Info-SUP : A2, Promo 2007

ECSTASY OF AGONY

Troisième Soutenance

QUADRAT Quentin, quadra_q, A2
KADIRI Anass, kadiri_a, A2

ECSTASY OF AGONY

QUADRAT QUENTIN,
KADIRI ANASS

Table des matières

1. Introduction

Ce projet comprend deux parties. La première partie a pour but de simuler le plus précisement possible la dynamique d'une voiture. La deuxième a pour but de représenter une ville de type américaine animée par la circulation automobile.

Lors de la première soutenance, notre voiture (modélisée en deux dimensions) était constituée d'une masse ponctuelle à laquelle était accrochée une roue par un ressort. Pour la deuxième soutenance, elle était modélisée par une barre de masse ponctuelle, placée en son centre, aux extrémités de laquelle étaient accrochées deux roues par des ressorts. Les roues possédaient des pneus ayant un comportement élastique. Seul le mouvement vertical de la voiture était visualisé. Pour la troisième soutenance, on passe de la théorie à la pratique : une voiture commandée par le joueur se déplace dans la ville en 3D. On peut voir le fonctionnement de la suspension lors des changements de pentes des routes.

Lors de la deuxième soutenance, la srtructure de la ville constituée de blocs, eux-mêmes constitués de batiments (tirés au hasard) et de routes a été définie. Le travail de la troisième soutenance a permis d'achever l'infrastructure de la ville : – textures de différents routes, – amélioration de la structure des immeubles (dessins, positionnements, tailles), – ajout du fleuve et des ponts, – ajout de la signalisation des routes, – ajout d'effets de lumières (brume, éclairage).

Grâce au traducteur 3D Studio Max vers OpenGL, réalisé pour la première soutenance, il est facile de créer les objets élementaires du jeu (voitures et batiments).

Travaux réalisés pour les trois premières soutenances :

Première soutenance
Apprentissage de Delphi, Pascal
OpenGL (Caméra, primitives, texturage)
Loader ASE
Dynamique
Deuxième soutenance
Dynamique des véhicules
Création du terrain et de la ville
Déplacement des voitures sur une route
Menu en Delphi
Troisième soutenance
Dynamique des véhicules
Automatisation du chargt des voits
Caméra et Frustum
Amélioration de la structure de la ville
Skybox, ponts, eau, immeubles
Blending, fog, lumières
Sons, vidéo d'introduction
format des textures (tga, bmp, jpeg)
Système de particules

2. Création d'une ville

2.1. **Structure de la ville.** Notre ville est de type américaine, c'est à dire qu'elle est constituée de blocs réguliers séparés par des routes nord-sud et est-ouest.

Chaque bloc de la ville (figure 1) est composé de :

- un bloc de maisons tirées au hasard dans une bibliothèque — réalisée dans 3DSMax et importée grâce au traducteur ;
- deux routes qui ne sont pas horizontales ;
- un carrefour horizontal avec une altitude et sa signalisation.

Un bloc de maison est composé, en fait, de quatre alignements de maisons de tailles différentes autour d'une cour intérieure. La taille de tous les blocs sont identiques comme dans les villes américaines.

La ville est un objet Pascal (une matrice d'objets de type bloc). Un bloc est un objet qui contient sa taille, sa position, ses deux routes, son carrefour et quatre tableaux de maisons — correspondant chacun à une rangée — (figure 1).

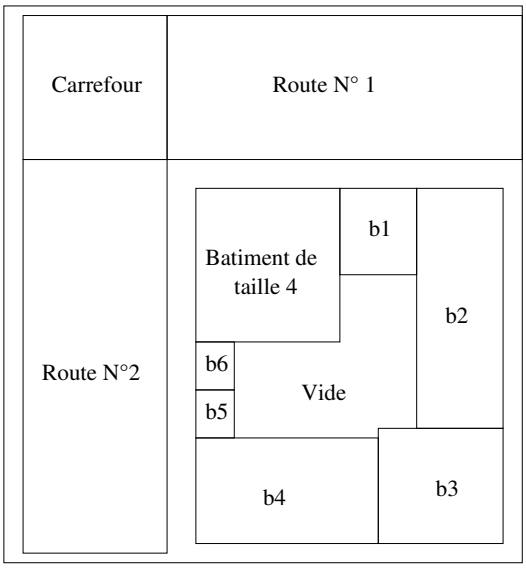

Fig. 1. Modèle d'un bloc de la ville

Une maison est un objet qui contient sa taille, sa position et un numéro d'indentification d'une maison importée de 3D studio Max.

A chacun de ces objets sont associés deux procédures : la première sert à créer sa liste d'affichage, la seconde à l'afficher avec OpenGL.

Une route est un enregistrement qui contient les quatre sommets. Le type carrefour est constitué d'un type route et de quatre feux tricolores. Des fonctions synchronisent les feux et affichent des textures colorées (rouge, jaune ou verte). Ces textures sont particulières puisqu'elles utilisent la transparence d'OpenGL (confére section 6).

Cette structure régulière est trés utile pour l'optimisation de la vitesse du jeu. On peut savoir facilement la case dans laquelle se trouve le joueur. Il

suffit de faire la division entière de la position de la voiture par la taille du bloc pour obtenir le numéro de la case.

Pour tester les collisions du joueur avec les immeubles, il suffit simplement de tester l'appartenance de la voiture au rectangle, formé par les quatres rangées de batiments, du bloc dans lequel se trouve la voiture, plutôt que de tester l'appartenance à un immeuble quelconque de la ville entière.

On affiche seulement neuf blocs : le bloc du joueur et les huit blocs connexes. Pour une ville de dix fois dix blocs, il y a environ deux millions de triangles, on divise par dix le nombre de triangles à afficher. De gros gains de temps restent encore possible avec cette méthode. De même, pour la circulation, il suffit d'afficher les voitures qui sont dans ces blocs. Chaque voiture étant constituée de 1500 à 3000 triangles de gros gains de temps pourront être obtenus pour l'affichage de la circulation.

2.2. **Améliorations.** Voici les améliorations réalisées depuis la deuxième soutenance.

Pour la deuxième soutenance, les routes ne possédaient pas de texture. Maintenant les routes ont une texture venant d'une image au format bmp, tga ou jpeg.

Les immeubles étaient représentés par des cubes de tailles entières entre 1 et n. Ces immeubles sont maintenant dessinés avec 3DS et sont de tailles quelconques, dont au moins un est de taille un. Les immeubles sont tirés au hasard dans un sous ensemble de la bibliothéque (choisie en fonction de la place disponible) jusqu'à remplir totalement la rangée (ce qui est possible à cause de la présence obligatoire d'un immeuble de taille un). La ville posséde un canal avec ses ponts et l'eau qui coule.

Dans la ville, la voiture était représentée par un repère, ne pouvait tourner sur elle-même et restait dans le plan horizontal. Actuellement, elle modelisée en 3D par une carcasse et quatre roues indépendantes et une dynamique représentant le tangage de la voiture par un sytème de trois équations différentielles. Le mouvement horizontal de la voiture et commandée par le clavier (vitesse et direction). La simulation d'une chute éventuelle de la voiture dans le canal a été impléementé.

3. Dynamique de la voiture

3.1. **Modélisation du tangage.** Le véhicule est modélisé en 2D, par une carcasse représenté par une barre de demi longueur l et de masse ponctuelle M à laquelle sont accrochées deux roues (de rayon R et de masse m) par des ressorts. On note $u(t)$ la l'altitude du sol, $y(t)$ est l'altitude de la carcasse, $y_1(t)$ et $y_2(t)$ les allongements des deux ressorts, θ le degré d'inclinaison du véhicule. On note g la gravité, θ le degré de penchement du véhicule (figure 2).

Les forces qui sont en jeux sont : la pesanteur des masses (roue et carcasse), la répulsion du sol sur les roues et la force des ressorts.

L'énergie cinétique de la voiture est :

$$\frac{M\dot{y}^2}{2} .$$

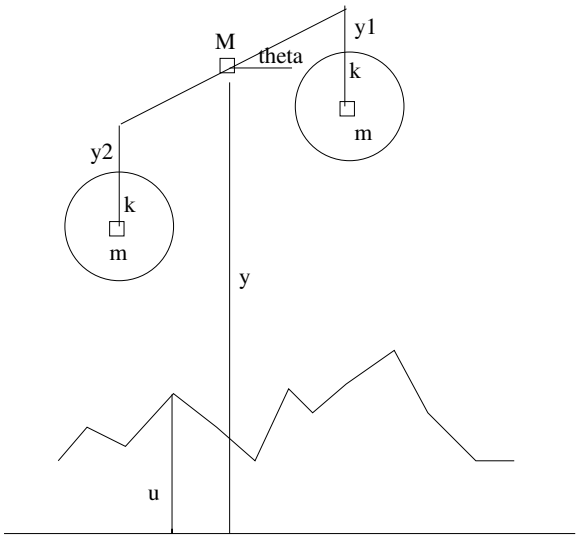

FIG. 2. Modélisation de la voiture

L'énergie potentielle de la voiture est :

$$Mgy \ .$$

L'énergie cinétique verticale de la roue de devant est $(\dot{y_2} + l\cos\theta\dot{\theta} + \dot{y})^2$, que l'on approxime en faisant l'hypothése θ petit par :

$$1/2m(\dot{y_1} + \dot{y} + l\dot{\theta})^2 \ .$$

De même, l'énergie cinétique verticale de la roue de derrière est :

$$1/2m(\dot{y_2} + \dot{y} - l\dot{\theta})^2 \ .$$

L'énergie potentielle due à la pesanteur des deux roues est :

$$mg(2y + y_2 + y_1) \ .$$

L'énergie potentielle du ressort de la roue avant est :

$$1/2ky_1^2 \ .$$

L'énergie potentielle du ressort de la roue arrière est :

$$1/2ky_2^2 \ .$$

L'énergie potentielle de réaction du sol sur la roue de devant est :

$$1/2([u(x + l) - (y_1 + y + l\theta - R)]^+)^2 \ ,$$

où A^+ désigne la partie positive de A.

L'énergie potentielle de réaction du sol sur la roue de derrière est :

$$1/2([u(x - l) - (y_2 + y - l\theta - R)]^+)^2 \ .$$

L'action à minimiser vaut donc :

$$\begin{aligned}\mathcal{A} = 1/2 \int \big\{ & M\dot{y}^2 + m(\dot{y}_1 + \dot{y} + l\dot{\theta})^2 + m(\dot{y}_2 + \dot{y} - l\dot{\theta})^2 \\ & - ky_1^2 - ky_2^2 - 2Mgy - 2mg(2y + y_2 + y_1) \\ & - ([u(x+l) - (y_1 + y + l\theta - R)]^+)^2 \\ & - ([u(x-l) - (y_2 + y - l\theta - R)]^+)^2 \big\} dt\end{aligned}$$

On trouve un système d'équation différentielle où les inconnues sont : trois altitudes (une pour la carcasse, une pour chaque roue) et enfin l'inclinaison de la carcasse (θ).

On note :

$$R_1 = [u(x+l) - (y_1 + y + l\theta - R)]^+ ,$$

$$R_2 = [u(x-l) - (y_2 + y - l\theta - R)]^+ ,$$

On a :

$$(M+2m)\ddot{y} + m\ddot{y}_1 + m\ddot{y}_2 = -g(2m+M) + R_1 + R_2 , \tag{1}$$

$$m(\ddot{y}_1 + \ddot{y} + l\ddot{\theta}) = -ky_1 - gm + R_1 , \tag{2}$$

$$m(\ddot{y}_2 + \ddot{y} - l\ddot{\theta}) = -ky_2 - gm + R_2 , \tag{3}$$

$$m(\ddot{y}_1 - \ddot{y}_2 + 2l\ddot{\theta}) = R_1 - R_2 . \tag{4}$$

En faisant (4) plus (3) moins (2) on obtient $0 = k(y_1 - y_2)$ et donc $y_1 = y_2$.
En faisant (1) moins (2) moins (3) on obtient :

$$M\ddot{y} = -gM + 2ky_1 . \tag{5}$$

L'equation (4) donne alors :

$$2ml\ddot{\theta} = R_1 - R_2 . \tag{6}$$

Puis, (2) moins $\frac{m}{M}$(5) moins $\frac{1}{2}$(6) donne :

$$m\ddot{y}_1 = -ky_1(1 + \frac{2m}{M}) + \frac{R_1 + R_2}{2} . \tag{7}$$

Finalement, on obtient, le système algébriquo-diférentiel suivant :

$$\begin{aligned}\ddot{y} &= -g + \frac{2ky_1}{M} , \\ \ddot{y}_1 &= -ky_1(\frac{1}{m} + \frac{2}{M}) + \frac{R_1 + R_2}{2m} , \\ y_2 &= y_1 , \\ \ddot{\theta} &= \frac{R_1 + R_2}{2ml} .\end{aligned}$$

Pour calculer les trajectoires des corps, nous pouvons approximer les équations différentielles par des équations récurrentes, où h désigne le pas de discrétisation en temps :

$$y(t+h) = 2y(t) - y(t-h) + h^2\left(-g + \frac{2ky_1}{M}\right) ,$$
$$y_1(t+h) = 2y_1(t) - y_1(t-h) + h^2\left(-ky_1(\frac{1}{m} + \frac{2}{M}) + \frac{R_1+R_2}{2m}\right) ,$$
$$\theta(t+h) = 2\theta(t) - \theta(t-h) + h^2\left(\frac{R_1+R_2}{2ml}\right) .$$

3.2. **Modélisation de la dynamique horizontale.** Pour modéliser les déplacements du véhicule dans le plan horizontal, on représente le véhicule par une barre de demi longueur l et de masse ponctuelle M à laquelle sont accrochées deux roues de masse m. Dans un repère fixe (Oxy) (voir figure 3), on note :

- $x(t)$ et $y(t)$ la position du centre de gravité de la voiture,
- $\phi(t)$ l'angle de la carcasse de la carcasse avec l'axe (Ox),
- $\psi(t)$ l'angle des roues avec l'axe (Ox).

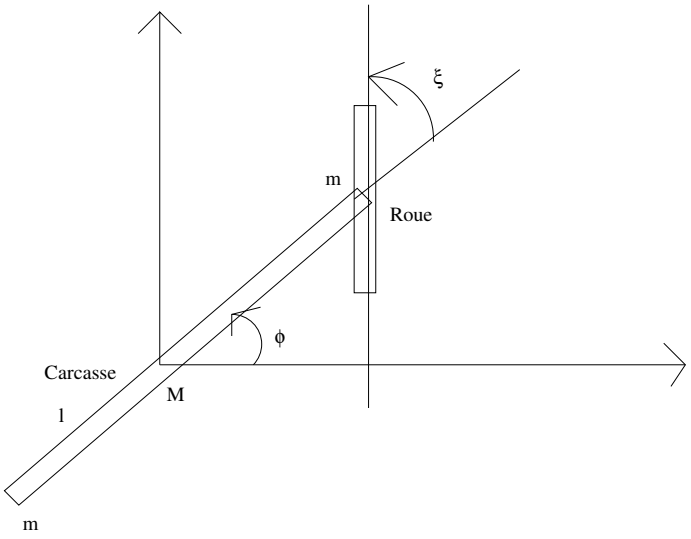

FIG. 3. Modélisation de la voiture (vue de dessus)

On note :

- $a(t)$ accélération donnée par le joueur à la voiture (la puissance du moteur),
- $\xi(t)$ le changement de direction donné par le joueur au véhicule.

La vitesse de rot du véhicule autour de son centre de gravite est proportionnel au chgt de direction ξ

Lors d'un changement de direction, l'élasticité des pneus produit un couple de rotation proportionel à ξ.

En négligeant les frottements de l'air, on obtient les équations du mouvement en appliquant le principe fondamental de la dynamique :

$$(2m+M)\ddot{x} = a\cos\phi ,$$
$$(2m+M)\ddot{y} = a\sin\phi ,$$

$$\dot{\phi} = \xi \ .$$

On peut discrétiser ces équations différentielles :

$$x(t+h) = 2x(t) - x(t-h) + h^2 \left(\frac{a(t)\cos\phi(t)}{2m+M} \right) \ ,$$

$$y(t+h) = 2y(t) - y(t-h) + h^2 \left(\frac{a(t)\sin\phi(t)}{2m+M} \right) \ ,$$

$$\phi(t+h) = \phi(t) + h\xi \ .$$

où h désigne le pas de discrétisation en temps.

3.3. **Autre.** Pour connaitre l'altitude de la voiture en fonction de sa position, on doit savoir sur quelle route se trouve cette voiture. On l'obtient en déterminant le bloc, puis la route N-S ou E-O dans lesquels se trouve la voiture. Ces objets étant rectangulaires, des tests simples résolvent le problème.

Connaissant la route, l'altitude de la route en un point donné s'obtient par interpolation linéaire de l'altitude de ses extremités.

4. Automatisation du chargement des voitures

Si un joueur veut définir lui-même sa propre voiture dans 3DS. Il peut facilement l'importer dans le jeu de la façon suivante :

- créer un nouveau dossier avec le nom de la voiture ;
- dessiner une carcasse de voiture avec 3D Studio Max et l'exporter sous *carcasse.ase* dans le dossier ;
- dessiner également une roue (la gauche) et l'exporter sous *roue.ase* toujours dans le même dossier ;
- prendre une photo de la voiture *photo.jpg* ;
- créer un fichier *info.txt* et mettre les paramètres de la voiture (poids de la roue, de la carcasse, raideur des ressort, position des roues etc) ;
- lancer le programme, sélectionner la nouvelle voiture et jouer ;
- si la voiture ne plait pas, alors la supprimer du dossier.

Remarques :

- on peut ajouter ou supprimer une nouvelle voiture et modifier ses paramètres dans son fichier *info.txt* en cours de jeu ;
- modifier les paramètres de la voiture, sans comprendre ce que l'on fait, peut conduire à des problèmes numériques ; par exemple, si la réaction du sol est trop forte ou le pas en temps est trop grand des instabilités numériques peuvent apparaitre.

La définition d'une voiture nécéssitant plusieurs fichiers (géométrie de la roue, géométrie de la caracasse, paramètres mécaniques et photo), Delphi ne possedant pas de procédures permettant de stocker dans un tableau les noms des dossiers contenus dans un dossiers, il a été nécéssaire de faire une procdure remplissant cette fonction :

```
(1)var sr : TsearchRec;
(2)      Liste : Tliste;

(4)   FindFirst(GetCurrentDir+'*.*', faDirectory, sr);
(5)   while ((FindNext(sr) = 0) and (Liste.long < 10) do
```

```
        begin
            Liste.long := Liste.long+1;
(7)         Liste.elt[Liste.long].Nom := sr.name;
        end;
```

Le type TSearchRec est rempli lors d'un appel aux fonctions FindFirst ou FindNext. En particulier, si un fichier est trouvé, le champ *name* de TSearchRec est rempli par le nom du fichier.

Une Tliste est un enregistrement qui posséde deux champs. Le premier *elt* est un tableau de strings et le deuxième *long* est la position du dernier élément.

La fonction `FindFirst`,

```
function FindFirst( const path : string ;
                    attr       : Integer;
                    var F      : TSearchRec ): Integer;
```

recherche dans le répertoire spécifié par *path* le premier fichier qui correspond au nom de fichier spécifié par *path* et aux attributs spécifiés par le paramètre *attr*. Le résultat est renvoyé dans le paramètre *F*. `FindFirst` renvoie 0 si un fichier a été localisé avec succès ; sinon, elle renvoie un code d'erreur Windows.

Le paramètre constant *path* correspond à un répertoire et un masque de fichier qui peut inclure des caractères génériques. Par exemple, $c :\backslash test\backslash *.*$ indique tous les fichiers du répertoire $C :\backslash TEST$).

Outre les fichiers normaux, le paramètre *attr* indique les fichiers spéciaux à prendre en compte. Par exemple : faReadOnly pour les fichiers en lecture seule ou faDirectory pour les fichiers répertoire, etc. Il est a noté que la fonction GetCurrentDir retourne le chemin courant. Par défaut, elle retourne le chemin où se trouve l'exécutable.

La fonction `FindNext`,

```
function FindNext( var F : TSearchRec ): Integer;
```

renvoie l'entrée suivante correspondant au nom de fichier et à l'ensemble des attributs préalablement définis lors d'un appel à la fonction FindFirst. L'enregistrement de recherche doit être identique à celui transmis à la fonction FindFirst. La valeur renvoyée est 0 si l'exécution de la fonction à réussi. Sinon, cette valeur est un code d'erreur Windows.

A la ligne (6), on ajoute le nom du dossier dans la dernière case du tableau. Remarque : le premier élément du tableau est la chaine "..", s'il existe un dossier parent.

5. Les lumières

5.1. **Sources lumineuses.** La lumière émise par une source est formée de trois composantes : la plus importante, la composante diffuse, est réfléchie par un objet dans toutes les directions. La composante spéculaire correspond à la lumiere qui est réfléchie dans une direction privilégiée (et qui est donc à l'origine de l'effet de brillance). La composante ambiante d'une scène est une lumière non directionnelle, que l'on peut considérer comme issue des multiples réflexions de rayons lumineux.

Pour des raisons d'efficacités, OpenGL ne calcule pas la couleur de chaque pixel d'un polygone : soit il remplit chaque polygone avec une couleur unie (model de remplissage *Flat*), soit il utilise un algorithme de Gouraud (mode *Smooth*), dont le principe est le suivant : pour un polygone donné, la couleur de chacun des sommets est calculée et le polygone est rempli avec un dégradé entre ces differentes couleurs. Pour calculer correctement la réflexion des rayons lumineux en un point OpenGL a besoin de connaître la perpendiculaire à la surface de l'objet au point donné (c'est la normale).

5.2. **OpenGl.** La phase d'initialisation de l'éclairage commence par la spécification du mode de remplissage des polygones avec :

```
glShadeModel(GL_SMOOTH);
```

Ensuite, on indique à OpenGL qu'on souhaite utiliser le calcul d'éclairage, en activant la variable d'état `GL_LIGHTING` :

```
glEnable(GL_LIGHTING);
```

OpenGL permet d'utiliser jusqu'à huit sources de lumière. Ces huit lampes sont indexées par les constantes `GL_LIGHT0` à `GL_LIGHT7`. Il faut activer chacunes des sources qu'on souhaite utiliser (une dans notre cas) :

```
glEnable(GL_LIGHT0);
```

Vient ensuite le paramètrage des lampes. Il se fait avec une unique fonction, `glLightfv()`, dont le prototype est le suivant :

```
procedure glLightfv( lampe    : Glenum;
                     nomparam : Glenum;
                     param    : GLType );
```

lampe désigne la lampe dont on veut modifier une propriété. *nomparam* est le nom du paramètre à modifier. Il s'agit d'une des dix propriétés de source lumineuses (`GL_DIFFUSE`, `GL_AMPBIANTE`, `GL_SPECULAR`, `GL_POSITION`, etc). *param* désigne la valeur à affecter au paramètre choisi (les paramètres sont passés sous forme de tableaux).

5.3. **Paramètres de matériaux.** Dans la réalité, un rayon lumineux est constitué d'un ensemble d'ondes de longueurs différents. A chaque longueur d'onde correspond une couleur. Un objet frappé par un rayon lumineux va absorber certaines longueurs d'onde et réfléchir les autres. C'est le principe des matériaux.

Lorsqu'un polygone est décrit, il se voit affecter le matériau courant. La modification du matériau courant se fait avec la fonction `GlMaterialfv()` :

```
procedure glMaterialfv( face     : Glenum;
                        nomparam : Glenum;
                        param    : GlType );
```

face indique la face (avant ou arrière) dont on souhaite modifier les paramètres. *nomparam* désigne la propriété qu'on souhaite changer, et *param* est un tableau contenant la nouvelle valeur à affecter à *nomparam*. Les valeurs de *nomparam* possibles sont : `GL_AMBIENT`, `GL_DIFFUSE`,`GL_SPECULAR`, `GL_SHININESS` (i.e. coefficient de brillance).

6. La transparence

6.1. **Fusionnement des couleurs.** Le quatrième paramètre de la commande `glColor4f`, appelé *alpha*, sert à spécifier la couleur d'un objet et est utilisé pour spécifier le taux de transparence d'un objet.

Plus la valeur alpha sera forte (le maximum étant de 1) plus la primitive sera opaque. Inversement, plus la valeur sera faible (le minimum est 0) plus la transparence sera forte.

Si on prend un cube ayant la valeur alpha à 1 et que l'on place devant une sphère avec une valeur alpha de 0.15, le cube ne laissera passer aucune couleur (mais ce n'est pas grave car il est situé à l'arrière de la sphère), par contre, la sphère laisse passer 15 pourcent de transparence. Logiquement, si on regarde la sphère, on devrait voir le cube. Mais si le blending d'OpenGL n'est pas activé et que les deux objets sont dessinés l'un par dessus l'autre, il les dessinera tous deux comme étant deux objets opaques. En effet lorsque le deuxième objet est affiché, OpenGL écrase chaque valeur chronomatique du cube pour les remplacer par celle de la sphère. Par contre, si le blending est activé, alors il conserve celles du cube et les accouple à celles de la sphère.

La sphère joue le rôle de la *source*, car ses valeurs chromatiques seront affectées à ceux du cube. Le cube joue le rôle de la *destination* car il est le résultat de l'accouplement de ses valeurs chromatiques avec celles de la sphère.

6.2. **OpenGL.** L'activation de la transparence, se fait grâce à la commande : `glEnable( GL_BLENDING )` et pour la désactiver : `glDisable( GL_BLENDING )`.

Pour voir les différents types d'accouplement entre la source et la destination en utilisant la commande suivante, confére le livre de reférence OpenGL :

```
glBlendFunc(Source : GLenum; Destination :  GLenum);
```

Prenons un exemple (le blending est activé) :

```
glBlendFunc(GL_ONE, GL_ZERO);
```

Si l'on dessine une primitive en utilisant le blending de cette façon, cela n'aura aucun effet de transparence (on obtiendra les mêmes résultats que si le blending était désactivé). Parce que les valeur chromatiques de la source sont multipliées par 1, ce qui ne change rien aux valeurs chromatiques de la source. Par contre les valeurs chromatiques de la destination sont écrasées car elles sont multipliées par 0.

7. La brume

7.1. **Utilité de la brume.** La brume (ou fog en anglais) est utile pour apporter du réalisme aux scènes 3D (par exemple si l'on dessine des montagnes). Sur des gros programmes en 3D, la brume peut améliorer les performances en éliminant les objets dont la distance est trop grande pour être perçue par la caméra.

Pour notre projet, il permet d'éviter l'apparition brutale des immeubles avec le frustum (voir section 8).

7.2. **OpenGL.** Voici l'implémentation de la brume avec OpenGL :

```
(1) FogCouleur : Array[0..3] of GLFloat = (0.5,0.5,0.5,1.0);
(2) glFogf(GL_FOG_MODE, GL_LINEAR );
(3) glFogf(GL_FOG_DENSITY, 0.35 );
(4) glFogi(GL_FOG_START, 3.0);
(5) glFogi( GL_FOG_END, 30.0);
(6) glFogfv( GL_FOG_COLOR,@FogCouleur );
(7) glEnable( GL_FOG );
```

A la ligne (1), le tableau `FogCouleur` contient la couleur de la brume grâce au quadruplet (R,G,B,A) qui désigne respectivement la couleur rouge, verte, bleue et alpha (opacité).

A la ligne (2), on détermine le type de brume que l'on souhaite avoir : plus ou moins épaisse selon la distance (linéaire ou exponentielle) : `GL_LINEAR`, `GL_EXP` ou `GL_EXP2`.

A la ligne (3), on détermine la densité de la brume. La plus petite valeur est 0 et la la plus élevée est 1.

A la ligne (4), on détermine la position où la brume commence à faire effet.

A la ligne (5), on détermine la position où la brume perd son effet et par conséquent où OpenGL arrête de dessiner les primitives car elles sont "perdues dans la brume".

A la ligne (6), on affecte une couleur à la brume, ici la couleur est contenue dans notre tableau `FogCouleur` .

A la ligne (7), on active la brume.

8. Réduction du nombre d'objets dans le cône de vision

Du fait de la régularité de la forme de la ville, on peut réduire le nombre d'objets à considérer dans la projection de visualisation (frustum en anglais). Lors de cette projection, OpenGL considère tous les objets de la scène. Il est donc dans ce cas utile de refaire à la main la projection au lieu de laisser faire OpenGL. On définit les objets que l'on veut projeter et on fait la projection après avoir retrouvé les paramètres de la projection utilisés par OpenGL qui sont les deux matrices : `GL_PROJECTION_MATRIX, GL_MODELVIEW_MATRIX`.

Actuellement, seulement neuf blocs sont considérés dans la projection, ce qui devrait faire gagner un facteur dix dans le calcul de la projection d'une ville de cent blocs. De groses économies sont encore possibles en ne considérant que les immeubles exposés. Les expériences numériques semblent confirmer ces gains.

9. La Vidéo d'Introduction

Pour monter la vidéo d'introduction, il a fallu utiliser un logiciel de montage vidéo intitulée Ulead Video Studio 6. Celui ci nous a permis de compiler plusieurs éléments (sons, titre etc...) à une vidéo téléchargé sur le net (www.jeuxvideo.com) du jeu Burnout 2. Ainsi réalisée, on a dû la jouer au démarrage du jeu, ceci grâce au media player integré de delphi. Celui ci ne jouant la vidéo qu'à sa taille initiale, il a donc fallu l'agrandir pour ainsi la

visualiser sur tout l'écran. Le logiciel Virtual Dub a permis donc de redimensionner cette vidéo en 800 x 600 et AVI2MPG a permis de la compresser en mpeg pour ainsi prendre moins de place.

10. Le moteur de particules

Pour simuler des phénomènes complexes tel que la pluie, le feu ou un jet d'eau, il va falloir utiliser un moteur à particules. Avant cela, quelques bases physiques et mathématiques sont nécessaires.

Pour commencer, un système de particules est un ensemble d'objets élémentaires identiques (les particules) soumis à des lois physiques determinées. Bien souvent, les particules sont émises par des sources et ont une durée de vie limitée. Prenons l'exemple d'une fontaine.

Le principe de la fontaine est de propulser de l'eau à la verticale, vers le haut, celle-ci retombant vers le sol à cause de la gravité terrestre. Il nous faut trouver un modèle physique qui va régir les mouvements de nos particules.

Le point de départ est le théorème du centre d'inertie. L'énoncé de ce théorème est le suivant : "Dans un référentiel galiléen, la somme des forces extérieures appliquées à un solide est égale au produit de sa masse par l'accélération de son centre d'inertie". Il ne reste plus alors qu'à exploiter cette loi pour déterminer le mouvement de nos particules.

Pour commencer, adoptons quelques conventions d'écriture : on note $x(t)$ le vexteur position d'une particule à l'instant t (si O désigne l'origine du repère et si la particule se trouve au point P, le vecteur position de la particule est OP). $v(t)$ et $a(t)$ désigne respectivement les vecteurs vitesse et accélération de la partivcule. La somme des forces auxquelles est soumise la particule à l'instant t est notéê $f(t)$. M désigne la masse de la particule. Avec le théorème du centre d'inertie et la définition des vecteurs vitesse et accélération, nous disposons des données suivantes :

$$f(t) = Ma(t) \ ,$$
$$a(t) = \frac{dv(t)}{dt} \ ,$$
$$v(t) = \frac{dx(t)}{dt} \ .$$

On obtient alors la formule suivante :

$$f(t) = M\frac{dv(t)}{dt} \ .$$

La notion de dérivée est une notion continue. Or notre système d'affichagfe est un système discret avec un nombre determiné d'images par seconde. Il nous faut discrétiser la fonction dérivée en faisnat l'approximation suivante :

$$\frac{v(i+1) - v(i)}{h} \ .$$

avec h le temps écoulé entre l'image i et $i+1$

On a donc à l'instant $i+1$:

$$f(i+1) = \frac{M((v(i+1) - v(i))}{h} \ ,$$

$$v(i+1) = \frac{(x(i+1) - x(i))}{h} .$$

Et par réarrangement, on en tire que :

$$v(i+1) = v(i) + \frac{f(i+1)h}{M} ,$$
$$x(i+1) = x(i) + v(i+1)h .$$

10.1. **Somme des Forces.** Il nous manque une dernière donnée : la somme des forces que subit la particules. Nous négligerons les autres forces qui interviennent dans la réalité (frottements de l'air).

La gravité se traduit par une force nommée poids s'exerçant à la verticale vers le bas. La valeur de cette force est notée P, et elle vaut Mg où g est l'accélération de pesanteur.

Nous allons supposer que le vecteur vitesse intervenant dans la somme des forces à l'instant $i+1$ est le vecteur vitesse à l'instant i et non pas à l'instant $i+1$. Cette astuce qui a peu d'influences sur le résultat visuel, simplifie les calculs. Sans cela, $v(i+1)$ interviendrait dans le calcul de $f(i+1)$ qui doit lui même servir à calculer $v(i+1)$.

10.2. **Récapitulatif.** Voici comment calculer la position d'une particule à l'instant $i+1$ en connaissant sa position et sa vitesse à l'instant i ainsi que l'intervalle de temps h entre i et $i+1$.

$$f(i+1) = v(i) + Mg ,$$
$$v(i+1) = v(i) + \frac{f(i)h}{M} ,$$
$$x(i+1) = x(i)h .$$

10.3. **Le moteur du système.** Une particule est en fait une texture OpenGl. Le moteur du système est divisé en deux parties.

La première sert à l'initialisation du système. C'est cette fonction qui englobe la modélisation de la source émettrice de particules. Les particules sont émises à l'origine et à la verticale vers le haut. Cependant, il est nécessaire d'introduire un minimum d'aléatoire pour perturber le vecteur initiale de chaque particule. En effet si toutes les particules ont le même vecteur initial, elles auront exactement le même comportement, nous verrons qu'une seule particule au final.

La seconde partie du moteur est une procedure qui réactualise la physique que nous avons vu précedemment. Les particules sont réinitialisées lorsqu'elles retombent sur le sol.

11. Le son

11.1. **Initialisation de FMOD.** L'initialisation de fmod s'effectue par la commande suivante :

```
bool lRet;
lRet = FSOUND_Init(44100, 32, 0);
```

Les paramètres sont, dans l'ordre :
– la fréquence de mixage,
– le nombre de canaux,

– les flags d'initialisation.

L'appel à FSOUND_Init() retourne *vrai* si l'initialisation s'est bien deroulée, *faux* sinon. Il est possible d'obtenir une description des erreurs, voir 11.6.

11.2. **Déclaration des variables et structures.** Nous allons déclarer les numéros de canaux pour la lecture des sons et des musiques, une structure `FSOUND_STREAM` pour la musique et des strutures `FSOUND_SAMPLE` pour les bruitages.

```
int int_canal;  // numero de canal pour les bruitages int
int\_canal_mp3; // numero de canal pour la musique
FSOUND_STREAM * my_mp3; // declaration du stream correspondant
                        // a la musique
char * my_musik = "Music.mp3"; // declaration du
                               // nom du fichier de musique
char * my_musik_play = my_musik; // nom du fichier de musique
                                 // qui doit etre jou\'e
                                 // (permet de gerer les
                                 // changements de musique par
                                 // simple affectation)
// bruitages
FSOUND_SAMPLE * sound1, * sound2, * sound3; // d\'eclaration des
sons
```

11.3. **Chargement de la musique et des sons.**

11.3.1. *Chargement de la musique.*

```
if (!(my_mp3 = FSOUND_Stream_OpenFile(my_musik_play,
FSOUND_LOOP_NORMAL, 0))) FMOD_ErrorString(FSOUND_GetError());
```

`FSOUND_Stream_OpenFile()` ouvre un fichier et le prépare pour la lecture. Les paramètres sont dans l'ordre : - le nom du fichier, - le mode spécifiant la façon de jouer le morceau, - 0 à une exception près. La valeur de retour est un pointeur sur une structure `FSOUND_STREAM`, `NULL` si le fichier n'a pas pu être ouvert. Pour fermer ce fichier, il suffira de faire appel à la fonction :

```
FSOUND_Stream_Close()}. bool lRet;
lRet = FSOUND_Stream_Close(MP3Stream);
FSOUND_Stream_Close()
```

ferme le stream passé en argument et libère les ressources allouées lors de l'ouverture. Une valeur de retour à true indique que la fermeture s'est bien deroulée, false sinon.

11.3.2. *Chargement des sons.*

```
sound1 = FSOUND_Sample_Load(FSOUND_FREE, "sounds/1.wav", 0, 0);
sound2 = FSOUND_Sample_Load(FSOUND_FREE, "sounds/2.wav", 0, 0);
sound3 = FSOUND_Sample_Load(FSOUND_FREE, "sounds/3.wav", 0, 0);
```

11.4. **Lecture.**

11.4.1. *Lecture de la musique.*

```
int\_canal\_mp3 = FSOUND\_Stream\_Play(FSOUND\_FREE, my\_mp3);
```

FSOUND_Stream_Play() lance la lecture du Stream passé en second paramètre. Le premier paramètre permet de spécifier le numéro du canal que l'on veut utiliser. FSOUND_FREE prend le premier canal libre. La valeur de retour est le numéro du canal attribué pour la lecture du morceau.

11.4.2. *Lecture des sons.*

```
int_canal = FSOUND_PlaySound(FSOUND_FREE, sound1);
```

11.5. **Arrêter la lecture de la musique.**

```
bool lRet;
lRet = FSOUND_Stream_Stop(my_mp3);
```

`FSOUND_Stream_Stop()` arrête la lecture du stream passé en paramètre. A noter que la lecture est arretée, mais que le Stream n'est pas libéré.

11.6. **Gestion des erreurs.**

```
int iErrCode;
iErrCode = FSOUND\_GetError();
```

Cette fonction renvoie un code d'erreur dont la description est :

- `FMOD_ERR_NONE` Pas d'erreur
- `FMOD_ERR_BUSY` Impossible d'appeler cette commande après `FSOUND_Init`. Appeler `FSOUND_Close` d'abord
- `FMOD_ERR_UNINITIALIZED` Cette command a échoué parce que `FSOUND_Init ou FSOUND_SetOutput` n'ont pas été appelés
- `FMOD_ERR_INIT` Problème à l'initialisation
- `FMOD_ERR_ALLOCATED` Problème à l'initialisation : le système est déjà en cours d'utilisation
- `FMOD_ERR_PLAY` Erreur de lecture
- `FMOD_ERR_OUTPUT_FORMAT` La carte son ne supporte pas ce format (16bit stereo output
- `FMOD_ERR_COOPERATIVELEVEL` Problème de cooperative level for hardware
- `FMOD_ERR_CREATEBUFFER` Erreur de création de buffer
- `FMOD_ERR_FILE_NOTFOUND` Fichier non trouvé
- `FMOD_ERR_FILE_FORMAT` Format de fichier inconnu
- `FMOD_ERR_FILE_BAD` Erreur de chargement du fichier
- `FMOD_ERR_MEMORY` Pas assez de ressources ou de mémoire
- `FMOD_ERR_VERSION` Version du format de ce fichier non supportée
- `FMOD_ERR_INVALID_PARAM` Un paramètre invalide a été passé à cette fonction
- `FMOD_ERR_NO_EAX` A essayé d'utiliser une commande EAX sur un canal inapte
- `FMOD_ERR_CHANNEL_ALLOC` Erreur d'allocation d'un nouveau canal
- `FMOD_ERR_RECORD` L'enregistrement n'est pas supporté sur cette machine

– `FMOD_ERR_MEDIAPLAYER` Windows Media Player n'est pas installé donc impossible de lire le format wma ou d'utiliser le streaming internet
– `FMOD_ERR_CDDEVICE` Erreur apparue lors de la tentative d'ouverture du CD device

12. Page Web

www.epita.fr\∼quadra_q

13. Conclusion

Lors de la quatrième et dernière soutenance :
– on continuera d'améliorer la structure de la ville. Il est probable que l'on rajoute des autoroutes (en hauteur ...).
– La dynamique horizontale de la voiture sera finie (confère section 3)
– On ajoutera de nombreux véhicules à cette ville, pour l'égayer.
– Le système de particule va nous permettre de modéliser des fontaines ainsi que des dérapages pour la voiture.