Info-SUP : A2, Promo 2007

ECSTASY OF AGONY

Deuxieme Soutenance

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

ECSTASY OF AGONY

QUADRAT QUENTIN,
KADIRI ANASS

Table des matières

1. Introduction

Notre projet comprend deux parties .Dans la première partie on simule précisement la dynamique d'une voiture. Dans la seconde, on représente 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 est accrochée une roue par un resort. Dans la deuxième soutenance, elle est modélisée par une barre de masse ponctuelle au centre de gravité (de la barre) à laquelle sont acdrochées deux roues par des ressorts. Les roues posséde des pneus ayant un comportement élastiques.

Le modèle a maintenant quatre degré de liberté.

– L'altitude du centre de gravité.
– Les deux allongements des ressorts.
– L'inclinaison (tangage) de la voiture.

Grâce au chargeur de fichier ASCII de 3D Studio Max, implémenté pour la première soutenance, il est plus facile de créer et placer des objets (immeubles, voitures) dans la scène.

L'univers du jeu comprend deux objets : un terrain 3D et une infrastructure. Le terrain servira de plateau pour la mise en place des batiments et permettra de tester la dynamique des voitures.

Objet	Quentin	Anass
Apprentissage de Delphi, Pascal		o
OpenGL (Caméra, primitives, texturage)		o
Loader ASE	o	
Dynamique	o	

Tab. 1. Travail de la première soutenance

Objet	Quentin	Anass
Dynamique des véhicules	o	
Création du terrain		o
Création de la ville	o	
Déplacement des voitures sur une route		o
Menu en Delphi		o

Tab. 2. Travail de la première soutenance

2. Création de terrain

2.1. **Le principe des heightfields.** Le principe de champs de hauteur est relativement simple à comprendre. Dans un premier temps, nous allons generer un maillage carre dans le plan XY.

Le maillage est illustré figure 1. Ce n'est ni plus ni moins qu'un ensemble de polygones carrés (appelés patch pour le distinguer du maillage complet) collés les uns aux autres. L'opération se réalise par deux boucles imbriquées. Une fois le maillage crée, il n'y a plus qu'à affecter aux sommets de chaque

patch une coordonnée en Z. En fonction de la position du sommet considéré dans le maillage, on établit une correspondance avec le pixel de l'image en niveau de gris. Si le pixel correspondant est noir, l'élévation du sommet sera minimale alors que s'il est blanc, la hauteur sera maximale.

2.2. **Des polygones qui dégènerent.** En utilisant le principe précédant, nous sommes confronté a un problème : nous allons créer des polygones degénérés. OpenGL définit strictement les propriétés que doit respecter un polygone pour obtenir un rendu correct. Les polygones doivent etre convexes (c'est à dire que toute ligne joignant deux points quelconques du ploygone doit être entiérement comprise dans le polygone), non auto-intersectants et tous les points doivent être sur le même plan. Or, cette dernière condition n'est pas verifiée lorsqu'on genère le terrain puisque les valeurs d'élévation des sommets sont issus d'une image, et il y a peu de chance que les quatre sommets de chaque patch soient dans un même plan. Dans le cas d'affichage en mode fil de fer, la dégéneration des polygones ne pose pas problème. En revanche, si on éclaire la scène, le rendu des faces en mode plein sera faux. Autant prendre les devants et afficher quelque chose de correct.

La solution au problème de dégénération est simple : il suffit de décomposer chacun de nos patchs en deux triangles. Notre maillage global ressemble donc à la figure 1. L'arrete transversale est disgracieuse. Nous allons nous en débarasser grâce en utilisant la fonction glEdgeFlag d'OpenGL qui permet de n'afficher que certaines arrêtes d'un polygone. Le prototype de glEdgeFlag est le suivant :

```
procedure glEdgeFlag(valeur : boolean);
```

valeur peut valoir TRUE ou FALSE. les arrêtes dont le premier sommet est défini lorsque le drapeau est sur TRUE seront affichées. En revanche, si le drapeau est sur FALSE, l'arrête n'est pas affichée. Bien sur ceci n'a d'intérer que lors d'un affichage en mode de fer.

2.3. **Le format RAW.** C'est avec des textures au format RAW que l'on va calculer la hauteur de chaque sommet du patch, car c'est un format permettant de lire une image comme un tableau de nombres entiers compris entre 0 et 255. Il est donc très utile pour passer d'une image à un tableau de calcul et réciproquement. Pour ce faire nous avons besoin d'utiliser la fonction blockread qui lit un ou plusieurs enregistrements d'un fichier ouvert et les place dans une variable. Voici comment fonctionne cette fonction :

```
procedure BlockRead( var Fichier : File;
                     var Tampon;
                     compte : Integer
                    [;var Transfere : Integer]
                   );
```

BlockRead lit au moins Compte enregistrements à partir du fichier Fichier et les transfère en mémoire en partant de l'octet occupé par Tampon. Le nombre réel d'enregistrements entiers lus (inférieur ou égal à Compte) est renvoyé dans Transfere. On utilisera aussi la fonction SizeOf(X) qui renvoie le nombre d'octets occupés par une variable ou un type (ici X).

En un mot, pour calculer la hauteur de

On créait la procedure LoadRawFile qui utilise le tableau de byte appelé Relief et une constante de type chaine, Chemin, qui indique le chemin de la texture.

```
CONST Chaine = 'C:\MaTexture.raw';
VAR {parametre global}
    Relief : array [0..1023,0..1023] of byte;

procedure LoadRawFile();
begin
{ Ouverture du fichier RAW,  Attention : on supose que le
 fichier existe
}
    AssignFile(Fichier, chemin);
    BlockRead(F,image,sizeof(image));
    CloseFile(Fichier);
end; {LoadRawFile}
```

2.4. **La création du terrain.** Le terrain est de dimension fixe : il sagit d'un carré dans le plan XY dont les extrémités sont les points $(-1,-1,), (-1,1), (1,-1), (1,)$. La densité du maillage on utilise une constante NB_SUBDIVISION qui représente le nombre de subdivisions sur chaque axe. Les tableaux P1,P2,P3 et P4 ont chacun quatre cases qui contiennent une alititude.

```
CONST NB_SUBDIVISIO = 4;
for i := 0 to NB_SUBDIVISION do
begin
    for j := 0 to NB_SUBDIVISION do
    begin
       P1[0] := -1+i*pas; P1[1] := -1+j*pas;
       P1[2] := Relief(i,j);

       P2[0] := -1+(i+1)*pas; P2[1] := -1+j*pas;
       P2[2] :=  Relief(i+1,j);

       P3[0] := -1+(i+1)*pas; P3[1] := -1+(j+1)*pas;
       P3[2] := Relief(i+1,j+1);

       P4[0] := -1+i*pas; P4[1] := -1+(j+1)*pas;
       P4[2] := Relief(i,j+1);
    end;
end;
```

2.5. **Les liste d'affichage d'OpenGL.** Les listes d'affichage contribuent a l'amélioration des perfomances graphiques parce qu'elles permettent de stocker des commandes OpenGL pour une utilisation future. L'utilité la plus évidente est l'enregistrement d'objets récurrents. Par exemple pour dessiner une salle de classe, il suffit d'enregistrer l'objet chaise dans une liste d'affichage, que l'on appelle autant de fois que l'on a de chaises dans la salle. Une fois la liste d'affichage créee, il n'est plus possible de la modifier, sinon les performances s'en trouveraient affectées par le sondage de la liste et la

gestion de la mémoire (fragmentation de la mémoire). Elles permettent notamment d'optimiser les opérations matricielles, les lumières, matériaux et les textures. Voici, comment elles fonctionnent.

```
function GlGenLists(n : GLsizei) : Gluint;
```

avec n le nombre de listes que l'on souhaite créer. La fonction renvoie un bloc de n identifiants.

```
proc\'edure glNewList(liste Gluint, mode GLenum);
```

avec liste doit être un entier retourné par glGenLists() ;. mode peut prendre les valeurs GL_COMPILE et GL_COMPILE_AND_EXECUTE. La dernière enregistre et exécute immediatement. Si on utilise GL_COMPILE, les instructions ne sont pas exécutées lors de l'enregistrement.

Les instructions OpenGl qui suivent l'appel de glNewList sont mémorisées dans la liste d'affichage jus'quà l'instruction qui marque la fin de l'enregistrement.

```
procedure glEndList();
```

Pour exécuter une liste d'affichage, il suffit d'appeler :

```
procedure glCallList(liste : Gluint);
```

où liste est l'inditifiant de la liste. Il est également possible d'éffacer une liste d'affichage avec :

```
procedure glDeleteLists(liste : Gluint; n : GLsizei);
```

où n désigne le nombre de liste à éffacer (le même nombre de liste réservées par glGenLists() ;).

3. Création d'une ville

3.1. **Structure de la ville.** Notre ville est de type américaine, à savoir constituée de blocs réguliers séparés par des routes nord–sud et est–ouest.

Chaque bloc (figure 2) est composé de maisons tirées au hasard dans une bibliothèque (à construire au cours des prochaines soutenances). Le bloc de maison est composé, en fait, de quatre alignements de maisons de tailles differentes autour d'une cour intérieure. La taille de tous les blocs sont identiques, mais pourront varier dans l'avenir.

A chaque sommet de la grille de la ville on tire au hasard l'altitude du carrefour qui est suposé être horizontal.

La ville est un objet Pascal (une matrice d'objets de type bloc) et une matrice contenant l'altitude des carrefours.

Un bloc est un objet qui contient sa taille, sa position, quatre tableaux de maisons qui cooresponden à chaque face du carré du bloc.

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

A chacun de ces trois objets sont associés deux fonctions : la première pour créer l'objet, la seconde pour l'afficher en OpenGL.

L'utilité de créér une ville sous la forme d'une grille est multiple.

- Savoir dans qu'elle case et sur qu'elle route se trouve le joueur sans faire de moindre test.
- Tester une collision qu'avec les immeubles de la case et non tous ceux de la carte.

– Uniquement pour les huit cases aux alentours, on teste si un batiment est dans le cône de vision de la caméra. Tous les autres ne sont pas affichés.
– Gérer que les voitures qui sont dans les huit cases aux alentours de la voiture.

3.2. **Les voitures.** Pour créer une certaine dynamique des voitures, il a fallu recourir aux lois de la physique, plus précisément de la mécanique. Ce sont les champs de force qui vont gérer les déplacements des voitures dans le jeu. Deux types de champ de forces sont donc nécessaires, les uns attractifs, les autres répulsifs. Les champs attractifs permettront à une voiture de se déplacer en direction d'un point donné. Les champs répulsifs eux permettront aux voitures neutres dans la ville de circuler sans s'engloutir dans un coin, c'est-à-dire qu'il y aura toujours une force qui permettra de laisser une distance de sécurité entre deux voitures afin d'éviter les accidents.

Pour modéliser cette dynamique des voitures, nous avons schématisé un circuit sous forme de carré. Chaque sommet du carré sera un point d'attraction pour la voiture. Ainsi la voiture initialisé sur un certain sommet, sera attirée successivement par chacun des sommets du carré, celle-ci fera donc le tour du circuit. Ensuite nous augmenteront le nombre de voitures sur le circuit, et mettront en place des champs de force répulsifs, cette fois ci entre les voitures afin d'éviter les collisions.

Au niveau de l'implémentation, il a fallu tout d'abord créer un type voiture où l'on stockerait toutes les informations concernant la voiture : la position, la vitesse, la vitesse maximum, la couleur, etc. Ceci étant fait, il ne reste plus qu'à appliquer les champs de force sur la voiture et afficher le tout.

La dynamique des voitures est définie à partir de deux champs de forces exercées par B sur A :
– le premier est attractif et d'intensitĺinéaire avec la distance entre les voitures :
$$\overrightarrow{F}_a = k_a \overrightarrow{AB}$$
– le second champ est répulsif et d'intensité inversement proportionnel à la distance (r) entre les voitures :
$$\overrightarrow{F}_r = -k_r \frac{\overrightarrow{AB}}{r^2} .$$

On supposera les masses des voitures identiques égales à 1. Alors l'acélération (γ) de A due au champ de force exercé par B vaut
$$\overrightarrow{\gamma} = \overrightarrow{F}_a + \overrightarrow{F}_r .$$

On impose de plus au vecteur vitesse les contraintes $|v| \leq v_{max}$, on en déduit la modification du vecteur vitesse v' lorsque A subit l'accélération due à B pendant une unité de temps :
$$v := \min\left(\max\left(v + \gamma, -v_{max}\right), v_{max}\right),$$

Pour que les voitures aient une distance de sécurité, il suffit d'ajuster les coefficients k_a et k_r de façon à ce que la somme des deux champs s'annule à cette distance de B (voir figure 4).

4. Dynamique de la voiture du joueur

4.1. **Introduction.** Le véhicule est modélisé en 2D, par une carcasse représenté par une barre 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, (figure 3).

4.2. **Les équations du mouvement.** 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 :

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

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 :

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

Comme dans la section précédente, 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 :

$$R_1 = k[u(x+l) - (y_1 + y + l\theta - R)]^+ ,$$
$$R_2 = k[u(x-l) - (y_2 + y - l\theta - R)]^+ ,$$

avec k une constante.

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 :

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

4.3. **Discrétisation des équations différentielles.** 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 :

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

5. Prévisions pour la troisième soutenance

- On continura d'améliorer la ville grâce à la fusion terrain–ville. On mettra en place des feux de signalisation, des routes de différentes largeurs et il n'y aura pas que des carrefours à quatre croisements.
- Continuation de la dynamique des voitures.
- Meilleure gestion des voitures dans les carrefours.

6. Conclusion

'O Fortunatos nimium sua si bona norint agricolas !', ceci voulant dire : 'Trop heureux les hommes des champs s'ils connaissaient leur bonheur !' (latin).

Vers de Virgile dont on ne cite souvent que la première partie, laquelle s'applique à ceux ceux qui jouissent d'un bonheur qu'ils ne savent pas apprécier. Cette citation se trouve en page une, dans *Asterix chez les bretons.*

7. Annexes

Ce travail ayant été affectué afin d'améliorer le jeu, n'a pas besoin d'être détaillé davantage.

- Implémentation d'une fenêtre Delphi qui permet(ra) des modifier les paramètres du jeu (Résolution de l'affichage, sons, choix et paramétrages des voitures)
- Enrregistrement des paramètres dans des fichiers pour qu'à chaque démarage le joueur ne reconfigure pas ses options.
- Nos voitures viennent du jeu Midtown Madness 2 (préalablement exportées dans 3D Studio Max). Sachant que ces voitures n'ont qu'une seule texture et formées de peu de polygones, elles sont donc parfaitement optimisées pour notre jeu.

FIG. 1. Maillage avec les faces transversales

FIG. 2. Modèle d'un bloc de la ville

FIG. 3. Le système dynamique