Modélisation et géométrie discrète

Compte-rendu TP3

Surfaces paramétriques

Louis Jean
Master 1 IMAGINE
Université de Montpellier

3 octobre 2023

Table des matières

1 Introduction

Ce TP explorait l'implémentation et la visualisation de surfaces paramétriques. À travers trois exercices distincts, nous avons examiné diverses méthodes pour générer et représenter graphiquement des surfaces cylindriques, réglées et de Bézier, en utilisant des points de contrôle, des droites et des grilles de points pour définir les paramètres des surfaces résultantes. Durant tout ce TP, j'ai réutilisé les fonctions de création de courbes de Bézier du TP précédent, et ai donc adapté mon code dans la même lignée.

2 Surface cylindrique

La première partie était dédiée à la conception d'une surface cylindrique, faisant appel à une courbe de Bézier et une droite directrice.

```
Point** CreerSurfaceCylindrique(Point* Bezier, Point* Directrice, long nbPointsBezier, long nbPointsDirectrice) {

    Point** res = new Point*[nbPointsDirectrice];

    for(long k = 0; k < nbPointsDirectrice; k++) {

        res[k] = new Point[nbPointsBezier];

    }

    for(long i = 0; i < nbPointsDirectrice; i++) {

        for(long j = 0; j < nbPointsBezier; j++) {

            res[i][j].x = Bezier[j].x + Directrice[i].x;
            res[i][j].y = Bezier[j].y;
            res[i][j].z = Bezier[j].z;

        }

    }

    return res;

}
```

Figure 1: Fonction CreerSurfaceCylindrique

Cette fonction prend en argument une courbe de Bézier (représentée par un tableau de Point, voir fonction dans le TP2), une courbe directrice (aussi un tableau de Point), le nombre de points présents sur la courbe de Bézier (donc la taille du tableau Bezier) et le nombre de points présents sur la droite directrice (la taille du tableau Directrice). Elle itère sur la droite directrice, en "récréant" nbPointsDirectrice fois la courbe de Bézier sur cette dernière. Elle retourne une surface cylindrique, sous forme de tableau à bidimensionnel.

Pour afficher le résultat de l'appel à cette fonction, j'ai créé une fonction pour dessiner le tableau à deux dimensions représentant la surface.

```
void drawSurface(Point** surfacePoints, long nbu, long nbv) {
    long maxUV = max(nbu,nbv);
    long minUV = min(nbu,nbv);
    glColor3f(1,0,1);
    for(int i = 0; i < minUV; i++) {
        glBegin(GL_LINE_STRIP);
        for(int j = 0; j < maxUV; j++) {
            glVertex3f(surfacePoints[i][j].x,surfacePoints[i][j].y,surfacePoints[i][j].z);
        }
        glEnd();
    }
}
```

Figure 2: Fonction drawSurface

Voici les résultats obtenus.

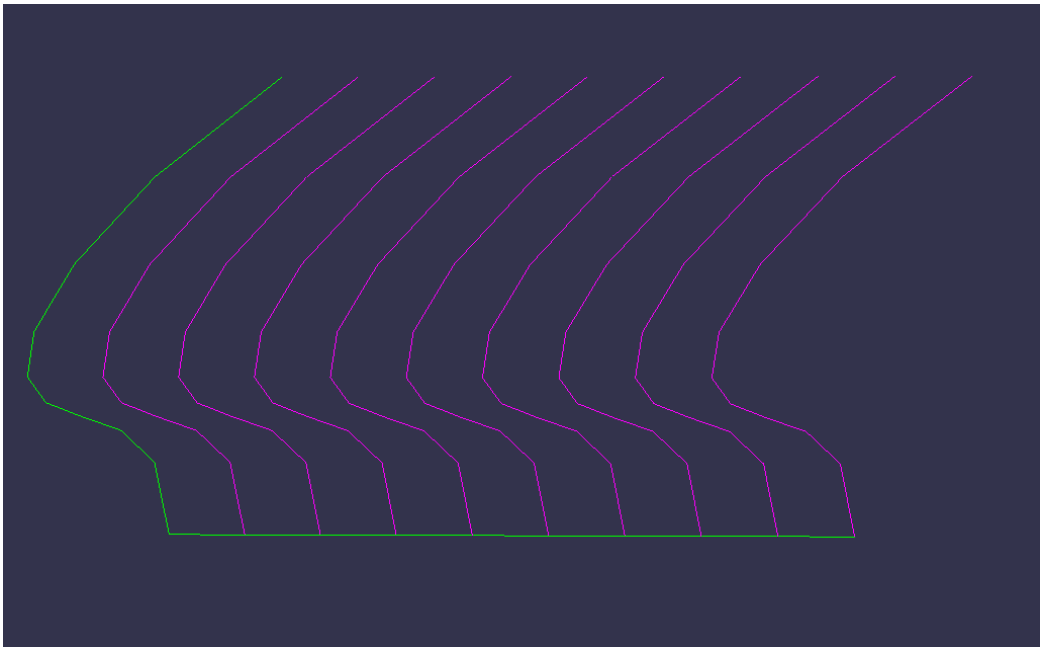

Figure 3: Rendu en 2D avec 5 points de contrôle et nbPointsBezier = nbPointsDirectrice = 10

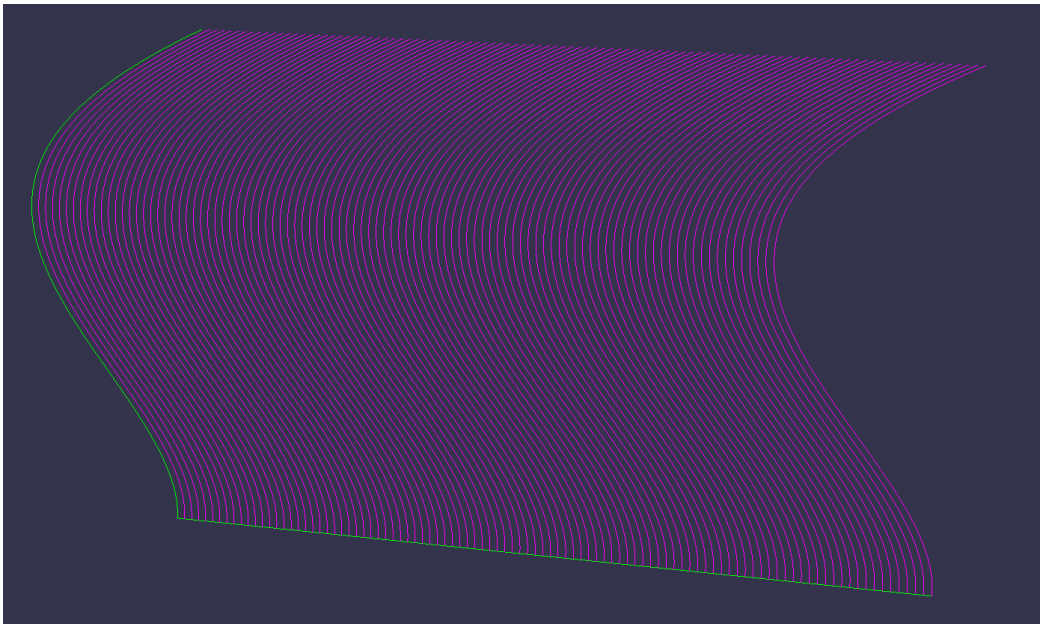

Figure 4: Rendu en 2D avec 5 autres points de contrôle et nbPointsBezier = nbPointsDirectrice = 100

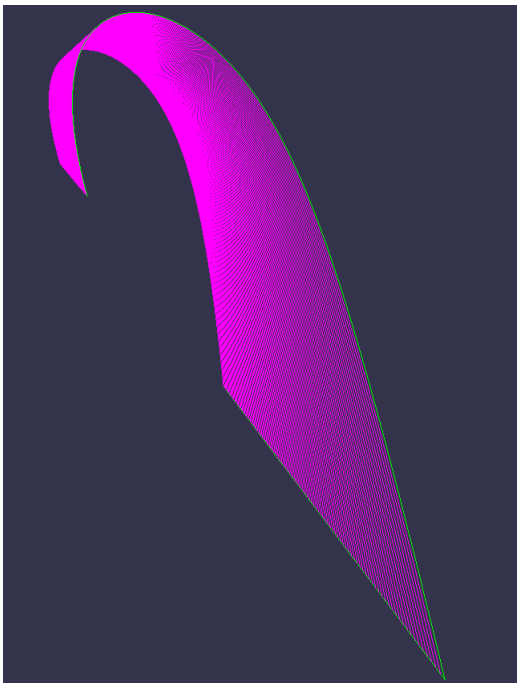

Figure 5: Rendu en 3D avec 5 autres points de contrôle et nbPointsBezier = nbPointsDirectrice = 100

Ensuite, j'ai effectué la facettisation de la surface, en reliant les points en quadrangles.

```
void drawSurfaceQuads(Point** surfacePoints, long nbu, long nbv) {
    long maxUV = max(nbu,nbv);
    long minUV = min(nbu,nbv);
    glBegin(GL_QUADS);
    glColor3f(1,0,1);
    for(int i = 0; i < minUV - 1; i++) {
        for(int j = 0; j < maxUV - 1; j++) {
            glVertex3f(surfacePoints[i][j].x, surfacePoints[i][j].y, surfacePoints[i][j].z);
            glVertex3f(surfacePoints[i][j+1].x, surfacePoints[i][j+1].y, surfacePoints[i][j+1].z);
            glVertex3f(surfacePoints[i+1][j+1].x, surfacePoints[i+1][j+1].y, surfacePoints[i+1][j+1].z);
            glVertex3f(surfacePoints[i+1][j].x, surfacePoints[i+1][j].y, surfacePoints[i+1][j].z);
        }
    }
    glEnd();
}
```

Figure 6: Code de la fonction drawSurfaceQuads

Voilà quelques rendus de ces maillages, en utilisant les surfaces vues ci-dessus.

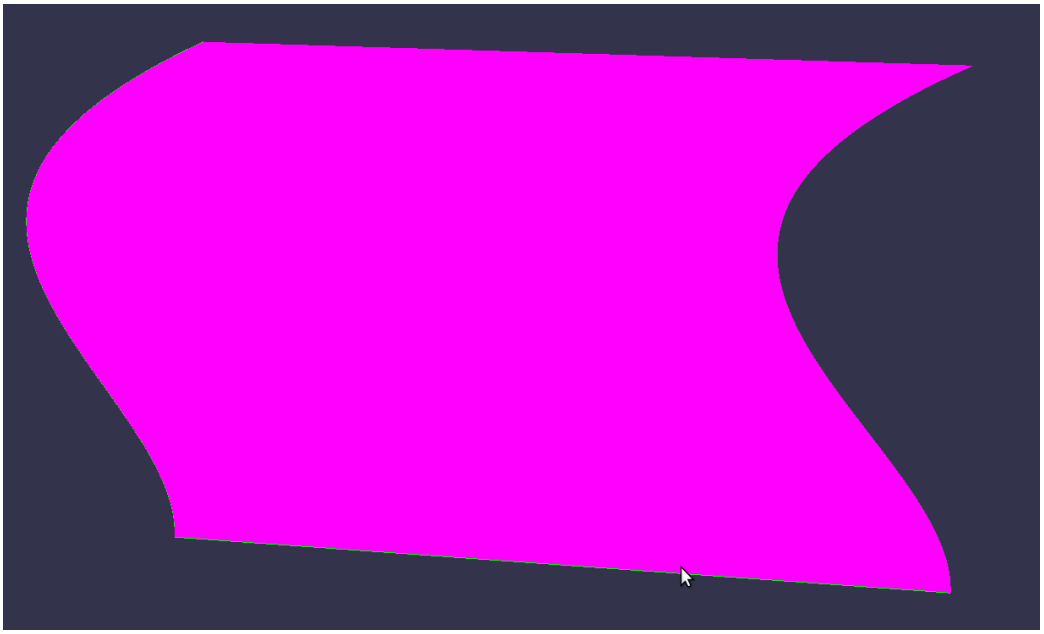

Figure 7: Rendu des quadrangles en 2D

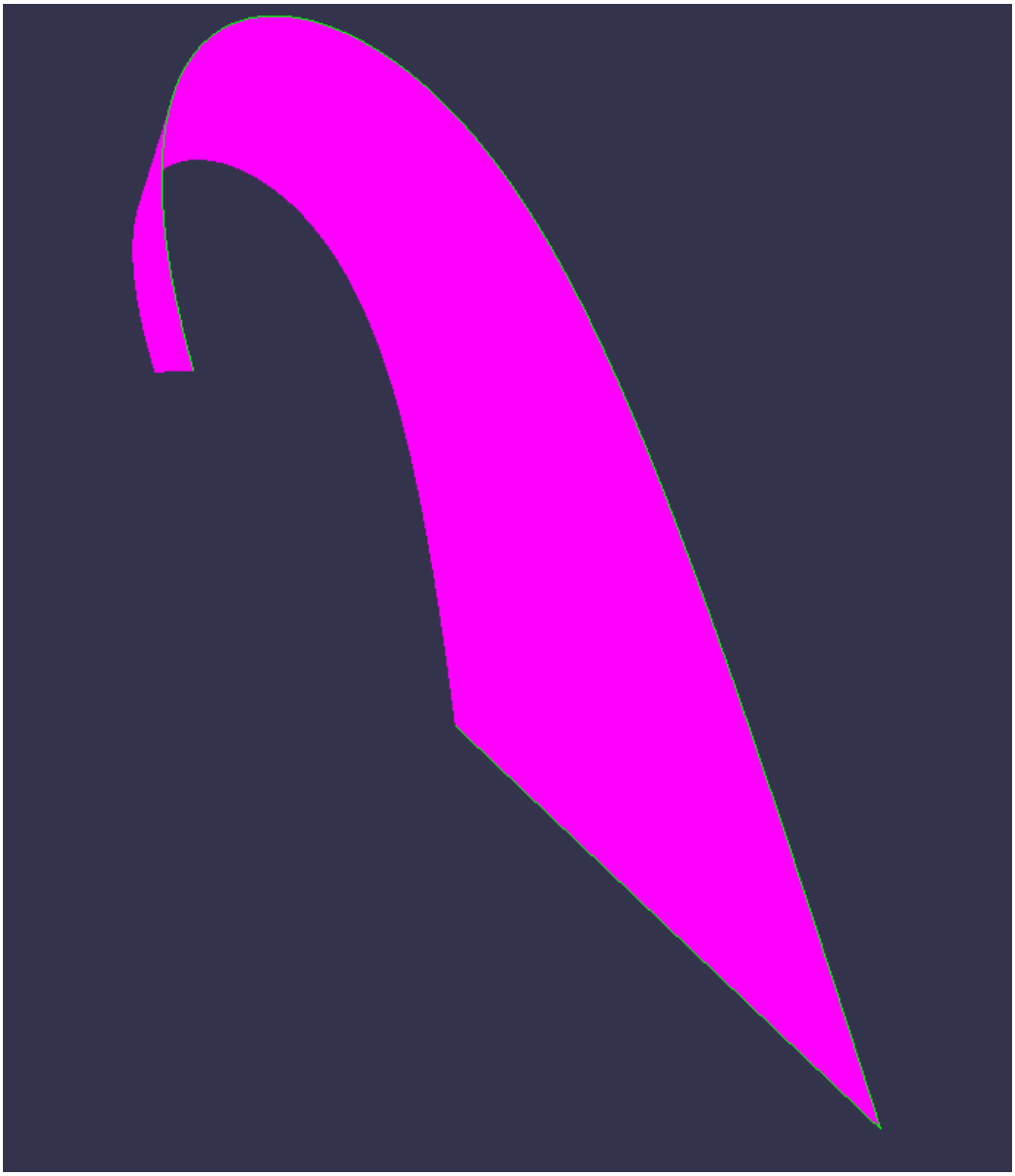

Figure 8: Rendu des quadrangles en 3D

3 Surface réglée

Prenant suite à l'exploration de la surface cylindrique, la deuxième partie se focalisait sur l'étude et la conception de surfaces réglées. Cette fois-ci, le défi résidait dans l'utilisation de deux courbes de Bézier comme directrices.

```
Point** CreerSurfaceReglee(Point* Bezier1, Point* Bezier2, long nbPointsBeziers, long nbPointsSegments) {

    Point** res = new Point*[nbPointsBeziers];

    for(long k = 0; k < nbPointsBeziers; k++) {

        res[k] = new Point[nbPointsSegments];

    }

    for(long i = 0; i < nbPointsBeziers; i++) {

        Point* segment = DroiteDirectrice(Bezier1[i],Bezier2[i],nbPointsSegments);

        for(long j = 0; j < nbPointsSegments; j++) {

            res[i][j].x = segment[j].x;
            res[i][j].y = segment[j].y;
            res[i][j].z = segment[j].z;

        }

    }

    return res;

}
```

Figure 9: Code de la fonction CreerSurfaceReglee

Cette fonction prend en argument deux courbes de Bézier, leur nombre de points (le même pour les deux) et le nombre de points désirés sur chaque segment directeur. Elle parcourt tous les points des deux courbes de Bézier et crée un segment (comprenant nbPointsSegments points) entre chacun d'eux (deux à deux). Elle retourne ensuite une surface réglée, sous forme de tableau à deux dimensions.

Ci-dessous les résultats obtenus avec l'appel à cette fonction.

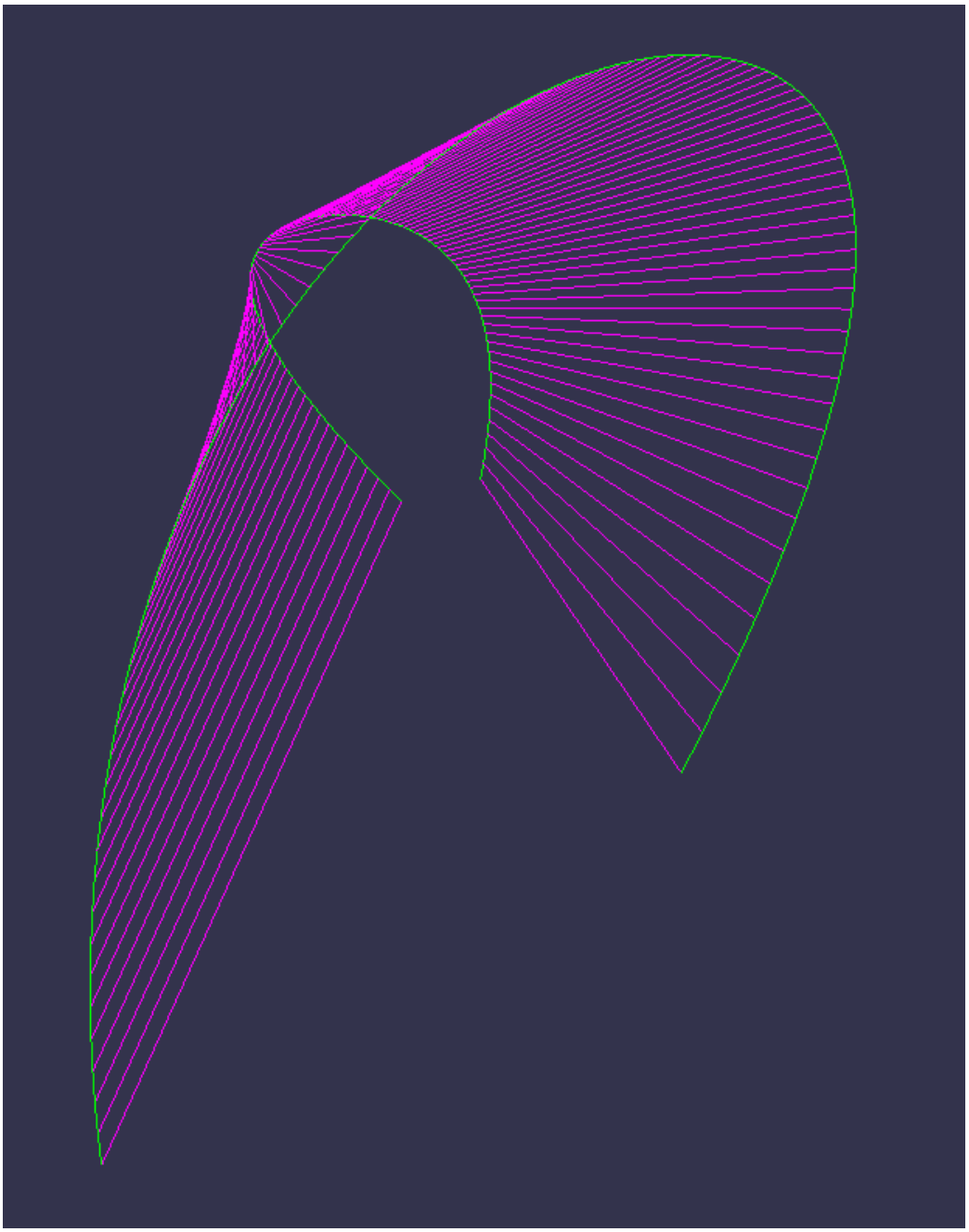

Figure 10: Rendu en 3D avec 5 points de contrôle (différents pour les 2 courbes de Bézier) et nbPointsBeziers = nbPointsSegments = 100

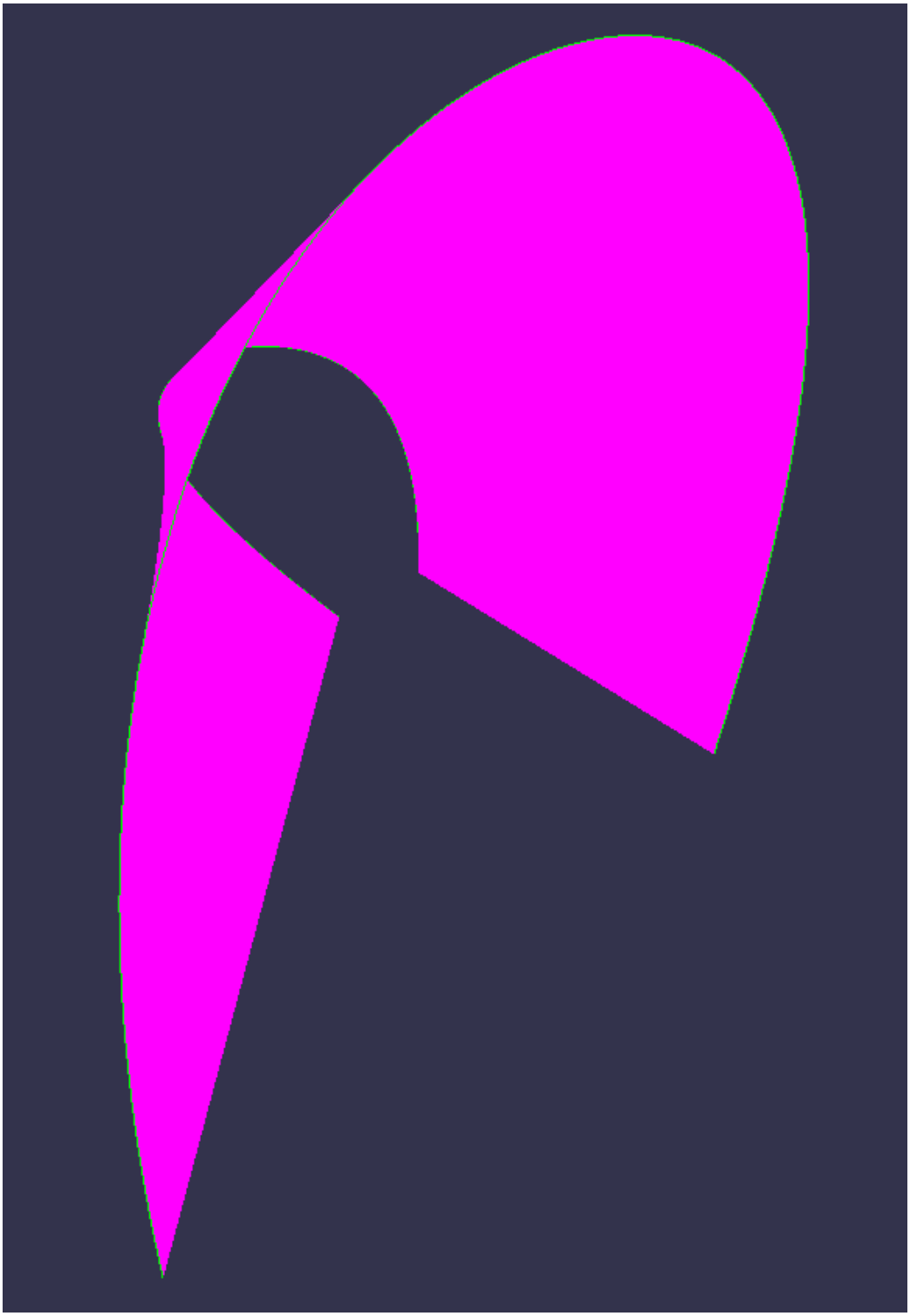

Figure 11: Même rendu que ci-dessus mais avec les quadrangles

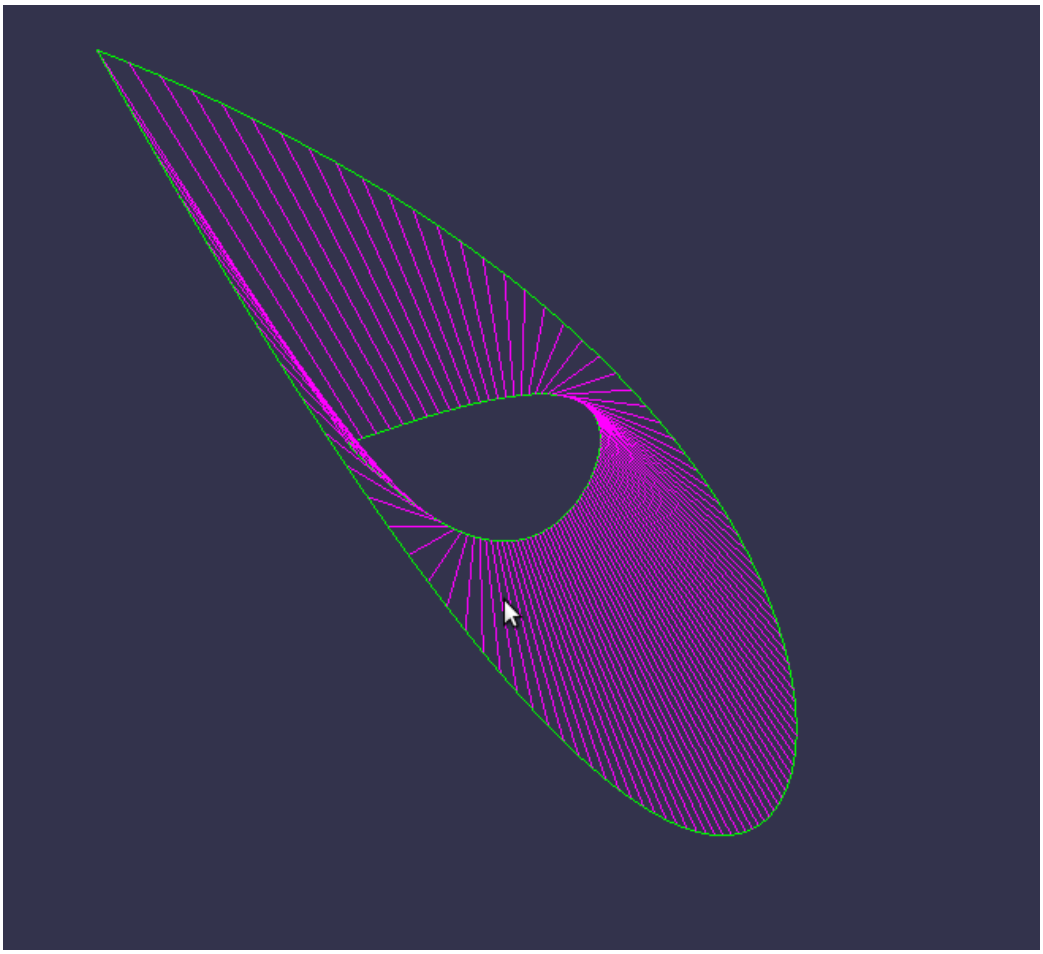

Figure 12: Rendu en 3D avec 2 courbes de Bézier fermées et nbPointsBeziers = nbPointsSegments = 100

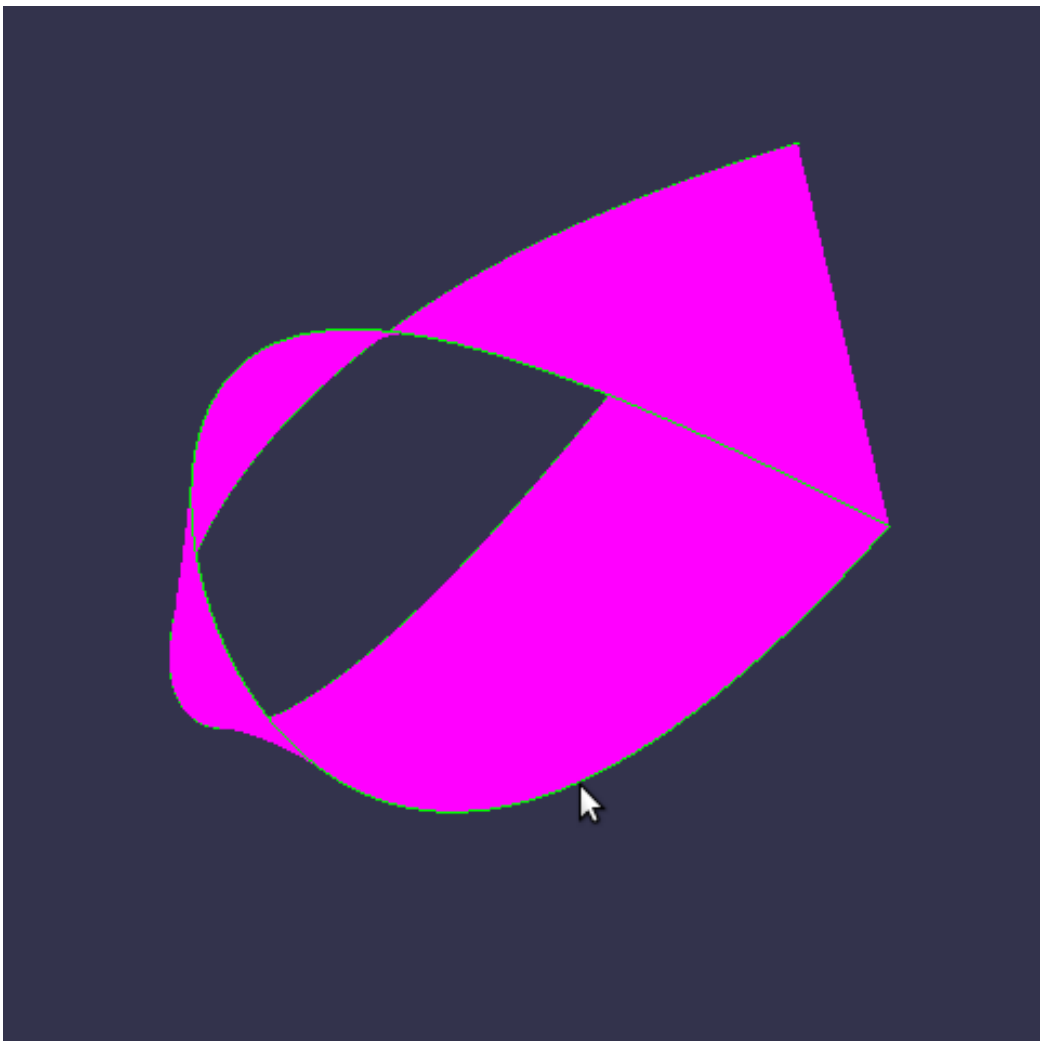

Figure 13: Même rendu que ci-dessus mais avec les quadrangles

4 Surface de Bézier par les polynômes de Bernstein

Dans cette dernière section, nous nous sommes tournés vers une approche différente en explorant la création de surfaces de Bézier en utilisant les polynômes de Bernstein. Ce mécanisme offre un contrôle précis de la forme de la surface à travers une grille de points de contrôle. Un carreau de Bézier est défini par une grille de points de contrôle et les polynômes de Bernstein :

$$S(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} B_i^n(u) B_j^m(v) P_{i,j}$$

avec $n+1$ le nombre de points de contrôle en u, $m+1$ le nombre de points de contrôle en v.

```
Point** CreerSurfaceBezierBernstein(Point** grilleControlPoints, long nbCPu, long nbCPv, long nbu, long nbv) {
    Point** res = new Point*[nbu];
    long degreU = nbCPu - 1;
    long degreV = nbCPv - 1;
    for(long k = 0; k < nbu; k++) {
        res[k] = new Point[nbv];
    }
    for(long i = 0; i < nbu; i++) {
        double u = (double)i / (double)(nbu-1);
        for(long j = 0; j < nbv; j++) {
            double v = (double)j / (double)(nbv-1);
            for(long k = 0; k <= degreU; k++) {
                double polynomeU = ((double)fact(degreU)/(float)(fact(k)*(fact(degreU-k)))) * pow(u,k) * pow((1-u),degreU-k);
                for(long l = 0; l <= degreV; l++) {
                    double polynomeV = ((((double)fact(degreV)/(float)(fact(l)*(fact(degreV-l))))) * pow(v,l) * pow((1-v),degreV-l)));
                    res[i][j].x += polynomeU * polynomeV * grilleControlPoints[k][l].x;
                    res[i][j].y += polynomeU * polynomeV * grilleControlPoints[k][l].y;
                    res[i][j].z += polynomeU * polynomeV * grilleControlPoints[k][l].z;
                }
            }
        }
    }
    return res;
}
```

Figure 14: Code de la fonction CreerSurfaceBezierBernstein

Cette fonction prend en entrée une grille de points de contrôle, ses dimensions (nombre de points en u et nombre de points en v) et le nombre de points souhaités en u et en v sur la surface. Elle effectue quatre boucles imbriquées. La première itère sur le nombre de points voulus en u et détermine la valeur de u actuelle. La seconde itère sur le nombre de points voulus en v et détermine la valeur de v actuelle. La troisième itère sur le degré en u de la courbe (autrement dit, le nombre de points de contrôle en u - 1) puis calcule le polynôme de Bernstein correspondant à u. La dernière, semblable à la troisième, itère sur le degré en v de la courbe (autrement dit, le nombre de points de contrôle en v - 1) pour calculer le polynôme de Bernstein correspondant à v, puis multiplie les polynômes en u et en v par les coordonnées du point de contrôle actuel (déterminé par les deux dernières boucles). Elle

somme ensuite ce résultat dans le tableau de la surface. La fonction retourne une surface de Bézier, sous la forme d'un tableau bidimensionnel. Prière de regarder le code pour voir la fonction, car malgré mes efforts pour minimiser la taille et rendre la capture d'écran lisible, cela reste tout juste passable.

Afin d'afficher la grille de points de contrôle, j'ai créé une fonction, qui affiche les points puis les relie.

```
void drawGrilleCP(Point** grillePoints, long nbu, long nbv) {
    long maxUV = max(nbu,nbv);
    long minUV = min(nbu,nbv);
    glColor3f(1, 0, 0);
    glPointSize(5);
    glBegin(GL_POINTS);
    for(int i = 0; i < minUV; i++) {
        for(int j = 0; j < maxUV; j++) {
            glVertex3f(grillePoints[i][j].x, grillePoints[i][j].y, grillePoints[i][j].z);
        }
    }
    glEnd();
    glBegin(GL_LINES);
    for(int i = 0; i < maxUV; i++) {
        for(int j = 0; j < minUV; j++) {
            if(j < nbu-1) {
                glVertex3f(grillePoints[i][j].x, grillePoints[i][j].y, grillePoints[i][j].z);
                glVertex3f(grillePoints[i][j+1].x, grillePoints[i][j+1].y, grillePoints[i][j+1].z);
            }
            if(i < nbv-1) {
                glVertex3f(grillePoints[i][j].x, grillePoints[i][j].y, grillePoints[i][j].z);
                glVertex3f(grillePoints[i+1][j].x, grillePoints[i+1][j].y, grillePoints[i+1][j].z);
            }
        }
    }
    glEnd();
}
```

Figure 15: Code de la fonction drawGrilleCP

Pour commencer, j'ai choisi de définir la même grille de points de contrôle que celle vue dans le cours.

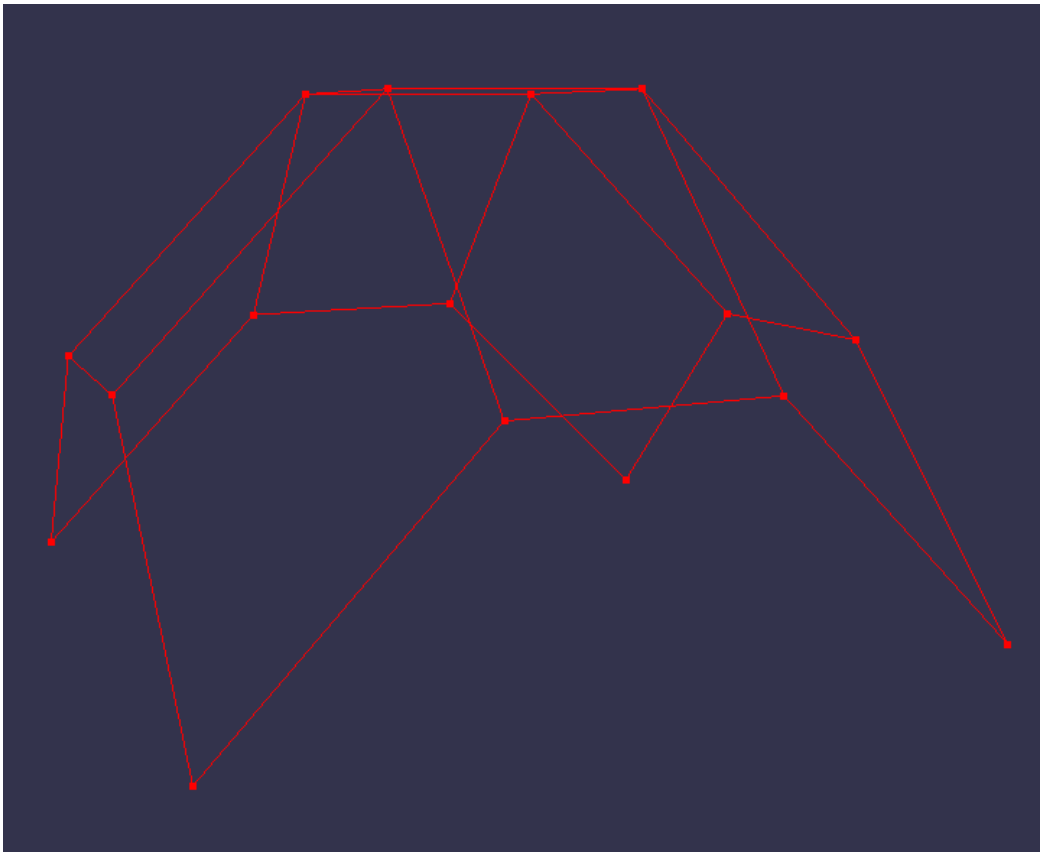

Figure 16: Grille de points de contrôle

À partir de là, j'ai pu créer diverses surfaces de Bézier.

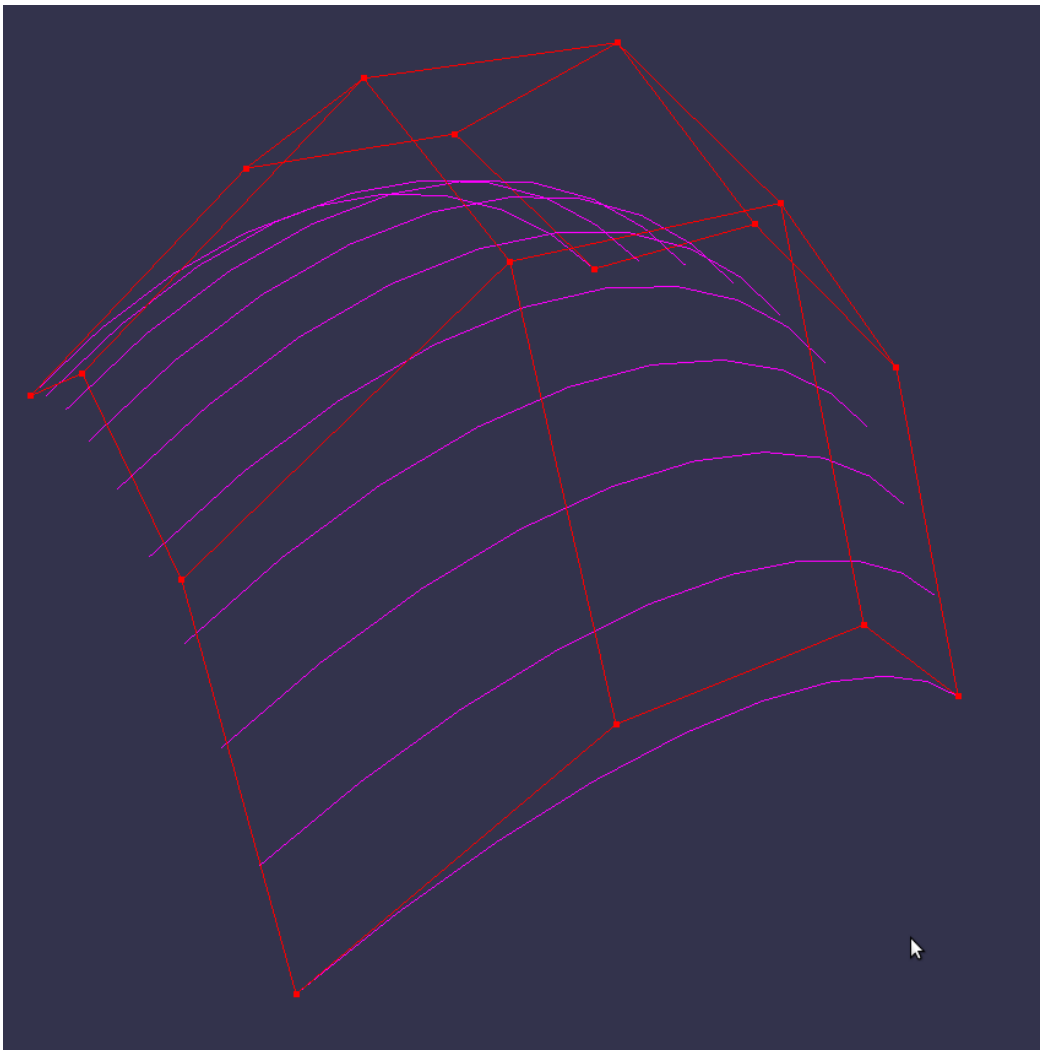

Figure 17: Rendu 3D d'une surface de Bézier avec nbu = nbv = 10

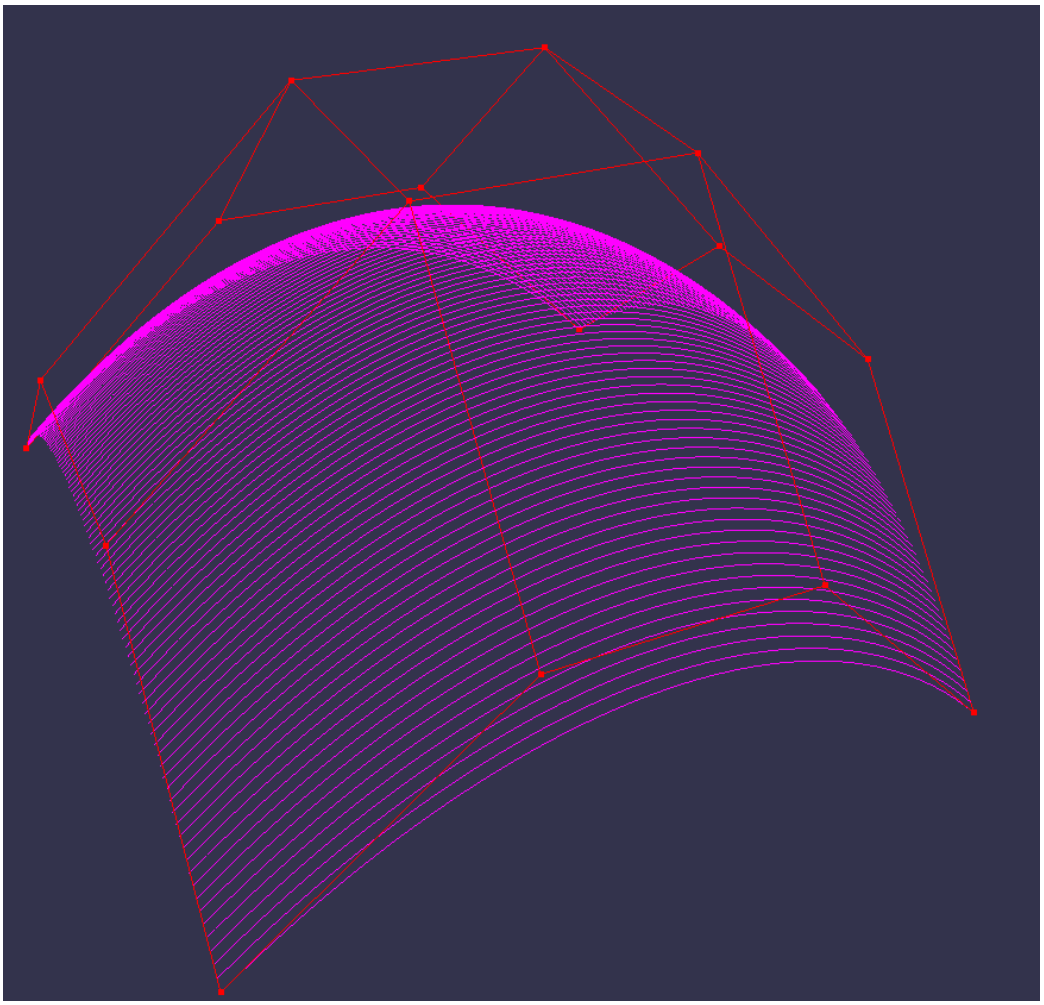

Figure 18: Rendu 3D d'une surface de Bézier avec nbu = nbv = 100

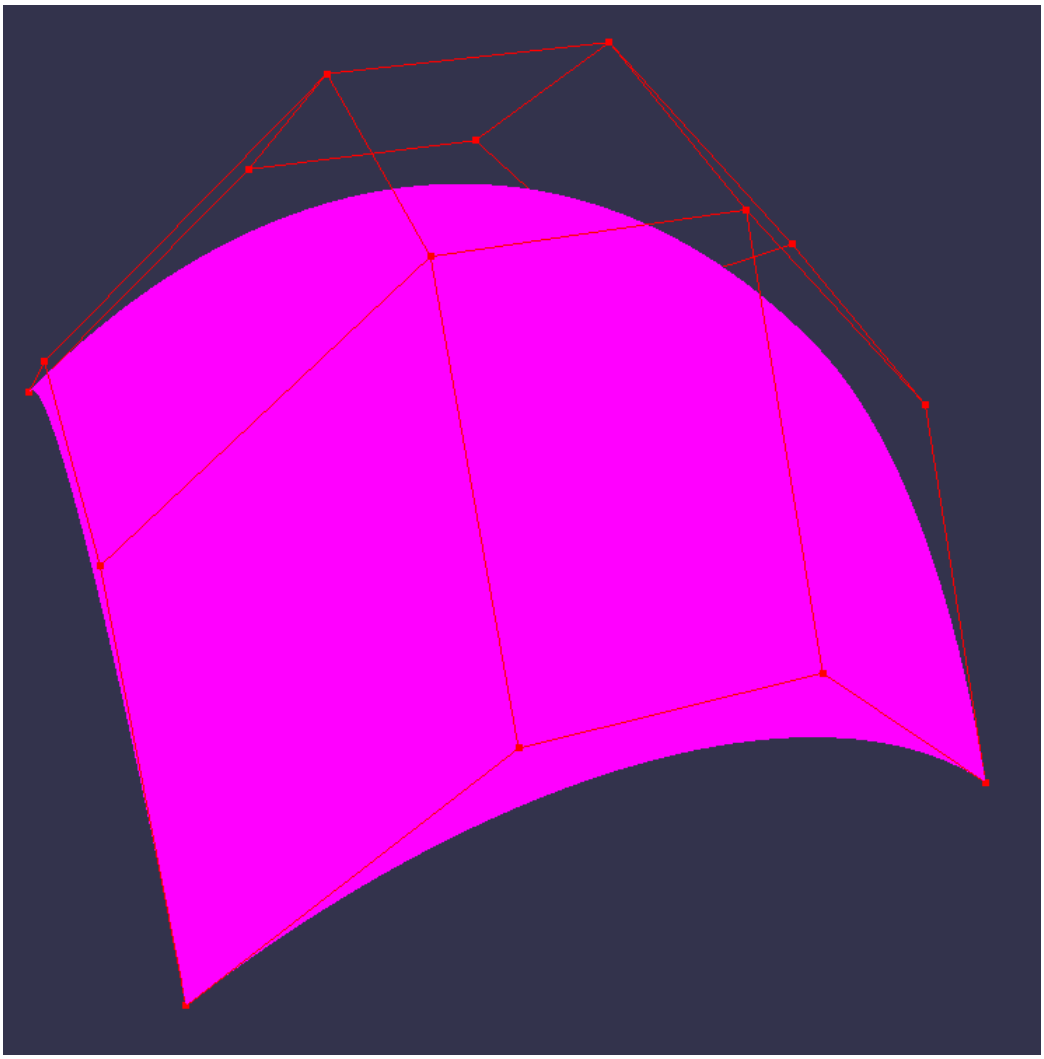

Figure 19: Même rendu que ci-dessus mais avec les quadrangles

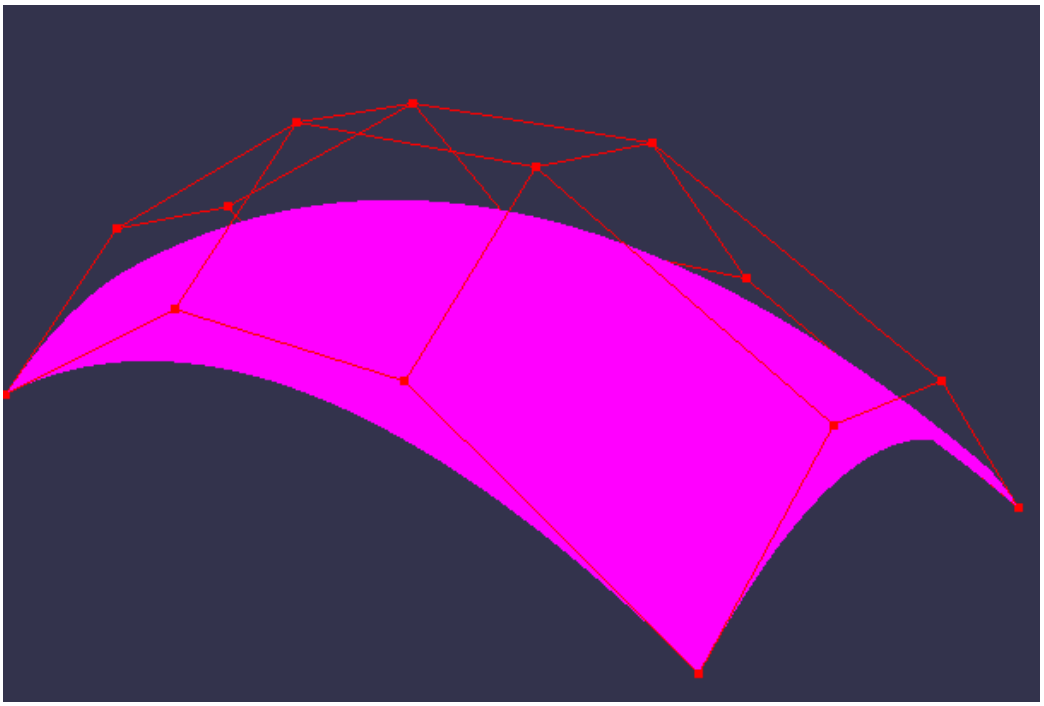

Figure 20: Rendu 3D d'une surface de Bézier avec une autre grille de points de contrôle et nbu = nbv = 100

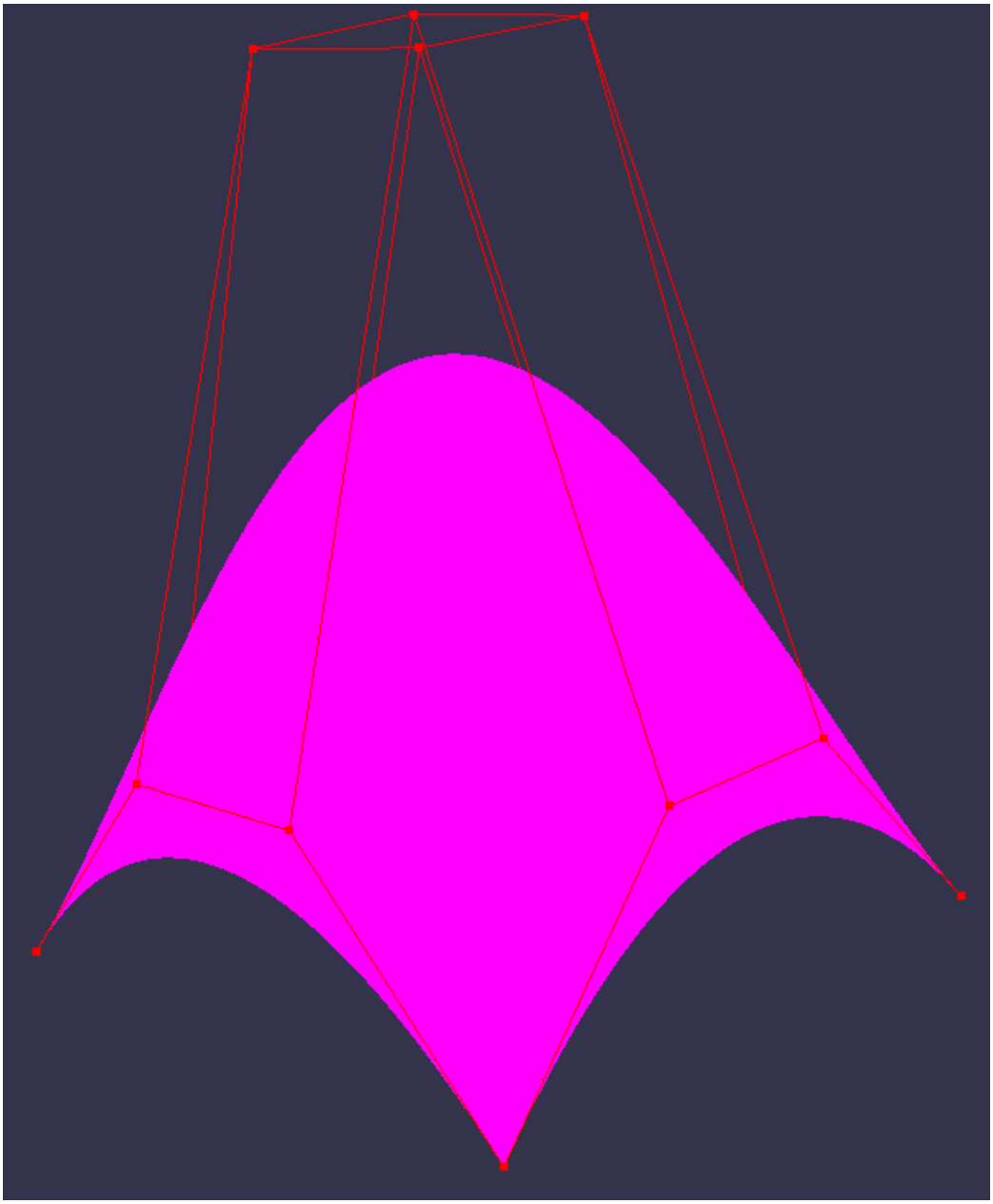

Figure 21: Rendu 3D d'une surface de Bézier avec une autre grille de points de contrôle et nbu = nbv = 100

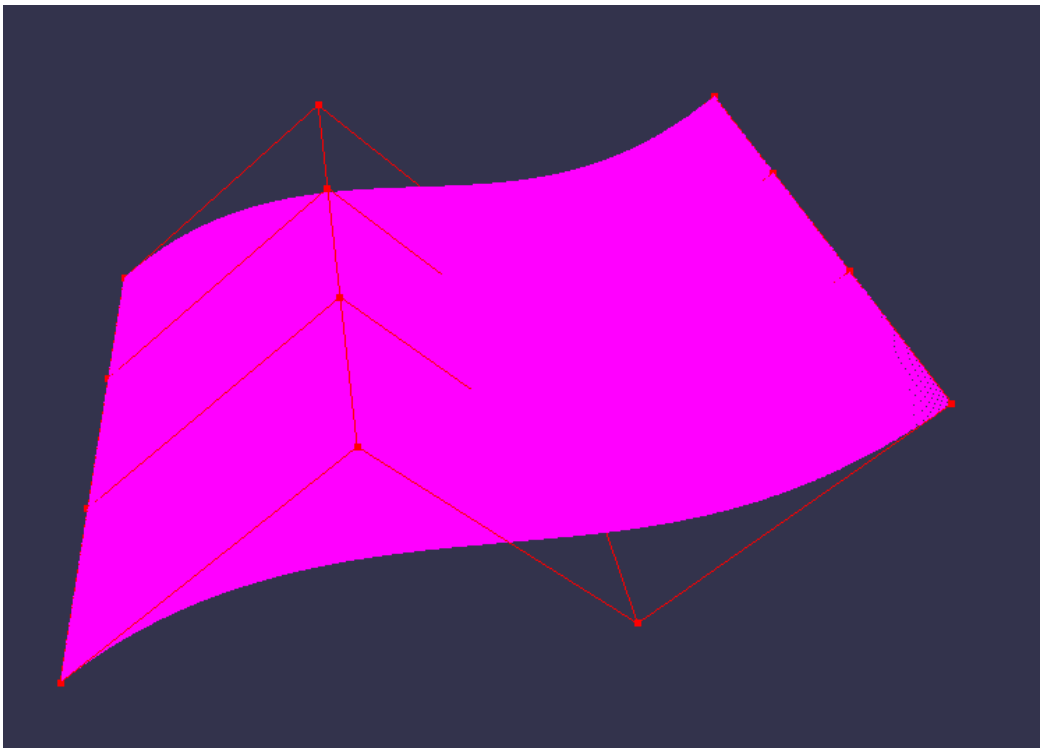

Figure 22: Rendu 3D d'une surface de Bézier avec une autre grille de points de contrôle et nbu = nbv = 100

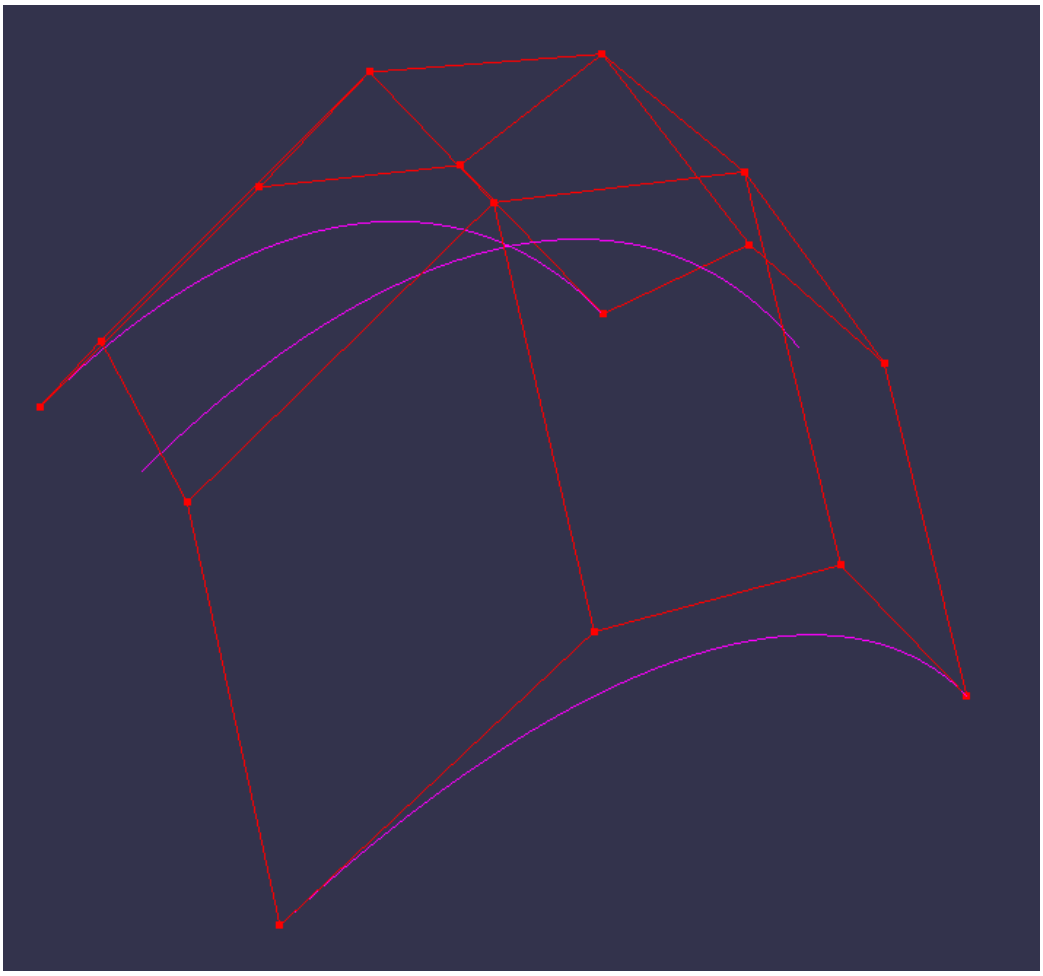

Figure 23: Rendu 3D d'une surface de Bézier avec nbu = 3 et nbv = 1000

Pour une raison qui m'échappe, lorsque c'est nbv qui est plus petit que nbu, j'obtiens ce genre de résultat ci-dessous. J'ai cherché la raison dans mon code, en vain.

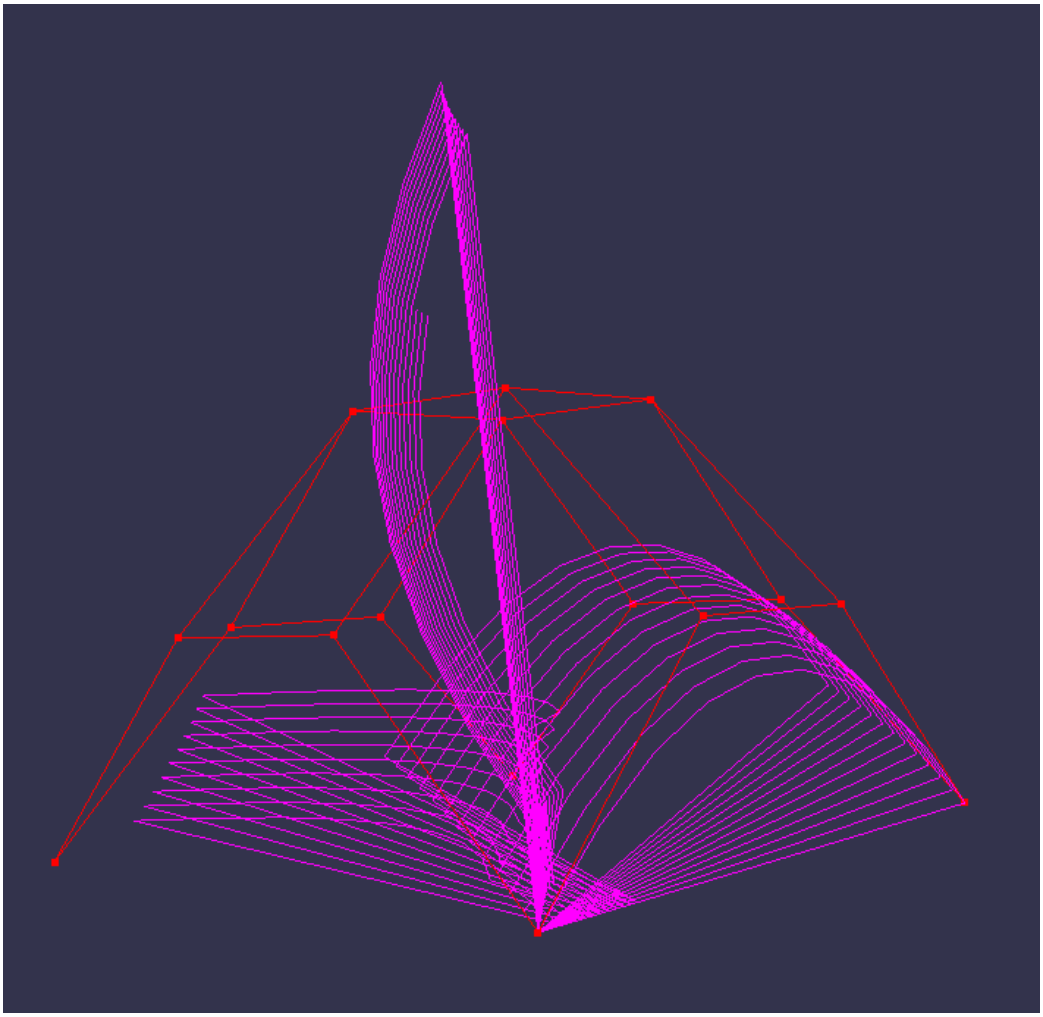

Figure 24: Exemple de bug avec nbu = 50 et nbv = 10

Merci pour le temps et l'attention que vous avez consacrés à la lecture de ce compte-rendu.