Modélisation et géométrie discrète

Compte-rendu TP2

Louis Jean
Master 1 IMAGINE
Université de Montpellier

26 septembre 2023

Table des matières

1 Introduction

Au cours de ce TP, nous avons exploré la réalisation de courbes paramétriques avec OpenGL, en mettant l'accent sur les courbes cubiques d'Hermite et les courbes de Béziers, via les polynômes de Bernstein et l'algorithme de De Casteljau. Pour la réalisation de ce TP, j'ai réutilisé le code du TP précédent, en l'allégeant grandement, ne gardant que ce qui était important pour l'affichage du rendu.

2 Courbe cubique d'Hermite

Pour le premier exercice, il fallait implémenter la modélisation d'une courbe cubique d'Hermite. Grâce au cours, nous savons qu'une courbe cubique d'Hermite est définie par deux points, P_0 et P_1, et deux vecteurs tangents, V_0 et V_1, aux points respectifs. L'équation paramétrique de la courbe est donnée par :

$$P(u) = (2u^3 - 3u^2 + 1)P_0 + (-2u^3 + 3u^2)P_1 + (u^3 - 2u^2 + u)V_0 + (u^3 - u^2)V_1$$

où u est le paramètre qui varie entre 0 et 1.

À partir de cette équation, j'ai pu en déduire l'implémentation. Mais pour cela, j'ai eu besoin de créer une structure Point. Il faut noter que j'aurai pu utiliser la structure Vec3 déjà présente, mais j'ai préféré faire une nouvelle nomenclature pour mieux m'y retrouver.

```
struct Point {
    float x;
    float y;
    float z;
};
```

Figure 1: Déclaration de la struct Point

J'ai aussi eu besoin de créer une fonction drawCurve, pour dessiner une courbe grâce à OpenGL. Il a fallu appeler cette fonction dans display.

```
void drawCurve(Point* curvePoints, long nbPoints) {
    glBegin(GL_LINE_STRIP);
    for(int i = 0; i < nbPoints; i++) {
        glVertex3f(curvePoints[i].x, curvePoints[i].y, curvePoints[i].z);
    }
    glEnd();
}
```

Figure 2: Code de la fonction drawCurve

```
drawCurve(tabPoints,nbu);
```

Figure 3: Appel de la fonction drawCurve dans la fonction display

Une fois ceci fait, j'ai codé la fonction HermiteCubicCurve.

```
Point* HermiteCubicCurve(Point P0, Point P1, Vec3 V0, Vec3 V1, long nbu) {

    Point* res = new Point[nbu];

    for(int i = 0; i < nbu; i++) {
        double u = (double)i / (double)(nbu-1);
        double u2 = u*u;
        double u3 = u2*u;
        res[i].x = (2*u3 - 3*u2 + 1)*P0.x + (-2*u3 + 3*u2)*P1.x + (u3 - 2*u2 + u)*V0[0] + (u3 - u2)*V1[0];
        res[i].y = (2*u3 - 3*u2 + 1)*P0.y + (-2*u3 + 3*u2)*P1.y + (u3 - 2*u2 + u)*V0[1] + (u3 - u2)*V1[1];
        res[i].z = (2*u3 - 3*u2 + 1)*P0.z + (-2*u3 + 3*u2)*P1.z + (u3 - 2*u2 + u)*V0[2] + (u3 - u2)*V1[2];
    }

    return res;

}
```

Figure 4: Code de la fonction HermiteCubicCurve

Pour tester ma fonction, j'ai dû déclarer des variables, que j'ai ensuite initialisées (aux valeurs conseillées dans l'énoncé du TP) dans le main de mon programme.

```
Point P0;
Point P1;
Vec3 V0;
Vec3 V1;
Point* tabPoints;
long nbu;
```

Figure 5: Déclaration des variables nécessaires au test

```
P0 = {0,0,0};
P1 = {2,0,0};
V0 = {1,1,0};
V1 = {1,-1,0};

tabPoints = HermiteCubicCurve(P0,P1,V0,V1,nbu);
glutDisplayFunc (display);
```

Figure 6: Initialisation de ces variables dans le main et appel de la fonction CubicHermiteCurve

Voici les résultats obtenus avec ces valeurs.

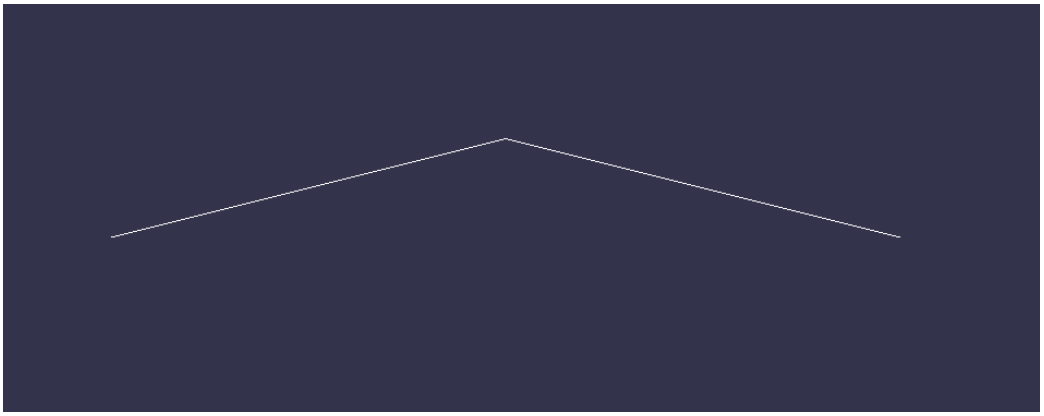

Figure 7: Courbe cubique d'Hermite avec nbu = 3

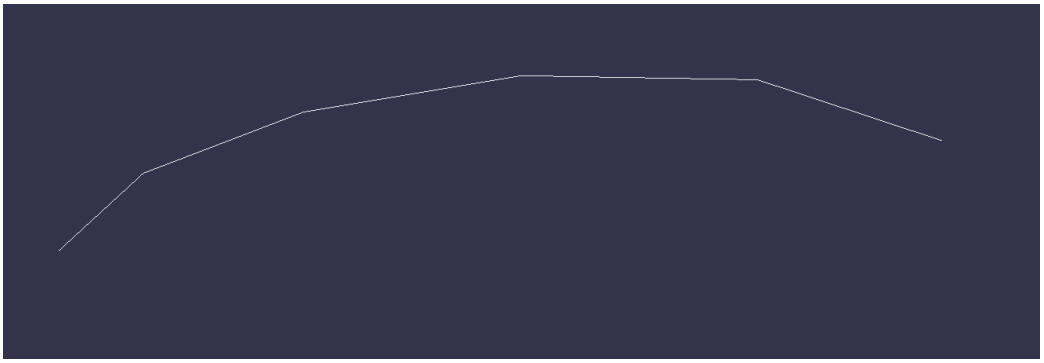

Figure 8: Courbe cubique d'Hermite avec nbu = 6

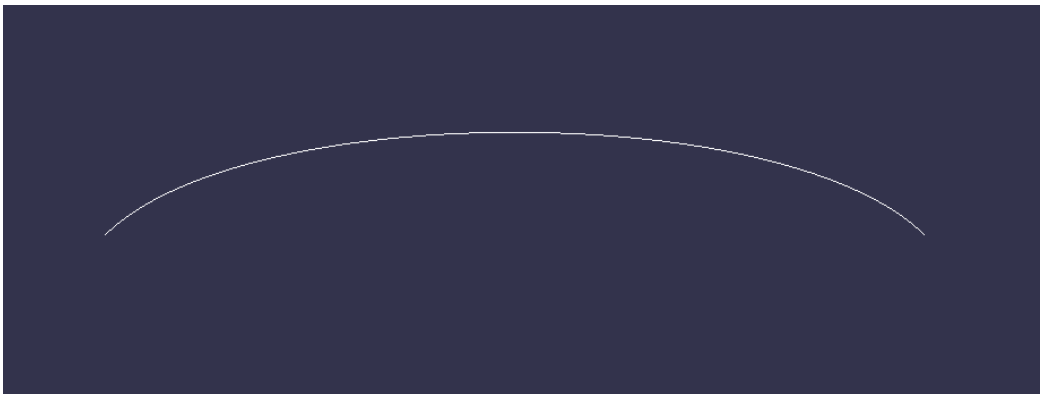

Figure 9: Courbe cubique d'Hermite avec nbu = 10 000

3 Courbe de Bézier par les polynômes de Bernstein

Pour cet exercice, nous avons exploré le tracé de courbes de Bézier via les polynômes de Bernstein.
Les polynômes de Bernstein de degré n sont définis comme suit :

$$B_i^n(u) = \frac{n!}{i!(n-i)!} u^i (1-u)^{n-i}$$

où $B_i^n(u)$ est le i-ème polynôme de Bernstein de degré n et u est le paramètre de la courbe.
Pour une courbe de Bézier définie par des points de contrôle $P_0, P_1, \ldots, P_n$, l'équation de la courbe est :

$$P(u) = \sum_{i=0}^{n} B_i^n(u) P_i \quad , \quad u \in [0,1]$$

Grâce à cela, j'ai spécifié la fonction BezierCurveBernstein.

```
Point* BezierCurveBernstein(Point * controlPoints, long nbControlPoints, long nbu) {

    Point* res = new Point[nbu];

    long degre = nbControlPoints - 1;

    for(int i = 0; i < nbu; i++) {
        double u = (double)i / (double)(nbu-1);
        for(int j = 0; j <= degre; j++) {
            double polynome = ((double)fact(degre)/(float)(fact(j)*(fact(degre-j)))) * pow(u,j) * pow((1-u),degre-j);
            res[i].x += polynome * controlPoints[j].x;
            res[i].y += polynome * controlPoints[j].y;
            res[i].z += polynome * controlPoints[j].z;
        }
    }

    return res;

}
```

Figure 10: Code de la fonction BezierCurveBernstein

Voici les valeurs que j'ai utilisées pour dessiner ma courbe de Bézier avec les polynômes de Bernstein.

```
long nbControlPoints = 5;
Point* controlPoints = new Point[nbControlPoints];
```

Figure 11: Déclaration des variables pour tracer la courbe de Bézier avec les polynômes de Bernstein

```
controlPoints[0] = P0;
controlPoints[1] = P1;
controlPoints[2] = (Point){3,0,0};
controlPoints[3] = (Point){5,0,0};
controlPoints[4] = (Point){4,2,0};
```

Figure 12: Initialisation du tableau de points de contrôle

Il fallait aussi dessiner les points de contrôles utilisés, ce que j'ai fait en écrivant la fonction drawControlPoints, puis en l'appelant dans la fonction display.

```
void drawControlPoints(Point* controlPoints, long nbControlPoints) {
    glBegin(GL_LINE_LOOP);
    glColor3f(1,0,0);
    for(int i = 0; i < nbControlPoints; i++) {
        glVertex3f(controlPoints[i].x, controlPoints[i].y, controlPoints[i].z);
    }
    glEnd();

    // On explicite les points de contrôle en les affichant comme des cercles
    glPointSize(5);
    glBegin(GL_POINTS);
    for(int i = 0; i < nbControlPoints; i++) {
        glVertex3f(controlPoints[i].x, controlPoints[i].y, controlPoints[i].z);
    }
    glEnd();

}
```

Figure 13: Code de la fonction drawControlPoints

```
drawControlPoints(controlPoints,nbControlPoints);
```

Figure 14: Ajout de drawControlPoints dans display

Les résultats obtenus sont les suivants.

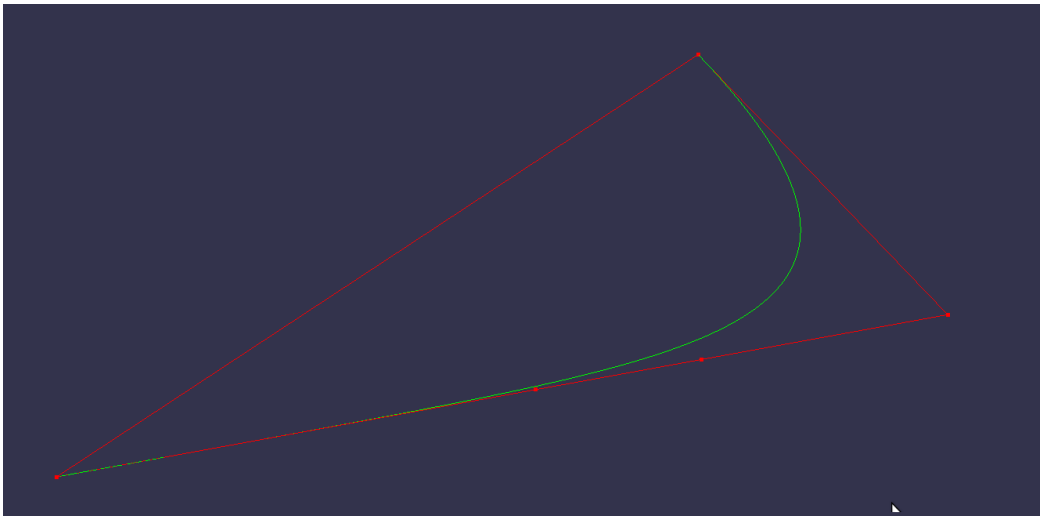

Figure 15: Tracé de la courbe de Bernstein et des points de contrôle

Si l'on change les points de contrôle et que l'on ajoute une composante z à au moins l'un d'entre eux, on peut obtenir une courbe en 3D.

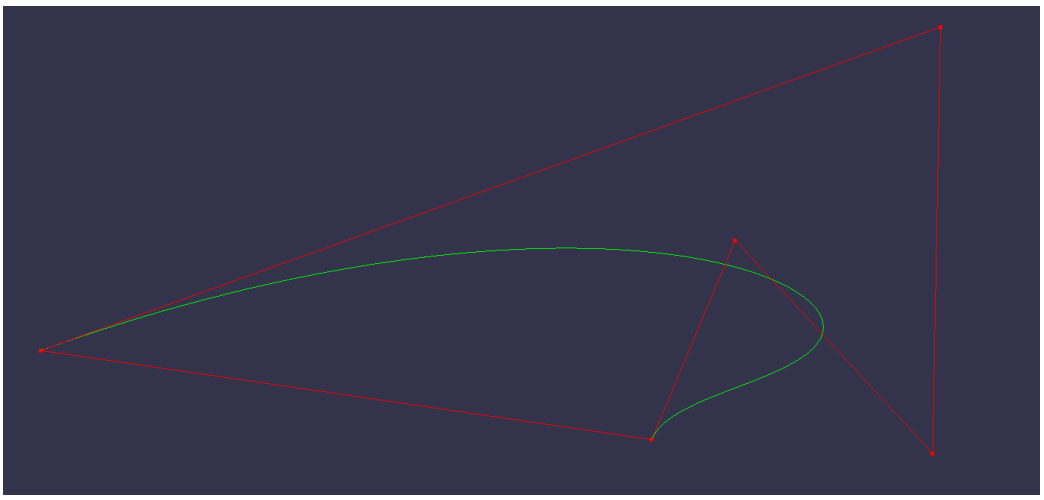

Figure 16: Tracé de la courbe de Bernstein en 3D

Il était ensuite suggéré d'ajouter des évènements clavier pour déplacer les points de contrôle et observer les résultats en temps réel. Pour cela, j'ai choisi d'adopter une approche aléatoire : à chaque pression sur la touche "p", un point de contrôle (choisi aléatoirement parmi tous) se déplace. Voici comment j'ai implémenté cette fonctionnalité.

```
std::random_device rd;
std::mt19937 gen(rd());
std::uniform_int_distribution<> distrib(0, nbControlPoints);
int randomNumber;
```

Figure 17: Déclaration des variables nécessaires à l'aléatoire

```
case 'p':
    randomNumber = distrib(gen);
    if(randomNumber % nbControlPoints <= 3) {
        controlPoints[randomNumber].x *=2;
    }
    else {
        controlPoints[randomNumber].y -= 3;
        controlPoints[randomNumber].z += 1;
    }
    tabPoints = BezierCurveBernstein(controlPoints,nbControlPoints,nbu);
    break;
```

Figure 18: Implémentation du déplacement aléatoire d'un point de contrôle lors de l'appui sur la touche "p"

J'ai essayé d'obtenir quelque chose d'intéressant mais je pense qu'on peut faire beaucoup mieux pour avoir des résultats plus concluants.

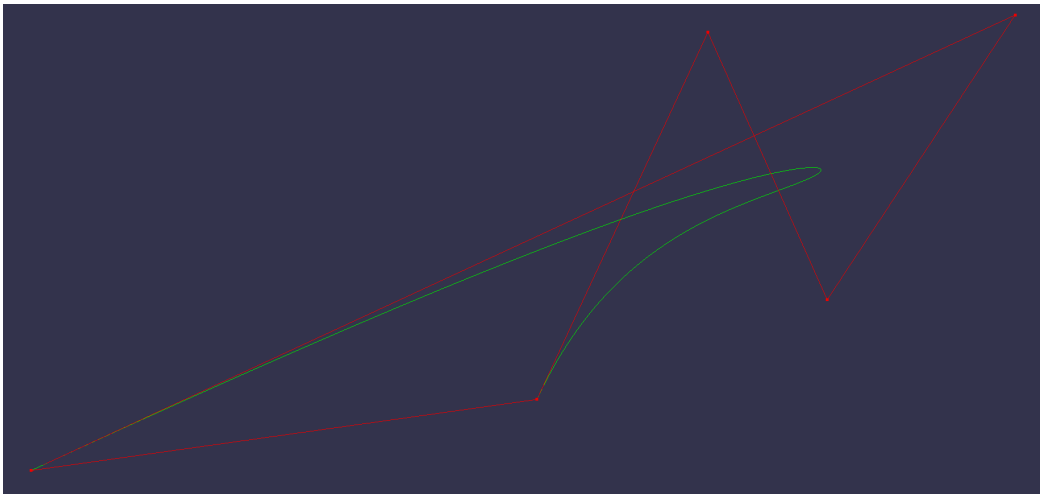

Figure 19: Exemple de courbe de Bézier après plusieurs appuis sur la touche "p"

4 Courbe de Bézier par l'algorithme de De Casteljau

Dans le cadre de cet exercice, nous avons étudié l'élaboration de courbes de Bézier par l'algorithme de De Casteljau. En utilisant l'équation de récurrence détaillée dans le cours :

$$p_i^k = (1-t)p_i^{k-1} + tp_{i+1}^{k-1}$$

on peut déduire une fonction qui trace une courbe de Bézier.

```
Point* BezierCurveDeCasteljau(Point* controlPoints, long nbControlPoints, long nbu) {

    Point* res = new Point[nbu];

    for(int i = 0; i < nbu; i++) {

        double u = (double)i / (double)(nbu - 1);

        Point* Q = new Point[nbControlPoints]; // On repart du tableau de points de contrôle de base
            for(int j = 0; j < nbControlPoints; j++) {
                Q[j] = controlPoints[j];
            }

        // De Casteljau
        for(int k = 1; k < nbControlPoints; k++) {
            for(int j = 0; j < nbControlPoints - k; j++) {
                drawLine(Q[j],Q[j+1]);
                Q[j].x = (1 - u) * Q[j].x + u * Q[j + 1].x;
                Q[j].y = (1 - u) * Q[j].y + u * Q[j + 1].y;
                Q[j].z = (1 - u) * Q[j].z + u * Q[j + 1].z;
                drawPoint(Q[j]);
            }
        }
        res[i] = Q[0];
        drawPoint(Q[0]);

    }

    return res;

}
```

Figure 20: Fonction BezierCurveDeCasteljau

Il était aussi demandé de tracer les étapes de construction de la courbe, afin de voir le fonctionnement de l'algorithme. Pour cela, j'ai écris deux fonctions, que j'ai utilisées dans la fonction BezierCurveDeCasteljau.

```
void drawLine(Point p1, Point p2) {
    glBegin(GL_LINES);
    glColor3f(1,1,0);
    glVertex3f(p1.x,p1.y,p1.z);
    glVertex3f(p2.x,p2.y,p2.z);
    glEnd();
}

void drawPoint(Point p) {
    glBegin(GL_POINTS);
    glColor3f(1,1,0);
    glPointSize(10);
    glVertex3f(p.x,p.y,p.z);
    glEnd();
}
```

Figure 21: Fonctions drawLine et drawPoint

Avec des points de contrôle basés sur 2 axes seulement, voilà le résultat obtenu.

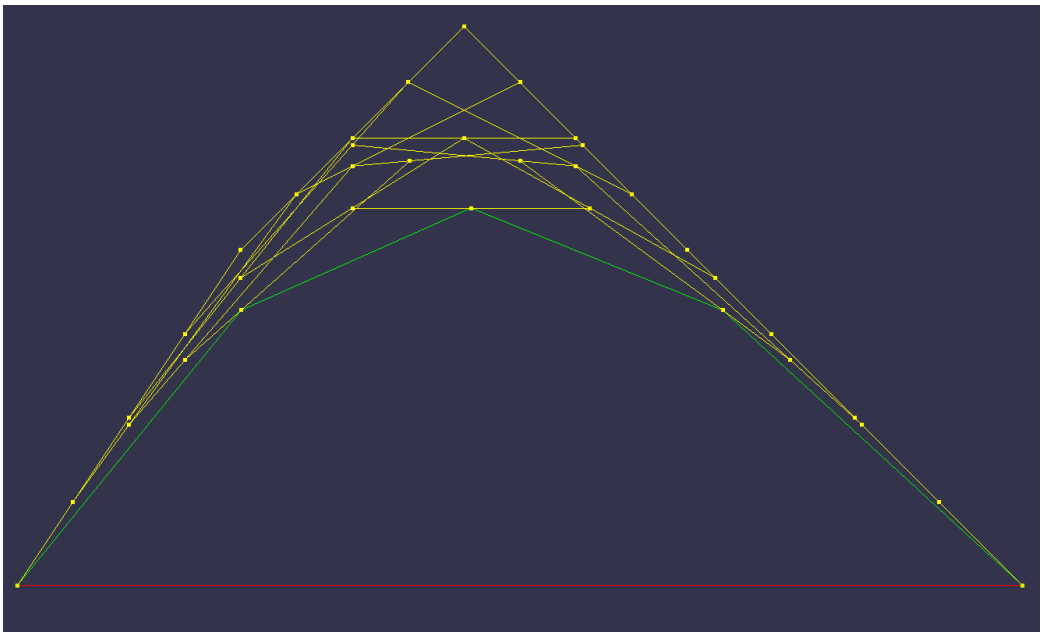

Figure 22: Courbe de Bézier par De Casteljau avec nbu = 5

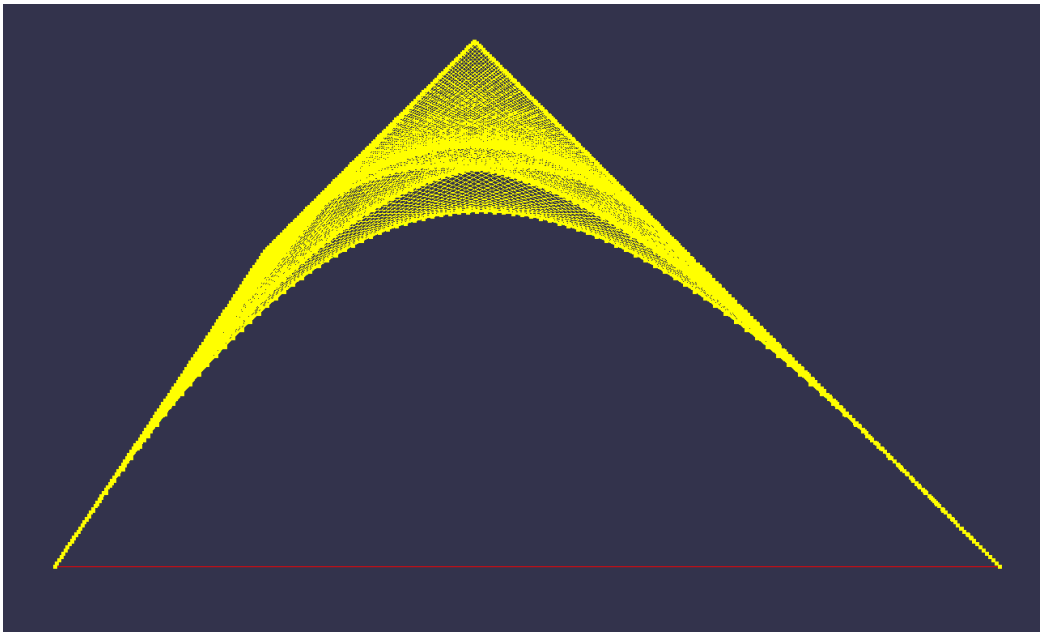

Figure 23: Courbe de Bézier par De Casteljau avec nbu = 100

En ajoutant une composante z aux points de contrôle, cela devient encore plus intéressant.

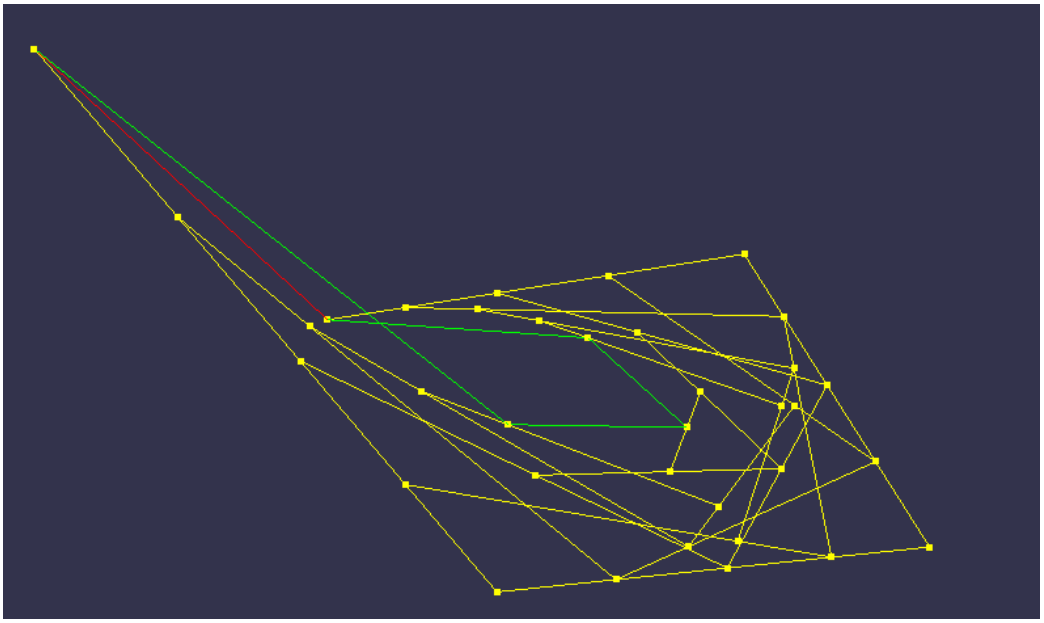

Figure 24: Courbe de Bézier par De Casteljau en 3D avec nbu = 5

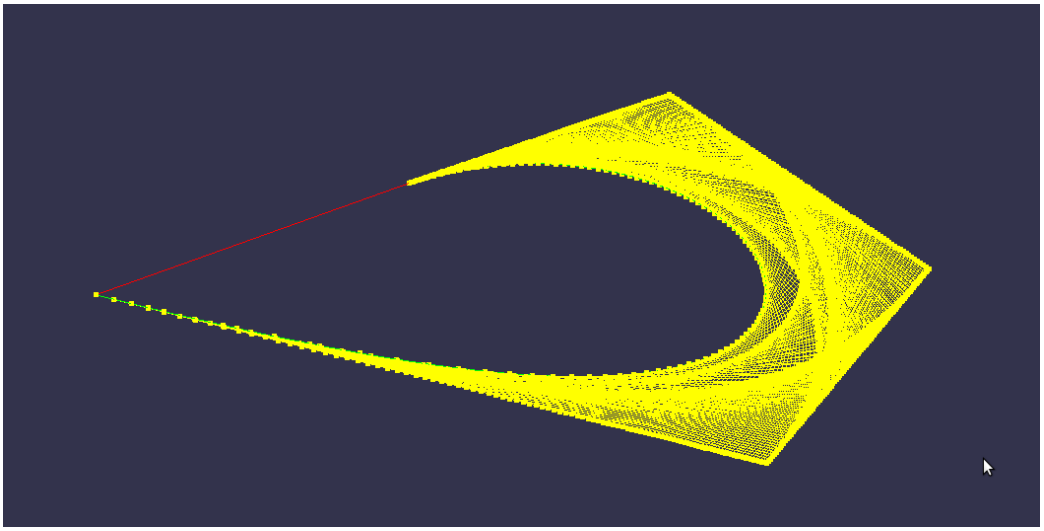

Figure 25: Courbe de Bézier par De Casteljau en 3D avec nbu = 100

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