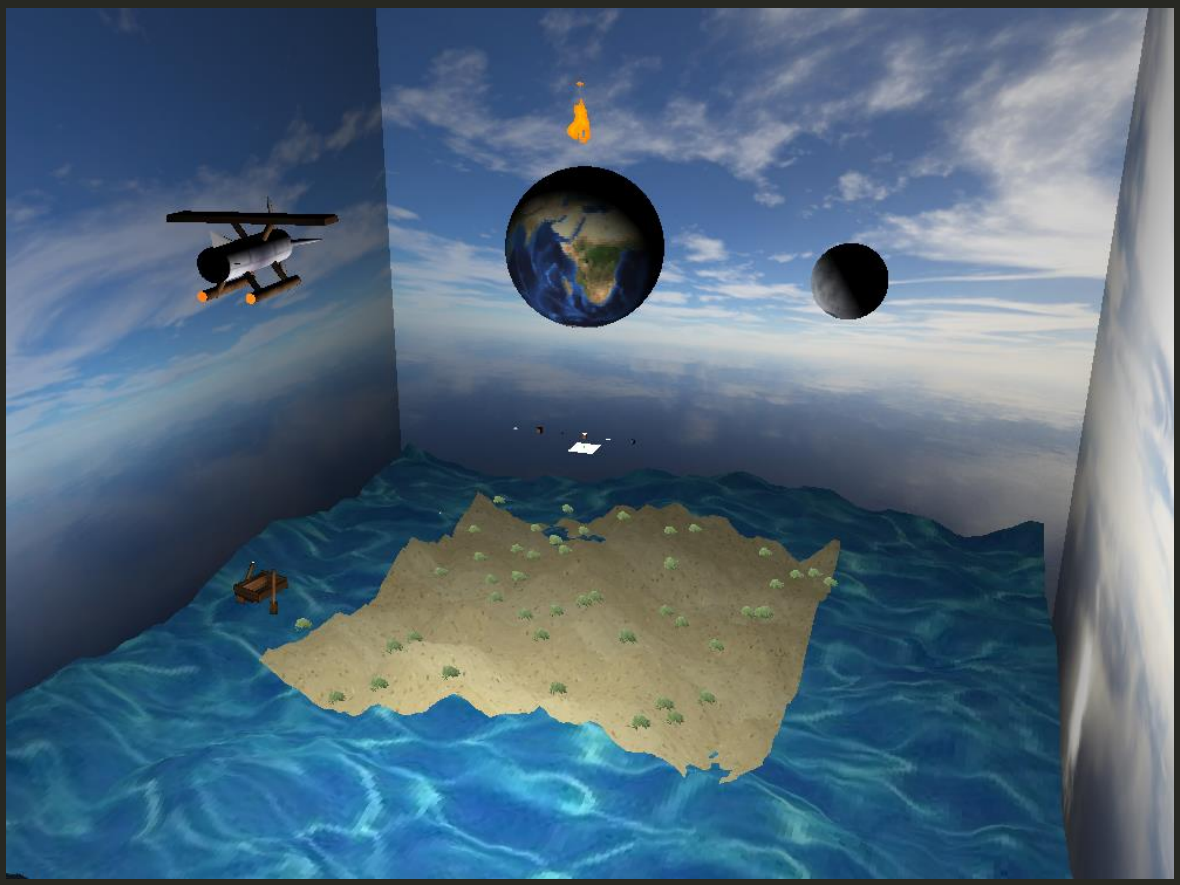
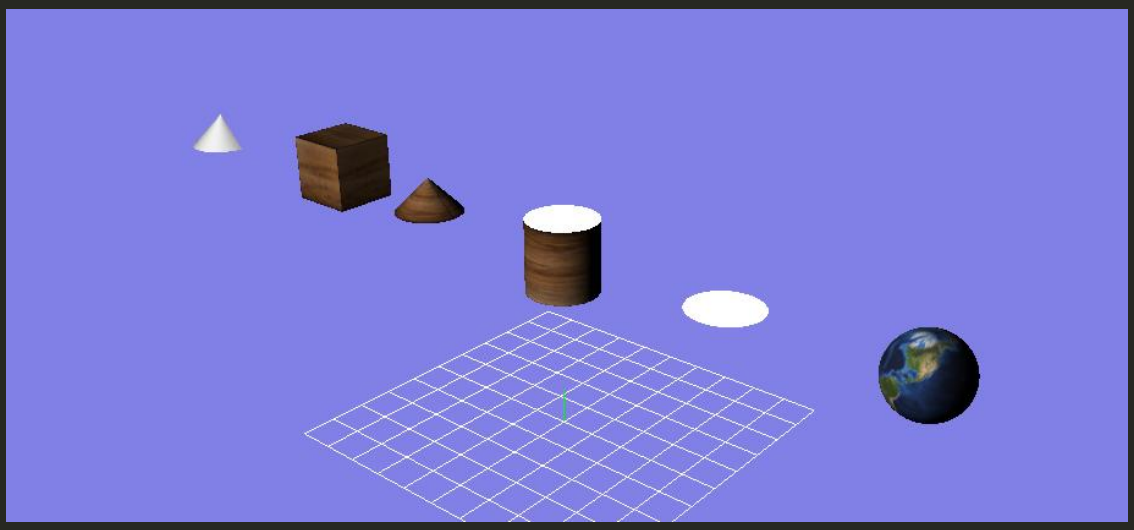
TOUIL Ines

L2 2021/2022

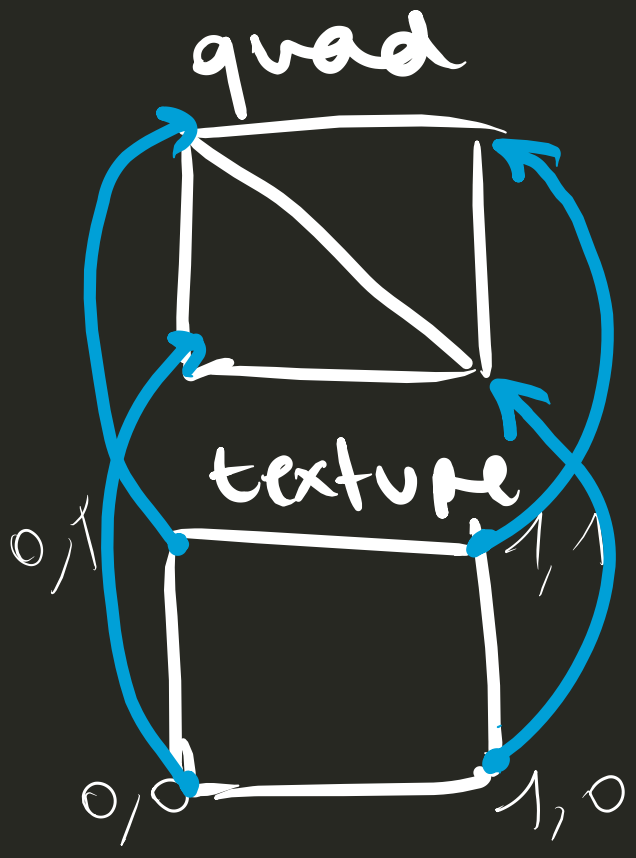
TP d’Introduction à l’Informatique Graphique



1. Manipulation des formes de base



Cube avec 6 quadrilatères :



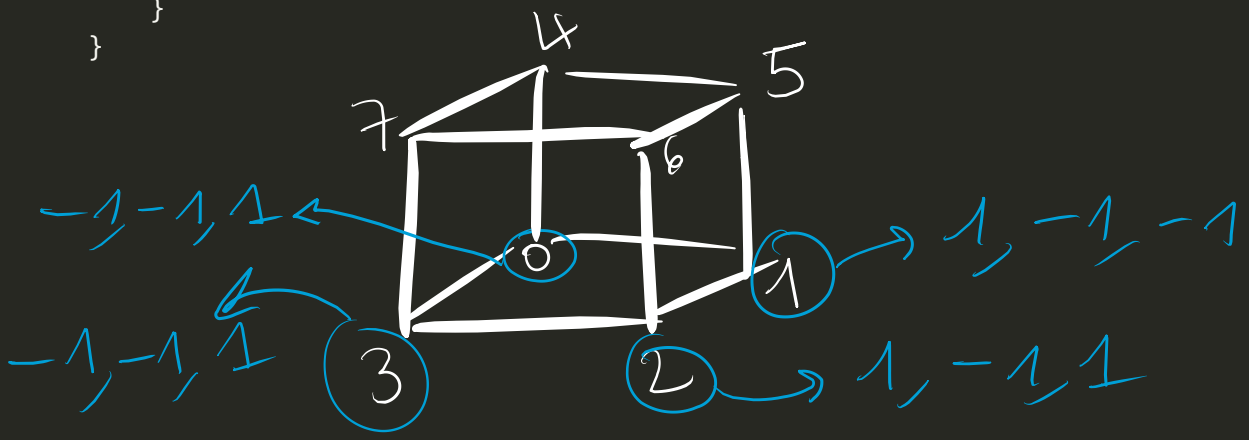

```
void ViewerEtudiant::init_quad(){
    //Une seule face

    m_quad = Mesh(GL_TRIANGLE_STRIP);

    m_quad.normal(0, 0, 1);

    m_quad.texcoord(0, 0);
    m_quad.vertex(-1, -1, 0);

    m_quad.texcoord(1, 0);
    m_quad.vertex(1, -1, 0);

    m_quad.texcoord(0, 1);
    m_quad.vertex( -1, 1, 0);

    m_quad.texcoord(1, 1);
    m_quad.vertex(1, 1, 0);
}

//J'applique des translations et rotations à chaque quad pour former un cube

void ViewerEtudiant::draw_cube_quad(const Transform& T, unsigned int tex){
//face avant
    gl.model(T);
```

```
    gl.texture(tex);
    gl.draw(m_quad);

//face arriere
    gl.model(T*Translation(0,0,-2)*RotationY(180));
    gl.texture(tex);
    gl.draw(m_quad);

//face droite
    gl.model(T*Translation(1,0,-1)*RotationY(90));
    gl.texture(tex);
    gl.draw(m_quad);

//face gauche
    gl.model(T*Translation(-1,0,-1)*RotationY(-90));
    gl.texture(tex);
    gl.draw(m_quad);

//haut
    gl.model(T*Translation(0,1,-1)*RotationX(-90));
    gl.texture(tex);
    gl.draw(m_quad);

//bas
    gl.model(T*Translation(0,-1,-1)*RotationX(90));
    gl.texture(tex);
    gl.draw(m_quad);
}
```

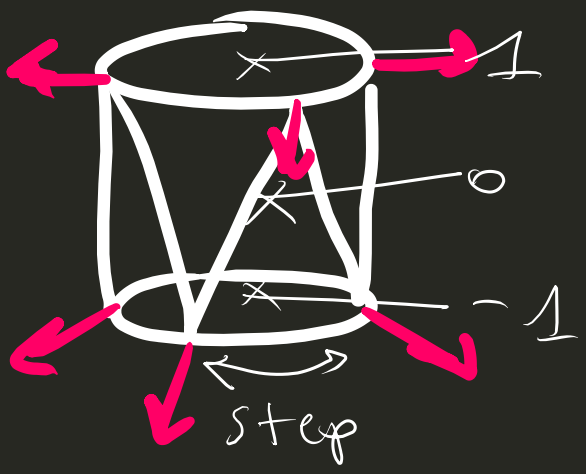
Cube avec une structure indexée :

```
//Je stock dans un tableau le numéro des sommets du cube pour récupérer les
coordonnées dans le tableau pt

void ViewerEtudiant::init_cube()
{
    //Sommets du cube
    static float pt[8][3] = { {-1,-1,-1}, {1,-1,-1}, {1,-1,1}, {-1,-1,1}, {-
1,1,-1}, {1,1,-1}, {1,1,1}, {-1,1,1} };
    //Faces du cube
    static int f[6][4] = { {0,1,2,3}, {5,4,7,6}, {2,1,5,6}, {0,3,7,4},
{3,2,6,7}, {1,0,4,5} };
    //Normales au cube
    static float n[6][3] = { {0,-1,0}, {0,1,0}, {1,0,0}, {-1,0,0}, {0,0,1},
{0,0,-1} };
```

```
    m_cube = Mesh(GL_TRIANGLE_STRIP);

    for (int i=0; i<6; i++)
    {
        m_cube.normal(n[i][0], n[i][1], n[i][2]);

        m_cube.texcoord(0,0);
        m_cube.vertex( pt[ f[i][0] ][0], pt[ f[i][0] ][1], pt[ f[i][0] ][2] );

        m_cube.texcoord(1,0);
        m_cube.vertex( pt[ f[i][1] ][0], pt[ f[i][1] ][1], pt[ f[i][1] ][2] );

        m_cube.texcoord(0,1);
        m_cube.vertex( pt[ f[i][3] ][0], pt[ f[i][3] ][1], pt[ f[i][3] ][2] );

        m_cube.texcoord(1,1);
        m_cube.vertex( pt[ f[i][2] ][0], pt[ f[i][2] ][1], pt[ f[i][2] ][2] );

        m_cube.restart_strip();
    }
}
```

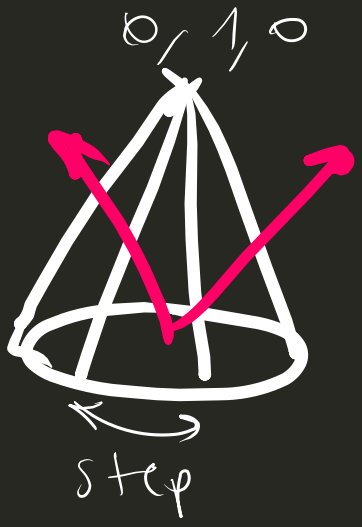


Cylindre :




```
void ViewerEtudiant::init_cylindre()
{
    const int div = 25;
    float step = 2.0 * M_PI / (div);

    m_cylindre = Mesh(GL_TRIANGLE_STRIP);

    for (int i=0; i<=div; i++)
    {
        float alpha = i * step;

        m_cylindre.normal( Vector(cos(alpha), 0, sin(alpha)) );
        m_cylindre.texcoord( float(i)/float(div), 1 );
        m_cylindre.vertex( Point(cos(alpha), -1, sin(alpha)) );

        m_cylindre.normal( Vector(cos(alpha), 0, sin(alpha)) );
        m_cylindre.texcoord( float(i)/float(div), 0 );
        m_cylindre.vertex( Point(cos(alpha), 1, sin(alpha)) );
    }
}
```

Cône :




```
void ViewerEtudiant::init_cone()
{
    const int div = 25;
    float step = 2.0 * M_PI / (div);
    m_cone = Mesh(GL_TRIANGLE_STRIP);

    for(int i=0; i<=div; i++)
    {
        float alpha = i * step;
        m_cone.normal(
            Vector( cos(alpha)/sqrtf(2.0),
                    1.0/sqrtf(2.0),
                    sin(alpha)/sqrtf(2.0) ));

        m_cone.vertex( Point( cos(alpha), 0, sin(alpha) ));

        m_cone.normal(
            Vector( cos(alpha)/sqrtf(2.0),
            1.0/sqrtf(2.0),
            sin(alpha)/sqrtf(2.0) ));
```

```
        m_cone.vertex( Point(0, 1, 0));
    }
}
```

Sphère :


```
void ViewerEtudiant::init_sphere()
{
    const int divBeta = 200;
    const int divAlpha = divBeta/2;
    float beta, alpha, alpha2;

    m_sphere = Mesh(GL_TRIANGLE_STRIP);

    for(int i=0; i<divAlpha; i++)
    {
        alpha= -0.5f * M_PI + float(i) * M_PI / divAlpha; // de -PI/2 à PI/2
        alpha2 = -0.5f * M_PI + float(i+1) * M_PI / divAlpha; // de -PI/2 à
PI/2 avec un décalage de 1 pour le maillage

        for (int j=0; j<=divBeta; j++)
        {
            beta = float(j) * 2.f * M_PI / (divBeta); // Un tour complet en
200 fois

            m_sphere.normal( Vector( cos(alpha)*cos(beta), sin(alpha),
cos(alpha)*sin(beta) ) );

            m_sphere.texcoord( beta/(2*M_PI), 0.5+(alpha/M_PI) );
            m_sphere.vertex( Point( cos(alpha)*cos(beta), sin(alpha),
cos(alpha)*sin(beta) ) );

            m_sphere.normal( Vector( cos(alpha2)*cos(beta), sin(alpha2),
cos(alpha2)*sin(beta) ) );

            m_sphere.texcoord(beta/(2*M_PI),0.5+(alpha/M_PI));
            m_sphere.vertex( Point( cos(alpha2)*cos(beta), sin(alpha2),
cos(alpha2)*sin(beta) ) );
        }
        m_sphere.restart_strip();
    }
}
```

2. Affichage à l'aide de transformations géométriques

Avion :

```
// Combinaison de plusieurs formes simples pour créer un objet complexe
void ViewerEtudiant::draw_avion(const Transform& T,const Transform& TH)
{
    gl.model(T);
    // Corps
    Transform Tcyl = T *
Translation(0,0,0)*Rotation(Vector(0,0,1),90)*Scale(1,4,1);
    draw_cylindre_ferme(Tcyl,m_avion_tex,m_avion_tex);


    // Hélices
    draw_sphere(T*Translation(4,0,0)*Scale(1.4,0.6,0.6),m_texture_cube);
    draw_sphere(T*TH*Translation(4,0,0)*RotationX(-
45)*Scale(0.2,3,0.2),m_avion_tex);
    draw_sphere(T*TH*Translation(4,0,0)*RotationX(45)*Scale(0.2,3,0.2),m_avion
_tex);


    // Pieds
    draw_cylindre(T*Translation(-1.5,-1,0.7)*RotationX(-
35)*Scale(0.2,1.5,0.2),m_texture_cube);
```

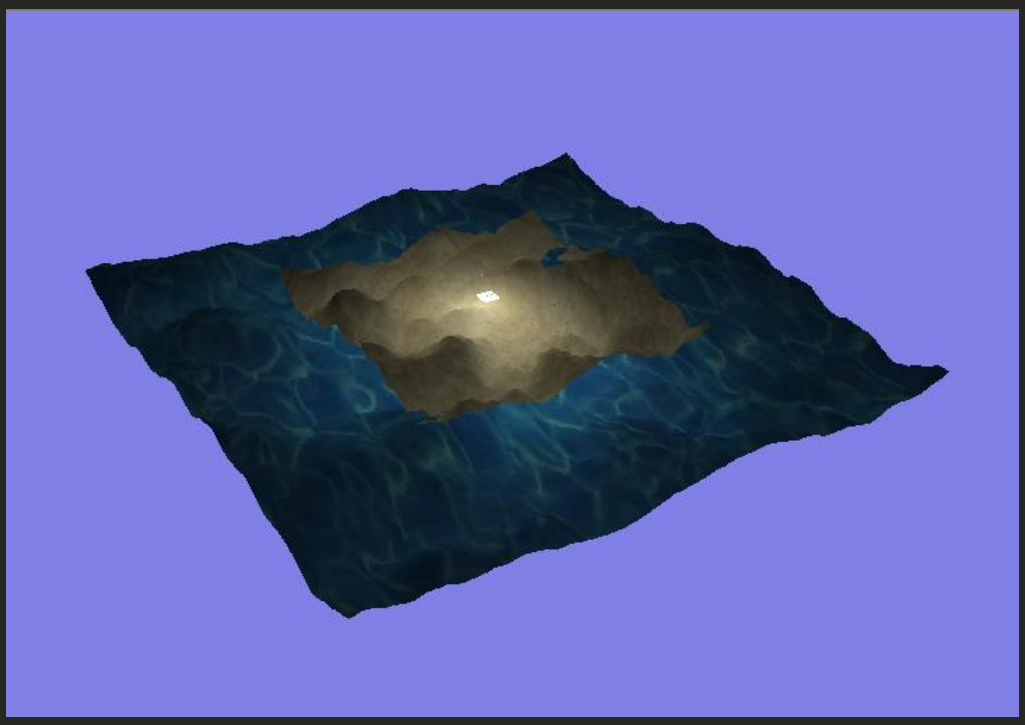
```
    draw_cylindre(T*Translation(-1.5,-1,-
0.7)*RotationX(35)*Scale(0.2,1.5,0.2),m_texture_cube);

    draw_cylindre(T*Translation(2,-1,0.7)*RotationX(-
35)*Scale(0.2,1.5,0.2),m_texture_cube);
    draw_cylindre(T*Translation(2,-1,-
0.7)*RotationX(35)*Scale(0.2,1.5,0.2),m_texture_cube);

    draw_cylindre_ferme(T*Translation(0,-
2,1.5)*RotationZ(90)*Scale(0.3,3,0.3),m_texture_cube,m_texture_cube);
    draw_cylindre_ferme(T*Translation(0,-2,-
1.5)*RotationZ(90)*Scale(0.3,3,0.3),m_texture_cube,m_texture_cube);


    // Toit et ailes
    draw_cylindre(T*Translation(1,1,0.7)*RotationX(35)*Scale(0.2,1.2,0.2),m_te
xture_cube);
    draw_cylindre(T*Translation(1,1,-0.7)*RotationX(-
35)*Scale(0.2,1.2,0.2),m_texture_cube);

    Transform Taile1 = T *
Translation(1,2,0)*Rotation(Vector(1,0,0),90)*Scale(1,5,0.2);
    gl.model(Taile1);
    gl.draw(m_cube);

    Transform Taile2 = T * Translation(1,2,0)*Rotation(Vector(-
1,0,0),90)*Scale(1,5,0.2);
    gl.model(Taile2);
    gl.draw(m_cube);

    Transform Taile3 = T * Translation(-4,0,0)*Rotation(Vector(0,0,1),-
90)*Scale(0.2,2,2);
    gl.model(Taile3);
    gl.draw(m_demi_cone);

    Transform Taile4 = T * Translation(-
4,0,0)*Rotation(Vector(0,1,0),90)*Rotation(Vector(0,0,1),90)*Rotation(Vector(1
,0,0),90)*Scale(0.2,2,2);
    gl.model(Taile4);
    gl.draw(m_demi_cone);

    Transform Tcone = T * Translation(-
4,0,0)*Rotation(Vector(0,1,0),90)*Scale(0.2,2,5);
    gl.model(Tcone);
    gl.draw(m_demi_cone);

}
```

Barque :

```
void ViewerEtudiant::draw_barque(const Transform& T, unsigned int tex)
{
    draw_extrusion(T*Translation(0,5,2)*RotationX(180)*Scale(1,1,4),tex);
    draw_extrusion(T*Translation(4,5,-
2)*RotationY(180)*RotationX(180)*Scale(1,1,4),tex);
    draw_cube(T*Translation(2,4.5,-2.1)*Scale(2,0.6,0.1),tex);
    draw_cube(T*Translation(2,4.5,2.1)*Scale(2,0.7,0.1),tex);

    // Rames
    draw_cylindre_ferme(T*Translation(-0.1,5,0)*RotationX(20)*RotationZ(-
13)*Scale(0.2,2,0.2),tex,tex);
    draw_cylindre_ferme(T*Translation(4.1,5,0)*RotationX(20)*RotationZ(13)*Sca
le(0.2,2,0.2),tex,tex);

    draw_cube(T*Translation(-0.6,2.8,-0.8)*RotationX(20)*RotationZ(-
13)*Scale(0.5,0.7,0.1),tex);
    draw_cube(T*Translation(4.6,2.8,-
0.8)*RotationX(20)*RotationY(13)*Scale(0.5,0.7,0.1),tex);
}
```

3. Terrain, texture, billboard (arbre) et cubemap

Terrain :

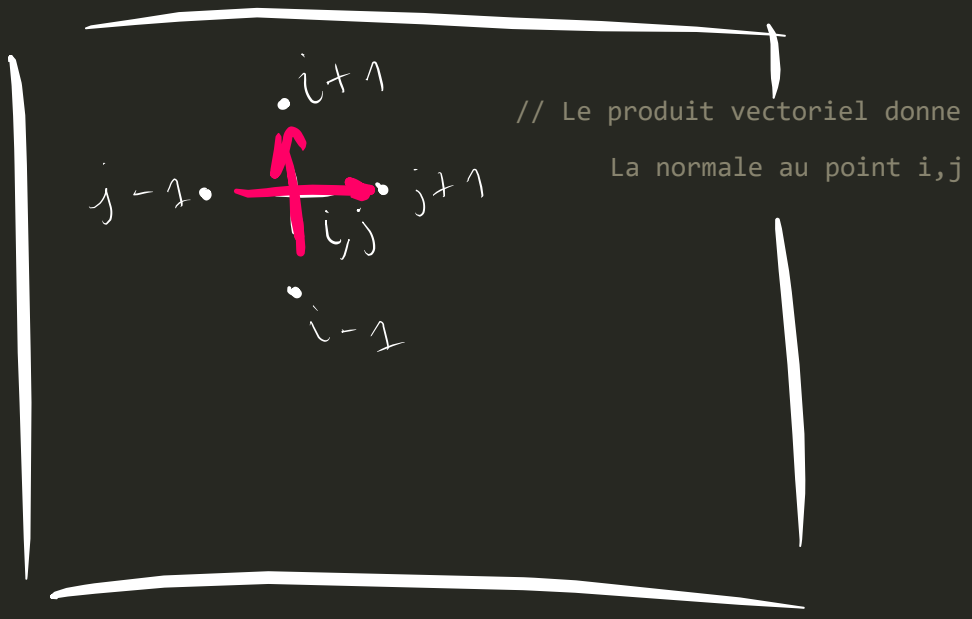


```
void ViewerEtudiant::init_terrain(Mesh& m_terrain, const Image& im){

    m_terrain = Mesh(GL_TRIANGLE_STRIP);
//On parcours l'image et pour chaque point on prend la hauteur en fonctions de
la composante rouge de sur l'image à cet endroit
//On fait un travail sur le point i et i+1 pour créer le maillage

    for(int i=1; i<im.width()-2; i++) // Parcours la largeur de l'image
    {
        for(int j=1; j<im.height()-1; j++) // Parcours la longeur de l'image
        {
            m_terrain.normal( terrainNormal(im, i+1, j) );
//Texture a la i largeur et j hauteur
            m_terrain.texcoord( float(i)/float(im.width()),
float(j)/float(im.height()) );
            m_terrain.vertex( Point(i+1, 25.f*im(i+1,j).r ,j) ); //la
composante rouge de l'image détermine la hauteur du point

            m_terrain.normal( terrainNormal(im,i,j) );

            m_terrain.texcoord( float(i)/float(im.width()),
float(j)/float(im.height()) );
            m_terrain.vertex( Point( i, 25.f*im(i,j).r, j));
        }
```

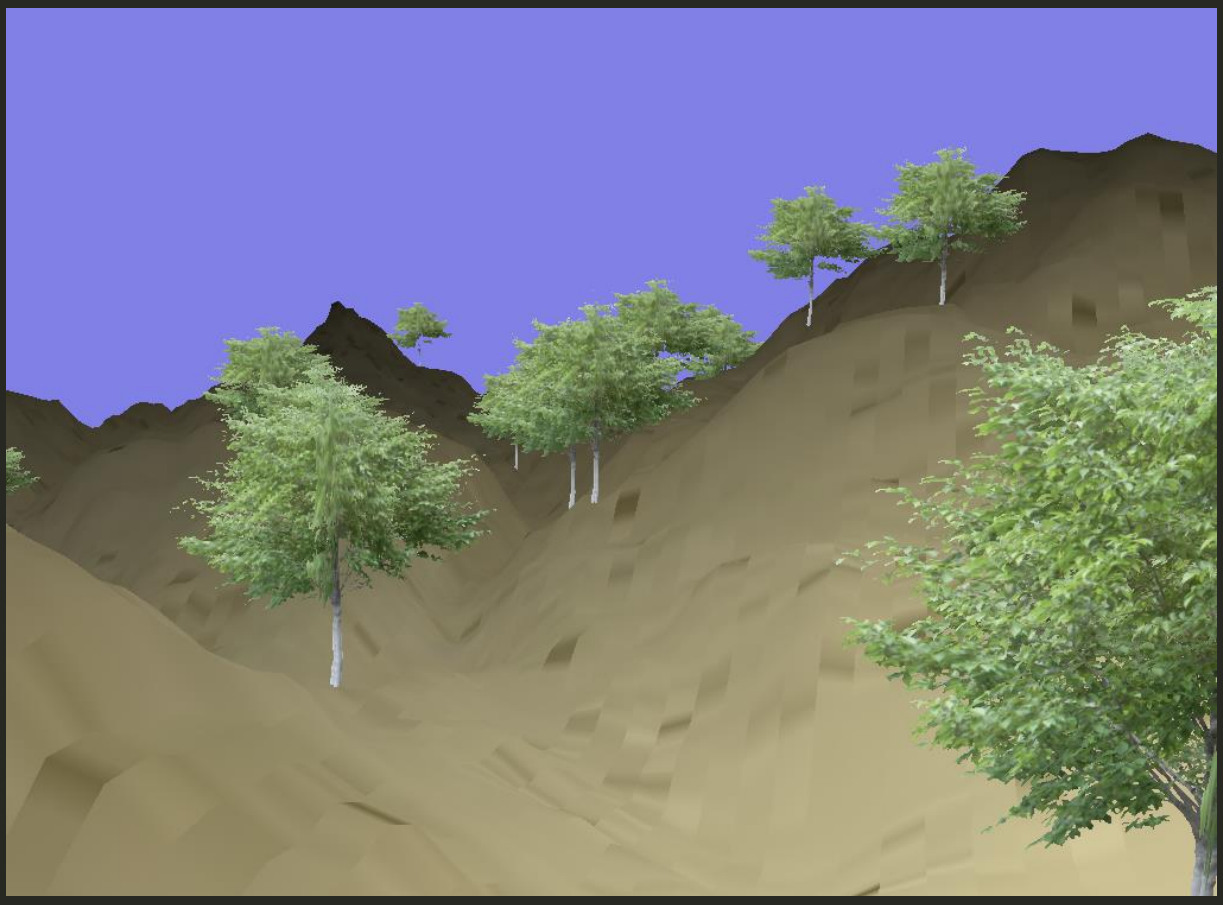
```
        m_terrain.restart_strip();
    }
}
```

Calcul de la normale du terrain :

```
Vector terrainNormal(const Image& im, const int i, const int j){

// Pour chaque point, on crée deux vecteurs qui suivent le plan puis on fait
leur produit vectoriel pour trouver la normale en ce point

    // Calcul de la normale au point (i,j) de l'image
    int ip = i-1;
    int in = i+1;
    int jp = j-1;
    int jn = j+1;

    Vector a( ip, im(ip, j).r, j );
    Vector b( in, im(in, j).r, j );
    Vector c( i, im(i, jp).r, jp);
    Vector d( i, im(i, jn).r, jn);

    Vector ab = normalize(b - a);
    Vector cd = normalize(d - c);

    //Produit vectoriel
    Vector n = cross(cd, ab);

    return n;
}
```



Billboard :

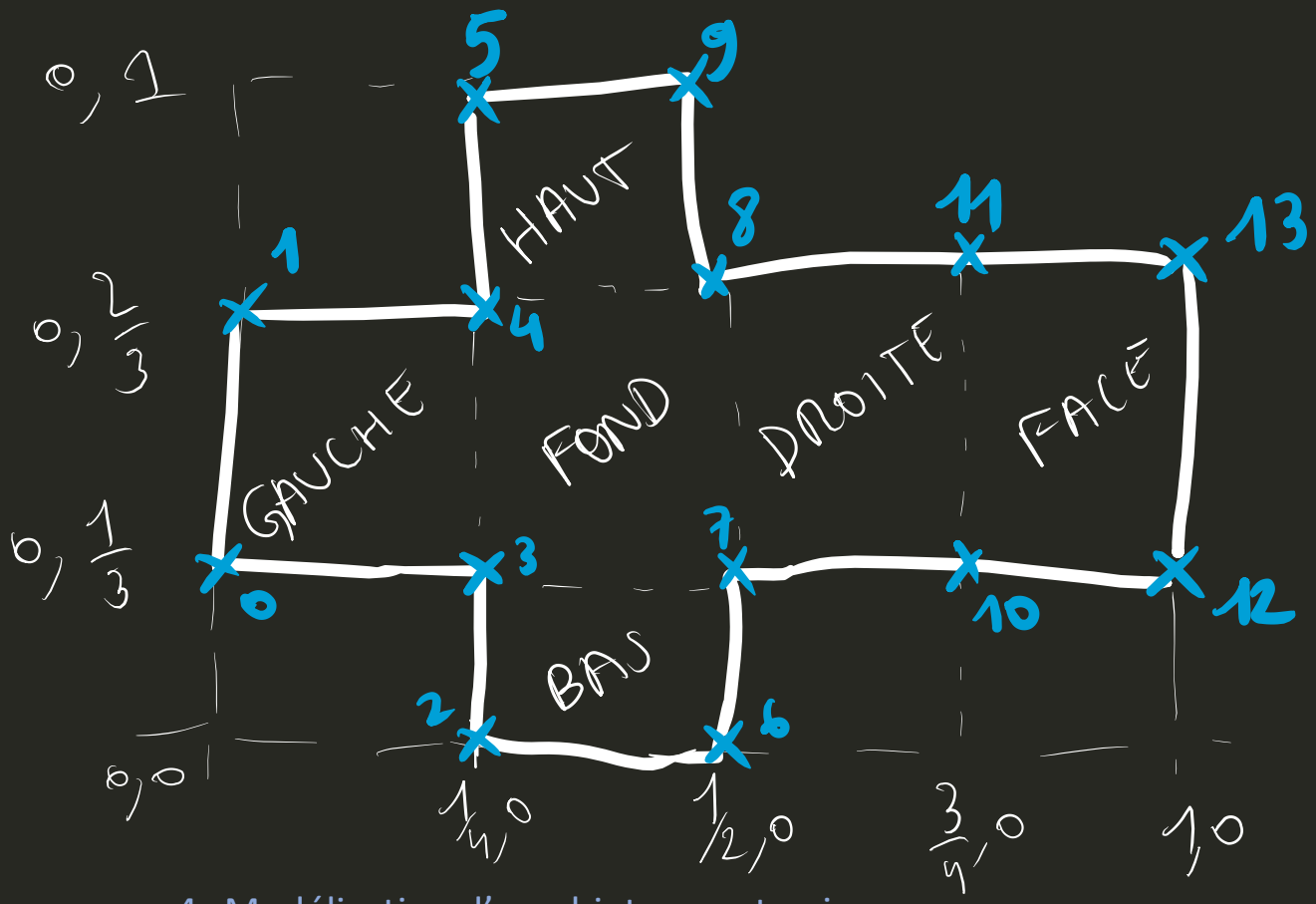


```
void ViewerEtudiant::init_billboard(const Image& im, int tableau[50][3]){

    for(int i=0; i<50; i++)
    {
         //hauteur en fonction de 50 points pris au hasard sur le terrain
         int x = (rand()%im.width());
         int z = (rand()%im.height());
         int y = 25.f*im(x,z).r;

        //- (im.width()/2) pour recentrer les billboard par rapport au terrain
         tableau[i][0] = x - (im.width()/2);
         tableau[i][2] = z - (im.height()/2);
         tableau[i][1] = y;
    }
}
```

```
void ViewerEtudiant::draw_billboard(const Transform&T, unsigned int tex, int
tableau[50][3])
{
  int nbArbres = 50;
    for(int j=0; j<nbArbres; j++)
    {
        for(int i=0; i<10; i++)
        {
            // Donne la hauteur en fonction des coordonnées grâce
init_billboard()
            // Fais un tour complet en 10 rotations pour créer de la
profondeur au billboard
            gl.model(T *
Translation(tableau[j][0],tableau[j][1],tableau[j][2]) *
Rotation(Vector(0,1,0),i*360/10) * Scale(5,5,5) );
            gl.texture(tex);
            gl.alpha(0.8); // Enlève la partie transparente de la texture
            gl.draw(m_quad);
        }
    }
    gl.alpha(0); // Remets la transparence à zéro
}
```

Cubemap :

```
void ViewerEtudiant::init_cubemap()
{
    static float pt[8][3] = { {-1,-1,-1}, {1,-1,-1}, {1,-1,1}, {-1,-1,1}, {-
1,1,-1}, {1,1,-1}, {1,1,1}, {-1,1,1} };
    static int f[6][4] = { {0,1,2,3}, {5,4,7,6}, {2,1,5,6}, {0,3,7,4},
{3,2,6,7}, {1,0,4,5} };

    //Normales inversées pour voir la texture de l'intérieur
    static float n[6][3] = { {0,1,0}, {0,-1,0}, {-1,0,0}, {1,0,0}, {0,0,-1},
{0,0,1} };

    static float tex[14][2] = { {0,0.333}, {0,0.666}, {0.25,0}, {0.25,0.333},
{0.25,0.666}, {0.25,1}, {0.5,0}, {0.5,0.333}, {0.5,0.666}, {0.5,1},
{0.75,0.333}, {0.75,0.666}, {1,0.333}, {1,0.666} };
// static float tex[14][2] = { {0,1/3}, {0,2/3}, {1/4,0}, {1/4,1/3}, {1/4,2/3},
{1/4,1}, {1/2,0}, {1/2,1/3}, {1/2,2/3}, {1/2,1}, {3/4,1/3}, {3/4,2/3}, {1,1/3},
{1,2/3} };
    static int texf[6][4] = { {3,7,6,2}, {8,4,5,9}, {10,7,8,11}, {3,0,1,4},
{12,10,11,13}, {7,3,4,8} };

    m_cubemap = Mesh(GL_TRIANGLE_STRIP);

    for (int i=0; i<6; i++)
    {
        m_cubemap.normal(n[i][0], n[i][1], n[i][2]);

        m_cubemap.texcoord(tex[ texf[i][0] ][0],tex[ texf[i][0] ][1]);
        m_cubemap.vertex( pt[ f[i][0] ][0], pt[ f[i][0] ][1], pt[ f[i][0] ][2]
);

        m_cubemap.texcoord(tex[ texf[i][3] ][0],tex[ texf[i][3] ][1]);
        m_cubemap.vertex(pt[ f[i][3] ][0], pt[ f[i][3] ][1], pt[ f[i][3] ][2]
);

        m_cubemap.texcoord(tex[ texf[i][1] ][0],tex[ texf[i][1] ][1]);
        m_cubemap.vertex( pt[ f[i][1] ][0], pt[ f[i][1] ][1], pt[ f[i][1] ][2]
);

        m_cubemap.texcoord(tex[ texf[i][2] ][0],tex[ texf[i][2] ][1]);
        m_cubemap.vertex( pt[ f[i][2] ][0], pt[ f[i][2] ][1], pt[ f[i][2] ][2]
);
        m_cubemap.restart_strip();
    }
}
```

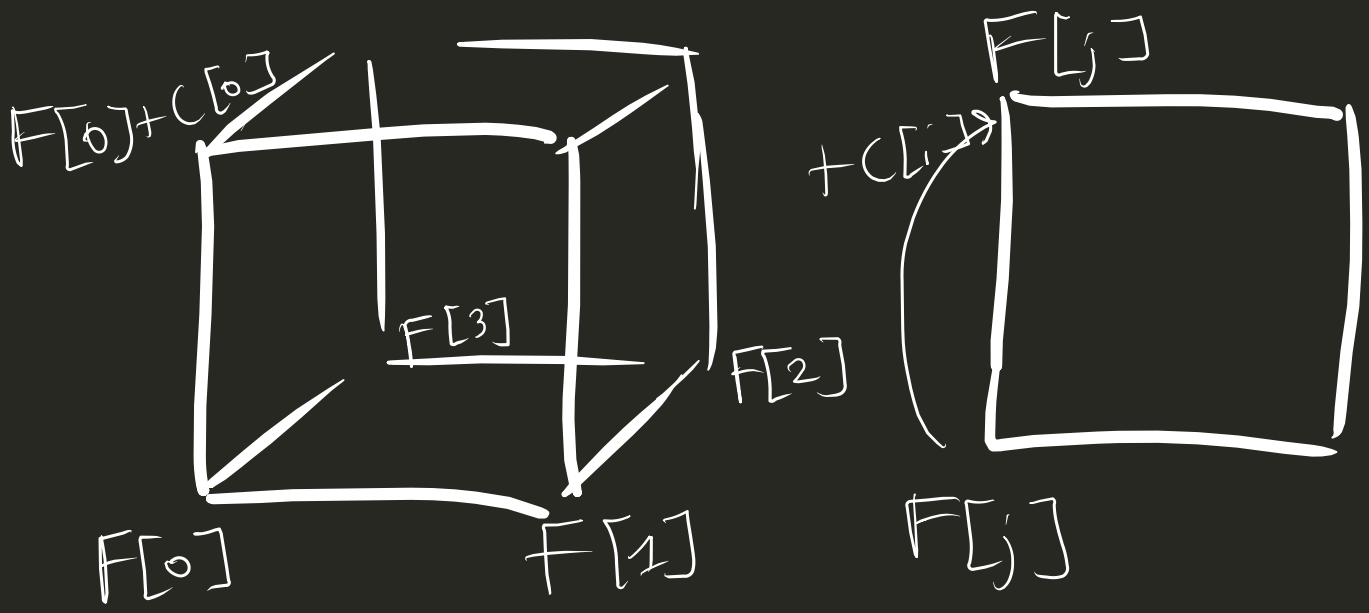


4. Modélisation d'un objet par extrusion

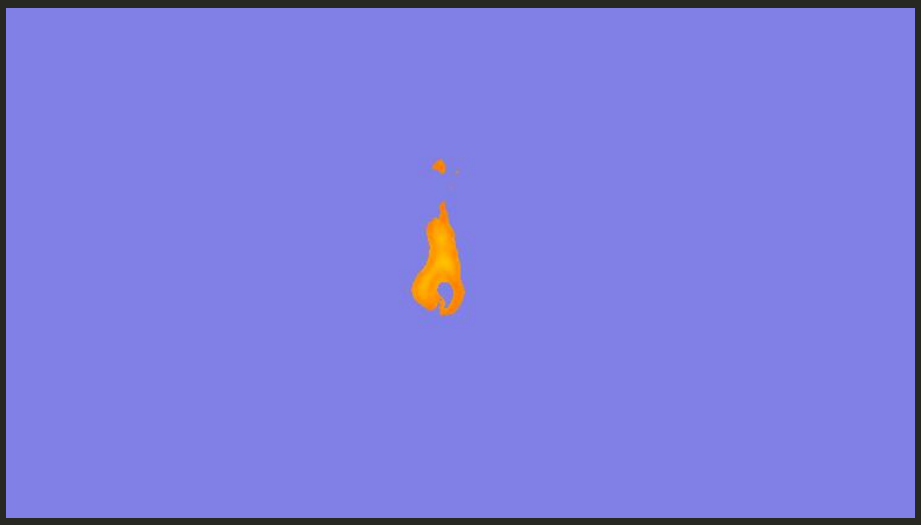
```
void ViewerEtudiant::init_extrusion(Point forme[], Point courbe[], int
tailleCourbe)
{
    m_extrusion = Mesh(GL_TRIANGLE_STRIP);

    //referemer le dessous
    for(int k=0; k<4; k++)
    {
        m_extrusion.vertex(forme[k]);
    }
    for(int i=0; i<tailleCourbe; i++) // boucle sur la courbe
    {
        for(int j=0; j<4; j++) // boucle sur la forme (carré)
        {
                if (j != 3) // Toutes les faces sauf la dernière
                {
                    m_extrusion.texcoord(0,0);
                    m_extrusion.vertex(forme[j]);

                    m_extrusion.texcoord(0,1);
                    m_extrusion.vertex(forme[j+1]);

                    m_extrusion.texcoord(1,0);
                    m_extrusion.vertex(forme[j] + courbe[i]);

                    m_extrusion.texcoord(1,1);
                    m_extrusion.vertex(forme[j+1] + courbe[i]);
                }
                else // refermer la dernière face
                {
                    m_extrusion.texcoord(0,0);
                    m_extrusion.vertex(forme[j]);

                    m_extrusion.texcoord(0,1);
                    m_extrusion.vertex(forme[0]);

                    m_extrusion.texcoord(1,0);
                    m_extrusion.vertex(forme[j] + courbe[i]);

                    m_extrusion.texcoord(1,1);
                    m_extrusion.vertex(forme[0] + courbe[i]);

                    //Apres avoir referemé la dernière face on déplace la
forme d'un étage
                    for(int k=0; k<4; k++)
                    {
                        forme[k] = forme[k] + courbe[i];
                    }
                }
```

```
            m_extrusion.restart_strip(); // Démarre un nouveau strip.
        }
    }
}
```

```
// Après avoir fais la
dernière face, on monte
d'un étage
```



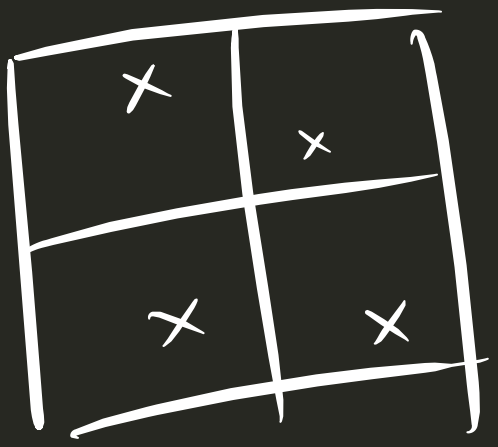
5. Texture animée



```
void ViewerEtudiant::draw_anim(const Transform& T)
{
    int te = int(ts);
```

```
    // Change de textures toutes les secondes et boucle pour revenir à la
première et recommencer
    if (te%9 == 0)
    {
        gl.texture(m_anim1);
    }
    else if (te%9 == 1)
    {
        gl.texture(m_anim2);
    }
    else if (te%9 == 2)
    {
        gl.texture(m_anim3);
    }
    else if (te%9 == 3)
    {
        gl.texture(m_anim4);
    }
    else if (te%9 == 4)
    {
        gl.texture(m_anim5);
    }
    else if (te%9 == 5)
    {
        gl.texture(m_anim6);
    }
    else if (te%9 == 6)
    {
        gl.texture(m_anim7);
    }
    else if (te%9 == 7)
    {
        gl.texture(m_anim8);
    }
    else if (te%9 == 8)
    {
        gl.texture(m_anim9);
    }
    gl.alpha(0.8);
    gl.model(T);
    gl.draw(m_quad);
    gl.alpha(0);
}
```

6. Animation



```
 // Un point au hasard dans chacune des zones
 Point a = Point(rand()%150, rand()%50, rand()%150);
 Point b = Point(-rand()%150, rand()%50, rand()%150);
 Point c = Point(-rand()%150, rand()%50, -rand()%150);
 Point d = Point(rand()%150, rand()%50, -rand()%150);
```

```
int ViewerEtudiant::update( const float time, const float delta )
{
    // time est le temps ecoule depuis le demarrage de l'application, en
millisecondes,
    // delta est le temps ecoule depuis l'affichage de la derniere image / le
dernier appel a draw(), en millisecondes.
    ts = time/1000.0; // Un mouvement toute les secondes
    int te = int(ts); // Temps entier

    float anglerot = 360.0/7.0; // Tour complet en 7 fois
    m_Tsphere = Rotation(Vector(0,1,0),anglerot*ts); // Transform rotation
autour de l'axe des Y
    m_Thelice = Rotation(Vector(1,0,0),anglerot*ts*4); // Rotation des hélices

    int taille = 4; // Nb de trajectoires
    Point V[taille] = { a, b, c, d}; // Tableau de points qui correspondent a
la trajectoire
    // Débuggage
    for(int i=0; i<taille; i++){
        cout << V[i] << endl;
    }
    int ite = te % taille; // Pour retourner a la pos 0 quand on arrive à 7
    int ite_suiv = (te+1) % taille;
    float poids = ts-te;
    Point p0 = V[ite];
    Point p1 = V[ite_suiv];
    Vector pos = Vector(p0) + (Vector(p1)-Vector(p0)) * poids;
    int ite_suiv_suiv = (ite_suiv+1) % taille;
    Point p2 = V[ite_suiv_suiv];
    Vector pos_suiv = Vector(p1) + (Vector(p2) - Vector(p1)) * poids;
    Vector dir = normalize(pos_suiv - pos); // Vecteur unitaire
    Vector up(0,1,0);
    Vector Codin = cross(dir,up); // Vecteur direction ^ up
    m_Tplane = Transform(dir,up,Codin,pos);  // Construction matrice de
passage
    /*|dir.x, up.x, codir.x, pos.x,|
      |dir.y, up.y, codir.y, pos.y,|
      |dir.z, up.z, codir.z, pos.z,|
      |0,     0,    0,       1     |*/
```

```
int taille2 = 10;
Point V2[taille2];
for(int k=0; k<taille2; k++)
{
    float angle = k * 2*M_PI / taille2;
    V2[k] = { Point(150*cos(angle),0,150*sin(angle)) };
}
    // Tableau de points qui correspondent a la trajectoire
    // Débuggage
    for(int i=0; i<taille2; i++){
        cout << V2[i] << endl;
    }
    ite = te % taille2; // Pour retourner a la pos 0 quand on arrive à 7
    ite_suiv = (te+1) % taille2;
    poids = ts-te;
    p0 = V2[ite];
    p1 = V2[ite_suiv];
    pos = Vector(p0) + (Vector(p1)-Vector(p0)) * poids;
    ite_suiv_suiv = (ite_suiv+1) % taille2;
    p2 = V2[ite_suiv_suiv];
    pos_suiv = Vector(p1) + (Vector(p2) - Vector(p1)) * poids;
    dir = normalize(pos_suiv - pos); // Vecteur unitaire
    Codin = cross(dir,up); // Vecteur direction ^ up
    m_Tplane2 = Transform(dir,up,Codin,pos);  // Construction matrice de
passage
    /*|dir.x, up.x, codir.x, pos.x,|
      |dir.y, up.y, codir.y, pos.y,|
      |dir.z, up.z, codir.z, pos.z,|
      |0,     0,    0,       1     |*/

    return 0;
}
```