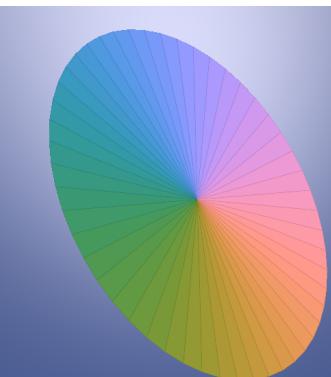


MIF02 TP1 (Sujet 3 : Modélisation de terrains)

Partie 1 : Fondamentaux en modélisation

1.1 Modelisation

Voici les objets générés avec le nombre de primitives, triangles et le temps de génération



Statistics	
Vertex	53
Triangles	52

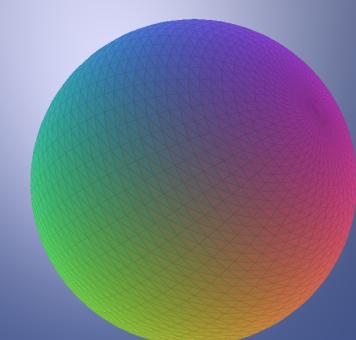
Duration
6ms

Disque : assez simple à réaliser

Calcule des sommets :

(avec alpha = $2\pi / \text{nb_division}$)

```
for (int i = 0; i <= div+1; ++i) {
    alpha = i * step;
    vertices.push_back(Vector(cos(alpha) * r + c[0], c[1], sin(alpha) * r + c[2]));
    normals.push_back(Vector(cos(alpha) * r, 0, sin(alpha) * r));
    AddTriangle(0, i, i + 1, i);
```

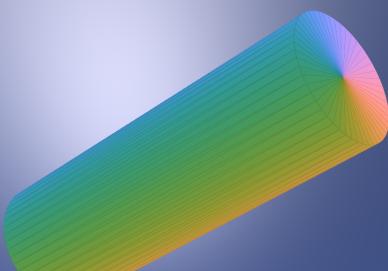


Statistics	
Vertex	2502
Triangles	5002

18ms

Sphère : la création des sommets se base sur l'algo d'une UV sphere

```
for (int i = 0; i < div; i++)
{
    float theta = ((i * M_PI) / div);
    for (int j = 0; j < div; j++)
    {
        float phi = j * ((2 * M_PI) / div);
        float x = std::sin(theta) * std::cos(phi);
        float y = std::cos(theta);
        float z = std::sin(theta) * std::sin(phi);
        vertices.push_back(Vector(x * radius, y * radius, z * radius));
        normals.push_back(Vector(x * radius, y * radius, z * radius));
```

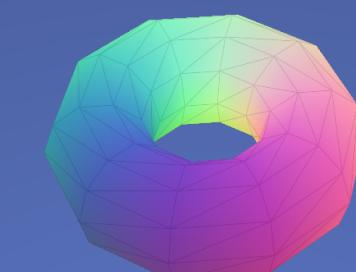


Statistics	
Vertex	202
Triangles	200

7ms

Cylindre : crée les sommets en-bas, puis enhaut

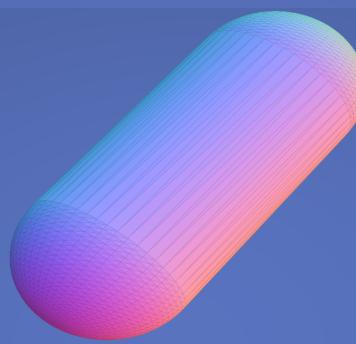
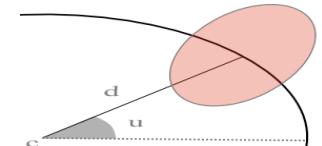
```
for (int i = 1; i <= div*2; ++i) {
    alpha = i * step;
    vertices.push_back(Vector(cos(alpha) * r, -h, sin(alpha) * r));
    normals.push_back(Vector(cos(alpha) * r, 0, sin(alpha) * r));
    vertices.push_back(Vector(cos(alpha) * r, h, sin(alpha) * r));
    normals.push_back(Vector(cos(alpha) * r, 0, sin(alpha) * r));
```



Statistics	
Vertex	100
Triangles	200

6ms

Tore : crée des sommets en cercle autour de chaque sommet



Statistics	
Vertex	5206
Triangles	10204

16ms

```
for (float i = 0; i < divR; i++) {
    for (float j = 0; j < divT; j++) {
        float u = j / divT * M_PI * 2.0;
        float v = i / divR * M_PI * 2.0;
        float x = (tore.getRadius() + tore.getThickness() * std::cos(v)) * std::cos(u);
        float y = (tore.getRadius() + tore.getThickness() * std::cos(v)) * std::sin(u);
        float z = tore.getThickness() * std::sin(v);
        vertices.push_back(Vector(x, y, z));
        normals.push_back(Vector(x, y, z));
```

Capsule : il suffit simplement de créer les sommets d'un cylindre, puis de deux demi-sphères

2.2 Transformation

Classe Matrix : admet les méthodes de multiplication, inverse et transposé

La matrice est représenté par une tableau à 2 dimensions et initialisé en tant qu'une matrice d'identité

```
double **tab; //!< 2d array of the matrix (3x3)
Matrix();
Matrix(double **tab);

bool Inverse(Matrix& m);

void Transpose();

Vector operator*(Vector &vector);
```

Transformations affines : 3 transformations ont été ajouté

- *Scale* : `vertices[i] *= s;`

- *Translation* : `vertices[i] += Vector(x,y,z);`

- *Rotation* : multiplication d'une matrice de rotation par un sommet

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta \\ 0 & \sin \theta & \cos \theta \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos \theta & 0 & \sin \theta \\ 0 & 1 & 0 \\ -\sin \theta & 0 & \cos \theta \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

```
void Scale(double);
void Translation(float x, float y, float z);
void RotaionX(double deg);
void RotaionY(double deg);
void RotaionZ(double deg);
```

Merge : Ajout des tableaux du mesh passé à la fin des tableaux du mesh initial

```
for (int i = 0; i < m.varray.size(); i++)
{
    varray.push_back(vertices.size() + m.varray[i]);
}

for (int i = 0; i < m.narray.size(); i++)
{
    narray.push_back(normals.size() + m.narray[i]);
}

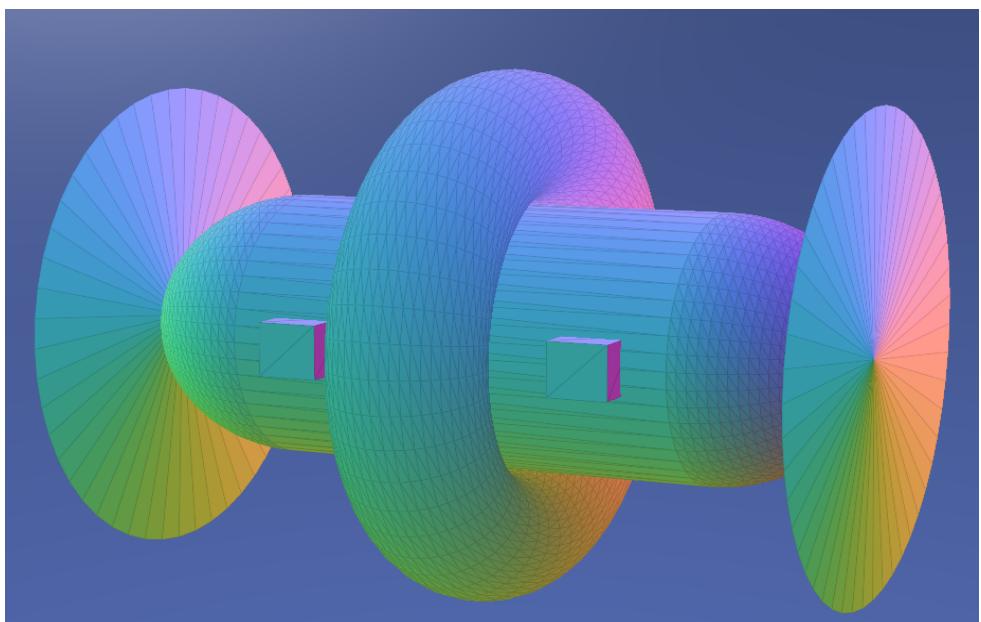
for (int i = 0; i < m.vertices.size(); i++)
{
    vertices.push_back(m.vertices[i]);
}

for (int i = 0; i < m.normals.size(); i++)
{
    normals.push_back(m.normals[i]);
}
```

2.3 Déformation :

SphereWarp : pas réalisé

Objet complexe :

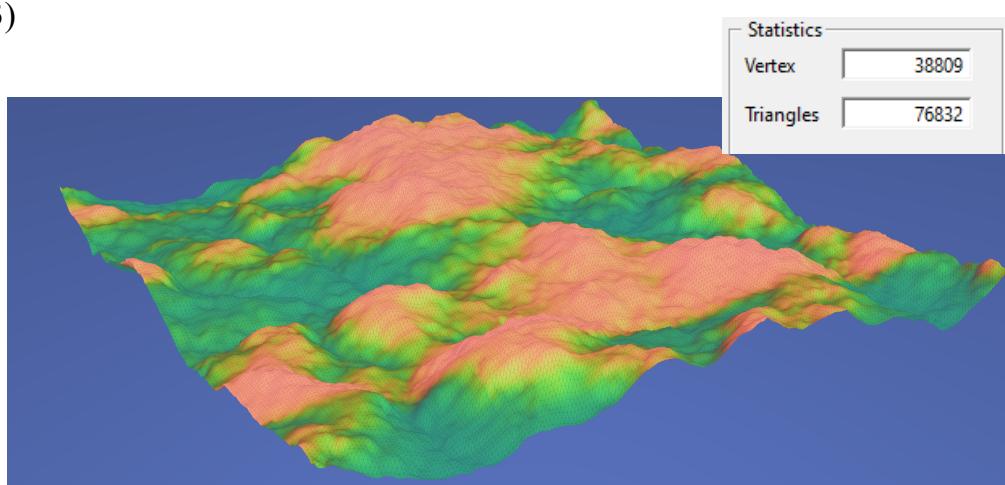


Partie 2 : Modélisation de terrains (TP3)

3 Modélisation

Classe HeightField : admet 2 constructeurs

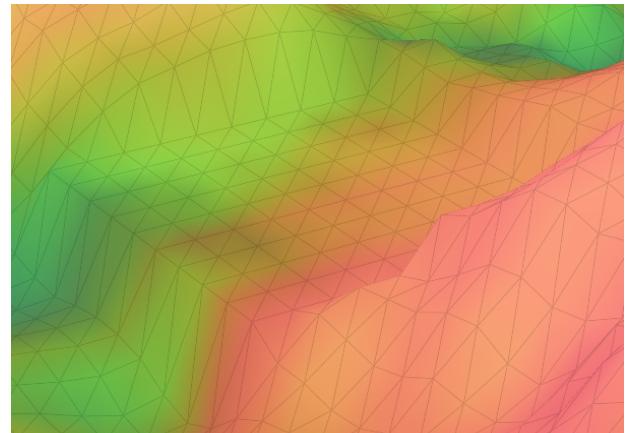
- sans image (avec des hauteurs choisi au hasard de 0 à 1)
- avec une image passer en paramètre qui à partir de son niveau gris calcule la hauteur de 0 à noise (noise la valeur qui modifie l'intensité de la hauteur (l'intervalle avec noise = 1 est de 0 à 255)



```
HeightField::HeightField(int nn, int mm, QImage im, int noise) {
    im = im.scaled(nn+1, mm+1);
    grid = new double* [nn];
    for (int i = 0; i <= nn; ++i) {
        grid[i] = new double[mm];
        for (int j = 0; j <= mm; ++j) {
            double grey =(double)qGray(im.pixel(i,j));
            double scaledGrey = (noise - 0)*((grey - 0) / (255)) - 0;
            grid[i][j] = scaledGrey;
        }
    }
}
```

```
for (int i = 0; i <= hf.getM(); i++) {
    for (int j = 0; j <= hf.getN(); j++) {
        vertices.push_back(Vector(i, j, hf.getHeight(i,j)));
        normals.push_back(-Vector(hf.getNormal(i,j)));
    }
}

int step = 0;
for (int i = 0; i < hf.getM(); i+=1) {
    for (int j = 0; j < hf.getN(); j += 1) {
        AddQuadrangle(step, step + hf.getN() + 1, step + hf.getN() + 2, step + 1);
        step++;
    }
    step++;
}
```

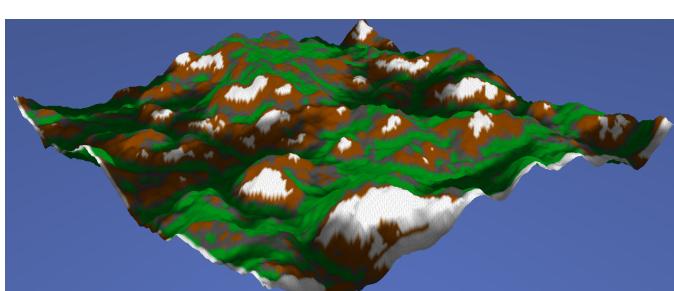


Terrassement : la fonction terrassement prend en paramètre un sommet et modifie la hauteur des sommets dans un périmètre passé en paramètre également

```
for (int i = 0; i < vertices.size(); ++i) {
    if (vertices[i][0] == x && vertices[i][1] == y) {
        for (int ii = x-d; ii < x+d; ++ii) {
            for (int jj = y-d; jj < y+d; ++jj) {
                modifyHeight(ii, jj, h);
            }
        }
        break;
    }
}
```

Coloration :

Cast un Ray de la position du camera, fait des petits pas vers le sommet choisi, puis en fonction de la hauteur calculée à l'intersection colorie le terrain. Plus on ajoute de minStep, plus on sera précis.



```
const float stepValue = 0.05f;
const float minStep = 0.001f;
const float maxStep = 100.0f;
float lh = 0.0f;
float ly = 0.0f;
for (float step = minStep; step < maxStep; step += stepValue)
{
    Vector d = Vector(rd[0] - (ro[0] * step), rd[1] - (ro[1] * step), rd[2] - (ro[2] * step));
    Vector p = ro + d;

    const float h = hf.getHeight(p[0], p[1]);
    if (p[2] < h && h != -1)
    {
        rest = step - stepValue + stepValue * (lh - ly) / (p[2] - ly - h + lh);
        return true;
    }
    lh = h;
    ly = p[2];
}
return false;
```

