Info-SUP : A2, Promo 2007

ECSTASY OF AGONY

Cahier des charges

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

ECSTASY OF AGONY

QUADRAT QUENTIN,
KADIRI ANASS

Table des matières

1. Introduction

Le titre de ce projet est très explicite : c'est un Doom-like en 3D, où des voitures combattent dans une ville. Mais voici l'histoire de *Ecstasy of Agony*. Dans un avenir plus ou moins lointain, des robots hostiles attaquent la Terre. La rebellion se met alors en place avec pour seuls témoins des maisons en ruines, de nombreuses carcasses de voitures calcinées et un paysage post-apocalyptique. Mais un groupe de rebels continuent la bataille...

2. Le fichier d'exportation ASCII de 3D Studio Max

2.1. **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.

2.2. **Définition d'objet de 3D 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 type 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 : $t_0 = (p_2, p_0, p_3)$.

2.3. **Définition d'un objet sous 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;
```

L'utilisation de liste chainées 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. Voir dessin.

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.

2.4. **Définition des textures et des matériaux de 3D 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
        }
}
```

On appelle, ici, un matériau Une texture

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'Materiau01'. 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.

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 $(0,0),(0,1),(1,1)$.

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.

2.5. **Les textures avec 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.

2.6. **Définition une caméra sous 3D Studio Max et OpenGL.** Voici comment est représentée une caméra cibleuse 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 on peut touner une caméra d'un angle quelconque sur l'axe du vecteur position–cible sans que l'image ne soit modifiée. Dans notre exemple, nous avons : $p = (100.0, 100.0, 100.0)$, $c = (1.0, 1.0, 1.0)$ et $o = (0.0, 1.0, 0.0)$.

3. Courbes de bezier

Il se peut que les courbes de bezier nous servent pour calculer les trajectoire d'une voiutre (sur des routes courbes, par exemple), ou pour créer un terrain. Dans le dernier cas, on utilisera des surfaces de bezier. Voici 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 $y(t)$.

4. Modélisation d'une dynamique simple de suspension de voiture

4.1. **Introduction sur le 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.

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

$$\delta\mathcal{A} = \int m_1\dot{x}_1\delta\dot{x}_1 + m_2\dot{x}_2\delta\dot{x}_2 - k(x_2 - x_1)(\delta x_2 - \delta x_1) + o(\|x_1 - x_2\|) \ ,$$

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 $m\vec{\gamma} = \vec{F}$.

4.2. **Etude d'une suspension de roue de voiture.** 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 1).

FIG. 1. Ordre de parcours

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 :

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

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

4.3. **Discrétisation de la trajectioire de la voiture.** Pour calculer les trajectoires des deux corps, nous pouvons approximer les équations différentielles (1) et (2) 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. Apprentissage de Delphi et OpenGl

5.1. **La syntaxe Pascal.** Pour commencer j'ai lu le livre L'intro développement Borland Delphi 6 qui m'a permis de maîtrisé l'environnement de développement intégré de Delphi. La maîtrise de Delphi qui utilise le langage de programmation Pascal orienté objet, m'a permis d'apprendre ce même langage pour ainsi pouvoir écrire de nouvelles fonctions et procédures pour la conception de mon programme. L'exemple realisé montre que j'ai appris comment utilisé des variables globales et des variables locales, la définition de constantes, la création d'un nouveau type, l'utilisation des boucles.

5.2. **Initialisation d'une fenêtre OpenGl et création d'une primitive.** Tout d'abord j'ai appris grâce au site web Nehe (nehe.gamedev.net) comment initialiser une fenêtre OpenGl, créer mon premier polygone en lui attribuant des couleurs ou en lui appliquant une texture, le déplacer sur un plan, le visualiser grâce à une caméra qui le suit, et finalement la détection des collisions. Pour initialiser une fenêtre OpenGl, il a fallu copier un code source télécharger sur le site même, qui permet a lui tout seul de le faire. Il a fallu néanmoins comprendre quelques notions essentielles pour la suite. Ensuite pour la création de ma première primitive, il a fallu comprendre ces quelques fonctions dans l'exemple suivant :

```
glClear(GL_COLOR_BUFFER_BIT Or GL_DEPTH_BUFFER_BIT);
glClearColor(0.0,0.0,0.0,0.0);

glMatrixMode(GL_MODELVIEW); glLoadIdentity;

gluLookAt( 0.0, 0.0, 7.0,
           0.0, 0.0, 0.0,
           0.0, 1.0, 0.0 );

glBegin( GL_TRIANGLES );
    glColor3f( 1.0, 1.0, 1.0 );
        glVertex2d( 2.0, -2.0 );
    glColor3f( 0.5, 0.5, 0.0 );
        glVertex2d( -2.0, -2.0 );
    glColor3f( 0.0, 0.5, 0.5 );
        glVertex2d( 0.0, 2.0 );
glEnd();

glClear(GL_COLOR_BUFFER_BIT Or GL_DEPTH_BUFFER_BIT);
```

Ici on vide le tampon des valeurs chronomatiques, autrement dit, où les couleurs sont logées. La couleur de vidage sera spécifiée à l'aide de la commande suivante glClearColor(...). Le deuxième attribut sert à vider le contenu du tampon de profondeur. Celui-ci à la tâche d'évaluer la distance entre chaque primitive. Si en rapport avec leur distance, une primitive en cache une autre alors celle étant cachée ne sera pas dessinée. Ceci à pour but encore de nous faire gagner de la vitesse et ajouter du réalisme à la scène.

```
glClearColor(0.0, 0.0, 0.0, 0.0);
```

Voilà la couleur de vidage. Elle se présente sous format RGBA donc (Rouge, Vert, Bleu, Alpha). Alpha est un taux de luminosité.

```
glMatrixMode(GL_MODELVIEW); glLoadIdentity;
```

La commande glMatrixMode spécifie la matrice active, trois paramètres sont possibles (GL_MODELVIEW, GL_PROJECTION, GL_TEXTURE. Ici la matrice de modélisation-visualisation à été choisie parce que nous nous préparons à construire nos primitives. Donc les appels de transformations successives affecteront la matrice de modélisation-visualisation. Et c'est aussi le temps maintenant d'effectuer sur nos primitives des rotations, des déplacements et des redimensions. La commande glLoadIndentity() ; vide la matrice active afin de prévoir les commandes de transformations suivantes.

```
gluLookAt( 0.0, 0.0, 7.0,
           0.0, 0.0, 0.0,
           0.0, 1.0, 0.0 );
```

Cette commande provient de la librairie GLU, elle permet de placer le spectateur en fonction de la scène à visualiser. Les trois premiers paramètres spécifie où est placer la caméra dans un univers 3D donc X, Y et Z. Les trois autres paramètres spécifie quel point la caméra fixe. Et les trois derniers paramètres spécifie la direction du Vecteur. Pour ce qui est des transformations il existe trois commandes que voici :

- glTranslate(X, Y, Z) Spécifie un déplacement dans la scène.
- glScale(X, Y, Z) Spécifie un changement d'échelle dans la scène.
- glRotate(X, Y, Z) Spécifie une rotation dans la scène.

```
glBegin( GL\_TRIANGLES ); / glEnd()
```

La commande glBegin(type) débute une liste de sommets décrivant une primitive géométrique. Comme vous le constaterez, nous allons construire un triangle. Le paramètre type peut prendre les différentes valeurs, dont les plus utilisées sont :

- GL_POINTS : Points isolés.
- GL_QUADS : Regroupe par quatre les sommets interprétés comme des polygones à quatre côtés.
- GL_POLYGON : Contour externe d'un polygone convexe simple.

```
glColor3f( 1.0, 1.0, 1.0 );
```

Spécifie la couleur des sommets. Remarquer ici la dernière lettre de cette commande, f et le nombre 3 si vous vous référez au tableau (début du premier tutorial) vous remarquerez que la couleur est donnée sous format (Rouge, Vert, Bleu) et le type de donnée pour les couleur sera float (Entre 0.0 et 1.0). Tant qu'il n'y aura pas une nouvelle commande de sélection de couleur, les sommet seront tous blanc (1.0, 1.0, 1.0).

```
glVertex2d( 2.0, -2.0 );
```

Encore une fois regarder le nombre 2 et la dernière lettre de la fonction d. Celle-ci dessine positionne un sommet à l'endroit voulu. Ici l'endroit est spécifié par glVertex2d(X, Y). Ce sommet aura la couleur spécifiée par la commande glColor*. Les sommet sont généralement donnée sous le format : glVertex3d(X, Y, Z).

Et voici un exemple de code définissant le champ de vision en fonction de la dimension du contrôle fenêtré.

```
glViewport(0, 0, ClientWidth, ClientHeight );
glMatrixMode(GL_PROJECTION );
glLoadIdentity;

gluPerspective( 45, Width / Height, 0.1, 10.0 );
glMatrixMode( GL_MODELVIEW );
Invalidate;
glViewport(0, 0, ClientWidth, ClientHeight);
```

Spécifie l'espace disponible sur le composant où OpenGL dessine. Les deux paramètres suivants spcéfient les coodonnées de la fenêtre située en haut à gauche donc (0, 0) . Les deux derniers paramètres spécifient la grandeur de la fenêtre donc en bas à droite (Largeur, Hauteur). Cette commande est appelée ici parce que si la dimension de la fenêtre change alors OpenGL se résignera à dessiner sa future scène dans cet enclos.

`glMatrixMode( GL\_PROJECTION ); / glLoadIdentity();`

Active la matrice de projection car nous allons défénir un champ de vision à la ligne suivantes. La commande glLoadIdentity() sert à initialiser la matrice de projection, car c'est elle qui est active maintenant.

`gluPerspective( 45, Width / Height, 0.0, 10.0 );`

Créer une nouvelle matrice de projection permettant de créer un champ de vision pour notre scène. Quatre paramètres sont nécessaires :

- Angles de vue
  Spécifie l'angle du champ de vue, ici nous lui attribuons un angle de 45. Sa valeur doit être comprise entre 0 et 180
- Ratio
  Représente l'aspect du ratio, donc la largeur divisé par la hauteur. Si par exemple nous lui aurions spécifié un ratio de 0.5 qui est égale à puissance 1/2, donc la largeur serait 2 fois moins dense.
- Proche
  La distance la plus proche du champ de vision, en pronfondeur, est (0.0).
- Loin
  La distance la plus éloignée donc 10.0. Si nous dessinons une primitives à 11.0 de profondeur et plus il nous serait impossible de l'aperçevoir car elle ne ferait pas partie de notre champ de vision.
  Il existe d'autres types de perspective comme glFrustum, glOrtho.
- Invalidate ;
  Cette commande sert à obliger le rafraîchissement d'un contrôle fenêtré. Par conséquent s'il est rafraîchi il est aussi rapaint et ses composants enfant aussi.

5.3. **Application d'une texture à une primitive.** De la même manière qu'une liste d'affichage, la texture doit être représenter par un numéro

d'identification et dont celui-ci sera nommé avec la commande glGenTextures(). Autre chose très importante : Vous devez absolument préciser le nombre de textures qui seront générées dès l'appel de cette commande car ensuite vous ne pourrez revenir sur vos pas à moins de supprimer ceux déjà existante.

```
glGenTextures( 1, @IDTexture );
```

Ici nous avons demander à OpenGL de nous créer une texture sous l'index 1. Le nom de la texture est ainsi retourner dans la variable IDTexture qui est de type Integer. Le premier paramètre spécifie l'index d'une texture et l'autre est un paramètre de type pointer par lequel cette commande renvoi le nom (ID) de la texture créée. Si vous n'avez plus besoin de la texture alors supprimez la à l'aide le la commande glDeleteTextures(1, @IDTexture) ;

Ensuite la commande glBindTexture() rend cette texture active. Donc si le texturage est activé alors toutes les primitives dessiner seront recouvert par la texture numéro "1" ou sinon celle étant spécifier par glBindTexture.

```
glBindTexture( GL\_TEXTURE\_$2$D, IDTexture );
```

Le premier paramètre consiste à préciser quel type de texture nous désirons utiliser. Ici nous avons choisi une texture 2 dimension mais cela aurait pu être GL_TEXTURE_1D ou GL_TEXTURE_3D. Le deuxième paramètre spécifie le nom (ID) de la texture. Si nous faisons appel à un nom de texture déjà existant alors la texture ne sera pas recrée mais plutôt sélectionner en mémoire comme étant la texture active. Par conséquent si vous désirez savoir si un ID possède déjà une texture alors servez-vous de cette fonction glIsTexture(ID) ;, elle retournera un Vrai ou Faux déterminant si ce ID est déjà occupé par une texture ou pas.

Précisez le mode d'application de la textue pour chaque pixel

Pour spéficier la manière dont la texture sera répétée et combiner aux pixels déjà présente dans le tampon chronomatique, nous ferons appel à cette commande :

```
glTexParameter*( Target :TGLenum;
                 Pname  :TGLenum
                 Param  :PGLfloat );
```

Il reste cette avant dernière commande qui spécifie le mode de combinaison que nous désirons effectuer entre le tampon chronomatique et les couleurs de la texture.

```
glTexEnv*( target :TGLenum;
           Pname  :TGLenum;
           param  :PGLfloat );
```

Voici la dernière commande qui est la plus importante :

```
glTexImage*( Target :TGLenum;
             Level          :TGLint;
             InternalFormat :TGLint;
             Width          :TGLsizei;
             Height         :TGLsizei;
             Border         :TGLint;
             Format         :TGLenum;
             Textel         :Pointer );
```

Cette commande sert à définir une textures soit à $1, 2$ ou 3 dimensions. Voici le résumé des paramètres de celle-ci :

- Target Ce paramètres peut prendre une des trois valeurs suivantes : GL_TEXTURE_1D, GL_TEXTURE_2D OU GL_TEXTURE_3D.
- Level Spécifie le taux de résolution.
- InternalFormat Type de format de données dans l'image. Les plus utilisées sont : GL_ALPHA, GL_RGB, GL_RGBA
- Width Largeur de l'image servant en guise de texture. A noter que le largeur et hauteur doivent être une puissance de 2.
- Height Hauteur de l'image servant en guise de texture. A noter que le largeur et hauteur doivent être une puissance de 2.
- Border Est-ce que la texture nécessite une bordure, si oui alors cela agira aussi sur la largeur et hauteur qui devront alors êtres sous cette forme : $2^x + 2b$ et $2^y + 2b$. D'où $2b$ représente les bordures de la texture.
- Format Type de format de données contenu dans l'image représentant la texture. La valeur de ce paramètre est souvent similaire à celui de InternalFormat.
- Texel Pointeur sur les données de l'image qui formeront la texture.

5.4. **Détéction des collisions.** On a utilisé une matrice $(4, n)$ sur un plan 2D où n est le nombre de carrés (obstacles) et 4 les 4 sommets du carré. Ainsi les détections de collisions se font en 2D, c'est-à-dire qu'ils se font sur les paramètres x et y. On a donné ensuite de la hauteur à ces carrés pour en faire des cubes. Si l'objet (représentant la voiture) se trouve à l'intérieur d'un carré, et si l'objet est rentré en collision par la face gauche ou droite du carré, on redonne l'ancienne valeur à la position x de la voiture, et si celle-ci se fait par le haut ou le bas du carré, on redonne l'ancienne valeur à la position y de la voiture. Ce test est effectué avant l'affichage du déplacement de la voiture, pour ainsi prévoir les éventuelles collisions et bloquer la voiture face à l'obstacle présent. On a fait le même test de collision pour la caméra, pour éviter qu'elle traverse les obstacles.

6. Conclusion

Pour cette première soutenance, nous sommes parvenus à tenir les objectifs que nous nous somme fixés dans notre cahier des charges. Nous avons appris à déplacer un objet, une caméra dans un espace 3D. Nous avons réussi à gérer des collisions simples. Le loader ase termin{'e, nous permettra de charger les immeubles et les voitures que l'ont aura créé sous 3D Studio Max. Pour la prochaine soutenance, nous cr''eerons notre univers 3D et nous y ferons circuler les voitures selon la dynamique des pneus dejá créés.