Rapport de stage

Dissection et anatomie numériques virtuelles 3D+t pour l'enseignement et la simulation chirurgicale

Clément PRENVEILLE

Effectué au laboratoire d'anatomie de Montpellier
Du 20 février au 31 juillet 2017

Tuteurs d'entreprise : M. Guillaume Captier et M. Gérard Subsol
Tuteur universitaire : M. Rodolphe Giroudeau

Master 2 informatique
Parcours IMAGINA

Remerciements

Je tiens à remercier mon tuteur professionnel M. Guillaume Captier pour les conseils qu'il m'a apportés concernant la partie médicale de mon stage, ainsi que mon co-tuteur M Gérard Subsol pour son aide sur la partie informatique.
Je remercie également mon tuteur universitaire M. Rodolphe Giroudeau pour son aide et ses conseils, et M. Mohamed Akkari pour avoir pris le temps d'identifier les structures anatomiques.
Je remercie enfin tout le personnel du LIRMM et plus particulièrement l'équipe ICAR pour m'avoir accueilli pendant ces 5 mois de stage.

Sommaire

1 Introduction

1.1 Contexte du stage

Dans le cadre du Master informatique IMAGINA, les étudiants doivent effectuer un stage d'une durée de 5 à 6 mois. Le stage que j'ai réalisé s'intitule "Dissection et anatomie numériques virtuelles 3D+t pour l'enseignement et la simulation chirurgicale". Ce projet est proposé et financé par le laboratoire d'anatomie de la faculté de médecine de Montpellier, en collaboration avec le LIRMM.
Les outils numériques sont de plus en plus utilisés dans le milieu médical et sont devenus un complément aux méthodes traditionnelles telles que la dissection. L'utilisation d'applications d'anatomie et de dissection virtuelles permet aux étudiants en médecine de se familiariser avec le corps humain avant de réaliser des dissections. Il est donc essentiel que ces applications soient très proches de la réalité afin de réduire la courbe d'apprentissage des étudiants.

1.2 Partenaires du projet

Laboratoire d'anatomie
Fondée au XII ème siècle, la faculté de médecine de Montpellier est la plus ancienne faculté de médecine en activité au monde. Elle a accueilli les plus grands médecins et chirurgiens de leur époque : Rabelais, Lapeyronie, Chaptal, Arnaud de Villeneuve, Gui de Chauliac.
Au cœur de cet établissement se trouve le laboratoire d'anatomie, structure dédiée à la recherche et l'enseignement à travers l'étude du corps humain. Bien que sa mission principale soit l'enseignement de l'anatomie humaine aux étudiants en médecine de la faculté de Montpellier, le laboratoire constitue également un lieu de formation privilégié pour les futurs chirurgiens.

Figure 1: Organigramme du laboratoire d'anatomie

LIRMM

Le LIRMM est le Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier. Il est dépendant de l'Université de Montpellier et du Centre National de la Recherche Scientifique (CNRS). Au sein de son département informatique se trouve l'équipe ICAR qui travaille sur l'interaction et le traitement de données visuelles telles que des images, des vidéos et des objets 3D. ICAR développe son activité selon 3 axes scientifiques : analyse et traitement, codage et protection, modélisation et visualisation.

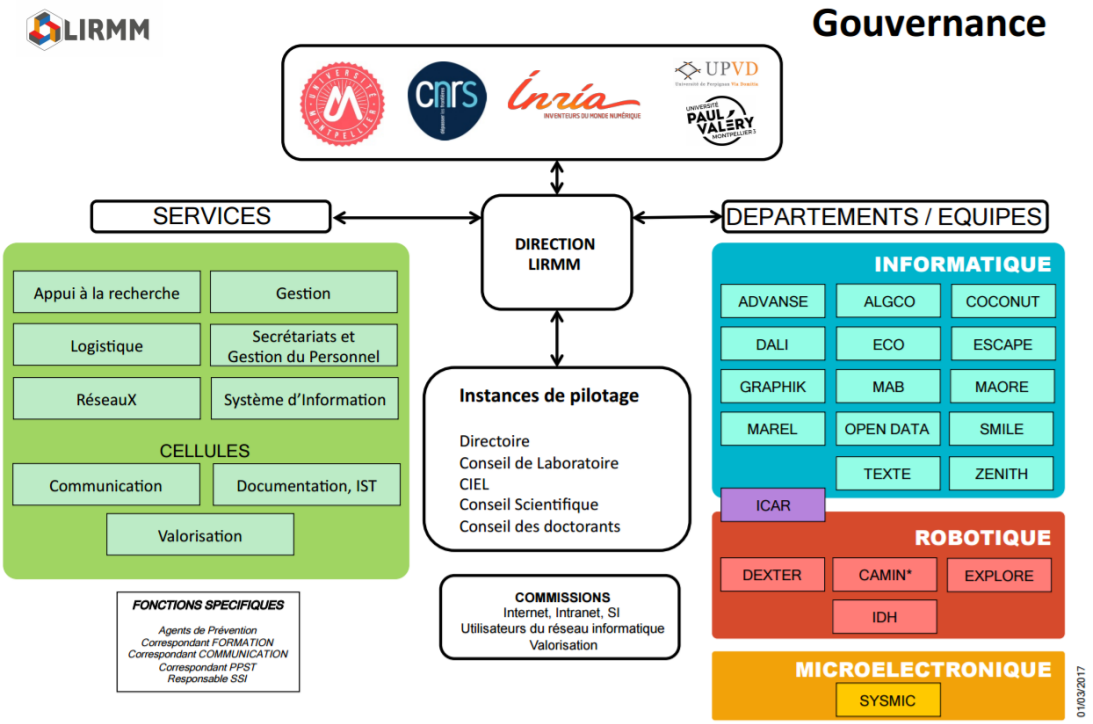


Figure 2: Organigramme du LIRMM

1.3 Présentation du sujet

Contexte

La dissection cadavérique est un outil traditionnel très performant pour l'enseignement de l'anatomie. Cependant, à cause du nombre croissant d'étudiants en médecine et du nombre limité de cadavres, cette pratique est de plus en plus rare. Cela cause deux problèmes principaux : les élèves réalisent peu de dissections durant leur scolarité et n'ont donc pas le droit à l'erreur, et les futurs chirurgiens ne peuvent pas suffisamment s'entraîner en répétant les gestes d'une opération. L'entraînement sur un modèle 3D virtuel ultra-réaliste permettrait de résoudre ces problèmes.

Des dizaines d'applications d'anatomie existent déjà mais elles utilisent toutes des mannequins modélisés sur ordinateur. Les structures anatomiques sont symétriques, colorées et facilement discernables (figures 3 à 5), mais cela ne reflète pas la réalité. A cause de ce décalage visuel, ces applications ne préparent pas assez l'étudiant à identifier les structures

anatomiques sur un corps humain. Le principe innovant derrière ce projet est de reconstituer un modèle 3D ultra réaliste à partir d'un vrai cadavre.

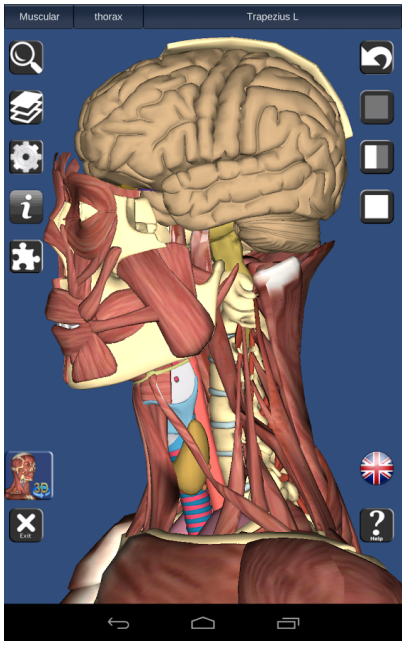


Figure 3: Application "3D Bones and Organs"

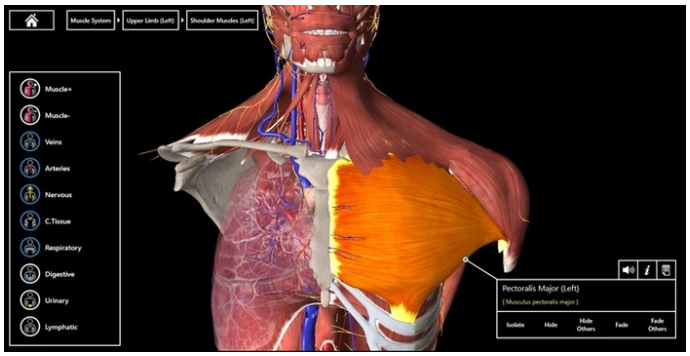


Figure 4: Application "Essential Anatomy"

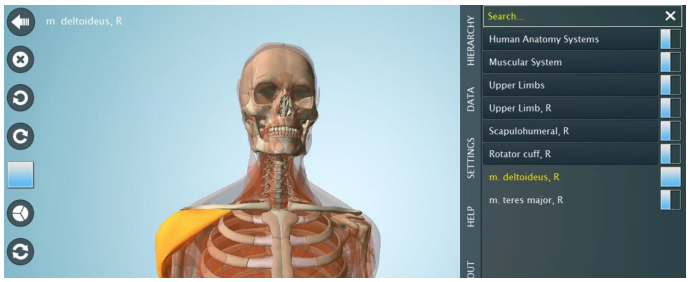


Figure 5: Application "Anatomy 3D"

Données

Une série de captures surfaciques ont été effectuées au laboratoire d'anatomie de Montpellier à l'aide du scanner 3D "Artec Spider". https://www.artec3d.com/3d-scanner/artec-spider

Ce scanner permet de modéliser l'objet scanné en un maillage surfacique avec une précision de 0.1 mm . Il peut capturer jusqu'à un million de points par seconde et ne nécessite pas l'utilisation de marqueurs visuels, ce qui rend la capture très rapide : environ 2 minutes pour scanner un buste humain. L'Artec Spider utilise également une caméra pour capturer la couleur de l'objet en même temps que sa géométrie. Il en ressort une image PNG de résolution 4K appelée texture qui est ensuite projetée sur le maillage 3D pour lui donner sa couleur.

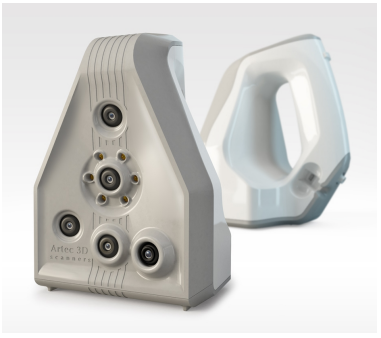


Figure 6: Scanner surfacique 3D Artec Spider

A la manière d'un appareil photo, le scanner 3D ne capture que ce qui est visible. Il faut

donc effectuer plusieurs captures du même cadavre à différents stade de la dissection afin de scanner les éléments anatomiques en surface comme ceux en profondeur

Les données que j'ai utilisées pour réaliser mon stage sont 8 maillages qui correspondent à une dissection de l'anatomie cervicale, du cou jusqu'à la trachée. Ces données ont été capturées en 2016 au laboratoire d'anatomie de Montpellier. Les maillages contiennent entre 2.6 et 7.6 millions de faces et sont tous accompagnés d'une texture de résolution 4K.

Figure 7: Couche 1 de l'anatomie cervicale

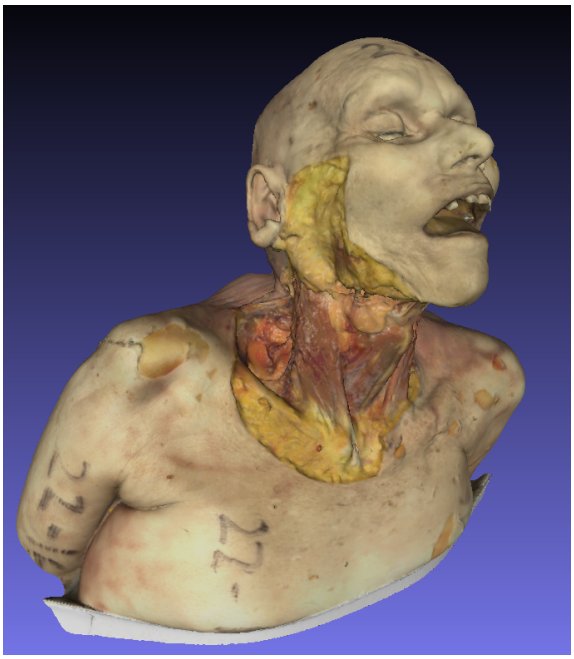


Figure 8: Couche 3 de l'anatomie cervicale

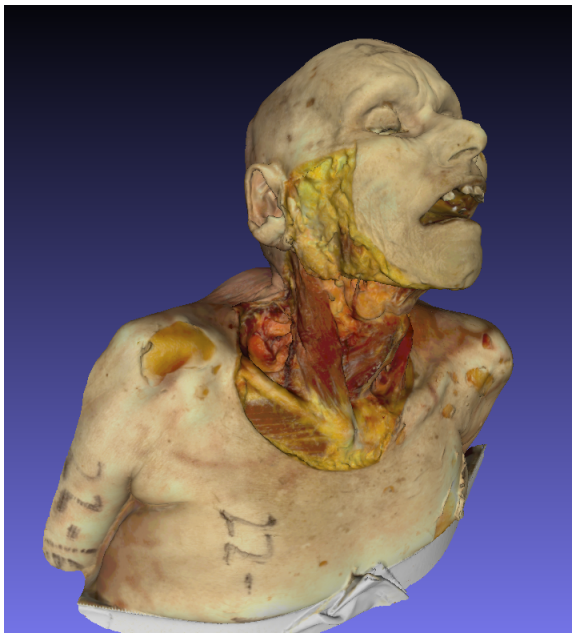


Figure 9: Couche 5 de l'anatomie cervicale

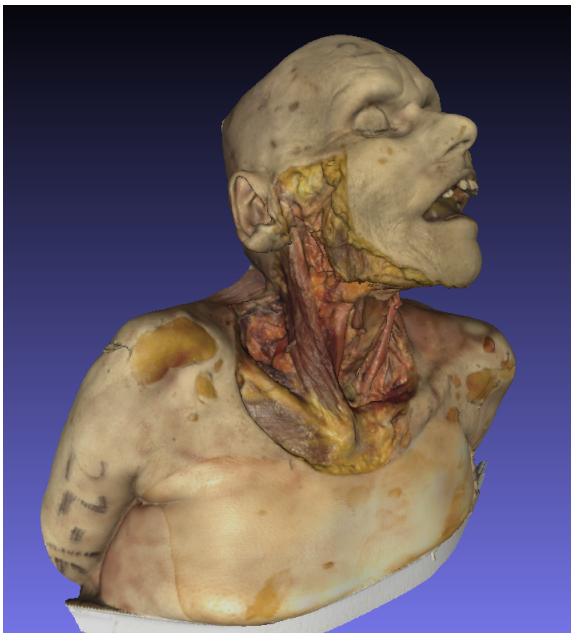


Figure 10: Couche 8 de l'anatomie cervicale

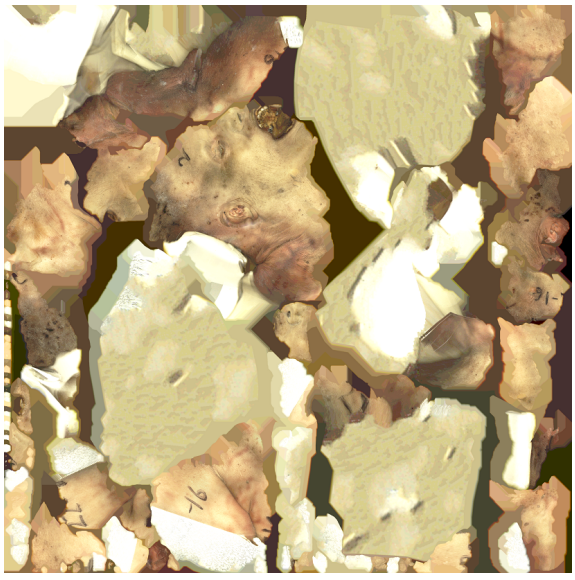


Figure 11: Texture de la couche 1

Figure 12: Texture de la couche 8

Le 18 avril 2017, une nouvelle série de captures a été effectuée au laboratoire d'anatomie lors d'une dissection du périnée féminin. A la suite de cette journée je dispose également de 7 maillages de dissection du périnée et des textures associées à ces maillages.

Objectif

L'objectif de ce stage est d'utiliser ces données pour créer deux applications : une application d'anatomie fonctionnant sur navigateur Web pour les étudiants en médecine et une application de simulation de dissection fonctionnant sur une tablette tactile. Pour cela il faut traiter les surfaces 3D, les recaler les unes sur les autres, et identifier les structures anatomiques.

1.4 État de l'art

Avant de me lancer dans la réalisation du projet, j'ai pris le temps d'étudier quelques articles de projets similaires au mien.

Novel Method of Anatomical Data Acquisition (2014)

Ref : Welsh E, Anderson P, Rea P, A Novel Method of Anatomical Data Acquisition Using the Perceptron ScanWorks V5 Scanner. International Journal on Recent and Innovation Trends in Computing and Communication ISSN: 2321-8169 Volume: 2 Issue: 8

Ce projet réalisé en Écosse en 2014 fut l'une des première utilisation du scanner surfacique en médecine. L'objectif était de créer une librairies d'images 3D anatomiques sans utiliser de IRM ou de scanner à rayons-X. En effet ces derniers demandent beaucoup de préparation et de moyens, ainsi qu'un gros travail de segmentation d'images. La capture surfacique s'est révélée être une alternative plus simple, rapide, et moins coûteuse.
Cette méthode avait toutefois des inconvénients, notamment l'absence de couleur sur les maillages : ils utilisaient des photos pour connaître la couleur et la texture des structures scannées. De plus les cadavres disséqués étaient formolés, ce qui donne un rendu peu naturel. Enfin, les captures sont réalisées en toute fin de dissection : on ne peut pas étudier les étapes de la dissection et observer les rapports entre les différents niveaux de profondeur.

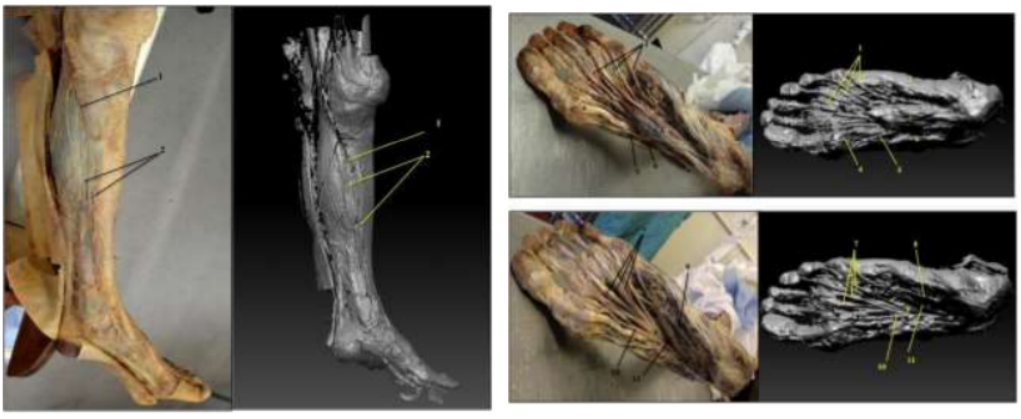


Figure 13: Captures surfaciques d'une dissection du pied, et photographies couleurs associées

D'autres travaux similaires existent mais personne n'a encore tenté de créer une application interactive d'anatomie ou de dissection à partir de données capturées par un scanner surfacique. Ce projet est expérimental et très innovant, et il n'existe pas encore d'études portant là-dessus. La technologie des scanners surfaciques est encore récente, et elle s'améliore d'année en année : l'Artec Spider utilisé pour obtenir les données de ce stage n'est sorti qu'en 2013.

Stage similaire (2016)
L'année dernière Killian Niollon, étudiant en licence professionnelle à l'université d'Aix-Marseille, a réalisé un stage sur le même sujet. Il a utilisé l'environnement WebGL pour afficher les maillages, et a implémenté des contrôles tels que le zoom et la rotation du modèle. Il a ensuite créé une interface pédagogique et un mode "évaluation" où l'utilisateur doit renseigner le nom de certaines structures anatomiques.
L'application était fonctionnelle mais elle comportait plusieurs défauts, parmi lesquels :

- Maillages grandement simplifiés. On perd en qualité et donc en réalisme
- Les structures anatomiques ne sont pas identifiées. On ne connaît ni leur position ni leur forme dans l'espace 3D
- Application liée aux données (non modulaire)

Pour ces raisons il a été décidé de reprendre le stage avec une nouvelle approche et d'utiliser un moteur de jeux pour réaliser l'application. Bien que je n'ai pas repris de code ou d'autres éléments du stage de Killian, son application m'a aidé à voir ce qu'on attendait de moi lors de ce stage.

2 Problématique

2.1 Problèmes

Taille des données
Le premier problème qui se pose est celui de la taille des données : nous disposons de 8 maillages de très haute qualité qui contiennent chacun jusqu'à 7.6 millions de faces. Le chargement d'un seul de ces maillages sur un logiciel de rendu 3D comme Blender ou MeshLab prend plus de 30 secondes, et l'utilisation de données aussi imposantes rend difficile le traitement en temps réel. Les maillages peuvent être simplifiés mais on perdrait alors les éléments les plus fins, tels que les fibres musculaires et les nerfs, ce qui nuirait au réalisme de l'application. Un des défis de ce projet est d'obtenir une application performante tout en gardant la meilleur qualité possible pour les données.
Pendant le stage j'ai également découvert que les bibliothèques de rendu 3D telles que OpenGL ou Direct3D ne peuvent pas charger de maillages contenant plus de 65 000 sommets. La plupart des logiciels de rendu utilisent une de ces bibliothèques et il est donc

impossible d'afficher un maillage de plusieurs millions de faces en un seul morceau, quel que soit l'environnement choisi.

Identification des structures anatomiques

Contrairement aux applications d'anatomie classiques où les éléments anatomiques sont modélisés un par un, l'utilisation de la capture surfacique sur un vrai cadavre implique que toutes les structures sont agglomérés en un seul gros maillage 3D. Afin de connaître la forme et la position des éléments anatomiques dans l'espace 3D, il faut procéder à une identification des structures anatomiques du modèle. Cette opération n'est pas réalisable de manière automatique car les structures ont des couleurs très proches entre elles et leurs contours sont souvent peu visibles. Une des missions de mon stage sera donc de créer une troisième application qui aura pour but d'étiqueter les structures anatomiques sur les données.

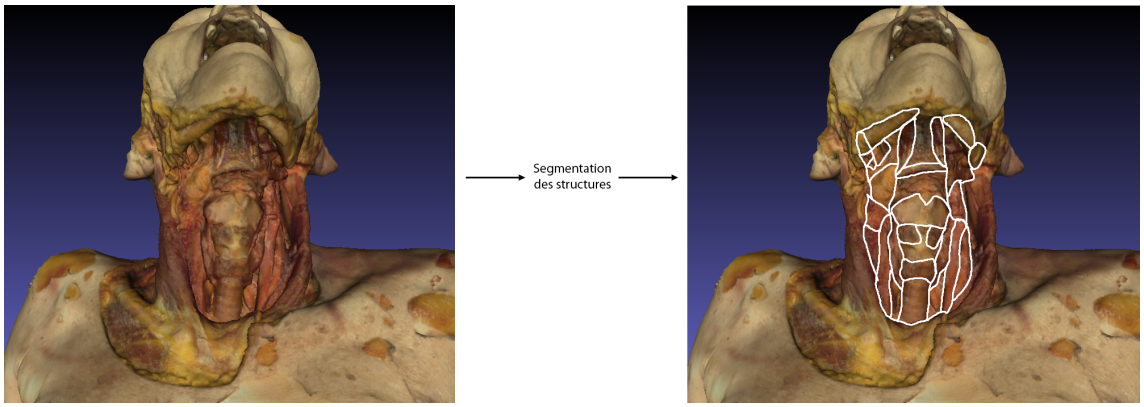


Figure 14: Problème d'identification des structures anatomiques

Multi-plateforme

On souhaite pouvoir utiliser l'application d'enseignement de l'anatomie sur navigateur, et l'application de simulation de dissection sur une tablette tactile de marque Wacom. Bien que ces applications aient un principe et un objectif différent, elles utilisent toutes les deux les mêmes fonctionnalités de base telles que le chargement des maillages, le contrôle du modèle et le chargement de la position des structures anatomiques. Créer deux applications différentes dans deux environnements de travail différents serait donc une perte de temps. L'idéal est de réaliser les deux applications sur le même environnement puis de les exporter sous forme d'application web pour l'une, et d'application locale tournant sur tablette pour l'autre. De cette manière les fonctionnalités partagées entre les applications n'ont besoin d'être codées qu'une seule fois.

2.2 Environnement et cahiers des charges

Environnement de travail

Lorsque le stage de 2016 a été réalisé, il avait été conclu que l'outil WebGL n'était pas assez puissant et ne disposait pas d'assez de fonctionnalités pour compléter l'ensemble des missions. Pour cette raison, le sujet a été reproposé sous un nouvel environnement : Unity3D.

Figure 15: Moteur de jeux Unity

Unity (https://unity3d.com/) est un moteur de jeux 2D et 3D qui est très utilisé tant par les gros studios que par les développeurs indépendants. Bien que ce logiciel soit principalement utilisé pour créer des jeux vidéos, on peut également s'en servir pour développer des applications de réalité virtuelle, de réalité augmentée, ou encore des simulations. Unity a la particularité d'être multi-plateforme : un projet peut facilement être exporté vers une multitude d'environnements tels que Windows, Mac, Android, iOS, ou encore WebGL. De plus, ce moteur à la particularité de proposer une licence gratuite sans aucune limitation au niveau du moteur, ce qui en fait le meilleur rapport qualité/prix parmi tous les moteurs de jeux existants. Pour toutes ces raisons, Unity s'est imposé comme un excellent choix pour réaliser ce projet.
Deux langages de programmation sont supportés par ce moteur : JavaScript et C#. C# offre des performances légèrement meilleures et possède plus de documentation, c'est donc le langage que j'ai choisi d'utiliser pour ce projet. L'IDE que j'ai utilisé pour l'écriture du code est Visual Studio 2015.

Enfin j'ai utilisé MeshLab (http://www.meshlab.net/), une application de rendu 3D légère et gratuite, pour visualiser et pré-traiter les maillages.

Cahier des charges de l'application d'anatomie

Cette première application est destinée aux étudiants en médecine. L'objectif est de leur enseigner l'anatomie humaine et de les entraîner à reconnaître différentes structures anatomiques sur un vrai cadavre avant de réaliser eux-mêmes une véritable dissection. L'entraînement sur l'application d'anatomie permettra aux élèves de faire moins d'erreurs lors de la dissection et ainsi d'en tirer un meilleur parti.
L'objectif est d'utiliser cette application en classe de médecine, elle doit donc pouvoir fonctionner sous format Web et tourner sur plusieurs machines simultanément.
Le principe de cette application est d'afficher les couches de dissection une par une : il n'y aura jamais plus d'une couche affichée à l'écran. L'utilisateur peut circuler librement entre les couches de manière dynamique.

L'application doit disposer de 3 modes différents. Le mode *découverte* permet à l'étudiant de se familiariser avec l'application et les contrôles. Il peut également librement réviser la position des structures anatomiques en cliquant sur le modèle. Le deuxième mode est le mode *évaluation*. L'utilisateur doit répondre à une série de questions avec une contrainte de temps. A la fin de l'évaluation un score est calculé et affiché à l'utilisateur. Il existe deux types de questions : nommer une structure anatomique pointée par l'application, ou au contraire pointer une structure nommée par l'application. Dans le premier cas la question proposera plusieurs réponses afin d'éviter les erreurs de mise en forme et d'orthographe ; dans le deuxième cas l'utilisateur devra circuler entre les couches pour faire apparaître la bonne structure, puis cliquer dessus. Enfin le troisième mode est le mode *parcours*, qui consiste à identifier une liste de structures dans un ordre spécifique. Ce parcours est paramétrable par les professeurs et sert à évaluer les étudiants sur des structures spécifiques.

Cahier des charges de la simulation de dissection

La deuxième application a pour but de simuler une dissection pour permettre aux futurs chirurgiens de répéter les gestes nécessaires à l'opération et à la dissection, et ainsi remédier au manque de cadavres.

Le laboratoire d'anatomie de Montpellier dispose d'une tablette de marque Wacom qui utilise le système d'exploitation Windows. Cette tablette possède un stylet de précision avec détecteur de pression. L'objectif est de faire tourner la simulation en local sur la tablette, et d'utiliser le détecteur de pression pour obtenir un rendu le plus réaliste possible. Contrairement à l'application d'enseignement, ici toutes les couches sont affichées en même temps. Le principe est d'aligner et de superposer les couches les unes sur les autres de manière à avoir la première couche en surface, et la dernière couche en profondeur. Au lancement de l'application, seule la couche 1 sera visible. L'utilisateur peut ensuite utiliser le stylet à la manière d'un scalpel pour découper le maillage et faire apparaître les couches inférieures. Les couches les plus profondes ne peuvent être vues que lorsque toutes les couches supérieures ont été découpées. Afin d'éviter que le modèle ne disparaisse, la dernière couche n'est pas découpable.

La pression exercée par l'utilisateur avec le stylet peut être utilisée de deux façons pour augmenter le réalisme de l'application. Premièrement, certains éléments anatomiques sont moins fragiles que d'autres : découper la peau ou le muscle demande donc plus de pression que découper les veines ou les nerfs. Deuxièmement si l'utilisateur appuie trop fort, il risque d'endommager les structures anatomiques,ce qui peut être simulé en prolongeant le découpage du maillage sur les couches inférieures.

L'application de dissection possède les 3 mêmes modes que l'application d'anatomie. Le mode *découverte* permet à l'utilisateur de s'entraîner librement et de réinitialiser le modèle pour recommencer une dissection. Le mode *évaluation* impose une limite de temps pour arriver jusqu'à la dernière couche de dissection. Enfin, dans le mode *parcours* l'utilisateur doit réaliser un scénario d'opération chirurgicale, comme par exemple retirer une structure

anatomique néfaste présente sur la dernière couche en disséquant tout autour de celle-ci.

Cahier des charges de l'application de pré-traitement (identification des structures anatomiques)
Avant de pouvoir utiliser les données dans les deux applications précédentes, les éléments anatomiques doivent être identifiées manuellement par un professeur d'anatomie. Une des missions de ce stage consiste à développer une application qui permette à une personne sans connaissance informatique d'identifier ces structures.
A la manière de l'application d'anatomie, les couches de dissection sont affichées unes par unes et l'utilisateur peut circuler librement entre elles. Il pourra alors créer des structures anatomiques, les nommer, et les supprimer librement. Il sélectionnera ensuite des échantillons du modèle 3D et les ajoutera aux structures qu'il aura définies. Une fois toutes les structures définies, le professeur pourra les exporter et les charger dans les applications d'anatomie et de dissection
Voici un exemple de l'utilisation de cette application :

1. Créer une nouvelle structure
2. La renommer "Clavicule"
3. Sélectionner les parties du maillage qui correspondent à la clavicule
4. Sauvegarder les structures

Une fois ceci fait, la clavicule est repérable dans l'espace. On peut alors demander à un étudiant de cliquer sur la clavicule et vérifier s'il a cliqué au bon endroit ou non.

Lorsque le professeur sauvegarde les structures, celles-ci sont exportées dans plusieurs fichiers sous forme de texte. Ces fichiers texte sont ensuite chargés par les applications d'anatomie et de dissection. L'identification des structures est donc une opération à ne faire qu'une seule fois puisque les structures anatomiques ne changent jamais.
L'intérêt de créer une application consacrée à l'identification des structures est que le laboratoire d'anatomie est susceptible de prélever de nouvelles données dans les mois ou les années à venir, et ces nouvelles données devront à leur tour être segmentées.

3 Développement

3.1 Prétraitement des données

Filtre utilisé
Les maillages de départs sont trop volumineux pour être utilisés tels quels : au chargement des données Unity ne répond plus et finit par planter après plusieurs minutes de chargement. Il a donc fallu effectuer une simplification des maillages, opération qui consiste à réduire le nombre de faces et de points dans un maillage, réduisant ainsi sa complexité.

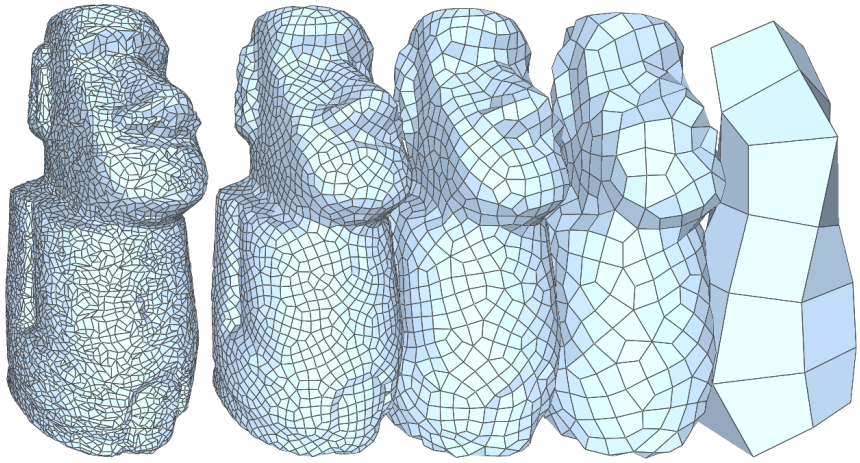


Figure 16: Simplification de maillage

MeshLab propose 3 filtres de simplification qui utilisent différents algorithmes pour réduire le nombre d'éléments du maillage. Le rendu des trois filtres est très similaire et n'est pas discernable à l'œil nu, mais seul l'un d'entre eux supporte la texture du maillage. En effet l'opération de simplification modifie la topologie du maillage (structure des sommets et triangles) ce qui rend l'ancienne projection de la texture sur le maillage invalide. Il faut alors recalculer une projection de la texture sur l'objet simplifié pour obtenir le même rendu qu'avant la simplification. Le filtre de MeshLab que j'ai utilisé s'appelle *"Simplification: Quadric Edge Collapse Decimation (with texture)"*.

Première simplification

Les maillages de départ sont tellement grands que Unity ne répondait plus lorsque j'essayais de les charger. J'ai donc réalisé une première simplification afin d'obtenir des maillages plus légers sur lesquels je pouvais travailler temporairement. J'ai décidé de réduire le nombre de triangles de toutes les couches à 100 000, ce qui reste assez conséquent mais permet tout de même un chargement rapide sur Unity. Sur la couche la plus volumineuse, cela correspond à une réduction de 99% du nombre de faces.

Deuxième simplification

Après avoir travaillé pendant plusieurs mois sur les maillages de 100 000 faces, j'ai discuté avec mon tuteur professionnel M. Captier de ce problème pour trouver un compromis entre la qualité du maillage et les performances de l'application. Certains éléments du corps humain tels que les fibres musculaires ou les nerfs sont de très petite taille et se font effacer lorsqu'on simplifie le maillage. Le modèle perd alors en réalisme, ce qui va totalement à l'encontre des objectifs de ce projet. D'un autre côté, il est impossible de conserver les données dans leur taille originale car elles ne peuvent pas être chargées sur Unity. La solution trouvée pour résoudre ce problème a été de réaliser une simplification partielle des données. Sur toutes les couches dont nous disposons, la peau du cadavre représente à chaque fois plus de 75% de la superficie du maillage. Or la peau est lisse et ne contient

pas de petits éléments qui soient pertinents (les rides apparaissent sur la texture). Les parties du maillage qui correspondent à la peau peuvent donc être grandement simplifiées sans affecter le réalisme du modèle. Pour réaliser ces sélections partielles, j'ai utilisé l'outil pinceau de MeshLab qui permet de sélectionner certaines faces du maillage. J'ai ensuite peint les zones disséquées du cadavre puis j'ai inversé la sélection pour ne garder que la peau.

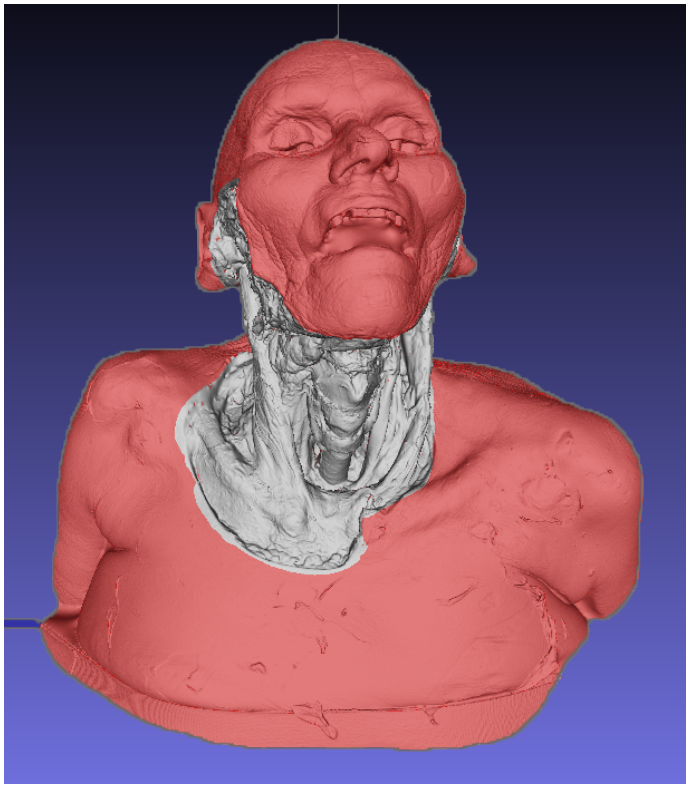


Figure 17: Sélection de la peau sur la couche 7

J'ai ensuite utilisé le filtre de simplification *Quadric Edge Collapse Decimation with texture* pour réduire le nombre de faces de la peau sélectionnée à 50 000. Cette opération a permis de réduire la taille des maillages de plus de 75%, passant par exemple de 7.3 millions de faces à 1.9 millions sur la septième couche. La première couche est particulière puisqu'elle représente le cadavre avant la dissection et ne contient donc que de la peau. Cette couche a donc été entièrement simplifiée, passant de 2.6 millions de faces à 20 000, ce qui représente une réduction de 99% du nombre de triangles.

Couches	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Nb faces avant	2.6 M	2.6 M	2.6 M	4.3 M	7.4 M	7.2 M	7.3 M	4.5 M
Nb faces après	20 000	500 000	530 000	1.1 M	1.8 M	1.7 M	1.9 M	1.1 M
Réduction	99%	81%	79%	74%	76%	76%	75%	76%

Cette simplification a également entraîné une diminution importante de la taille des données : les 8 maillages de base (sans texture) font plus de 6.4 Go tandis que les 8 maillages simplifiés font 1 Go (réduction de 85%).

Couches	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Taille avant	534 Mo	521 Mo	516 Mo	879 Mo	1.47 Go	1.44 Go	712 Mo	433 Mo
Taille après	2.11 Mo	57 Mo	61 Mo	131 Mo	212 Mo	200 Mo	217 Mo	131 Mo
Réduction	99.6%	89%	88%	85%	86%	86%	70%	70%

Ci-dessous une capture d'écran montrant le résultat de la simplification. On remarque facilement la frontière entre la partie disséquée qui est constituée d'une multitude de petits triangles, et le visage constitué de triangles 100 fois plus gros.

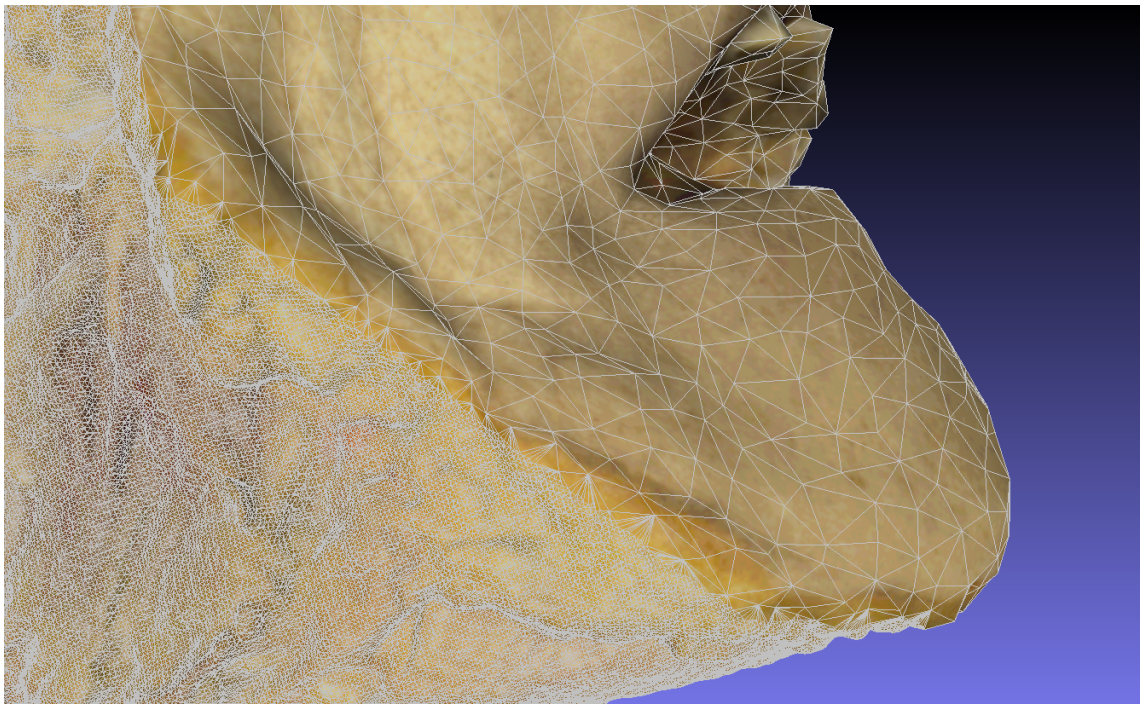


Figure 18: Simplification partielle du maillage

3.2 Chargement et affichage des données

Importation des données

Mon premier objectif fut de charger les maillages sur Unity pour pouvoir les faire apparaître à l'écran. Pour cela, Unity facilite grandement les choses puisque qu'il suffit de déposer les fichiers de maillage sur l'interface pour les importer. En important un maillage de 100 000 faces Unity averti l'utilisateur qu'un maillage ne peut pas contenir plus de 65 000 sommets. Cette limite découle de la bibliothèque graphique OpenGL qui est utilisée par le moteur pour afficher des objets dans une scène.

Pour contourner cette limite, Unity divise le maillage en plusieurs sous-maillages qui contiennent chacun moins de 65 000 sommets. Les 8 maillages importés ont été divisés en 2 sous-maillages chacun car ils contenaient tous entre 65 000 et 130 000 sommets.

Hiérarchie des données

Au lancement de l'application, la scène ne contient qu'un objet vide appelé Models. Cet objet contient le script LoadData qui va créer d'autres objets ainsi que charger les maillages et les textures dans la scène. Le script cherche les maillages dans un dossier spécifique. Si le dossier n'est pas vide, cela signifie qu'il existe au moins un maillage à afficher. Le script va alors sélectionner le premier fichier du dossier et va créer un nouvel objet enfant de Models. Ce nouvel objet appelé Model_1 va contenir tous les sous-maillages qui forment la première couche de dissection. Comme un objet ne peut pas contenir plusieurs maillages, le script créé un autre objet appelé Part_1 qui est enfant de Model_1 et qui va contenir le premier maillage sélectionné par le script. Le premier maillage du dossier est alors affiché dans la scène.

Le script sélectionne le fichier suivant et analyse son nom, ce qui lui permet de déterminer s'il s'agit d'un sous-maillage de la même couche, ou du premier maillage de la couche suivante. Dans le premier cas, il créé un nouvel objet Part_j qui va contenir le maillage, et qui est enfant de l'object Model_i actuel. Dans le second cas il créé un objet Model_i+1 enfant de Models, puis un objet Part_1 enfant ce Model_i+1, qui contiendra le maillage. Le script continue ce traitement jusqu'à avoir affiché tous les maillages présents dans le dossier.

Ci-dessous un diagramme montrant la hiérarchie des objets, dans le cas ou les maillages de base ont été divisés en 2 sous-maillages par Unity. On dispose alors de 8 objets Model_i qui contiennent chacun 2 objets Part_1 et Part_2.

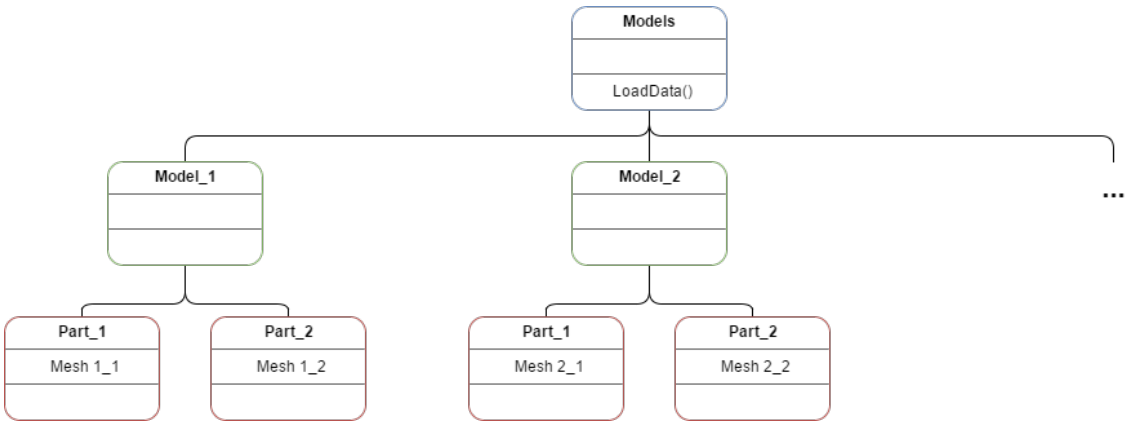


Figure 19: Hiérarchie des objets dans Unity, version 1

La hiérarchie obtenue est très solide et s'adapte bien aux limites d'Unity (un seul maillage par objet). Grâce à cette structure on n'est pas limité par le nombre de couches, ni par le nombre de sous-maillages, ce qui rend très facile le chargement de nouvelles données.

Affichage des maillages

Même si les maillages sont divisés en plusieurs morceaux, leur topologie est inchangée ce qui signifie que la projection de la texture reste valide. En affichant les sous-maillages côte-à-côte, il est impossible de remarquer que le modèle a été divisé en plusieurs morceaux. L'affichage du modèle se fait donc de manière classique.

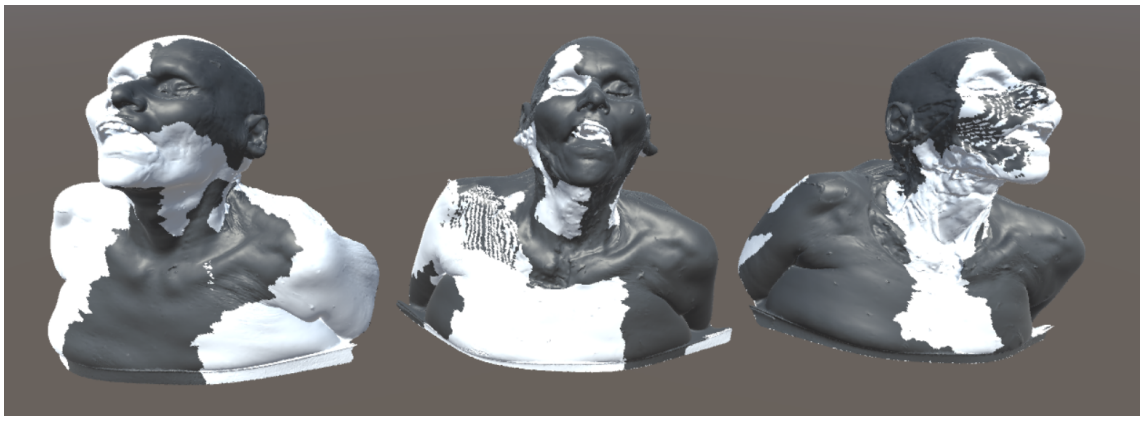


Figure 20: Sous-maillages mis en évidence sur les couches 1 à 3

3.3 Contrôles du modèle

Une fois les données affichées, ma seconde mission fut d'implémenter des contrôles pour permettre à l'utilisateur de manipuler le modèle. L'avantage de la hiérarchie détaillée plus haut est qu'une transformation appliquée à l'objet Models s'appliquera à toutes les couches de dissections, qu'elles soient affichées ou non. Cela signifie que si l'utilisateur fait une rotation de la couche 1, puis décide d'afficher la couche 2, celle-ci sera orientée de la même façon, ce qui conservera l'effet de transition dynamique entre les couches.
J'ai commencé par implémenter les contrôles de base que l'on retrouve dans tous les logiciels de rendu 3D : la translation, la rotation, et le zoom. Puis j'ai ajouté des contrôles plus spécifiques tels que le contrôle de la lumière, et la circulation entre les couches.

Translation

La translation consiste à déplacer le modèle sur les axes X (de gauche à droite) et Y (de haut en bas). Cette transformation est très utile lorsqu'on l'associe au zoom pour se concentrer sur un endroit spécifique du modèle, ou bien pour recentrer le maillage devant la caméra.
J'ai codé la translation de la manière suivante : au premier clic de souris, la position du curseur est stockée, puis à chaque image suivante tant que l'utilisateur garde le clic enfoncé, je calcule la différence entre la position stockée et la nouvelle position du curseur (deltaPosition). J'inverse la différence selon l'axe X puis l'ajoute à la position actuelle du modèle. Enfin je mets à jour la position du curseur stockée pour l'image suivante. Avec cet algorithme l'utilisateur peut cliquer n'importe où sur l'application, et son mouvement de souris est répercuté sur le modèle, ce qui donne une translation très facile à prendre en main.

```
void UpdateTranslating() {

    //if clicked this frame
    if (Input.GetMouseButtonDown(0)) {
        //update mouse position
        lastMousePosition = Input.mousePosition;

    //if still clicking
    } else if (Input.GetMouseButton(0)) {
        //compute the difference in position
        Vector3 deltaPosition = Input.mousePosition -
           lastMousePosition;
        //invert x position
        deltaPosition.x = -deltaPosition.x;
        //assign new position
        transform.position += deltaPosition;
        //update mouse position
        lastMousePosition = Input.mousePosition;
    }

}
```

Rotation

Une autre transformation primordiale pour toute application 3D est la possibilité de faire pivoter le modèle. L'algorithme de rotation reprend le même principe que celui de translation pour ce qui est du calcul du vecteur deltaPosition. Ce vecteur est normalisé pour éviter les mouvements de rotation trop brusques, puis j'utilise la fonction Rotate() de Unity pour faire pivoter l'objet selon deltaPosition.

```
void UpdateRotating() {

    //if clicked this frame
    if (Input.GetMouseButtonDown(0)) {
        //update mouse position
        lastMousePosition = Input.mousePosition;

    //if still clicking
    } else if (Input.GetMouseButton(0)) {
        float deltaX = Input.mousePosition.x - lastMousePosition.x;
        float deltaY = Input.mousePosition.y - lastMousePosition.y;
        //for touch support
        if (Input.touchCount > 0) {
            deltaX = Input.touches[0].deltaPosition.x;
```

```
            deltaY = Input.touches[0].deltaPosition.y;
        }
        //rotate object
        transform.Rotate(new Vector3(-deltaY, -deltaX,
            0).normalized * Time.deltaTime * 150, Space.World);
        //update mouse position
        lastMousePosition = Input.mousePosition;
    }

}
```

Une fois la rotation implémentée, j'ai pu constater qu'elle n'était pas naturelle et qu'il était difficile de pivoter le modèle comme on le souhaitait. Cela est du au fait que le centre de la rotation se trouvait tout en bas du maillage et non pas au centre de celui-ci. Pour résoudre ce problème j'ai ajouté une fonction au lancement de l'application qui calcule le barycentre des sous-maillages de la première couche puis actualise la position de toutes les couches afin de centrer le point de pivot sur l'objet principal Models.

```
void InitializeModelPosition() {

    //  global barycenter
    Vector3 modelBarycenter = Vector3.zero;

    //for each submesh in layer 1
    foreach (Transform child in transform.GetChild(0))
        //  add its center to barycenter
        modelBarycenter +=
            child.GetComponent<MeshCollider>().bounds.center;

    //  divide barycenter by nb of submeshes
    modelBarycenter /= transform.GetChild(0).childCount;

    //  move this object to barycenter position
    transform.position = modelBarycenter;
    //  and move layers to compensate
    foreach (Transform child in transform)
        child.localPosition = -modelBarycenter;

}
```

Zoom

Le troisième contrôle est différent des deux précédents car c'est une transformation appliquée à la caméra, et non pas aux maillages 3D. Il est possible de modifier l'échelle des maillages 3D de manière dynamique, mais c'est une opération très coûteuse et le résultat n'est pas fluide. J'ai donc décidé d'effectuer un zoom avec la caméra, ce qui donne un résultat similaire de manière beaucoup plus performante.
L'algorithme de zoom consiste à calculer les mouvements de curseur sur l'axe X et de modifier le champ de vision de la caméra en conséquence. Déplacer la souris vers la gauche en maintenant le clic enfoncé va augmenter le champ de vision de la caméra, ce qui donne l'impression que l'objet s'éloigne de nous. Inversement, déplacer la souris vers la droite en cliquant va diminuer le champ de vision et donner une impression de zoom avant.

```
void UpdateScaling() {

    //if clicked this frame
    if (Input.GetMouseButtonDown(0)) {
        //update mouse position
        lastMousePosition = Input.mousePosition;

    //if still clicking
    } else if (Input.GetMouseButton(0)) {
        //compute X difference
        float deltaX = Input.mousePosition.x - lastMousePosition.x;
        //update camera field of view
        Camera.main.GetComponent<CameraBehavior>().UpdateFieldOfView(deltaX
            * 0.1f);
        //update mouse position
        lastMousePosition = Input.mousePosition;
    }

}
```

Lumière

Le contrôle de la lumière est un outil moins répandu mais qui est primordial pour une application 3D dans le domaine médical. En effet, lors d'une opération les chirurgiens sont souvent amenés à modifier l'orientation et l'intensité de la lumière. De plus il est possible qu'en pivotant le modèle dans une position spécifique la lumière se reflète trop fortement sur le modèle et cause une surexposition (figure 21). J'ai donc rajouté un contrôle de la lumière, qui permet à la fois de modifier l'intensité mais aussi l'orientation de la source lumineuse directionnelle.

Figure 21: Modèle surexposé

L'algorithme du contrôle de la lumière consiste à calculer une fois de plus la variation de la position du curseur. J'utilise ensuite la composante X de deltaPosition pour modifier l'angle de la lumière, et la composante Y pour modifier son intensité. Il m'a paru naturel d'utiliser un mouvement vertical pour l'intensité et un mouvement horizontal pour la direction.

```
void UpdateMovingLight() {

    //  if clicked this frame
    if (Input.GetMouseButtonDown(0)) {
        lastMousePosition = Input.mousePosition;

    // if still clicking
    } else if (Input.GetMouseButton(0)) {
        float deltaX = Input.mousePosition.x - lastMousePosition.x;
        //  rotate light according to X variation
        SceneLight.transform.Rotate(Vector3.up, deltaX * 0.3f,
           Space.World);

        float deltaY = Input.mousePosition.y - lastMousePosition.y;
```

```
        //  change intensity according to Y variation
        SceneLight.GetComponent<Light>().intensity += deltaY *
            0.002f;
        //update mouse position
        lastMousePosition = Input.mousePosition;
    }

}
```

Figure 22: Modèle après diminution de l'intensité de la lumière

Figure 23: Modèle après changement de l'orientation de la lumière

Changement de couche

La fonction de changement de couche permet de naviguer librement entre les différents modèles. Pour obtenir un effet dynamique j'ai limité la circulation aux couches adjacentes : on ne peut pas passer de la couche 1 à la couche 3 sans passer par la couche 2.
Ce contrôle est codé différemment des 4 précédents. J'utilise une variable globale currentModel qui correspond à l'index du modèle à afficher. Au lancement de l'application, currentModel est initialisé à 0 pour pouvoir afficher la première couche. J'ai ensuite créé une fonction UpdateModelDisplay() qui met à jour l'affichage du modèle en activant la bonne couche et en désactivant les autres. Le changement de couche consiste donc à modifier la valeur de currentModel, puis d'appeler la fonction UpdateModelDisplay().

```
//display the next layer
public void GoDeeper() {
    ActivateModel(Global.currentModel + 1);
}

//display the previous layer
public void GoShallower() {
    ActivateModel(Global.currentModel - 1);
}

//activate model number i
public void ActivateModel(int i ) {
    //make sure i is valid
    if (i < 0 || i > Global.nbModels - 1)
        return;
    Global.currentModel = i;
    UpdateModelDisplay();
}

//activate the current model, deactivate the others
public void UpdateModelDisplay() {
    for(int i=0; i<Global.nbModels; i++) {
        Transform child = transform.GetChild(i);
        child.gameObject.SetActive(child.GetSiblingIndex() ==
            Global.currentModel);
    }
}
```

La hiérarchie d'objets que j'ai implémentée (voir 3.2) facilite grandement l'activation et la désactivation des couches. En effet les objets Model_i contiennent l'ensemble des maillages composants chaque couche, et l'objet Models est parent de tous les objets Model_i. Il suffit donc à Models d'activer son n-ième enfant, où n correspond à la variable globale currentModel. Tous les autres enfants sont désactivés.

Interface

Les contrôles doivent être facilement utilisable sur ordinateur, mais également de manière tactile sur tablette. Pour les 4 premiers contrôles, j'ai décidé d'utiliser un système de boutons à bascule (toggle) ainsi qu'une variable globale currentState (état actuel). Cette variable peut prendre les valeurs Translating, Rotating, Zooming, MovingLight. L'utilisateur peut faire varier l'état actuel grâce aux toggles et ainsi choisir la transformation de son choix. La transformation sélectionnée est mise en évidence par la couleur de l'icône du bouton associé à cette transformation. Ce système d'états permet une utilisation simple des contrôles du modèle, à la fois sur ordinateur et sur tablette tactile.

Figure 24: Boutons à bascule avec la translation activée

Afin de faciliter l'usage des contrôles sur ordinateur, j'ai rajouté des raccourcis claviers et souris qui permettent de transformer le modèle de manière plus rapide et naturelle :

Raccourci utilisé	**Transformation engendrée**
Clic droit	Rotation
Défilement molette	Zoom
Clic molette	Translation
Ctrl + clic	Translation
Alt + clic	Rotation

Ces contrôles passent outre l'état de la variable currentState et sont pour la plupart utilisables sur tablette car celle-ci dispose de boutons Ctrl, Alt, et clic droit.

Concernant le déplacement entre les couches, j'ai utilisé 2 boutons qui permettent à l'utilisateur d'afficher la couche suivante et la couche précédente. J'ai également ajouté un élément texte qui correspond à l'index de la couche affichée sur le nombre total de couches. Ce texte est mis à jour automatiquement à chaque changement de couche et permet à l'utilisateur de se repérer dans les étapes de dissection.

Figure 25: Changement de couche

Code général

Les 4 premiers contrôles sont implémentés dans un script ClickBehavior qui appartient à l'objet Models, ce qui permet de transformer Models directement depuis le script. Unity facilite l'accès à la caméra et à la lumière donc ces transformations sont également simples à effectuer. ClickBehavior contient une fonction principale Update() qui est appelée à chaque image. Dans cette boucle, le script détecte les raccourcis clavier ou souris pour appeler la fonction associée, et si aucun raccourci n'est utilisé il appelle la fonction correspond à la variable currentState.

Le changement de couche est codé dans un script ModelsBehavior. Après avoir codé les fonctions GoDeeper() et GoShallower() présentées plus haut, il m'a suffit de lier ces fonctions aux boutons fléchés pour terminer l'implémentation de ce contrôle.

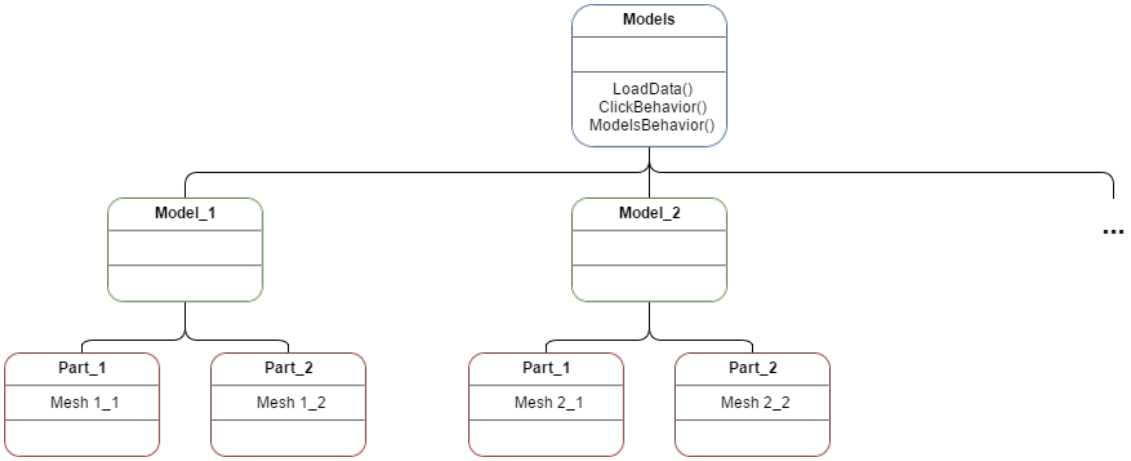


Figure 26: Hiérarchie des objets dans Unity, version 2

Ci-dessus la nouvelle hiérarchie des objets après l'implémentation des contrôle. La structure des objet n'a pas changé, seul l'objet Models a été modifié et contient maintenant 2 scripts supplémentaires.

3.4 Identification des structures anatomiques

La segmentation et l'identification des structures anatomiques devait être un mission assez anodine de mon stage mais elle s'est finalement révélée être beaucoup plus complexe que prévu. L'application professeur (d'identification des structures) a demandé beaucoup de temps et de recherche, ce qui m'a poussé à mettre de côté l'application de dissection et à repousser l'application d'anatomie. La segmentation des structures consiste à sélectionner une partie du maillage et à définir en tant que structure anatomique pour qu'on puisse connaître sa forme et sa position dans l'espace 3D. Il est impossible de commencer l'implémentation des questionnaires d'anatomie ou de la dissection avant d'avoir totalement terminé l'application professeur, puisque celles-ci vont se baser sur la structure de données qui sera utilisée pour stocker les éléments anatomiques.

3.4.1 Lancer de rayon (Raycast)

Pour pouvoir effectuer une sélection du maillage, il faut être capable de repérer un clic de l'utilisateur sur le modèle. Or ce dernier est en 3D, tandis que la souris se déplace dans un espace 2D (l'écran) : le curseur ne touche jamais vraiment le maillage, il est simplement affiché par dessus celui-ci. Afin de déterminer le point du maillage qui se trouve sous le curseur, on utilise une technique appelée lancer de rayon.
Cette technique consiste à tirer un rayon invisible à partir d'un point et vers une direction donnée, afin de détecter la présence d'objet sur le chemin de ce rayon. En tirant un rayon depuis la caméra et en passant par le curseur de la souris, on peut déterminer le point d'impact et donc le point du maillage qui est survolé par le curseur au moment du clic (ou l'absence de point).

Unity propose cette fonctionnalité de raycast et si le rayon détecte un objet on peut facilement récupérer le point d'impact, l'objet touché, ou encore la face de maillage touchée.

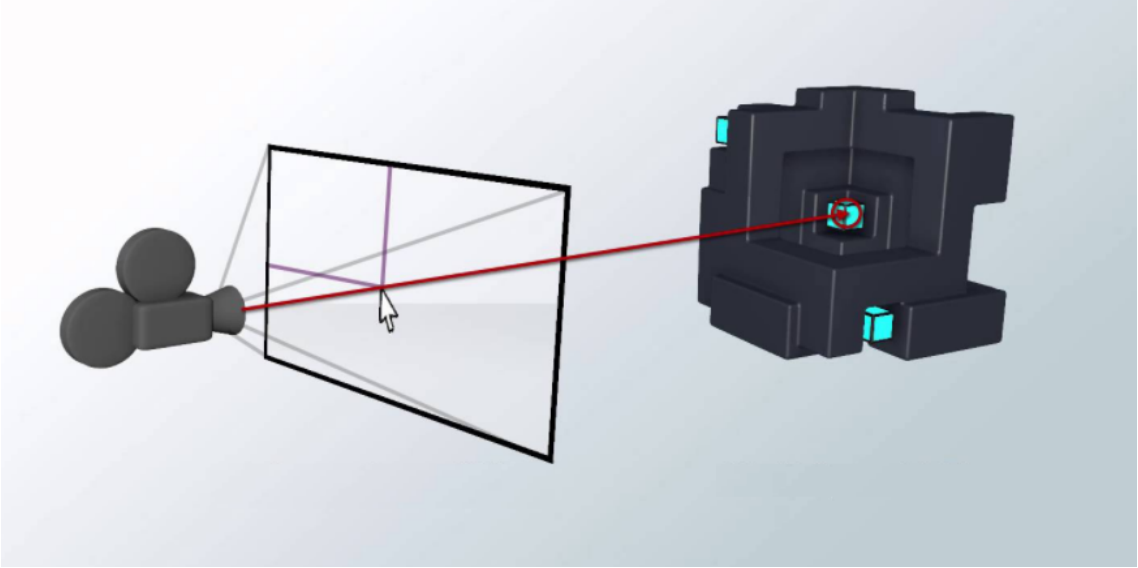


Figure 27: Schéma de rayon lancé depuis la caméra et passant par le curseur

3.4.2 Interface

La création de structures anatomiques demande un travail considérable au niveau de l'interface. L'utilisateur doit disposer de plusieurs fonctionnalités telles que créer une structure, la sélectionner, la supprimer, et la renommer. Il est également important qu'il ne soit pas limité par le nombre de structures qu'il peut créer, ni par la longueur de leur nom qui peut dans certains cas être très long.

Liste déroulante

J'ai commencé par créer une liste déroulante pour accueillir le nom des structures anatomiques. Lorsque l'utilisateur créé un nouvel élément anatomique, celui-ci est ajouté à la liste déroulante sous la forme d'un toggle et d'un label. Le toggle permet à l'utilisateur de sélectionner la structure, tandis que le label indique le nom de la structure. Si la liste contient trop d'éléments pour être affichés en une seule fois, une barre de défilement vertical apparaît à droite de la liste pour permettre à l'utilisateur de naviguer entre les différentes structures. Grâce à ce système, le professeur peut créer un nombre infini de structures sans être limité par l'interface. Le nom des structures est affiché sur 2 lignes, ce qui permet de nommer les structures de manière très détaillée (figure 20).
J'ai également implémenté un interrupteur permettant de cacher/montrer la liste déroulante pour offrir la possibilité de manipuler le modèle sur la totalité de l'écran. Ce dernier se trouve en haut à droite de la liste déroulante.

Champ de saisie

Par défaut les structures sont nommées "Unnamed". Lorsque le professeur créé ou sélectionne une structure, un champ de saisie apparait au dessus de la liste des structures. Ce champ contient le nom actuel de la structure sélectionnée, qui peut être modifié pour renommer la structure.

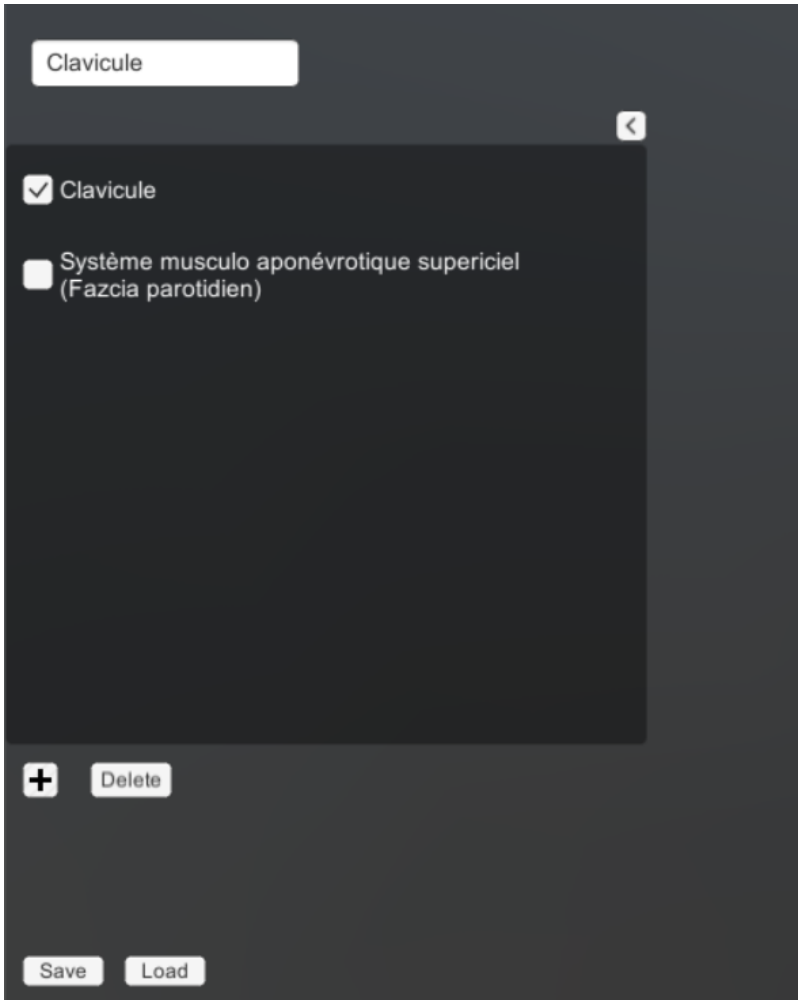


Figure 28: Interface contenant la liste des structures anatomiques

Boutons

J'ai enfin créé 4 boutons qui permettent respectivement de créer une nouvelle structure, supprimer une structure sélectionnée, sauvegarder l'ensemble des structures dans des fichiers texte, et charger les fichiers textes contenant les structures. Les fonctions appelées par ces boutons ont du être implémentées une première fois pour supporter le modèle de structures sphériques, puis une deuxième fois pour la deuxième représentation des structures.

3.4.3 Segmentation par sphères

Principe

Le premier modèle de structure que j'ai codé utilisait un centre et un rayon pour définir une structure anatomique. Bien entendu une représentation sphérique des éléments anatomiques est loin d'être idéale ; cette première implémentation était un prototype qui a servit de base pour pouvoir ensuite créer une structure de données plus complexe, disposant d'un outil de sélection plus précis.

J'ai commencé par créer un nouvel état correspondant à la création de structures. Lorsque le professeur clique sur le bouton de création de structure, il rentre dans cet état ce qui lui permet de cliquer sur modèle pour définir la position de la structure. En utilisant la technique de lancer de rayon, je récupère le point du maillage qui se trouve sous le curseur au moment du clic. Je créé alors un nouvel objet 3D sphérique en ce point, auquel j'assigne un matériel transparent. Un curseur apparait alors en bas de l'écran, permettant au professeur de modifier la taille de la sphère.

Hiérarchie

Pour stocker les structures sur Unity, j'ai créé pour chaque couche un nouvel objet appelé Structures qui est le parent de toutes les sphères. Il contient également un script Behavior() qui gère la création, la suppression et la sélection de structures.

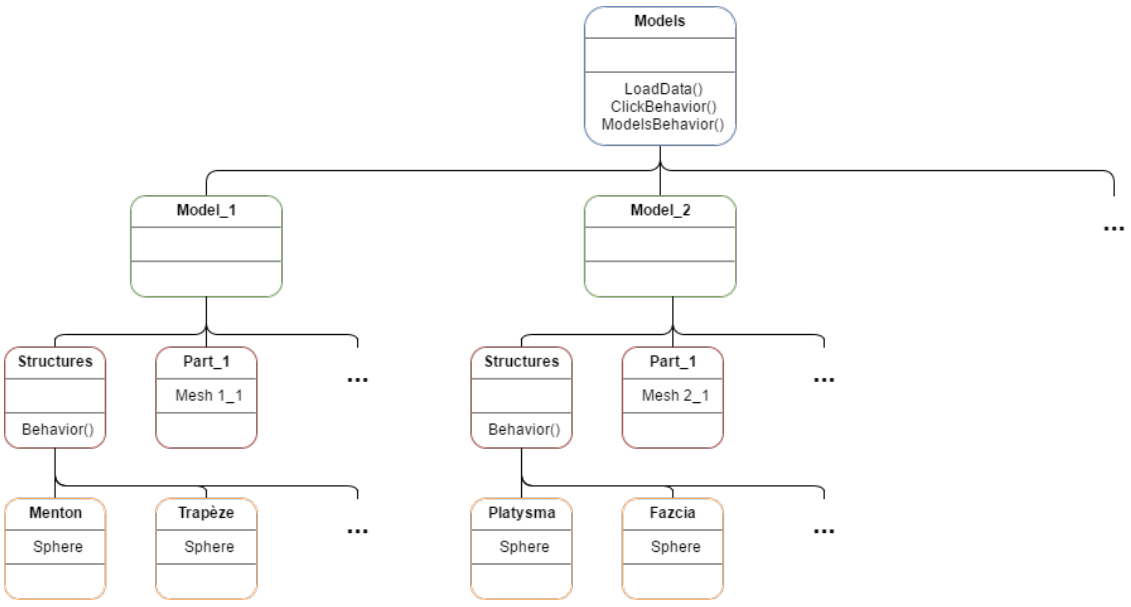


Figure 29: Hiérarchie des objets dans Unity, version 3

Détails

J'ai défini deux matériaux transparent de couleurs différentes afin de différencier la structure anatomique sélectionnée par le professeur des autres structures. Lorsque le professeur sélectionne une nouvelle structure, j'assigne la couleur classique à l'ancienne structure sélectionnée et la couleur de sélection à la nouvelle structure sélectionnée. Puis en utilisant un raycast j'ai implémenté la détection de clic sur la sphère sélectionnée afin de pouvoir déplacer la structure sur le modèle. Pour cela j'effectue un lancer de rayon tant que l'utilisateur garde le clic enfoncé, ce qui permet de calculer la nouvelle position de la sphère. J'ai également du prendre en compte l'écart entre la position du curseur et le centre de la sphère, car l'utilisateur ne clique pas forcément au centre de la structure : cet écart doit être conservé lorsqu'on déplace la sphère.

Exportation et importation

Une fois la création de structures terminée, il a fallu trouver un moyen de les stocker. Pour cela j'ai créé un fichier texte par couche de dissection. Les fichiers texte contiennent la liste des structures présentes sur cette couche, avec pour chaque structure son nom, sa position (x, y, z), et son rayon. Ci-dessous un exemple de fichier texte contenant 3 structures anatomiques.

Nom	x	y	z	r
Oeil droit	142.2	39.7	-37.1	9.6
Oeil gauche	142.2	39.7	-37.1	9.6
Bouche	132.9	32.1	-12.0	25

```
struct Structure {

    public Vector3 center;
    public float radius;
    public string name;

    public Structure(Vector3 _center, float _radius, string _name){
        center = _center;
        radius = _radius;
        name = _name;
    }

    //  returns true if the point is in the structure
    public bool isIn(Vector3 point){
        return Vector3.Distance (point, center) <= radius;
    }

}
```

Pour l'importation des données, j'ai codé la structure de données ci-dessus, qui contient un nom, un centre et un rayon. Cette structure est également dotée d'un constructeur et d'une fonction booléenne qui détermine si un point donné se trouve dans la sphère.

Pour importer les éléments anatomiques, il suffit alors de lire les fichiers texte ligne par ligne, de diviser la ligne en 5 parties (nom, x, y, z, rayon) puis de créer une nouvelle structure anatomique à partir de ces données.

Conclusion

De manière générale, la représentation sphérique fonctionne très bien. Grâce à la méthode isIn() des structures anatomiques et à un lancer de rayon, il est très simple de déterminer si un élève a cliqué sur un élément anatomique ou non. Le positionnement des sphères se fait de manière naturelle et l'interface est très réactive aux besoins de l'utilisateur.

Figure 30: Prototype de l'application professeur avec représentation sphérique des structures anatomiques

Je n'ai pas poussé la représentation sphérique plus loin car elle ne servait que de prototype à l'implémentation suivante, et elle n'est pas présente sur l'application finale. En effet cette technique n'est pas assez précise pour segmenter des structures très fines et très allongées comme par exemple le nerf facial (figure 31). Elle m'a cependant permis de mettre en place l'interface et de me familiariser avec certains outils comme les raycast ou la lecture et l'écriture de fichiers texte avec Unity.

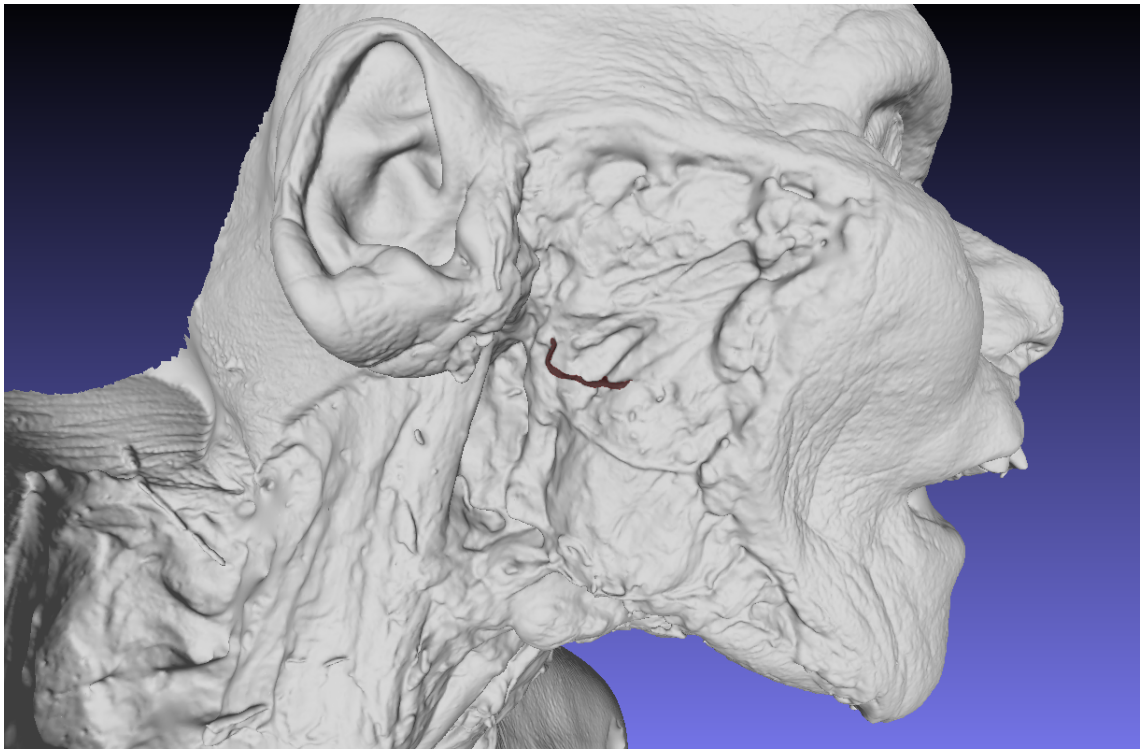


Figure 31: Nerf facial : exemple de structure qui demande une segmentation très précise

3.4.4 Segmentation par peinture

Principe

L'idée de la deuxième méthode de segmentation est de développer un outil de de sélection par peinture. Avec cet outil, le professeur doit pouvoir peindre des zones sur le modèle et les assigner à des structures anatomiques. Ces zones sont définies par les faces du maillage, il est donc essentiel de bien comprendre la structure de données des maillages utilisée par Unity.

Structure des maillages sur Unity

Un maillage contient des points (ou sommets) et des cellules (ou faces). Dans la grande majorité des cas, les cellules sont de forme triangulaire et relient 3 points entre eux. Il existe de multiples façon de stocker un maillage, certaines favorisant la taille mémoire et d'autres les performances. La méthode de stockage dont se sert Unity consiste à utiliser deux tableaux qui contiennent respectivement les sommets et les faces.

Les sommets sont de type Vector3, c'est à dire un triplet de coordonnées (x, y, z) servant à représenter un point dans l'espace 3D. Le tableau de sommets est donc un tableau de Vector3 qui contient tous les points du maillage, et dont la taille ne peut pas dépasser 65 000. La position d'un point dans le tableau des sommets est appelé "index".

Les faces sont définies comme un triplet d'index de sommets. La tableau des faces est donc un tableau d'entiers dont la taille est égale à 3 fois le nombre de triangles.

L'exemple ci-dessous utilise cette structure de données pour représenter un cube. Le maillage est composé de 8 sommets et de 12 faces (6 faces carrées divisées en 2 triangles chacune). Le tableau des faces est en réalité un tableau contenant 36 entiers, mais pour des raisons de clarté il est ici représenté sous forme de matrice de taille 3 x 12.

Index	**Sommet**
0	(0, 0, 1)
1	(1, 0, 1)
2	(0, 1, 1)
3	(1, 1, 1)
4	(0, 0, 0)
5	(1, 0, 0)
6	(0, 1, 0)
7	(1, 1, 0)

Tableau contenant les 8 sommets du cube

Index	**Sommet 1**	**Sommet 2**	**Sommet 3**
0	0	1	2
1	1	3	2
2	2	3	7
3	2	7	6
4	1	7	3
5	1	5	7
6	6	7	4
7	7	5	4
8	0	4	1
9	1	4	5
10	2	6	4
11	0	2	4

Tableau contenant les 12 faces du cube

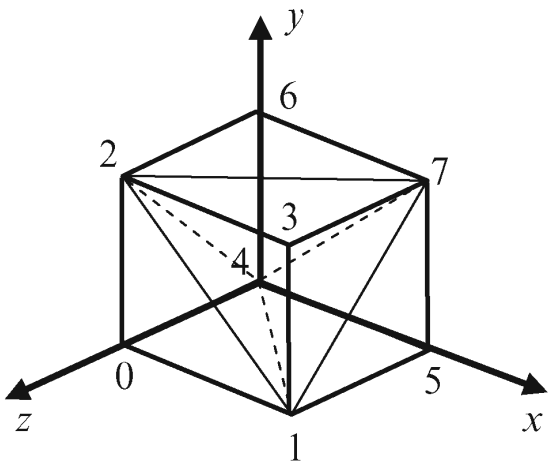


Figure 32: Cube formé par le maillage triangulaire ci-dessus

Cette structure de données est la plus simple et la plus légère qui existe, mais elle est peu adaptée pour certaines opérations comme par exemple la recherche de faces voisines.

Détails

Pour peindre le maillage, on réutilise la technique de lancer de rayon qui passe par le curseur de la souris. Si ce rayon touche le modèle, Unity nous donne les coordonnées de l'impact, mais également l'index du triangle qui a été touché. A partir de cet index on peut retrouver les trois sommets qui forment le triangle, et il on peut alors peindre cette face.

Hiérarchie

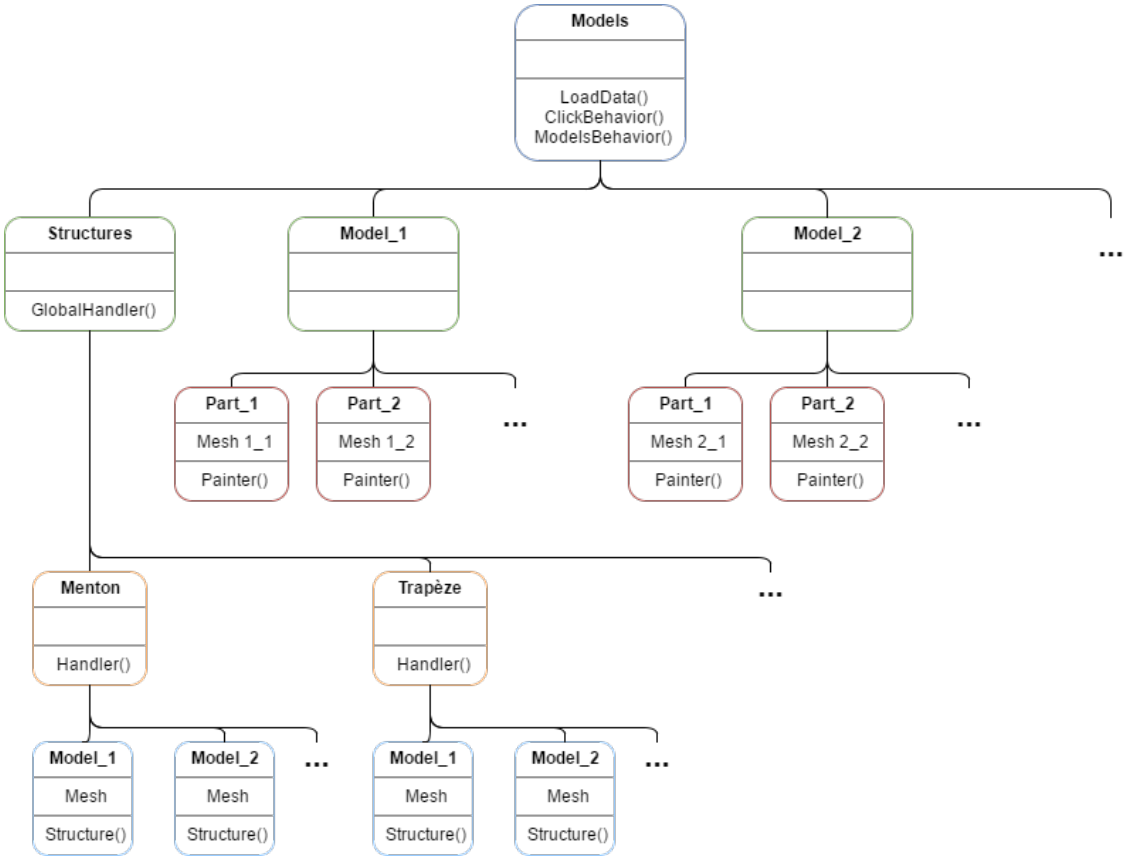


Figure 33: Hiérarchie des objets dans Unity, version 4

Dans un premier temps j'ai utilisé la même hiérarchie d'objets que pour la représentation sphérique (figure 29). Puis je me suis rendu compte qu'elle rendait difficile certaines opérations, par exemple la suppression de structure anatomique qui nécessite d'activer les couches de dissection une par une, ou le fait de garder une structure sélectionnée d'une couche à l'autre.

Pour ces raisons j'ai choisi de totalement restructurer la hiérarchie des structures anatomiques que j'utilisais jusqu'à maintenant, ce qui a demandé d'adapter une grande partie des fonctions écrites auparavant. Tous les éléments anatomiques appartiennent maintenant à un même objet Structures, ce qui facilite grandement l'importation et l'exportation des struc-

tures anatomiques. Ces structures ont autant d'enfants qu'il existe de couches de dissection car certains éléments anatomiques sont amenés à bouger ou disparaître entre les couches.

Trois scripts permettent de gérer le comportement des structures anatomiques. Le premier script, GlobalStructuresHandler() appartient à l'objet Structures. Il contient les fonctions servant à créer, sélectionner, et supprimer une structure anatomique, ainsi que l'importation et l'exportation des fichiers texte contenant les structures. Lorsqu'il créé une nouvelle structure, il lui donne le composant StructureHandler(). Ce deuxième script sert à créé les objets sous-jacents (Model_1, Model_2, ...) et à gérer leur affichage (afficher le bon enfant selon la couche actuelle), mais aussi à renommer la structure. En créant les objets enfants il leurs affecte à son tour le script Structure(). Ce dernier script sert à créer un maillage sur l'objet auquel il appartient, puis à gérer ce maillage. Il permet notamment d'ajouter un sommet, ajouter un triangle, supprimer un triangle, et rafraîchir l'affichage du maillage après avoir modifié sa topologie.

Stockage des structures

Ma première idée pour le stockage des structures fut d'exporter le centre des faces qui composent chaque structure. L'avantage de cette méthode est qu'elle est purement géométrique et ne s'appuie pas sur la topologie du maillage : il est alors possible de simplifier le maillage sans rendre invalide la position des structures anatomiques. Pour vérifier si l'utilisateur cliquait sur un structure, j'effectuais un test de distance pour chaque face de chaque structure. Cette opération était très coûteuse et je me suis vite rendu compte que cette implémentation était loin d'être idéale.

La deuxième idée fut d'utiliser les index des triangles pour assigner une structure à chaque face du maillage (ou une absence de structure). Cette méthode est dépendante de la topologie du maillage et n'est donc pas résistante à une simplification du modèle, mais elle offre de nombreux avantages en terme de consistance, de performances et de précision. L'exportation consiste à créer un premier fichier texte contenant la liste de toutes les structures anatomiques identifiées par le professeur (figure 34), puis à créer pour chaque sous-maillage un fichier référençant le premier (figure 35). Les fichiers référence contiennent des nombres entiers qui correspondent aux index des éléments anatomiques dans la liste du premier fichier. Le nombre d'index que contient un fichier est égal au nombre de triangles dont dispose le sous-maillage associé à ce fichier. Ainsi chaque triangle de chaque sous-maillage de chaque couche de dissection se voit affecter une structure anatomique. Si une face n'a pas été peinte, le nombre -1 lui est associé ce qui indique l'absence de structure anatomique sur cette face.

```
1  Mandibule
2  Menton
3  Trapèze
4  Clavicule
5  Trachée
6  Cartilage cricoïde
7  Incisure sternale
8  Os hyoïde
9  Platysma coli
10 ...
```

Figure 34: Fichier contenant la liste des structures anatomiques

```
1  -1
2  -1
3  -1
4  3
5  3
6  4
7  2
8  -1
9  0
10 -1
11 ...
```

Figure 35: Fichier référençant les structures anatomiques

Feedback visuel

Une fois que la face a été associée à une structure anatomique, il est important d'indiquer à l'utilisateur que ce triangle a bien été traité. Une solution évidente est de colorier la face avec une couleur transparente, ce qui permet au professeur de voir les zones peintes sans pour autant masquer les éléments anatomiques. Cependant Unity ne permet pas de modifier la couleur d'une face : l'ensemble du maillage partage le même matériel, et donc la même texture ou la même couleur.
Pour obtenir ce feedback visuel il faut créer un nouveau maillage et copier les faces cliquées par l'utilisateur vers ce nouveau maillage. De cette manière chaque structure anatomique a son propre maillage qu'ont peut colorier comme on le souhaite.

Cette manipulation de duplication du maillage cause cependant un problème de rendu. Comme les deux maillages se trouvent au même endroit, Unity ne sait pas lequel afficher par dessus l'autre ce qui cause un entrelacement des deux maillages (figure 36). Ce phénomène est accentué par le fait que la position des sommets est parfois arrondie en les dupliquant vers un autre objet.

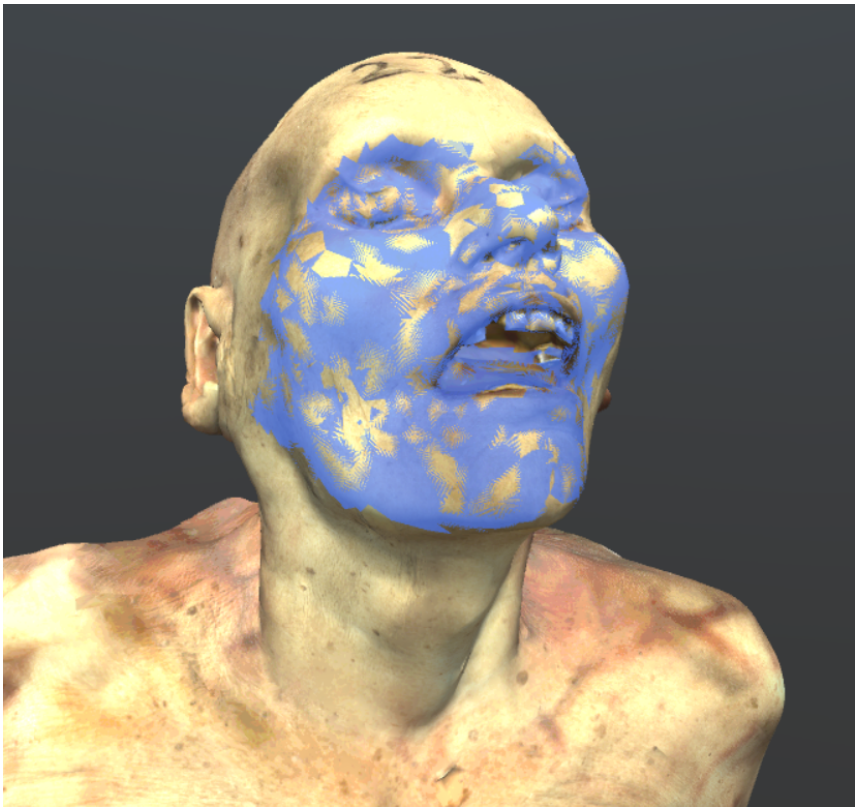


Figure 36: Problème visuel d'entrelacement des maillages

Il est difficile de résoudre ce problème de manière géométrique car le "devant" du modèle dépend de sa rotation et de la position de la caméra. Une translation du maillage coloré pourrait supprimer le phénomène au niveau du visage, mais il serait intensifié sur les épaules, les oreilles et la nuque. De la même manière, un changement d'échelle du maillage résoudrait le problème sur le front mais l'accentuerait sur le menton.
Une solution possible est de placer le maillage coloré entre le modèle et la caméra, et d'actualiser sa position à chaque mouvement du modèle. Cependant cette technique de calcul en temps réel diminue les performances de l'application, et elle ne fonctionne pas toujours sur les éléments les plus détaillés (oreilles, bouche).

Ce problème a finalement été mis de côté pendant plusieurs semaines, puis il a été résolu grâce à une bibliothèque gratuite appelée "Outline Effect" disponible sur l'Asset Store d'Unity. Cette bibliothèque permet de mettre en évidence un maillage grâce à un contour et une coloration, même si celui-ci se trouve derrière d'autres objets. (figure 37).

Figure 37: Mise en évidence de la clavicule grâce à une duplication du maillage et à l'asset Outline Effect

Gomme

La gomme fonctionne de la même manière que le pinceau : par lancer de rayon. Une fois qu'on récupère l'index de la face, au lieu de lui assigner une structure anatomique et lui appliquer un feedback visuel, on désassigne la structure et supprime le feedback.

Comme expliqué plus haut, chaque face est assignée à un indice de structure anatomique, et un indice -1 correspond à une absence de structure anatomique en cette face. Pour désassocier la structure à la face il suffit donc de lui réassigner -1.

Pour supprimer le feedback visuel il faut récupérer le maillage correspondant à la structure actuelle, et supprimer la face qu'on veut gommer. La suppression de face consiste à supprimer le triplet d'index de sommets du tableau des faces, puis à actualiser l'affichage du maillage.

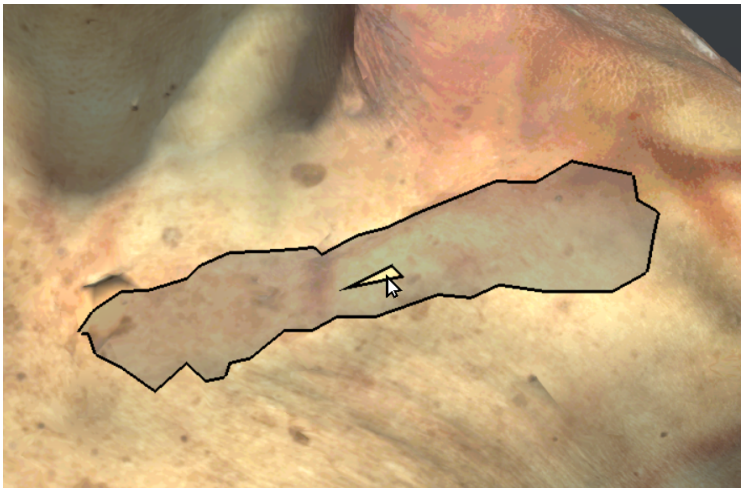


Figure 38: Utilisation de la gomme sur la clavicule

Élargissement du pinceau

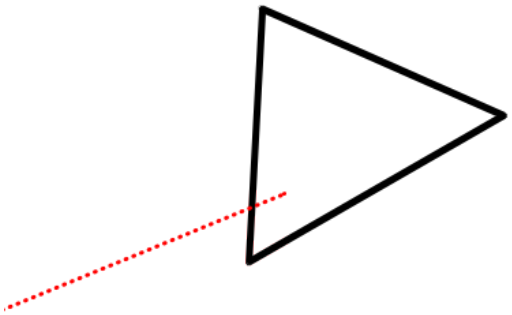
Cette méthode de peinture/gommage fonctionne très bien, mais elle prend beaucoup de temps car il faut passer la souris sur tous les triangles qui composent une structure anatomique. Pour accélérer l'identification des structures il est essentiel de pouvoir élargir la zone de peinture ou de gommage.

L'élargissement du pinceau consiste à rechercher les faces voisines de celle qui reçoit le rayon, ce qui se fait généralement en utilisant la topologie du maillage. Cependant la structure de données des maillages utilisée par Unity (voir page 34) ne facilite pas cette opération. En effet elle oblige à comparer toutes les faces deux à deux entres elles ce qui est très coûteux en terme de ressources. De plus, le modèle a été divisé en plusieurs sous-maillages donc les voisins d'un triangle peuvent se trouver sur un autre maillage que le sien. Pour ces raisons, et compte tenu de la taille des données utilisées, il n'était pas envisageable de faire une recherche de voisins en temps réel en utilisant la topologie des maillages.

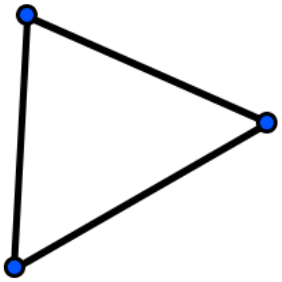
La première technique implémentée fut de réaliser plusieurs lancers de rayon dans un quadrillage pour ainsi récupérer plusieurs indices de triangles. Cette technique a l'avantage d'être très facile à coder puisqu'il suffit de relancer plusieurs rayons vers la souris mais avec à chaque fois un décalage sur les axes X et Y de l'écran. Un des problèmes de cette méthode est de déterminer ce décalage entre les rayons car dû à la simplification partielle des maillages, la taille des triangles varie énormément. Si le décalage est trop faible plusieurs rayons peuvent atterrir sur la même face et la méthode n'est pas efficace, et si le décalage est trop élevé certaines faces ne seront pas traitées et on aura un effet de peinceau clairsemé. Cette méthode n'est pas assez consistante pour pouvoir être utilisée, mais elle montre qu'il est possible d'effectuer plusieurs raycast ce qui a ensuite inspiré une deuxième méthode de recherche de voisins.

La deuxième technique consiste toujours à lancer plusieurs rayons, mais cette fois de manière optimale pour être certain de toucher un triangle voisin. Pour cela on procède de la manière suivante :

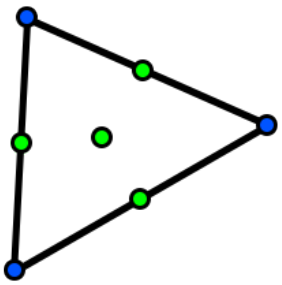
On lance un premier rayon passant par le curseur de la souris et on récupère l'index du triangle touché



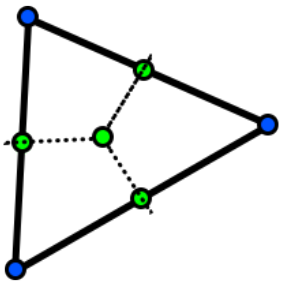
A partir de l'index de la face on trouve la position de ses 3 sommets



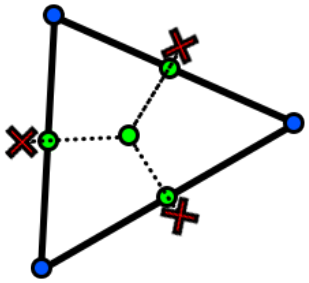
On peut alors calculer la position du centre des segments et du centre de la face



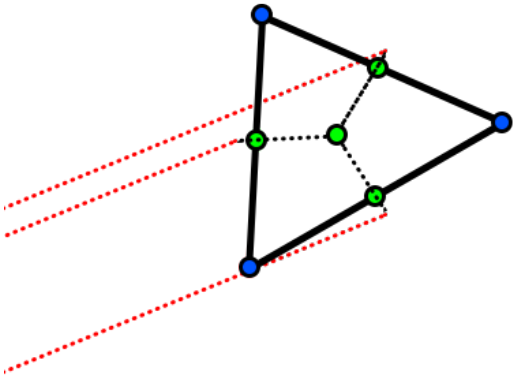
En décalant le centre des segments par rapport au centre de la face...



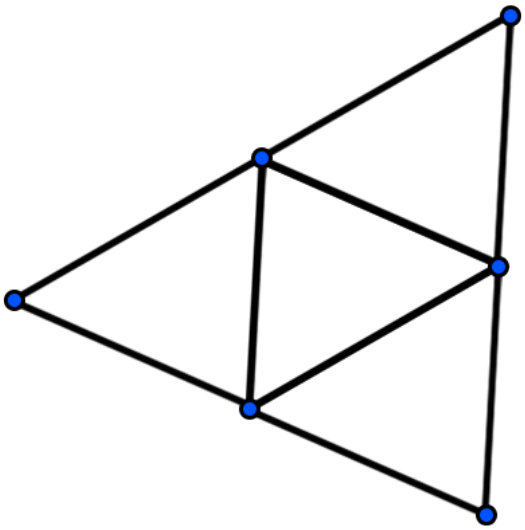
On trouve trois points qui appartiennent aux faces voisines...



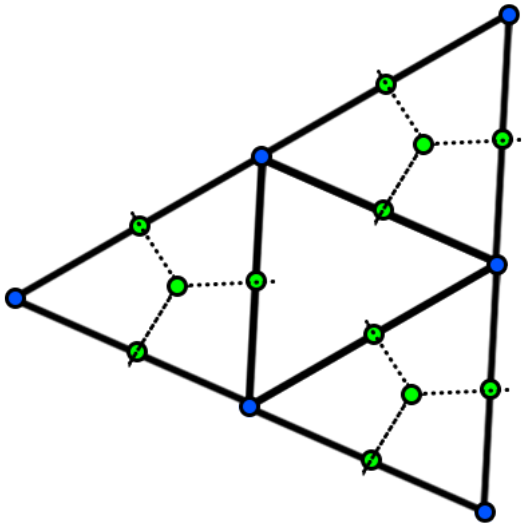
Et vers lesquels on peut lancer des rayon



Pour ainsi récupérer les index des faces voisines



On peut ensuite réitérer l'opération sur ces faces pour continuer à élargir la zone de peinture ou de gommage



Cette technique permet de trouver les voisins d'une face en temps réel et la taille de la zone peut être définie par le nombre d'itérations que l'on effectue. Ce nombre d'itérations est défini par un curseur qui se trouve en bas de l'écran. Lorsque le curseur est à son minimum, le pinceau n'affecte que la face touchée, en déplaçant le curseur la pinceau peut affecter jusqu'à 30 degrés de voisins de la face touchée. L'identification des structures anatomiques de grande taille est alors grandement simplifiée.

Figure 39: Peinture de base

Figure 40: Peinture avec élargissement de la zone

Code

La peinture de maillage utilise principalement deux fonctions : ReceiveRaycast() et ColorFace(). La première sert à recevoir le lancer de rayon, puis elle appelle ColorFace() et enfin elle actualise l'affichage du maillage pour faire apparaître la peinture.

```
void ReceiveRaycast(RaycastHit hit) {

    //  get current structure
    Structure currentStructure =
       GlobalStructuresHandler.instance.GetSelectedStructure();
    if (currentStructure == null)
        return;

    //  get triangle hit
    int tIndex = hit.triangleIndex;
    if (clickedTris.Contains(tIndex))
        return;

    //  start recursion
    ColorFace(currentStructure, tIndex, 0);

    //  clear list of treated triangles
    currentTreatedTris.Clear();

    //  refresh mesh renderer
    currentStructure.RefreshMesh();

}
```

ColorFace() effectue la technique de recherche de voisins expliquée précédemment, associe la structure anatomique actuelle aux triangles peints puis duplique les faces pour réaliser le feedback visuel.

```
public void ColorFace( Structure currentStructure, int tIndex, int
    loop) {

    //  if triangle already processed, return
    if (currentTreatedTris.Contains(tIndex))
        return;
    //  add to processed triangles
    currentTreatedTris.Add(tIndex);

    //  get the 3 vertices positions
    Vector3 vert1 = myMesh.vertices[indices[tIndex * 3]];
    Vector3 vert2 = myMesh.vertices[indices[tIndex * 3 + 1]];
    Vector3 vert3 = myMesh.vertices[indices[tIndex * 3 + 2]];

    //  if current triangle has no structure
    if (structuresTab[tIndex] == -1) {
        //  set to current structure
        structuresTab[tIndex] =
            GlobalStructuresHandler.instance.GetSelectedStructureIndex();
        //  add triangle to current structure's mesh (color
            feedback)
        currentStructure.AddTriangle(vert1, vert2, vert3);
    }

    //  check for recursion
    if (loop >= Global.searchDepth) return;

    //  switch to world space
    vert1 = transform.TransformPoint(vert1);
    vert2 = transform.TransformPoint(vert2);
    vert3 = transform.TransformPoint(vert3);

    //  compute centers of segments + offset
    Vector3[] centers = new Vector3[3];
    float offset = 0.05f;
    centers[0] = (1 + offset) * 0.5f * (vert1 + vert2) - offset *
        vert3;
    centers[1] = (1 + offset) * 0.5f * (vert2 + vert3) - offset *
        vert1;
    centers[2] = (1 + offset) * 0.5f * (vert3 + vert1) - offset *
        vert2;
```

```
    Vector3 origin = Camera.main.transform.position;

    //  for each center, launch a raycast and recurse with hit
        triangle
    for (int i=0; i<3; i++) {
        Vector3 goal = centers[i];
        Vector3 direction = goal - origin;
        RaycastHit hit;
        if (Physics.Raycast(origin, direction, out hit, 1000.0f,
            modelsLayerMask)) {
            TrianglesPainter tRemover =
                hit.collider.GetComponent<TrianglesPainter>();
            if(tRemover != null)
                tRemover.ColorFace(currentStructure,
                    hit.triangleIndex, loop + 1);
        }
    }
}
```

4 Conclusion

4.1 Résultats obtenus

Application professeur

Au moment de la rédaction de ce rapport, l'application d'identification des structures est fonctionnelle et permet d'identifier, de segmenter et d'exporter les structures anatomiques. Quand nous avons remarqué que le problème des structures anatomiques était complexe et prendrait beaucoup plus de temps que prévu, il était évident que toutes les missions du stage ne seraient pas réalisables dans le temps imparti. Il a alors été décidé que je concentrerais mon travail sur l'application d'identification des structures, afin que celle-ci soit solide et puisse servir de base à d'autres personnes qui voudraient poursuivre ce projet. L'application peut charger un maillage d'anatomie (humaine ou non) peut importe sa taille, et l'afficher à l'écran. Le modèle peut être contrôlé de différentes façons : translation, rotation, zoom et contrôle de la lumière. Si plusieurs couches ont été chargées il est également possible de circuler librement entre elles. L'utilisateur peut alors créer autant de structure anatomiques qu'il le souhaite, et peindre le modèle avec une précision proportionnelle à la résolution du maillage. Il est ensuite possible d'exporter les données sous forme de fichiers texte. Pour modifier des structures déjà définies, il suffit d'importer les fichiers texte, faire les changements nécessaires, puis exporter pour écraser les fichiers précédents.

Application de dissection

L'application de dissection est très particulière et demande beaucoup de recherche et de travail pour pouvoir fonctionner correctement. L'utilisation de la tablette tactile apporte des contraintes supplémentaires pour l'interface, les contrôles, mais également les performances de l'application. La superposition des couches n'est pas facile à réaliser car malgré l'attention des chirurgiens, le cadavre est voué à bouger pendant la dissection. A cause de cela il est impossible de complètement cacher une couche sous une autre si on ne traite pas les maillages auparavant. Une solution possible est de découper les maillages pour ne garder que les nouveaux éléments anatomiques de la couche (il n'est pas nécessaire d'avoir le visage en 8 exemplaires). Ce découpage pourrait se faire de manière automatique en calculant la distance entre une couche et sa couche supérieure : si cette distance dépasse un certain seuil on peut estimer qu'il s'agit d'une nouvelle structure anatomique. Les maillages devraient ensuite être recalés pour ne pas laissé d'écart entre les différentes couches.

Application d'anatomie

L'application d'anatomie en elle même n'a pas encore été créée au moment de la rédaction du rapport, mais elle est très similaire à l'application professeur. Elle reprend en effet la majorité de ces fonctionnalités : chargement et affichage des maillages, contrôles des modèles, et chargement des structures anatomiques. Terminer cette application est une des priorités de mon dernier mois de stage.

4.2 Difficultés rencontrées

Principe unique

Ce projet est très innovant de par son principe, mais à cause de cela il est difficile de trouver des informations sur Internet pour résoudre les problèmes que j'ai rencontré.
Unity est un moteur qui sert principalement à créer des jeux vidéos, et aucun jeu n'utilise de maillage de plus de 65 000 faces, et encore moins de plusieurs millions de faces. La division en sous-maillages n'est donc pas un problème récurent chez les utilisateurs d'Unity.
Les opérations de maillage comme la sélection ou la coloration se font en temps normal en prétraitement sur des logiciels de rendus 3D (MeshLab, Blender) mais très peu de gens cherchent à implémenter ces outils sur Unity.
Pour ces raisons il était difficile de trouver des conseils sur Internet et la hiérarchie des objets ainsi que les outils de sélection et de peinture de maillage ont du être implémenté sans aucune aide extérieure.

Taille des données

La taille très importante des données a causé une multitude de problèmes tout au long de mon stage. La division des gros maillages en plusieurs sous-maillages n'a pas d'influence sur le visuel du modèle, mais elle complique grandement le chargement des données et la recherche de faces voisines. Les performances sont aussi grandement affectées puisque l'application d'identification des structures met prêt d'une minute à démarrer simplement

pour charger les maillages. Il faut ensuite charger les structures ce qui peut doubler le temps de chargement (selon le nombre de structures identifiées). Enfin selon la puissance de l'ordinateur qu'on utilise et la taille du pinceau, la peinture des structures n'est pas toujours fluide car le nombre de calculs pour trouver la face touchée par le rayon augmente avec le nombre de faces du modèle.

Sélection de maillage

La sélection de maillage est un outil qui peut paraître simple au premier abord. En effet les logiciels de rendu 3D proposent une sélection par peinture qui est très fluide et naturelle, peu importe la taille du maillage. Cependant ces logiciels utilisent une structure de maillage complexe, lourde en mémoire, mais adaptée à ce genre de problématique. Unity utilise une structure de données très basique et ne dispose pas assez d'outils pour gérer les maillages, ce qui rend très difficile l'implémentation de cet outil.

Peinture de maillage

La peinture pour le feedback visuel a également été plus compliquée que prévu car il est impossible de modifier la couleur d'une face d'un maillage. Il est possible d'implémenter une peinture de texture en changeant la valeur des pixels, mais elle ne fonctionnerait pas avec nos données car les morceaux de textures sont trop proches les uns des autres sur l'image PNG (figure 11). Ce problème a finalement été résolu en dupliquant le maillage et en coloriant la copie, mais cela diminue encore les performances en rajoutant des objets dans la scène, et a causé le problème visuel d'entrelacement des maillages (figure 36), ensuite résolu grâce à une bibliothèque de l'Asset Store.

4.3 Missions restantes et perspectives d'amélioration

A l'écriture de ce rapport il me reste encore 1 mois de stage à effectuer, pendant lequel je travaillerai sur les missions suivantes par ordre de priorité :

Identifier les structures anatomiques

Maintenant que l'application professeur est fonctionnelle, l'identification des structures anatomique est réalisable sur les modèles de la dissection des cervicales, ainsi que les modèles de la dissection du périnée. Cette identification sera effectuée avec la contribution de mon tuteur M Captier ou de M Akkari, afin que la position et la forme des éléments anatomiques soient aussi correctes que possible.

Hiérarchiser les structures anatomiques

A l'heure actuelle les éléments anatomiques sont sous forme de liste, et il n'existe aucune forme d'agencement entre eux. La fédération FIPAT (Federative International Programme for Anatomical Terminology) propose une hiérarchie internationale de tous les éléments du corps humain. L'incorporation de cette hiérarchie dans l'application serait utile pour

s'assurer qu'aucun élément anatomique n'ait été oublié pendant l'identification, et elle permettrait plus de contrôles comme par exemple l'affichage de tous les nerfs du modèle.

Application d'enseignement de l'anatomie

L'application d'anatomie réutilise la majorité du travail effectué sur l'application professeur. Pour terminer cette application il suffit d'implémenter les 3 modes définis dans le cahier des charges : découverte, évaluation, parcours.

Le mode découverte est le plus simple à implémenter puisqu'il consiste simplement à charger le modèle et la position des éléments anatomiques. Lorsque l'élève clique sur une face, on récupère son indice puis on regarde si une structure anatomique est associée à cette face. Si tel est le cas on affiche le nom de cette structure.

Pour le mode évaluation il faut implémenter deux types de questions. Le premier type consiste à nommer une structure de manière aléatoire, et demander à l'élève de cliquer dessus. Nous disposons d'une liste qui contient toutes les structures anatomiques identifiées et il est donc très facile d'en prélever une aléatoirement. Le second type de question consiste également à choisir une structure aléatoire, mais cette fois on la met en évidence (comme pour la sélection par peinture) et on propose à l'élève plusieurs noms d'éléments anatomiques parmi lesquels il doit choisir le bon.

Le mode parcours est le plus délicat puisqu'il doit être modulable par les professeurs. Il faut pouvoir lire un fichier texte qui contiendra plusieurs paramètres (structures importantes à connaître, temps imparti, score minimum pour passer à un niveau supérieur, ...) et adapter l'évaluation selon ces paramètres.

Projection des structures anatomiques

Certaines structures anatomiques sont présentes sur plusieurs couches, et il est dommage de devoir les peindre plusieurs fois alors que leur position ne varie que très peu entres les couches. Une amélioration possible de l'application professeur serait de pouvoir projeter une structure anatomique sur les couches inférieures. On pourrait alors peindre le menton sur la couche 1, puis le projeter jusqu'à la couche 8. La projection ne serait pas parfaite car le cadavre bouge légèrement pendant la dissection, et la topologie n'est pas la même entre les couches donc il serait possible d'obtenir des trous dans les structures projetées. Cet outil servirait tout de même à avoir une base qui puisse être retouchée avec le pinceau et la gomme et constituerait un gain de temps non-négligeable.

Application de dissection

L'application de dissection demande sûrement assez de travail pour être un sujet de stage à part entière. En discutant avec mes tuteurs nous avons en effet pu constater qu'elle demandait des implémentations très spécifiques comme le découpage automatique de maillage, la superposition des couches, l'imitation de l'élasticité de la peau, les différentes résistances des structures anatomiques au scalpel, ...

Au vu du travail requis pour cette application je n'aurai pas le temps de l'implémenter durant mon dernier mois de stage.

4.4 Apports

Durant ma formation de Master informatique j'ai eu l'occasion de découvrir le moteur de jeux Unity et j'ai apprécié sa puissance ainsi que ses fonctionnalités variées. Je suis ravi d'avoir réalisé un projet entier sur ce logiciel car cela m'a permis d'en apprendre beaucoup plus sur ses capacités mais également sur ses limites. A l'issu de ce stage je suis beaucoup plus expérimenté sur le fonctionnement de Unity, et je suis capable de déterminer si un projet est réalisable sur cet environnement ou non.
Ce stage a été pour moi l'occasion de découvrir le milieu médical notamment lorsque j'ai assisté à une dissection cadavérique couplée à une capture surfacique. Je suis maintenant beaucoup plus familier avec les techniques de modélisation médicales et j'ai conscience des limites des applications d'anatomie existantes.
Grâce aux réunions d'équipes du LIRMM j'ai pu apprécier le travail de mes collègues et renforcer mes capacités à soutenir devant un public puisque j'ai eu l'occasion de faire 2 présentations sur mon sujet de stage et son avancement.
Malgré le fait que le stage soit professionnel, il contenait également une quantité non négligeable de recherche. Expérimenter avec ces deux parcours m'a permis de me faire une meilleur idée de ce que je souhaitais pour mon futur professionnel.

5 Annexes

List of Figures

Ci-dessous les scripts principaux qui ont été écrits au cours de ce stage.

5.1 Script Global()

```
public enum State {
    Translating,
    Rotating,
    Scaling,
    Searching,
    Painting,
    Erasing,
    MovingLight
}

public class Global : MonoBehaviour {

    public static State currentState;
    public static int currentModel = 0;
    public static int nbModels;
    public static int searchDepth = 0;  //  number of iterations
        when painting

}
```

5.2 Script LoadData()

```
public class LoadData : MonoBehaviour {

    void Awake () {

        //  Path of the StreamingAssets folder
        string applicationPath = Application.streamingAssetsPath +
            "/";

        //**********   LOAD TEXTURES   **********
        //  List of all the textures
        List<Texture> myTextures = new List<Texture>();
        WWW w;
        DirectoryInfo dir1 = new DirectoryInfo (applicationPath);
        //  Get all .png files
        FileInfo[] myFiles = dir1.GetFiles ("*.png");
        //  For each of them
        foreach (FileInfo f in myFiles) {
            //  Load the file
            w = new WWW ("file:///" + applicationPath + f.Name);
            //  Wait until it's done
            while (!w.isDone) {
            }
            //  Add the texture to list
            myTextures.Add (w.texture);
        }

        //**********   LOAD MESHES   **********
        //  Nb of meshes = nb of textures
        int nbMeshes = myTextures.Count;
        Global.nbModels = nbMeshes;
        //  Load all the files in Resources/Meshes
        Object[] myMeshes = Resources.LoadAll ("Meshes");

        //**********   DISPLAY MODELS   **********
        //  Index to iterate the meshes files
        int fileIndex = 1;
        //  For each mesh
        for (int i = 0; i < nbMeshes; i++) {
            //  Create new material
            Material myMaterial = new Material (Shader.Find
                ("Diffuse"));
            //  Apply texture to new material
            myMaterial.SetTexture ("_MainTex",
                (Texture)myTextures[i]);
```

```
            //  Create child object "Model"
            GameObject myChild = new GameObject ("Model_" + (i +
                1));
            myChild.transform.parent = transform;

            bool isSubmesh = true;
            //  Index for
            int submeshIndex = 1;
            //  For each submesh
            while (fileIndex < myMeshes.Length && isSubmesh) {
                //  If file is a submesh
                if (myMeshes[fileIndex].name.Length > 6 &&
                    myMeshes [fileIndex].name.Substring (0,
                    7).Equals ("default")) {
                    //  Create child object
                    GameObject newObject = new GameObject ("Part_"
                        + submeshIndex);
                    newObject.transform.parent = myChild.transform;
                    newObject.layer = 9;
                    Mesh myMesh = myMeshes [fileIndex] as Mesh;
                    //  Add Mesh Filter
                    MeshFilter myMeshFilter =
                        newObject.AddComponent<MeshFilter> ();
                    myMeshFilter.mesh = myMesh;
                    //  Add Mesh Renderer
                    MeshRenderer myMeshRenderer =
                        newObject.AddComponent<MeshRenderer> ();
                    //  Add Mesh Collider
                    newObject.AddComponent<MeshCollider> ();
                    //  Apply material
                    myMeshRenderer.material = myMaterial;
                    //  Add TrianglesRemover
                    newObject.AddComponent<TrianglesPainter>();
                    submeshIndex++;
                } else {
                    isSubmesh = false;
                }
                fileIndex++;
            }
        }
    }
}
```

5.3 Script GlobalStructureHandler()

```
public class GlobalStructuresHandler : MonoBehaviour {

    public static GlobalStructuresHandler instance;

    public GameObject selectedStructure;
    public GameObject StructuresGameObject;

    public List<StructureHandler> Structures = new
       List<StructureHandler>();

    int nbStructures;

    void Awake() {
        //  singleton
        if (!instance) {
            instance = this;
        } else
            Destroy(gameObject);

        nbStructures = 0;
    }

    void Start() {
        //  Structures GameObject
        StructuresGameObject = new GameObject("Structures");
        StructuresGameObject.transform.parent = transform;
    }

    public void CreateStructure() {
        CreateStructure("Unnamed");
    }

    public void CreateStructure(string structureName) {

        InputFieldHandler.instance.gameObject.SetActive(true);
        InputFieldHandler.instance.ChangeText(structureName);

        GameObject newStructureGameObject = new
           GameObject(structureName);
        newStructureGameObject.transform.parent =
           StructuresGameObject.transform;
        newStructureGameObject.transform.localRotation =
           Quaternion.identity;
        newStructureGameObject.transform.localPosition =
           Vector3.zero;
```

```
        StructureHandler newStructure =
            newStructureGameObject.AddComponent<StructureHandler>();
        newStructure.structureIndex = nbStructures;
        Structures.Add(newStructure);

        nbStructures++;
        StartCoroutine(PrintStructuresEndFrame());
        SelectStructure(newStructureGameObject);
    }

    public void UpdateStructuresDisplay() {
        StructuresGameObject.BroadcastMessage("UpdateStructureDisplay");
        foreach(StructureHandler sh in Structures) {
            sh.SendMessage("Unselect");
        }
        if(HasStructureSelected())
            selectedStructure.SendMessage("Select");
    }

    public Structure GetSelectedStructure() {
        if (selectedStructure == null) return null;
        return
            selectedStructure.GetComponent<StructureHandler>().GetCurrentSubstructu
    }

    public int GetSelectedStructureIndex() {
        return
            selectedStructure.GetComponent<StructureHandler>().structureIndex;
    }

    public void SelectNthStructure(int i) {
        GameObject structureToSelect =
            StructuresGameObject.transform.GetChild(i).gameObject;
        SelectStructure(structureToSelect);
    }

    public StructureHandler GetStructure(int i ) {
        return Structures[i];
    }

    public void SelectStructure(GameObject structure) {
        if (HasStructureSelected())
            selectedStructure.SendMessage("Unselect");
        selectedStructure = structure;
        selectedStructure.SendMessage("Select");

        InputFieldHandler.instance.gameObject.SetActive(true);
```

```
        InputFieldHandler.instance.ChangeText(selectedStructure.name);
    }

    public void UnselectStructure() {
        if (HasStructureSelected()) {
            selectedStructure.SendMessage("Unselect");
            selectedStructure = null;
            MessagesHandler.instance.SetTogglesOff();
        }
        InputFieldHandler.instance.ChangeText("");
        InputFieldHandler.instance.gameObject.SetActive(false);
    }

    public void RenameStructure( string str ) {
        if (!HasStructureSelected())
            return;
        selectedStructure.name = str;
        selectedStructure.SendMessage("Rename", str);
        PrintStructures();
        StartCoroutine(SetToggleAfterDelay(selectedStructure.transform.GetSiblingI
    }

    bool HasStructureSelected() {
        return (selectedStructure != null);
    }

    public void DeleteStructure() {
        if (!HasStructureSelected()) return;
        BroadcastMessage("RemoveStructure",
           GetSelectedStructureIndex());
        Destroy(selectedStructure);
        StartCoroutine(PrintStructuresEndFrame());
        InputFieldHandler.instance.ChangeText("");
        InputFieldHandler.instance.gameObject.SetActive(false);
        MessagesHandler.instance.SetTogglesOffDelayed();
    }


    public void SaveStructuresFiles() {

        int tempActivatedModel = Global.currentModel;
        for(int i=0; i < transform.childCount - 1 ; i++) {
            GetComponent<ModelsBehavior>().ActivateModel(i);
            BroadcastMessage("ExportFile");
        }

        GetComponent<ModelsBehavior>().ActivateModel(tempActivatedModel);
```

```
    //  Structures list file
    string fileName = "Structures_list.txt";
    if (File.Exists(fileName)) {
        File.Delete(fileName);
    }
    StreamWriter sr = File.CreateText(fileName);
    string temp = "";
    foreach(StructureHandler str in Structures) {
        temp += str.structureName + ';';
    }
    sr.WriteLine(temp);
    sr.Close();
}

public void LoadStructuresFiles() {

    //  Load structures list
    string fileName = "Structures_list.txt";
    if (File.Exists(fileName)) {
        StreamReader sr = new
           StreamReader(Application.dataPath + "/../" +
           fileName);
        string fileContent = sr.ReadToEnd();
        sr.Close();
        if(fileContent.Length < 2) {
            Debug.LogError("Structures_list not valid");
            return;
        }

        string[] splittedContent = fileContent.Split(';');
        for(int i=0; i < splittedContent.Length - 1; i++) {
            CreateStructure(splittedContent[i]);
        }

    }
        int tempActivatedModel = Global.currentModel;

    for (int i = 0; i < transform.childCount - 1; i++) {
        GetComponent<ModelsBehavior>().ActivateModel(i);
        BroadcastMessage("LoadFile");
    }

    for (int i = 0; i < transform.childCount; i++) {
        GetComponent<ModelsBehavior>().ActivateModel(i);
        for (int j=0; j < Structures.Count; j++)
            Structures[j].SendMessage("RefreshAllMeshes");
```

```
        }

        GetComponent<ModelsBehavior>().ActivateModel(tempActivatedModel);
    }

    public void PrintStructures() {
        MessagesHandler.instance.ClearMessage();
        foreach (Transform structure in
            StructuresGameObject.transform) {
             MessagesHandler.instance.AddToggle(structure.name);
        }
    }


    IEnumerator PrintStructuresEndFrame() {
        yield return new WaitForEndOfFrame();
        PrintStructures();
    }

    IEnumerator SetToggleAfterDelay(int i ) {
        yield return new WaitForEndOfFrame();
        MessagesHandler.instance.SetToggle(i, true);
    }

}
```

5.4 Script StructureHandler()

```
public class StructureHandler : MonoBehaviour {

    public int structureIndex;
    public string structureName;
    Material SelectedStructureMaterial;
    Material StructureMaterial;

    void Awake() {

        SelectedStructureMaterial =
            Resources.Load("SelectedStructureMaterial") as Material;
        StructureMaterial = Resources.Load("StructureMaterial") as
            Material;

        for (int i=0; i<Global.nbModels; i++) {
            GameObject SubStructure = new GameObject("Model_" +
                (i+1));
            SubStructure.transform.parent = transform;
            SubStructure.transform.localPosition = Vector3.zero;
            SubStructure.transform.localRotation =
                Quaternion.identity;
            SubStructure.AddComponent<Structure>();
        }
        UpdateStructureDisplay();
    }

    void RefreshAllMeshes() {
        BroadcastMessage("RefreshMesh");
    }

    void Select() {
        transform.GetChild(Global.currentModel).GetComponent<cakeslice.Outline>()
            = 0;
    }

    void Unselect() {
        transform.GetChild(Global.currentModel).GetComponent<cakeslice.Outline>()
            = 1;
    }

    public Structure GetCurrentSubstructure() {
        return
            transform.GetChild(Global.currentModel).GetComponent<Structure>();
    }
```

```
    void Rename( string str ) {
        structureName = str;
    }

    void UpdateStructureDisplay() {
        foreach(Transform child in transform) {
            child.gameObject.SetActive(child.GetSiblingIndex() ==
                Global.currentModel);
        }
    }

}
```

5.5 Script Structure()

```
public class Structure : MonoBehaviour {

    public int structureIndex;
    public string structureName;

    Mesh myMesh;
    List<Vector3> verts;
    List<int> tris;

    void Awake() {
        verts = new List<Vector3>();
        tris = new List<int>();
        structureName = "Unnamed";
        myMesh = new Mesh();
        gameObject.AddComponent<MeshFilter>();
        gameObject.GetComponent<MeshFilter>().mesh = myMesh;
        gameObject.AddComponent<MeshRenderer>();
        gameObject.GetComponent<MeshRenderer>().material =
            Resources.Load("StructureMaterial") as Material;
        myMesh.triangles = tris.ToArray();
        myMesh.vertices = verts.ToArray();

        gameObject.AddComponent<cakeslice.Outline>();
    }

    //  add a vertex and return its index
    int AddVertex( Vector3 vert ) {
        //  check if vertex already exists
        int vertexIndex = FindVertex(vert);
        if(vertexIndex != -1)
            return vertexIndex;

        verts.Add(vert);
        return (verts.Count - 1);
    }

    //  add a triangle
    public void AddTriangle( Vector3 a, Vector3 b, Vector3 c) {

        //  add vertices and get their indexes
        int ia = AddVertex(a);
        int ib = AddVertex(b);
        int ic = AddVertex(c);

        //  add face
```

```
        tris.Add(ia);
        tris.Add(ib);
        tris.Add(ic);

    }

    //  try to delete a triangle and return true if it succeeded
    public bool DeleteTriangle(Vector3 a, Vector3 b, Vector3 c ) {

        int ia = FindVertex(a);
        if (ia == -1) return false;
        int ib = FindVertex(b);
        if (ib == -1) return false;
        int ic = FindVertex(c);
        if (ic == -1) return false;

        for(int i=0; i<tris.Count; i += 3) {
            if (tris[i].Equals(ia) && tris[i+1].Equals(ib) &&
                tris[i+2].Equals(ic)) {
                tris.RemoveRange(i, 3);
                return true;
            }
        }
        return false;
    }

    //    return index of vertex, or -1 if not found
    int FindVertex(Vector3 vert) {
        for (int i = 0; i < verts.Count; i++) {
            if (verts[i].Equals(vert))
                return i;
        }
        return -1;
    }

    //    update mesh renderer
    public void RefreshMesh() {
        myMesh.vertices = verts.ToArray();
        myMesh.triangles = tris.ToArray();
    }

}
```

5.6 Script TrianglePainter()

```
public class TrianglesPainter : MonoBehaviour {

    Mesh myMesh;
    List<int> currentTreatedTris = new List<int>();
    List<int> clickedTris = new List<int>();
    List<int> indices;
    Vector3 hitPoint;
    LayerMask modelsLayerMask;
    int[] structuresTab;

    void Awake(){
        modelsLayerMask = 1 << 9;
        myMesh = GetComponent<MeshFilter>().mesh;
        indices = new List<int>(myMesh.triangles);
        structuresTab = new int[(myMesh.triangles.Length)/3];
        ClearStructureTab();
    }

    void ReceiveRaycast(RaycastHit hit) {

        //  get current structure
        Structure currentStructure =
           GlobalStructuresHandler.instance.GetSelectedStructure();
        if (currentStructure == null)
            return;

        //  get triangle hit
        int tIndex = hit.triangleIndex;
        if (clickedTris.Contains(tIndex))
            return;

        //  start recursion
        ColorFace(currentStructure, tIndex, 0);

        //  clear list of treated triangles
        currentTreatedTris.Clear();

        //  refresh mesh renderer
        currentStructure.RefreshMesh();

    }

    public void ColorFace( Structure currentStructure, int tIndex,
       int loop) {
```

```
//  if triangle already processed, return
if (currentTreatedTris.Contains(tIndex))
    return;
//  add to processed triangles
currentTreatedTris.Add(tIndex);

//  get the 3 vertices positions
Vector3 vert1 = myMesh.vertices[indices[tIndex * 3]];
Vector3 vert2 = myMesh.vertices[indices[tIndex * 3 + 1]];
Vector3 vert3 = myMesh.vertices[indices[tIndex * 3 + 2]];

//  if current triangle has no structure
if (structuresTab[tIndex] == -1) {
    //  set to current structure
    structuresTab[tIndex] =
       GlobalStructuresHandler.instance.GetSelectedStructureIndex();
    //  add triangle to current structure's mesh (color
       feedback)
    currentStructure.AddTriangle(vert1, vert2, vert3);
}

//  check for recursion
if (loop >= Global.searchDepth) return;

//  switch to world space
vert1 = transform.TransformPoint(vert1);
vert2 = transform.TransformPoint(vert2);
vert3 = transform.TransformPoint(vert3);

//  compute centers of segments + offset
Vector3[] centers = new Vector3[3];
float offset = 0.05f;
centers[0] = (1 + offset) * 0.5f * (vert1 + vert2) -
   offset * vert3;
centers[1] = (1 + offset) * 0.5f * (vert2 + vert3) -
   offset * vert1;
centers[2] = (1 + offset) * 0.5f * (vert3 + vert1) -
   offset * vert2;

Vector3 origin = Camera.main.transform.position;

//  for each center, launch a raycast and recurse with hit
   triangle
for (int i=0; i<3; i++) {
    Vector3 goal = centers[i];
    Vector3 direction = goal - origin;
    RaycastHit hit;
```

```
        if (Physics.Raycast(origin, direction, out hit,
            1000.0f, modelsLayerMask)) {
            TrianglesPainter tRemover =
                hit.collider.GetComponent<TrianglesPainter>();
            if(tRemover != null)
                tRemover.ColorFace(currentStructure,
                    hit.triangleIndex, loop + 1);
        }
    }
}

Vector3 GetCenter(int tIndex ) {
    return
        transform.TransformPoint((myMesh.vertices[indices[tIndex
        * 3]] + myMesh.vertices[indices[tIndex * 3 + 1]] +
        myMesh.vertices[indices[tIndex * 3 + 2]]) / 3.0f);
}

void ReceiveRaycastDelete( RaycastHit hit ) {
    Structure currentStructure =
        GlobalStructuresHandler.instance.GetSelectedStructure();
    if (currentStructure == null)
        return;
    int tIndex = hit.triangleIndex;
    DeleteFace(currentStructure, tIndex, 0);
    currentTreatedTris.Clear();
    currentStructure.RefreshMesh();
}

public void DeleteFace(Structure currentStructure, int tIndex,
   int loop) {

    if (currentTreatedTris.Contains(tIndex))
        return;
    currentTreatedTris.Add(tIndex);

    Vector3 vert1 = myMesh.vertices[indices[tIndex * 3]];
    Vector3 vert2 = myMesh.vertices[indices[tIndex * 3 + 1]];
    Vector3 vert3 = myMesh.vertices[indices[tIndex * 3 + 2]];

    if(currentStructure.DeleteTriangle(vert1, vert2, vert3))
        structuresTab[tIndex] = -1;

    if (loop >= Global.searchDepth) return;

    vert1 = transform.TransformPoint(vert1);
    vert2 = transform.TransformPoint(vert2);
```

```
        vert3 = transform.TransformPoint(vert3);

        Vector3[] centers = new Vector3[3];

        float offset = 0.03f;

        centers[0] = (1 + offset) * 0.5f * (vert1 + vert2) -
            offset * vert3;
        centers[1] = (1 + offset) * 0.5f * (vert2 + vert3) -
            offset * vert1;
        centers[2] = (1 + offset) * 0.5f * (vert3 + vert1) -
            offset * vert2;

        //Vector3 origin = 50 * Vector3.Scale(vert2 - vert1, vert3
            - vert1) + vert1;

        for (int i = 0; i < 3; i++) {
            Vector3 origin = Camera.main.transform.position;
            Vector3 goal = centers[i];

            //Debug.DrawLine(origin, goal, Color.red, 15);
            Vector3 direction = goal - origin;

            RaycastHit hit;
            if (Physics.Raycast(origin, direction, out hit,
                1000.0f, modelsLayerMask)) {
                TrianglesPainter tRemover =
                    hit.collider.GetComponent<TrianglesPainter>();
                if (tRemover != null)
                    tRemover.DeleteFace(currentStructure,
                        hit.triangleIndex, loop + 1);
            }
        }
    }

    void ClearStructureTab() {
        for (int i = 0; i < structuresTab.Length; i++) {
            structuresTab[i] = -1;
        }
    }

    public void RemoveStructure(int structureIndex) {
        for(int i=0; i<structuresTab.Length; i++) {
            if (structuresTab[i] == structureIndex)
                structuresTab[i] = -1;
        }
    }
```

```
    public void ExportFile() {
        string fileName = "Structures_" + Global.currentModel +
            "_" + transform.GetSiblingIndex() + ".txt";
        if (File.Exists(fileName)) {
            File.Delete(fileName);
        }
        StreamWriter sr = File.CreateText(fileName);
        sr.WriteLine(string.Join(";", new
            List<int>(structuresTab).ConvertAll(i =>
            i.ToString()).ToArray()));
        sr.Close();
    }

    public void LoadFile() {
        string fileName = "Structures_" + Global.currentModel +
            "_" + transform.GetSiblingIndex() + ".txt";
        if (File.Exists(fileName)) {
            StreamReader sr = new
                StreamReader(Application.dataPath + "/../" +
                fileName);
            string fileContent = sr.ReadToEnd();
            sr.Close();
            string[] splitLine = fileContent.Split(';');
            if(splitLine.Length == structuresTab.Length) {
                for(int i=0; i< structuresTab.Length; i++) {
                    int structureIndex = int.Parse(splitLine[i]);
                    if (structureIndex != -1) {
                        structuresTab[i] = structureIndex;
                        GlobalStructuresHandler.instance.Structures[structureIndex
                            * 3]], myMesh.vertices[indices[i * 3 +
                            1]], myMesh.vertices[indices[i * 3 +
                            2]]);
            }
                }
            } else {
                Debug.LogError("Length of " + fileName + " is
                    incorrect");
            }
        } else {
            Debug.LogError("File " + fileName + " not found!");
        }
    }

}
```