Rapport du projet OpenGL

Kraus Alban, Pensier Philémon, Schittek Gabin

17 mai 2015

Table des matières

1 Introduction

Dans le domaine de la géomatique, la visualisation de données à trois dimensions sur ordinateur est devenue incontournable, et par conséquent les élèves de l'École nationale des sciences géographiques doivent apprendre à manipuler au moins une bibliothèque graphique. C'est dans cet objectif de formation que s'inscrit le cours d'OpenGL, qui est évalué par un projet dont voici le rapport.

Ce projet consiste à représenter un graphe en trois dimensions, et à y faire circuler un ou plusieurs mobiles. Le graphe, supposé non orienté, ainsi que l'interface de programmation permettant de le manipuler, sont des données de ce projet. Le résultat est une application possédant les propriétés suivantes :

1. Représentation tridimensionnelle du graphe
2. Représentation des sommets (sous forme de sphère)
3. et des arcs (dans une autre couleur)
4. Déplacement des mobiles
5. Les mobile sont des trains (ensemble articulé d'au moins deux cubes)
6. Gestion des collisions :
   — Les mobiles doivent s'attendre lorsqu'ils se dirigent vers le même sommet
   — Ils doivent éviter le nez à nez
   — Une distance de sécurité doit etre respectée lorsque deux mobiles occupent successivement les mêmes arcs.
7. Interaction clavier/souris
8. Rapport
9. Tout effort de rendu 3D, et d'ergonomie sera récompensé !

L'ordre de ces propriétés n'était *a priori* pas imposé, mais plutôt conseillé. Cependant, le barême de notation est en faveur d'un tel classement. Nous n'avons pas eu le temps de tout faire : par exemple, la gestion du clavier et de la souris n'a pas du tout été abordée, et l'esthétique est très limitée.

2 Choix de travail

Le but premier de ce projet est de nous faire manipuler la bibliothèque graphique OpenGL. Cette bibliothèque contient des fonctions pour gérer la projection et des primitives graphiques de base. Il existe une version enrichie de la bibliothèque, appelée GLU pour *OpenGL Utility*, contenant des fonctions avancées pour gérer le point de vue et des primitives plus sophistiquées. Ces deux bibliothèques nécessitent cependant un contexte de dessin : il nous faut donc une couche supplémentaire permettant de gérer la fenêtre et les intéractions avec l'utilisateur.

2.1 Le toolkit

Les gestionnaires de fenêtres (tels que le bureau Windows[1], `X` et compagnie, ...) proposent en général une boîte à outils (*toolkit*) permettant de dialoguer avec eux. Celui vu en cours est GLUT (pour *OpenGL Utility Toolkit*), basé apparemment sur `X`, mais il en existe d'autres : certains élèves nous ayant précédés ont pu utiliser la partie de la SDL (*Simple Directmedia Layer*) ou de Qt s'y rapportant.

Nous n'avons guère reconnu l'utilité d'installer une bibliothèque graphique entière pour n'utiliser que ses capacités de gestion de fenêtre. De plus, nous avions déjà vu GLUT en cours. Notre choix s'est donc porté sur cette bibliothèque.

1. `dwm.exe`

Dans notre code, la gestion de la fenêtre est réaliser dans la fonction `GLvoid initialiser_gl()` de `main.cpp`, qui porte mal son nom : en effet, on y initialise davantage GLUT que la partie propre à OpenGL.

2.2 Le langage

OpenGL peut être utilisé dans de nombreux langages : Python, C, C++, OCaml ... Comme l'API graphe est écrite en C++, nous allons programmer en C++.

Néanmoins, ce choix n'est pas idéal : ce langage présente quelques problèmes de portabilité, et le choix du compilateur peut être un enjeu crucial.

2.3 Le compilateur

Tous les membres du groupe de travail ayant des environnements différents, il nous fallait une solution très portable. Du C++ compilé avec GNU `gcc` ou avec MS Visual Studio peut produire des résultats très différents. Nous avons donc décidé d'utiliser le compilateur GNU `gcc`, avec un Makefile rédigé de manière portable par `qmake`, une variante de CMake développée par Qt.

2.4 Choix relatifs à OpenGL

Jusqu'à présent, nous avons davantage parlé de notre environnement de travail que de la programmation en OpenGL, nécessitant pourtant certains choix.

2.4.1 La projection

De manière générale, il y a deux types de projections accessibles en OpenGL : la projection orthométrique (penser à la perspective cavalière) ou la projection à point de fuite (comme notre vision). La première est parfaitement adaptée à la représentation d'objets purement géométriques : elle est facile à utiliser en 2D pour, par exemple, dessiner des graphiques, mais bien moins intuitive en 3D. La deuxième est plus intuitive car elle correspond à nos habitudes de vision ; elle est très utilisée dans les jeux vidéos et la bibliothèque GLU propose des fonctions permettant de faciliter son utilisation.

Outre que, jusqu'à présent, nous n'avons guère abordé les graphes que par une approche mathématique, la perspective orthométrique nous a paru plus simple de prime abord. Nous avons pu définir une pseudo-caméra, et non pas une vraie caméra car cela n'aurait aucun sens dans cette perspective où le point de vue n'influe pas sur la perception des distances (pas de « loin »).

Ce choix fut donc historique ; à la réflexion, nous aurions dû utiliser une perspective conique, qui aurait facilité la gestion du point de vue. Ce choix explique aussi que le contrôle du point de vue au clavier a été négligé.

2.4.2 Distances et vitesses

L'unité de distance utilisée est celle définie par les coordonnées des sommets du graphe fourni, variant sur des entiers entre 0 et 20 environ (0 et 3 pour l'altitude). Le champ de vision (qui n'a pas vraiment de sens en perspective orthométrique) est défini de manière à ce que l'ensemble du graphe tienne dans la fenêtre. Voir le code 1.

Dans ce code, nous comparons la rectangularité du graphe avec celle de la fenêtre, afin de faire toucher la dimension du graphe correspondante. Par exemple, dans ce **if**, on teste si le graphe est moins rectangulaire que la fenêtre, auquel cas il la touchera par les valeurs extrêmes de son axe des ordonnées ; quant aux valeurs selon l'axe des abscisses, elles seront majorées afin d'avoir un repère normé.

On calcule ensuite l'échelle `u` de l'unité du graphe, dans le but de centrer son affichage

Listing 1 – Initialisation de la projection

```
if(ratio_graphe < ratio) {
  glOrtho(pseudo_bas_gauche.X, pseudo_haut_droit.X*ratio/ratio_graphe,
          pseudo_bas_gauche.Y, pseudo_haut_droit.Y, 10, -10);

  GLdouble u = (pseudo_haut_droit.Y - pseudo_bas_gauche.Y) /
               (GLdouble) hauteur_fenetre;

  glTranslated(((GLdouble) largeur_fenetre)/2 *u
               - (pseudo_haut_droit.X - pseudo_bas_gauche.X)/2, 0, 0);

} else {
  ...
}
```

dans la fenêtre (ligne `glTranslate`). À présent, si vous redimensionnez la fenêtre, le graphe gardera les mêmes proportions et sera correctement centré.

Cet extrait de code est situé dans une fonction `initialiser_projection` (`main.cpp`) qui est appelée par GLUT à chaque redimensionnement de la fenêtre.

L'unité de temps est donnée par la fréquence de rafraîchissement de GLUT. La fonction appelée par GLUT lors du rafraîchissement de l'écran s'intitule `dessiner_scene` (`main.cpp`) et se charge à la fois du dessin et de faire avancer les mobiles.

3 Modélisation

3.1 API Graphe

En plus de quelques exercices destinés à préparer ce projet mais ne faisant pas l'objet d'un rendu, une API représentant un graphe était fournie. Les points sont représentés par la classe `CSommet` ou `CPointAnnexe`, qui héritent toutes les deux d'une classe représentant un point 3D (dont les coordonnées sont de type *float*), `CPoint3f`. Les arcs sont représentés par la classe `CArc`, tandis que la classe `CGraphe` représente l'ensemble du graphe. Cette dernière fournit également un certain nombre de méthodes utilitaires. L'API contient aussi d'autres classes utilisées en interne, et à ce titre elles n'ont pas retenu notre attention.

La fonction permettant d'initialiser le graphe était fournie également. Nous n'avons pas cherché à l'altérer, de même que nous avons gardé l'API intacte ; cependant cette fonction avait un comportement propice au débogage, affichant beaucoup d'informations en console, et nous aurions peut-être dû la rendre plus silencieuse.

3.2 Affichage et manipulation du graphe

Les premières exigences du sujet concernent l'affichage du graphe. La classe fournie dans l'API est une représentation fort convenable d'un graphe, et nous n'avons pas eu besoin de développer un autre modèle de données. La fonction affichant le graphe est appelée : `GLvoid dessiner_graphe ( CGraphe &graphe )` et est la première fonction contenue dans `dessin.cpp`.

3.3 La pseudo-caméra

Le fichier `config.h`, qui par ailleurs manifeste des velléités avortées de lister des paramètres de configuration de notre application, définit une classe `Camera`. Cette classe était initialement prévue pour gérer une perspective conique, mais nous avons néanmoins trouvé à l'utiliser en perspective orthométrique.

Elle représente un œil qui se déplace en translation. En perspective orthométrique, déplacer un œil consiste en réalité à déplacer le monde en sens inverse (d'où la ligne `glTranslatef` dans `dessiner_scene`). Les rotations auraient pu être un apport intéressant pour l'ergonomie de notre application, mais étaient compliquées à implémenter, surtout en perspective orthométrique : nous les avons abandonnées.

3.4 Les mobiles

Une autre des exigences du sujet était le déplacement de mobiles le long du graphe, et plus précisément, de plusieurs ensembles de mobiles solidaires. Nous avons donc créé une classe `Train`, composée de `Wagons`.

3.4.1 La classe Wagon

Un wagon possède des caractéristiques géométriques : ses dimensions et sa position. Les dimensions sont un triplet de trois flottants (longueur, largeur, hauteur) dans l'unité du graphe, et ont été codés dans un `CPoint3f`. Étant donné qu'un wagon ne peut circuler que sur son rail, à une seule dimension, sa position linéique (le long du rail) suffit, en supposant que le rail en question peut être positionné.

C'est justement l'objet des attributs suivants : les coordonnées du sommet initial et du sommet final du tronçon de voie qui porte le wagon, ainsi que l'indice de l'arc auquel il appartient.

Enfin, pour enjoliver le rendu, nous pouvons attribuer une couleur à un wagon, codée sur trois entiers entre 0 et 255.

Ces attributs sont principalement utilisés lors du dessin du wagon. Voir à ce propos la section 4.1.2.

3.4.2 La classe Train

Par définition, un train est un ensemble (au sens de `std::vector`) de `Wagons`. Le choix d'un tableau de taille fixe aurait même été parfaitement justifié tant que le nombre de wagons ne varie pas.

Un train occupe de la place dans l'espace, et en pratique on a souvent besoin d'accéder aux arcs et sommets du graphe en relation avec le train. On stocke donc plusieurs attributs :

- le graphe. Une référence aurait été plus appropriée pour éviter la duplication de l'objet, mais nous avions des difficultés à l'initialiser (passer un paramètre obligatoire au constructeur par défaut de `Train`).
- la liste ordonnée des points et des arcs par lesquels passe le train, car les wagons se suivent. Voir 4.2.
- la vitesse actuelle et la vitesse maximale du train (donc de tous les wagons), exprimée en unités de graphe par période de rafraîchissement de l'écran. [2]
- deux attributs pour gérer les priorités aux carrefours, voir 4.3.1.

La classe train possède également une méthode `avancer` et `dessiner` dont les noms parlent d'eux-même. Enfin, en périphérie de cette classe, on peut noter la fonction `construire_trains` où sont paramétrés les trains par défaut.

2. Donc calculée à la louche

3.5 Organisation du code

Le fichier `config.h` définit un certain nombre de constantes de notre application, telles que son nom[3], une macro pour initialiser les paramètres de la caméra, et surtout le chemin relatif ou absolu vers le fichier de données.

Les fichiers `train.cpp`, `train.h`, `wagon.cpp`, `wagon.h` implémentent et définissent les méthodes de la classe `Train` et `Wagon`, respectivement.

Les fichiers `dessin.cpp` et `dessin.h` regroupent les fonctions réalisant l'interface avec les méthodes de dessin et les fonctions gérant les collisions. Le point fédérateur de ces deux catégories de fonctions est la variable semi-globale `std::vector<Train> trains` contenant la liste de tous les trains en circulation.

Enfin, `main.cpp` contient essentiellement des fonctions d'initialisation.

Notons que `tests.h` n'est pas réellement utile : il ne sert qu'à dessiner un petit repère de couleur dans un coin en bas à gauche du graphe. Il pourra être retiré sans scrupules.

Listing 2 – Ordre de dessin

```
GLvoid dessiner_scene(int v) {
    // effacement de la scene
    glClear(GL_COLOR_BUFFER_BIT);

    // reinitialisation de la
    // matrice de vue
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    TEST_DESSIN_REPERE

    // definition de la vue
    glTranslatef(-camera->oeil.X,
                 -camera->oeil.Y,
                 -camera->oeil.Z);

    // dessin
    dessiner_graphe(*graphe);
    deplacer_train();
    dessiner_trains(*graphe);

    glFlush();

    if(v >= 0) {
        glutTimerFunc(v,
            dessiner_scene, v);
    }
}
```

4 Algorithmique

4.1 Dessin

4.1.1 Généralités

L'algorithme de dessin est assez simple. On efface ce qu'il y avait à l'écran, on dessine, et on affiche (`glFlush`). Comme on utilise une fonction de callback de GLUT, il faut relancer le callback à la fin de la fonction, perpétuellement. Le paramètre `v` peut servir à ajuster l'intervalle d'appel ou l'action à effectuer.

Le dessin proprement dit se fait en trois temps. D'abord, on dessine le graphe avec, pour chaque arc, tous ses points annexes et son sommet final. Ensuite, on fait avancer tous les trains. Finalement, on dessine les trains (appel de la fonction de dessin des wagons).

4.1.2 Dessin des wagons

Avant de dessiner un wagon, on le place au bon endroit avec un premier `glTranslate`. Ensuite, on fait tourner le monde afin d'aligner l'axe des abscisses avec la direction du rail. Mais pour ce faire il faut déterminer l'angle et l'axe de cette rotation tridimensionnelle.

On commence par ramener le vecteur donnant la direction du tronçon de voie à l'origine du repère : on nomme ce vecteur `troncon`. Un

3. Un train de Disneyland Paris

Listing 3 – Dessin d'un wagon

```
void Wagon::dessiner() const {
    glPushMatrix();
    glTranslatef(si_courant.X, si_courant.Y, si_courant.Z);
    CPoint3f troncon(sf_courant.X - si_courant.X,
                     sf_courant.Y - si_courant.Y,
                     sf_courant.Z - si_courant.Z);
    if(troncon.Y || troncon.Z) {
        CPoint3f axe(0, -troncon.Z, troncon.Y);
        GLfloat angle = std::acos(troncon.X / norme);
        glRotatef(angle*180/M_PI, axe.X, axe.Y, axe.Z);
    } else if(troncon.X < 0) {
        glRotatef(180, 0, 0, 1);
    }
    glTranslatef(position, 0, 0);
    glColor3ub(rouge, vert, bleu);
    glScalef(dimensions.X, dimensions.Y, dimensions.Z);
    glutSolidCube(1);
    glPopMatrix();
```

vecteur colinéaire à l'axe des abscisses est le cas le plus simple : la seule rotation à calculer est éventuellement une symmétrie centrale dans le cas où le vecteur serait de sens opposé à X (`troncon.X` < 0).

Dans le cas où ces vecteurs ne sont pas colinéaires, il va falloir calculer l'angle et l'axe de la rotation. L'axe s'obtient simplement avec un produit vectoriel :

$$\begin{pmatrix}1\\0\\0\end{pmatrix} \wedge \begin{pmatrix}\texttt{troncon.X}\\\texttt{troncon.Y}\\\texttt{troncon.Z}\end{pmatrix} = \begin{pmatrix}0\\-\texttt{troncon.Z}\\\texttt{troncon.Y}\end{pmatrix}$$

Conformément à ce produit vectoriel, l'angle que nous devons calculer est $(X, \widehat{\texttt{troncon}})$ dont le côté adjacent est `troncon.X` et l'hypoténuse est la longueur du tronçon. L'attribut `norme` et la fonction `calculer_norme` servent pour cela. Il faut noter que le côté opposé est difficilement calculable dans le nouveau repère ; nous utiliserons donc la fonction `acos` :

$$\mathit{angle} = \arccos\left(\frac{\texttt{troncon.X}}{\texttt{norme}}\right)$$

On peut ensuite appliquer la rotation, sans oublier de convertir les radians de la librairie standard en dégrés.

Une fois l'axe des abscisses aligné avec le tronçon de voie, on place le wagon à la bonne position linéaire le long du nouvel axe des abscisses : `glTranslatef`. Après avoir déclaré la couleur du wagon, on le dessine avec `glutSolidCube` qui dessine un cube. Pour avoir un rectangle aux dimensions de notre wagon, il faut préalablement déformer les unités graphiques à l'aide d'un `glScalef`.

Toutes ces modifications du contexte graphique ne sont valables que pour ce wagon ; on les isole donc entre un `glPushMatrix` et un `glPopMatrix`.

4.2 Déplacement des mobiles

4.2.1 Bases du déplacement d'un wagon

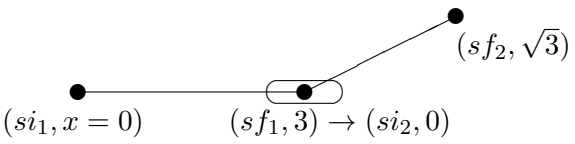
Cette fonctionnalité est réalisée par la fonction `Train::avancer` et les fonctions suivantes dans le fichier `train.cpp`. On itère sur tous les trains puis sur tous les wagons d'un même train. L'opération de base est effectuée à la toute première ligne de cette fonction (listing n° 4). C'est la seule opération pouvant être réalisée indépendamment du reste du train, ce qui explique que les fonctions liées au déplacement se trouvent dans le fichier `train.cpp`.

Listing 4 – Avancer un wagon (base)

```
wagon_courant->position += vitesse;
```

Lorsqu'un wagon arrive à la fin d'un tronçon, sa position et son orientation doivent être mises à jour. Voir la figure 1. Pour ce faire, on remplace le sommet initial du wagon par le sommet final. Il faut néanmoins libérer convenablement le sommet initial précédent (par exemple dans le cas où c'est un carrefour et qu'un autre train attend le feu vert) et calculer le nouveau sommet final. De plus, la position étant linéaire par rapport au sommet initial, on doit en retrancher la norme du tronçon précédent.

FIGURE 1 – Situation lors d'un changement de tronçon : echaînement des sommets initiaux et finaux, et abscisse linéique associée.



La fonction s'occupant de calculer le nouveau tronçon pour ce wagon est `Train::changer_troncon`. Suite à l'appel de cette fonction, on calcule la norme du nouveau troncon sur lequel se déplace le wagon. Ce calcul est réalisé ici et son résultat est stocké, pour ne pas avoir à le faire à chaque fois qu'on dessine le wagon. Pour rappel, la norme d'un vecteur est donnée par :

$$\|\overrightarrow{\mathrm{AB}}\| = \sqrt{(x_\mathrm{B} - x_\mathrm{A})^2 + (y_\mathrm{B} - y_\mathrm{A})^2 + (z_\mathrm{B} - z_\mathrm{A})^2}$$

Lorsque le dernier wagon du train change de tronçon, on retire le sommet initial du tonçon précédent de la liste des points par lesquels passe le train (`etendue`). De même, on retire l'arc correpondant. Si ce point était un carrefour, on indique que ce train a quitté le carrefour en retirant ce point de la liste de carrefours, et on autorise le prochain train à pénétrer dans le carrefour. Voir à ce propos 4.3.1.

4.2.2 Changer de tronçon

Le comportement de cette fonction est totallement différent selon que le wagon objet du changement est le premier wagon (la locomotive, voir 4.2.3) ou un autre wagon. Son but est néanmoins dans les deux cas de trouver le point final du tronçon suivant.

Pour la locomotive, changer de tronçon revient à rechercher dans tout le graphe le tronçon suivant celui sur lequel elle se trouve.

Dans le cas d'un autre wagon, le tronçon suivant est simplement celui emprunté par le wagon précédent. Récupérer ce tronçon nécessite de stocker dans l'ordre la liste des tronçons empruntés successivement par le train.

4.2.3 Trouver le tronçon suivant pour la locomotive

Cette section est implémentée dans la fonction `choisir_troncon`. Dans la plupart des

cas, la locomotive se trouve sur un point annexe d'un des arcs du graphe. Il suffit donc de parcourir la liste des points annexes de cet arc et de prendre le point annexe suivant. On le rajoute ensuite à la liste des points par lesquels passe le train (`etendue`).

Deux cas particuliers peuvent néanmoins subvenir.

La locomotive n'est pas sur un point annexe mais sur un des sommets du graphe. Le tronçon suivant est donc le premier tronçon (terminé par le premier point annexe [4]) d'un autre arc. Le choix de l'arc suivant se fait en parcourant la liste des arcs sortants de ce sommet, et en en choisissant un au hasard. En faisant cela, on considère implicitement que le graphe n'est pas orienté, ce qui est en désaccord avec le sujet mais facilite les calculs : deux trains ne peuvent pas se trouver nez-à-nez. On considère également que tout sommet a au moint un arc sortant (pas de puits), ce qui est, là encore, une hypothèse forte.

La locomotive est sur le dernier point annexe, donc le point suivant est le sommet final de l'arc. Il peut s'agir d'un carrefour, donc la prévention des collisions est indispensable. Voir à ce propos 4.3.1.

Dans tous les cas, on profite de ce moment où la locomotive est sur un point annexe pour vérifier qu'il n'y a pas de train lent devant elle. Voir 4.3.2.

4.3 Gestion des collisions

À partir du moment où plusieurs mobiles se déplacent simultanément, il peut y avoir des collisions entre eux. Ces collisions sont gérées dans le fichier `dessin.cpp`. Plus précisément, il y a deux types de collisions : deux trains s'engagent sur un même carrefour ; ou un train plus rapide rattrappe celui qui le précède. La collision nez-à-nez, lorsque deux trains s'engagent sur le même arc par deux extrémités opposées, a été évitée par hypothèse.

4.3.1 À un carrefour

À chaque fois qu'un train est en approche d'un carrefour (c'est-à-dire sur le dernier point annexe de l'arc y menant), on vérifie d'abord qu'aucun train ne se trouve en plein milieu du carrefour. On considère qu'un train est engagé dans un carrefour lorsqu'un de ses wagons se trouve sur un tronçon se terminant à ce carrefour et que ce train a une vitesse non nulle. Un tel train a la priorité absolue (valeur 0).

En plus d'un train engagé, il peut y avoir un ou plusieurs trains attendant leur tour, c'est-à-dire arrêtés sur un point annexe final. À chacun avait été attribuée une priorité au moment où ils s'étaient arrêtés pour céder le passage ; éventuellement -1 si le train en question vient juste d'arriver au carrefour.

On trie alors la liste des trains en attente de manière à ce que celui qui attend depuis le plus longtemps (priorité positive la plus faible) se trouve en tête de liste.

On parcourt alors cette liste de trains en attente. On attribue un nouvel ordre de priorité aux trains. Si jamais un des trains se voit attribuer la priorité 0, cela signifie que le carrefour est libre et que c'est lui qui est en tête de la file d'attente. On le démarre alors en réglant sa vitesse à sa vitesse maximale. En principe, il n'y a aucun train entre lui et le carrefour, donc il n'est pas utile d'ajuster sa vitesse davantage. Tous les trains se voyant attribuer une priorité strictement positive se voient immobilisés.

4. Ou le sommet final si l'arc n'a pas de point annexe ; cette situation n'a d'ailleurs pas été prévue...

4.3.2 Liées à la vitesse

La fonction `collision_verifier_vitesse` se charge de mettre à jour la vitesse du train passé en paramètre en fonction de celle des trains le précédent sur l'arc. Bien évidemment, si ce train est arrêté à un carrefour, sa vitesse ne doit être modifiée que par la fonction gérant les collisions aux carrefours (4.3.1) : on ne fait donc rien.

La première étape est de parcourir tous les trains en circulation autres que le train sélectionné, et de vérifier pour chacun s'il le précède. On définit qu'un train en précède un autre lorsque le dernier wagon du premier est sur le même arc que la locomotive du deuxième. Néanmoins, cette définition n'est pas rigoureuse car deux trains alignés sur le même arc pourront prendre la même vitesse quelque soit leur ordre.

5 Conclusion

Ce projet est en retard, et n'est pas terminé. Plus rigoureusement, si l'on réexamine les exigences du sujet :

1. Représentation tridimensionnelle du graphe
2. Représentation des sommets (sous forme de sphère)
3. et des arcs (dans une autre couleur)
4. Déplacement des mobiles
5. Les mobile sont des trains (ensemble articulé d'au moins deux cubes)
6. Gestion des collisions :
   - Les mobiles doivent s'attendre lorsqu'ils se dirigent vers le même sommet
   - Ils doivent éviter le nez à nez
   - Une distance de sécurité doit etre respectée lorsque deux mobiles occupent successivement les mêmes arcs.
7. Interaction clavier/souris
8. Rapport
9. Tout effort de rendu 3D, et d'ergonomie sera récompensé !

nous avons correctement traité les 5 premiers points. La gestion des collisions a été réalisée de manière efficace aux carrefours [5]. Déjà en retard à cette période, nous avons simplifié la suite du sujet : les arcs sont orientés pour éviter le nez-à-nez, et nous n'avons pas implémenté de distance de sécurité. L'intéraction clavier/souris a été en partie prévue (grâce à la pseudo-caméra) mais n'a pas été implémentée par manque de temps.

Le rendu 3D pourrait également être amélioré : nous nous sommes contentés de dessiner des wagons de couleur, de taille variable, mais en forme de parallélépipède. On pourrait leur rajouter des textures, ainsi que de réaliser un arrière-plan.

Néanmoins, ce projet nous a permis de comprendre les mécanismes propres à OpenGL, et dans une moindre mesure de développer notre maîtrise du C++.

5. La gestion des longs trains s'étalant sur plusieurs carrefours a été très peu testée

A Manuel d'installation

Deux modes d'installation sont envisageables.

Exécution d'un binaire compilé Cela nécessite les extensions de l'application propres à OpenGL et GLUT (`.dll` ou `.so`). Certaines sont pré-installées sous la plupart des systèmes d'exploitation (par exemple `openGL32.dll` sous Windows), d'autres doivent être téléchargées (sans doute le cas de `glut32.dll` pout Windows). Sous un système GNU/Linux mettant à disposition des paquets pré-configurés, vous devriez pouvoir récupérer des paquets intitulés `libgl1`, `libglu1` et `freeglut3`.

Compilation depuis les sources Cela nécessite d'installer complètement les 3 bibliothèques OpenGL, GLU, GLUT, un compilateur C++, et de préférence GNU `make`.

Les sections suivantes détaillent la compilation depuis les sources.

Notez dans tous les cas qu'OpenGL ne fonctionne qu'avec des cartes graphiques compatibles.

A.1 Les bibliothèques

Nous vous enjoignons à vous rendre sur www.opengl.org/wiki/Getting_started, qui fournit toutes les informations pour télécharger OpenGL.

Vous devez également installer GLUT, par exemple distribué par freeglut. Les instructions d'installation pour des systèmes autres que Unix/Linux semblent lacunaires.

Sous GNU/Linux, il vous suffit d'installer la version de développement de freeglut, sûrement enpaquetée par votre distribution sous un nom semblable à `freeglut3-dev`.

Nous allons essayer de fournir la librairie OpenGL et GLUT déjà compilée pour Windows, mais nous ne pouvons le garantir à l'heure où nous écrivons ces lignes.

A.2 Le compilateur

Les compilateurs n'étant pas interopérables, nous vous recommandons très vivement d'utiliser comme nous GNU `GCC` ou sa version pour Windows `mingw`[6].

Le fichier `bin/*/Makefile` indique les commandes et les options à utiliser pour compiler le programme.

En pratique, vous aurez plutôt envie d'utiliser GNU Make (ou `mingw32-make` pour Windows) pour lire le Makefile et réaliser les bonnes opérations. Vous n'aurez plus qu'à lancer :

Listing 5 – Compiler le programme

```
bin/*/ $ make install
```

A.3 L'environnement de développement

Pour votre information, nous avons développé le projet sous Qt Creator dont nous livrons également notre `.pro`.

6. À installer dans un répertoire sans espaces... Donc pas `C:\Program Files` !