# Freestyle : Sculpting Meshes with Self-Adaptive Topology

## Rapport Recette

Étudiants : Charles Garibal, Maxime Robinot, Mathieu Dachy

Tuteur : Loïc Barthe

20/02/2015

I) Introduction
    Rappel : Objectif du chef d'oeuvre
    Spécifications du projet
II) Couverture des spécifications
    Création d'objets de base
    Importation et Exportation d'objets 3D
    Conversion en maillage quasi-uniforme
    Maintien du maillage quasi-uniforme
    Gestion des changements de topologie
    Modification des paramètres du maillage quasi-uniforme par l'utilisateur
    Implémentation de 3 opérateurs de déformations: Etirer, Bomber/Creuser, Tordre
    Framework indépendant
III) Améliorations possibles
    Exportation
    Optimisation
IV) Description de l'architecture du logiciel
    Conversion en maillage quasi-uniforme
    Maintien du maillage quasi-uniforme
    Gestion des changements de topologie
    Etirer, Bomber/Creuser, Tordre
V) Planning final
VI) Conclusion

# I) Introduction

## A. Rappel : Objectif du chef d'oeuvre

Le but de notre chef-d'oeuvre était de comprendre et d'implémenter au sein d'un moteur graphique des algorithmes issus d'un article de recherche. Cette publication propose une nouvelle approche de modélisation 3D à travers la manipulation d'outils intuitifs permettant à l'utilisateur d'appliquer des déformations sur un objet. Ainsi, même si ce genre d'application est reservée aux infographistes, un néophyte pourrait modifier l'apparence d'un objet à sa guise.

D'un point de vue technique, l'utilisateur manipule un maillage particulier dit "quasi-uniforme" et peut modifier sa topologie en formant des trous ou en fusionnant des régions. La complexité d'un tel module réside dans le contrôle et la modification de la structure du maillage, la détection et le traitement des changements de topologie, l'implémentation des outils de déformations ainsi que la conception de l'interface qui permettra de les utiliser.

## B. Spécifications du projet

Au moment de la spécification de notre projet, le cahier des charges et les fonctionnalités que nous avions fixées étaient les suivantes:

- ➢ Création d'objets de base pour une nouvelle sculpture (sphère, cube, tore)
- ➢ Importation et Exportation d'objets 3D
- ➢ Conversion en maillage quasi-uniforme
- ➢ Maintien du maillage quasi-uniforme
- ➢ Gestion des changements de topologie
- ➢ Modification des paramètres du maillage quasi-uniforme par l'utilisateur
- ➢ Implémentation de trois opérateurs de déformations: Etirer, Bomber/Creuser, Tordre
- ➢ Framework indépendant

Le logiciel gère les éventuels changements de topologie de la forme de base qui pourraient être le fruit des différents opérateurs (apparition d'un trou, chevauchement de surfaces, etc…).

# II) Couverture des spécifications

## Création d'objets de base

Au départ nous pensions donner à l'utilisateur le choix de plusieurs formes de base pour qu'il puisse débuter une nouvelle sculpture (sphère, cube et tore). Cependant, après réfléxion, nous n'avons retenu que la sphère. En effet, un cube n'est pas une forme géométrique dont la surface est continue et le tore peut être facilement obtenu après l'application de l'opérateur de déflation.

## Importation et Exportation d'objets 3D

Nous avions prévus de permettre à l'utilisateur de pouvoir charger un objet depuis un fichier 3D ainsi que de sauvegarder la sculpture courante dans un fichier. Le module d'importation utilisant la librairie Assimp était déjà implémenté dans le moteur, il permet d'ouvrir un grand nombre de type de fichiers. En voici certains:

- **Collada** ( *.dae* )
- **Blender 3D** ( *.blend* )
- **3ds Max 3DS** ( *.3ds* )
- **3ds Max ASE** ( *.ase* )
- **Wavefront Object** ( *.obj* )
- **Stanford Polygon Library** ( *.ply* )
- **Stereolithography** ( *.stl* )
- **Object File Format** ( *.off* )

Cependant, la grande variété de type en entrée n'est pas un avantage car le module d'exportation que nous avons développés ne sauvegarde que les informations géométriques du maillage sans prendre en compte les matériaux, textures, lumières, etc… Pour un logiciel de modélisation 3D, le maillage suffit amplement, mais si l'utilisateur utilise un fichier 3D avec plus d'informations, elles seront ignorées à la sauvegarde. De plus Assimp ne propose que quelques formats:

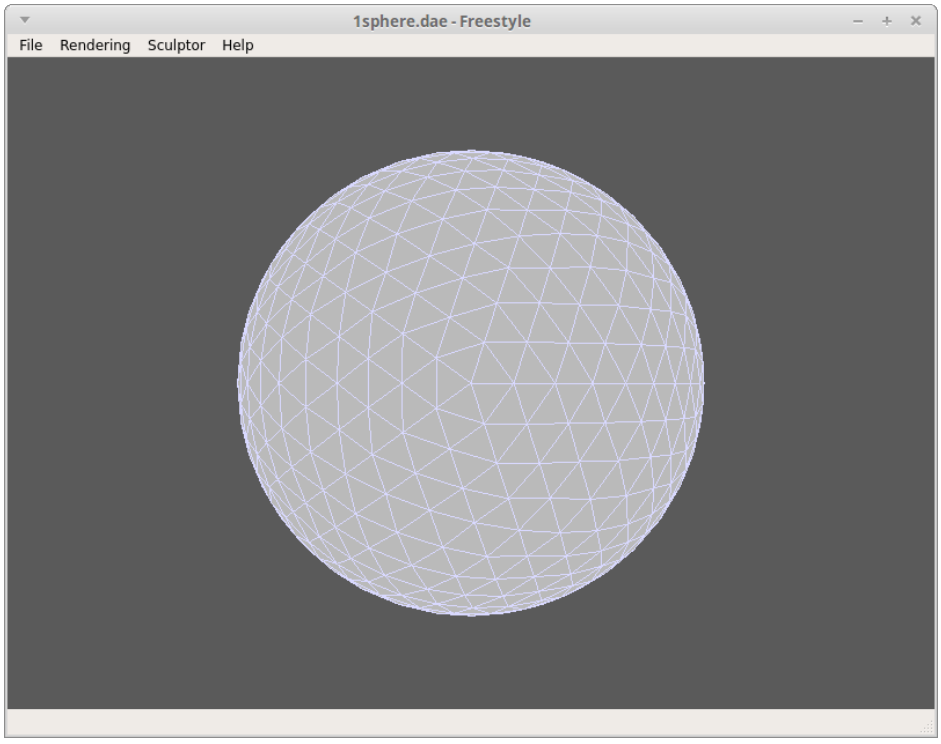
- **Collada** ( *.dae* )
- **Wavefront Object** ( *.obj* )
- **Stereolithography** ( *.stl* )
- **Stanford Polygon Library** ( *.ply* )

Notre module d'exportation accepte ces quatre formats, cependant, pour les .ply et les .dae les résultats ne sont pas satisfaisants, le format .dae rends le maillage plus clair et le .ply rend le maillage fortement spéculaire. De plus, le logiciel ne gère la sculpture que d'un objet à la fois. Si plusieurs sont préscents dans le fichier, seul le premier est pris en compte et seul le premier est sauvegardé.

## Conversion en maillage quasi-uniforme

Un des points essentiels du chef d'œuvre était la conversion du maillage utilisé dans le moteur de rendu en maillage quasi-uniforme. Pour rappel, un maillage quasi-uniforme est un maillage qui respecte une taille minimale et maximale de longueur d'arête ainsi que deux propriétés de ses attributs (ratio $max/min >= 2$ (1) et $4*d^2move <= d^2thickness - (max^2)/3$ (2)).

Le module de conversion a été implémenté et est fonctionnel. Les paramètres du maillage sont calculés automatiquement au chargement pour qu'ils soient adaptés au maillage et qu'ils respectent les propriétés (1) et (2). Tout d'abord les longueurs minimales, maximales et moyennes des arêtes sont déterminées. Ensuite, on cherche le min et le max le plus approprié compte tenu des statistiques précédement extraites, tout en s'assurant qu'ils respectent la propriété (1). Enfin on calcule *dthickness* et *dmove* de sorte qu'ils respectent la propriété (2).




*Nouvelle sculpture de base proposé avec son maillage quasi-uniforme*

## Maintien du maillage quasi-uniforme

Les bonnes propriétés du maillage quasi-uniforme permettent de résoudre des opérations habituellement complexes, comme le changement de topologie, relativement facilement. Il faut donc s'assurer de maintenir les propriétés du maillage tout au long de son évolution. Cette maintenance se fait via le module de conversion précédent. Le module dispose d'une autre fonction similaire n'appliquant la conversion que dans la région ciblée par la déformation. Ceci permettant d'optimiser les temps de traitement.

## Gestion des changements de topologie

En ce qui concerne la modification de la topologie du maillage, lorsque l'utilisateur utilise un outil il agit sur un ensemble de sommets dont on observe la distance de chacun par rapport à tous les sommets qui ne leur sont pas adjacents. Si la distance d'un sommet du champ à un sommet non adjacent est inférieure à une distance prédéfinie (*dthickness*), alors les deux sommets sont fusionnés en attachant leur premier anneau respectif.
Dans le papier, il est aussi également précisé qu'il faut une technique de *clean* pour nettoyer les éventuelles faces ou arêtes redondantes dans un anneau (notamment à la suite d'une fusion suivie d'une reconstruction en maillage quasi-uniforme). Cette technique consiste à enregistrer tous les nouveaux sommets crées à partir des conversions ponctuelles en maillage quasi-uniforme (il n'a pas été nécessaire d'implémenter cette partie dans notre programme du fait qu'OpenMesh ne crée pas de redondances dans sa structure).

La structure apportée par OpenMesh a grandement simplifié cette étape, notamment au niveau de la création et de la suppression d'éléments.

## Modification des paramètres du maillage quasi-uniforme par l'utilisateur

Le maillage quasi-uniforme se construit en fixant plusieurs paramètres (*min, max, dthickness, dmove*), nous avions souhaité donner la possibilité à l'utilisateur de choisir certains de ces paramètres et de pouvoir les modifier au cours de l'exécution du logiciel. Cependant ces paramètres et les propriétés qui doivent êtres respectées les rendent fortement dépendants entre eux, ce qui rend presque impossible la modification de l'un d'eux sans briser les propriétés. Nous avons donc décidé de ne pas donner cette possibilité à l'utilisateur et de faire tout automatiquement.

## Implémentation de 3 opérateurs de déformations: Etirer, Bomber/Creuser, Tordre

Le papier de recherche que nous devons implémenter décrit trois opérateurs de déformations. Nous avions spécifiés que nous les développerions tous.

**Etirer:**

La fonction Etirer permet de tirer un sommet et déplacer ses voisins afin que la maillage garde une consistence volumique. Ainsi, il n'est pas possible de modifier la topologie du maillage avec cet objet. L'utilisateur contrôle la déformation en sélectionnant le sommet et en bougeant la souris la où on veut le voir se déplacer.

**Bomber/Creuser:**

Cet opérator possède deux modes permettant de pousser l'ensemble des sommets "vers l'extérieur" où vers "l'intérieur de maillage". Cet outil peut permettre de créer des trous et de relier différentes parties du maillage. L'intéraction est indépendante du mouvement de la souris, seul le sommet cliqué et la taille de sphère influente importe.

**Tordre:**

Ce dernier opérateur permet d'effectuer une torsion sur les sommets de la zone d'influence qui vont effectués une rotation autour de l'axe normal du point cliqué. Cette opérateur, comme le précédent, peut entrainer des changements de topologie. L'utilisateur doit maintenir le sommet cliqué et rester appuyé pour tordre le maillage en ce point.

## Framework indépendant

Nous avions pour objectif de faire du framework QuasiUniform un framework ne dépendant uniquement que d'OpenMesh. L'architecture de ce framework lui permet de s'associer à tout type de projet. Dans notre application, l'application qui est charge d'assurer le lien est le *SculptorController.* Notre framework est également générique car il est possible de rajouter aisement des opérateurs de déformation.

# III) Améliorations possibles

## A. Exportation

Notre module d'exportation est minimaliste, il ne sauvegarde que le maillage et ignore les éventuels matériaux, textures, lumières, etc… Une amélioration serait donc de prendre en compte plus d'éléments de la scène dans la sauvegarde.

## B. Optimisation

Notre implémentation souffre d'une lenteur sur des maillages de taille moyenne. Il est peut être possible d'optimiser un peu mieux le code afin d'avoir du temps réel sur des maillages plus gros. Une idée serait de parralléliser le traitements qui sont pour la plupart indépendants les uns des autres.

# IV) Description de l'architecture du logiciel

Comme décrit dans le rapport conception, notre logiciel est divisé en 3 grandes parties :

- ➢ Moteur graphique "vortexengine"
- ➢ Application qui contient notre IHM
- ➢ Module de sculpture

Nous n'allons pas rentrer dans une description exhaustive de chaque module, mais nous allons plutôt détailler l'implémentation de chaque fonctionnalité que nous avons donné dans la partie II. De plus, nous ne détaillerons que les fonctionnalités qui nous ont vraiment demandé de la production de code.

## A. Conversion en maillage quasi-uniforme

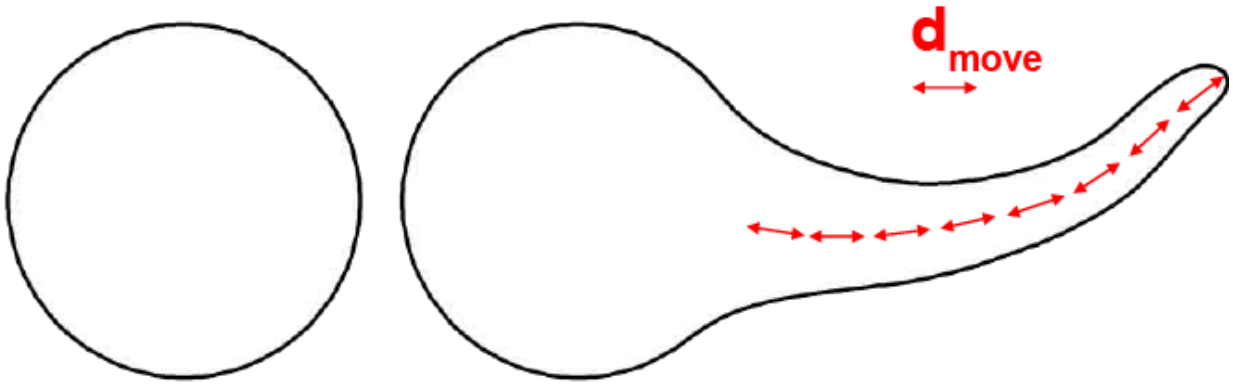
La conversion en maillage quasi-uniforme se fait à l'aide de la fonction statique *makeUniform* de la classe *QuasiUniformMeshConverter*. Cette fonction prend en paramètres les valeurs des longueurs minimales et maximales d'arêtes voulues. Après la conversion effectuée, on effectue une suppression effective des éléments inutiles à l'aide du garbage_collection d'OpenMesh.

## B. Maintien du maillage quasi-uniforme

Afin de maintenir le maillage quasi-uniforme, on applique la fonction statique *makeUniformField* de la classe *QuasiUniformMeshConverter* sur la partie du maillage qui a été déformé après chaque opération de déformation. En gardant le maillage quasi-uniforme, on garde les bonnes propriétés qui facilite la gestion des changements de topologie.

## C. Gestion des changements de topologie

Pour ce qui concerne le changement de topologie, on a besoin de connaître la position des sommets du champ que l'on est en train de modifier (*field_vertices*) et de la classe *TopologicalHandler* qui implémente les fonctions *handleJoinVertex* et *cleanup*.
La première gère la fusion de deux sommets non adjacents (le premier étant un sommet du champ de modification des outils et le second juste un sommet non adjacent) et est appelé dans la fonction *loop* de la classe *Sculptor* qui s'exécute, pour simplifier, à chaque déplacement *dmove* comme dans la figure ci-dessous :




Cette fonction est utilisée également pour le maintien du maillage en quasi-uniforme et en fin de traitement elle vérifie si l'outil utilisé est susceptible (ou pas) de déclencher un changement topologique avec la méthode *getTopologicalChange()* de la classe *Operator* qui est une méthode spécifié pour chaque opérateur *InfDefOperator, SweepOperator* et *TwistOperator.*

Si, comme dans le cas du *InfDefOperator* (Bomber-Creuser), il y a possibilité de changement topologique à chaque déplacement *dmove* alors on déclenche le traitement de fusion.
Pour cela on parcourt tous les sommets du champ *field_vertices*, puis on calcule la distance de chacun de ces sommets avec tous les autres sommets non adjacents. Et si une distance entre un sommet de *field_vertices* et un sommet non adjacent est inférieur à *dthickness*, alors on appelle la méthode *handleJoinVertex().*

Puis, on appelle la méthode*makeUniformField()* qui permet de rendre la zone de fusion quasi-uniforme comme pour le reste du maillage.

La fonction *cleanup* est appelée lorsque de nouveaux sommets sont crées dans le maillage à la suite d'une conversion en maillage quasi-uniforme, cette méthode gère les éventuelles redondances au niveau de la création des faces et des arêtes.
Par exemple, dans le cas de la fusion, pour chaque nouveau sommet créé à la suite d'une conversion en maillage quasi-uniforme, on stocke les nouveaux sommets dans le vector *connecting_edges*, puis lorsque on applique la fonction *cleanup* après la conversion sur chacun des *connecting_edges.*

## D. Etirer, Bomber/Creuser, Tordre

Le framework implémente trois opérateurs qui permettent de déformer le maillage. Ces opérateurs hérite de la classe générique *Operator* dont le rôle est de définir le comportement des transformations.

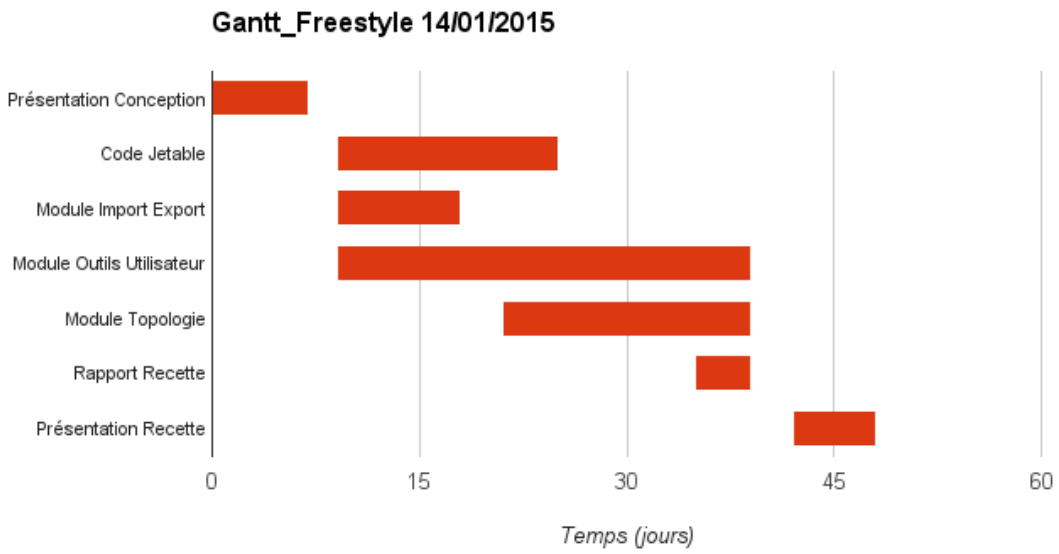
Chaque opérateur accède à divers paramètres qui lui permet d'effectuer son travail :

- Génériques : Ce sont les paramètres sur lesquels repose le principe de déplacement par champs vectoriel. Ils sont composés par exemple du sommet cliqué et du rayon de la sphère d'influence.
- Supplémentaires : Ce sont des paramètres qui influent le plus souvent sur l'allure de la déformation. C'est au développeur de s'assurer que ces paramètres sont correctement initialisés avant chaque transformation. Nous avons choisi de déléguer ce travail au contrôleur en charge du fonctionnement du module de sculpture.

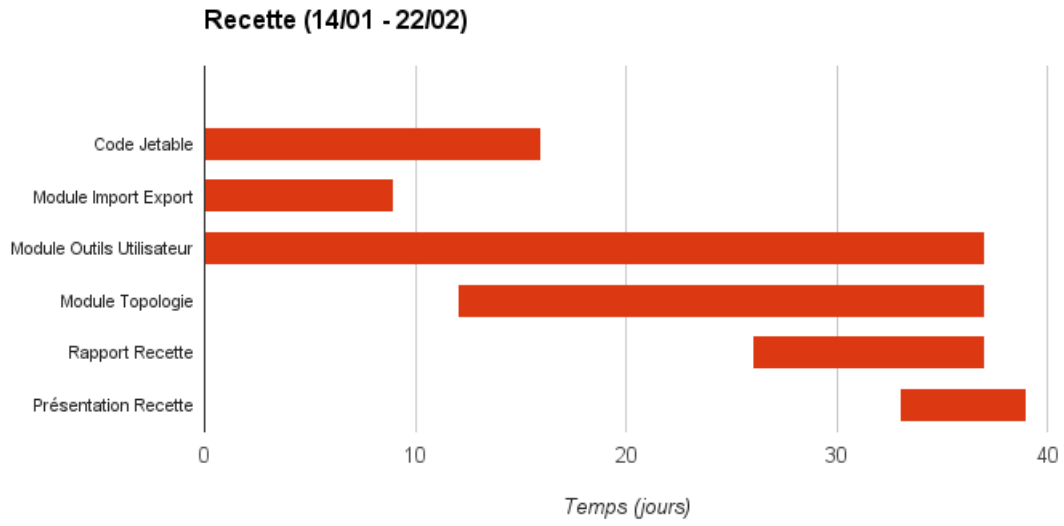
La classe *SculptorController* est la plus importante de l'application. Elle assure le lien entre notre framework et le reste de l'application. Elle reçoit les évenements provenant des vues permettant d'intéragir avec le module de sculpture (*ToolsDialog*, *ParametersDialog*, *OpenGLWidget*), les interpréte et les transmet à la classe *Sculptor* pour qu'il effectue les déformations adéquates. Comme tout contrôleur, son rôle est aussi de synchroniser les différents modules et de transmettre les changements de données aux vues de l'application.

# V) Planning final

Le diagramme ci-dessous montre la planification que nous avions au moment de la conception le 14/01.




Et ce diagramme montre le temps réel que nous avons mis pour réaliser les différentes tâches prévues dans les spécifications de départ.




Comme on peut le voir sur les deux diagrammes de Gantt, le développement des modules nous a pris plus de temps que prévu.

# VI) Conclusion

La majorité des fonctionnalités attendues ont été implémentées. Notre objectif était d'offrir un framework adaptable à  tout type de projet dans une application à l'architecture robuste. Il reste cependant certaines optimisations à effectuer pour garantir un fonctionnement en temps réel pour tous les tailles et niveaux de détails de maillage.