Rapport IIG3D

Gauthier BOUYJOU

fait le 24 février 2019

1. Construction des maillages

a) UV_SPHERE

La construction se fait de la même manière que l'on coupe un melon en quartier, puis en carré.

Le but étant de trouver chaque sommet de chaque carré qui formeront par la suite deux triangles.

On peut savoir d'avance le nombre de carré, de triangle et de sommet :

```
int nbRect = _nbSlices * (_nbRings - 1);
int nbTriangle = nbRect * 2 + _nbSlices * 2;
int nbVertices = nbRect + _nbSlices + 2;
```

_nbSlices et _nbRings avec le rayon seront les seuls paramètres de la UV sphère.

On initialise chaque vecteurs :

```
_vertices.clear();
_normals.clear();
_indices.clear();
_vertices = std::vector<GLfloat>(3 * nbVertices);
_normals = std::vector<GLfloat>(3 * nbVertices);
_indices = std::vector<GLuint>(3 * nbTriangle);
```

On crée d'avance les deux pôles (leurs indices se trouve en fin de vecteur) :

```
int iNorthPole = 3 * nbVertices - 6;
_vertices[iNorthPole] = 0;
_vertices[iNorthPole + 1] = _radius;
_vertices[iNorthPole + 2] = 0;
_normals[iNorthPole] = 0;
_normals[iNorthPole + 1] = 1;
_normals[iNorthPole + 2] = 0;

int iSouthPole = 3 * nbVertices - 3;
_vertices[iSouthPole] = 0;
_vertices[iSouthPole + 1] = -_radius;
_vertices[iSouthPole + 2] = 0;
_normals[iSouthPole] = 0;
_normals[iSouthPole + 1] = -1;
_normals[iSouthPole + 2] = 0;
```

On pré-calcule les delta en radians :

```
float deltaRing = M_PI / (_nbRings + 1);
float deltaSlice = 2 * M_PI / _nbSlices;
```

On va boucler du pôles nord au sud, en parcourant tous les quartiers à chaque fois.

```
for (int iRing = 0, iVertice = 0; iRing < _nbRings; iRing++) {
  float teta = (iRing + 1) * deltaRing;

  for (int iSlice = 0; iSlice < _nbSlices; iSlice++, iVertice += 3) {
    float alpha = iSlice * deltaSlice;
```

L'angle téta varie de 0 à 180 du pole nord au pole sud (rotation d'axe z).

L'angle alpha varie de 0 à 360 (rotation d'axe y).

On calcule chaque composante du point en haut à gauche du carré courant.

```
float x = sin(teta) * cos(alpha);
float y = cos(teta);     Δ implici
float z = sin(alpha) * sin(teta);
```

On insère ce nouveau point directement dans le vecteur :

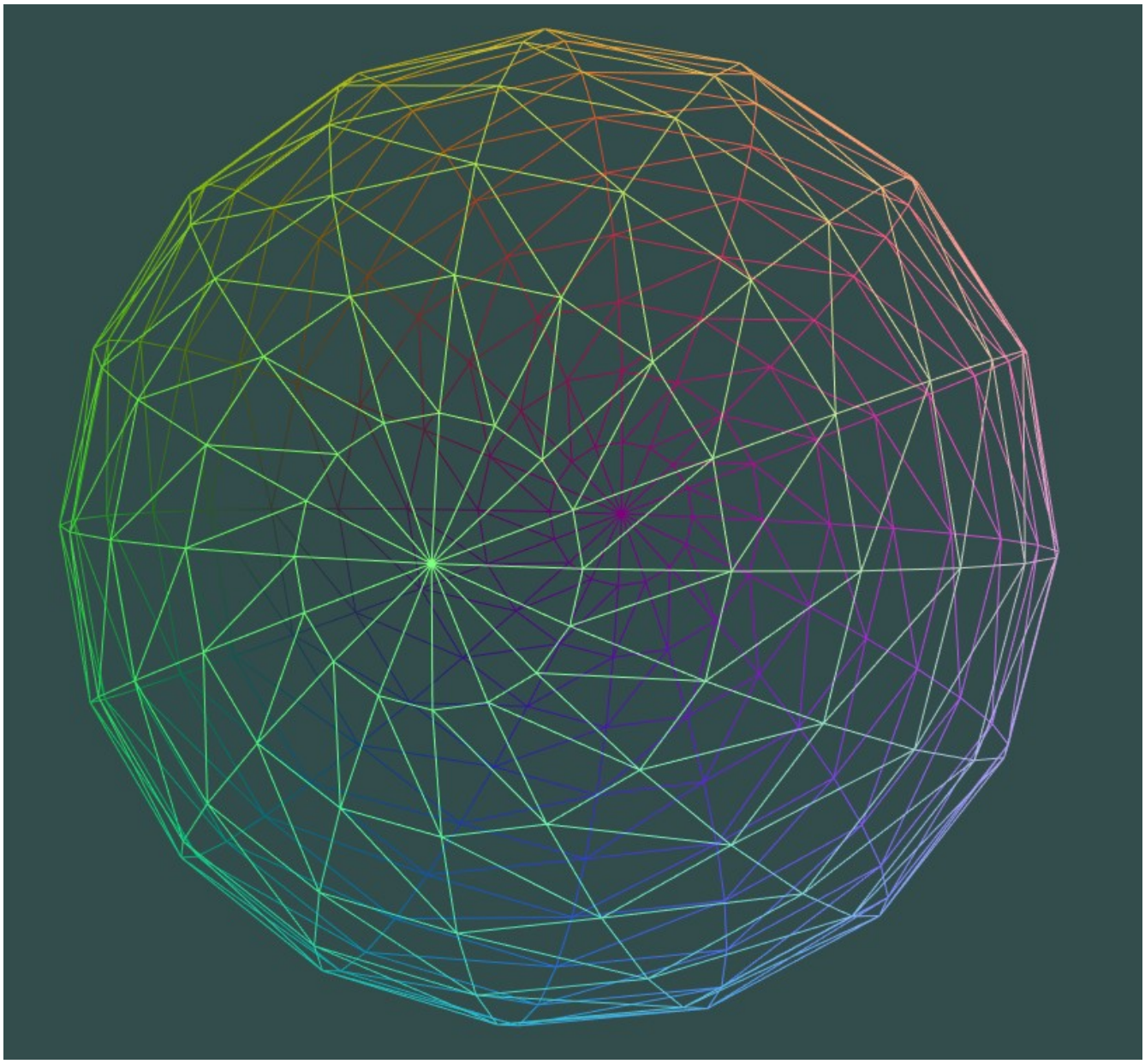
```
_vertices[iVertice] = x * _radius;
_vertices[iVertice + 1] = y * _radius;
_vertices[iVertice + 2] = z * _radius;
```

Il faut ensuite renseigner les triangles dans le vecteurs indices.

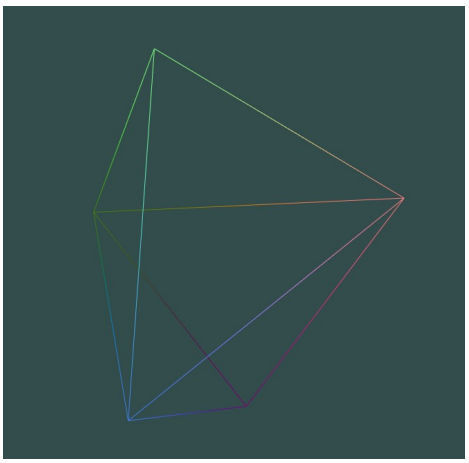
Il y a trois cas particuliers (voir code pour plus de détails).

1. Si on est sur la première itération en haut, on doit renseigner chaque triangle qui on pour coté les deux cotés hauts du rectangle courant et le pôle nord.
2. Si on est sur la dernière itération en bas, idem (1.) mais symétrique.

3. Pour une hauteur donné, si on est en fin de tranche il faut recoller avec la première tranche.Voilà on a tous les indices de sommets, on peut enfin rendre.



UV sphere (226 sommets, 448 triangles).



Plus petite UV sphère que l’on peut avoir :

(5 sommets, 6 triangles)

b) ICO_SPHERE

Première partie : création de l'icosaèdre à l'aide du solide de Platon

On doit partir sur un icosaèdre déjà définit, à l'aide du nombre d'or on peut avoir un icosaèdre à 12 sommets.

```
float t = (1.0 + sqrt(5.0)) / 2.0; // gold number
```

On connaît les coordonnées des sommets.

```
_nbVertices = 12;
_nbTriangles = 20;

_vertices = {-1, t,  0, 1, t,  0, -1, -t, 0, 1, -t, 0,  0, -1, t,  0, 1, t, 0,
             -1, -t, 0, 1, -t, t, 0,  -1, t, 0, 1,  -t, 0, -1, -t, 0, 1};
```

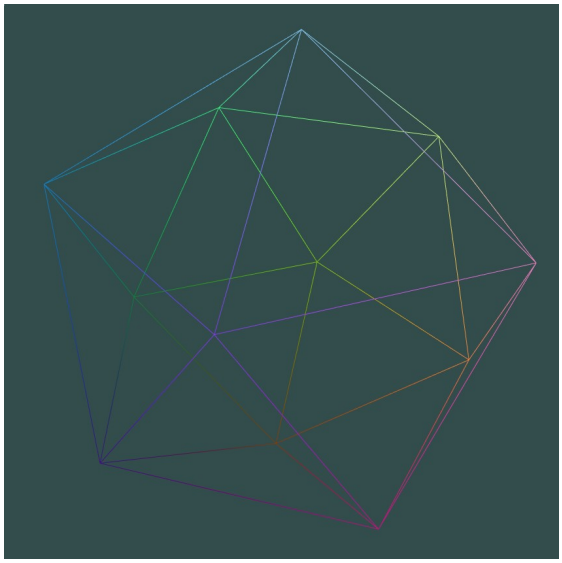
Pour chaque sommets on calcule les normales au points, et on s'assure que chaque point sont à une distance de 1 du centre (0, 0, 0).

Le rayon de la sphère circonscrite est normalisé à un de cette façon, se qui permet lors de la division un temps de calcul en moins, pour changer de rayon il faudra juste scale.

On connaît les indices à l'avance.

```
_indices = {0, 11, 5, 0, 5,  1,  0,  1,  7,  0,  7, 10, 0, 10, 11,
            1, 5,  9, 5, 11, 4,  11, 10, 2,  10, 7, 6,  7, 1,  8,
            3, 9,  4, 3, 4,  2,  3,  2,  6,  3,  6, 8,  3, 8,  9,
            4, 9,  5, 2, 4,  11, 6,  2,  10, 8,  6, 7,  9, 8,  1};
```

On peut rendre notre premier icosaèdre de 20 faces :



Deuxième partie : division de l’icosaèdre (voir code pour plus de précision)

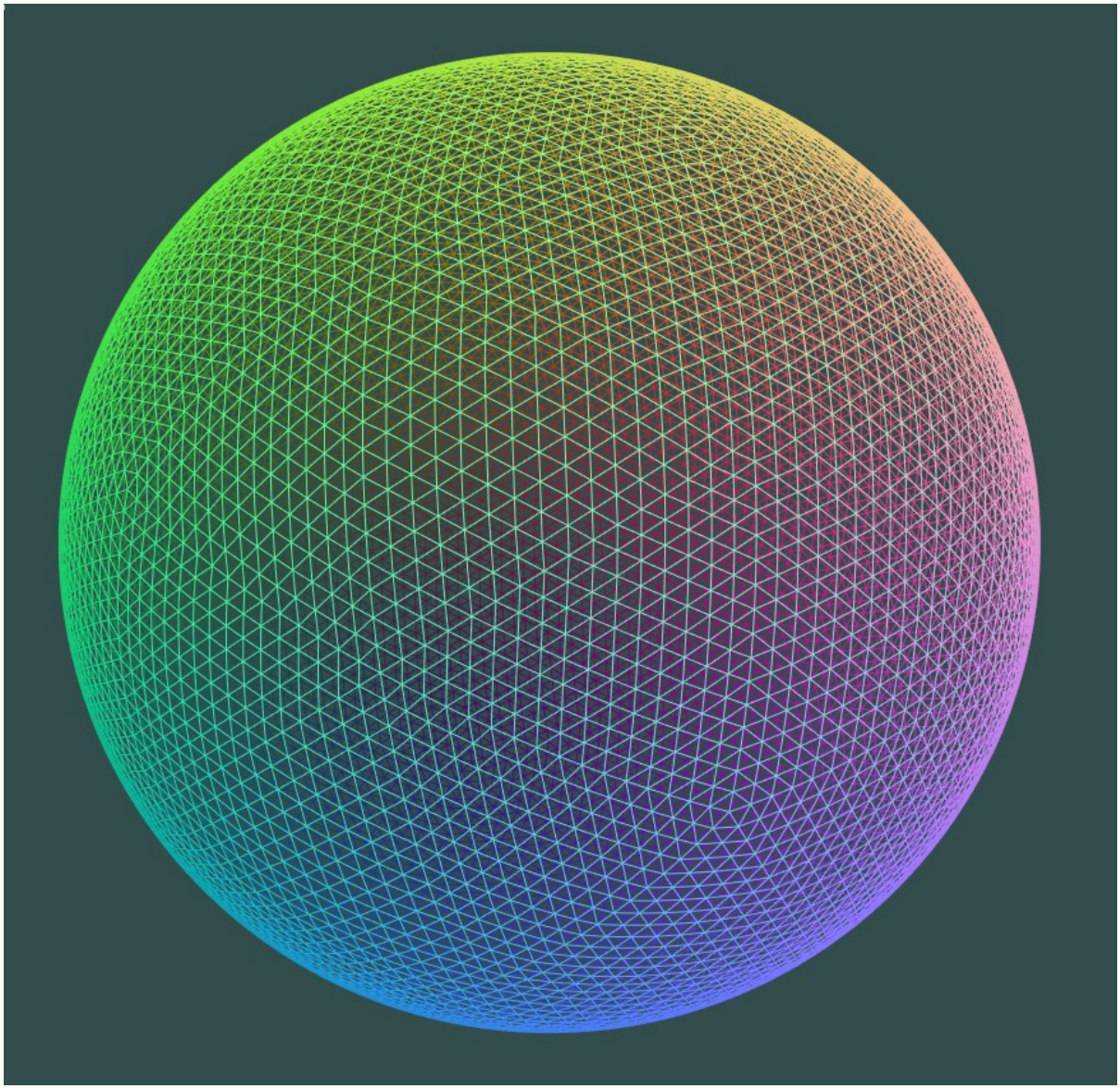
Pour chaque triangles connus, on veut créer 4 triangles en coupant les cotés en deux (+normalisation pour extruder).

Afin d’éviter de recouper des cotés déjà couper par un triangle voisin on utilise un map de paire de l’indice min, indice max de sommet.

Map renvoyant le nouvelle indice de sommet crée durant la division.

On utilise la formule d’Euler comme invariant.

```
assert(_vertices.size() /3 == _nbTriangles / 2 + 2);
```



2. Éclairage diffus

Utilisation d'un vertex shader basique :

```
static const char *vertexshader_source = "#version 410 core\n\
        layout (location = 0) in vec3 position;\n\
        layout (location = 1) in vec3 inormal;\n\
        uniform mat4 model;\n\
        uniform mat4 view;\n\
        uniform mat4 projection;\n\
        out vec3 normal;\n\
        out vec3 pos;\n\
        void main()\n\
        {\n\
            pos = vec3(model *vec4(position, 1.0));\n\
            // Note that we read the multiplication from right to left\n\
            gl_Position = projection * view * vec4(pos, 1.0f);\n\
            normal = inormal;\n\
        }\n";
```

Pour le fragment shader, faut calculer la lumière ambiante et diffuse :

```
static const char *fragmentshaderLight_source = "#version 410 core\n\
        in vec3 normal;\n\
        in vec3 pos;\n\
        out vec4 color;\n\
        uniform vec3 lightPos;\n\
        uniform vec3 lightColor;\n\
        uniform vec3 objectColor;\n\
        void main()\n\
        {\n\
            float ambientStrength = 0.1;\n\
            vec3 ambient = ambientStrength * lightColor;\n\
            vec3 norm = normalize(normal);\n\
            vec3 lightDir = normalize(lightPos - pos);\n\
            float diff = max(dot(norm, lightDir), 0.0);\n\
            vec3 diffuse = diff * lightColor;\n\
            vec3 result = (ambient + diffuse) * objectColor;\n\
            color = vec4(result, 1.0);\n\
        }\n";
```

J'ai repris les shader directement à partir du tutoriel.

3. Erreur d’approximation

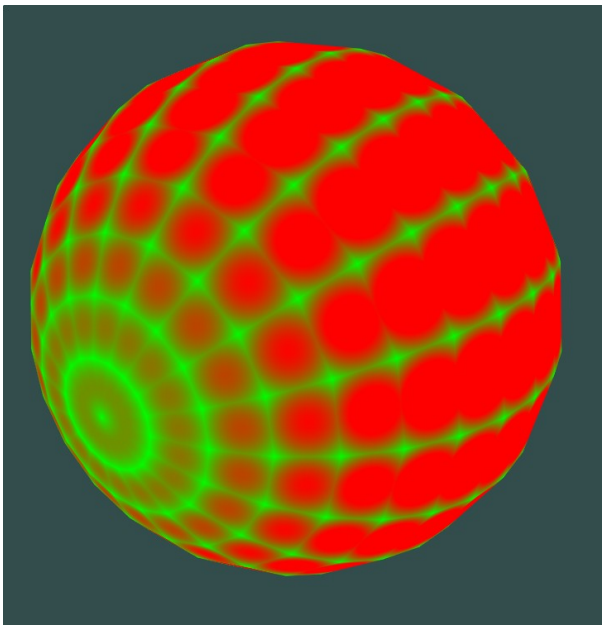
Ce qui est de l’affichage de l’erreur, on utilise un vertex shader :

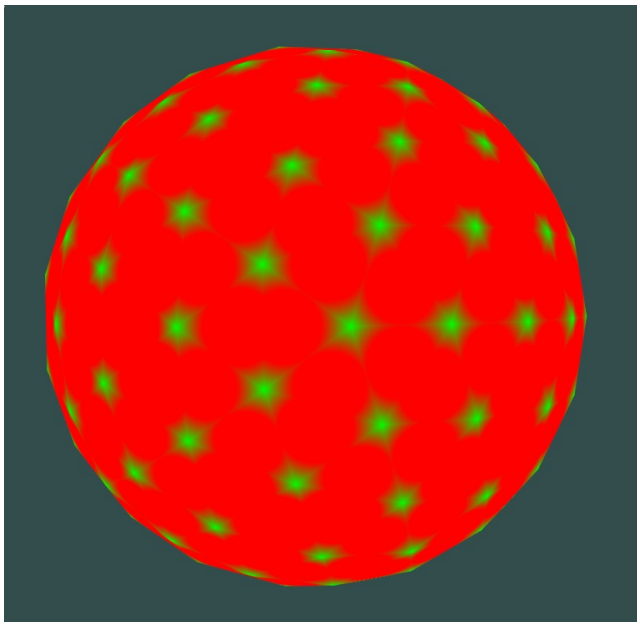
```
static const char *fragmentshaderError_source = "#version 410 core\n\
        in vec3 normal;\n\
        in vec3 pos;\n\
        out vec4 color;\n\
        void main()\n\
        {\n\
        float len = length(pos);\n\
        float err = (1.0 -len) *100;\n\
        color = vec4(err, 1.0 -err, 0.f, 1.0f);\n\
        }\n";
```

length(pos) me renvoie directement la distance du pixel par rapport à l’origine.

L’erreur est calculé en faisant la différence entre un point sur la sphère circonscrite de rayon 1 et le len calculé précédemment.

Le facteur 100 permet de mieux voir les petites différences.



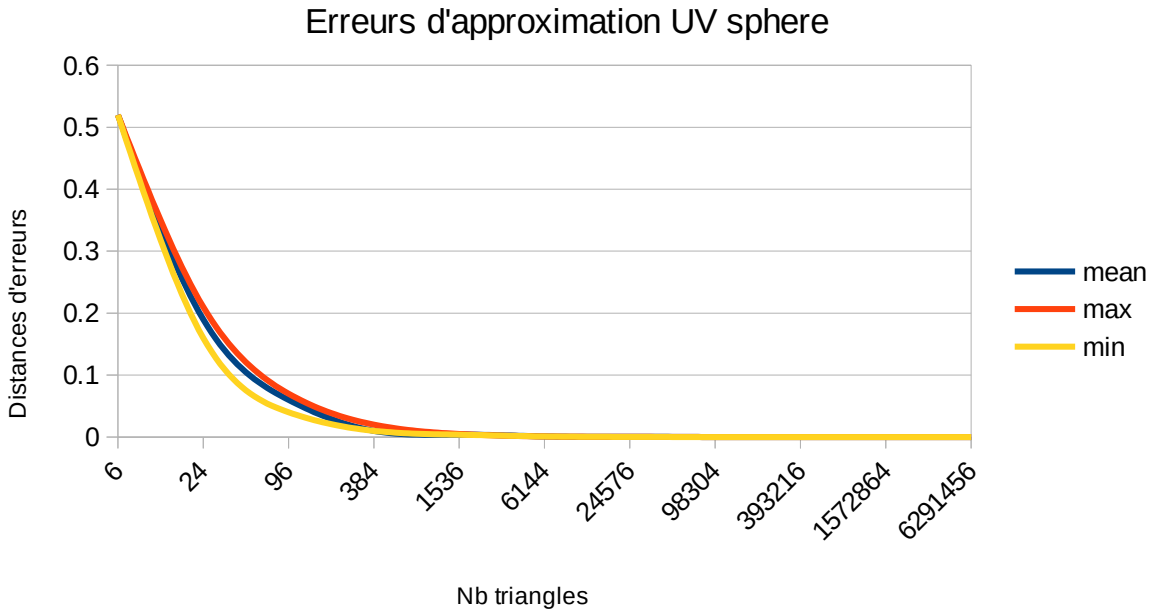


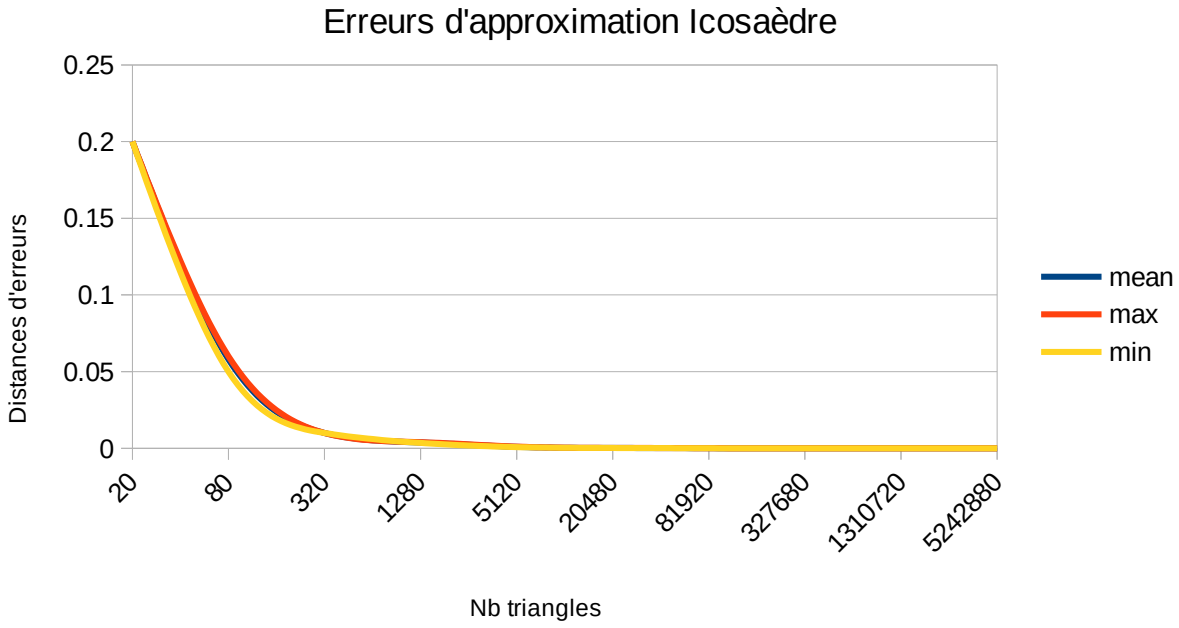
On remarque pour l’UV sphère une plus grande concentration de triangles sur les pôles, ce qui implique une erreurs d’approximation plus forte au niveau de l’équateur, car la surface du rectangle est maximale, la distance avec la sphère circonscrite est la plus grande au centre des rectangle se trouvant sur l’équateur.

L’erreur d’approximation sur l’icosaèdre est régulier, car maillage régulier.

Pour comparer l’efficacité des deux solutions, on va calculer pour chaque division de sphère (nombre de triangle fixe), un score qu’on trouvera en faisant la moyenne de la distances entre chaque centres de tous les triangles et le cercle circonscrit.

On calcule aussi l’écart min et max, pour pouvoir identifier la régularité de l’erreur.

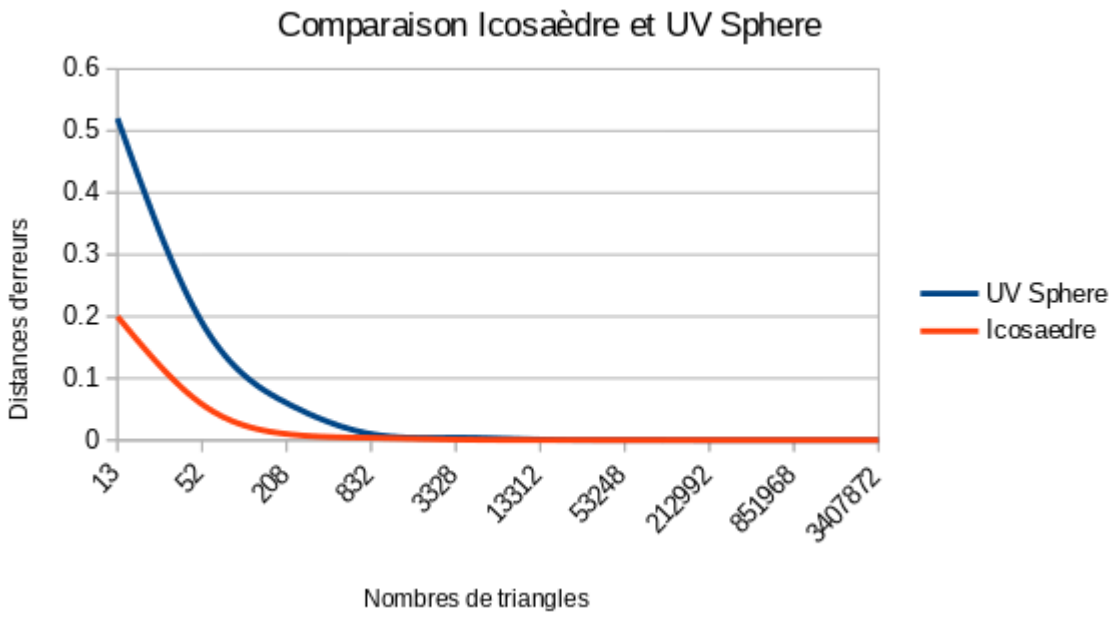




En comparant les deux solutions, on peut remarquer que l’UV Sphère admet une plus grande variance que son homologue lors de la division en 100 triangles.

C’est du au fait que les taux d’erreurs aux pôles sont inférieur qu’a l’équateur.

Dans l’Icosaèdre les erreurs sont les mêmes pour tous les triangles car maillage régulier.



L’icosaèdre admet des taux d’erreurs inférieur à l’UV Sphère jusqu’à 1000 triangles ou la les deux solutions proposent un maillage de sphère avec une erreurs proche de 0.

L’icosaèdre tend plus vite vers un taux d’erreurs proche de 0.

4. Chargement model mtl

Utilisation du code du tutoriel avec glfw3, avec qt j'avais des erreurs avec le gl core de qt.

Faites make, et executer le a.out pour tester.

Chargement des textures :

```
unsigned int diffuseMap = loadTexture("resources/textures/container2.png");
unsigned int specularMap =
    loadTexture("resources/textures/container2_specular.png");
```

Chargement des shader pour l'éléphant :

```
Shader elephantShader("elephant.vs", "elephant.fs");
```

Vertex shader :

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoords;

out vec2 TexCoords;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    TexCoords = aTexCoords;
    gl_Position = projection * view * model * vec4(aPos, 1.0);
}
```

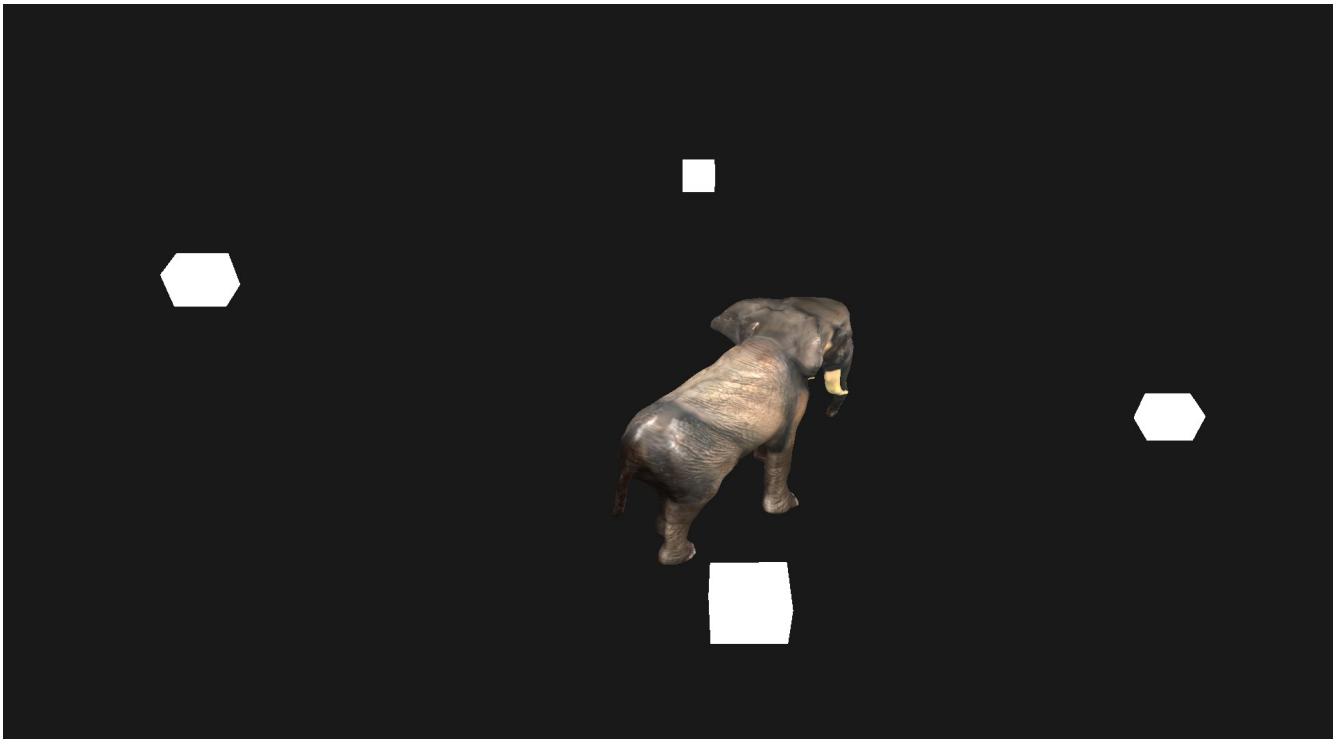
Fragment shader :

```
#version 330 core
out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D texture_diffuse1;

void main()
{
    FragColor = texture(texture_diffuse1, TexCoords);
    // FragColor =TexCoords;
}
```

Chargement du model :

```
Model elephantModel("resources/objects/elefante.obj");
```

Éclairage 4 points, non 3 points :



Sans éclairage particulier.

Avec éclairage ambiant, diffus et spéculaire.