonnées du sommet de prise de vue positivement à l'appui de la touche Z (déplacement vers l'avant), négativement à l'appui de la touche S (déplacement vers l'arrière).

La caméra libre permet de se déplacer dans tout l'espace, il est donc nécessaire de pouvoir tourner la caméra afin d'avoir une vision à 360°.

On crée donc deux angles phi et téta, que l'on modifie à l'appui du bouton gauche de la souris, en récupérant les coordonnées de la souris au lâché du bouton, en utilisant la même méthode que celle utilisée pendant les TD. On a évité néanmoins une élévation de 90° pour éviter un changement brusque de la caméra : passage d'un plan à l'autre.

On calcule par cela les nouvelles valeurs du vecteur Fwd, qui s'oriente dans la direction du point visé par la caméra.

Ensuite on calcule donc les coordonnées du vecteur «posTar», visé par la caméra. Ces coordonnées sont celles du sommet de la caméra auxquelles on ajoute le vecteur « Fwd » qui varie en fonction du zoom, et des rotations selon les angles phi et téta.

V. Paysage

On définit un temps « mSPF : milli second per frame » de recalcul et de redessin du graphe et du paysage égal à 30.

a) Dessin du graphe

On commence par dessiner le graphe en noir, simplement en récupérant l'ensemble des sommets et points annexes contenus dans les matrices SIF et PAXYZ, et en les liant par une droite.

Extrait du code correspondant :

```
// sommet initial
glVertex3f(graphe.list_sommet[i_SI].X,graphe.list_sommet[i_SI].Y,graphe.li
st_sommet[i_SI].Z);
// points annexes
for(int j=0; j<NPA; j++)
{
int iPA = arc.list_point_annexe.at(j);
float x = graphe.list_point_annexe[iPA].X;
float y = graphe.list_point_annexe[iPA].Y;
float z = graphe.list_point_annexe[iPA].Z;
glVertex3f(x,y,z);
}
// sommet final
glVertex3f(graphe.list_sommet[i_SF].X,graphe.list_sommet[i_SF].Y,graphe.li
st_sommet[i_SF].Z);
glEnd();
```

b) Dessin du ciel

On se penche ensuite sur le dessin du ciel, sous la forme d'une voûte céleste. On dessine donc une sphère qui aura l'ensemble du graphe sous son emprise. On texture ensuite cette sphère à l'aide d'une texture téléchargée gratuitement.

Extrait du code correspondant :

```
// glutSolidSphere(16,100,100);
GLUquadricObj *sphere;
sphere = gluNewQuadric();
gluQuadricTexture(sphere,GL_TRUE);
glBindTexture(GL_TEXTURE_2D,ciel);
gluSphere(sphere,20,100,100);
gluDeleteQuadric(sphere);
glPopMatrix();
glDisable(GL_TEXTURE_2D);
```

c) Dessin du sol et d'objets décoratifs

Nous avons le choix de mettre de l'herbe au niveau du sol. Pour cela une autre texture a été téléchargée. On définit une zone rectangulaire à laquelle on va appliquer cette texture. Cette zone englobe l'ensemble du dessin. Le fait d'avoir un plan intersecté par une sphère donne l'impression, grâce à l'effet de perspective, qu'on est sur une sphère,

Extrait du code correspondant :

```
// herbe
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, herbe);
glBegin(GL_QUADS);
glTexCoord2d(-10,-10); glVertex3d(-10,-10,0); glColor3ub(255,255,255);
glTexCoord2d(40,-10); glVertex3d(40,-10,0); glColor3ub(255,255,255);
glTexCoord2d(40,40); glVertex3d(40,40,0); glColor3ub(255,255,255);
glTexCoord2d(-10,40); glVertex3d(-10,40,0); glColor3ub(255,255,255);
glEnd();
glDisable(GL_TEXTURE_2D);
```

On rajoute également des arbres texturés au niveau du sol.

Remarque : La fonction de chargement des textures a été récupérée sur l'un des tutoriels du site du zéro et nécessite la librairie SDL pour qu'elle fonctionne.