IN55 : Rapport de projet

Génération paramétrique de morilles.

Brogle Boris
DURAND Lilian
Poncet Yann

SOMMAIRE

Introduction

Dans le cadre de l'UV **IN55 - Synthèse d'image** enseignée à l'UTBM, il nous a été proposé de réaliser un projet autour de cette thématique, en utilisant les outils OpenGL et le langage C++. Nous avons choisi le premier sujet : la **construction d'objets paramétriques 3D**.

Plus précisément, nous avons décidé de réaliser un "générateur" de morilles. Les morilles sont des champignons avec une forme générale très particulière et dont les variations de forme et de couleur sont nombreuses selon les sous espèces.

Morilles noire, blondes, et grises à gauche. Un morillon à droite.

Nous avons choisi de travailler sur les morilles car cela allait nous permettre de toucher un peu à tout, mais aussi car il était très facile de trouver des paramètres ajustables, par exemple au niveau de la forme générale, de celle du chapeau, de la couleur etc…

La répartition du travail s'est faite naturellement : alors que Boris et Yann se sont principalement occupés de la partie génération, Lilian s'est occupé de tout ce qui était autour, dont les shaders.

I. Installation et utilisation

Dans le dossier rendu, il est possible de trouver un exécutable dans le dossier **Executable**, qui peut être lancé via run.bat. Il faut cependant attendre un certain moment avant de voir la fenêtre s'afficher.
Il est aussi possible de trouver toutes les sources dans le dossier **Sources**, à ouvrir avec Qt, et bien sûr le présent rapport.

Lorsque la fenêtre est ouverte, il est possible de s'y déplacer avec les touches **ZQSD** et la **molette de la souris**.

II. Structure de base du projet

Pour la partie OpenGL, nous sommes parti du template donné en TP. Rien de spécial à ajouter concernant cette partie.
Ensuite, pour la génération des morilles à proprement parler, nous avons créé une classe **Morel** : représentant l'ensemble du champignon et chargée principalement de transmettre les vertices à OpenGL sous la bonne forme, elle contient un pied, géré par la classe **Stem**, et un chapeau (**Cap**), qui sont elles chargées de générer les vertices et de procéder à toutes les transformations nécessaires pour obtenir la forme voulue. Les classes Stem et Cap héritent toutes deux d'une classe **MorelPart**, regroupant les fonctions communes aux deux classes.

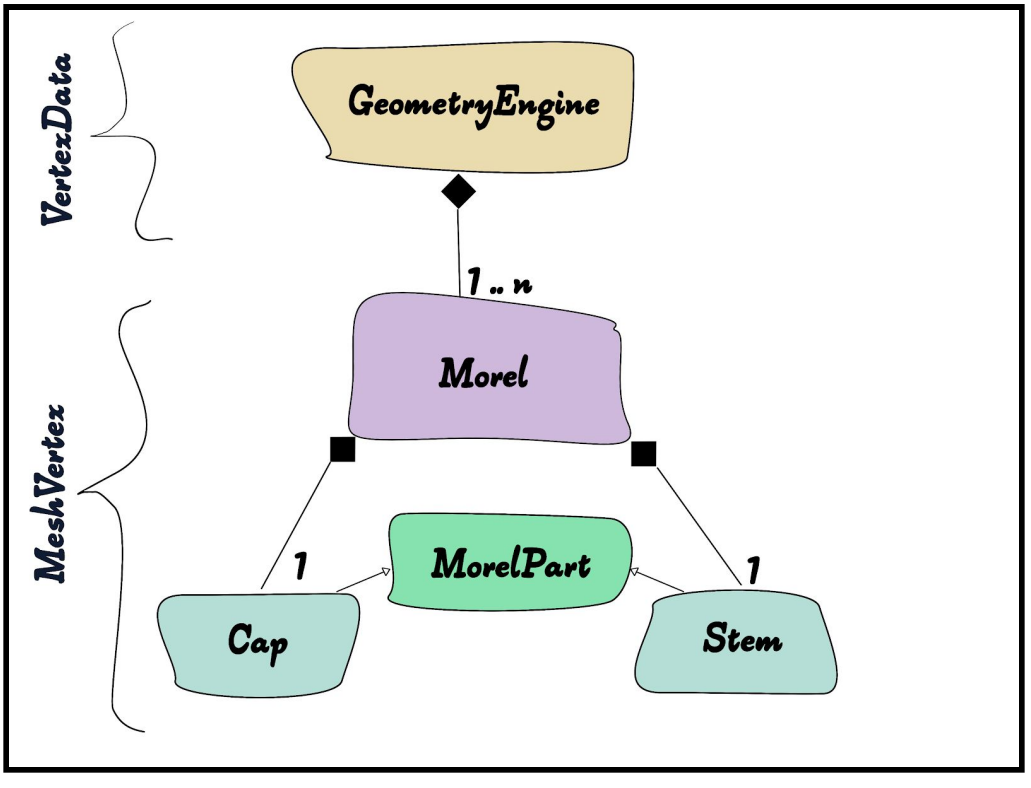
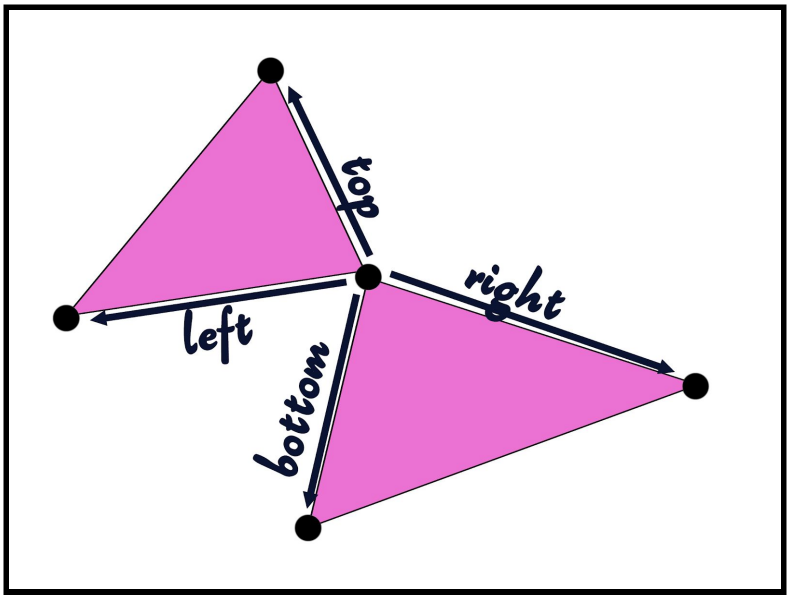


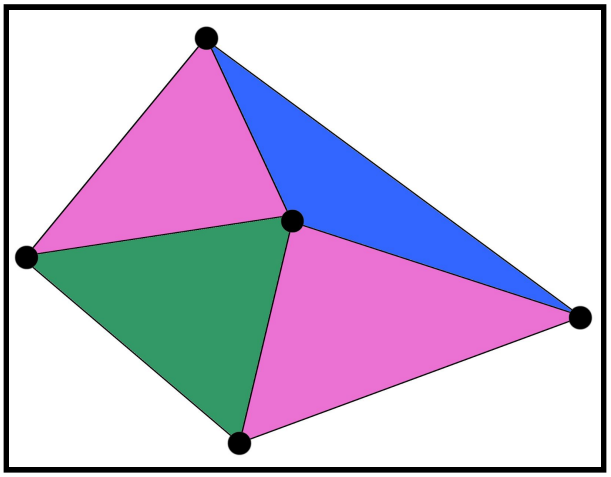
Schéma du "coeur" de notre générateur.

Comme on peut le voir sur le schéma, la structure de donnée contenant les vertices au niveau de la génération (Morel + Cap + Stem) est une liste de **MeshVertex**. Un MeshVertex est une classe contenant un pointeur vers ses voisins, ainsi, pour chaque vertex généré, on peut accéder à ses voisins de droite, gauche, bas et haut.
Cela nous permet notamment de calculer les vecteurs normaux à nos vertices au bon moment et de facilement générer notre tableau d'indices à partir des voisins.
On peut voir dans l'image ci dessous les triangles générés par le tableau d'indices :



Un MeshVertex au centre, et les triangles formés par le tableau d'indices par la suite.

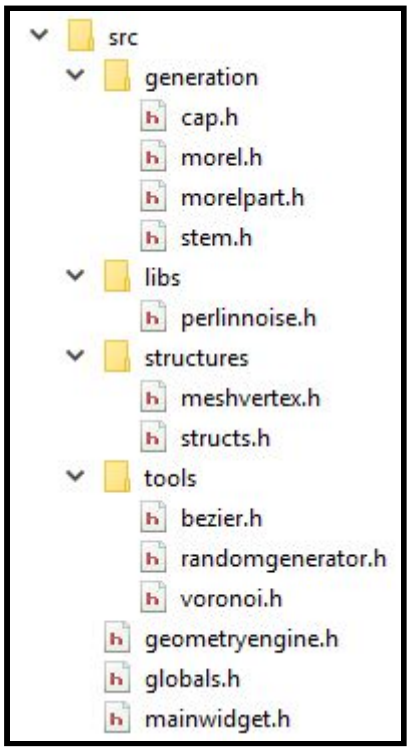
Puisque chaque vertex sera entouré de 4 triangles qu'il forme avec ses 4 voisins, pour calculer les vecteurs normaux associés à chaque vertex, on calcule un vecteur normal à chaque triangle grâce au produit vectoriel des deux vecteurs issus du vertex et formant le triangle.



Le calcul des vecteurs normaux se fait en fonction des surfaces adjacentes

Ainsi, chaque vecteur résultant aura une norme proportionnelle à l'aire de son triangle, et il suffira d'additionner ces vecteurs pour obtenir la normale à notre vertex.

La structure MeshVertex contient également des méthodes de transformations, et des attributs utilisés pendant la génération qui n'ont pas besoin d'être transmis à notre moteur de rendu.
Ainsi, lorsque la génération est terminée, on transmet nos vertices au GeometryEngine sous la forme d'un tableau de **VertexData**, avec toutes les données nécessaires au rendu (positions, couleurs, normaux, textures).



Nos sources ont été séparées de la façon dont le montre l'image ci contre : les quatre classes permettant la génération ont été placées dans le dossier **generation**, les librairies externes utilisés dans le dossier **lib**, les outils que nous avons développé dans le dossier **tools**, et les structures de données dans le dossier **structures**.

Nous avons également un fichier regroupant les variables globales (**globals.h** et **globals.cpp**) à la racine, parmi lesquelles figurent entre autres les paramètres ajustables pour changer la génération.

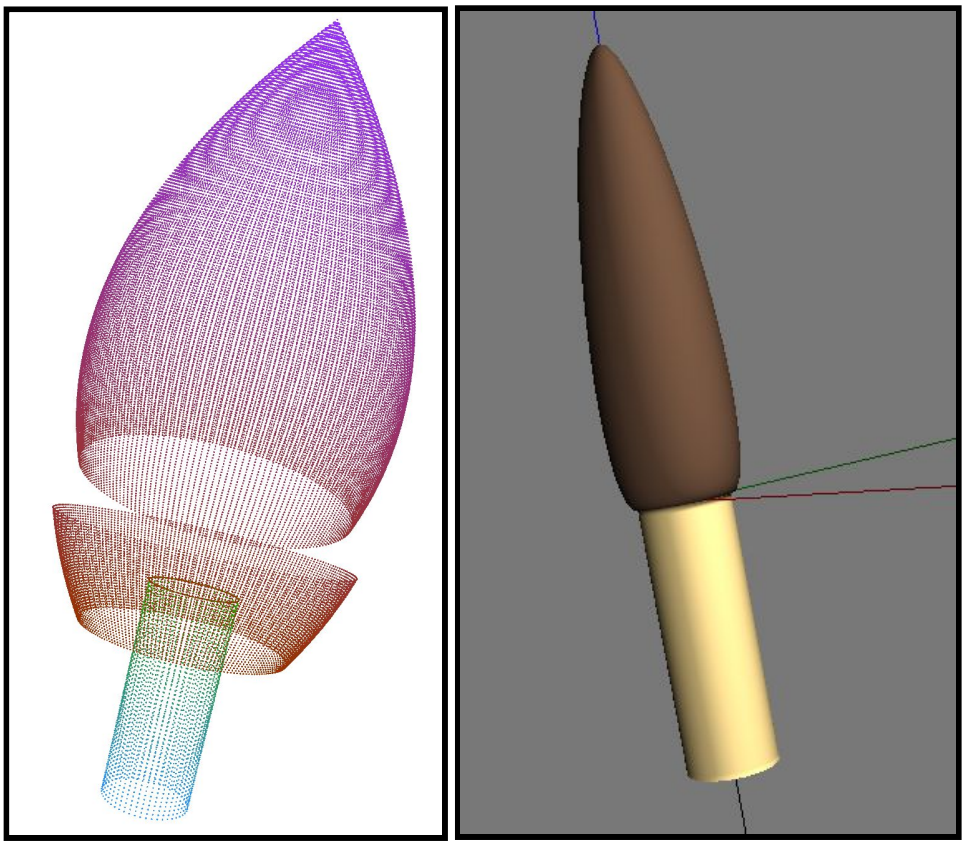
III. Les différents outils utilisés pour la génération

Pour générer nos morilles, nous sommes parti d'une forme globale simple que nous avons affiné à l'aide de plusieurs outils.
Ainsi, dans cette partie, nous allons détailler étape par étape la façon dont la génération se fait.

1) Maillage et forme de base

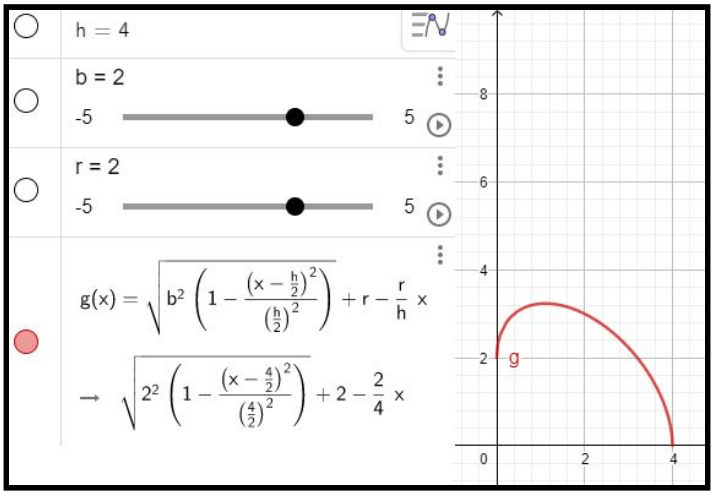
La forme de base est générée de la même façon pour le pied et le chapeau. Pour cela, on divise la hauteur voulue par un nombre n et pour chaque "couche", on place k points sur un cercle.

On garde un rayon constant pour le pied, ce qui nous donne un cylindre, alors que pour le chapeau, on utilise une ellipsoïde qui modifie le rayon en fonction de la hauteur.



*A gauche, notre maillage initial avec **40 000** points pour le chapeau, **2 500** pour le pied*
A droite, la forme que l'on a actuellement au début de la génération

Comme abordé plus haut, la fonction utilisée pour générer le chapeau correspond à une demi ellipse, à laquelle on ajoute la prise en compte du rayon, pour qu'elle ne démarre pas à 0.

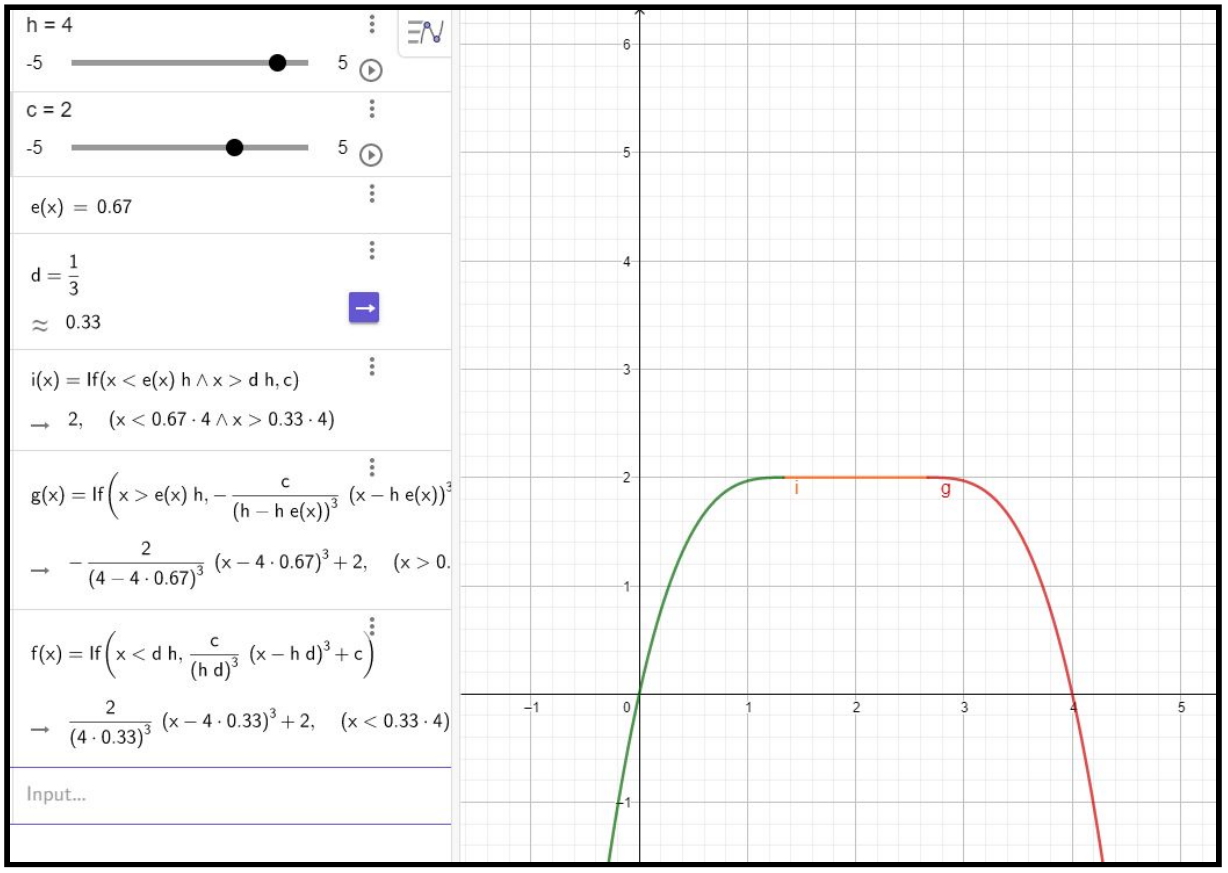


Fonction utilisée pour la forme de base du chapeau, avec rayon=2 et hauteurMax=4

Initialement, le chapeau est généré au dessus de z=0, et le pied en dessous, ce qui permet de plus facilement laisser invariant le cercle situé en z=0, jonction entre le chapeau et le pied.

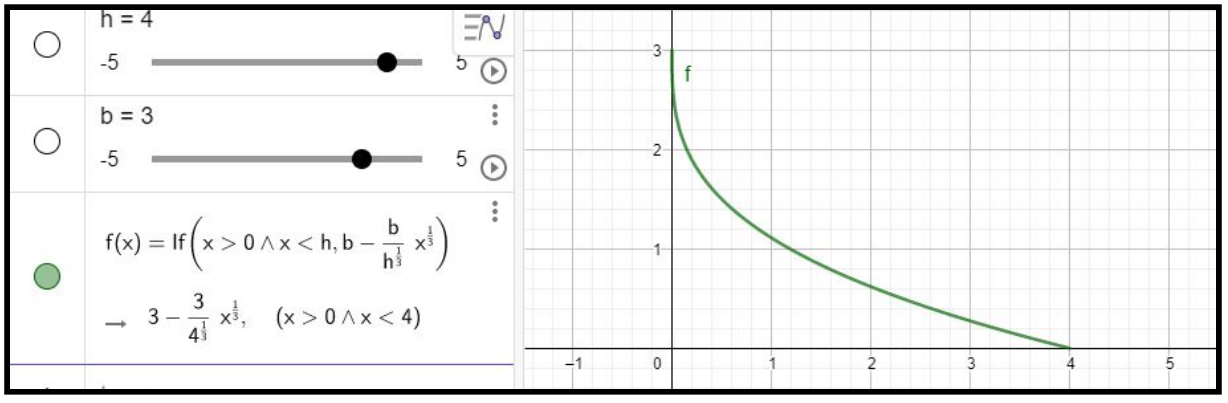
2) AFFINAGE DE LA FORME GLOBALE

L'affinage de la forme se fait via une fonction mathématique, aussi bien pour le pied que pour le chapeau.
Voici la fonction utilisée pour le chapeau :



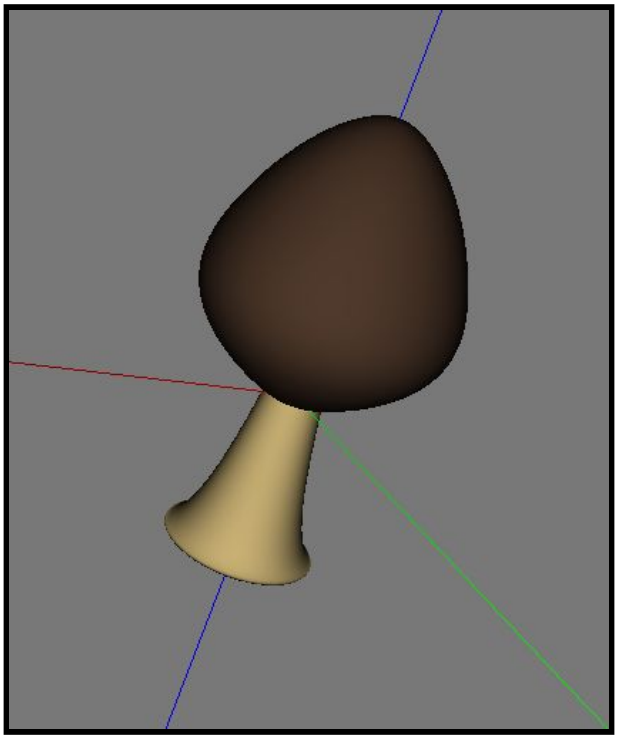
Fonction utilisée pour affiner la forme du chapeau, avec hauteurMax=4

Celle utilisée pour le pied est une fonction racine cubique :



Fonction utilisée pour affiner la forme du pied, avec hauteurMax=4

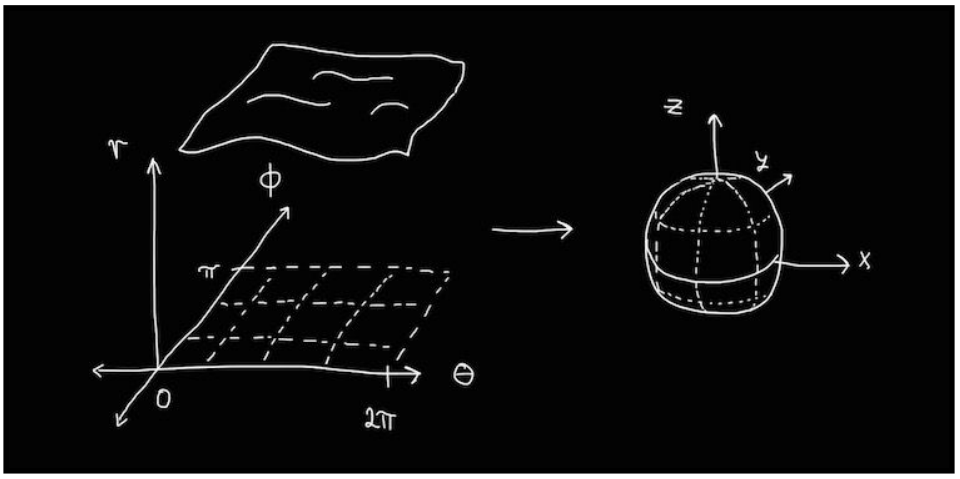
Ainsi, on obtient un résultat semblable à celui ci après l'application de ces deux fonctions :

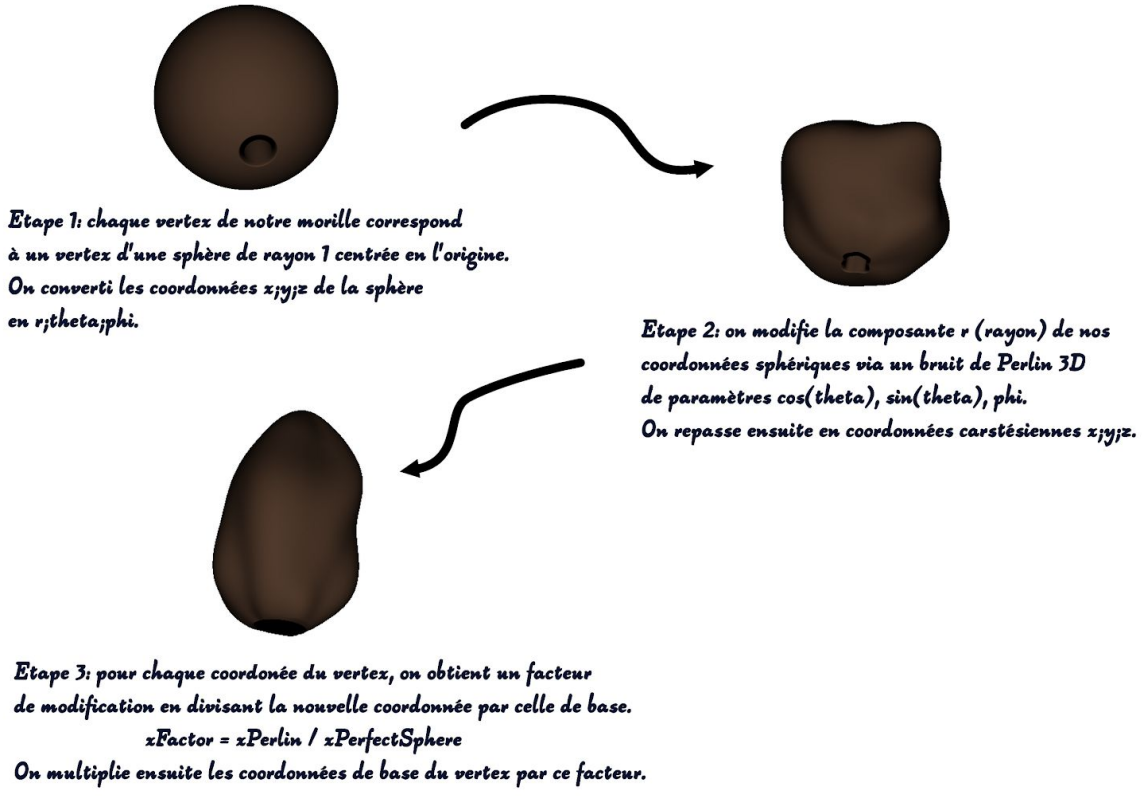


Notre morille avec la forme affinée

3) Déformation par bruit de Perlin et coordonnées sphériques

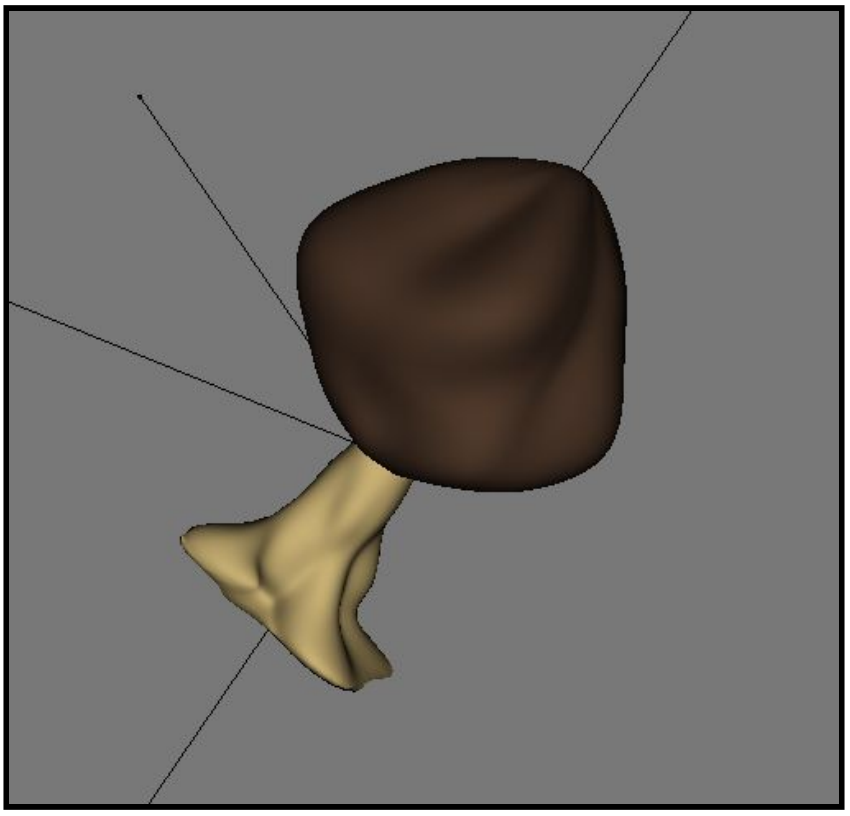
La prochaine étape consiste en l'application d'un bruit de Perlin aussi bien sur le chapeau que sur le pied, pour rendre la forme globale un peu moins régulière et donc plus réaliste.
Pour cela, on utilise les coordonnées sphériques. Bien que le chapeau ne soit pas tout à fait une sphère, cela fonctionne tout de même relativement bien, cela permettant même d'avoir des variations plus faibles en haut, ce qui est plus naturel. Pour le pied, le fait d'utiliser des coordonnées sphériques et non cylindriques offre également un avantage, en créant des sortes de "racines" à la base, très proches de la réalité.
Afin d'obtenir nos variations, on "normalise" d'abord nos coordonnées, on se ramène ainsi à une sphère de rayon 1 centrée en l'origine, permettant d'avoir un contrôle de nos valeurs. Nos coordonnées cartésiennes x;y;z deviennent alors r;theta;phi, et on fait varier le rayon r par bruit de Perlin comme le montre le schéma ci dessous :





Procédé utilisé pour appliquer notre Perlin

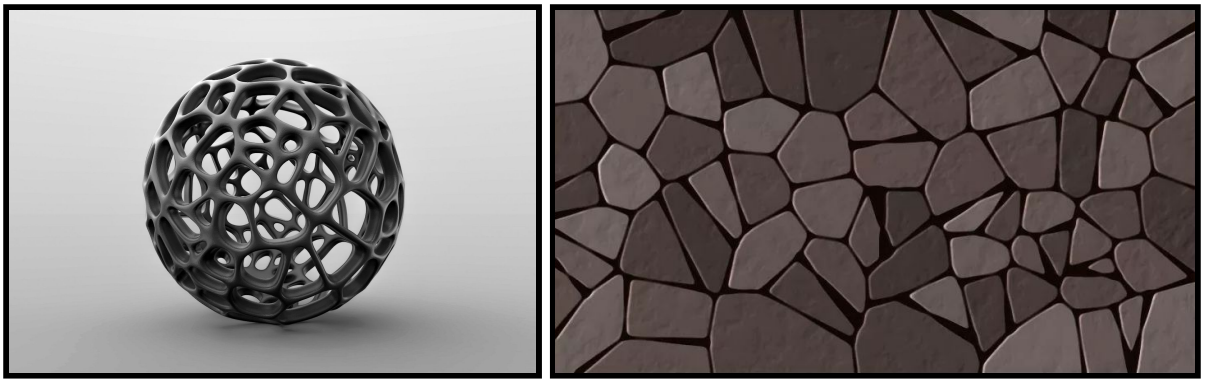
Il suffit ensuite de convertir à nouveau en coordonnées cartésiennes et d'appliquer nos modifications à nos coordonnées réelles en faisant le rapport entre les anciennes coordonnées et les nouvelles, et en multipliant par ce rapport. On obtient ainsi un relief sur notre morille :



Résultat obtenu en amplifiant la puissance du Perlin pour bien le montrer

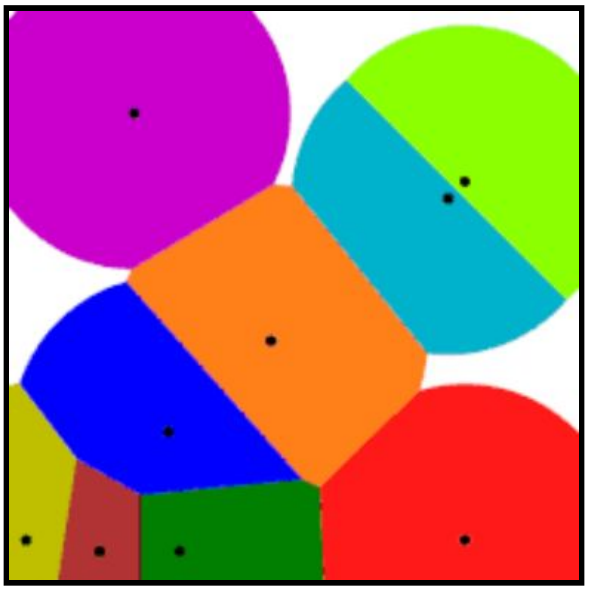
4) Voronoï tessellation pour les alvéoles

La **Voronoï tessellation**, aussi appelée **Voronoï diagram**, est une forme de tessellation permettant de générer des polygones d'une façon assez proche de celle que l'on peut trouver sur certaines morilles.
Les applications sont nombreuses, principalement en science et technologie. On peut notamment les retrouver pour simuler des cellules vivantes.



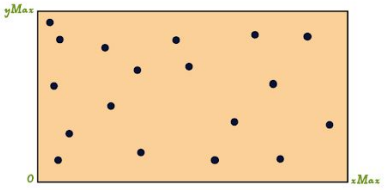
Deux applications de la Voronoï tessellation

Il est possible de trouver une animation en visitant ce lien.



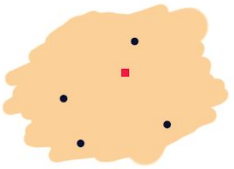
Construction d'une tessellation (extrait de l'animation)

De la même façon que pour le Perlin, notre tessellation est construite à la base sur une sphère de rayon 1 centrée en l'origine.
Pour chaque Vertex de cette sphère, on fait appel à la classe Voronoï, en pour nous donner un facteur entre fMin et fMax, correspondant aux variations de hauteur, on obtient ainsi fMax lorsque l'on est sur une ligne entre deux couleurs dans l'exemple ci dessus, et fMin lorsque l'on est sur un point. Une fonction inverse est utilisée pour faire décroitre rapidement le facteur jusqu'à son minimum.
Comme un schéma vaut mieux qu'un long discours, voici son fonctionnement en bref :

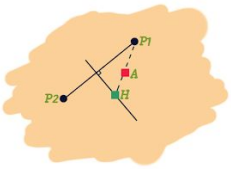


Voronoi en bref

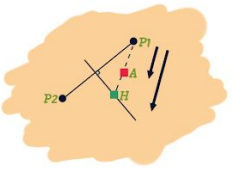
Etape 1: on place aléatoirement des points entre 0;0 et xMax;yMax.
On fait attention à ne pas trop les rapprocher.



Etape 2: pour tout point dont on veut connaître son facteur, on reçoit ses coordonnées x et y entre 0 et xMax;yMax et on trouve les deux points les plus proches.
Attention, un point proche de xMax peut très bien avoir un point proche de zero comme point le plus proche, c'est un cylindre.

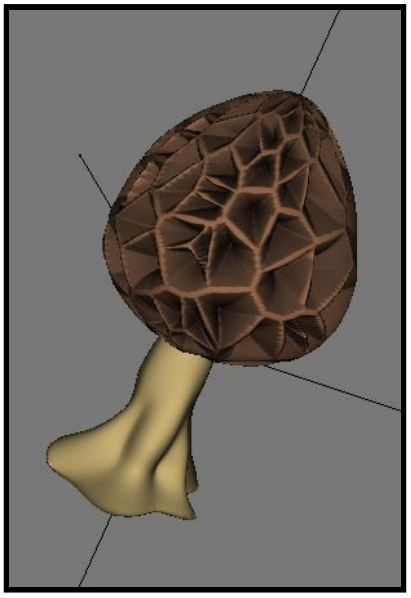


Etape 3: on "trace" la ligne entre P1 (point le plus proche), et notre point, A, et on trouve son intersection avec la ligne perpendiculaire à P1P2 passant par son milieu.
On note ce point H.



Etape 4: on calcule le rapport P1A / P1H. On passe ce résultat dans une fonction inverse pour accentuer la décroissance: on en ressort notre facteur.

Voici le résultat obtenu :

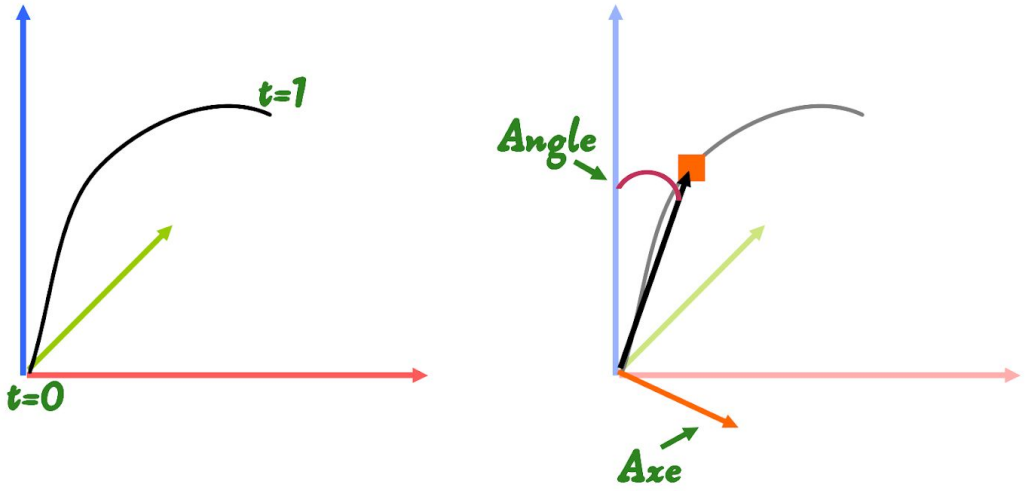


Notre morille après tessellation

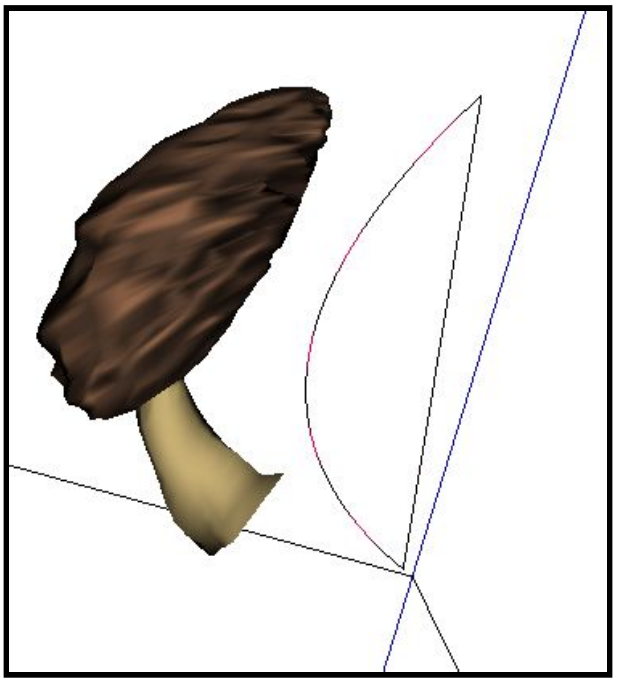
5) Bézier pour la courbure

Afin de générer des courbures dans nos morilles, nous avons décidé d'intégrer une courbe de Bézier à trois points.

Pour cela, en tout point de notre morille, on calcule le point de la courbe de Bézier correspondant en divisant par la hauteur totale, cela nous donne donc t variant de 0 à 1.

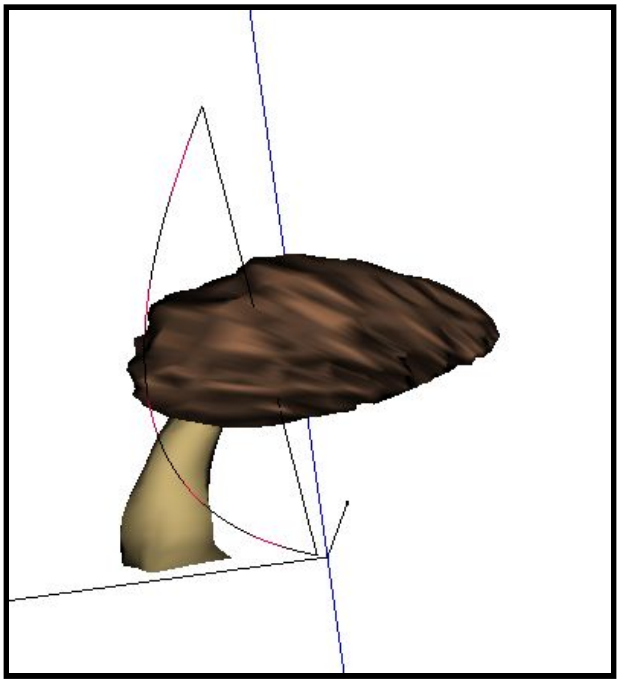


On calcule ensuite l'angle entre le vecteur origine-point et l'axe z, c'est notre angle de rotation. L'axe de rotation est donné par le produit vectoriel de l'axe z et du même vecteur. On en extrait ainsi un quaternion de rotation.
A partir de là, on obtient le résultat suivant (la courbe de Bézier est aussi affichée) :



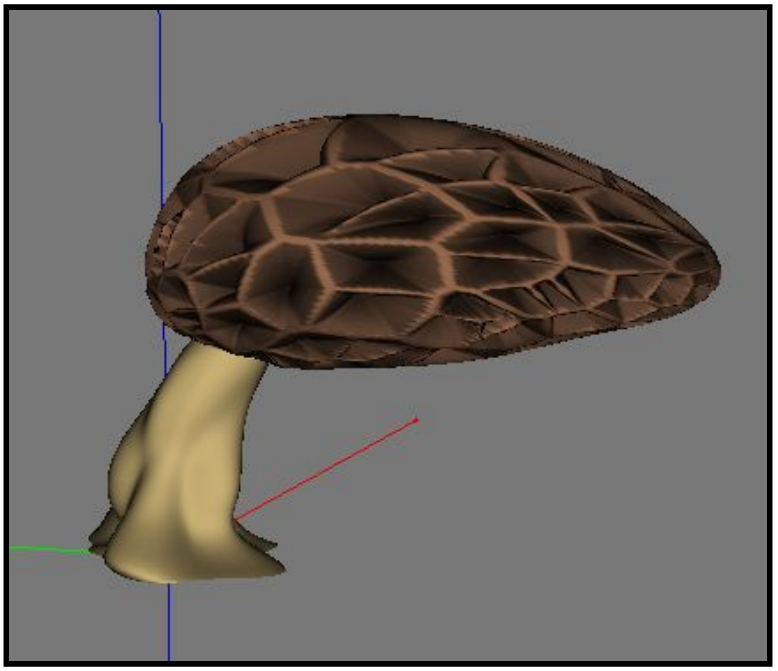
La courbe de Bézier appliquée à une morille simplifiée et translatée, sans redressement

La morille suit bien la courbe, mais son pied n'est pas perpendiculaire à l'axe des z.
Il faut donc la redresser en soustrayant l'angle entre la base et l'axe z :



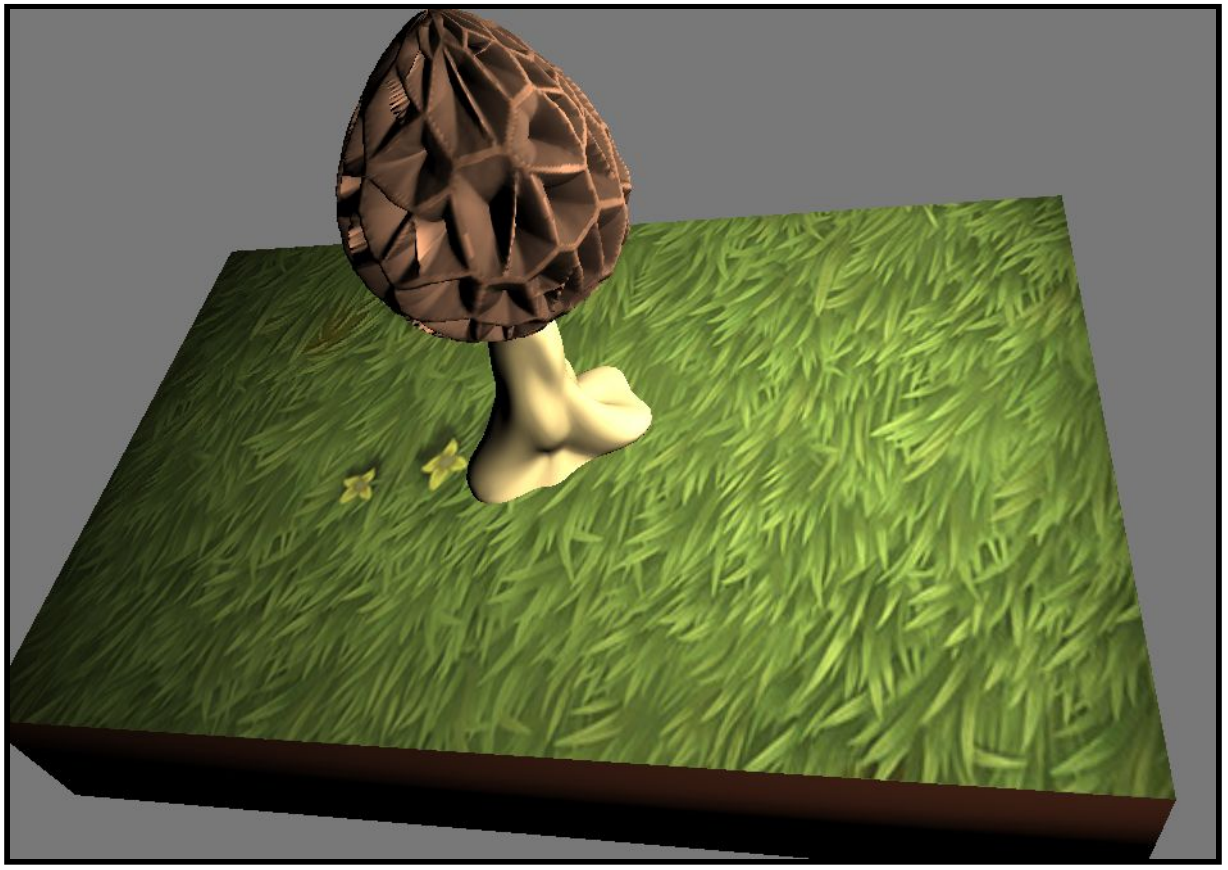
La même morille, mais cette fois redressée

Après bézier, quelques fonctions supplémentaires permettent d'affiner encore davantage la morille (petites variations de couleur, Perlin pour ajouter un peu de relief de près…), sans différence flagrante toutefois.
On se retrouve ainsi avec le résultat final semblable à celui ci (la plupart des morilles générées avec les paramètres actuels sont cependant assez droites) :



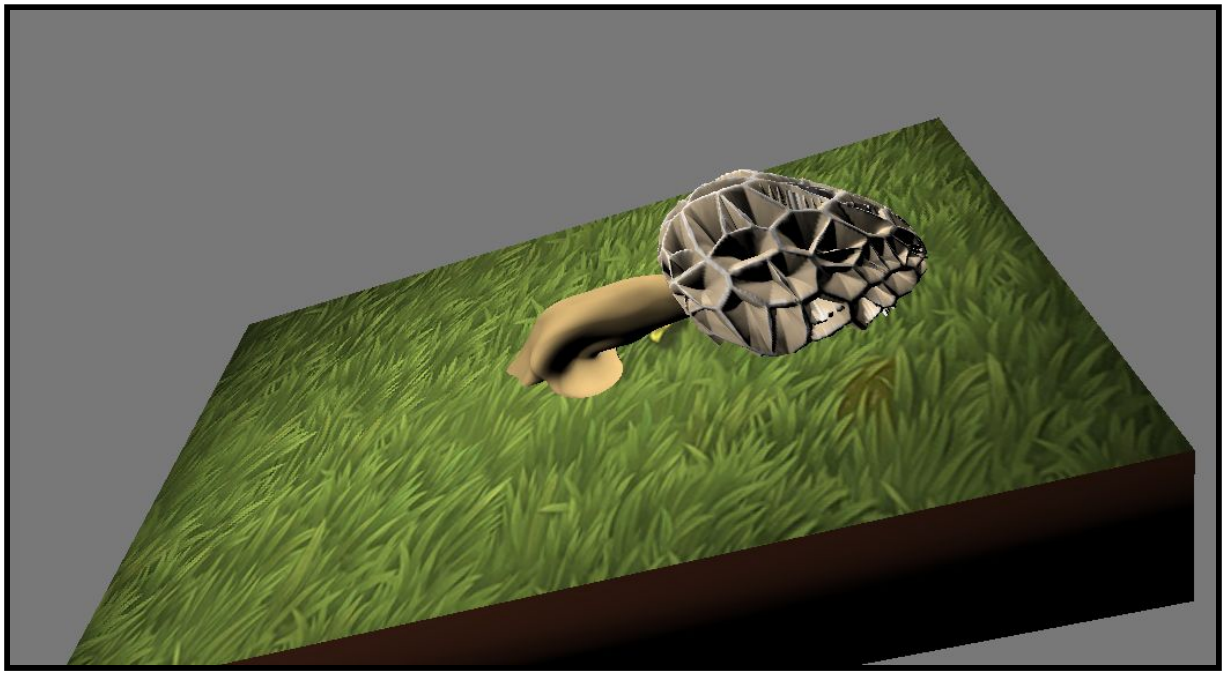
La même morille avec toutes les transformations

Voici la même morille avec les paramètres de base et un sol sous son pied :



Notre version quasi-finale

Bien entendu, cela n'empêche pas de tomber sur des morilles courbées "à l'extrême", chose que nous n'avons pas décidé d'empêcher, car nous trouvons cela assez amusant à voir :



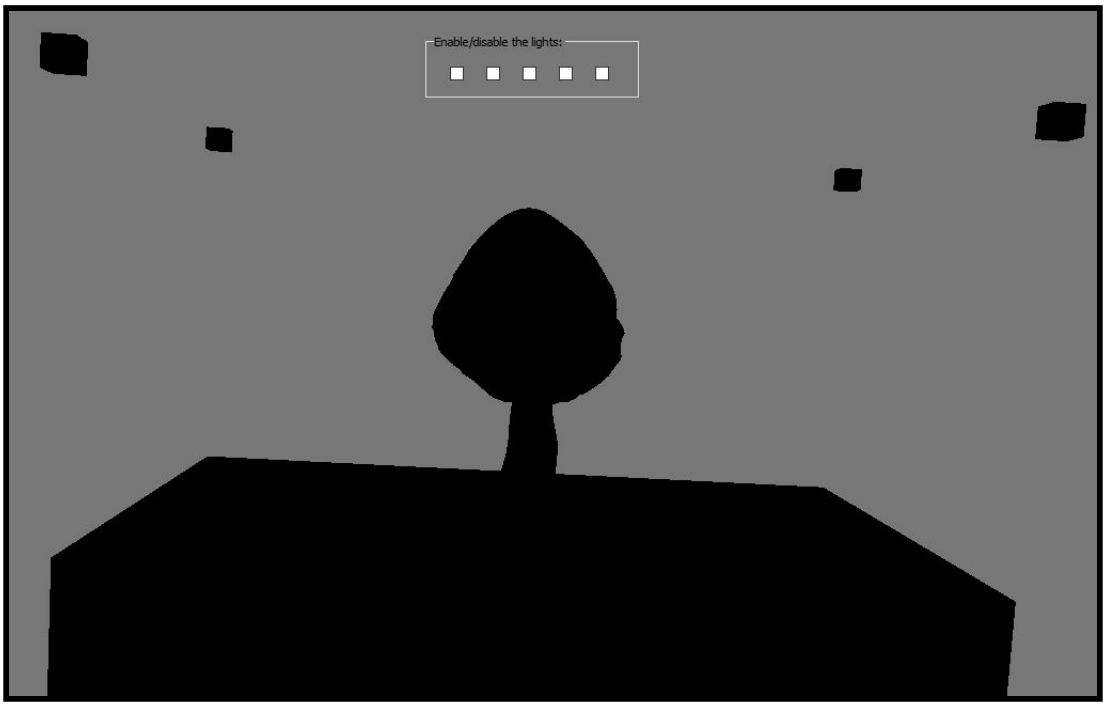
Quelquefois, on obtient de drôles de morilles

IV. Modèle d'illumination

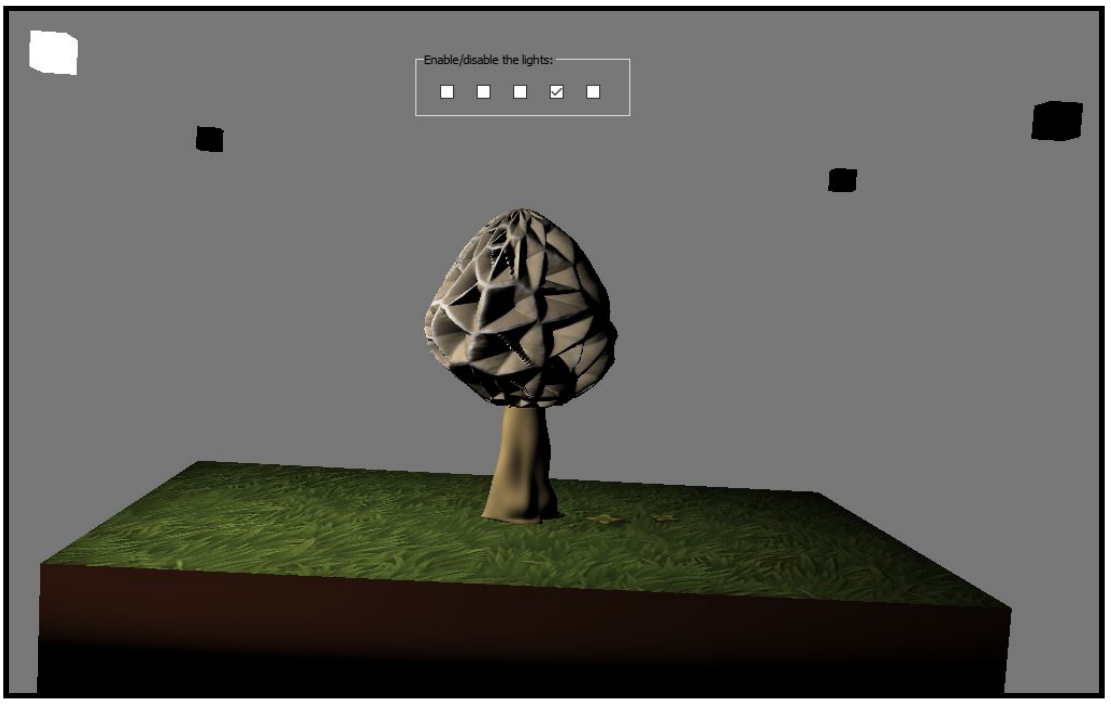
Afin de faire ressortir les détails de notre morille et plus particulièrement les **cavités** de son chapeau nous avons ajouté **différentes lumières** en utilisant les **shaders**.

1) Architecture globale

Nous avons décidé d'utiliser **cinq sources de lumières** blanches localisées, les quatres premières sont situées en hauteur au niveau des angles de la plateforme et la dernière se trouve sur la caméra. Chaque source peut être **activée** ou **désactivée** à l'aide de l'interface graphique (cf. VI. Interface utilisateur). Par défaut la lumière de la caméra est activée, toutes les autres sont désactivées.



Aucune lumière activée



Seule une lumière d'angle est activée

Bien que plus coûteux, le modèle d'ombrage utilisé est celui de **Phong**. Le programme n'utilise actuellement que des surfaces Lambertiennes, cependant, pour des améliorations futures, comme l'ajout de gouttes d'eaux ou de "bave" de limace sur les morilles, il nous a semblé préférable que le modèle de Phong soit déjà implémenté.

2) Matérialisation des sources lumineuses

Afin de matérialiser les sources de lumières, à l'exception de la lumière caméra, le programme draw un **cube** ayant comme centre la **position de la lumière**.

La **couleur de chaque cube** dépendra de l'**état de sa source de lumière** associée :
- la lumière est **activée**, la couleur du cube est **blanche**
- la lumière est **désactivée**, la couleur du cube est **noire**

Cela permet aux cubes dont la lumière est désactivée de ne pas être éclairés par les autres lumières.

Les "normals" de chaque vertex du cube sont calculés simplement avec le vecteur allant du vertex au centre du cube (position de la lumière), elles sont donc dirigées vers l'**intérieur du cube**.
Le but est de fournir à l'utilisateur un **support visuel** permettant de savoir avec précision d'où proviennent les sources de lumières ainsi que leur état d'où l'utilisation de ces méthodes.

V. Les différents paramètres

La plupart des paramètres qu'il est utile de changer pour ajuster la génération se trouvent dans une structure globale située dans le fichier **globals.cpp**.

Nous avons également fait en sorte que certains paramètres suivent une loi normale pour y ajouter des variations aléatoires, ainsi, on peut en changer la variance. Par exemple pour la courbure de la morille, la plupart du temps, la morille sera droite, mais grâce à une loi normale et en fonction de la variance qu'on lui donne, on se retrouve avec des morilles plus tordues de temps en temps.

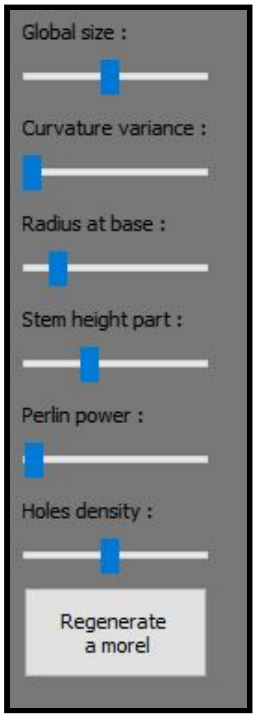
Voici les paramètres les plus intéressants sur lesquels il est possible de jouer :
- la **taille** globale, suivant une loi normale
- la **hauteur** de la morille
- la **part du pied** dans la hauteur totale
- les différents **rayons** : rayon à la jonction entre pied et chapeau, rayon max du chapeau, rayon à la base du pied, ..., suivant des lois normales pour certains
- le **meshing**, c'est à dire le nombre de points horizontaux et verticaux
- la **courbure** et la position de l'angle de courbure, suivants une loi normale
- les **couleurs** possibles pour la morille, au nombre de 4 actuellement (blonde, brune, noire et grise)
- la **puissance des perlins**
- le **nombre d'alvéoles** du chapeau, la **largeur des arêtes**

- et d'autres…

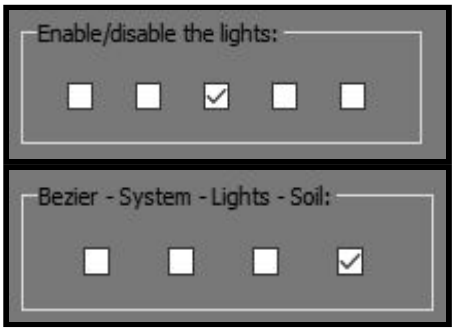
Pour les paramètres suivant une loi normale, on peut paramétrer l'espérance (médiane) et la variance.
Bien sûr, d'autres paramètres qui ne sont pas enclin à trop changer sont présents dans les fonctions même.

VI. INTERFACE UTILISATEUR

Vu le nombre de paramètres, nous en avons sélectionnés quelques uns à ajouter sur l'interface utilisateur.



On peut ainsi modifier 6 paramètres qui nous semblaient intéressants d'avoir sur l'UI, et ce via des sliders.
Un bouton permet de régénérer la morille après avoir changé -ou pas- les paramètres.



Il est aussi possible d'illuminer ou non la scène avec des lumières placées aux quatre coins de la scène, et une située sur la caméra. Également, on peut activer ou non différents éléments : la courbe de bézier générée, le système de coordonnées (les axes), les lumières et le terrain sous la morille.

Bien sûr, on peut zoomer avec la **molette** de la souris et se déplacer avec **ZQSD**.

Améliorations à apporter et conclusion

Voici quelques améliorations que nous aurions aimé ajouter au projet si nous avions eu plus de temps à y consacrer :

- générer **plusieurs morilles** sur la même scène, pas aussi simple qu'il n'y paraît car il faut prendre en compte les orientations et tailles de chacunes pour s'assurer qu'il n'y ait pas de "collisions" entre elles (sans avoir à parser tous les vertex pour cela)
- bien **texturer** notre champignon pour le rendre plus réaliste
- shader **d'ombre**
- ajouter des dégradations : principalement des **trous de limaces** par exemple

En bref, ce projet nous a non seulement permis de mieux intégrer les différents concepts vus en cours (quaternions, shaders, …) et de mieux comprendre comment fonctionne OpenGL, mais il nous a aussi fait travailler avec différents outils que nous avons vus pour certains lors de nos précédents semestres : coordonnées sphériques, Perlin, Bézier, Voronoï, géométrie dans l'espace…
Nous avons en tous cas trouvé le projet très intéressant à réaliser.