Projet de Visualisation
Master 2
Ingénierie Mathématique

Réalisé par:

Ghilaine ABO EL EINEIN
Amine HARZLI
Philippe TIGREAT

Enseignants :

Vincent FEUVRIER
Pierre PANSU

Sommaire

L'objectif du projet a été de réaliser, en parallèle avec trois étudiants de Centrale en option automatisme, un travail permettant de recréer un dôme d'immersion en réalité virtuelle.
Le partage des tâches est ainsi fait : la partie des centraliens (le serveur) porte essentiellement sur des aspects pratiques (capteurs, écran, projecteur, trouver un financement …), tandis que la notre (le client) se charge du programme de visualisation qui commande l'affichage du dôme. Cette dernière, réalisée à l'aide des connaissances acquises en cours et codée en C++ pour un souci de portabilité, consiste à afficher un labyrinthe dans lequel une personne, placée au centre du dôme créé par la partie serveur, aurait l'impression d'être totalement immergée.
Le client a la possibilité de choisir ce qui sera affiché (le labyrinthe dans notre cas) ainsi que les principales données nécessaires à sa caractérisation. Le serveur, interagissant continuellement avec le client, recueille alors ces informations afin de simuler l'immersion.

1) Classes utilisées pour la génération du labyrinthe

Tout au long de ce projet, plusieurs classes ou fichiers, préalablement fournis par notre enseignant, ont été utilisés afin de mener à bien notre travail.

Tout d'abord le fichier common.h regroupe tous les types dont nous pouvions avoir besoin (point,rect, vect, plane, triangle, quad…) ainsi que les opérateurs permettant de les manipuler facilement(produit scalaire, produit vectoriel…).
Ensuite, world.h contient une classe abstraite world supposée modéliser un univers de réalité virtuelle, avec des méthodes draw et trace. De plus, engine.h contient la classe engine, qui permet de faire fonctionner une classe world.

a) La classe matrix :

Il nous a donc été demandé de créer une classe Matrix caractérisée entre autres par son nombre de lignes (Height), son nombre de colonnes (Width), et par plusieurs fonctions permettant notamment de remplir une matrice avec la même valeur, de récupérer ou de fixer le nombre de lignes ou de colonnes (respectivement getWidth, getHeight, setWidth et setHeight), ou encore de surcharger les opérateurs parenthèses et crochets pour accéder à un élément de la matrice à l'aide respectivement de deux entiers (opérateur parenthèses) ou d'un point (opérateur crochet) :

```
int getWidth(void) const {
  return Width;
}

void setWidth(const int width){
  resize(width,Height);
}
```

Resize étant une fonction déclarée dans la classe Matrix également, et permettant de redimensionner une matrice (à noter que la fonction resize fait intervenir deux fonctions, allocate et unallocate définies précédemment dans la même classe).
Les fonctions getHeight et setHeight se construisent de la même manière.

Surcharge des opérateurs ( ) et [ ] :

```
int& operator () (const int x,const int y) const {
  if (checkRange(x,y))
    return Coeffs[y][x];
  else
    return Dummy=0;
}



int& operator [] (const point p) const {
  if (checkRange(p.X,p.Y))
    return Coeffs[p.Y][p.X];
  else
    return Dummy=0;
}
```

La fonction checkRange également codée dans cette classe Matrix a pour but de vérifier que l'on essaie d'accéder à une valeur qui existe vraiment et que l'on ne sort pas des bornes définies par la Matrix :

```
const bool checkRange(const int x,const int y) const {
  return x>=0 && x<Width && y>=0 && y<Height;
}
```

Viennent alors s'ajouter les méthodes saveToFile et loadFromFile pour écrire et lire les données depuis un fichier, ainsi qu'un opérateur << pour afficher une version dans le terminal.

b) La classe laby :

Il a été nécessaire de créer une classe Laby héritant des données des classes Matrix et World et possédant en plus les champs Entry et Exit de type point (représentant respectivement l'entrée et la sortie du labyrinthe considéré).

i. La méthode solve

Le travail que nous avons eu à fournir pour cette classe a été de créer une fonction solve, permettant de construire une matrice de la même dimension que le labyrinthe étudié et de la remplir de manière à obtenir les distances de graphe à l'entrée, et renvoyer la distance entre la sortie et l'entrée.
Pour cela, il a été préférable de définir au préalable une variable Stack afin de stocker les différents voisins de chaque case étudiée de la matrice :

```
typedef struct{
  point Elements[10000];
  int count;
}Stack;
```

La taille du tableau de points Elements a été fixée à 10 000 afin de ne pas prendre le risque d'allouer plus de mémoire sachant que nous n'aurons sûrement jamais besoin de plus d'éléments.
L'algorithme général de cette méthode solve consiste a initialiser tous les éléments de la matrice rentrée en paramètre à un nombre très élevé proche de l'infini (INT_MAX en C++) à l'exception de la case de départ fixée à 0 (et des cases où se trouvent les murs), pour ensuite parcourir une par une les cases de la matrice et comparer la valeur contenue à celle des cases voisines. Pour cela, on utilise deux Stacks, slast contenant les cases courantes et snew contenant les voisins de ces cases.
Pour chaque élément de slast, on étudie les valeurs des cases voisines et si la valeur de la case courante est plus petite que celle de la case voisine, on augmente cette dernière valeur de 1. Une fois tous les éléments de slast parcourus, on intervertit les deux listes de cases snew et slast à l'aide d'une Stack intermédiaire, afin de pouvoir recommencer le travail avec les éléments de snew (les cases voisines vont alors devenir les cases courantes dont il faudra étudier les voisines et ainsi de suite). Cela se fera tant qu'il restera au moins un élément dans la pile slast. Ce qui donne de manière plus explicite :

```
int solve(matrix &M){
  M.fill(INT_MAX);                    // on la remplit avec des infinis sauf la premiere case mise a 0
  M[Entry]=0;
  static Stack T[2];                  // Tableau de deux pointeurs vers un Stack
  Stack *slast;                       // Stack contenant les cases courantes
  Stack *snew;                        // Stack contenant les voisins de ces cases
  slast=T;                            // slast pointe vers le premier Stack
  snew=T+1;                           // snew pointe vers le deuxieme Stack
  slast->count=1;                     // au debut, slast contient une case courante
  slast->Elements[0]=Entry;
  int dist=1;
  do {
       snew->count=0;       // on remet a zero la nouvelle liste des voisins.
       for(int i=0;i<slast->count;++i){       //parcourt des elements de slast
            const point p=slast->Elements[i];
            const int delta[4][2]={{-1,0},{1,0},{0,-1},{0,1}};
            for(int j=0;j<4;++j){       //parcours des elements de delta
                const point q={p.X+delta[j][0],p.Y+delta[j][1]};

            // tests sur les voisins de chaque elements de slast
                if((*this)[q] && M[q]>dist){
                  M[q]=dist;
                  snew->Elements[snew->count]=q;  // on rajoute l'element a la liste des voisins snew
                  snew->count++;                  // on augmente alors la taille de snew
               }
            }

       }

  Stack *sinter;       // variable intermediaire permettant de permutter slast et snew
  sinter=slast;        // slast (liste des cases courantes) va alors contenir les voisins trouves
  slast=snew;          // precedemment sur lesquels on va effectuer les memes operations.
  snew=sinter;
  ++dist;
  }
  while(slast->count && M[Exit]==INT_MAX);   // boucle a faire tant qu'il reste un element dans slast
  return (M[Exit]==INT_MAX)?-1:M[Exit];      // (tant qu'il reste des elements a modifier)
}
```

ii. La méthode generate

La fonction solve expliquée précédemment est utilisée pour un test de réalisabilité à l'intérieur d'une fonction fournie par notre professeur et il s'agit d'une heuristique basée sur le voisinage dans ce cas le voisinage d'un labyrinthe laby est un labyriynthe labyNew pour qui il existe un unique couple(i,j) tel que laby(i,j)≠labyNew(i,j), cette methode fournit *rapidement* (en temps polynomial en la taille des données) une solution réalisable, pas nécessairement optimale et cela nous suffit vue la nature NP-Complet du problème.

Le principe est assez simple, après avoir choisi une entrée est une sortie on initialise tous les éléments de la matrice à 1 ainsi qu'un indice de distance maxi DistMax à 0 et une matrice LabyBest à une matrice de même taille que le labyrinthe, remplie de 1. L'algorithme se présente comme une procédure itérative qui répète N fois la procédure suivante :

Si la distance(fournie par solve) de la matrice courante dist≥0:
{
  //donc le labyinthe est résolevable
  Si Dist>DistMax alors:
  {
    //on a donc un labyrinthe plus complexe(le plus court chemin est plus grand que l'ancien).
    -Recopier le labyrinthe actuel dans LabyBest.
    -Stocker Dist dans DistMax(la plus grande distance).

    //la prochaine action est une perturbation de la solution actuelle est ainsi obtenir une solution voisine.
    -Choisir au hasard une case du labyrinthe distincte de la sortie et de l'entrée et qui vaut 1 et y mettre zéro.
  }

Sinon (si dist<0)
 {
    //même perturbation
     Choisir au hasard une case du labyrinthe distincte de la sortie et de l'entrée et qui vaut zéro
    (il y en a forcément au moins une, sinon le labyrinthe serait solvable) et y mettre 1.
  }

}
-Recalculer dist(en lançant la fonction solve.

Cette procédure non déterministe permet à la fin d'obtenir un labyrinthe le plus long que possible cette longueur est relative au N itérations et qui fournit par expérience d'assez bons résultats.

2) Génération des faces du labyrinthe, Amine Harzli

Cette partie consiste à fournir toutes les données essentielles pour la construction adéquate des faces qui constituent le labyrinthe.

a) Choix de la structure de données :

Le travail que l'on a fait a été bien encadré ainsi l'architecture et la structure nous a été imposée dés le départ par notre professeur suivant un cadre bien précis pour ne pas créer d'incompatibilités avec les centraliens et d'éviter d'éventuels problèmes entre nos méthodes.

Afin de traiter convenablement les faces qui seront manipuler par nous trois, on se doit de sauvegarder les champs qui décrivent ces faces et le cadre imposé la structure de données appelée quad (quadrilatères), la structure est la suivante :

```
typedef struct {
 vect Origin;
 base Base;
 float SizeU,SizeV; //les dimension de la face
  int Texture;
  float TextureCoord[2][2];
 vect boundMin,boundMax;
 float color[3];
} quad;
```

Le champ Origin représente le point d'origine de la face : c'est de la que le début de construction du quad est identifié ; suivi du champ Base qui lui regroupe deux vecteurs de la base orthonormée caractérisant la face et un vecteur normal à la face qui servira pour l'éclairage de Ghilaine et les tests de collisions de Philippe. Enfin un boundMin et un boundMax correspondant au bornes x,y,z minimaux et maximaux respectivement et qui serviront aussi au test de collisions.
Pour ce qui est de la texture qui recoit l'indice de la texture a utilisé (charger a partir d'un fichier) et textCoord représente les coordonnée sur lesquels la texture sera plaqué.

b) La construction des faces :

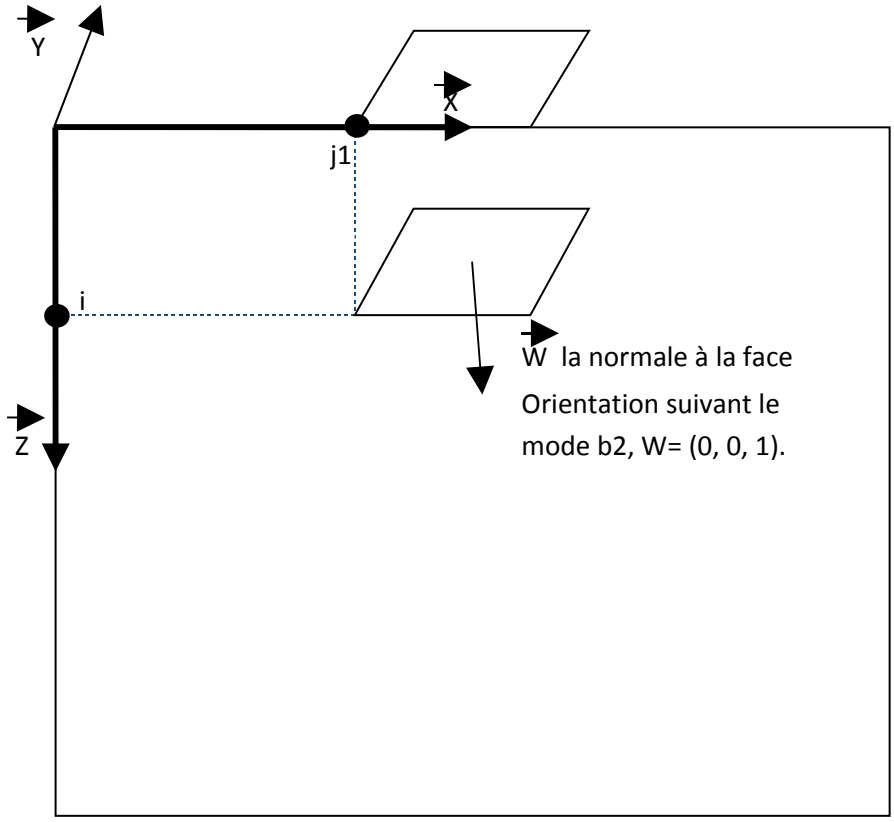
L'étape cruciale de cette partie est de trouver comment construire ces faces à partir de la matrice binaire générée précédemment ; pour cela on écrit la fonction « makeFaces » qui comprend deux boucles imbriquées pour le parcours par ligne et aussi deux autres pour le parcours en colonnes.

```
void makeFaces(void){
  int j1,j2;//coordonnées de départ
  int coeff1,coeff2;//variables de stockage
  bool b1,b2;//mode b1 ou b2(orientation de la normale)
```

Apres la déclaration des variables utilisées on parcourt toutes les lignes dans boucle qui suit et on trouvera les faces orientées horizontalement c'est-à-dire les x positifs.

```
 for(int i=0;i<matrix::Height+1;i++){
    b1=false;b2=false;j1=0;j2=0;
    for(int j=0;j<matrix::Width+1;j++){
      coeff1=(*this)(i-1,j);//la ligne de dessus
      coeff2=(*this)(i,j);//la ligne courante
```

Les instructions situées plus bas complètent cette partie du code et la même chose est faite pour les faces verticales sauf que dans ce cas on ferra un parcours colonne par colonne en deux boucles imbriquées.



Dans cette partie du code on fournit la coordonnée suivant l'axe x de l'origine du quadrilatère **(1)**, la même coordonné j **(2)** du sommet de fin de traitement et la coordonnée i suivant l'axe z du sommet d'origine **(2)** (voir figure précédente). La suite du code utilisé est la suivante :

```
if (coeff2==1 && coeff1==0){
        if (!b2){
          b2=true;j2=j;
        }
      } else {
        if (b2){
          makeHorizontalQuad(j2,j,i,true);
          b2=false;
        }
      }
      if (coeff2==0 && coeff1==1){
        if (!b1){
          b1=true;j1=j;
        }
      } else {
        if (b1){
          makeHorizontalQuad(j1,j,i,false);
          b1=false;
        }
      }
    }
  }
```

Le code qui précède contient deux blocs de condition une qui a pour première condition (coeff2==1 && coeff1==0)et l'autre bloc (coeff2==0 && coeff1==1) : ceci vérifie l'existence d'un mur, ces deux blocs contiennent des conditions sur les modes b1,b2, c-à-d si on change de mode quadrilatère est construit et on oriente la normale selon le mode, par exemple si on prend les quadrilatère horizontaux si on est on mode b2 on affecte true à la variable positive de la fonction makeHorizontalQuad(int x1,int x2,int y,bool positive) et ainsi on affecte a la normale sortante du quadrilatère le vecteur (0,0,1), pour le mode b1 la variable positive recevra false est la normale sera égal a (0,0,-1),

Les fonction makeHorizontalQuad et makeVerticalQuad servent à affecter les valeurs aux champs des quads horizontaux (suivant l'axe X) et les quads verticaux respectivement puis stocké dans un conteneur vector de la STL(Standard Temlpate Library).

```
int WallTexIndex;//indice de texture

void makeHorizontalQuad(int x1,int x2,int y,bool positive){
  if (!positive){
    quad q={{x1*LARGEUR,0,y*LARGEUR},{VX,VY,-VZ},(x2-
x1)*LARGEUR….  ,HAUTEUR,WallTexIndex,{{0,0},{x2-x1,1}},Vect(x1*LARGEUR,
0,y*LARGEUR),… Vect(x2*LARGEUR,HAUTEUR,y*LARGEUR)};
    Faces.push_back(q);
  } else {
    quad q={{x2*LARGEUR,0,y*LARGEUR},{-VX,VY,VZ},(x2-x1)*LARGEUR,…
HAUTEUR,WallTexIndex,{{x2-x1,0},{0,1}},…. Vect(x1*LARGEUR,
0,y*LARGEUR),Vect(x2*LARGEUR,HAUTEUR,y*LARGEUR)};….
    Faces.push_back(q);
  }
}

void makeVerticalQuad(int x1,int x2,int y,bool positive){
  if (positive){
    quad q={{y*LARGEUR,0,x1*LARGEUR},{VZ,VY,VX},(x2-
x1)*LARGEUR,HAUTEUR,WallTexIndex,{{0,0},{x2-x1,1}},Vect(y*LARGEUR,
0,x1*LARGEUR),Vect(y*LARGEUR,HAUTEUR,x2*LARGEUR)};
    Faces.push_back(q);
  } else {
    quad q={{y*LARGEUR,0,x2*LARGEUR},{-VZ,VY,-VX},(x2-x1)*LARGEUR,…
HAUTEUR,WallTexIndex,{{x2-x1,0},{0,1}},… Vect(y*LARGEUR,
0,x1*LARGEUR),Vect(y*LARGEUR,HAUTEUR,x2*LARGEUR)};
    Faces.push_back(q);
  }
}
```

c) Paramétrage de l'état du monde :

Avant la construction des faces, des constantes qui ont des valeurs logiques et sans contradiction entre elles, doivent être définis à travers la fonction initialize ; tout d'abord il nous faut un grand quad de avec les dimensions des coté [Width*Largeur, Height*Largeur] (générée elle aussi dans cette fonction) qui représente le sol. Dans cette meme fonction on décide aussi des textures à charger pour les murs et le sol.

Ensuite viennent les constantes tel que le rayon de contact Radius=1 qui doit être raisonnablement petit devant larg la largeur du plus petit couloir afin de ne pas donner l'impression d'être arrêté avant le mur, ainsi qu'une vitesse=5 pas trop élevée par rapport à la taille du monde, et une gravité égale à 10 sans spécifier le coefficient de friction à qui on affecte la valeur 0 car notre labyrinthe est dépourvu de pentes.

3) Représentation des faces du labyrinthe, Ghilaine Abo El Einein

a) Première approche : affichage de quads :

L'objectif de cette partie est de représenter les différents murs (ou faces) du labyrinthe à l'aide de la liste des quads fournie par les fonctions précédemment décrites (le type quad étant défini dans le fichier common.h dont nous nous sommes servis tout au long de ce projet).

Dans un premier temps, une fonction draw simple a été créée dans cet unique but sans se soucier de certaines contraintes telle l'éclairement. Afin d'alléger les notations et donc de simplifier la lecture du code, deux fonctions ont été mises en place :
Une première fonction quadVect définie de la manière suivante :

```
vect quadVect(const quad &q, float x,float y){
     return q.Origin + x*q.SizeU*q.Base.U+y*q.SizeV*q.Base.V;
}
```

Cette fonction permet de déterminer les coordonnées des différents sommets d'un quad caractérisé notamment par un point représentant l'origine, une base sur laquelle sont décomposées ses différentes arrêtes et la taille de ces arrêtes.

Une seconde fonction permettant cette fois-ci l'affichage des sommets de chaque quad du labyrinthe a été utilisée :

```
void quadVertex(const quad &q, float x,float y){
      vect v=quadVect(q,x,y);
      glVertex3f(v.X,v.Y,v.Z);
}
```

La partie principale de la fonction draw alors écrite, consiste à parcourir la liste des murs du labyrinthe à représenter, à l'aide d'une boucle for incluse entre un glBegin(GL_QUADS) et un glEnd(), et d'afficher ainsi pour chacun de ces quads, les différents sommets les constituant en tenant compte de l'ordre dans lequel ils seront placés. C'est à ce niveau-là également, que la normale à chaque quad a été spécifiée. Il a fallu bien évidemment définir au préalable les caractéristiques de la fenêtre d'affichage et de la caméra ainsi que mettre à jour les matrices de projection et de modélisation. Tout ceci a pu se faire comme ci-dessous :

```
glViewport(viewPort.X,viewPort.Y,viewPort.Width,viewPort.Height);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(angle,ratio,ZMIN,ZMAX);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(eye.X,eye.Y,eye.Z,center.X,center.Y,center.Z,up.X,up.Y,up.Z);
glColor3f(1,1,1);
```

(ZMIN et ZMAX représentant respectivement la plus petite et la plus grande distance autorisée d'un point à la camera ; les autres variables intervenant dans les différentes fonctions décrites ci-dessus sont données en paramètres dans la fonction draw à coder).

b) Seconde approche : affichage de triangles :

La fonction draw alors finie peut être nettement améliorée, notamment concernant l'éclairement. En effet, la méthode de rendu des surfaces sujettes à un éclairement que propose OpenGl est basée sur l'interpolation des intensités lumineuses de chaque extrémité de la face considérée, et il est donc nécessaire de découper ces surfaces de manière à obtenir un rendu plus précis et réaliste de l'effet d'une source lumineuse (notamment dans le cas d'un effet de lampe torche). Naturellement, il faut s'arranger pour que seules les faces les plus proches de la caméra soient finement découpées, les faces très éloignées ne nécessitant pas de détails très précis.

Afin de réaliser ce travail, il a fallu tout d'abord introduire de nouvelles fonctions intermédiaires, dont la fonction drawTriangle permettant de tracer un triangle contenu dans le quad donné en paramètre. Une fonction récursive recDrawTriangle a également été créée et sera décrite un peu plus loin. De plus, il a été nécessaire de créer une variable Camera stockant les six plans de la pyramide de vision de la manière suivante :

```
struct {
  plane Planes[6];
  double Projection[16],Modelview[16];
  int Viewport[4];
 } Camera;
```

(le type plane ayant été défini dans le fichier common.h).

La fonction recDrawTriangle consiste à ne sélectionner que les parties d'un triangle qui sont contenues dans la pyramide de vision. En effet, elle permet de compter le nombre d'intersections du triangle donné en paramètre avec les différents plans de la pyramide de vision afin d'éliminer les parties qui ne sont pas visibles du point de vue de la caméra. Trois cas se présentent :

- le cas où le nombre d'intersections entre le triangle et l'un des plans de la pyramide de vision vaut 0 : le triangle entier est conservé.
- le cas où ce nombre vaut 1 : on coupe alors le quadrilatère restant en deux triangles.
- le cas où ce nombre vaut 2 : on ne tient compte que du triangle constitué du sommet restant et contenu dans le plan de la caméra.

Dans le cas où ce nombre vaut 3, le triangle est en dehors du champ de vision et il n'est alors pas tracé. Ce dernier cas n'est donc pas pris en compte.
Ensuite, après avoir fini de sélectionner les triangles à afficher (ceux contenus dans le plan de la caméra donc, suite au découpage précédemment décrit), on détermine pour chacun d'eux, le plus grand côté que l'on va diviser en 2 si ce dernier n'est pas plus petit qu'un certain seuil que l'on pose égal à 50 par exemple (en effet un triangle déjà très petit ne nécessite pas d'être découpé en deux ; c'est le cas des triangles très éloignés de la caméra). On appelle alors à nouveau la fonction recDrawTriangle mais cette fois-ci en l'appliquant aux 2 triangles ainsi obtenus et ainsi de suite.
Pour cette méthode, il a tout de même fallu faire attention à ne pas oublier de projeter les coordonnées des sommets de chaque triangle dans la fenêtre. Ce sont ces coordonnées obtenues grâce à la fonction gluProject, qui seront utilisées dans les calculs effectués dans la fonction recDrawTriangle. Pour faciliter la lecture du code ici aussi, une fonction projVect a été introduite :

```
vect projVect(const vect& v){
        double d1,d2,d3;
      gluProject(v.X,v.Y,v.Z,Camera.Modelview,Camera.Projection,Camera.View
port,&d1,&d2,&d3);
      return Vect(d1,d2,0);
}
```

Enfin, la fonction draw peut être modifiée en rajoutant la définition des six plans constituant la pyramide de vision de la manière suivante :

```
Camera.Planes[0]=Plane(b.W,eye,ZMIN);
Camera.Planes[1]=Plane(-b.W,eye,-1*ZMAX);
Camera.Planes[2]=Plane(b.W+1/tan(angle*PI/360)*b.V,eye);
Camera.Planes[3]=Plane(b.W-1/tan(angle*PI/360)*b.V,eye);
Camera.Planes[4]=Plane(b.W+1/tan(angle*PI/360)/ratio*b.U,eye);
Camera.Planes[5]=Plane(b.W-1/tan(angle*PI/360)/ratio*b.U,eye);
```

(avec b definie de la manière suivante :
base b=Base(center-eye,up);),

et une fois les matrices de projection et de modélisation et le viewport récupérés et modifiés à l'aide des fonctions glGetDoublev et glGetIntegerv, il suffit d'effectuer un affichage de triangles (toujours à l'aide d'une boucle for parcourant la liste de quads constituant le labyrinthe, mais incluse cette fois ci entre un glBegin(GL_TRIANGLES) et un glEnd() puisque l'on s'occupe de la représentation de triangles et non plus de quads).
On appelle alors récursivement deux fois la fonction recDrawTriangle, une première fois appliquée à T1 le premier triangle obtenu lorsque l'on coupe un quad selon l'une de ses diagonales, et une seconde fois appliquée à T2, le deuxième triangle issue du découpage en deux du quad donné en paramètre de la fonction.

```
glBegin(GL_TRIANGLES);
 for(std::vector<quad>::iterator i=Faces.begin();i!=Faces.end();++i){
  recDrawTriangle(*i,T1);
  recDrawTriangle(*i,T2);
}
glEnd();
```

T1 correspond a un demi rectangle coupé selon l'une de ces diagonales, T2 correspond au demi rectangle complémentaire. Faces est un champ de la classe laby utilisée au cours de ce projet, contenant la liste de quads du labyrinthe. Faces.begin() représente donc le premier élément de cette liste et Faces.end() le dernier.

Un travail supplémentaire consisterait à rajouter des textures aux faces du labyrinthe et engendrerait quelques petites modifications intervenant notamment dans la fonction quadVertex et la fonction principale draw (où il est necessaire d'activer les textures à l'aide de l'option : glEnable(GL_TEXTURE_2D);)

4) Tests de collision, Philippe Tigréat

Un aspect important de la simulation d'un labyrinthe en réalité virtuelle consiste à s'assurer que l'on ne puisse pas passer à travers les murs. Nous avons codé, pour satisfaire à cette contrainte, la fonction 'trace' qui suit. Celle-ci prend en argument les coordonnées de deux points dans le repère de l'espace, correspondant respectivement au point où l'on se trouve et à celui où l'on souhaite aller. Elle prend deux arguments supplémentaires optionnels, correspondant à deux adresses auxquelles on souhaiterait stocker la distance à la face la plus proche rencontrée par le vecteur déplacement, ainsi qu'un vecteur normal de celle-ci.
La tâche effectuée par la fonction consiste en fait à passer en revue une liste de quadrangles représentant des faces du labyrinthe et à déterminer si le vecteur déplacement les coupe ou non. La fonction renvoie un booléen, qui vaut 1 si au moins une face est rencontrée, et 0 sinon.
Le code suit donc la procédure suivante :

- itération sur toutes les faces du labyrinthe
- pour chaque face, on teste si le vecteur direction traverse le plan de cette face
- le cas échéant, on teste si cette intersection est contenue dans la face
- si tel est le cas, on teste si une précédente face intersectée n'est pas plus proche
- si ce n'est pas le cas, la distance à la face et son vecteur normal sont stockés

```
bool trace(const vect& from,const vect& direction,float* distance,vect*
normal){
        vect I;
        vect O;
        vect F;
        bool coupe=false; /* Par defaut, pas de collision detectee */
        float n=norm(direction),bestdistance;
        vect d=direction/n;
        for (std::vector<quad>::iterator i=Faces.begin();i!=Faces.end(); +
+i){
                float D1=(from-(*i).Origin)*(*i).Base.W;
                float D2=(from+direction-(*i).Origin)*(*i).Base.W;
                if ((D1>=0 && D2<0)){
```

//teste que le vecteur direction atteint le plan de la face :
//si les deux produits scalaires sont de signes opposés, le point de départ et le point d'arrivée sont de part et d'autre de la face.

```
                        float r=fabs(D1/(D1-D2));
                        I=from+direction*r;
                        F=I-(*i).Origin;
                        float A=(F*(*i).Base.U);
                        float B=(F*(*i).Base.V);
                        float a=(*i).SizeU;
                        float b=(*i).SizeV;
                        if (0<=A && A<=a && 0<=B && B<=b){
```

//teste que l'intersection avec le plan est dans la face :
//on calcule les coordonnées du point d'intersection dans un repère orthonormé formant un angle de la face. On vérifie qu'elles sont comprises dans les intervalles correspondant aux points de la face.

```
                                if (!coupe || r<bestdistance){
```

//teste si une precedente face intersectee serait plus proche

```
                                        bestdistance=r;
                                        if (normal) *normal=(*i).Base.W;
                                        coupe=true;
                                }
                        }
                }
```

```
        }
        if (distance && coupe) *distance=bestdistance;
        return coupe;
};
```