Rapport de stage

Insertion d'aiguille et coupe échographique dans un maillage 3D

Tristan Guillard

Tuteur : Fabrice Jaillet

Master 2 Informatique ID3D
Université Claude Bernard Lyon 1
Mars 2021 - Juillet 2021

Résumé

Ce document résume les productions et résultats du stage de recherche et développement de 5 mois effectué par Tristan Guillard sur le simulateur du projet SPARTE (cf. 1.1.2). Ce projet a pour aboutissement l'élaboration d'un logiciel informatique permettant l'entraînement des nouveaux rhumatologues au geste de ponction de l'épaule.

Le stage s'est porté sur les travaux de rendu et l'amélioration de la base de code NExT (cf. 1.1.3) par l'ajout de nouveaux systèmes, l'augmentation de la modularité du code afin de pouvoir intégrer facilement de nouvelles fonctionnalités à l'application dans le futur et l'amélioration de systèmes existants. Les réalisations se sont concentrées sur les systèmes de déformation des tissus sous l'insertion d'une aiguille et d'affichage de la coupe échographique, ainsi que la pipeline graphique de l'application en général.

Ce document utilise l'écriture inclusive française. L'annexe B.1 donne les formes utilisées dans ce document et des liens utiles.

Abstract (English)

This document summarizes the productions and results of the 5-month long research and development internship carried out by Tristan Guillard on the simulator of the SPARTE project (cf. 1.1.2). The result of this project is the advancement of the development of a computer software for the training of new rheumatologists in the gesture of shoulder puncture.

The internship goal to enhance the NExT code base (cf. 1.1.3) by adding new systems, the increase in modularity of the code to be able to easily integrate new functionalities into the application in the future and the improvement of existing systems. Achievements were focused on tissue deformation systems under the insertion of a needle and display of the ultrasound section, as well as the graphics pipeline of the application in general.

Remerciements

Je tiens à remercier toute l'équipe ORIGAMI pour leur accueil. Bien que le stage ait eu lieu en distanciel, l'équipe m'a permis de prendre part à des activités de chercheurs comme leur groupe de lecture hebdomadaire ce qui m'a permis de comprendre mieux la dynamique d'une équipe de recherche en informatique.

J'aimerais aussi remercier toute l'équipe du LIRIS - particulièrement Jean-Marc Petit, directeur du LIRIS - de m'avoir offert l'opportunité de faire ce stage. Le LIRIS m'a permis de me rendre dans les locaux du laboratoire pour les réunions touchant au stage les plus importantes et de récupérer les travaux des précédents chercheurs.

Je remercie la fédération Informatique de Lyon pour le financement de ce stage dans le cadre du projet SPARTE (projets 2021-2023).

Un merci particulier à Fabrice Jaillet mon maître de stage, de m'avoir accompagné tout au long de ces 5 mois malgré le contexte sanitaire, de m'avoir aiguillé tout au long de mes recherches et aidé dans mes travaux. Je le remercie aussi pour son soutien.

Je remercie aussi mes camarades stagiaires, tout particulièrement la promotion ID3D, avec qui j'ai pu partager mes accomplissements et difficultés rencontrées tout au long du stage.

Je remercie enfin ma famille et mes ami·es pour leur soutien.

Table des matières

1 Environnement **8**
1.1 Équipe et Projet . . . 8
1.1.1 Équipe . . . 8
1.1.2 Projet SPARTE (Simulator of Puncture for ARTiculations under Echography) . . . 9
1.1.3 Projet NExT (Numerical Experiment Tool) . . . 10
1.2 Contexte du stage . . . 12
1.2.1 Groupes de recherche . . . 12
1.2.2 Travail parallèle avec le LBMC (Laboratoire de Biomécanique et Mécanique des Chocs) . . . 12
1.2.3 Recherches bibliographique sur le geste de ponction . . . 13
1.3 Plate-forme logicielle . . . 14
1.3.1 Base de code NExT . . . 14
1.3.2 Architecture de la bibliothèque NExT . . . 16
1.3.3 Architecture du *viewer* NExT . . . 18
1.3.4 Outils mathématiques et bibliothèques externes . . . 20

2 Travaux **22**
2.1 Mise en place d'une scène pour les développements du stage . . . 22
2.2 Système d'animation . . . 23
2.2.1 Keyframes . . . 23
2.2.2 Fichier d'animation et Reader . . . 23
2.2.3 Animator . . . 24
2.2.4 Interpolation et *interpolated keyframe* . . . 25
2.2.5 Vélocité . . . 26
2.2.6 Composant . . . 26
2.2.7 Système . . . 27
2.2.8 Résumé et extensions possibles . . . 28
2.3 Système de Rendu . . . 28
2.3.1 Analyse du système avant travaux . . . 28
2.3.2 Identification des 2 étapes de calculs . . . 30
2.3.3 Calculs mis en place dans la première étape . . . 30
2.3.4 Mise en place de la pipeline de post traitement . . . 32
2.3.5 Effets de post traitements en place . . . 33
2.3.6 Réalisations . . . 34
2.4 Système de déformation . . . 36

3 Bilan du stage **38**
3.1 Méthode de Travail 38
3.1.1 Réunions 38
3.1.2 Développement 38
3.2 Conclusion 39

A Code et documentation **41**
A.1 Documentation fichier d'animation 42
A.2 deformation.frag 43
A.3 displacement.frag 45
A.4 blur.frag 47
A.5 cometTail.frag 49

B Autre **51**
B.1 Écriture inclusive 51

Table des figures

1.1 ORIGAMI parmi les équipes de recherches du laboratoire LIRIS [source : LIRIS] . . . 8
1.2 Prototype pour les développements du projet SPARTE [source : ORIGAMI] 9
1.3 Prototype final du projet SPARTE [source : ORIGAMI] . . . 11
1.4 Bras mécanique du projet SPARTE [source : ORIGAMI] . . . 11
1.5 Demi-maillage produit par le LBMC pour leurs simulations . . . 13
1.6 Architecture NExT ECS générale . . . 17
1.7 Description basique des communications entre composants . . . 18
1.8 Interface de l'application NExT . . . 19

2.1 Architecture ECS de la scène de travail . . . 22
2.2 Fichier de d'animation démonstration . . . 24
2.3 Fichier de d'animation enregistré par le système d'animation . . . 24
2.4 Les 3 types d'interpolations implémentés dans NExT . . . 25
2.5 Héritage des composants d'animation . . . 27
2.6 Fonctionnement de la pipeline avant le stage . . . 29
2.7 Résumé du fonctionnement de la pipeline graphique en fin de stage . . . 30
2.8 Effet "Comet tail" de référence [source : [5]] . . . 31
2.9 Effets de rendu . . . 32
2.10 Déformation partiellement bloquée par l'os . . . 34
2.11 Différents effets de *blur* sur la coupe échographique . . . 35

Chapitre 1

Environnement

1.1 Équipe et Projet

1.1.1 Équipe

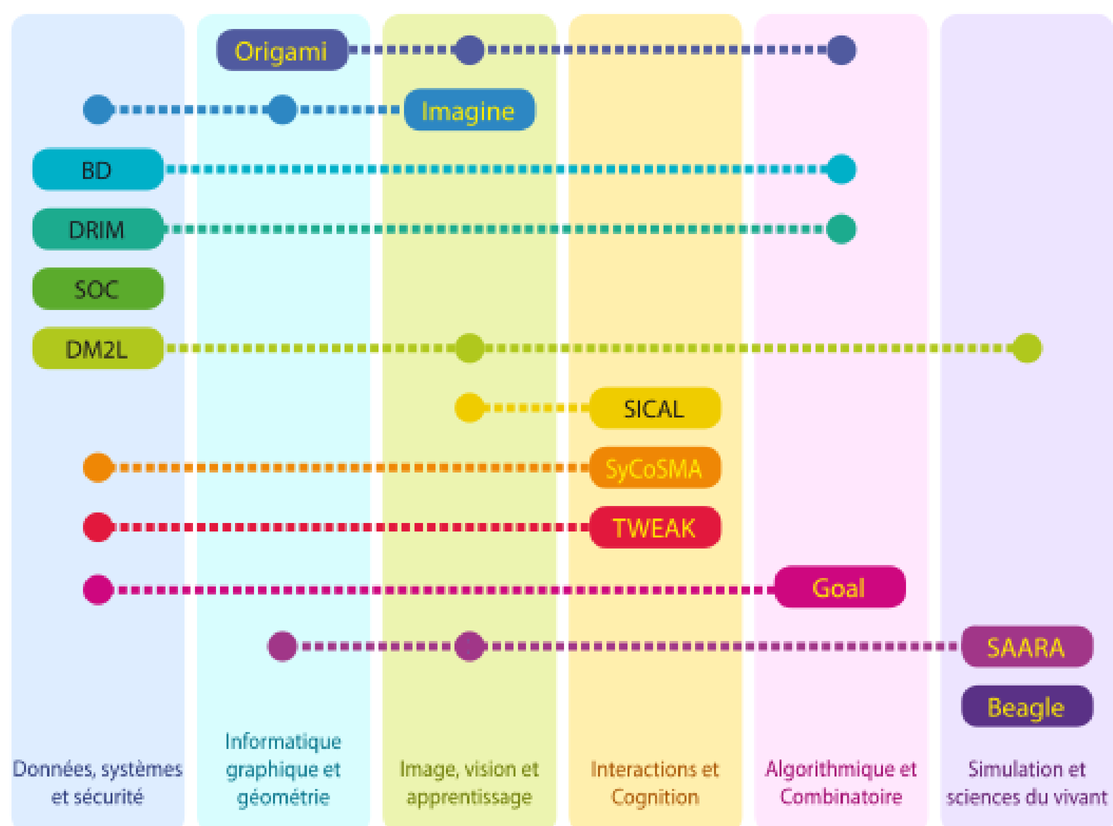


FIGURE 1.1 – ORIGAMI parmi les équipes de recherches du laboratoire LIRIS [source : LIRIS]

Dans le cadre de ce stage, j'ai été accueilli·e par l'équipe de recherche ORIGAMI, du laboratoire LIRIS, qui est spécialisée dans trois domaines de l'informatique graphique.

— La géométrie, où leurs recherches se concentrent sur le développement d'algorithmes d'analyses et de création de maillages, surfaces, nuages de points et sur des générations procédurales. Leurs travaux recouvrent aussi le domaine des maillages

dynamiques comme dans l'application NExT (cf. 1.1.3) dont le développement a été au coeur du stage.
- Le rendu, où l'équipe se focalise sur la simulation d'éclairage, l'échantillonnage et la réalité virtuelle. Les travaux se basent majoritairement sur l'amélioration du rendu réaliste ou de ses performances.
- La simulation dans les domaines de déformation et discrétisation des fluides.

Des travaux récents de l'équipe portent par exemple sur le re-meshing, les compressions et l'affichage de maillage en réalité virtuelle ainsi que sur des outils mathématiques comme des estimateurs différentiels pour nuage de points.

L'organisation des équipes de recherche du LIRIS est décrite en Figure 1.1, le LIRIS étant une unité mixte de recherche du CNRS. Le laboratoire possède plus de 300 membres actifs à ce jour, dont une centaine de doctorant·es et post-doctorant·es et une autre de professeur·euses et maître·sses de conférences, sans compter les différent·es stagiaires recruté·es chaque année. L'équipe ORIGAMI, quant à elle, compte 24 membres permanents, 10 doctorant·es, 3 post-doctorant·es, 16 étudiant·es en cours de thèse et 9 stagiaires.

Au sein de cette équipe, j'ai été sous la tutelle de Fabrice Jaillet, mon maître de stage et de Florence Zara, enseignante référente de l'université Claude Bernard. Les deux enseignant·es-chercheur·euses m'ont aidé à comprendre la base de code NExT (cf. 1.1.3). Ils m'ont également aidé dans mes recherches de développements relatifs à l'application. Nous avons tenu des réunions de manière bi-hebdomadaires (en moyenne) avec Fabrice Jaillet afin qu'il suive ma progression tout au long du stage, qu'il me conseille sur les prochains objectifs et qu'il me donne des retours sur mes travaux.

L'équipe tient des groupes de lecture pour que les différents chercheur·euses partagent leurs travaux en cours. J'ai eu la chance de pouvoir participer à certains groupes de lecture, notamment sur des outils mathématiques sur des maillages, et autres domaines proche de mon sujet de stage.

1.1.2 Projet SPARTE (Simulator of Puncture for ARTiculations under Echography)

FIGURE 1.2 – Prototype pour les développements du projet SPARTE [source : ORIGAMI]

Le projet SPARTE est un projet de recherche ayant pour but d'aider des praticien·es du domaine médical (rhumatologie) à s'entraîner à effectuer certains gestes complexes afin de réduire les risques liés à l'apprentissage sur les premiers patients. Un premier prototype de ce projet a été développé par le laboratoire LIRIS et ses partenaires - le laboratoire Ampère, le LIBM (Laboratoire Interuniversitaire de Biologie de la Motricité), et le service de rhumatologie de l'Hôpital Lyon Sud.

Le premier geste sur lequel l'équipe s'est penchée est un geste de ponction de l'épaule. Les jeunes médecins suivent une formation particulière pour ce geste. Leur permettre d'avoir un entraînement préliminaire est donc l'objectif de ce projet. Ce geste a pour but d'injecter ou de prélever du fluide se situant le long de la bourse sous-acromiale dans l'épaule des patients afin de soulager l'arthrite. Ce geste est complexe, car il requiert l'utilisation coordonnée d'une aiguille d'une main et d'une sonde échographique de l'autre, afin de pouvoir avoir des images internes à l'épaule de manière non-invasive. La complexité vient aussi du fait que la vue est limitée à une image ultrasonore et que l'aiguille doit être insérée le long du plan de la sonde échographique afin qu'elle puisse être vue et guidée jusqu'à l'endroit de l'injection ou du prélèvement.

L'objectif final du projet est d'avoir une application, liée à du matériel physique, sur laquelle les praticien·nes pourront s'entraîner. Le matériel est constitué d'une épaule imprimée en 3D ainsi que de deux manettes pour contrôler l'aiguille et la sonde. La manette de l'aiguille est une manette à retour haptique à 6 degrés de liberté afin de simuler les retours de force ressentis lors de la pénétration de l'aiguille. L'utilisateur·ice s'entraînera alors à localiser l'aiguille dans la coupe échographique, à repérer les différentes couches composant l'anatomie lors de l'insertion de l'aiguille et à atteindre le bon endroit avec l'aiguille afin de réaliser l'injection ou le prélèvement.

Le simulateur a besoin de prendre en compte beaucoup de paramètres de la pénétration de tissus par une aiguille de manière à permettre un entraînement correct. En effet, il est nécessaire de faire des calculs de déformation des tissus sous l'action de l'aiguille car elle est invisible hors du plan échographique. Lae praticien·ne va d'ailleurs faire des petits mouvements avec l'aiguille, dans le but de pouvoir la localiser grâce aux mouvements des tissus. La coupe échographique doit aussi avoir un rendu réaliste afin d'assurer la qualité de l'entraînement, mais cela ne doit pas se faire à l'encontre des performances du logiciel.

À long terme, le logiciel fournira des exercices d'entraînement avec évaluation des performances des praticien·nes, afin d'avoir un retour et de pouvoir améliorer leur geste de façon à limiter toute blessure et faux mouvement. Pour ce faire, il faudra développer des systèmes d'enregistrement et d'évaluation. Il pourrait être intéressant de faire un système de relecture de la performance pour expliquer les faux mouvements effectués tout au long de l'exercice.

1.1.3 Projet NExT (Numerical Experiment Tool)

Le projet NExT est une base de code développée par l'équipe ORIGAMI afin de pouvoir effectuer divers calculs et simulations. Il a notamment été utilisé pour créer un simulateur d'accouchement pour l'entraînement des obstétricien·nes, permettant de travailler la pose des forceps, où la déformation de maillage et la simulation de forces ont une part très importante.

Ce projet permet d'effectuer des simulations avancées sur divers maillages (surfaciques, volumiques, nuages de points).

Le projet NExT est divisé en deux :

— Une bibliothèque comportant des systèmes et composants qui fonctionnent in-

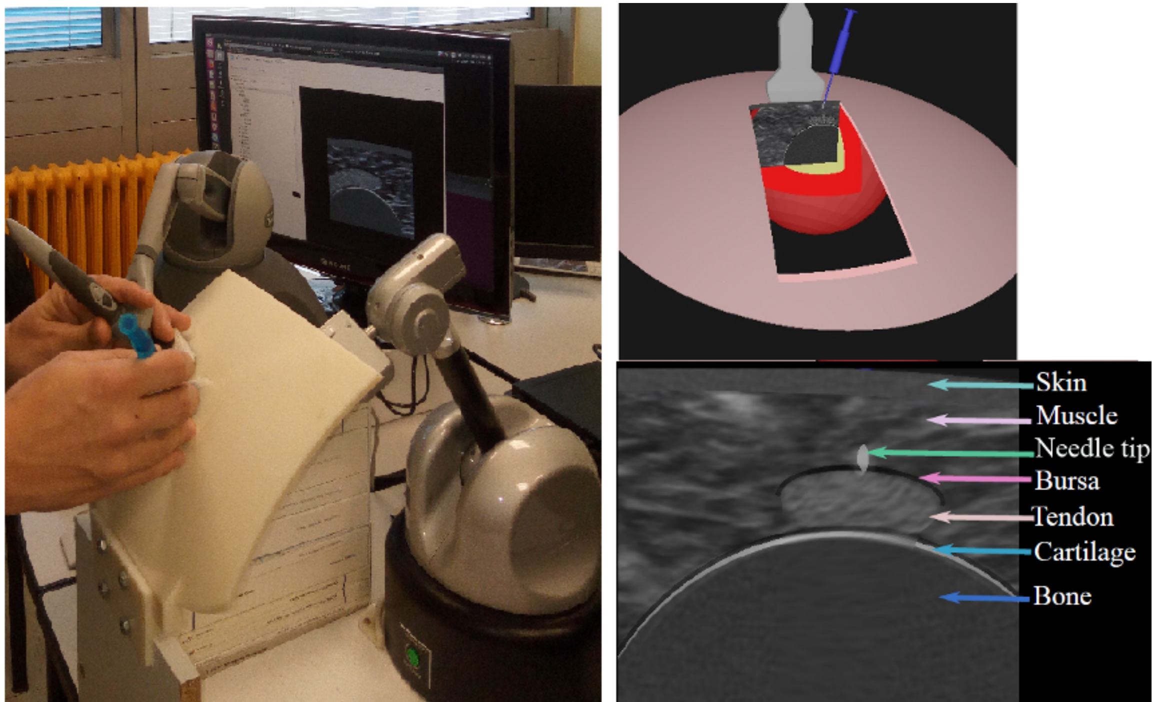


FIGURE 1.3 – Prototype final du projet SPARTE [source : ORIGAMI]

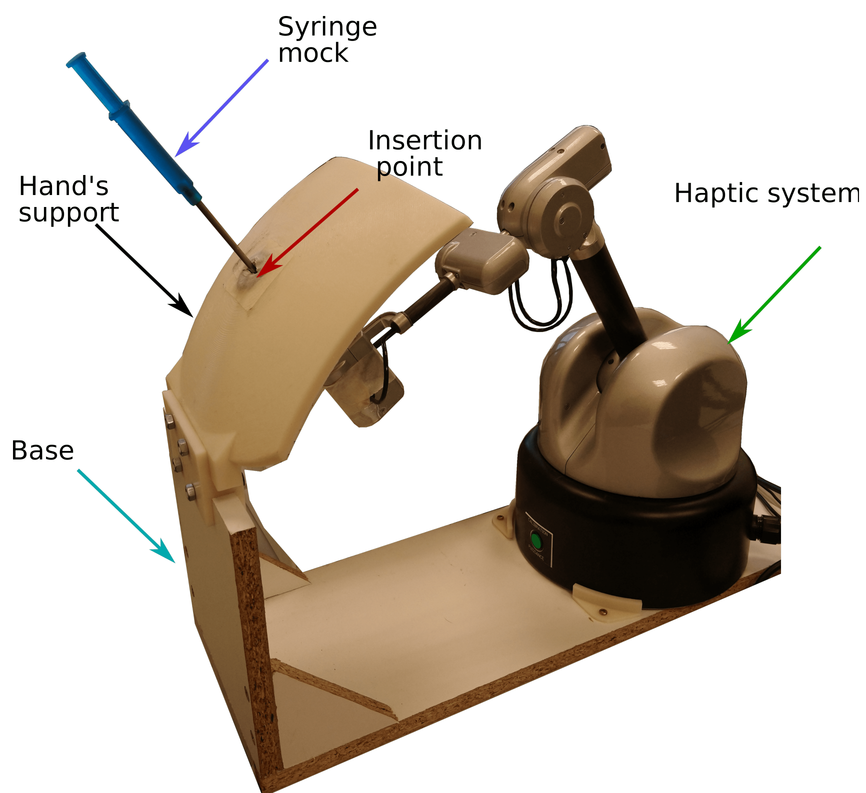


FIGURE 1.4 – Bras mécanique du projet SPARTE [source : ORIGAMI]

dépendamment du cas d'utilisation. Cette bibliothèque contient une variété de systèmes pouvant fonctionner en parallèle (en utilisant le *design pattern ECS* cf. 1.3.2).
— Une interface visuelle QT permettant l'utilisation de la bibliothèque en créant différentes scènes.

C'est autour de la prise en main et de l'amélioration du projet NExT que s'est articulé le stage.

1.2 Contexte du stage

1.2.1 Groupes de recherche

Étant en équipe de recherche, il m'a été permis d'être présent·e à plusieurs évènements en rapport avec la recherche.

Groupe de lecture Les groupes de lectures tenus par l'équipe ORIGAMI m'ont permis d'avoir quelques aperçus de la composition de l'équipe de recherche et des travaux en cours. J'ai pu assister à des lectures sur les thèmes de *vantage point tree* ou sur des outils mathématiques plus généraux pour l'intégration.

Eurographics 2021 J'ai également eu la chance d'assister aux visioconférences d'EUROGRAPHICS 2021. Certaines de ces conférences se rapprochent du domaine de mon stage, comme celles sur la déformation de maillage pour simuler des mouvements musculaires, synthèse procédurale de vaisseaux sanguins ou insertion d'aiguille molle. Elles m'ont permis d'avoir de nouvelles idées sur le stage.

1.2.2 Travail parallèle avec le LBMC (Laboratoire de Biomécanique et Mécanique des Chocs)

Le stage a été effectué en parallèle d'un autre stage dans le LBMC (Lyon 1). Ce stage de Master 2 - réalisé par Mathilde Équine - a eu pour but de travailler sur la déformation précise du maillage sous l'insertion de l'aiguille. Les stages ont été effectués en parallèle pour que les travaux de Mathilde puissent être utiles à NExT. Nous avons tenu environ une réunion par mois pour mettre en commun nos travaux.

Mathilde a travaillé sur la création d'un maillage spécifique et la réalisation de simulations très précises de la déformation sous l'insertion de l'aiguille. Le maillage choisit est un maillage hexaédrique. Comme montré en Figure 1.5, le maillage est très détaillé autour de l'axe d'insertion de l'aiguille, permettant l'insertion d'éléments de cohésion et une meilleure précision des résultats dans cette zone. L'utilisation des éléments de cohésions est la méthode la plus adaptée pour la simulation de rupture avec le logiciel de simulation LS DYNA. Ce sont les éléments qui vont rompre s'ils sont contraints au-delà de leur limite, ils permettent ainsi de guider la rupture et sa propagation tout au long de la simulation.

Afin de pouvoir utiliser ces résultats dans NExT, l'objectif était de créer un méta modèle de la déformation calculable en temps réel.

Du coté NExT, une architecture de code a été mise en place. Elle permet d'intégrer le méta modèle dès son obtention. Les résultats du LBMC seront intégré ultérieurement à l'application, car il reste encore du travail pour mettre en correspondance le maillage

spécialisé et les maillages des différents scénario utilisés dans NExT et vérifier l'intégration des résultats et des adaptations ajoutés au méta modèle. Cette étape est fondamentale, car elle permettra ensuite de vérifier les interpolations et extrapolations qui seront mises en place.

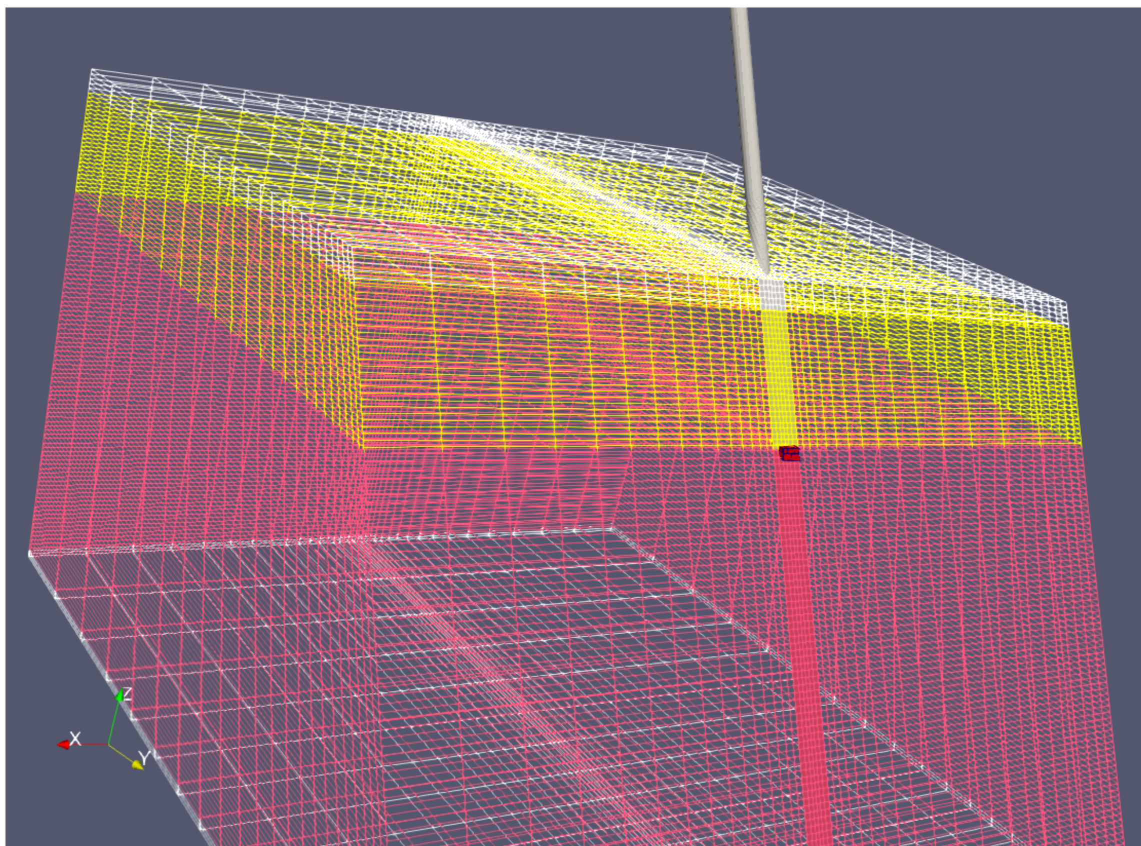


FIGURE 1.5 – Demi-maillage produit par le LBMC pour leurs simulations

1.2.3 Recherches bibliographique sur le geste de ponction

Préambule

Ce stage a été effectué comme un stage de recherche et développement - le but étant d'améliorer au plus l'application NExT pour que le projet SPARTE voit le jour - la partie développement ayant vraiment été au coeur des travaux. Le stage a toutefois commencé sur une recherche bibliographique extensive sur le geste de la ponction.

Pour commencer, il m'a fallu me familiariser avec le rendu échographique (cette playlist youtube [1] m'a beaucoup aidé) ainsi qu'avec l'anatomie de l'épaule.

Thèse de Charles Barnouin

Ce stage s'inscrit dans la continuité des travaux de Charles Barnouin pour sa thèse [2]. Elle est résumée dans ce papier [3]. Le but de cette thèse était de mettre en place tout le nécessaire pour créer l'application du projet SPARTE permettant aux apprentis de se former. Dans cette thèse, il a mis en place le calcul d'une coupe échographique en temps réel dans un maillage (cf. 2.3), un système de contrôle des objets 3D via des manettes haptiques ainsi que leur retour de force, et des débuts de travaux relatif à l'insertion de

l'aiguille dans le maillage. Étant de la recherche et non du développement, la plupart de ses travaux sont cependant restés à l'état de prototype.

Dans sa thèse, Charles présente les travaux qu'il a effectués pour développer le projet SPARTE. Afin de mettre en place la coupe échographique, les deux manières classiques sont :

— Simuler le rendu échographique avec un vrai calcul de lancé de rayon (comme les vraies échographies) directement dans le maillage. Ce rendu est le plus réaliste, mais est bien trop lent pour une application temps réel qui ne cherche pas nécessairement un réalisme parfait.

— Mettre en place une scène très précise avec tous les tissus et avoir des rendus de coupe dans ce cadre précis. Il faut alors acquérir une image ultrason 3D de la scène et la construire à partir de nombreuses coupes 2D. Il faut ensuite mettre en correspondance les images échographiques et le maillage. Le rendu est alors bon et instantané, mais pour chaque nouveau cas (partie du corps, pathologie, etc.) il faut prendre plusieurs semaines de travail pour chaque nouvelle scène, pour l'acquisition et le calibrage, ce qui devient très vite limitant sur un logiciel d'apprentissage où l'on souhaiterai composer avec le plus de cas et de pathologies possibles.

La technique que Charles a mise en place dans ses travaux de thèse est une nouvelle technique intermédiaire qui met le réalisme du rendu au second plan pour obtenir un rendu instantané et qui fonctionne immédiatement dans toute scène dès qu'un maillage correct est fourni. Cela permet à l'application NExT d'être modulaire pour les enseignements, ainsi, il suffit de très peu de changements pour ajouter une pathologie spécifique à un exercice existant. L'épaule étant la partie du corps où l'articulation est la plus dure à atteindre, mais où une aiguille rigide convient (pas besoin d'une aiguille souple qui est bien plus complexe à utiliser et nécessite un entraînement complexe) mais où une sonde échographique est tout de même nécessaire pour le geste, c'est l'endroit qui a été gardé pour les développements, mais le logiciel fonctionne sur n'importe quelle autre partie du corps si on lui fourni les données nécessaires.

Papiers supplémentaires

La thèse de Ma. Alamilla Daniel [4] a été d'une grande aide pour comprendre le geste de ponction ainsi que l'état de l'art dans ce domaine et les intégrations dans une certaine partie de NExT. Cette thèse a été effectuée dans le laboratoire Ampère, en parallèle de la thèse de Charles Barnouin, qui a implémenté de nombreux systèmes qui y ont été utiles.

Les différents papiers sur le logiciel (dont ce papier (A real-time ultrasound rendering with model-based tissue deformation for needle insertion)) ont été utiles pour comprendre le but du projet et les implémentations déjà présentes.

1.3 Plate-forme logicielle

1.3.1 Base de code NExT

NExT est un projet visant à faire des simulations géométriques et de l'affichage spécialisé, son développement a commencé il y a plus de 10 ans. Le stage a repris le développement de l'application là où les travaux de Charles Barnouin (cf. 1.2.3) se sont arrêtés. La base de code possède les caractéristiques suivantes :

— La base de code est en langage C++.

— La compilation est effectuée avec CMake.

— Elle est hébergée sur un dépôt Git.
— Elle ne contient pas de tests unitaires, ni d'architecture prête à en accueillir.

Git

Le dépôt Git est constitué de plusieurs branches, dont l'intégration des travaux en parallèle a été une des contraintes tout au long du stage. Le but étant que les développements puissent être facilement récupérés par l'équipe d'ORIGAMI pour la suite de leurs travaux.

— La branche `develop` est la branche principale du dépôt. Elle contient l'application dans une version stable.
— La branche `AfterCharles` est une version contenant les travaux de Charles Barnouin pour sa thèse. Cette branche n'a pas pu être intégrée à `develop` du fait des travaux qui étaient encore sous forme de prototype. C'est de cette branche que le stage a débuté.
— D'autres branches non relatives au stage ont été créées par l'équipe pour des développements d'autres étudiant·es pour leurs *Projets d'Orientation Master* de Master 1.

Le déroulement des travaux a suivi une méthode agile, ressemblant à une méthode *scrum* (cf. 3.1.1). Les *issues* git ont à la fois servi de *backlog* et ont aussi permis de suivre l'avancée dans les développements. Une branche principale a été créée, elle a pour but de rester le plus stable possible pour pouvoir être intégrée dès que nécessaire à `develop` (partiellement). Dans l'ensemble, chaque nouveau développement a suscité la création d'une branche qui est ultérieurement intégrée dans la branche principale dès que stable. Les travaux intégrés sur `develop` depuis d'autres branches (comme celle de l'équipe ORIGAMI) ont été intégrées à la branche principale du stage tout au long de ce dernier. Cela a notamment permis d'avoir des corrections de bugs sur la branche de travail sans perdre du temps à les développer lors du stage.

Architecture des dossiers et compilation

La base de code contient 3 dossiers principaux : `libraries`, `NExT_Viewer` et `data`

— `libraries` contient la partie bibliothèque de NExT (cf 1.3.2).
— `NExT_Viewer` contient la partie *Viewer* contenant les scènes et l'interface QT.
— `data` contient les données telles que les maillages, les textures, les animation, etc. Ce dossier est copié dans le dossier de build à chaque compilation par la configuration CMake.

Chaque dossier contient un `CMakeLists.txt` permettant a CMake de configurer les dossiers en cascade. Le `CMakeLists.txt` du dossier principal n'est cependant pas fonctionnel en l'état, il est donc nécessaire de précompiler la bibliothèque en un `.a` et de travailler directement dans le dossier `NExT_Viewer`. Cependant, ceci ne fonctionne que lorsque la bibliothèque est exclusivement en *header*, car tout changement dans la bibliothèque sera recompilé grâce aux *includes* des fichiers du *viewer*. La compilation de l'application est relativement longue (plusieurs minutes) due à la nature *header only* de la bibliothèque et les nombreux *includes* superflu dans le *viewer*. Pour cette raison, mais aussi pour des problèmes d'*includes* uniques de *headers* OpenGL `STBI`, on a choisi de dénaturer la bibliothèque en séparant les fichiers sur lesquels beaucoup de travail était nécessaire (comme les systèmes de rendu (2.3) et d'animation (2.2) en un *header* et un fichier d'implémentation `.cpp`. Cela réduit le temps de compilation à quelques secondes (car tous les fichiers incluant le *header* n'ont pas à être recompilés) et permet un gain d'efficacité important

lors du développement. Cependant, le `CMakeList.txt` principal ne fonctionnant pas, il faut recompiler la bibliothèque pour reconstruire le `.a` mis à jour à chaque modification de la bibliothèque, et ensuite de recompiler le *viewer*.

L'architecture ainsi modifiée étant relativement peu standard, l'utilisation d'un IDE complet n'avait pas vraiment d'avantages, j'ai donc développé sous *VSCode* un semi-IDE se couplant très bien à une compilation en ligne de commande. J'ai aussi développé quelques scripts *bash* pour pouvoir compiler simplement et rapidement en lignes de commande, ainsi que d'autres utilitaires permettant un gain de productivité.

Au tout début du stage, une étude empirique des vitesses de compilation des différents compilateurs disponibles a été effectuée. Il s'est avéré que dans le cadre du projet NExT, `clang++` est deux fois plus rapide à compiler que `g++`. En effet, `clang++` fonctionne mieux quand les *headers* contiennent beaucoup de code.

1.3.2 Architecture de la bibliothèque NExT

L'un des premiers enjeux du stage a été de comprendre l'architecture du code et les *design patterns* employés. Le *design pattern* autour duquel tourne l'application est le *design pattern* **Entity Component System** (ECS).

Pattern ECS

Le *design pattern* ECS permet de différencier les différents calculs en différents **systèmes** précis. Il est souvent utilisé dans le jeu vidéo, c'est d'ailleurs selon ce *design pattern* que fonctionnent les applicatifs de création de jeux comme Unity ou Unreal.

Une application aura plusieurs **systèmes** qui tourneront indépendamment les uns des autres, par exemple un système pour la dynamique (gravité, collisions), un autre pour les calculs d'affichage, etc. (cf. 1.3.2 pour les systèmes de NExT).

L'application sera ensuite divisée en plusieurs objets, dits **entités**, auxquels seront attachés divers **composants**. Ils sont souvent regroupés sous forme de **scène** ou de **niveau**. Chaque composant indiquera au système correspondant que l'entité sera inclue dans les calculs, et stockera les différentes valeurs d'attribut correspondantes. Les composants ne servent que pour stocker les données, ce sont les systèmes qui feront tous les calculs. Par exemple une entité n'ayant pas de composant `dynamic` ne sera pas soumis à la gravité. S'il en possède un, c'est dans ce composant que seront stockés sa masse et les autres valeurs relatives à la dynamique.

Les systèmes fonctionnant indépendamment, il est alors possible de mettre de nombreux composants sur la même entité. Cela permet une grande flexibilité de création des scènes.

Fonctionnement de NExT avec le pattern ECS

Dans l'application NExT, le pattern ECS a été implémenté de la façon suivante :

L'application fonctionne par `scene`.

Une scène contient un `engine manager` qui regroupe un nombre d'`engines` (moteurs) définit par scène. Cet `engine manager` s'assure de la transmission des messages entre l'interface Qt (cf. 1.3.3) et les différents moteurs. Chaque moteur tourne sur un thread différent, ce qui permet une parallélisation des calculs CPU ainsi que des fréquences de rafraîchissement indépendantes, donc adaptables. Il faut faire attention aux problèmes liés au multithreading. Chaque moteur contient un ou plusieurs `systems` (cf. 1.3.2). On va par exemple mettre en place un moteur uniquement pour le système haptique afin d'avoir

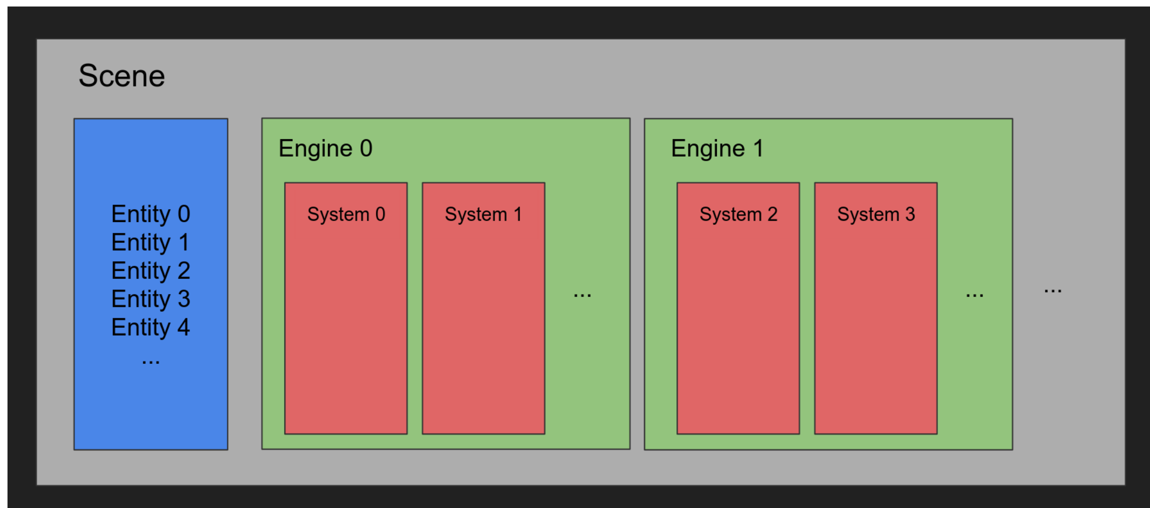


FIGURE 1.6 – Architecture NExT ECS générale

une fréquence de rafraîchissement de 1000Hz, un moteur pour le système de rendu (cf. 2.3) afin d'avoir le plus grand nombre d'images par seconde possible, et un moteur pour plusieurs systèmes annexes peu gourmands en calcul comme animation, graphic, etc.

Une scène contient aussi un ensemble d'entités, auxquelles on associe différents composants.

Chaque système contient une méthode `step()` (pas) qui permet de faire une étape (par exemple celle du système de rendu va redessiner une nouvelle frame, celle du système d'animation va calculer la nouvelle position des entités, etc.). À chaque émission du message `step` par l'interface Qt (appel unique par l'utilisateur·ice ou automatique avec la fréquence de rafraîchissement correspondante), l'`engine manager` va appeler les méthodes `step` des moteurs ayant besoin d'être mis à jour qui vont ensuite transmettre l'appel à leurs systèmes. Si plusieurs systèmes appartiennent au même `engine`, ils s'exécutent dans un ordre donné (cf. 1.3.2).

Modules

L'application NExT comporte un certain nombre de systèmes et de composants regroupés par modules. Ci-dessous la liste des modules classés par ordre d'exécution des systèmes.

- `InterfaceData` : fait le lien entre l'interface graphique et la scène.
- `Collision` : cherche les intersections entre les différentes géométries et donne les forces résultantes au `solver`.
- `Coupling Rigid Soft` : permet l'implémentation d'objets mixtes rigidbody et objets déformables. Notamment utilisé dans la scène d'accouchement pour la déformation de la tête d'un bébé sous la poussée de son corps rigide.
- `Haptique` : récupère les données des manettes haptiques pour modifier l'état des objets dans la scène.
- `Animation` : rejoue ou enregistre les mouvements d'objets dans la scène. Décrit section 2.2.
- `Solver` : résout les équations de la mécanique pour les objets dynamiques de la scènes (gravité, forces de collision, etc.).
- `Déformation` : modifie la géométrie des objets déformables selon des fonctionnelles 3D.

— `Adaptation` : rectifie le maillage après application des forces (remaillage, résolution de problèmes de sur-élasticité, etc.)
— `Graphic` : système intermédiaire réunissant les informations des autres systèmes et préparant l'affichage par le système de rendu (par exemple, modifie les buffers openGL quand il y a besoin pour éviter de les réévaluer à chaque frame). Ce système va calculer les primitives à afficher en fonction des modes sélectionnés par l'utilisateur·ice. Ce système est central dans l'application NExT, car c'est le seul qui va partager des informations au système de rendu, tout système voulant communiquer avec ce dernier passera donc par le composant graphique de l'entité.
— `Render` : système OpenGL permettant l'affichage de la scène. Il sert aussi aux calculs de la coupe échographique, de post traitement, etc.

Composant graphique et communications

On notera que le composant graphique est devenu au cours des différents développements un composant de communication entre les différents modules. En effet, le système de rendu (cf. 2.3) a besoin d'information des différents autres modules (la position 3D du système haptique ou d'animation, les informations de collision, etc.), ils sont donc la plupart du temps écrits dans le composant graphique, car le système de rendu n'a accès qu'à ce composant selon la composition traditionnelle des scènes. Il aurait été possible de partager tous les composants à tous les systèmes, mais il aurait fallu retravailler toutes les scènes existantes, cet artefact de communication à travers le composant ne respecte pas vraiment le pattern ECS, mais réhabiliter le pattern aurait pris beaucoup de temps de développement et de tests.

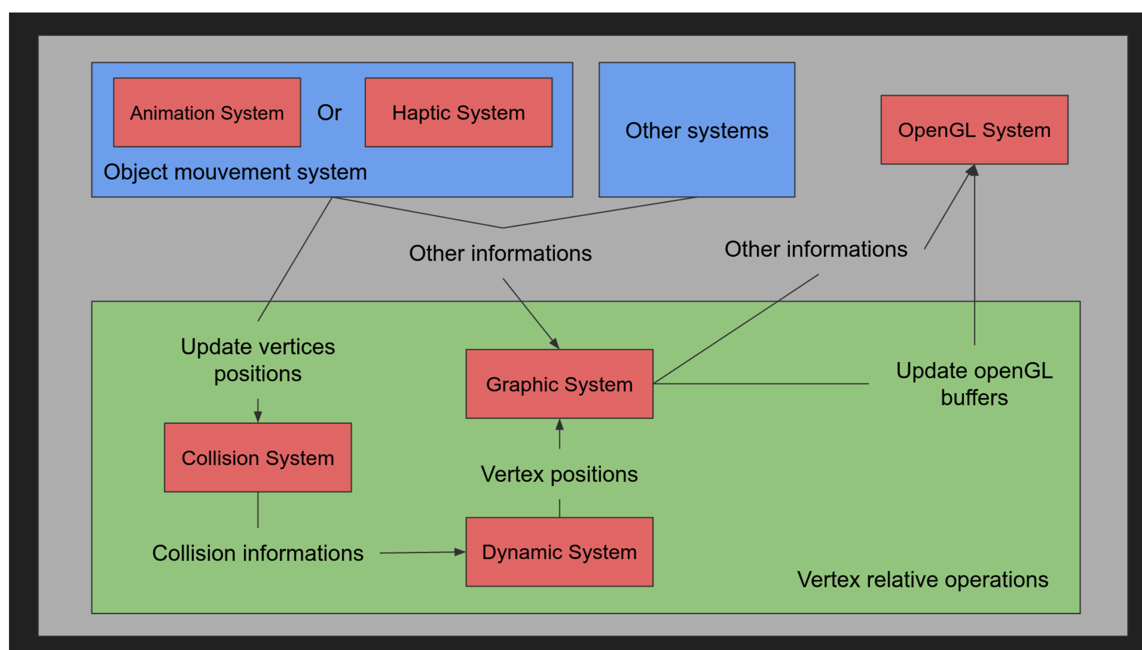


FIGURE 1.7 – Description basique des communications entre composants

1.3.3 Architecture du *viewer* NExT

Le *viewer* est toute la partie hors bibliothèque de l'application. Cette partie utilise la bibliothèque NExT à travers son API. C'est dans cette partie que l'on va créer les scènes

et l'interface graphique, ici avec Qt. Le *viewer* décrit ici est celui utilisé lors du stage, mais d'autres existent pour d'autres projets.

Builders

Les scènes et entités sont créées dans des `builder` qui seront appelés à la création de la scène. Les builders d'entités permettent la lecture des maillages, topologies, textures, composants et valeurs qui seront utilisées dans les différents systèmes.

Les builders de scènes vont utiliser les builders d'entités pour construire les objets et potentiellement modifier certains composants. Ils construiront aussi les moteurs en les remplissant des systèmes nécessaires au déroulement de la scène.

Interface QT

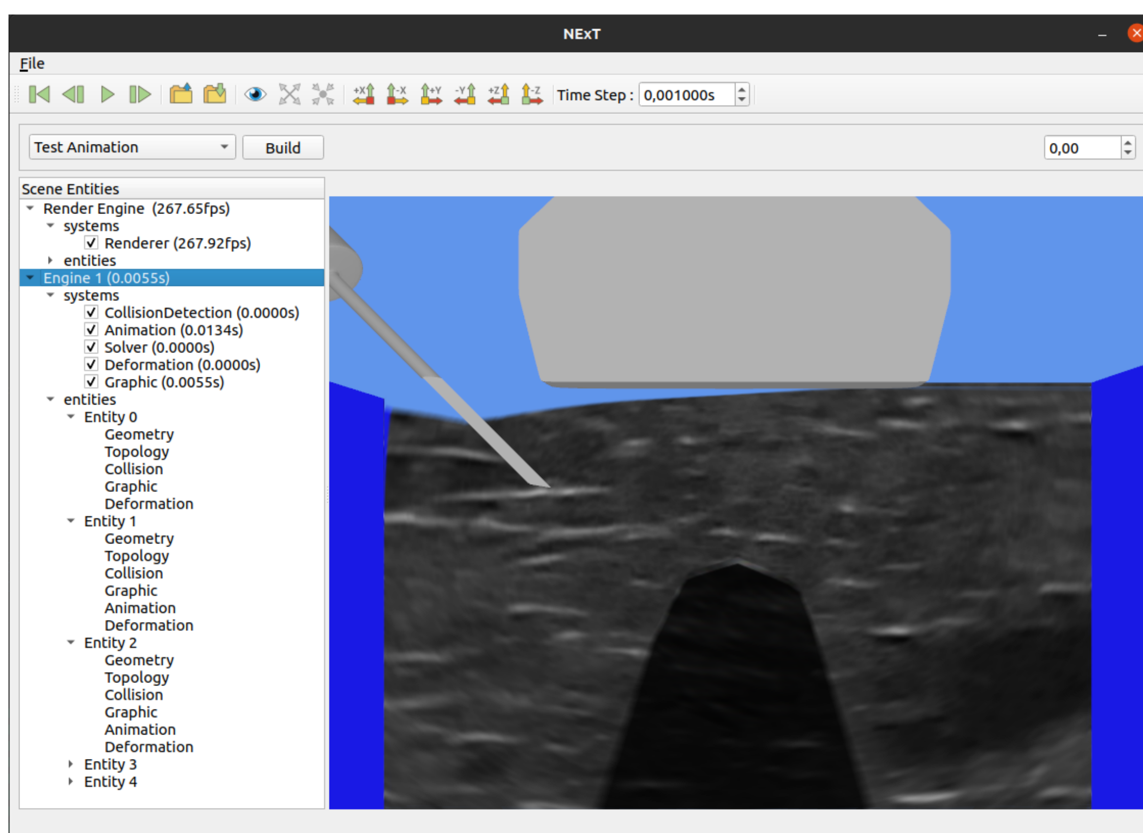


FIGURE 1.8 – Interface de l'application NExT

Le *viewer* est fait grâce à l'interfaceur graphique QT. L'application QT va permettre de lancer la construction d'une scène via son builder (cf. 1.3.3), une fois construite d'afficher le résultat du système de rendu (cf. 2.3). Il permet aussi de

— activer ou désactiver des paramètres pour chaque système ou composant (on peut par exemple désactiver les effets de post processing comme le *blur* ou la visibilité du composant graphique d'une des entités.

— faire avancer les systèmes (dynamique, animation, etc.) pas à pas ou en temps réel.

— revenir en arrière dans les systèmes qui le permettent (animation, déformation).

— faire des mouvements de caméra (rotations, translation, zoom).

1.3.4 Outils mathématiques et bibliothèques externes

Bibliothèques

L'application NExT intègre aussi quatre bibliothèque externes.
- Eigen, largement utilisée pour les outils mathématiques décrits plus tard.
- Haption pour la partie intégration des manettes haptiques.
- IGL pour les calculs sur géométrie.
- STM, une bibliothèque semi externe (faite pour next, mais n'est pas comprise dans la librairie NExT par défaut) qui permet de faire des calculs dynamiques avec des contraintes.

Eigen

Eigen fournit une grande variété d'outils mathématiques, principalement les calculs de matrices, très utiles pour la géométrie 3D. On utilise en effet tout le long du code principalement des `Eigen::Vector` et des `Eigen::Transform` qui sont des structures de données de vecteurs et matrices alignées en mémoire.

Matrices générales Eigen est une bibliothèque de matrices. Ces matrices sont des zones stockées comme des `std::array` à savoir des zones contiguës de mémoire pour les données. Par défaut, les matrices Eigen sont stockées en column-major, c'est-à-dire colonne par colonne. Par exemple la matrice

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

sera stockée en mémoire comme l'array $[1, 4, 7, 2, 5, 8, 3, 6, 9]$. Cela correspond à la manière dont openGL lit les matrices, il est donc possible de passer des `uniform Matrix4f` simplement grâce à `Eigen::Matrix4::data()` qui renvoie la zone mémoire où sont stockées les valeurs de la matrice.

Vector Les `Vector` principalement `Vector3` et `Vector4` sont matrices 3x1 ou 4x1. On les utilise pour stocker les positions 3D des points, les couleurs RGB ou RGBA de différents objets ou résultats de calculs.

Transform Les `Transform` sont un sous type de matrices carrées 4x4 permettant de décrire la position, rotation et taille d'objets 3D. Elle sont de la forme suivante

$$T = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & s_h \end{bmatrix}$$

avec R une matrice 3x3 représentant une rotation (et possiblement des changements de tailles différents selon les axes), T la translation (position) de l'objet. Quand on voudra calculer le résultat de l'application de la `transform` à un point A 3D, on le passera en coordonnées homogènes

$$A_h = \begin{bmatrix} A_x \\ A_y \\ A_z \\ 1 \end{bmatrix}$$

et on le multipliera à gauche par la matrice de transformation. La valeur s_h est usuellement 1, mais peut permettre de calculer un changement de taille uniforme selon les 3 directions. En effet, la 4ème valeur de A_h est égale à s_h après application de T, il suffit de multiplier les trois autres coordonnées de A_h par $1/s_h$ pour retrouver des coordonnées homogènes, l'objet a alors changé de taille.

Les `transform` peuvent se combiner en les multipliant, on pourra ainsi faire aisément des changements de repères. On fera attention à la non-commutativité des multiplications dans l'espace des matrices.

OpenGL

OpenGL est la bibliothèque graphique utilisée par le projet NExT dans le système de rendu (cf. 2.3). Elle permet de développer du rendu en *cross platform* car elle abstrait le *hardware*, notamment la carte graphique à travers l'utilisation des shaders *GLSL*. Dans NExT, OpenGL est utilisé pour dessiner des triangles, lignes et points dans l'espace.

Chapitre 2

Travaux

2.1 Mise en place d'une scène pour les développements du stage

Afin de pouvoir tester tous les développements futurs, la scène principale de la thèse de Charles Barnouin a été dupliquée, nettoyée et mise à jour pour correspondre aux développements du stage.

Elle contient un système de rendu dans un premier moteur et dans un second les systèmes *graphic*, *collision*, *dynamic*, *animation* et *deformation*. La scène originelle contenait un 3eme moteur pour le système haptique, mais il n'a pas été utilisable pour le stage (cf. 2.2).

Elle contient 5 entités.

- Deux maillages tétraédriques représentant 2 couches de peau de différentes rigidités pour les tests de déformation. Ils peuvent être soumis au système de déformation en tant qu'objet déformable (cf. 2.4).
- Un maillage représentant un os afin de tester les déformations proches d'un objet non-déformable.
- Une aiguille avec des composants d'animation et d'objets déformants.
- Une sonde échographique avec les mêmes composants.

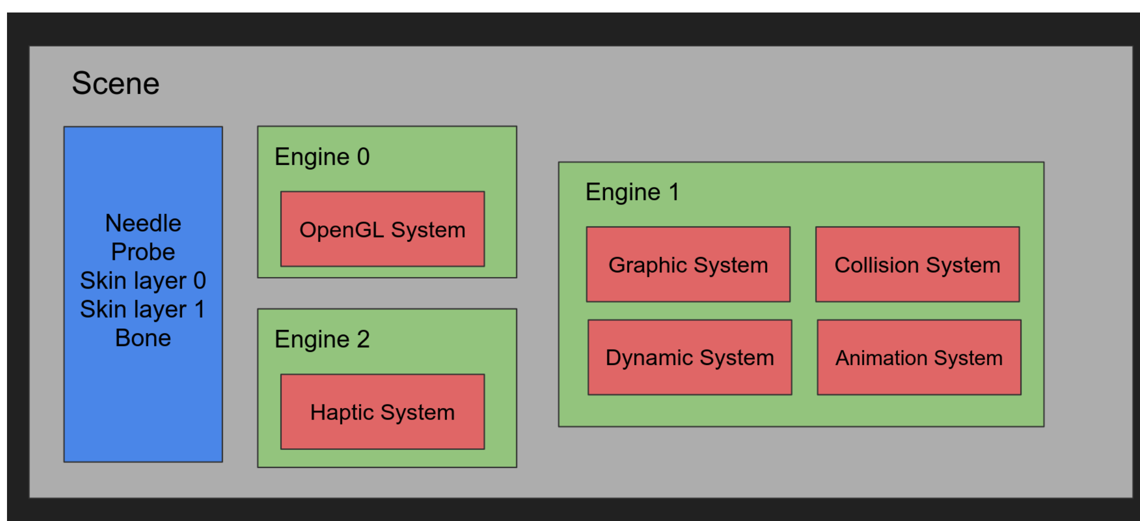


FIGURE 2.1 – Architecture ECS de la scène de travail

2.2 Système d'animation

Préambule : Intérêt

Le système d'animation est un système permettant de lire des fichiers ou de les enregistrer, afin de rejouer le déplacement d'une entité dans la scène 3D. Ce système a été mis en place pour plusieurs raisons.

Tout d'abord, un système permettant d'enregistrer et de relire une action faite aux manettes haptiques sera utile pour mettre en place le système d'évaluation de performance de lae praticien·ne lors de son geste. On pourra ainsi comparer les fichiers avec des fichiers cible et déterminer les plus grosses erreurs et rejouer les performances des apprenti·es pour permettre à des professeurs de les évaluer. Ce système permettra aussi de jouer des simulations explicatives sans avoir a stocker des vidéos, on pourra donc rejouer le geste d'un·e praticien·ne expérimenté·e pour un meilleur apprentissage.

Ensuite, les manettes haptiques utilisées par Charles Barnouin lors de sa thèse ont perdu leur support Linux avec la 20.04 de Ubuntu (elle fonctionnaient encore sous la 18.04). Par ailleurs ces manettes sont principalement faites pour une utilisation Windows. De plus, le stage étant en distanciel à cause de la pandémie de COVID-19 et les manettes coûtant chacune plusieurs milliers d'euros, il aurait été difficile de les faire sortir du laboratoire du LIRIS pour une utilisation à domicile. Enfin, même si les points précédents n'avaient pas posé de problèmes, l'utilisation de manettes ralenti considérablement la plupart des tests, car à chaque lancement de l'application, il faut connecter les manettes, ce qui peut prendre plusieurs dizaines de secondes.

En résumé, l'idée d'un tel système nous est venue très vite dès que l'on a compris qu'il nous faudrait tester le même mouvement en boucle pour les développements et qu'il était plus pratique de développer un système de la sorte.

Le but est d'avoir un système d'animation "à la blender/maya" où l'utilisateur·ice crée des `keyframe` (poses clefs) et le logiciel va interpoler pour déplacer l'objet entre les différentes keyframes.

2.2.1 Keyframes

Une structure de données de pose clef qui regroupe un temps (cf. 2.2.7) et une `transform`. Le but est qu'à ce temps précis, les positions et rotations de l'entité doivent correspondre à la `transform`.

2.2.2 Fichier d'animation et Reader

Le système d'animation permet d'enregistrer le mouvement d'une entité au cours du temps et/ou de relire cet enregistrement. Pour cela, un système de fichier interne a été mis en place. Ce fichier se présente comme dans l'exemple figurant en Figure 2.2. La documentation complète est disponible en annexe A.1.

Les lignes commençant par `#` sont des commentaires. On notera 3 parties :

— *Variables* : permet de définir les propriétés de l'animation comme le type d'interpolation ou le fait qu'elle puisse boucler.

— *Initial transform* : Une `transform` initiale. Toutes les *transforms* du fichier auront pour origine cette *transform* au lieu de $(0, 0, 0)$.

— Les keyframes de l'animation. Elles commencent par le temps (utilisateur) auquel la *keyframe* doit se passer, puis décrivent la position (translation). Enfin, elles comportent la rotation selon les 3 axes euleriens. Ces rotations ne sont pas écrites

```
# Variables
interpolationType polySin
repeat

# Initial Transform (scene placement) (translation - rotation)
1.0   0.0   0.0    0.0   0.0   0.0

# It is recommended to add keyframes in order
# Time      X       Y       Z      RX    RY    RZ
   0.0     0.0     0.0     0.0
   1.0     0.0     1.0     0.0
   2.0     0.0     1.0     0.0       0     0    90
   3.0     0.0     1.0     0.0       0     0     0
   4.0     0.0     0.0     0.0
```

FIGURE 2.2 – Fichier de d'animation démonstration

si elles sont identiques aux rotations des *keyframes* les précédant, cela permet de faire diminuer de beaucoup le poids de ces fichiers pour l'enregistrement, surtout pour les animations un peu longues.

Le fichier une fois lu sera sauvegardé sous forme d'`animator`.

```
# Automatically recorded animation file for entity

interpolationType linear

#    TIME            X          Y          Z          RX         RY         RZ
  0.00000       0.00000    0.00000    0.00000
  0.04167       0.00000    0.12501   -0.12501
  0.08333       0.00000    0.12505   -0.12505
  0.13282       0.00000    0.12513   -0.12513
  0.17819       0.00000    0.12523   -0.12523
  0.22559       0.00000    0.12537   -0.12537
  0.27088       0.00000    0.12553   -0.12553
  0.31639       0.00000    0.12571   -0.12571
  0.36224       0.00000    0.12592   -0.12592
  0.40854       0.00000    0.12617   -0.12617
  0.45368       0.00000    0.12643   -0.12643
  0.49951       0.00000    0.12672   -0.12672
  0.54572       0.00000    0.12703   -0.12703
  0.59124       0.00000    0.12736   -0.12736
```

FIGURE 2.3 – Fichier de d'animation enregistré par le système d'animation

2.2.3 Animator

L'`animator` est un ensemble de `keyframes` sous forme de `std::set`, un ensemble totalement ordonné. La relation d'ordre des `keyframes` est faite uniquement sur leur temps, la relation d'ordre est donc une relation de préordre car elle ne respecte pas l'antisymétrie. En effet, 2 `keyframes` différentes en terme de `transform` mais identiques en temps seront égales par rapport à la relation. Si deux `keyframes` égales en temps sont

insérées dans l'`animator`, il aura une *undefined behaviour* entre les `keyframes` précédents et suivants cette `keyframe`. Il aurait pu être décidé de faire qu'une `keyframe` ne puisse être ajoutée à l'animator que s'il ne possède pas de `keyframe` de même temps, mais en pratique le fichier de `keyframe` est petit et fait main sans vraiment de possibilités de se tromper, ou enregistré par le système d'animation, cette situation ne pouvant alors pas se produire.

La classe `animator`, une fois créée à l'ouverture de la scène, fonctionne comme une boîte noire par rapport aux `keyframes`, elle se comporte simplement comme une fonction qui, donné un temps, renvoie un `transform` (dans une `interpolated keyframe`).

2.2.4 Interpolation et *interpolated keyframe*

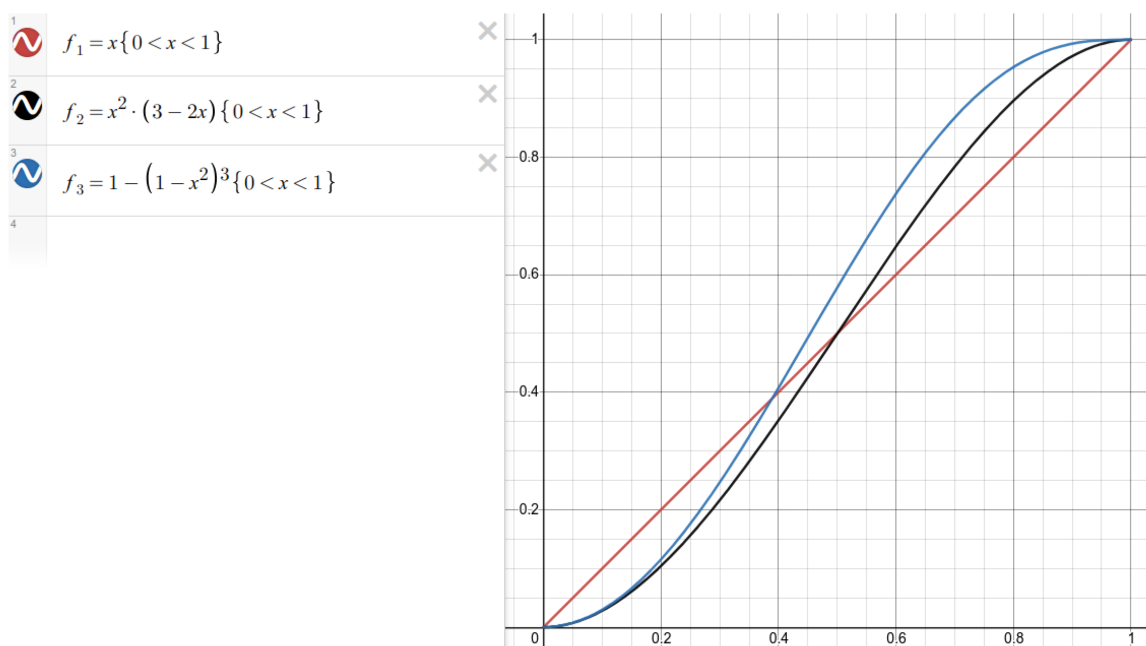


FIGURE 2.4 – Les 3 types d'interpolations implémentés dans NExT

Lors de la création, il faut choisir le type d'interpolation voulu parmi les 3 disponibles. Donné un temps, l'`animator` va interpoler les deux `keyframes` précédents et suivants ce temps. Le calcul de l'interpolation est ainsi :

Soit X l'`interpolated keyframe` résultante, A et B les `keyframes` à interpoler, sachant que $A < B$ (on s'en assure avec un `std::swap` si besoin). Posons a et b les `transforms` des `keyframes` à interpoler, x la `transform` résultante, et qa et qb leur rotation respective sous forme de quaternions.

On va chercher la `keyframe` interpolée du temps t. On cherche les `keyframes` supérieures et inférieures A et B les plus proches de t (dans le *set* ordonné, elles sont consécutives). On calcule $T = (t - a.time())/(b.time() - a.time())$. Ce T se trouve entre 0 et 1 et correspond au poids linéaire de chacune des `keyframes` dans l'interpolation. On va modifier T en fonction du type d'interpolation souhaité (cf. Figure 2.4).

— $T = T$ pour une interpolation linéaire (1)
— $T = T^2 * (3 - 2 * T)$ pour une interpolation dite *smoothstep* ou *sinus-like* (2)
— $T = 1 - (1 - T^2)^3$ pour une interpolation dite *human-like* (3)

On calcule ensuite la transform souhaitée :

$$\begin{aligned} &x.translation() = T * a.translation() + (1 - T) * b.translation();\\ &x.rotate(qa.slerp(T, qb)); \end{aligned} \tag{2.1}$$

On notera l'utilisation de la fonction `slerp` qui signifie *spherical linear interpolation* qui permet de conserver les tailles des objets lors des interpolations entre rotations. En effet, interpoler linéairement des quaternions provoque des pertes de volume des objets lors des rotations.

La Figure 2.4 est une représentation graphique des différents types d'interpolation.

- f_1 est l'interpolation linéaire, elle convient parfaitement quand le pas de temps entre les `keyframes` est petit (par exemple avec le système d'enregistrement) ou que l'on veut représenter des mouvements mécaniques.
- f_2 est symétrique autour de $(0.5, 0.5)$ et va ressembler à un sinus (pente amortie sur les bords)
- f_3 est un peu plus raide que f_2 au début, mais fait un quasi-plateau pour $x > 0.9$, ce qui donne un effet très proche du mouvement humain (un mouvement rapide en début et assez précis à la fin).

La classe `interpolated keyframe` résultante hérite de `keyframe` et possède 2 pointeurs sur `keyframe` correspondant aux 2 `keyframes` qu'elle interpole, ainsi que le type d'interpolation et la valeur T et une vélocité.

2.2.5 Vélocité

La vitesse d'un objet peut être calculée à partir d'une `interpolated keyframe`.

Soit δ_p la différence de position entre les `keyframes` A et B le long desquels X, l'`interpolated keyframe` a été interpolée et δ_t la différence de temps entre A et B.

Soit dt la pente de l'interpolation, on la calcule avec la dérivée de la formule d'interpolation.

- $dt = 1$ pour une interpolation linéaire (1)
- $dt = -6 * T * (T - 1)$ pour l'interpolation (2)
- $dt = 6 * T * (T^2 - 1)^2$ pour l'interpolation (3)

La vitesse de l'objet au moment de X est alors $\delta_p * dt/\delta_t$. Bien que la vitesse varie entre deux `keyframes` si l'interpolation n'est pas linéaire, l'objet se déplacera toujours en ligne droite, la vitesse est donc toujours orientée le long de δ_p.

2.2.6 Composant

Le composant d'animation existe sous quatre formes.

- Le composant de base sauvegardant la géométrie de base pour que le système d'animation puisse modifier la position de l'objet.
- Le composant d'animation en lecture, héritant du composant de base et y ajoutant un `animator`.
- Le composant d'enregistrement, héritant du composant de base et ajoutant les informations nécessaires à l'enregistrement, notamment le dernier moment d'enregistrement pour limiter la fréquence d'écriture ou encore la `transform` précédente pour la compression.
- Un composant de test dit "dual" qui hérite des deux composants précédents. Ce composant a principalement servi à tester le composant d'enregistrement grâce à l'animation du composant d'animation. On notera le problème dit problème du

diamant venant du fait que les composant d'enregistrement et de lecture héritent de la même classe. Il a alors fallu faire appel à de l'héritage virtuel pour éviter la duplication des données de la classe de base.

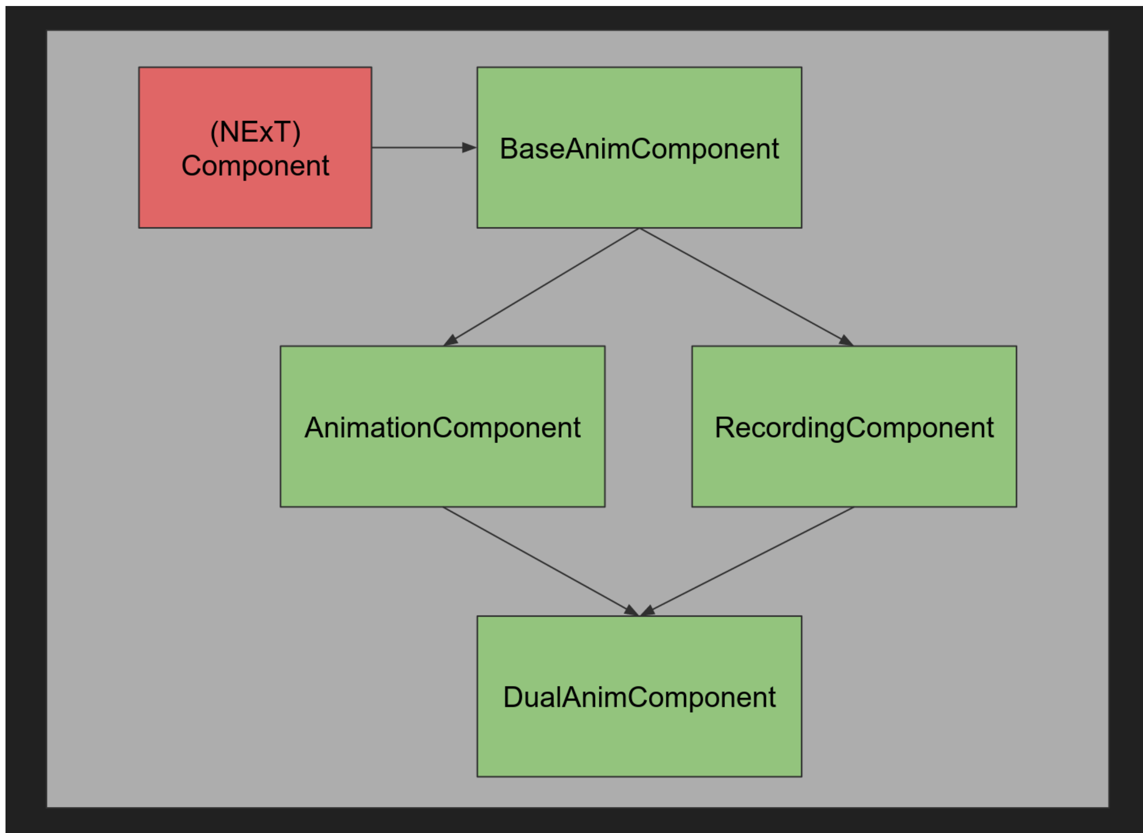


FIGURE 2.5 – Héritage des composants d'animation

2.2.7 Système

Une fois toute l'architecture mise en place avec le lecteur de fichier d'animation, l'`animator` et les `interpolated keyframe`, il suffit au système de gérer le temps utilisateur. Ce temps utilisateur avance indépendamment du taux de rafraîchissement du système, il avance d'un pas fixe si l'utilisateur·ice utilise la fonction `step` de l'interface, il avance en temps réel si le *viewer* est en mode `play`, et est en pause quand les systèmes n'avancent pas. Il calcule l'avancement dans le temps d'animation, met à jour les `animators` des entités animées, et récupérera les résultats des `animators`, calculés dans une `interpolated keyframe` pour l'appliquer au maillage.

Le maillage des objets animés, une fois créé dans la scène, est stocké en tant que maillage initial dans le composant. Ce maillage sera alors recalculé avec la `transform` à appliquer et replacé dans la scène.

Pour les objets en enregistrement, le système va récupérer la `transform` actuelle, soit avec le composant d'animation en lecture ou haptique, si l'entité les possède, soit en la déduisant depuis la position des points 3D et la position initiale grâce à un calcul de minimisation effectué grâce à la fonction `Eigen::unemaya` qui prend les deux ensembles de points et donne la `transform` la plus proche permettant de passer de l'un à l'autre. Une fois la `transform` calculée, il va créer la `keyframe` correspondante avec le temps actuel et la noter dans le fichier d'enregistrement. La fréquence de rafraîchissement de l'enregistreur

n'a pas besoin d'être plus haute que 24Hz (standard dans l'animation cinématographique 3D) pour avoir des résultats très bons, elle peut cependant être augmentée, ce qui aura pour conséquence d'augmenter la taille du fichier d'enregistrement. À partir de cette fréquence-là, l'interpolation linéaire est largement suffisante, utiliser une autre pourrait même donner un phénomène d'*à-coup*.

2.2.8 Résumé et extensions possibles

Le système d'animation est fonctionnel est l'état, par simple ajout du système dans n'importe quel *engine* et de composants sur les entités à enregistrer et/ou rejouer. Il permet d'enregistrer le déplacement d'objets contrôlés par le système haptique ou tout autre système qui fonctionne de manière similaire. Il permet aussi de rejouer tout enregistrement ou des fichiers édités à la main. Ce système a été développé entièrement pendant le stage, en se basant sur les idées du système haptique pour utiliser la même façon de faire, il peut totalement remplacer le système haptique si besoin, ou le complémenter.

Il pourrait être amélioré par rapport aux axes suivants.

— Le *scale* (troisième propriété des `transform`) n'a pas été implémenté dans les calculs d'interpolation. Il ne serait probablement pas très dur de le faire, mais il n'y en avait pas l'utilité dans le cadre du stage, et cela aurait demandé un certain temps pour les tests.

— Ajouter un système d'édition du fichier d'animation directement dans l'application. Écrire les fichiers d'animation à la main ne pose pas de soucis pour des animations de quelques mouvements, mais peut poser problème pour faire certaines choses assez précises. Implémenter une interface d'édition d'animation pourrait être intéressant pour certains cas, mais il faudrait alors ajouter des systèmes permettant de déplacer les objets à la main dans le *viewport* ce qui n'est pas non plus le but principal de l'application.

— Ajouter des interpolation et des déplacement non-linéaires entre les `keyframes`. Cela pourrait aider pour donner un mouvement circulaire à un objet à la main sans calculer trop de `transforms`.

— Render les déplacements automatisables selon une formule, on pourrait par exemple vouloir faire se déplacer un objet selon une parabole, il faudrait alors rajouter un système d'interpréteur de formules dans le fichier de lecture d'animation.

2.3 Système de Rendu

2.3.1 Analyse du système avant travaux

Le système de rendu est le système permettant l'affichage. À chaque `step()`, il produira une nouvelle image avec les dernières informations fournies au composant graphique (cf. 1.3.2). Il utilise OpenGL pour générer ses images. C'est aussi ce système qui va, grâce à OpenGL, faire tous les calculs liés à l'échographie et une certaine partie de la déformation (cf. 2.3.3).

OpenGL, dans son utilisation classique, va dessiner des objets. Pour ce faire, il prend un ensemble triangles (points et indices) et va déterminer pour chaque pixel le triangle le plus proche pour en récupérer les informations (couleur) et l'afficher. L'image résultat est stockée dans un `framebuffer`, à savoir une image texture avec un `z-buffer` qui détermine la profondeur de chaque pixel (à quelle distance sont les triangles dessinés de la caméra pour chaque pixel).

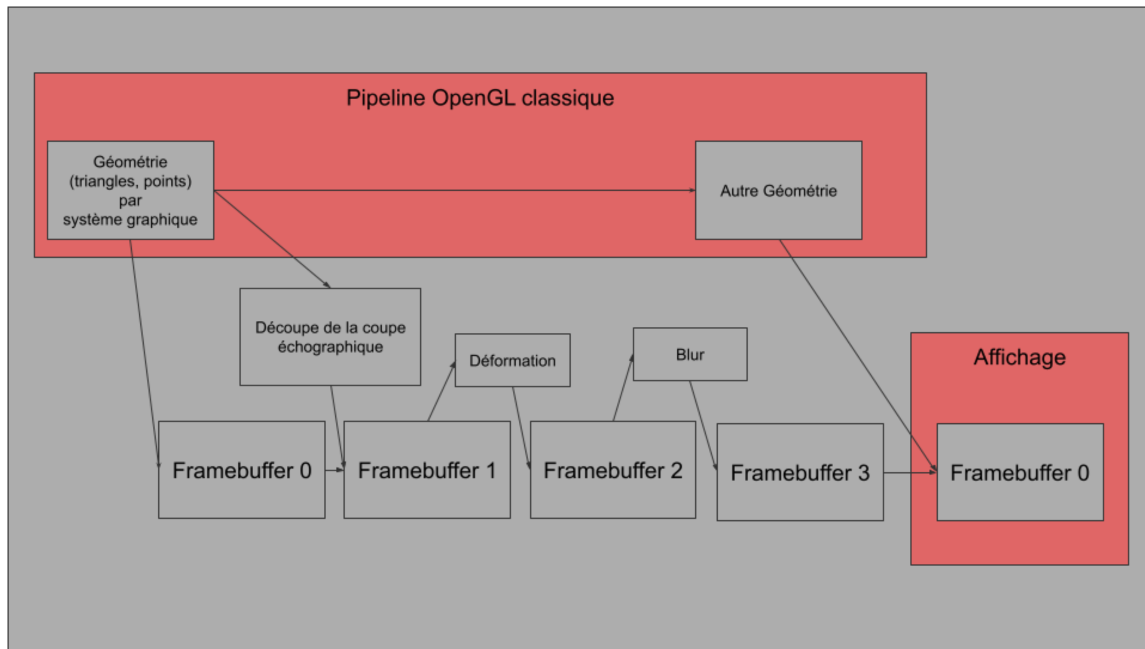


FIGURE 2.6 – Fonctionnement de la pipeline avant le stage

Pour faire des effets de *post processing* (post traitement), on va utiliser le `framebuffer` précédemment dessiné contenant la géométrie et on va rendre par-dessus un quadrilatère ou un grand triangle (ici un quadrilatère) qui va couvrir tous les pixels de la caméra, ainsi, la pipeline classique OpenGL peut être utilisée pour faire faire des calculs à la carte graphique en chaque pixel.

Le `z-buffer` est aussi utilisé pour faire des ombres dans l'application, ici une seconde caméra se trouvant au niveau de la sonde va "regarder" en projection perspective. On ne lui donne alors que les objets qui projettent des ombres. Ainsi, le résultat de son `z-buffer` nous permet de déterminer si un objet est en trajectoire de la lumière ou derrière un objet bloquant la lumière.

C'est selon ces trois grands principes que le système de rendu de l'application NExT a été développé. On peut voir en figure 2.6 le fonctionnement du système de rendu en début de stage. L'application commence par dessiner les objets 3D pleins, puis en réinitialisant le `z-buffer`, va dessiner la coupe échographique sur laquelle il va calquer la texture échographique. Les plans qui donnent un effet "extrusion" à la coupe échographique sont aussi générés par des `draws OpenGL` successifs. Le rendu peut être observé en Figure 1.8.

Cette manière de mettre en place la coupe échographique a été discutée en 1.2.3 et a pour bénéfice d'être rapide et adaptable à n'importe quelle scène.

Les triangles et points discutés plus tôt sont calculés par l'application dans le `system` (ECS) **graphic**, ce qui permet d'éviter de recalculer les positions des vertex à chaque frame. Ce système est aussi plus généralement le seul à communiquer des informations au système de rendu, les autres systèmes passent donc par le composant graphique des entités s'ils ont besoin de donner des informations. Ce détail est assez important, car c'est le système graphique qui va faire tourner toute l'application, et non le système de rendu comme il est habituel que ce soit le cas dans les petites applications.

La pipeline, telle quelle, est fonctionelle, mais très peu maintenable et il est complexe de développer quoi que ce soit en l'état. Les parties suivantes vont décrire la refonte de

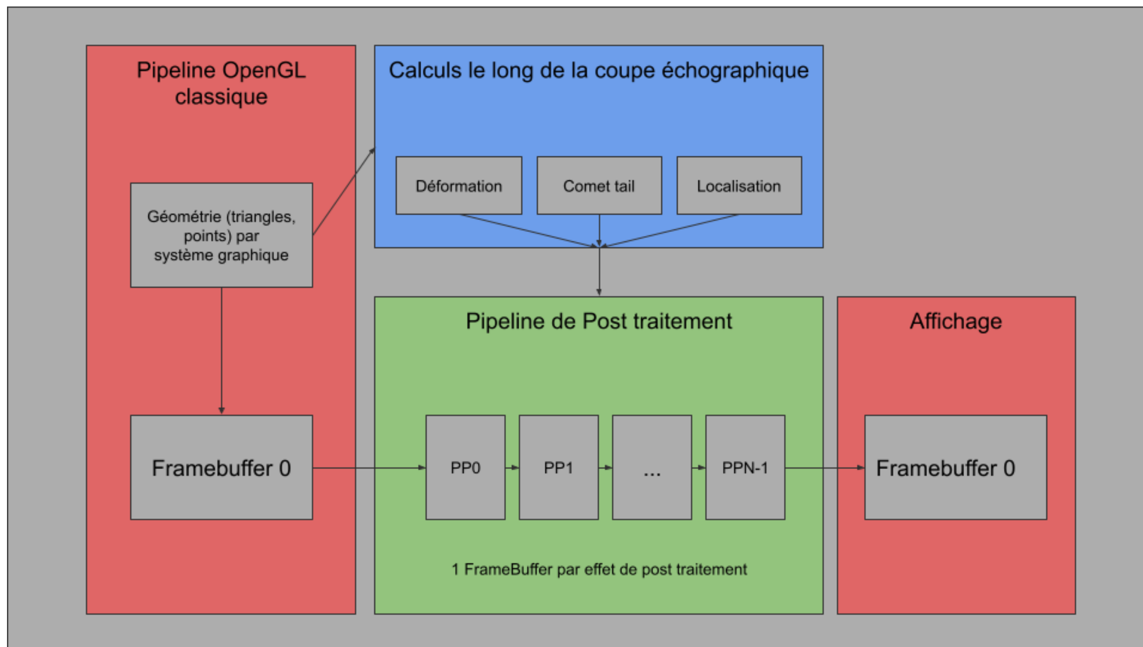


FIGURE 2.7 – Résumé du fonctionnement de la pipeline graphique en fin de stage

ce système, comment elle a été abordée et quel est l'état final de ce système.

2.3.2 Identification des 2 étapes de calculs

Lors du refactoring de la pipeline graphique, il a fallu mettre en place un grand nombre d'automatisations. Pour les shaders openGL, on différencie 2 types de calculs principaux : calculs le long de la coupe échographique et calculs sur la texture de rendu (*screenspace*). Pour les premiers, des calculs supplémentaires ont été mis en place. Premièrement, au lieu de stocker un `Vector3 position` dans le composant graphique, on utilise la `transform` de l'objet, on peut ainsi en déduire la direction de la sonde échographique. Grâce à cette `transform`, les calculs dits "le long de la coupe échographique" fonctionnent même si la sonde effectue une rotation. Les seconds sont des effets de post traitement appliqués à l'image du rendu. Ils permettent notamment d'afficher les résultats calculés lors des calculs le long de la coupe échographique. La Figure 2.7 résume la pipeline graphique à la fin du stage.

2.3.3 Calculs mis en place dans la première étape

Les calculs de localisation de la coupe échographique

Ces calculs permettent de faire une correspondance entre les pixels de la coupe écho et les pixels de la texture affichée à l'écran. Le résultat permet entre autre de limiter les effets - notamment les effets de post traitement comme le *blur* (cf. 2.3.5) - uniquement à la coupe échographique.

Calculs liés au comet tail

L'effet *comet tail* appliqué sur la texture écran par un effet de post process (cf. 2.3.5). Ce calcul est décrit en annexe A.5. Il cherche à simuler l'effet en Figure 2.8. Pour ce faire, un effet sera calculé pour chaque point donné au shader comme "point de comet tail".

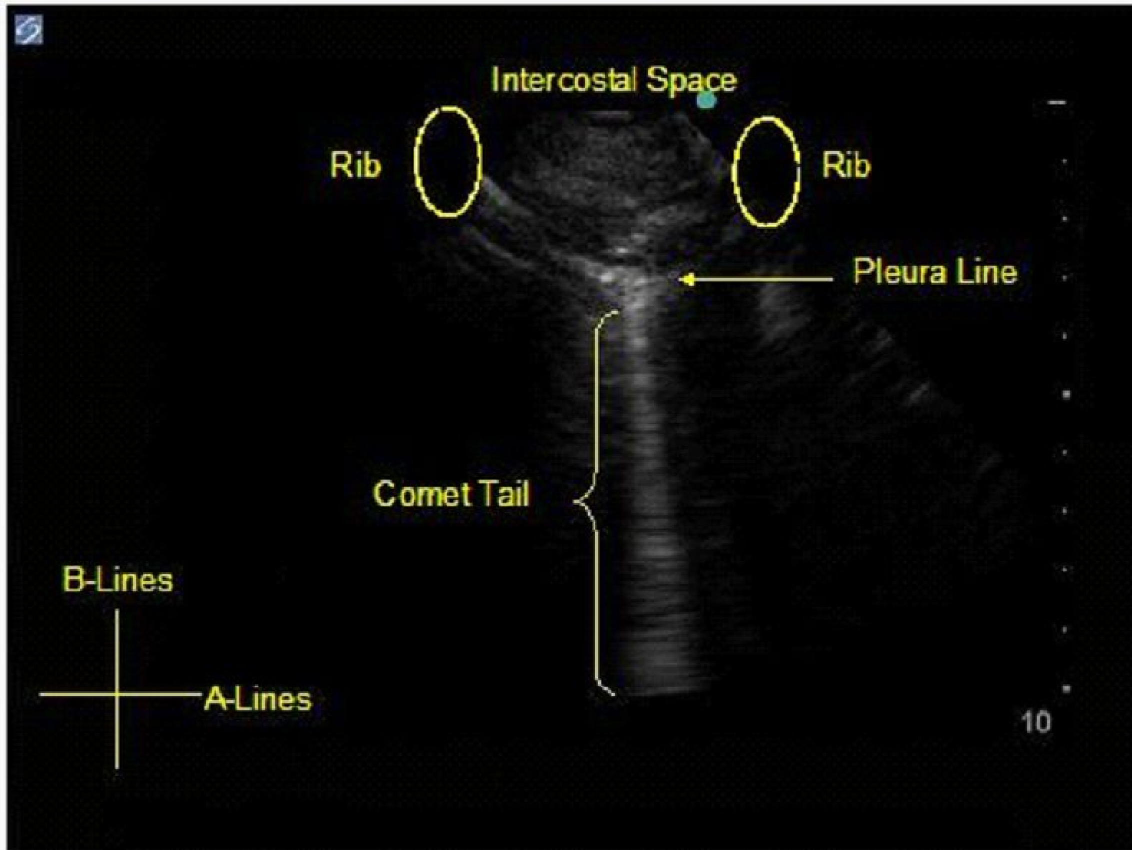


FIGURE 2.8 – Effet "Comet tail" de référence [source : [5]]

Cet effet est directement dans l'axe de la sonde échographique, et va faire un effet de rayure blanche sous le point. Cet effet est simulé par deux *smoothstep* autour de la pointe de l'aiguille (selon le plan normal à l'axe échographique), qui est atténué par un sinus le long de l'axe échographique et un *smoothstep* jusqu'à une certaine distance.

Déformation le long de la coupe échographique

Une première ébauche de déformation a été mise en place directement grâce à l'API openGL. Le but est de déformer directement les pixels de texture sur la coupe échographique au lieu de déformer un maillage. Il y a plusieurs avantages à faire ainsi :

- La résolution du maillage peut être grossière à certains endroits, ce qui peut poser des problèmes, si par exemple la pointe très fine de l'aiguille vient déformer entre deux vertex. En déformant les pixels, on a une résolution extrêmement fine sur la déformation.
- Les calculs sont faits directement sur des shaders, ce qui est très rapide comparé à un déplacement des points du maillages s'il est très fin.

On utilise cette technique aussi pour s'abstenir d'un système de déformation dynamique (déjà en place dans NExT) car on veut une application qui tourne en temps réel, la déformation étant un calcul parmi beaucoup d'autres et qu'il faut une immersion totale de l'apprenant, on privilégie les performances et le visuel au réalisme mécanique.

Cette déformation est calculée en 2 parties : tout d'abord, on va calculer la quantité de déformation de chaque pixel, et ensuite le shader de post traitement de *displacement* va appliquer cette déformation sur la texture de l'écran.

La texture utilisée pour les résultats de la déformation est une texture 3D, permettant

de stocker indépendamment les résultats de déformation induits par différents objets. On peut ainsi combiner les différentes déformations dans le shader de *displacement*, cela a permis de mettre en place les déformations de l'insertion de l'aiguille et de l'appui de la sonde en simultané. Cette déformation ne prend pas en compte la raideur des matériaux (cf. 2.3.5 paragraphe *displacement* pour la prise en compte de ce calcul).

Pour la déformation due à l'aiguille, on a mis en place un effet temporaire de déformation en attendant les résultats fournis par le LBMC (cf. 1.2.2). Le modèle n'a pas encore été mis en place dans l'application car il nécessite encore des travaux, le modèle de déformation actuel a donc été conservé temporairement.

Cet effet est décrit en détail en annexe A.2. C'est un effet qui perd de la force avec la

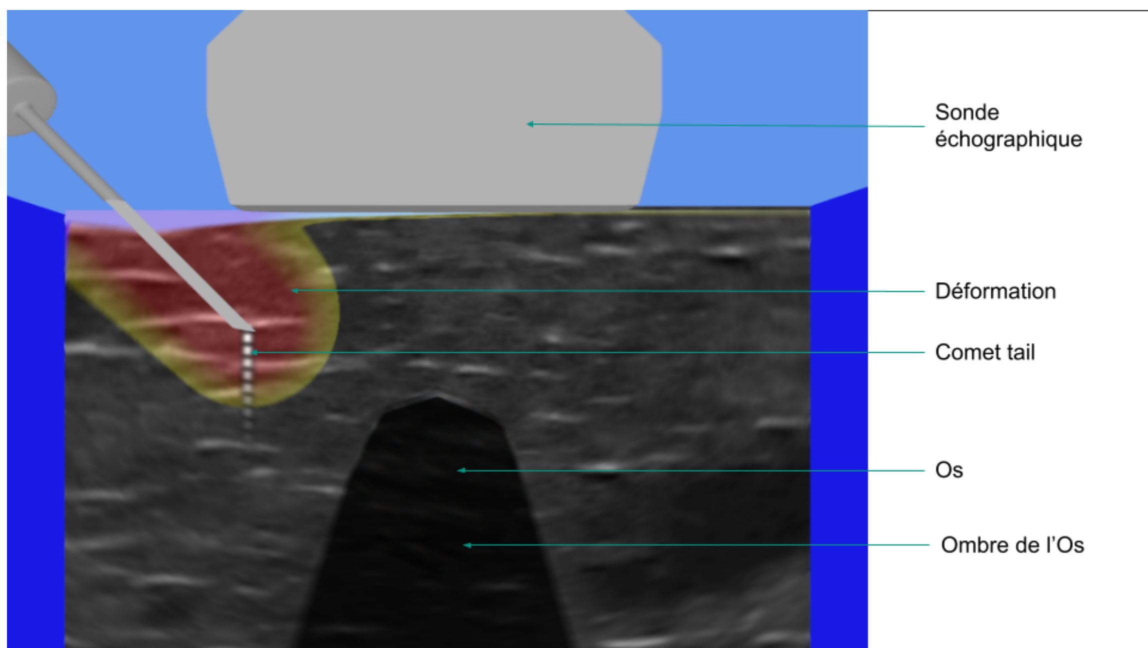


FIGURE 2.9 – Effets de rendu

distance par rapport à l'aiguille, de manière circulaire face à la pointe et cylindrique le long de l'aiguille. Cet effet a pour direction le sens de l'insertion.

2.3.4 Mise en place de la pipeline de post traitement

La seconde vague de calculs effectués par des shaders a été regroupée sous le titre d'"effets de post traitement". Ces effets ayant beaucoup de caractéristiques communes, il a été possible de développer une pipeline de post traitement. Concrètement, grâce à cette pipeline, il est possible de rajouter de nouveaux effets de post traitement en seulement 2 étapes :

— Créer un nouveau `program` *OpenGL*, avec `createPostProcessProgram()` en spécifiant les noms des fichiers sources des *vertex* et *fragment shaders* et `uniforms` des textures à prendre en compte.

— Ajouter les effets à calculer au pipeline (un programme créé n'est pas obligatoirement ajouté au pipeline, par exemple si l'utilisateur·ice le désactive à travers l'interface) dans un `std::vector` et y ajouter les différents `uniforms` (les `uniforms` texture étant déjà définis à la création du programme) et les textures.

La pipeline gère alors automatiquement la compilation, le *linking* des shaders, la création des textures de rendu et les rendus successifs.

Pour les rendus successifs d'effets de post traitement, il faut à chaque effet redessiner l'image pour ajouter l'effet. Deux solutions sont alors possibles, mettre en place 2 textures et rendre alternativement dans l'une et l'autre ou alors créer un nombre de textures égal au nombre maximum d'effets possibles pour rendre successivement dans chaque texture. La seconde solution a été choisie, car elle est non significativement impactante en terme de mémoire, mais elle permet de récupérer les résultats intermédiaires si besoin, ce qui peut rendre le débug bien plus facile tant que l'application est en cours de développement, le code est aussi plus simple à comprendre avec cette solution. On notera que la pipeline permet de modifier facilement la texture à laquelle on applique les effets de post traitement. Actuellement, elle n'est utilisée que sur `GL_FRAMEBUFFER_0` en entrée et sortie, à savoir la texture affichée à l'écran à la fin du *draw OpenGL*, mais elle pourrait aussi être utilisée sur d'autres textures pour leur appliquer des effets comme des flou (*blur*).

2.3.5 Effets de post traitements en place

Actuellement 3 effets de post traitement sont mis en place dans l'application.

Displacement

L'effet de *displacement* qui va déplacer les pixels de la texture de l'échographie en accord avec les résultats des calculs du shader de déformation. Cet effet est agrémenté d'une *heatmap* permettant de visualiser la force de déformation en chaque pixel. Cet effet est limité à la texture échographique grâce au résultat du shader de localisation de la coupe échographique dans le *screenspace*.

La déformation des tissus dépend de leur raideur. Un système basique a été mis en place, avec pour idée de le remplacer plus tard par les équations de déformation des solides, avec par exemple le module de Young, pour avoir des résultats plus réalistes. Dans ce système, chaque objet déformable a sa propre valeur de résistance à la déformation, 0 étant objet non-déformable (os) et 1 étant objet le plus déformable (calibré à la vue). On va donner à chaque pixel la valeur de déformation du tissu lié à ce pixel en projetant la texture échographique sur les objets 3D. On va ensuite se servir de cette valeur pour réduire la déformation des pixels et de leurs alentours.

Une fois la déformation finale calculée, on vient reporter la valeur du pixel qui est censé se retrouver à cette place pour donner l'effet de déformation de la texture. Les calculs sont détaillés en annexe A.3.

Cet effet a malheureusement des effets de bord, soit il est limité à la coupe échographique et alors la partie au échographique au-dessus de la sonde est ignorée, soit il n'y est pas limité et alors il va déformer dans tout le champ de déformation des objets, coupe échographique ou non. Cependant ce second effet de bord ne se voit pas si la coupe est face à la caméra (comme en Figure 2.10).

Blur

L'effet de *blur*, lui aussi limité à la coupe échographique, permet un meilleur rendu de la coupe. Les textures ont besoin d'être floutées pour rendre de manière plus réaliste, les calculs n'étant pas de vrais lancés de rayon. Un *blur gaussien* a été mis en place dans un premier temps, mais l'implémentation d'un *blur circulant* ou *spinning blur*, qui floute circulairement autour d'un pivot (ici la sonde échographique) donne un rendu plus proche des rendus échographiques réels, car la propagation des ondes dans les tissus est toujours bruitée. Cet effet est calculé en moyennant de manière pondérée les pixels qui sont le long

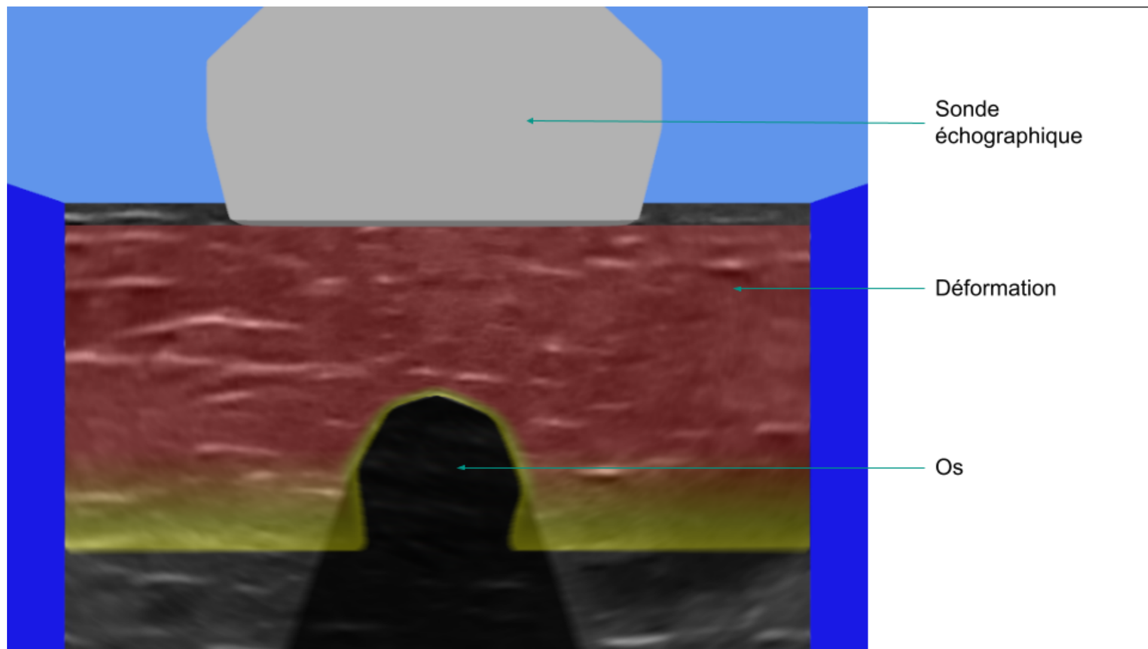


FIGURE 2.10 – Déformation partiellement bloquée par l'os

d'un cercle de centre le pivot (pixel précis) passant par le point à flouter. Les effets de *blur* sont visibles en Figure 2.11. Les calculs sont disponibles en annexe A.4.

Comet tail

L'effet dit *Comet Tail* [6], qui simule un artefact échographique lié au rebond successif des ondes sonores entre 2 parois propres d'un ordre de grandeur de la longueur d'onde émise. Ceci se produit notamment au bout de l'aiguille. L'effet est visible sur la Figure 2.9.

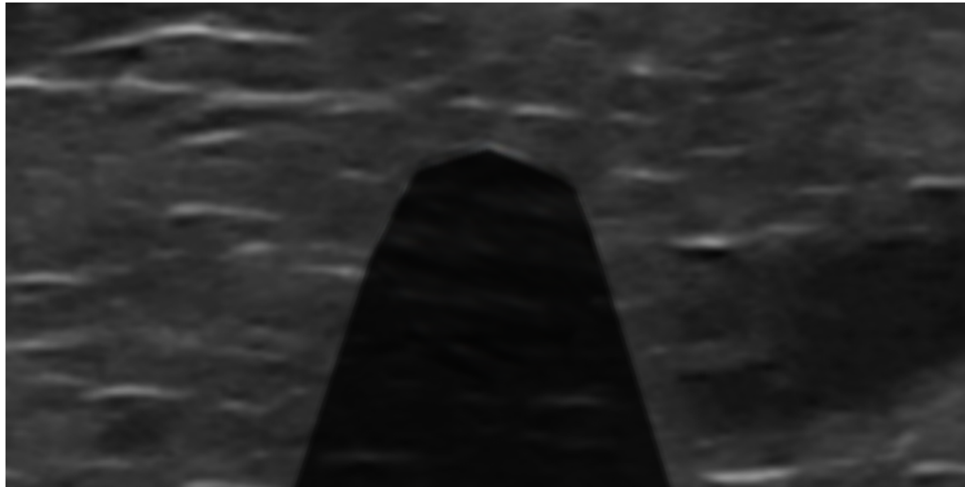
Cet effet peut apparaître sous plusieurs formes. Si l'onde sonore rebondit dans un fluide, comme une poche d'eau, il n'y aura que très peu d'atténuation, les ondes vont alors remonter successivement jusqu'à la sonde et donner un effet dit *short path reverberation*. S'il y a beaucoup d'atténuation comme proche de la pointe de l'aiguille, l'effet s'atténue et on parle alors d'effet *comet tail*.

Les effets de déplacement et de *comet tail* sont importants à implémenter car au-delà de l'aspect graphique, ce sont des effets réels qui permettent aux praticien·nes de repérer la pointe de l'aiguille lorsqu'elle n'est pas tout à fait dans l'axe de la coupe échographique.

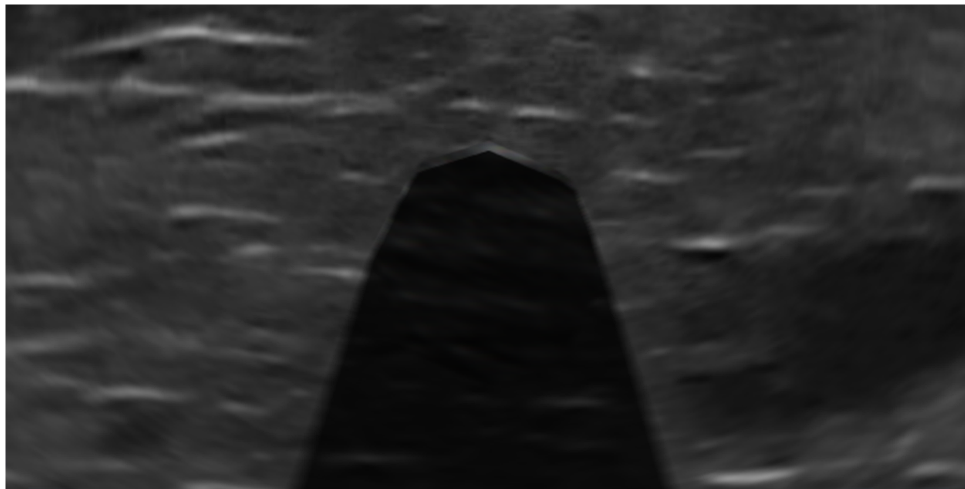
2.3.6 Réalisations

Originellement, le système ne servait qu'aux calculs OpenGL. Cependant, ce système en particulier a été dénaturé par les différents développements. On y trouvait beaucoup de code en dur, sans structures de données, des magic number, beaucoup de copié-collé de code et autres *code smell*. Un certain temps du stage a été dépensé pour corriger cette partie de code et rétablir en partie le design pattern originel de l'application, sans quoi le travail sur cette partie du code était trop laborieux.

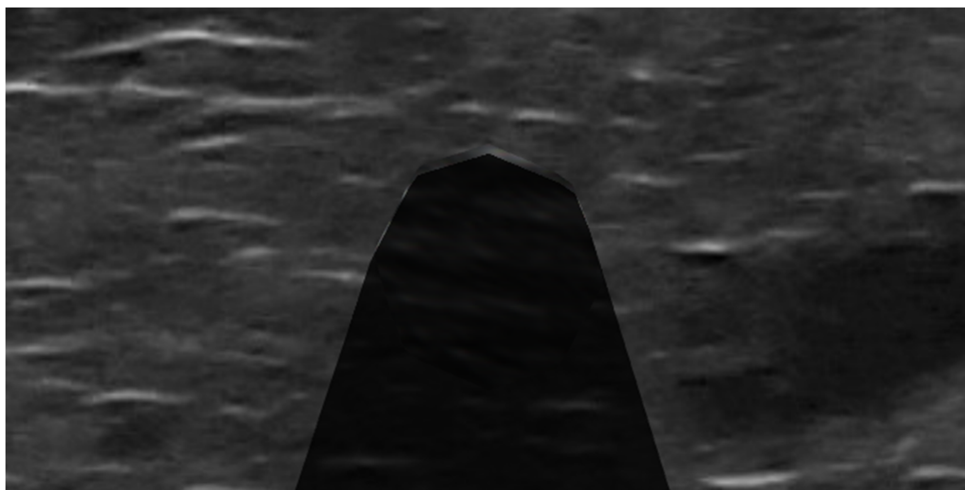
Une certaine modernisation d'openGL a aussi été effectuée pour la simplicité de code et les prochain·es développeur·euses qui auront pour mission de retravailler cette partie de l'application. Par exemple l'utilisation de glTexStorage (OpenGL 4.1) au lieu de glTexImage (OpenGL 2.0), quand judicieux, permet une bien meilleure compréhension du code pour la création de textures.



(a) Kernel Blur



(b) Spinning Blur



(c) Blur désactivé

FIGURE 2.11 – Différents effets de *blur* sur la coupe échographique

Un travail de nettoyage de code important a aussi été mis en place pour rectifier les modifications des formules en dur pour compenser des erreurs de code. Le code était très instable, à chaque modification, le code ne fonctionnait plus. Il fallait alors trouver les liens mal faits ou les valeurs en dur qui en étaient responsables, une énorme partie du débug était nécessaire pour le développement dans cette base de code. Par exemple, les textures mises en place pour calculer un champ de déformation étaient en *unsigned byte*, or les résultats étant des vecteurs de directions signés, ils ne rentraient pas dans la texture, les vecteurs avaient alors été normalisés entre -0.5 et 0.5 puis ajouté 0.5 pour avoir un résultat positif normalisé pour entrer dans la texture. Cela posait des problèmes, car la force de déformation était clampée entre deux valeurs arbitraires et on avait de la perte de précision. Une fois les textures passées en `GL_FLOAT_32`, les équations correctes ont pu être mises en place, produisant un code beaucoup plus cohérent et plus général, pour cet exemple.

Enfin, un travail de fouille de code a été nécessaire, car beaucoup des travaux ont été laissés tels quels, avec beaucoup de lignes commentées pour avoir des résultats à montrer, comme des tests de développement pour des démonstration ou des images pour des articles. Il a alors fallu déterminer quelles lignes étaient celles qui correspondaient aux recherches les plus avancées et quels étaient les autres blocs de code commenté qui leur correspondaient.

Pour la suite des travaux sur ce système, il serait intéressant de découpler la coupe échographique du rendu direct. En effet, dans tout le reste de l'application, le système de rendu ne sert qu'à afficher ce que le composant `graphic` lui donne, il serait alors logique de faire la même chose, créer un autre système utilisant OpenGL, mais pas pour les calculs de rendu, simplement pour effectuer la partie relative à la coupe échographique et ses effets. Ainsi, le système de rendu pourrait se contenter de faire l'affichage là où le système `graphic` ferait les calculs, comme le voudrait le design originel de l'application.

2.4 Système de déformation

Un système de déformation à part entière a été mis en place pour remplacer ou compléter le système de déformation (cf. 2.3.3) selon les cas. Ce système comporte 2 types de composants, objets déformables et objets déformants. L'idée de ce système est, tout comme la déformation directement dans la coupe échographique, d'éviter de mettre en place le système dynamique qui va chercher à calculer les équations de la physique, mais est bien trop lent pour l'applicatif du projet SPARTE. Cependant, contrairement au système de déformation sur coupe échographique, ce système déforme bien le maillage, ce qui permet la prise en compte des déformations par tous les autres systèmes et non pas simplement pour l'affichage. L'idée de ce système est, comme pour la déformation de coupe, de remplacer le système dynamique pour les déformation, et ainsi éviter les calculs éléments finis trop long pour l'applicatif temps réel.

Ce système va, à chaque étape, générer des champs de déformation 3D depuis les objets déformants et ensuite les appliquer au maillage des objets déformés. L'avantage comparé au système de déformation de la coupe est que le maillage est réellement déformé avant la coupe openGL, on gagne en quelque sorte un "niveau" d'antériorité dans les calculs comparé à la déformation qui fait seulement parti du rendu. De plus, cette déformation sera alors incluse dans les calculs de collision, qui avant se faisaient sur le maillage non déformé. En outre, ce système est capable de supporter n'importe quelle déformation,

comme le méta-modèle que fournira le LBMC (cf. 1.2.2) dans le futur, et l'appliquera sur tous les points du maillage, là où la déformation de la coupe échographique n'est déformée que selon une coupe dans la fonctionnelle de déformation 3D. On aura alors un contrôle plus fin du moment de pénétration qui sera important dans l'implémentation du méta-modèle, là où la déformation sur la coupe échographique ne le permettait pas.

Plusieurs choix de calculs ont été mis en évidence. Soit redéplacer le maillage initial à chaque étape, soit calculer la différence de déformation par rapport à l'étape précédente et l'appliquer aux points déjà déplacés. Les deux méthodes viennent avec leurs inconvénients : la première ayant un certain nombre d'effets de bord (si l'objet déformant se déplace beaucoup, les premiers points ne seront plus influencés), la seconde induit une dérive au cours du temps. La première solution a été implémentée et convient aux résultats souhaités dans le cadre du stage.

Dans les deux cas, la précision du maillage doit être assez élevée, sinon les objets déformants comme l'aiguille pourrait passer entre les *vertices* et la déformation ne serait pas très réaliste.

Ce système a été développé tard dans le stage, il existe sous forme de *proof of concept*. Les textures échographiques ne sont pas déformées par ce système pour l'instant, car elles ne sont pas reliées au maillage, mais simplement posées dans la scène 3D. On peut utiliser les deux systèmes de déformation en parallèle pour déformer la texture (cf. 2.3.3), il suffit de s'assurer que les modèles de déformation coïncident.

Chapitre 3

Bilan du stage

3.1 Méthode de Travail

3.1.1 Réunions

Le travail court terme s'est articulé à l'image de la méthode *scrum* adaptée à une équipe de un développeur et un consultant (Fabrice) au sein de l'équipe ORIGAMI. Nous avons aussi fait des réunions mensuelles avec le LBMC (cf. 1.2.2) pour mettre en commun nos objectifs et orienter nos recherches.

Le rythme moyen des réunions avec Fabrice Jaillet a été de 2.5 réunions par semaine, adaptées à l'avancement des divers développements et des problèmes rencontrés. Ces réunions ont eu lieu en distanciel, comme le reste du stage, excepté deux qui ont eu lieu dans les locaux en fin de stage. Les réunions au sein d'ORIGAMI ont tourné autour des deux axes qui suivent.

Faire le point sur les travaux développés depuis la dernière réunion Ainsi, Fabrice a pu me faire des retours sur les méthodes designs pattern utilisées, et j'ai pu lui décrire comment le code était avant et après mes développements. Nous discutions alors des développements supplémentaires possibles ou des nécessités de reprise des développements, ainsi que des limites de ce qui a été fait.

Discuter des prochains développements prévus à court et moyen terme Ceci inclut la mise en place de nouveaux systèmes ou de reprises de code nécessaires pour ajouter de nouvelles features et corrections de bugs.

C'est lors de cette étape que Fabrice m'informait des travaux déjà effectués, s'il y en avait, et des grandes lignes de l'état du code par rapport à ces développements.

C'est aussi à ce moment de Fabrice m'a aidé pendant les *brainstorming* pour les idées de développements et leur réalisation. Cela a été très utile pour m'éviter des recherches sur des sujets trop complexes pour le stage, et pour développer des choses importantes pour la suite du projet.

3.1.2 Développement

En termes de recherche et de développement, ma méthode de travail a été la suivante :

— Trouver un objectif court terme à développer pour mettre en oeuvre les idées dont nous avions discuté durant les réunions.

— Chercher dans le code les travaux en rapport avec cet objectif, s'il y en avait au préalable. Si c'était le cas, remettre le code d'aplomb en terme de design pattern et de qualité de code, pour permettre un développement optimal des améliorations à apporter à cette partie du code et des futurs travaux.
— Une fois le code prêt à l'accueillir, développer l'objectif courant. C'est lors de cette étape que les idées sont parfois remises en question et réévaluées pour correspondre au mieux au logiciel NExT et aux intégrations futures. La phase de test a toujours été effectuée avec minutie pour réduire la dette technique générale de l'application et permettre aux prochain·es chercheur·euses des développement les plus simples possibles.

3.2 Conclusion

Le stage s'est concentré sur la réalisation du système d'animation, la remise en place de la pipeline graphique avec OpenGL et le développement d'outils pour cette dernière, comme la pipeline de post traitement, et du développement des features d'immersions des utilisateur·ices comme les effets de déformations ou les effets graphiques liés à la coupe échographique. Les développements ont étés faits de sorte à ce qu'ils soient le plus utile possibles aux successeur·euses des mes travaux.

Les principales difficultés rencontrées pendant le stage provenaient d'une base de code instable et non maintenable. Il a été un des objectifs principaux de ne pas ajouter un niveau d'instabilité à la base de code, mais au contraire de simplifier et nettoyer au maximum les fichiers pour les développements du stage et futurs. Une fois le système de rendu remis en place, les développements ont pu avoir lieu dans un environnement bien plus agréable et modulaire qu'auparavant. Les systèmes d'animation et de déformation ont été fait de manière à s'intégrer parfaitement dans le reste de l'application et à ne posent aucun problème de performance.

Ce stage m'aura personnellement permis d'apprendre beaucoup. D'une part, les designs patterns relatifs à la base de code sont des designs de conceptions complexes et très intéressants qui me permettent de penser et de développer mes idées différemment. D'autre part, travailler sur une base de code aussi complexe m'a permis d'apprendre beaucoup sur les paradigmes avancés de la programmation, sur les nombreux goulots d'étranglement des applicatifs complexes qui posent alors des problèmes de performance, et le travail en groupe en général. Enfin, ce stage m'aura permis de mettre un pied dans le domaine de la recherche informatique, de comprendre comment les sujets sont travaillés et étudiés, puis mis en place dans un logiciel concret à la fois pour tester les hypothèses et mettre à l'épreuve les calculs et idées.

Bibliographie

[1] Société d'imagerie MUSCULO-SQUELETTIQUE. *Ultrasound of the Shoulder*. 2018. URL : https://www.youtube.com/playlist?list=PLGV2jHWN573fIsiTBrGp_1mfkKw3l-kN9.

[2] Charles BARNOUIN. *Outil pédagogique de ponction des grosses articulations sous échographie*. 2020. URL : https://perso.liris.cnrs.fr/fabrice.jaillet/data/PhD-CBarnouin-2020.pdf.

[3] Charles BARNOUIN, Florence ZARA et Fabrice JAILLET. *A real-time ultrasound rendering with model-based tissue deformation for needle insertion*. 2020. URL : https://hal.archives-ouvertes.fr/hal-02415740.

[4] Ma de los ANGELES ALAMILLA-DANIEL. *Development of a haptic simulator for practicing the intraarticular needle injection under echography*. 2020. URL : https://tel.archives-ouvertes.fr/tel-03078578/document.

[5] Douglas T. SUMMERFIELD et Bruce D. JOHNSON. *Lung Ultrasound Comet Tails — Technique and Clinical Significance*. 2012. URL : https://www.intechopen.com/chapters/44070.

[6] James RIPPEY. *Commet tail Artifact*. 2020. URL : https://litfl.com/comet-tail-artefact/.

Annexe A

Code et documentation

A.1 Documentation fichier d'animation

Animation configuration file

Example

```
# Variables
interpolationType polySin
repeat

# Initial Transform (scene placement) (translation - rotation)
1.0   0.0   0.0    0.0   0.0   0.0

# It is recommended to add keyframes in order
# Time      X      Y       Z     RX    RY   RZ
   0.0    0.0    0.0    0.0
   1.0    0.0    1.0    0.0
   2.0    0.0    1.0    0.0      0    0    90
   3.0    0.0    1.0    0.0      0    0     0
   4.0    0.0    0.0    0.0
```

Variables

A set of variables can be set whereever in the file but in the blocks. Syntax must be exact.

Interpolation type

Interpolation type between two KeyFrames.

Interpolation Variables

- `interpolationType` : sets following value to both animator (default : `linear`). overwritten locally by the two other variables.

Interpolation Values

- `linear` : sets interpolation to linear.
- `polySin` : sets interpolation to sinusoid like betwen two keyframes.
- `polyPlateau` : sets interpolation to pseudo linéar at the beginning of the frame and with a "stop plateau" before reaching next keyframe. Most natural way.

Repeat

If `repeat` is present, animation will loop starting at time 0 modulo max time of keyframes.

Transforms

The file is composed of a lines fomatted this way :

```
time x y z rz ry rz
```

with `time` the time of the key frame, `x, y, z` the position (relative to object instanciation position) and optionnal `rx, ry, rz` eulerian rotations (if not specified current frame will copy last frame's).

Init transform

File must have a init transform (that will be apply on top of animation) for stuff like scene set up. By default this can be `0 0 0`.

Comments

Lines starting with a `#` are ignored. They dont break the block. Following block is valid :

```
# Comment

t x y z
# Comment
t x y z
# Comment
```

A.2 deformation.frag

```
#version 330

layout(location = 0) out vec4 defoNeedle;
layout(location = 1) out vec4 defoProbe;

in vec3 vertPosition;

uniform vec3 tipNeedle;
uniform vec3 directionNeedle;
uniform float velocityNeedle;

uniform vec3 tipProbe;
uniform vec3 directionProbe;
uniform float pushLength;

// influence radius of the needle
const float r = 0.04;

void deformationNeedle()
{
    vec3 BA = vertPosition - tipNeedle;
    if (length(directionNeedle) == 0)
    {
        defoNeedle = vec4(vec3(0.0), 1.0);
        return;
    }
    vec3 direction = normalize(directionNeedle);
    float d = dot(BA, direction);

    float dist = d > 0 ? length(BA) :
        length(cross(BA, normalize(direction)));
    float t = dist / r;
    float influence = smoothstep(1.0, 0.0, sqrt(t));

    float strength = 3000.0f;
    vec3 deformation = velocityNeedle * direction
        * strength * influence;
    defoNeedle = vec4(deformation, 1.0);
}

void deformationProbe()
{
    vec3 BA = vertPosition - tipProbe;
    if (length(directionProbe) == 0)
    {
        defoProbe = vec4(vec3(0.0), 1.0);
        return;
```

```
    }
    vec3 direction = normalize(directionProbe);
    float d = dot(BA, direction);

    float diffusion_factor = 10.0;
    float influence = clamp(1.0 - diffusion_factor * d, 0.0, 1.0);

    float strength = 1.0;
    vec3 deformation = direction * strength *
        pushLength * influence;
    defoProbe = vec4(deformation, 1.0);
}

void main()
{
    deformationNeedle();
    deformationProbe();
}
```

A.3 displacement.frag

```
#version 330 core

out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D screenTexture;
uniform sampler2DArray displacementTexture;
uniform sampler2D maskTexture;
uniform sampler2D maskEchoTexture;
uniform float deplacementPixel;
uniform bool drawHeatMap;

void colorDisplacedPixels(inout vec3 pixelColor, vec2 strength)
{
    // blue, green, red
    vec3 color[3] = vec3[](vec3(1.0, 1.0, 0.0), vec3(0.5, 0.5, 0.0),
        vec3(0.5, 0.0, 0.0));

    float value = length(strength) / 4.0;
    value = min(value, 2.0);

    int idx1 = int(floor(value));
    vec3 heatmap = mix(color[idx1], color[idx1 + 1], value - idx1).rgb;

    if (idx1 > 0.3) // Heatmap display condition
        pixelColor += heatmap / 2.0;
}

// Check the surrounding pixels for tissues information : example if a
//   bone is nearby, the soft tissue will not move much.
float checkSurroundings(vec2 texelSize)
{
    float value = 0.0;
    int numberNeigbour = 30;
    vec2 direction = vec2(1.0, 0.0) * texelSize;
    vec2 directionOrtho = vec2(direction.y, -direction.x);
    for (int i = -numberNeigbour / 2; i < numberNeigbour / 2; i++)
    {
        for (int j = -numberNeigbour / 2; j < numberNeigbour / 2; j++)
        {
            vec2 coord = TexCoords + i * direction + j * directionOrtho;
            value += clamp(texture(maskTexture, coord).x, 0.0, 1.0);
        }
    }

    float meanValue = value / (numberNeigbour * numberNeigbour);
```

```
    return 1.0 - meanValue;
}

void main()
{
    vec3 pixelColor = vec3(0.0);

    vec2 texelSize = 1.0 / textureSize(screenTexture, 0);

    vec2 displacement = vec2(0.0);
    if (texture(maskEchoTexture, TexCoords) != vec4(0.0))
    {
        vec2 deformation = vec2(0.0);
        deformation += texture(displacementTexture, vec3(TexCoords, 0)).yz;
        deformation += texture(displacementTexture, vec3(TexCoords, 1)).yz;

        float resistance = checkSurroundings(texelSize);

        displacement = resistance * deformation;
    }

    vec2 displacementValue = displacement * texelSize;
    vec2 texFetchCoord =
        clamp(TexCoords - displacementValue, vec2(0.0), vec2(1.0));
    pixelColor += texture(screenTexture, texFetchCoord).rgb;

    if (drawHeatMap)
        colorDisplacedPixels(pixelColor, displacement);

    FragColor = vec4(pixelColor, 1.0);
}
```

A.4 blur.frag

```
#version 330 core

out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D screenTexture;
uniform sampler2D echoMaskTexture;

uniform bool clampToEcho;

const float offset = 1.0 / 300.0;
const float pi = 3.14159265359;
const float degree = pi / 180.0;

const int n = 6;

vec2 Rotate(vec2 texcoord, float alpha)
{
    mat2 rotation = mat2(
        cos(alpha), sin(alpha),
        -sin(alpha), cos(alpha)
    );

    vec2 centerOfRotation = vec2(0.5) + vec2(0.0);
    vec2 newtexcoord = rotation * (texcoord - centerOfRotation)
        + centerOfRotation;
    return newtexcoord;
}

vec4 kernelBlur()
{
    vec2 offsets[9] = vec2[](
        vec2(-offset,  offset), // top-left
        vec2( 0.0f,    offset), // top-center
        vec2( offset,  offset), // top-right
        vec2(-offset,  0.0f),   // center-left
        vec2( 0.0f,    0.0f),   // center-center
        vec2( offset,  0.0f),   // center-right
        vec2(-offset, -offset), // bottom-left
        vec2( 0.0f,   -offset), // bottom-center
        vec2( offset, -offset)  // bottom-right
    );

    float kernel[9] = float[](
        1.0 / 16, 2.0 / 16, 1.0 / 16,
        2.0 / 16, 4.0 / 16, 2.0 / 16,
```

```
        1.0 / 16, 2.0 / 16, 1.0 / 16
    );

    vec4 sampleTex[9];
    for (int i = 0; i < 9; i++)
        sampleTex[i] = texture(screenTexture, TexCoords.st + offsets[i]);

    vec4 col = vec4(0.0);
    for (int i = 0; i < 9; i++)
        col += sampleTex[i] * kernel[i];

    return col;
}

vec4 spinningBlur()
{
    vec4 blurColor = vec4(0.0);
    // check if in echo !!
    int sampleNumber = 0;
    for (int i = -n; i < n; i++)
    {
        vec2 pixelCoord = Rotate(TexCoords, degree * i / n);
        vec4 value = texture(screenTexture, pixelCoord);
        bool isInEchoPlane = texture(echoMaskTexture, pixelCoord).x > 0.0;
        if (!clampToEcho || isInEchoPlane)
        {
            sampleNumber++;
            blurColor += value;
        }
    }
    blurColor /= sampleNumber;

    float texStrength = 0.0;
    return texStrength * texture(screenTexture, TexCoords)
        + (1 - texStrength) * blurColor;
}

void main()
{
    if (clampToEcho && texture(echoMaskTexture, TexCoords).x < 1.0)
        FragColor = texture(screenTexture, TexCoords);
    else
        FragColor = spinningBlur();
}
```

A.5 cometTail.frag

```
#version 330

out vec4 FragColor;

in vec3 vertPosition;

uniform mat4 echoTransform;
#define NB_POINTS_MAX 10
uniform vec3 points[NB_POINTS_MAX];
uniform int nbPoints;

const float M_PI = 3.14159265359;
const float cometTailWidth = 0.003;
const float cometTailDepth = 0.05;
const float nbOscillationToFixedDistance = 300.0;
const float strength = 1.0;

void main()
{
    mat3 t = mat3(echoTransform);
    vec3 probeDirection = t * vec3(0.0, 0.0, -1.0);
    vec3 probePlaneDirection = t * vec3(0.0, 1.0, 0.0);
    vec3 probePlaneOrhtoDirection = t * vec3(1.0, 0.0, 0.0);

    vec3 color = vec3(0.0);
    float dissipationMin = 0.0;
    for (int i = 0; i < nbPoints; ++i)
    {
        vec3 diff = vertPosition - points[i];
        float z = dot(diff, probeDirection);
        float y = abs(dot(diff, probePlaneDirection))
            + abs(dot(diff, probePlaneOrhtoDirection));

        float xDissipation =
            (1.0 - smoothstep(0.0, 1.0, z / cometTailDepth))
            * float(z > 0);
        float yDissipation = 1.0 -
            smoothstep(0.0, 1.0, y / cometTailWidth);
        float dissipation = xDissipation * yDissipation;

        if (dissipation > dissipationMin)
        {
            dissipationMin = dissipation;

            float sinEffect = 1.0 -
                (0.5 * cos(2.0 * M_PI * nbOscillationToFixedDistance * z)
                    + 0.5);
```

```
            sinEffect = 1.0 - sinEffect * sinEffect
                * sinEffect * sinEffect;

            color = strength * sinEffect * dissipation
                * vec3(1.0, 1.0, 1.0);
        }
    }
    FragColor = vec4(color, dissipationMin);

}
```

Annexe B

Autre

B.1 Écriture inclusive

Ce document utilise de l'écriture inclusive.

Un document complet est disponible ici ou téléchargable depuis ce lien **[ministère chargé de l'égalité entre les femmes et les hommes, de la diversité et de l'égalité des chances]**. Ce document est l'oeuvre de l'agence de communication Mots-Clés.

Les règles utilisées dans ce document sont les suivantes

- L'utilisation du point médian "·" pour dégenrer les noms et adjectifs. exemples :
  - "praticien·ne" signifie "praticienne ou praticien"
  - "praticien·nes" signifie "praticiennes et praticiens"

  On accorde aussi en genre neutre grâce au point médian (exemples : iel a été accueilli·e, les prochain·es développeur·euses)
- Les néologismes comme "lae" (prononcé "laé") sont des articles neutres (exemple : lae praticien·ne) et "iel" un pronom neutre.