

Modélisation et géométrie discrète

Compte-rendu TP1

Louis Jean
Master 1 IMAGINE
Université de Montpellier

19 septembre 2023

Table des matières

1 Introduction

L'objectif de ce TP était de se familiariser avec la modélisation 3D en utilisant OpenGL. Pour cela, nous avions à disposition une base de code, et nous devions implémenter deux fonctions afin de discréteriser une sphère puis un plan.

2 Crédation d'un maillage triangulaire de sphère 3D

2.1 Positionnement des sommets

Grâce à l'équation paramétrique donnée dans l'énoncé du TP, j'ai pu implémenter une fonction en C++ qui me donnait des points parfaitement placés sur une sphère.

$$(\theta, \varphi) \in [0, 2\pi] \times \left[-\frac{\pi}{2}, \frac{\pi}{2}\right] :$$

$$x = \cos(\theta) \cdot \cos(\varphi), y = \sin(\theta) \cdot \cos(\varphi), z = \sin(\varphi)$$

```
for(double theta = 0; theta <= 2*M_PI ; theta += (2*M_PI)/nX) {  
    for(double phi = (-M_PI)/2; phi <= M_PI/2 ; phi += M_PI/nY) {  
        double x = cos(theta) * cos(phi);  
        double y = sin(theta) * cos(phi);  
        double z = sin(phi);  
        Vec3 newVertice = Vec3(x,y,z);  
        o_mesh.vertices.push_back(newVertice);  
    }  
}
```

Figure 1: Traduction de l'équation en code C++

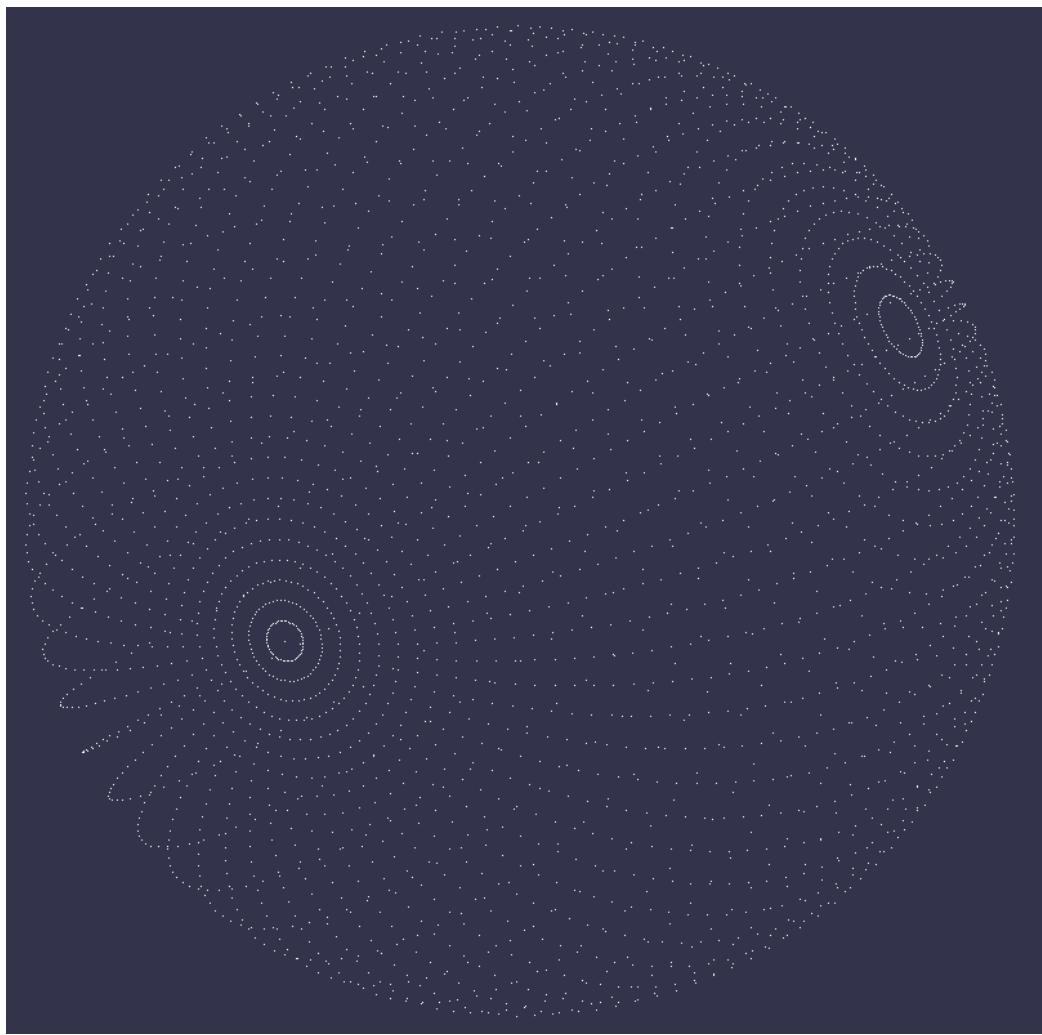


Figure 2: Points représentant une sphère

2.2 Crédit des triangles

J'ai ensuite relié tous ces sommets pour créer des triangles, notamment en utilisant la fonction indiceImage, qui permet de se déplacer plus facilement dans un tableau.

```
for(int x = 0; x < nX - 1; x++) {  
  
    for(int y = 0; y < nY - 1; y++) {  
  
        int p0 = indiceImage(x,y,nY);  
        int p1 = indiceImage(x + 1,y,nY);  
        int p2 = indiceImage(x,y + 1,nY) + 1;  
        int p3 = indiceImage(x + 1,y + 1,nY) + 1;  
  
        if(p0 < nbVertices && p1 < nbVertices && p2 < nbVertices) {  
  
            o_mesh.triangles.push_back(Triangle(p1,p0,p2));  
        }  
  
        if(p2 < nbVertices && p1 < nbVertices && p3 < nbVertices) {  
  
            o_mesh.triangles.push_back(Triangle(p1,p2,p3));  
        }  
    }  
}
```

Figure 3: Code C++ implémentant la création de triangles sur la sphère

```
int indiceImage(int x, int y, int w) {  
  
    return y * w + x;  
}
```

Figure 4: Fonction indiceImage

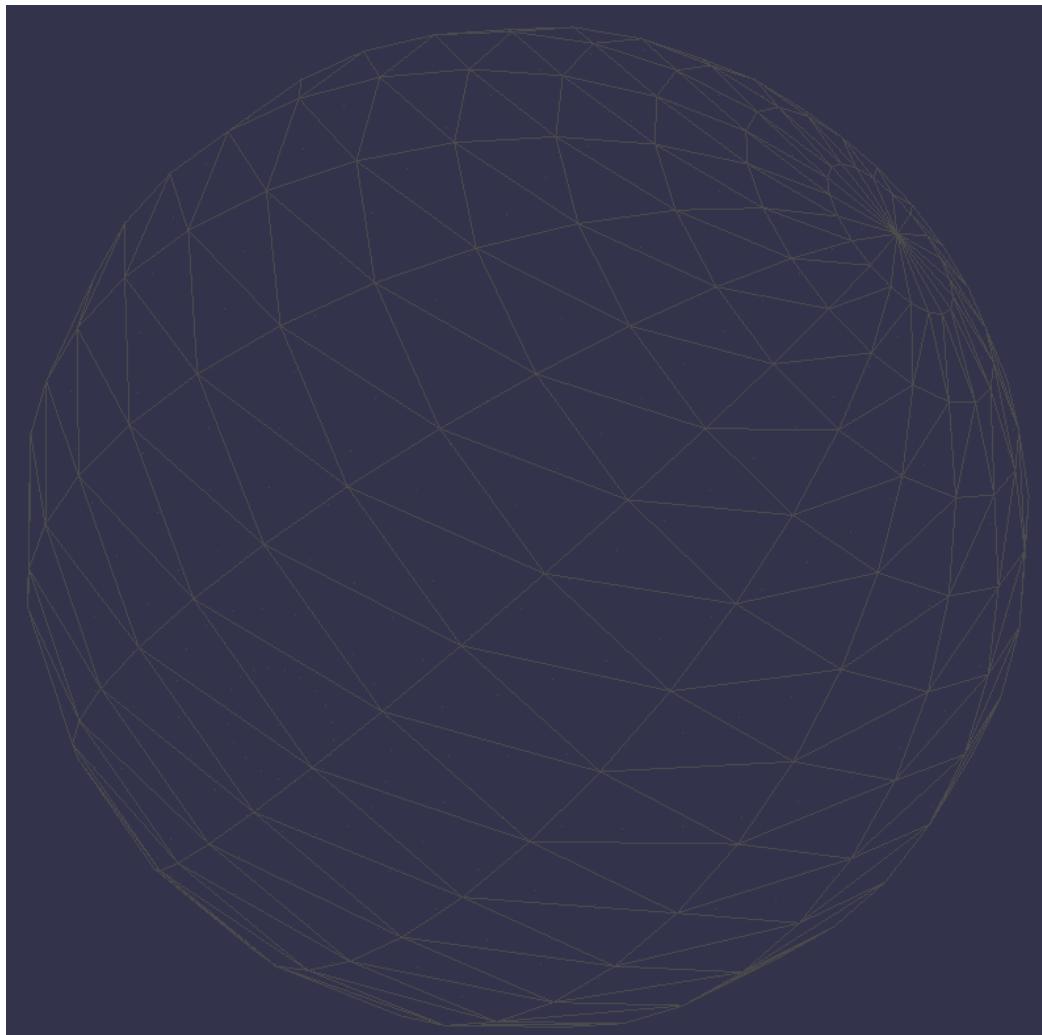


Figure 5: Sphère avec triangles



Figure 6: Sphère avec triangles rendus

2.3 Calcul des normales

Ensuite, j'ai procédé au calcul des normales aux sommets. Comme nous travaillons sur une sphère centrée en l'origine, le calcul des normales est trivial car la normale de chaque sommet correspond simplement au vecteur reliant l'origine au sommet en question, c'est-à-dire à la position-même de ce dernier. Bien sûr j'ai pris soin de normaliser les vecteurs grâce à la fonction normalize que j'ai écrite.

```
o_mesh.normals.push_back(normalize(Vec3(x,y,z)));
```

Figure 7: Implémentation des normales de la sphère en C++

```
Vec3 normalize(Vec3 v) {  
    float divider = sqrt(pow(v[0],2) + pow(v[1], 2) + pow(v[2],2));  
    return Vec3(v[0]/divider,v[1]/divider,v[2]/divider);  
}
```

Figure 8: Fonction permettant de normaliser un vecteur



Figure 9: Sphère avec normales (et donc éclairage bien calculé)

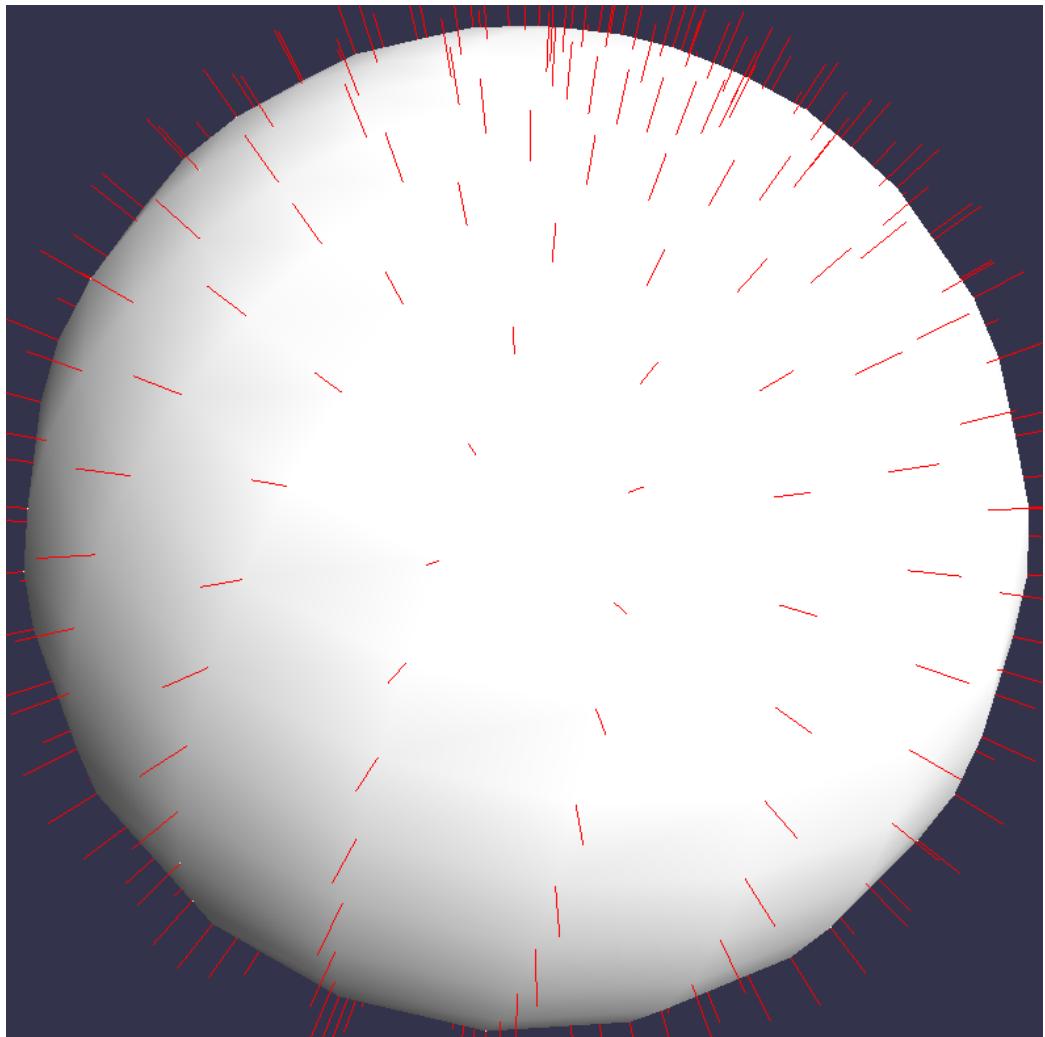


Figure 10: Sphère avec normales explicitées

2.4 Ajout des couleurs

Par la suite, j'ai ajouté des couleurs aux sommets de la sphère. Pour cela, j'ai choisi arbitrairement de prendre la valeur absolue de la position de chaque sommet et de lui attribuer en couleur.

```
o_mesh.colors.push_back(Vec3(abs(x),abs(y),abs(z)));
```

Figure 11: Code C++ pour rajouter des couleurs aux sommets

Grâce à une interpolation des couleurs bienvenue lors du rendu des faces, voici le résultat.



Figure 12: Implémentation des couleurs sur la sphère

2.5 Variation des méridiens et parallèles

Il était par la suite demandé d'implémenter un mécanisme permettant d'augmenter et de diminuer le nombre de méridiens et de parallèles sur la sphère lors de l'appui respectif sur les touches + et -. Grâce à la fonction key, déjà implémentée dans le code, il fut facile de rajouter un cas sur les deux touches concernées. J'ai pour cela créé deux variables globales, l'une correspondant au nombre de méridiens et l'autre au nombre de parallèles. Je les incrémente ou décrémente selon la touche appuyée, puis je rappelle la fonction d'affichage de la sphère avec ces nouvelles valeurs.

```
case '+': // Incrémenteur nX_sphere et nY_sphere
    nX_sphere++;
    nY_sphere++;
    setUnitSphere(unit_sphere,nX_sphere,nY_sphere);

case '-': // Décrémenteur nX_sphere et nY_sphere
    nX_sphere--;
    nY_sphere--;
    setUnitSphere(unit_sphere,nX_sphere,nY_sphere);
```

Figure 13: Switch sur le nombre de méridiens et de parallèles de la sphère

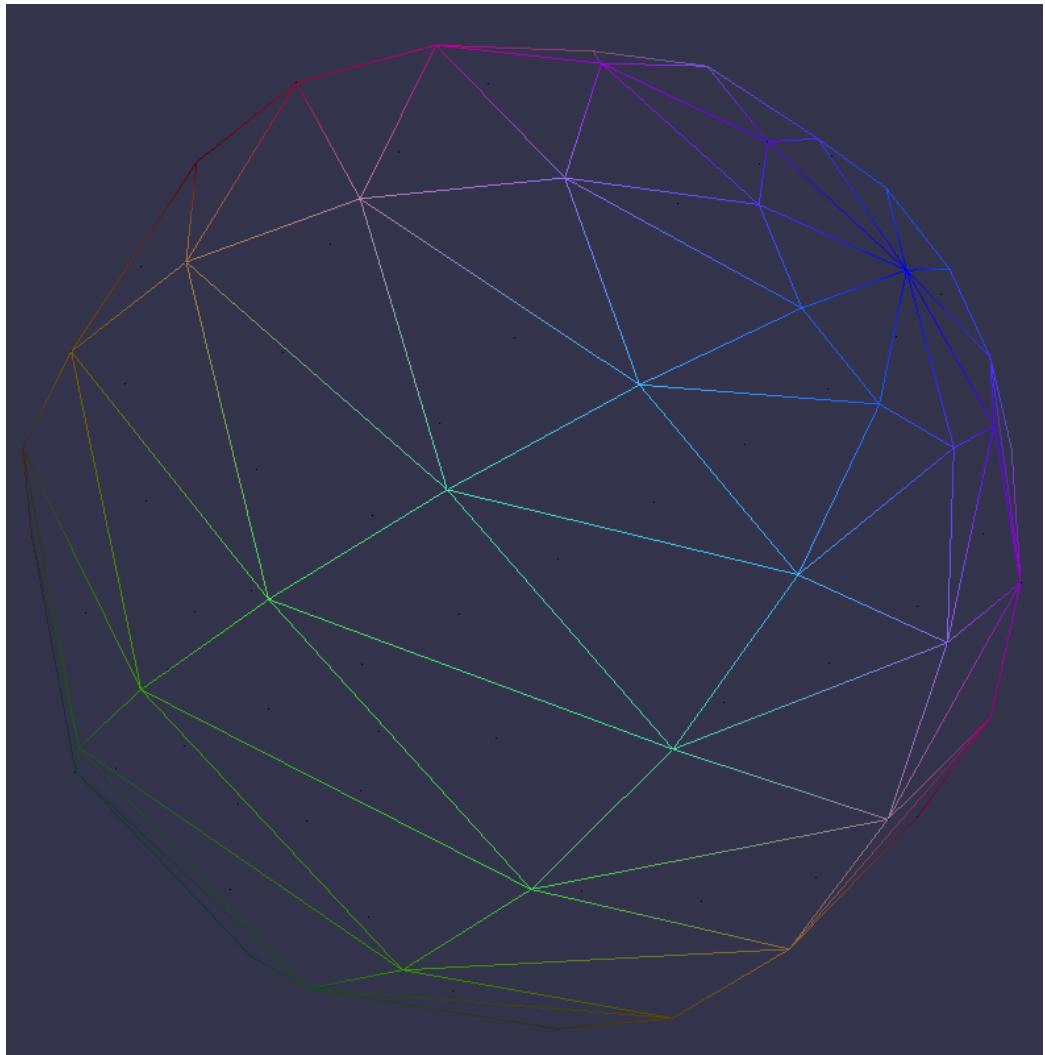


Figure 14: $nX_sphere = nY_sphere = 10$

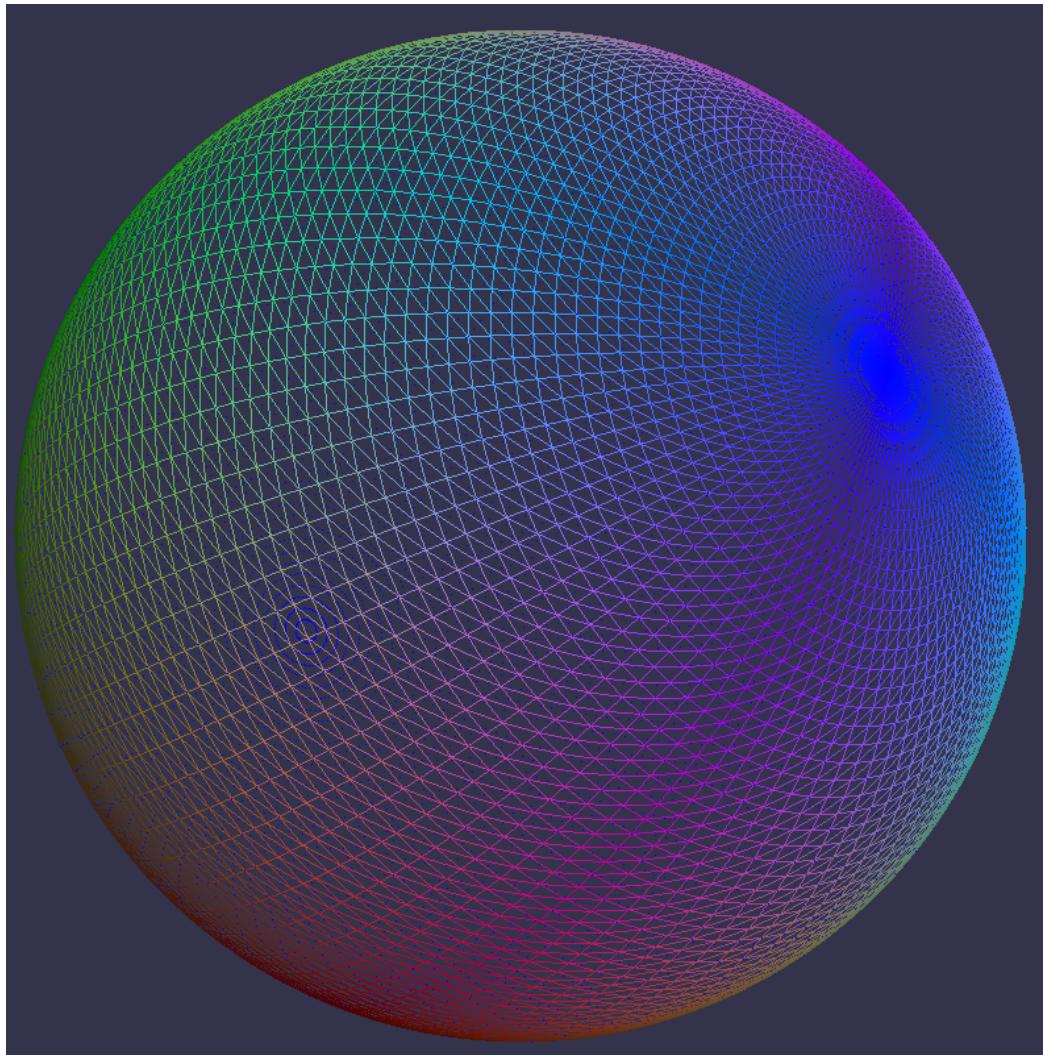


Figure 15: $nX_{sphere} = nY_{sphere} = 100$

2.6 Bruit de Perlin

Par curiosité, j'ai intégré du bruit de Perlin aux positions des sommets de la sphère.

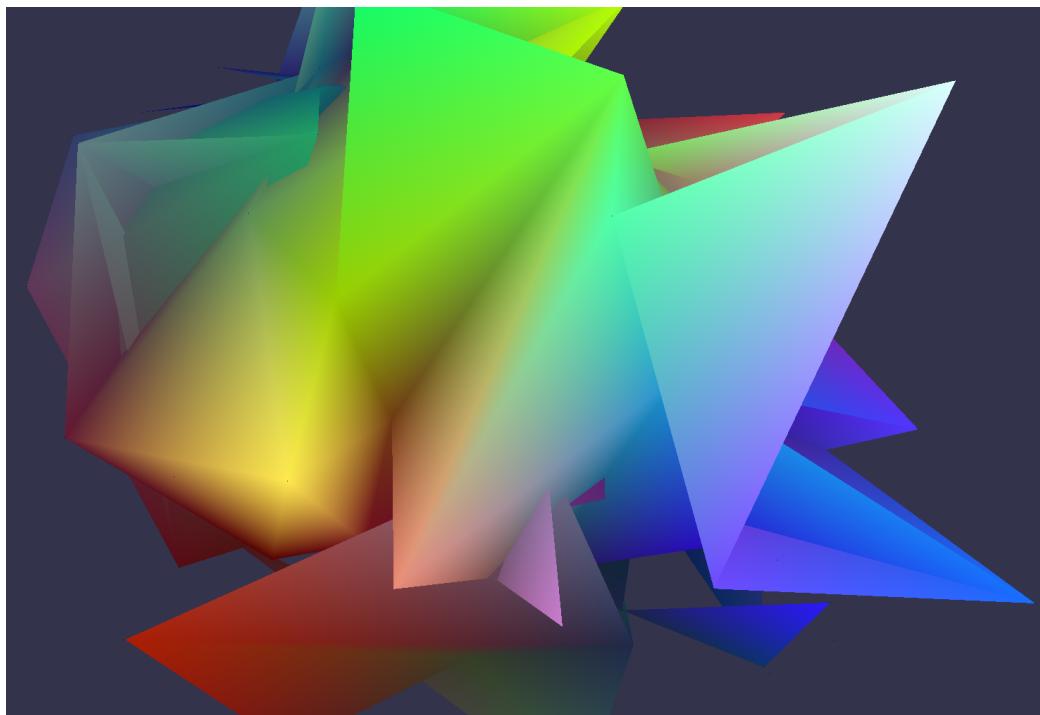


Figure 16: Bruit de Perlin appliqué aux sommets de la sphère

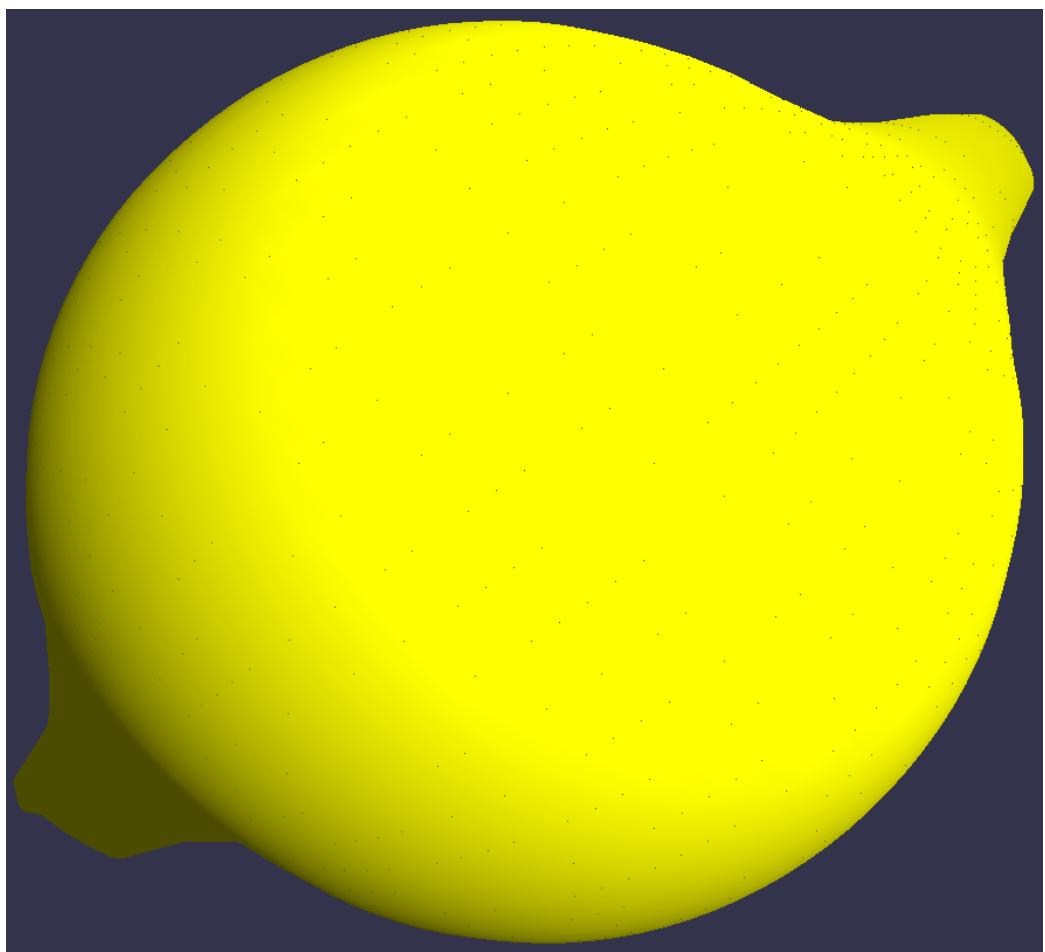


Figure 17: Citron obtenu avec du bruit de Perlin

3 Génération de terrain

3.1 Crédit des points du plan

À l'aide de deux boucles for imbriquées, il a été facile de générer les points de mon plan.

```
for(int i = 0; i < nX_square; i++) {  
    for(int j = 0; j < nY_square; j++) {  
        float x = i;  
        float y = j;  
        o_mesh.vertices.push_back(Vec3(x,y,0));  
    }  
}
```

Figure 18: Code C++ utilisé pour générer les points du plan

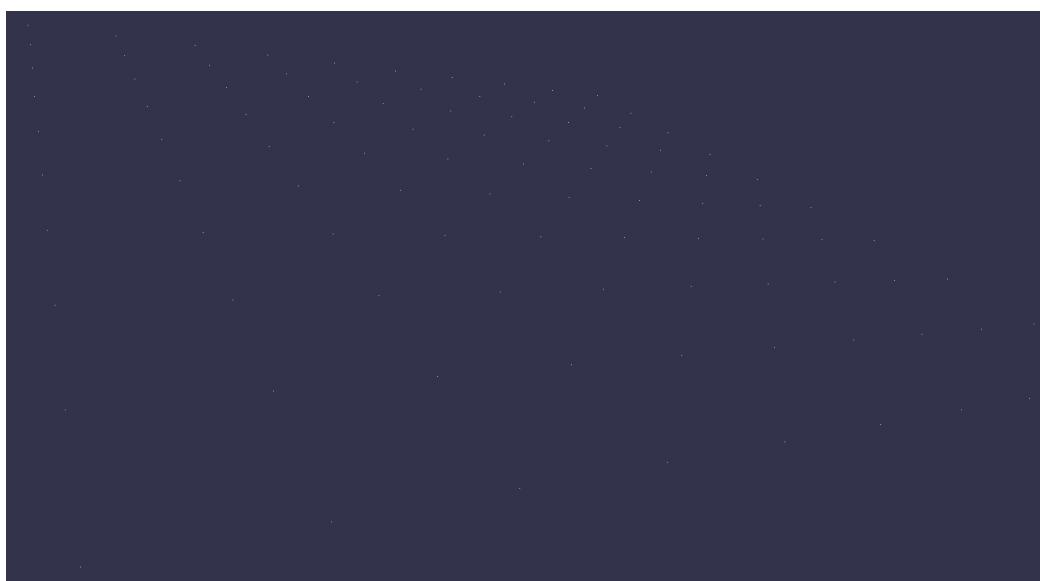


Figure 19: Rendu des points du plan

3.2 Génération des triangles

Pour relier les sommets entre eux, j'ai imaginé dans ma tête comment était disposés les points dans le tableau. Étant donné la méthode de génération des points utilisée, je peux obtenir le sommet situé au-dessus d'un sommet donné en ajoutant le nombre de points par ligne. À partir de là, j'ai pu générer les triangles, d'abord d'une manière (non optimale, voir figure ci-dessous) puis d'une autre manière optimale (voir figure ci-dessous aussi).

```
// Génération des triangles
for(int i = 0; i < nX_square - 1; i++) {
    for(int j = 0; j < nY_square - 1; j++) {
        int index = indiceImage(i,j,nY_square);
        o_mesh.triangles.push_back(Triangle(index, index + nY_square, index + nY_square + 1));
        o_mesh.triangles.push_back(Triangle(index, index + nY_square + 1, index + 1));
    }
}

// La manière ci-dessous était pas mal mais dessinait des triangles sous le plan, mettant à mal le calcul des normales
for(int i = 0; i < (int)o_mesh.vertices.size() - (nX_square + 1); i++) {
    o_mesh.triangles.push_back(Triangle(i,i+nX_square,i+nX_square+1));
    o_mesh.triangles.push_back(Triangle(i,i+nX_square+1,i+1));
}
```

Figure 20: Code C++ implémentant les triangles du plan

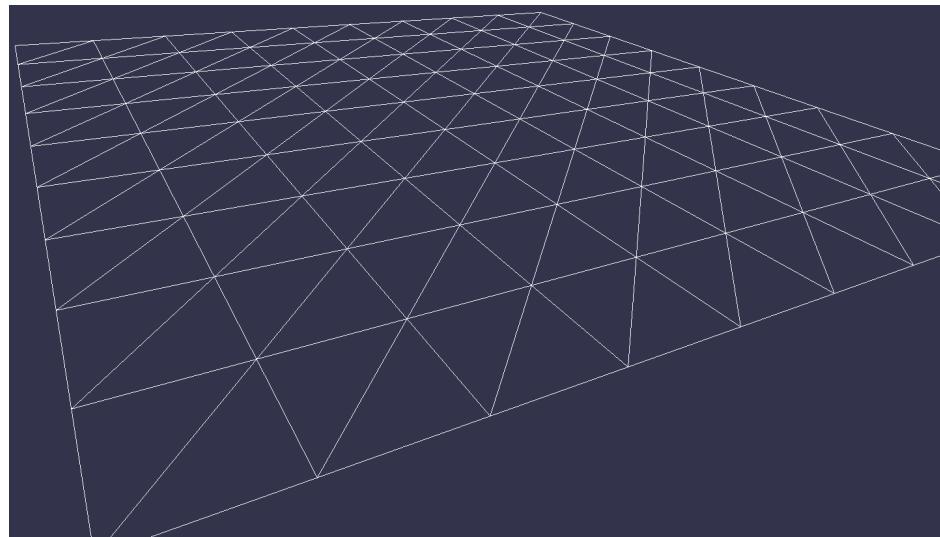


Figure 21: Rendu des triangles du plan

3.3 Ajout de bruit de Perlin sur la composante z

Il était demandé d'ajouter du bruit de perlin sur la coordonnée z. Pour ce faire, j'ai utilisé la fonction GetNoise de l'objet noise.

```
float z = noise.GetNoise((float)i*100,(float)j*100);
```

Figure 22: Code C++ permettant d'ajouter du bruit de Perlin sur l'axe z



Figure 23: Bruit obtenu avec $i*100$ et $j*100$ en arguments



Figure 24: Bruit obtenu avec $\exp(i)$ et $\exp(j)$ en arguments

3.4 Calcul des normales

Dans ce contexte, le calcul des normales aux sommets était moins évident que pour la sphère, surtout avec l'ajout du bruit sur la composante z (les normales sur un plan plat étant triviales). J'ai donc utilisé la formule du cours pour calculer les normales à chaque face, puis pour chaque sommet, j'ai ajouté les normales des faces adjacentes à ce sommet, avant d'enfin normaliser le tout pour obtenir les normales aux sommets.

```
// Calcul des normales à chaque sommet

for(int i = 0; i < (int)o_mesh.triangles.size(); i++) {

    Triangle tri = o_mesh.triangles[i];

    Vec3 v0 = o_mesh.vertices[tri[0]];
    Vec3 v1 = o_mesh.vertices[tri[1]];
    Vec3 v2 = o_mesh.vertices[tri[2]];

    Vec3 faceNormal = normalize(crossProduct(v1 - v0,v2 - v0));

    for(int j = 0; j < 3; j++) {
        o_mesh.normals[tri[j]] += faceNormal;
    }
}

for(int i = 0; i < (int)o_mesh.normals.size(); i++) {

    o_mesh.normals[i] = normalize(o_mesh.normals[i]);
}
```

Figure 25: Code C++ implémentant les normales aux sommets sur le plan



Figure 26: Rendu avec normales

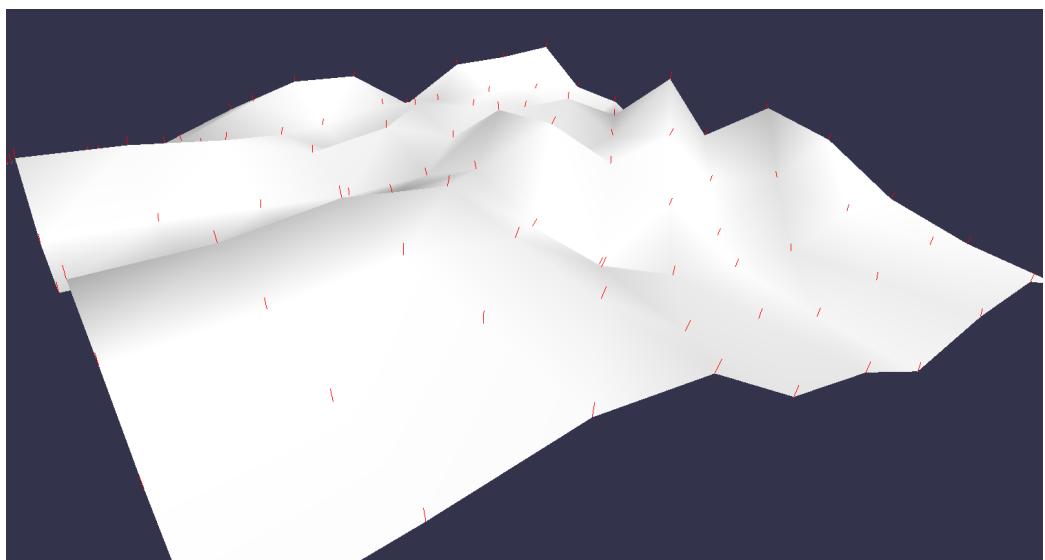


Figure 27: Rendu avec normales explicitées

Pour faciliter l'observation et la navigation dans la fenêtre de rendu de ce TP, je me suis permis de modifier la longueur de l'affichage des vecteurs des normales, et d'accélérer la vitesse de déplacement avec la souris lors du clic droit.

```
if(display_normals){  
    glLineWidth(1.);  
    glColor3f(1.,0.,0.);  
    for(unsigned int pIt = 0 ; pIt < i_mesh.normals.size() ; ++pIt) {  
        Vec3 to = i_mesh.vertices[pIt] + 0.1*i_mesh.normals[pIt];  
        drawVector(i_mesh.vertices[pIt], to);  
    }  
}
```

Figure 28: Modification de la taille des normales (modification en surbrillance)

```
void motion (int x, int y) {  
    if (mouseRotatePressed == true) {  
        camera.rotate (x, y);  
    }  
    else if (mouseMovePressed == true) {  
        camera.move (((x-lastX)*5)/static_cast<float>(SCREENWIDTH), ((lastY-y)*5)/static_cast<float>(SCREENHEIGHT), 0.0);  
        lastX = x;  
        lastY = y;  
    }  
}
```

Figure 29: Modification de la vitesse de déplacement lors du clic droit (modification en surbrillance)

3.5 Ajout de couleur en fonction de l'orientation des normales

Ici, il fallait colorier le sommet en vert si sa normale était verticale, en brun sinon. Comment savoir si une normale est verticale ? Tout simplement en regardant sa composante sur z, si la valeur absolue (au cas où la normale aille vers le bas, mais ici ce n'est pas censé être le cas) de cette dernière est égale à 1 (car les vecteurs sont normalisés), alors la normale est verticale. Comme nous travaillons sur des machines et en nombres flottants, il est quasi-impossible d'obtenir le nombre 1 exact, j'ai donc utilisé un seuil à 0.99 (quasi-vertical).

```
// Coloration des normales selon leur orientation (vert si verticale, brun sinon)

float vertical = 0.99; // Comme les normales sont normalisées, une normale verticale aura une coordonnée z = 1, ici on met 0.99 comme seuil car 1 est idéaliste

for(int i = 0; i < (int)o_mesh.normals.size(); i++) {
    if(fabs(o_mesh.normals[i][2]) >= vertical) {
        o_mesh.colors.push_back(Vec3(0,1,0)); // Coloration en vert
    }
    else {
        o_mesh.colors.push_back(Vec3(0.5,0.2,0)); // Coloration en brun
    }
}
```

Figure 30: Code C++ pour implémenter la coloration des sommets selon l'orientation des normales

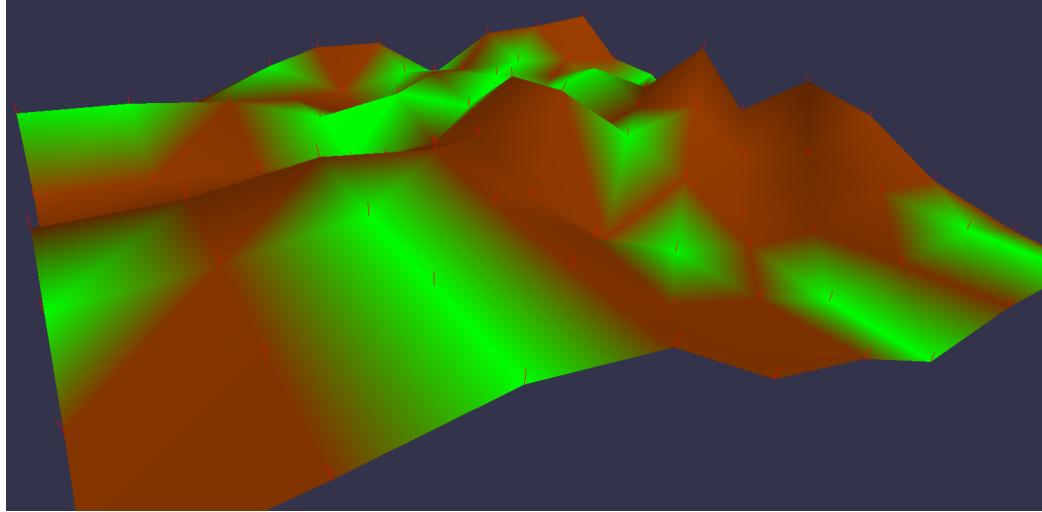


Figure 31: Coloration des sommets en fonction des normales

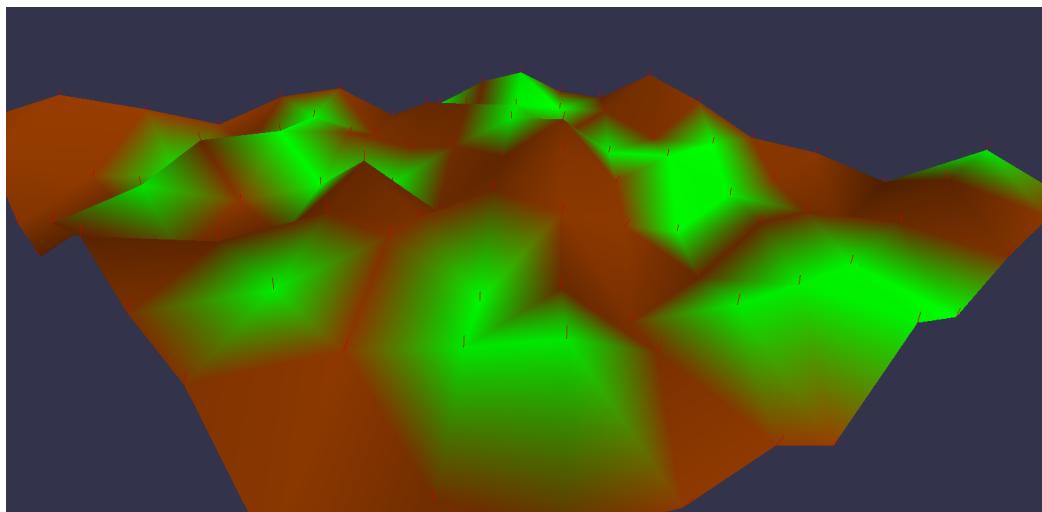


Figure 32: Autre exemple avec un bruit sur z différent

3.6 Ajout d'un décalage sur le bruit

Pour décaler le calcul du bruit selon x et y lors de l'appui sur une touche, j'ai là encore créé deux variables globales, X_noise et Y_noise. J'ai rajouté ces variables dans le calcul de la position z des sommets. De cette manière, l'incrémentation et la décrémentation de ces variables auront un effet sur la composante z du sommet, en décalant le bruit.

J'ai aussi, exactement de la même manière que pour la sphère, permis d'augmenter/diminuer la taille du plan en appuyant sur +/-.

```
// Génération des points

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

    for(int j = 0; j < nY_square; j++) {

        float x = i;

        float y = j;

        float z = noise.GetNoise((float)i*X_noise,(float)j*Y_noise);

        o_mesh.vertices.push_back(Vec3(x,y,z));

    }

}
```

Figure 33: Code C++ pour permettre l'ajout et le décalage de bruit sur la composante z

```

case 'd': // Incrémenter le bruit en X pour le plan
    X_noise+=100;
    setTesselatedSquare(tesselation,nX_square,nY_square);
    break;

case 'q': // Décrémenter le bruit en X pour le plan
    X_noise-=100;
    setTesselatedSquare(tesselation,nX_square,nY_square);
    break;

case 'z': // Incrémenter le bruit en Y pour le plan
    Y_noise+=100;
    setTesselatedSquare(tesselation,nX_square,nY_square);
    break;

case 's': // Décrémenter le bruit en Y pour le plan
    Y_noise-=100;
    setTesselatedSquare(tesselation,nX_square,nY_square);
    break;

```

Figure 34: Switch C++ pour implémenter le décalage du bruit lors de l'appui sur les touches z,q,s et d

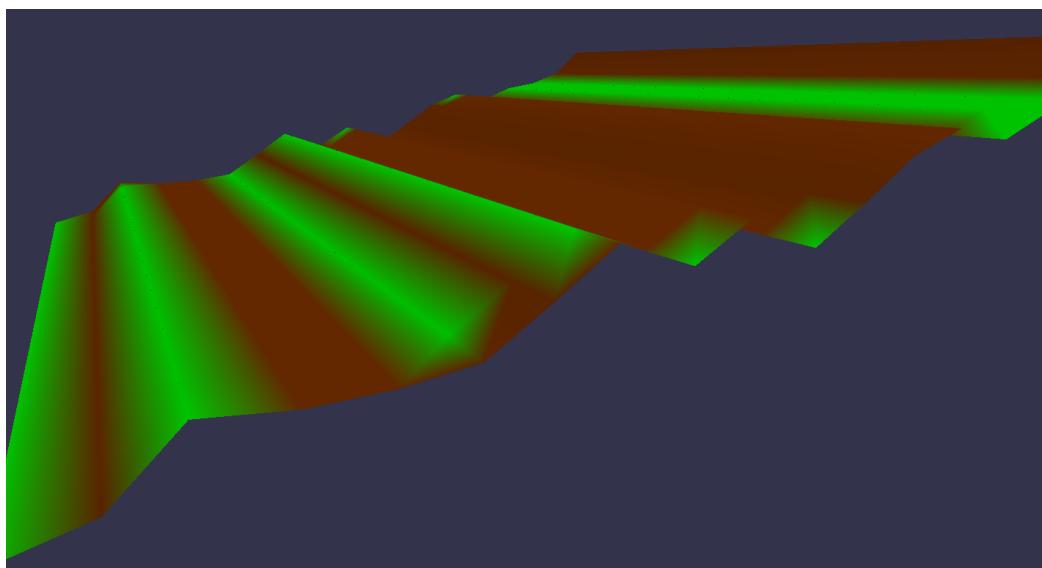


Figure 35: Exemple après plusieurs pressions sur la touche d

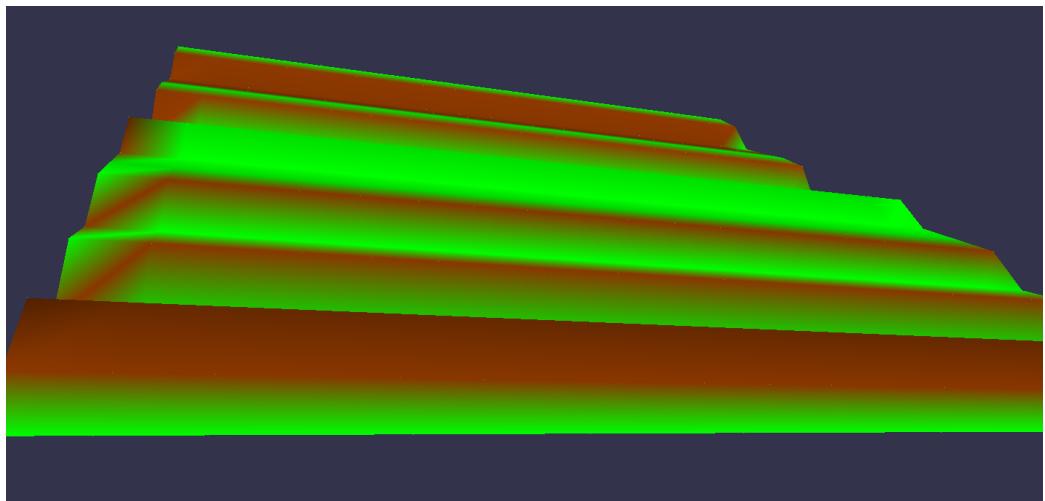


Figure 36: Exemple après plusieurs pressions sur la touche z

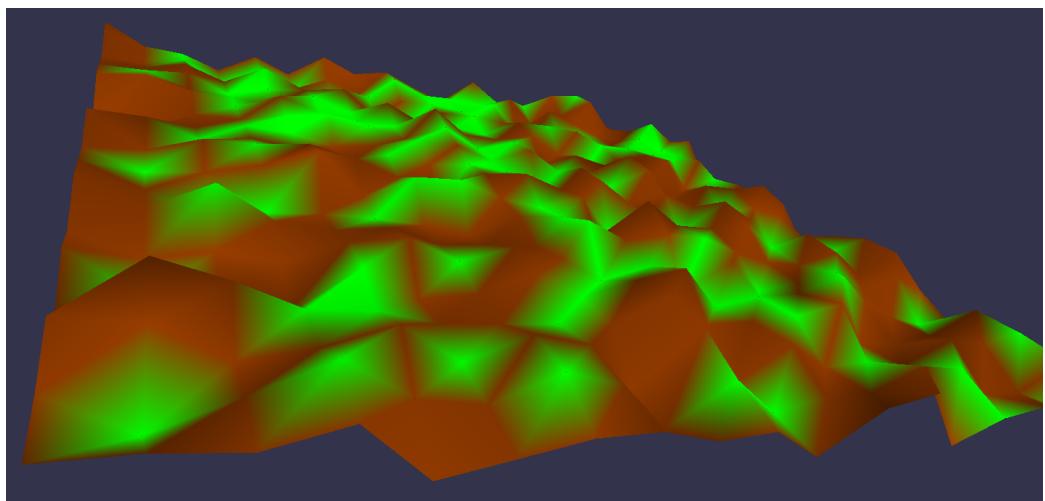


Figure 37: Exemple après plusieurs pressions sur la touche z puis sur la touche d

4 Bonus : bugs et rendus innatendus

Durant le TP, j'ai pu obtenir des figures plutôt amusantes.

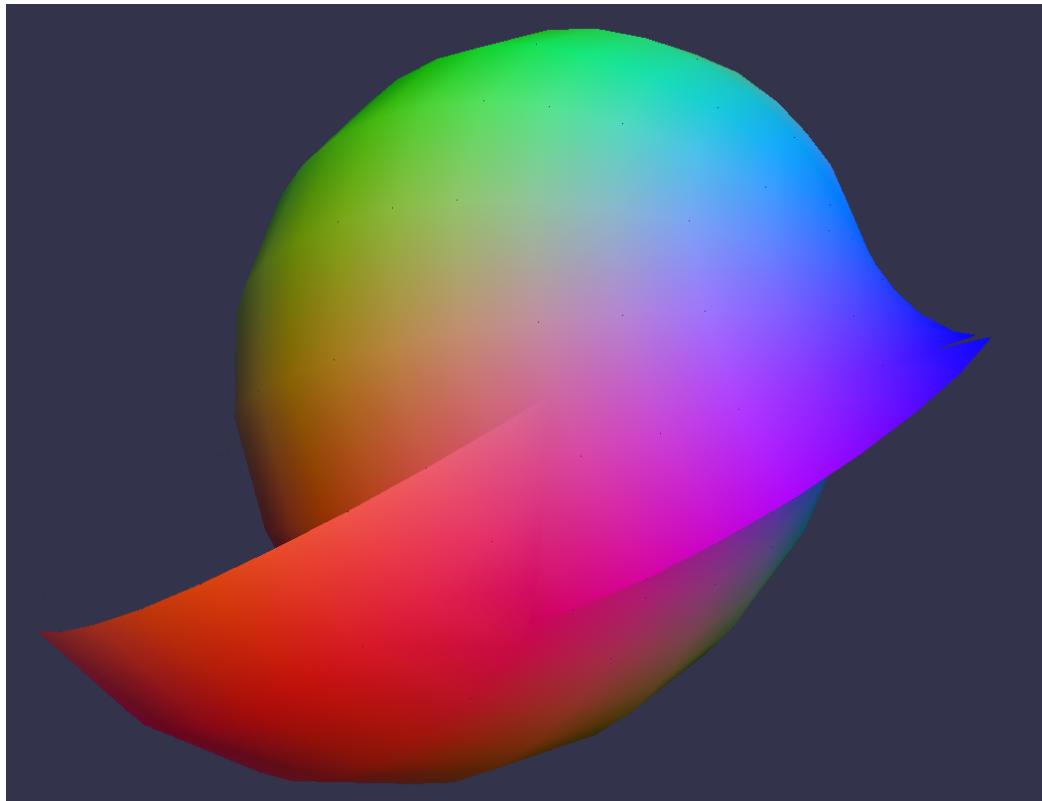


Figure 38: Comme un air d'oiseau de Twitter... (R.I.P)

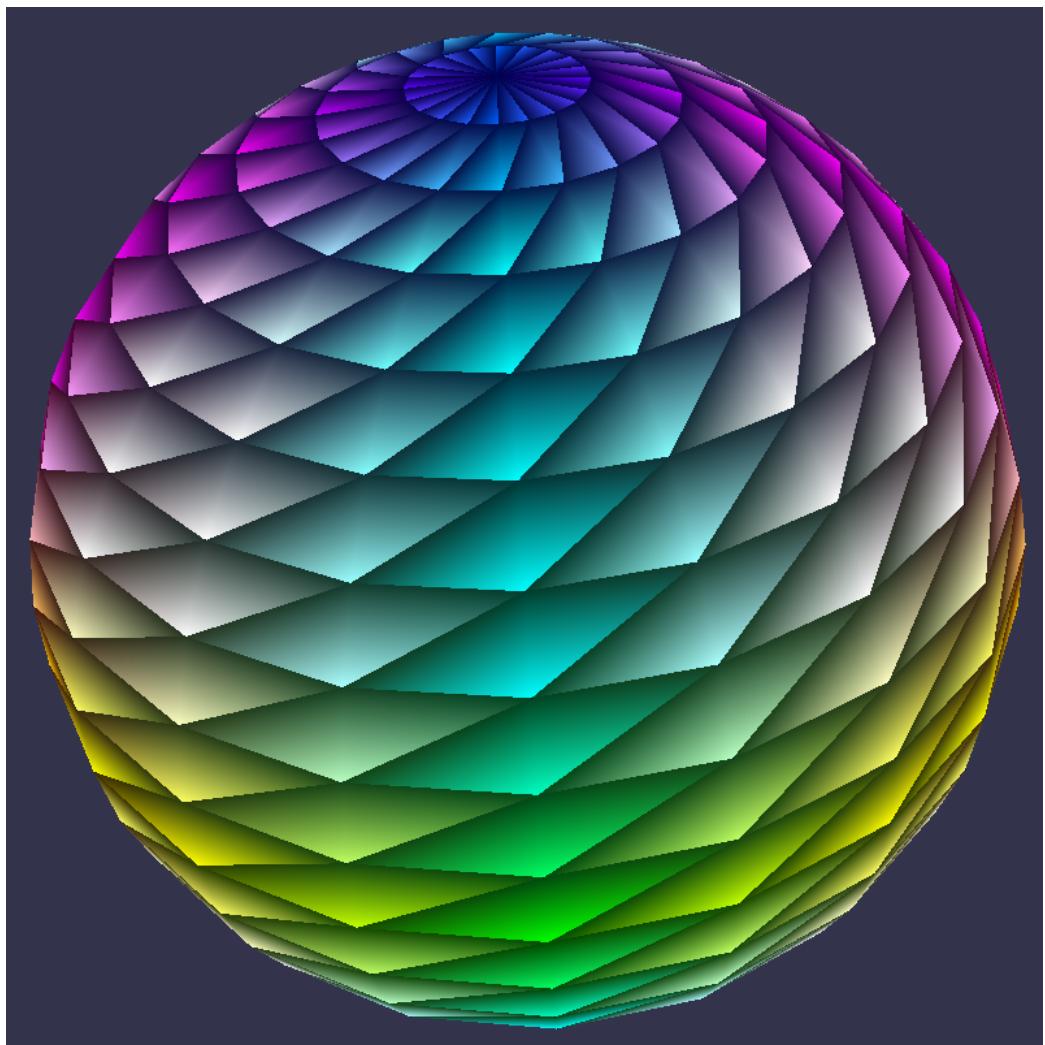


Figure 39: Joli rendu de triangles sur la sphère

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