Info-SUP : A_2, Promo 2007

ECSTASY OF AGONY

Quatrième Soutenance

Quadrat Quentin, quadra_q
Kadiri Anass, kadiri_a

ECSTASY OF AGONY

Table des matières

1. Introduction

Ecstasy est un projet réalisé en OpenGl et Delphi. Il comprend deux parties : – la première partie a pour but de simuler le plus précisement possible la dynamique d'une voiture, – la deuxième de représenter en 3D la circulation automobile d'une ville de type américaine (c'est à dire ayant une forme géométrique régulière).

Cette idée a été inspirée par des jeux tels que GTA ou Midtowm Madness et un projet scientifique appelé STARDUST [1] (Towards Sustainable Town Development : A Research on Deployment of Urban Sustainable Transport) dont l'objectif est de simuler une ville et ses véhicules, pour en étudier les problèmes de circulation.

2. Cahier des charges

Voici, le cahier des charges avec tous ses objectifs que nous nous étions fixé. On peut remarquer, que presque tous ont été respecté (sachant qu'une personne a abandonné en cours de route).

2.1. **Objet de l'étude.**

Ce projet a plusieurs objectifs.

Le premier objectif est d'apprendre à travailler en groupe, et à transmettre nos idées de façon claire. Un groupe anarchique, où il y aurait quatre chefs au lieu d'un, est un groupe qui ne survit généralement pas longtemps. Il faudra apprendre à concillier les attentes et les désirs de chacun et donc d'apprendre à gérer au mieux les capacités de chacun.

Le deuxième objectif est d'apprendre sous Delphi, le langage Pascal, la manipulation de la 3D sous OpenGL et 3D Studio Max et le son avec Fmod. Aussi, sous Emacs, nous utiliserons LaTeX pour rédiger les cahiers de soutenance. Enfin, il faudra créer notre prope page WEB grâce à DreamWeaver.

2.2. **Découpage du projet.**

Le projet se découpe en quatre grandes parties :

- Construction d'un univers 3D (terrain, ville).
- Réalisation d'une dynamique des voitures la plus réaliste possible.
- Réalisation d'une logique des voitures ennemies et neutres.
- Finalisation du jeu pour une meilleure jouabilité (effets spéciaux, sons, menus).

Dans un premier temps, la création d'une base de données des différentes voitures et immeubles, préalablement dessinés sous 3DStudio Max, vont permettre la création d'un univers plus complexe au fur et à mesure des soutenances. Dans un premier temps, grâce à la dynamique du véhicule, la voiture du joueur pourra joyeuseument 'gambader' sur les terrains caillouteux pour tester ses nouvelles suspensions et faire de joyeuses embardées pour s'encastrer dans le décor. Dans un second temps, de nombreuses voitures neutres vont parcourir les routes cabossées sur lesquelles des voitures ennemies-amies vont se poursuivre par des champs attractifs/répulsifs. On tentera d'égayer le jeu par des tirs, des effets spéciaux grâce à OpenGL et des effets sonores grâce à Fmod. On y intégrera un menu créé par Delphi permettant de lancer le jeu et de le configurer.

En résumé, nous avons :

– *Construction d'un monde aléatoire :*
  Création d'un terrain aléatoire, construction d'une ville avec des blocs d'immeubles et des routes. Le tout généré par programmation.
  Elève responsable du bon fonctionnement de cette partie : Kadiri Anass
– *Dynamique des véhicules :*
  Mise en place de la dynamique des voitures, c'est-à-dire de l'accélération, des vitesses, des frottements, des suspensions des roues par rapport à la hauteur du terrain et des rotations des roues (avant, arrière, gauche, droite). Collisions entre les véhicules et le décor ainsi que des voitures entre elles.
  Elève responsable du bon fonctionnement de cette partie : Quadrat Quentin.
– *Scénarisation et logique des voitures :*
  Mise en place d'un scénario : comment les voitures ennemies et amies vont se poursuivre (champs attractifs, répulsifs, tirs, déplacement des voitures neutres).
  Elève responsable du bon fonctionnement de cette partie : Ngombe Jessica.
– *Finalisation du projet :*
  Création des menus (fenêtres Delphi), mise en place des sons, modélisations des voitures et des immeubles, effets spéciaux avec OpenGl (tirs, explosions ...).
  Elève responsable du bon fonctionnement de cette partie : tout le monde.

2.3. **Première soutenance.** Création d'un exportateur de fichier ASCII de 3DStudio Max utilisable pour OpenGl. Chacun va se spécialiser dans un domaine : chaque personne va concevoir des objets élémentaires (une personne pour créer des batiments, une autre pour les blocs de voitures et les blocs de paysages pour la dernière). Ceci va céer notre base de donnée d'objets. Dans un deuxième temps, tout le monde va apprendre à utiliser 3DStudio Max en fonction du loader fraichement créer, puis apprendre à programmer OpenGl et Pascal avec le loader.

Actions	Quadrat	Kadiri	N'gombe
Loader ASE vers Opengl	X	-	-
Apprendre la syntaxe Pascal	-	X	X
Premiers exemples en Delphi	-	X	X
Apprendre Opengl	X	X	X
Premiers exemples en OpenGl	X	X	X
HTML et LaTeX	X	X	X

2.4. **Deuxième soutenance.** Assemblage de la ville par programmation. Une personne s'occupe de l'assemblage des bouts de batiments pour donner au final des blocs de batiments. Une autre pour créer des voitures et une derniere pour créer des paysages 3D. La première voiture articulée va être créée. Mais, il n'y aura pas encore de déplacement.

Actions	Quadrat	Kadiri	N'gombe
Articulation des voitures	X	-	-
Création des voitures	X	-	-
Création d'une ville simple	-	X	-
Création de routes simples	-	-	X
Création de paysages	-	-	-
HTML et LaTeX	X	X	X

2.5. **Troisième soutenance.** Au moins une voiture articulée se déplace dans la ville. Création de la logique des voitures (stratégie, poursuites) sur un plan 2D. Sons, menus, effets spéciaux.

Actions	Quadrat	Kadiri	N'gombe
Déplacement d'une voiture	X	-	-
Logique des voitures neutres	X	X	-
Logique des voitures ennemies	-	-	X
Effets spéciaux avec OpenGl	X	X	X
Création de menus	-	-	X
Sauvegarde et chargements	-	-	X
Sons	-	X	-
HTML et LaTeX	-	-	X

2.6. **Quatrième soutenance.** Stratégie finie, assemblage de tous les éléments. Tirs, explosions. Finiolage, amélioration de la beauté du jeu et relecture des bogues. Création du CD, site web, version light, install Shield.

Actions	Quadrat	Kadiri	N'gombe
Introduction AVI	-	-	X
Déplacements des voitures	X	X	-
Tirs	X	X	-
Pauffinage du jeu	X	X	X
Déboggage	X	X	X
Améliorations des dessins	X	X	X
Install Schield	-	-	X
CD, pochette, web, papiers	-	-	X

2.7. **Conclusion du projet.** Ce projet va permettre pour la plupart d'entre nous de débuter en informatique et de concevoir un jeu vidéo. Ceci va nous contraindre à des démarches personelles pour apprendre un nouveau langage de programation qu'est le Pascal, et la modélisation 3D grâce à OpenGL et 3D Studio Max. Ce projet nous permettra donc d'acqérir de l'éxpérience au niveau de la programmation et d'appliquer efficacement nos cours reçus en classe.

3. Ce qui a été réalisé dans ce projet

	Soutenance N° 1	Soutenance N° 2	Soutenance N° 3	Soutenance N° 4
Apprentissage de Delphi, Pascal	✓			
OpenGL (Caméra, primitives, texturage)	✓		✓	
Loader ASE	✓			
Dynamique des véhicules		✓	✓	✓
Création du terrain		✓		
Création de la ville		✓	✓	✓
Menu Delphi		✓		
Automatisation du chargt des voits			✓	
Caméra et Frustum			✓	
Amélioration de la structure de la ville			✓	
Ville infinie				✓
Skybox, ponts, eau, immeubles			✓	
Blending, fog, lumières			✓	
Circulation des voitures				✓
format des textures (tga, bmp, jpeg)			✓	
Héritage, encapsulation				✓
Sons, Vidéos			✓	
Plan de feux				✓

3.1. **La voiture.** 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 (cf section 5.2).

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 (cf section 5.3).

Pour la troisième soutenance la voiture commandée par le joueur pouvait se déplacer dans la ville en 3D . Le fonctionnement de la suspension est visible lors des changements de pentes des routes. Seul le tangage (cf section 5.4) avait un sens mécanique.

Pour la quatrième soutenance la dynamique du mouvement horizontal a été réalisée. Pour faire avancer la voiture, le joueur donne des consignes de direction et de puissance du moteur, une dynamique du mouvement horizontal d'une voiture est simulée de façon très simplifiée (cf section 5.4). En particulier l'inertie de rotation est représentée, par contre les frottements avec le sol sont très simplifiés.

3.2. **La structure de la ville.** L'élaboration de la ville n'a commencée que lors de la préparation à la deuxième soutenance. La structure de la ville constituée de blocs, eux-mêmes constitués de batiments (tirés au hasard) et de routes a été définie (cf section 4.2). Les routes ne possédaient pas de

textures et les immeubles étaient représentés par des cubes de tailles entières entre 1 et n.

Le travail pour la troisième soutenance a permis d'achever l'infrastructure de la ville : – textures au format bmp, tga ou jpeg, – 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) (cf section 8.5, 8.4, 8.2). 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 grâce à de la présence obligatoire d'un immeuble de taille un).

Le travail pour la dernière soutenance, a été d'améliorer les bords de la ville. On l'a rendue sans limite en la plaçant sur un tore 2D. La ville peut donc être vue comme de taille infinie mais périodique (cf section 4.2). L'aspect des immeubles a été amélioré en rajoutant des textures pour le jour et pour la nuit. Les particules des feux tricolores ont été arrangées, car lors de la deuxième soutenance car elles ne s'affichaient pas correctement. Les plans de feux ont été mis en place.

3.3. **La circulation.** La circulation n'a été implémentée que pour la dernière soutenance . La ville est découpée en blocs réguliers. Chaque bloc possède deux routes à quatre voies. A chaque voie est associé la file de voiture qu'elle contient . Les voitures se déplacent et s'arrêtent aux feux rouges (cf section section 6). Chaque voiture est simulée par un système de onze équations différentielles. Le jeu a fonctionné avec 432 voitures dans une ville de 36 blocs à 18 images 1600x1200 par seconde sur un PC Athlon 1.8Ghz possédant une carte graphique Radeon 8500.

3.4. **Utilisation de OpenGL.** Un chargeur pour OpenGL d'objets 3D — créé dans 3D Studio Max (ASE) — a été réalisé pour la première soutenance. Cela a permis la création de carcasses de voitures sophistiquées (cf section 7).

Pour la deuxième soutenance une débauche de création de terrain 3D en fil de fer (plus couramment appelé height map) a été réalisée. Cette structure n'a pas été utilisée dans la suite par manque de temps (cf section de 8.7).

Pour la troisième soutenance, on a étudié comment mettre en place les lumières, la brume et la transparence dans notre ville (cf section de 8.2 8.4, 8.5). Ensuite, on a implémenté un petit système de particules pour simuler des effets tels que la pluie, etc.

3.5. **Utilitaires.** Pour la deuxième soutenance, on a créé un menu en Delphi pour prendre en compte les différents paramètres du jeu. Pour la troisième soutenance, on a mis en place la vidéo d'introduction et le son avec FMod puis DirectSound.

Pour la dernière soutenance, on a débogué, nettoyé et commenté nos codes sources. On a ainsi obtenu des gains en temps de calcul. Finalement on a étudié la notion d'héritage des objets et on a amélioré les autorisations d'accés aux champs des objets.

4. La ville

4.1. **Introduction.** 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. Pour éviter la description du comportement du bord de la ville, elle a été placée sur un tore 2D. Ce qui signifie qu'elle est infinie et périodique. Dans cette ville, circulent des voitures dans les deux sens sur les artères nord-sud et est-ouest, munies de feux de circulation. Une voiture guidée par le joueur se déplace dans la ville. Le comportement mécanique de chaque voiture est décrite par un système de onze équations différentielles.

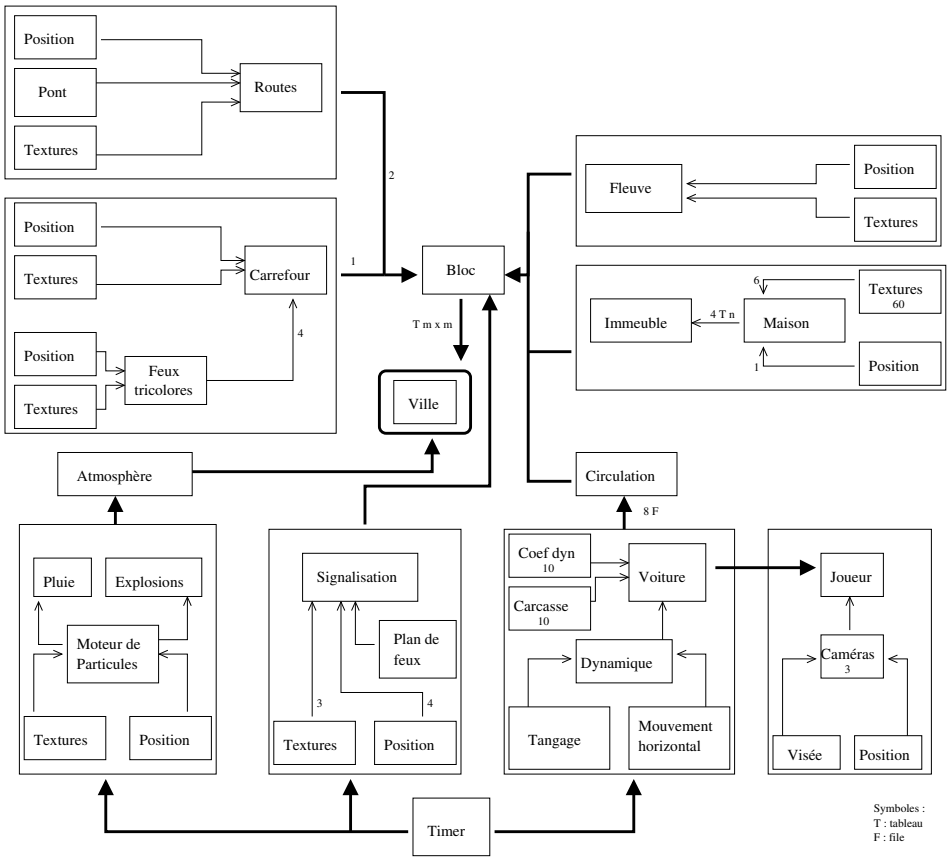

Fig. 1. Le squelette du projet

4.2. **Le bloc.** La ville est un tableau $n \times n$ de blocs. Chaque bloc de la ville, figure 2, est une classe Delphi composée de :

- un bloc de maisons tirées au hasard dans une bibliothèque — réalisées sous 3DSMax, puis importée grâce au chargeur ASE. Il est composé de quatre alignements de maisons de tailles différentes autour d'une cour intérieure.
- deux routes de quatre voies qui ne sont pas horizontales ;
- un carrefour horizontal avec une altitude et sa signalisation constituée de quatre feux ;

– deux tableaux de quatre files de voitures contenant des pointeurs sur la voiture suivante et la voiture précédente.

Il est affiché au moyen d'une liste d'affichage OpenGL.

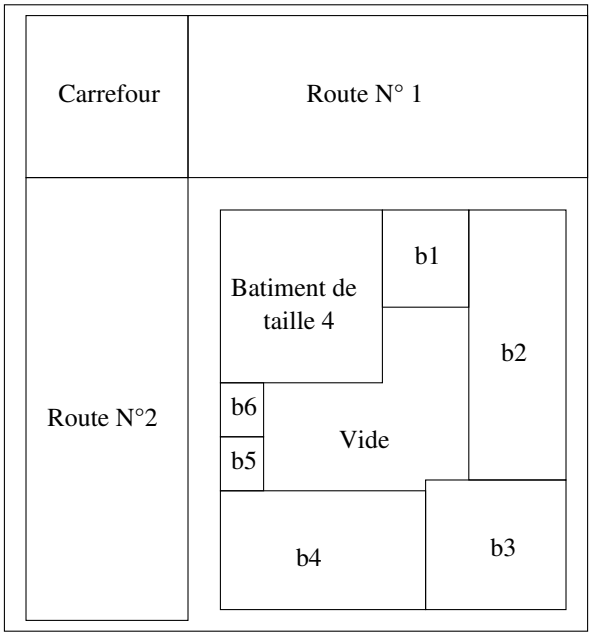

FIG. 2. Modèle d'un bloc de la ville

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

Une route est un enregistrement qui contient les coordonnées de ses quatre sommets.

Le type carrefour est constitué d'un type route et de quatre feux tricolores. Une fonction réalise le plan de feux du blocs et affichent des textures colorées (rouge, jaune ou verte). Ces textures sont particulières puisqu'elles utilisent la transparence d'OpenGL (confére section 8.4).

Chaque liste chainée de type TFile est associée à une voie de circulation. Chaque rue à quatre voies est à deux sens. Dans l'état actuel du jeu, les voitures autres que le joueur restent sur leur voie mais s'arrêtent aux feux (cf 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 une voiture. Il suffit de faire la division entière de sa position par la taille du bloc pour obtenir le numéro de la case.

Le fait que les voitures soient attachées au bloc permet des gains de temps d'affichage. En effet, dans le pire des cas, seul neuf blocs sont affichés : le bloc du joueur et les huit connexes en utilisant un algorithme permettant d'éviter de calculer les objets n'appartenant pas au cône de vision (cf section 9.3).

5. La dynamique des voitures

La partie la plus difficile théoriquement est la modélisation de la mécanique de la voiture incluant une suspension simplifiée. Le tangage et le mouvement horizontal ont été modélisés. Le roulis a été négligé.

Fig. 3. Une Audi Monster

5.1. **Principe de la moindre action.** On appelle action $\mathcal{A}$ d'un système mécanique l'intégrale le long du mouvement de la différence de son énergie cinétique et de son énergie potentielle :

$$\mathcal{A}(x()) = \int (\mathcal{E}_c(x(t)) - \mathcal{E}_p(x(t))dt \ ,$$

où $\mathcal{E}_c(x(t))$ designe l'énergie cinétique, $\mathcal{E}_p(x(t))$ l'énergie potentielle et $t \mapsto x(t)$ la trajectoire du système.

Le principe de la moindre action nous dit que la trajectoire du système est celle qui minimise l'action.

Pour trouver cette trajectoire on calcule la variation de l'action $\delta\mathcal{A}$ associée a la une variation de la trajectoire δx et on détermine les conditions qui assurent que $\delta\mathcal{A}$ soit nul quelque soit δx.

Prenons le cas de deux masses, de poids respectifs m_1 et m_2, accrochées l'une à l'autre par un ressort de force $F = -kl$ ou l désigne l'allongement du ressort ($|x_1(t) - x_2(t)|$) avec $x_1(t)$ et $x_2(t)$ les positions à l'instant des deux masses.

$$\begin{aligned}\delta\mathcal{A} = &\int (m_1(\dot{x_1} + \delta\dot{x_1})^2 + m_2(\dot{x}_2 + \delta\dot{x}_2)^2 - k(x_2 + \delta x_2 - x_1 - \delta x_1)^2)dt \\ &- \int (m_1\dot{x}_1^2 + m_2\dot{x}_2^2 - k(x_2 - x_1)^2)dt \ , \\ \delta\mathcal{A} = &\int m_1\dot{x}_1\delta\dot{x}_1 + m_2\dot{x}_2\delta\dot{x}_2 - k(x_2 - x_1)(\delta x_2 - \delta x_1) + o(\|x_1 - x_2\|) \ ,\end{aligned}$$

Par intégration par partie on obtient :

$$\delta\mathcal{A} = \int -m_1\ddot{x_1}\delta x_1 - m_2\ddot{x}_2\delta x_2 - k(x_2 - x_1)(\delta x_2 - \delta x_1) + o(\|x_1 - x_2\|) \ ,$$

car on suppose que les variations des trajectoires sont nulles aux extrémites.

Finalement on trouve :

$$\delta\mathcal{A} = \int (-m_1\ddot{x_1} + k(x_2 - x_1))\delta x_1 + \int (-m_2\ddot{x}_2 + k(x_1 - x_2))\delta x_2 + o(\|x_1 - x_2\|) \ .$$

L'action doit être minimale c.a.d.

$$\delta A = 0, \quad \forall \delta x_1, \ \forall \delta x_2,$$

donc :

$$-m_1\ddot{x_1} + k(x_2 - x_1) = 0, \quad -m_2\ddot{x_2} + k(x_1 - x_2) = 0 \ .$$

On obtient les mêmes équations que celles que l'on on obtiendrait en appliquant la loi fondamentale de la dynamique $m\vec{\gamma} = \vec{F}$.

5.2. **Monocycle.** Le véhicule est modélisé en 2D, par une carcasse de masse ponctuelle M_v accrochée à une roue (de rayon r et de masse M_r) par un ressort. On note $u(t)$ la l'altitude du sol par rapport à au repére, $y(t)$ est l'altitude de la carcasse, $z(t)$ l'alongement du ressort, et $y(t)+z(t)$, l'altitude de la roue. On note g la gravité (figure 5).

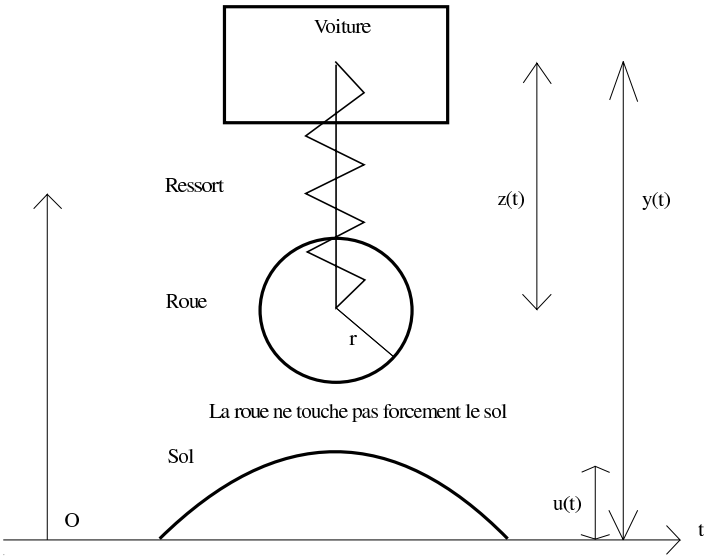

FIG. 4. Le monocycle

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

L'énergie cinétique de la roue (notée $\mathcal{E}_r^c$) est : $\mathcal{E}_r^c = 1/2M_r(\dot{y} + \dot{z})^2$.

L'énergie cinétique de la voiture (notée $\mathcal{E}_v^c$) est : $\mathcal{E}_v^c = 1/2M_v\dot{y}^2$.

L'énergie ressort (notée $\mathcal{E}_r$) est : $\mathcal{E}_r = kz^2;$.

L'énergie potentiel de la voiture (notée $\mathcal{E}_v^p$) est : $\mathcal{E}_v^p = M_v g y$.

L'énergie potentiel de la roue (notée $\mathcal{E}_r^p$) est : $\mathcal{E}_r^p = M_r g(y + z)$.

L'énergie de reaction du sol (notée $\mathcal{E}_s$) est : $\mathcal{E}_s = ((u - (y + z - r))^+)^2$, c'est à dire que $\mathcal{E}_s$ vaut $(u - (y + z - r))^2$ quand $u - (y + z - r) > 0$, sinon il vaut 0.

$$\mathcal{A}(x()) = \int (\ 1/2M_r(\dot{y} + \dot{z})^2 + 1/2M_v\dot{y}^2 - kz^2 - M_v g y - M_r g(y + z) \\ - (u - (y + z - r))^+)^2 \ )dt$$

Comme dans la section précédente, on trouve un système d'équation différentielle où les inconnues sont l'altitude de la roue et de la carcasse :

$$\ddot{y}(M_v + M_r) + y(M_v + M_r) - (u - (y + z - r))^+ = 0 \,, \tag{1}$$

$$\ddot{z}(M_r) + M_r g + kz - (u - (y + z - r))^+ = 0 \,. \tag{2}$$

Pour calculer les trajectoires des deux corps, nous pouvons approximer les équations différentielles (4) et (5) par les équations récurrentes, où h désigne le pas de discrétisation :

$$y(t+h) = 2y(t) - y(t-h) + \frac{h^2}{M_r + M_v}(u(t) - y(t) - z(t) + r)^+ - h^2 g \,,$$

$$z(t+h) = 2z(t) - z(t-h) + \frac{h^2}{M_r}(u(t) - y(t) - z(t) + r)^+ - \frac{h^2 k z(t)}{M_r} - h^2 g \,.$$

5.3. **Moto.** 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 5).

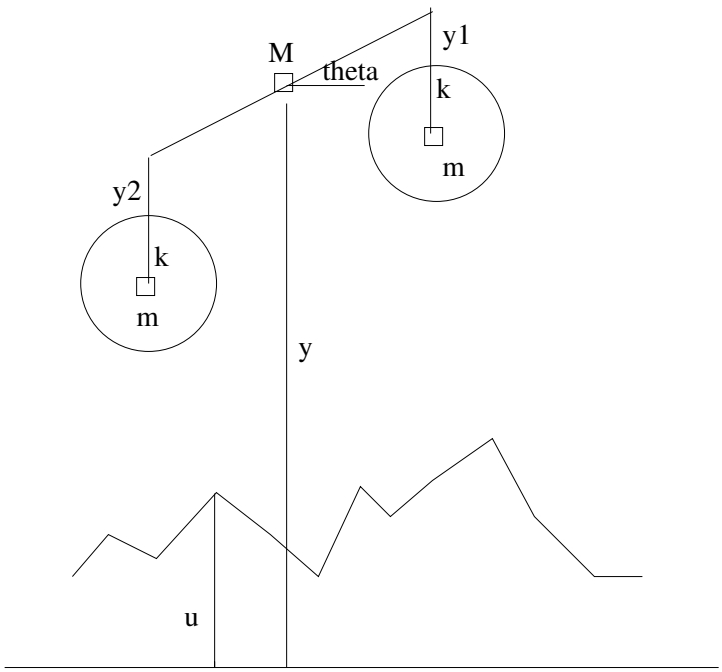

FIG. 5. Modélisation de la voiture

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 $(y_2 + l\cos\theta\dot{\theta} + \dot{y})^2$, 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 :

$$\begin{aligned}R_1 &= [u(x + l) - (y_1 + y + l\theta - R)]^+\ , \\ R_2 &= [u(x - l) - (y_2 + y - l\theta - R)]^+\ ,\end{aligned}$$

On a :

$$(M + 2m)\ddot{y} + m\ddot{y_1} + m\ddot{y_2} = -g(2m + M) + R_1 + R_2\ , \tag{3}$$

$$m(\ddot{y_1} + \ddot{y} + l\ddot{\theta}) = -ky_1 - gm + R_1\ , \tag{4}$$

$$m(\ddot{y_2} + \ddot{y} - l\ddot{\theta}) = -ky_2 - gm + R_2\ , \tag{5}$$

$$m(\ddot{y_1} - \ddot{y_2} + 2l\ddot{\theta}) = R_1 - R_2\ . \tag{6}$$

En faisant (6) plus (5) moins (4) on obtient $0 = k(y_1 - y_2)$ et donc $y_1 = y_2$.

En faisant (3) moins (4) moins (5) on obtient :

$$M\ddot{y} = -gM + 2ky_1\ . \tag{7}$$

L'equation (6) donne alors :

$$2ml\ddot{\theta} = R_1 - R_2 \ . \tag{8}$$

Puis, (4) moins $\frac{m}{M}(7)$ moins $\frac{1}{2}(8)$ donne :

$$m\ddot{y}_1 = -ky_1(1 + \frac{2m}{M}) + \frac{R_1 + R_2}{2} \ . \tag{9}$$

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 :

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

5.4. **Mouvement horizontal.** 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 6), 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).

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 rotation du véhicule autour de son centre de gravite est proportionnelle au changement de direction ξ indiquée. On suppose que lors d'un changement de direction, l'élasticité des pneus — adhérents sur le sol — 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 :

$$\begin{aligned}
(2m + M)\ddot{x} &= a\cos\phi \ , \\
(2m + M)\ddot{y} &= a\sin\phi \ , \\
\dot{\phi} &= \xi \ .
\end{aligned}$$

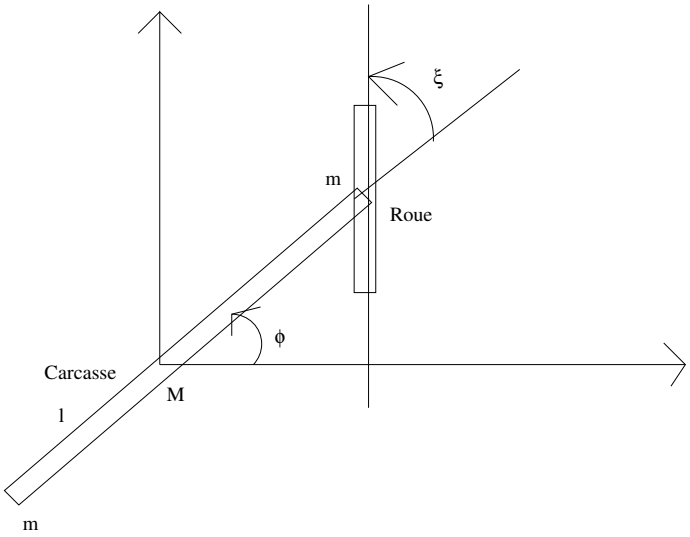

FIG. 6. Modélisation de la voiture (vue de dessus)

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.

L'intégration de ces équations donne le mouvement horizontal des voitures.

6. La circulation

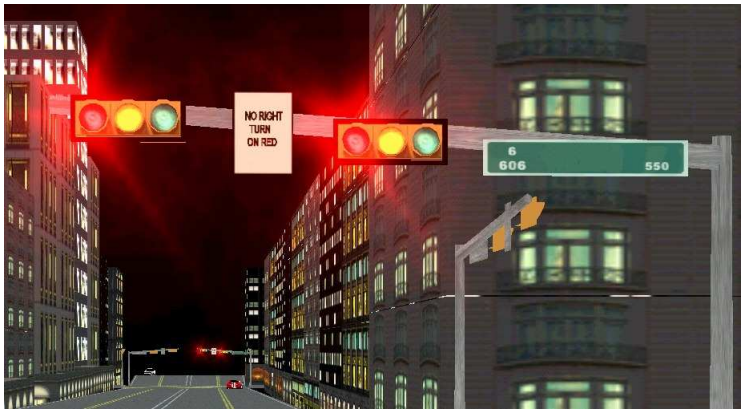

FIG. 7. La pomme est mure

La circulation de la ville comprend, trois éléments : – les feux de signalisation, – le joueur, – les voitures dans les rues.

Le plan de feux de chaque carrefour est simple : il est donné par les deux diagrammes de la figure 8.

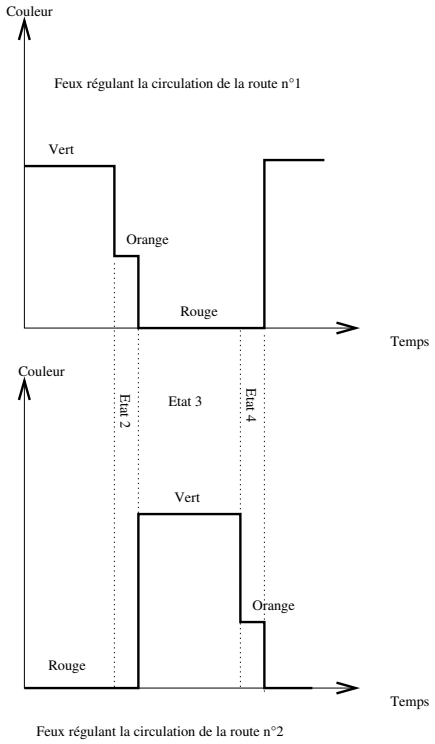

FIG. 8. Plan de feux

Le joueur est une voiture particulière commandée au clavier ou à la souris, en accélération et en direction. Les touches DOWN et UP donnent l'accélération et LEFT et RIGHT la direction. Au joueur, sont associées trois caméras permettant de visualiser les trois scènes suivantes : – vue aérienne sur plusieurs blocs, – vue de la voiture du joueur, – vue de la place du conducteur du joueur. La commande F1 permet de passer de l'une à l'autre. Les comportements dynamiques des voitures dépendent du modéle. La voiture n'est pas contrainte à rester sur les routes, ce qui conduit à des comportements spectaculaires lorsqu'on veut monter sur un immeuble ou sortir et plonger dans le fleuve.

Les voitures circulent toujours sur la même voie, mais s'arrêtent aux feux. La vitesse des voitures est régie par trois facteurs : – une vitesse désirée de la voiture fluctuant avec le temps de façon alétoire, – la vitesse du véhicule précédent – la distance au feu s'il est au rouge. Une voiture étant associée à un bloc, il faut gérer cette appartenance lors des changements de blocs.

Le type TVoiture est une classe Delphi qui contient l'état de la voiture constitué des onze paramètres définissant l'état de sa dynamique. TVoiture descend de TObject, elle hérite donc de ses méthodes. A cela, s'ajoute de nouvelles procédures qui initialisent, actualisent la dynamique et dessinent le véhicule.

A partir de TVoiture deux nouvelles classes ont été créées : TJoueur et TGoodies. La première est la voiture contrôlée par le joueur, la deuxième est une voiture circulant dans la ville.

Les mécanismes d'héritages des classes de Delphi ont été utilisés comme la propriété `inherited`. TJoueur possède de nouvelles méthodes : les proédures qui actualisent le son et la caméra.

TGoodies est une TVoiture qui possède en plus deux pointeurs sur la voiture précédente et suivante. Des méthodes permettent de faire les changements de blocs des voitures.

Les TGoodies sont enchainées sous forme de file associée à une voie d'un bloc et constitue un objet TCirculation. A chaque bloc, est associé un tableau de huit TCirculation.

7. Le chargeur ASE

7.1. **Introduction.** Notre projet comprendra deux types d'objets : des objets statiques comme la ville, le terrain... et des objets dynamiques comme les voitures. Nous les afficherons grâce à OpenGl, mais comme il est imposible de dessiner une maison ou un véhicule triangles par triangles à la main (bien que OpenGl ne comprenne que ça), nous devons utiliser un logiciel spécialisé dans la création 3D : Studio Max. Il permet d'exporter une scène 3D sous un format ASCII : c'est le format ASE (....).

Une scène de Studio Max, notée S, est un ensemble d'objets O_i tels que des plans, des tores, des théières, etc. Donc $S = \{O_0, O_1, ..., O_n\}$. Chaque objet O_i est composé d'un ensemble de triangles t : $O_i = \{t_0, t_1, ..., t_m\}$. Nous appelons triangle, une surface (colorée) définie par trois sommets : $t = (p_1, p_2, p_3)$ où chaque sommet est défini par trois coordonnées : $p_i = (x_i, y_i, z_i)$. Enfin de compte une scène n'est qu'un ensemble de triangles.

7.2. **Les objets 3D avec Studio Max.** Voici comment est représenté une scène composée de deux plans, seul l'essentiel a été pris.

```
*GEOMOBJECT {
        *NODE_NAME "Plan01"
        *MESH {
                *MESH_NUMVERTEX 4
                *MESH_NUMFACES 2
                *MESH_VERTEX_LIST {
                        *MESH_VERTEX 0 -50.0   -50.0   0.0
                        *MESH_VERTEX 1   0.0   -50.0   0.0
                        *MESH_VERTEX 2 -50.0     0.0   0.0
                        *MESH_VERTEX 3   0.0     0.0   0.0
                }
                *MESH_FACE_LIST {
                        *MESH_FACE 0:  A: 2   B: 0   C: 3
                        *MESH_FACE 1:  A: 1   B: 3   C: 0
                }
        }
}
*GEOMOBJECT {
```

```
        *NODE_NAME "Plan02"
        *MESH {
                *MESH_NUMVERTEX 4
                *MESH_NUMFACES 2
                *MESH_VERTEX_LIST {
                        *MESH_VERTEX 0  0.0    0.0   0.0
                        *MESH_VERTEX 1 45.0    0.0   0.0
                        *MESH_VERTEX 2  0.0   30.0   0.0
                        *MESH_VERTEX 3 45.0   30.0   0.0
                }
                *MESH_FACE_LIST {
                        *MESH_FACE 0:  A: 2   B: 0   C: 3
                        *MESH_FACE 1:  A: 1   B: 3   C: 0
                }
        }
}
```

On voit deux types de mots clés : – ceux qui utilisent des champs et – ceux qui ont une accolade entrante. Les premiers sont soit des sommets, soit des triangles, soit des informations, – les autres constituent des listes.

Chaque objet est représenté par le mot clé `GEOMOBJECT`. Son nom (ici c'est un plan) est désigné par `NODE_NAME`. `MESH_NUMVERTEX` et `MESH_NUMFACES` sont respectivement le nombre de sommets et de triangles de notre objet. `MESH_VERTEX_LIST` est la liste des sommets. Chaque sommet est défini par le mot clé `MESH_VERTEX` et quatre paramètres : un entier et trois réels. L'entier est le numéro d'indentification du sommet, les réels représentent une position dans l'espace. Par exemple :

```
*MESH_VERTEX 0 -50.0   -50.0   0.0
```

est le premier sommet de position $(-50, -50, 0)$ et s'écrit : $p_0 = (-50, -50, 0)$.

`MESH_FACE_LIST` est la liste des triangles de notre objet. Chaque triangle est défini par le mot clé `MESH_FACE` et sept paramètres. Par exemple :

```
*MESH_FACE 0:  A: 2   B: 0   C: 3
```

Le 0 : est le numero d'identification du triangle. Le `A :` est le premier sommet du triangle, le `B :` le second et enfin le `C :` le troisième. Le 2 de `A : 2` est un pointeur vers le troisième sommet de la liste des sommets. Il pointe sur :

```
*MESH_VERTEX 2 -50.0   0.0   0.0
```

Le principe est le même pour le 0 de B : 0 , il pointe sur :

```
*MESH_VERTEX 0 -50.0   -50.0   0.0
```

et enfin le 3 de C : 3 , pointe sur :

```
*MESH_VERTEX 3 0.0   0.0   0.0
```

En définitive, nous avons : $t_0 = (p_2, p_0, p_3)$.

7.3. **Les objets 3D avec OpenGl.** Une scène avec Opengl est elle aussi un ensemble de triangles. Voici comment est représentée la même scène toujours composée de deux plans.

```
glBegin(GL_TRIANGLES);
// Premier triangle du premier plan
```

```
   glVertex3f(-50.0,0.0,0.0);
   glVertex3f(-50.0,-50.0,0.0);
   glVertex3f(0.0,0.0,0.0);

// Deuxieme triangle du premier plan
   glVertex3f(0.0,-50.0,0.0);
   glVertex3f(0.0,0.0,0.0);
   glVertex3f(-50.0,-50.0,0.0);

// Premier triangle du deuxieme plan
   glVertex3f(0.0,30.0,0.0);
   glVertex3f(0.0,0.0,0.0);
   glVertex3f(45.0,30.0,0.0);

// Deuxieme triangle du deuxieme plan
   glVertex3f(45.0,0.0,0.0);
   glVertex3f(45.0,30.0,0.0);
   glVertex3f(0.0,0.0,0.0);
glEnd;
```

Le problème principal du chargeur est de stocker correctement la scène, pour qu'elle prenne le moins de place mémoire. L'utilisation de liste chainée est plus astucieuse que celle de tableau, car nous ne connaissons pas à l'avance le nombre exact d'objets ou de triangles. On crée alors une liste de liste. La première contient tous les objets. Dans chaqu'une de ses cases on stocke la liste de triangles propre à chaque objet.

Une remarque importante est que la ligne suivante n'est pas compléte :

```
*MESH_FACE 0:  A: 2   B: 0   C: 3
```

Elle s'écrit, en faite :

```
*MESH_FACE 0: A: 2 B: 0 C: 3 AB: 1 BC: 0 CA: 1 *MESH_SMOOTHING 1
 *MESH_MTLID 0
```

Les valeurs de `AB :`, `BC :` et `CA :` sont des booléens représentés sous la forme de 1 ou de 0. Ce sont des drapeaux d'arrête (Edge Flag), à savoir des bascules qui permetent de n'afficher que certaines arrêtes d'un polygone. `glEdgeFlag` est une procédure OpenGl qui prend en paramétre un booléen. Seules les arêtes sur `TRUE` seront affichées. Ceci n'a d'intérêt que pour un affichage en mode fil de fer.

7.4. **Textures et matériaux avec Studio Max.** Maintenant que nous pouvons exporter n'importe qu'elle scène, nous allons l'égayer en appliquant une texture à nos triangles. Le placage de texture est une technique permettant d'accroître le réalisme d'un rendu 3D. Il consiste à coller une image sur un objet 3D à la manière d'une tapisserie.

Voici comment est représenté une scène composée de d'un seul plan à quatre triangles et d'une texture appelée camouflage.bmp. Elle se trouve dans le chemin : `"C :\camouflage.bmp"` (seul l'essentiel a été pris).

```
*MATERIAL_LIST {
        *MATERIAL_COUNT 1
        *MATERIAL 0 {
```

```
                *MATERIAL_NAME "Materiau #1"
                *MATERIAL_AMBIENT 0.2 0.1 0.1
                *MATERIAL_DIFFUSE 0.5 0.2 0.2
                *MATERIAL_SPECULAR 0.9 0.9 0.9
                *MATERIAL_SHINE 0.2
                *MATERIAL_SHINESTRENGTH 0.05
                *MATERIAL_TRANSPARENCY 0.0
                *MAP_DIFFUSE {
                        *MAP_NAME "Texture01"
                        *MAP_CLASS "Bitmap"
                        *BITMAP "C:\camouflage.bmp"
                        *UVW_U_OFFSET 0.0
                        *UVW_V_OFFSET 0.0
                }
        }
}
*GEOMOBJECT {
        *NODE_NAME "Plan01"
        *MESH {
                *MESH_NUMTVERTEX 8
                *MESH_TVERTLIST {
                        *MESH_TVERT 0 0.0 0.0 0.0
                        *MESH_TVERT 1 1.0 0.0 0.0
                        *MESH_TVERT 2 0.0 0.0 0.0
                        *MESH_TVERT 3 1.0 0.0 0.0
                        *MESH_TVERT 4 0.0 0.0 0.0
                        *MESH_TVERT 5 1.0 0.0 0.0
                        *MESH_TVERT 6 0.0 1.0 0.0
                        *MESH_TVERT 7 1.0 1.0 0.0
                }
                *MESH_NUMTVFACES 2
                *MESH_TFACELIST {
                        *MESH_TFACE 0 6 4 7
                        *MESH_TFACE 1 5 7 4
                }
                *MESH_NUMTVFACES 2
                *MESH_TFACELIST {
                        *MESH_TFACE 0 6 4 7
                        *MESH_TFACE 1 5 7 4
                }
        *MATERIAL_REF 0
        }
}
```

Pour créer une texture sur un triangle, Studio Max définit d'abord les caractéristiques du matériau, ensuite il calcule les sommets et dans un dernier temps, il définit le triangle grâce à la liste de sommet et au numéro d'identification du bitmap.

Notre liste de matériau est appelée par le mot clé : `MATERIAL_LIST`. Ici, nous en n'avons qu'un seul (`MATERIAL_COUNT 1`), appelé Materiau 01. Les mots clés suivants : de `MATERIAL_AMBIENT` à `MATERIAL_TRANSPARENCY` ne sont utiles que pour la réflection de la lumière (ambiante, spéculaire, diffuse) et la transparence de l'objet. Pour plus d'informations sur les lumières confére section 8.2.

Mais ce qui nous intéresse le plus est le chemin du bitmap (`BITMAP "C:\camouflage.bmp"`) et le décalage de la texture sur l'axe u et v (définis respectivement par `UVW_U_OFFSET` et `UVW_V_OFFSET`). En effet, pour positionner une texture sur un polygone, on parle de 'UV mapping'. Le principe est simple : on affecte un système de coordonnées (u, v) à la texture suivant l'illustration. Positionner la texture consiste à affecter à chaque sommet d'un polygone la coordonnée de la texture en ce point. Dans notre cas, notre plan se divise en triangles, les coordonnées aux sommets des points des faces sont : $(0,0),(1,0),(1,1)$ et $(0,0),(0,1),(1,1)$.

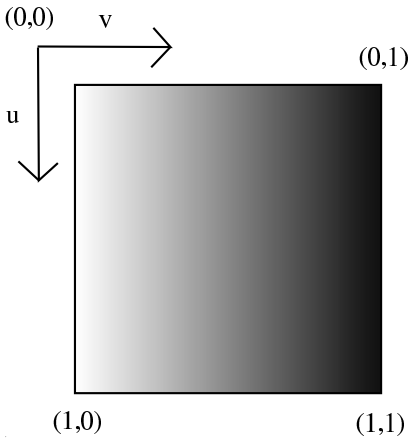

FIG. 9. Coordonnées d'un matériau sur un objet carré

Une fois que la définition de notre texture finie, nous allons créer notre liste de triangles de texture. Le principe est exactement le même que dans la section précédente. Chaque texture est définie par le mot clé `MESH_TVERT`, un entier et trois réels. L'entier est le numéro d' indentification de la texture, les réels définissent la position dans l'espace. Par exemple :

```
*MESH_TVERT 0 0.0 0.0 0.0
```

est le premier sommet $p_0 = (0,0,0)$.

`MESH_TFACELIST` est notre listre de texture. Chaque texture est appelée par `MESH_FACE_LIST` et quatre paramétres (quatre entiers) : un numero d'indentification et trois pointeurs vers un `MESH_TVERT`. Par exemple : $t_0 = (p_6, p_4, p_7)$. est appelé par :

```
*MESH_TFACE 0 6 4 7
```

Pour l'instant, nous ne savons pas quelle texture utilisée (bien qu'il n'y en ait qu'une ici). C'est `MATERIAL_REF` qui nous l'indique. L'entier qu'il prend en paramétre est un pointeur sur le numéro du matériau. Par exemple `MATERIAL_REF 0` pointe sur `MATERIAL 0`.

7.5. **Les textures d'OpenGl.** Voici ce que nous voulons obtenir sous OpenGl (un seul triangle est représenté) :

```
procedure LoaderTexture(NumeroTexture : gluint;
                        Chemin : string);
var texture1: PTAUX_RGBImageRec;
begin
//Chargement de la texture avec glaux
  texture1 := auxDIBImageLoadA(Pchar(Chemin));

//Initialisation de la texture
  glGenTextures(1, NumeroTexture);
  glBindTexture(GL_TEXTURE_2D, NumeroTexture);
  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,
                  GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,
                  GL_LINEAR);
  glTexImage2D(GL_TEXTURE_2D, 0, 3, texture1^.sizeX,
               texture1^.sizeY,0,GL_RGB,
               GL_UNSIGNED_BYTE, texture1^.data);
end;

procedure AfficherTrangle((NumeroTexture : gluint);
begin
//Plaquage de la texture 'NumeroTexture'
  glBindTexture(GL_TEXTURE_2D,NumeroTexture);

//Affichage
  glBegin(GL_TRIANGLES);
     glTexCoord3f(1.0,0.0,0.0); glVertex3f(50.0,0.0,0.0);
     glTexCoord3f(1.0,1.0,0.0); glVertex3f(50.0,10.0,0.0);
     glTexCoord3f(0.0,0.0,0.0); glVertex3f(0.0,0.0,0.0);
  glEnd;
end;
```

Une texture est définie par un entier (un `Gluint`) lors de l'initialisation. Chaque `glTexCoord3f` indique les sommets de la texture.

7.6. **Caméra cible.** Voici comment est représentée une caméra cible sous Studio Max (seul l'essentiel a été pris) :

```
*CAMERAOBJECT {
        *NODE_NAME "Camera01"
        *CAMERA_TYPE Target
        *NODE_TM {
                *NODE_NAME "Camera01"
                *TM_POS 100.0 100.0 100.0
        }
        *NODE_TM {
                *NODE_NAME "Camera01.Target"
                *TM_POS 1.0 1.0 1.0
```

```
      }
}
```

Voici comment Opengl gére une caméra :

```
gluLookAt(100.0,100.0,100.0,
          1.0,1.0,1.0,
          0.0,1.0,0.0)
```

Une caméra est définie par une position p dans l'espace $p = (p_x, p_y, p_z)$, un point qu'elle cible $c = (c_x, c_y, c_z)$ et un vecteur d'orientation (normale) $o = (o_x, o_y, o_z)$ car touner une caméra d'un angle quelconque sur l'axe du vecteur position–cible modifie le sens d l'image (par exemple, on voit différament selon notre environnement si on a la tête à l'envers). Dans notre exemple, nous avons : $p = (100.0, 100.0, 100.0)$, $c = (1.0, 1.0, 1.0)$ et $o = (0.0, 1.0, 0.0)$.

8. Effets spéciaux

8.1. **Les liste d'affichage d'OGL.** Les listes d'affichage contribuent à 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 d'utilisation fréquente. 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 créée, 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.

```
procedure 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()`).

8.2. **Les lumières.**

8.2.1. *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 lumière 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'efficacité, 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*)[1]. 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).

8.2.2. *Implémentation avec 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).

8.3. **Paramètres de matériaux.** Dans la réalité, un rayon lumineux est constitué d'un ensemble d'ondes de longueurs différentes. 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 selon les caractéristiques de ce matériau.

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 );
```

[1]Le principe de l'algorithme de Gouraud 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

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).

8.4. **La transparence.**

8.4.1. *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. Si le blending n'est pas activé et que les deux objets sont dessinés l'un par dessus l'autre, OpenGL les dessinera comme étant deux objets opaques. En effet lorsque le deuxième objet est affiché, OpenGL écrase chaque valeur chromatique du cube pour les remplacer par celle de la sphère. Par contre, si le blending est activé, alors OpenGL 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.

8.4.2. *Implémentation.* 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, 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.

8.5. **La brume.**

8.5.1. *Utilité.* La brume (ou fog en anglais) est utile pour apporter du réalisme aux scènes 3D (par exemple pour dessiner 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 dans le cône de vision lorsqu'on ne les visualise pas tous (voir section 9.3).

8.5.2. *Implémentation.* 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).

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.

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

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

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".

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

Enfin, à la ligne (7), on active la brume.

8.6. **Moteur de particules.**

8.6.1. *introduction.* 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). $v(t)$ est $a(t)$ 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 :

$$f(t) = Ma(t) \ ,$$

$$a(t) = \frac{\delta v(t)}{dt} \ ,$$

$$v(t) = \frac{\delta x(t)}{dt} \ .$$

On obtient alors la formule suivante :

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

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 $i+1$:

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

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

Et par réarrangement, on en tire que :

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

$$x(i+1) = x(i) + v(i+1)h \ .$$

8.6.2. *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 $v(i+1)$.

8.6.3. *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 $i+1$.

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

8.6.4. *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.

8.7. **Le terrain.**

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

Le maillage est illustré figure 10. 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.

8.7.2. *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 10. 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.

8.7.3. *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 (infrieur ou égal 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}
```

8.7.4. *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 $(-1,-1,)$, $(-1,1)$, $(1,-1)$, $(1,1)$. 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.

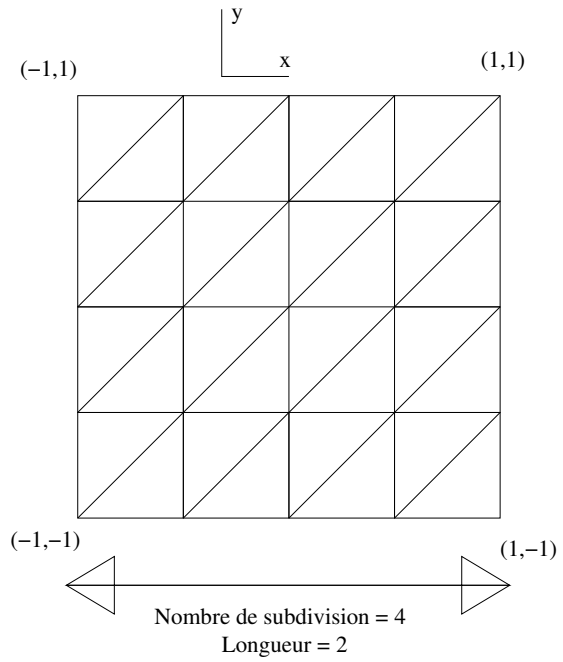

FIG. 10. Maillage avec les faces transversales

```
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;
```

9. UTILITAIRES

9.1. **Importation de voiture.** 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:\test\*.*` indique tous les fichiers du répertoire `C:\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.

9.2. **Courbes de Bezier.** Les courbes de Bezier pourraient servir pour raccorder les routes de façon plus lisse. Bien que dans l'implémentation actuelle cela n'a pas été faite — pour pouvoir mieux apprécier le fonctionnement des suspensions — donnons une introduction aux courbes de Bezier.

Les courbes de Bezier sont des courbes à fonctions paramétriques $x(t)$ et $y(t)$ de degré trois, c'est à dire qu'elles sécrivent sous la forme : $x(t) = a_0 + b_0 t + c_0 t^2 + d_0 t^3$ et $y(t) = a_1 + b_1 t + c_1 t^2 + d_1 t^3$.

Une courbe de Bezier est définie par ses deux extremités de position respectifs $(x_0, y_0) = (x(0), y(0))$ et $(x_1, y_1) = (x(1), y(1))$ et ses deux demi-tangentes $(u_0, v_0) = (x'(0), y'(0))$ et $(u_1, v_1) = (x'(1), y'(1))$. Voir le dessin. Pour calculer les coefficients a_0, b_0, c_0 et d_0, il suffit de résoudre le système suivant :

$$\begin{cases} x(0) &= b_0 \\ x'(0) &= a_0 \\ x(1) &= a_0 + b_0 + c_0 + d_0 \\ x'(1) &= b_0 + 2c_0 + 3d_0 \end{cases} \Longrightarrow \begin{cases} a_0 &= x_0 \\ b_0 &= u_0 \\ c_0 &= 3x_1 - 3x_0 - 2u_0 - u_1 \\ d_0 &= u_1 - 2x_1 + 2x_0 + u_0 \end{cases}$$

On fait de même pour les coefficients de $y(t)$.

9.3. **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`.

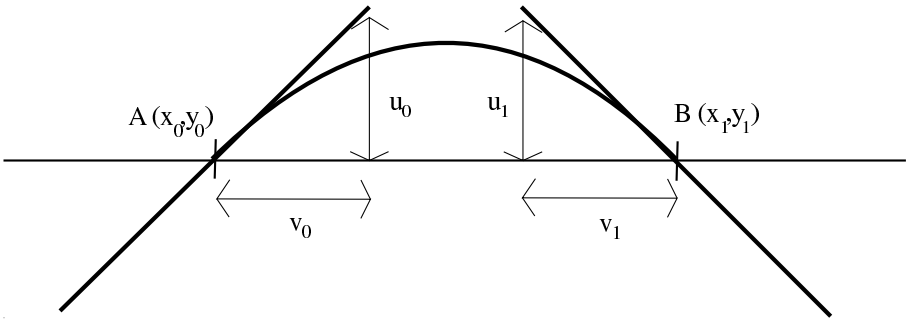

FIG. 11. Courbe de Bezier

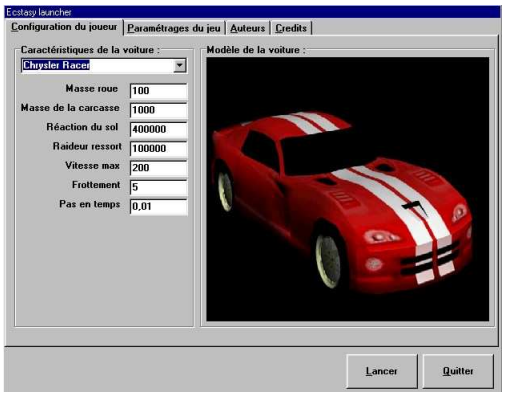

FIG. 12. Le menu d'Ecstasy

9.4. **Menu Delphi.** Lors de la deuxième soutenance, nous avons présenté le menu réalisé sous Delphi. La fenêtre comporte quatre onglets.

- Dans le premier onglet, Une combobox permet de sélectionner la voiture du joueur. Le coloris de la carcasse est alors affiché, ainsi que ses paramètres (raideur du ressort, masses, etc).
- Dans le deuxième, le joueur peut modifier la résolution de l'écran, l'activation de la brume, des lumières, le choix entre le jour et la nuit, ainsi que la pente maximale des routes.
- Dans le troisième et quatrième onglet, s'affichent respectivement les crédits et la license.

9.5. **Sons et video.** L'utilisation de DirectSound, permet d'augmenter le réalisme du jeu. Il existe trois types de sons : – celui du joueur, – le bruit de fond de la ville, – bruits occtionnels.

9.6. **Installation.** L'utilisation d'Install Shield Express a permis de créer le fichier d'installation du jeu. Sur le Cdrom, on trouve, comme demandé, l'exécutable, les sources et les librairies, la page web, les rapports et l'aide.

10. PAGE WEB ET REMERCIEMENTS

10.1. **Page web.** On peut télécharger les rapports de la soutenace un à quatre, le cahier des charges ainsi que le projet (version normale et lite), sur le site : www.ifrance.com\anassk ou sur : www.epita.fr\~quadra_q.

10.2. **Remerciments.** Quentin remercie :
– son père, pour lui avoir expliquer le principe de la moindre action ;
– Le site mm2zone pour ses modéles de voitures librement téléchargeables.

Références

[1] http ://www.soton.ac.uk/ trawww/stardust

[2] V. Arnold, *Méthode mathématiques de la mécanique classique*, Mir, Moscou, 1974.

[3] P. Appel, *Traité de mécanique rationnelle*, Gauthier-Villars, 1921.

[4] M. Woo, J. Neider, T. Davis D. Shreiner *OpenGL*, Campus Press, 2000.

[5] Xavier Michelon, xavier@linuxgraphic.org, *OpenGL : Génération de terrain*, Linux Magazine N° 33 — 34.

[6] Xavier Michelon, xavier@linuxgraphic.org, *OpenGL : Génération de terrain*, Linux Magazine N° 33 — 34.

[7] Xavier Michelon, xavier@linuxgraphic.org, *OpenGL : Placage de textures*, Linux Magazine N° 31 — 32.

[8] Xavier Michelon, xavier@linuxgraphic.org, *OpenGL : Eclérage et matériaux*, Linux Magazine N° 30.

[9] Xavier Michelon, xavier@linuxgraphic.org, *OpenGL : Les transformations de visualisation*, Linux Magazine N° 29.

[10] Xavier Michelon, xavier@linuxgraphic.org, *OpenGL : Transformations*, Linux Magazine N° 28.

[11] Xavier Michelon, xavier@linuxgraphic.org, *Programmer avec OpenGL*, Linux Magazine N° 26 — 27.