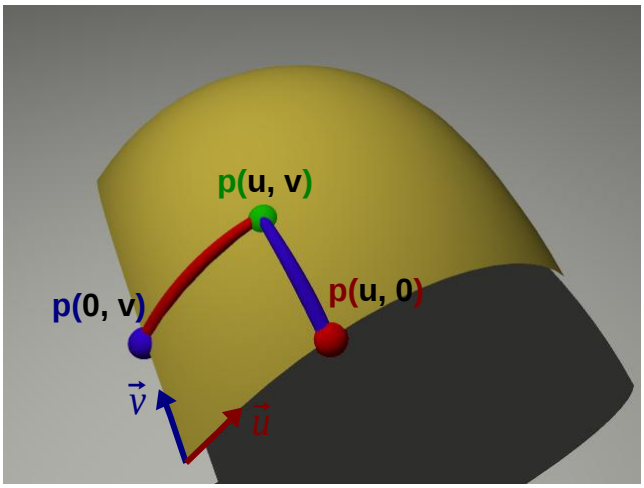
1 Scéance 3 : génération procédurale de maillage.

L'objectif de ce TP est d'implémenter, à partir d'une classe généraliste, la création et la visualisation de surfaces paramétriques telles que le plan, la sphère le cône et le cylindre.

Tout d'abord rappelons comment exprimer une surface paramétrique. On utilise une fonction $f:\mathbb{R}^2\to\mathbb{R}^3$ qui définit pour chaque point (u,v) de la surface une correspondance avec un point de l'espace (x,y,z) :



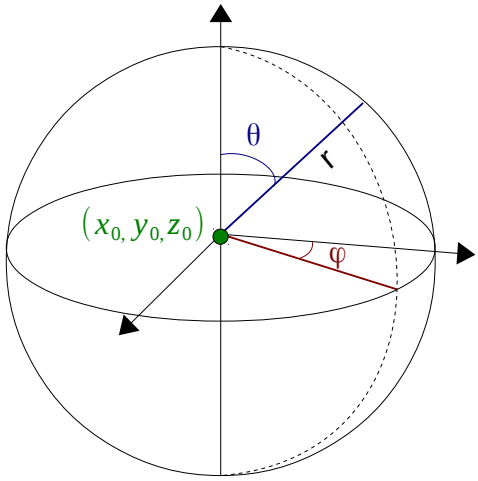
Comme on peut le voir un point de la surface peut être exprimé selon un repère 2D $(O,\vec{u},\vec{v})$ local à la surface. Trouver un point sur la surface mais exprimé selon un repère de l'espace $(O,\vec{x},\vec{y},\vec{z})$ se fait naturellement avec une équation paramétrique :

$$f(u,v)=\begin{cases}x(u,v)\\y(u,v)\\z(u,v)\end{cases}$$

Voici les équations dont vous aurez besoin :

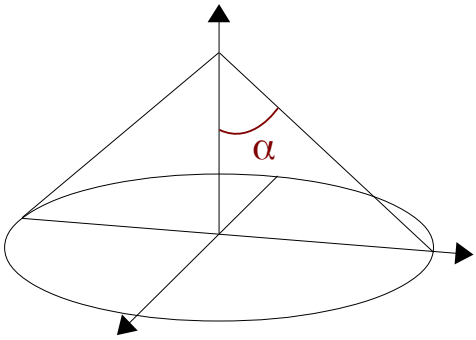
Sphère :

$$f(\varphi,\theta)=\begin{cases}x_0+r\cos\varphi\sin\theta\\y_0+r\sin\varphi\sin\theta\\z_0+r\cos\theta\end{cases}\qquad 0\le\varphi<2\pi,\ 0\le\theta<\pi$$



Cylindre :

$$f(\varphi,h)=\begin{cases}x_0+r\cos\varphi\\y_0+r\sin\varphi\\h\end{cases}\qquad 0\le\varphi<2\pi,\ h\in\mathbb{R}$$

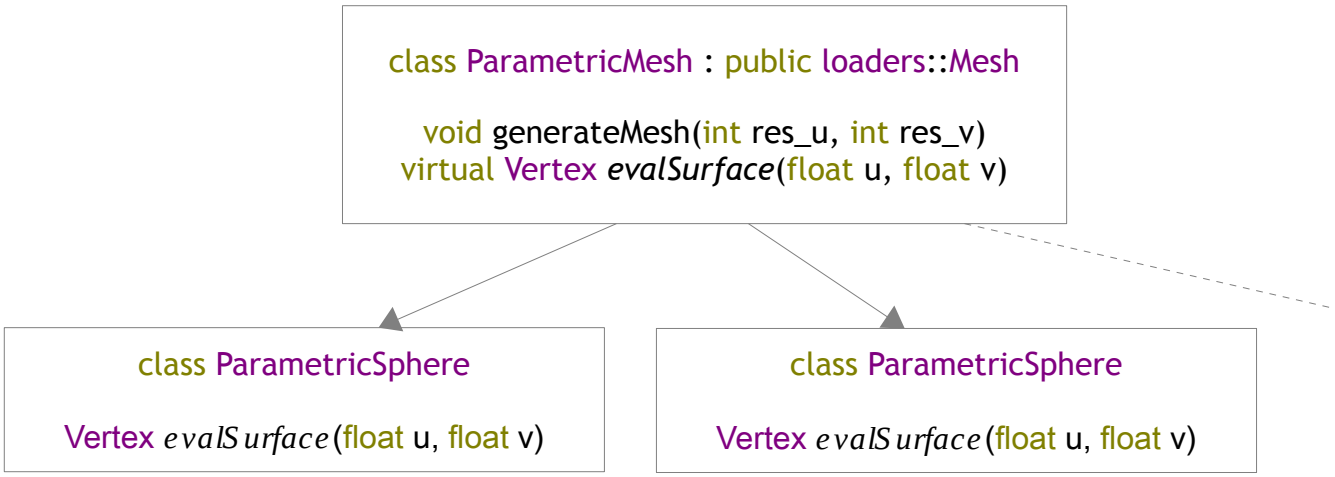


Cône :

$$f(\varphi,h)=\begin{cases}x_0+h\tan\alpha\cos\varphi\\y_0+h\tan\alpha\sin\varphi\\z_0+h\end{cases}\qquad 0\le\varphi<2\pi,\ h\in\mathbb{R}$$

1.1 Implémentation :

L'objectif est d'obtenir le code le plus compact et le plus réutilisable possible. Nous allons séparer la partie topologique de la surface (l'algorithme de 'maillage') de la partie géométrique (définition d'un point de la surface). Pour cela nous utiliserons des fonctions virtuelles pour obtenir du polymorphisme de classe. Une classe mère sera chargé d'implémenter l'algorithme de maillage de notre surface. Elle disposera d'une méthode d'évaluation du point de la surface qui pourra être spécialisé selon les différentes dérivations de la classe mère :



La fonction generateMesh() est chargé de mailler et d'évaluer un point de la surface grâce à la fonction virtuel *evalSurface*(). Cette méthode d'évaluation n'est rien d'autre que le calcul de la fonction d'une surface paramétrique. Une telle surface peut être exprimée par de simples équations comme celles données plus haut, ou bien sous des formes plus complexes comme le produit tensoriel de deux courbes de Bézier (e.g patch paramétriques).

Dans un premier temps implémentez la classe suivante dans renderer.cpp :

```
class ParametricMesh : public loaders::Mesh {
public:
    ParametricMesh() : Mesh() { }
    void generateMesh(int res_u, int res_v) { }

protected:
    virtual Vertex evalSurface(float u, float v) const { }
};
```

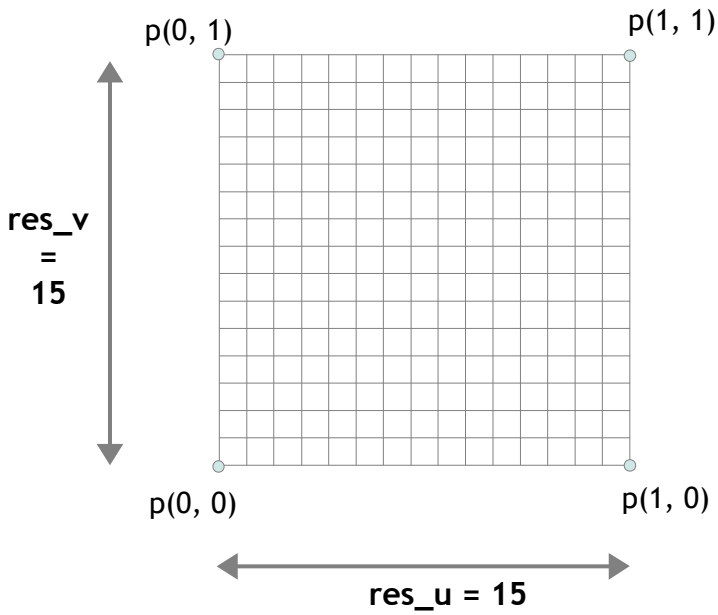
Le comportement par défaut de ParametricMesh est de générer un plan.

La fonction evalSurface(int u, int v) est donc chargée d'évaluer les points sur le plan d'équation *z=0* avec $u = x \in [0,1]$ et $v = y \in [0,1]$.

La méthode generateMesh() est chargé de remplir le vecteur d'index des triangles et le vecteur de sommets du maillage calculés par la méthode evalSurface(). Ainsi, chaque spécialisation de la classe n'aura à implémenter que l'algorithme de génération de la géométrie via la fonction evalSurface().

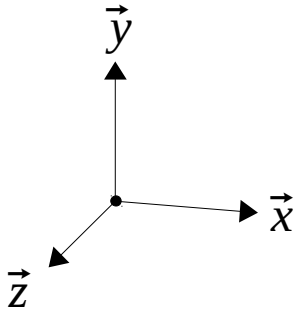
Les paramètres res_u et res_v de generateMesh() définissent le nombre de faces de la surface (si l'on considère qu'une face est un quadrilatère comme sur l'exemple ci-dessous).

- ✔ Avant d'aller plus loin testez la classe ParametricMesh en l'affichant en mode filaire (touche 'w') et plein (touch 'f').



Vous pouvez maintenant implémenter les classes filles pour la sphère, le cylindre et le cône.

1.2 Bonus : Dessine moi un repère orthonormé



Vous allez maintenant utiliser les primitives de bases que vous avez générées pour construire un objet plus complexe : un repère orthonormé. Au centre du repère une sphère avec chaque axe matérialisé par une flèche (en utilisant le cylindre et le cône).

Afin de placer les objets dans la scène vous devrez penser à modifier la matrice de vue 'g_viewMatrix'. Avant chaque dessin il vous faudra modifier cette matrice et penser à répercuter les changements sur les paramètres *uniform* de votre *shader*. Ainsi la transformation effectué dans le vertex shader sera différente à chaque changement de matrice.

Les fonctions glm suivantes pourons vous être utiles pour accomplir cette tâche :

- ✔ glm::rotate(...), glm::scale(...), glm::translate(...)

Nous vous conseillons fortement l'utilisation d'une fonction intermédiaire pour fixer les paramètres *uniform* avant chaque dessin. Une telle fonction pourrais prendre en paramètre la matrice de vue et de projection et configurer les paramètres *uniform* comme il se doit.