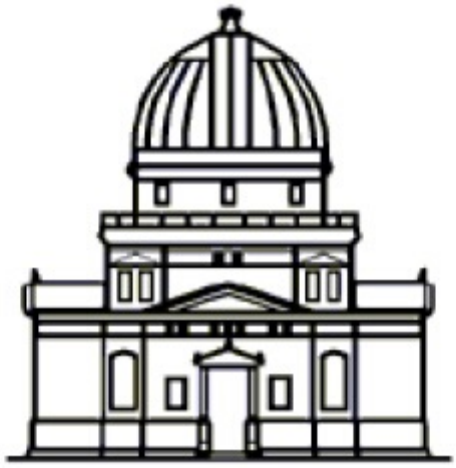
IUT Nancy Charlemagne
Université de Lorraine
2 ter Boulevard Charlemagne
54052 Nancy Cedex
Dépt. Informatique

Visualisation 3D de données astronomiques dans un navigateur Web, plus aspect Réalité Virtuelle

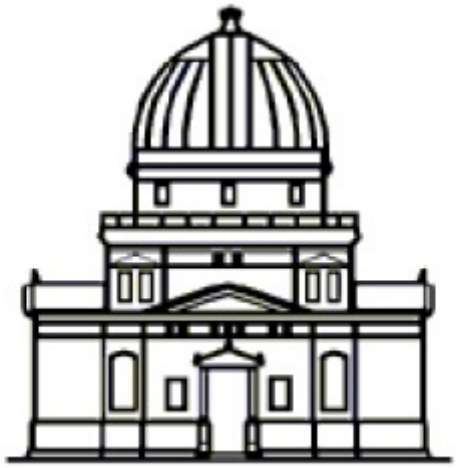


Rapport de stage DUT informatique
Observatoire astronomique de Strasbourg

Jérôme DESROZIERS
Promotion 2015-2016

IUT Nancy Charlemagne
Université de Lorraine
2 ter Boulevard Charlemagne
54052 Nancy Cedex
Dépt. Informatique

Visualisation 3D de données astronomiques dans un navigateur Web, plus aspect Réalité Virtuelle



Rapport de stage DUT informatique
Observatoire Astronomique de Strasbourg
11 rue de l'Université - 67000 Strasbourg

Jérôme DESROZIERS
Tuteur en entreprise: Mr. André SCHAAFF
Parrain du stage: Mr. Denis ROEGEL

Remerciements

Tout d'abord, je tiens à remercier l'ensemble du personnel de l'Observatoire de Strasbourg pour son accueil chaleureux, et plus particulièrement André Schaaff, mon tuteur, dont les indications m'auront été précieuses tout au long de mon stage, ainsi que Mark Allen, directeur du Centre de Données Astronomiques (CDS), pour m'avoir accepté en tant que stagiaire.

Je tiens aussi à remercier les membres du CDS, notamment Gilles Landais, en charge de la base de données de VizieR, qui m'a offert ses conseils lorsque j'ai mis en place la fonctionnalité de chargement des données réelles sur l'application; ainsi que Thomas Boch qui m'a aiguillé sur les protocoles d'accès à ces données.

Merci aussi à Nicolas Gillet et Nicolas Deparis, thésards à l'Observatoire, qui ont activement échangé avec moi et contribué à tester l'application tout au long de son développement, me permettant d'y apporter de nombreuses corrections.

Enfin, je remercie Hervé Wozniak, actuel directeur de l'Observatoire, pour m'avoir permis d'assister aux nombreuses conférences qui y ont eu lieu durant la période de mon stage.

Table des matières

Introduction

Lors de l'année 2015, deux stagiaires (Arnaud Steinmetz, en seconde année de DUT, puis Pierre Lespingal, en 2eme année d'école d'ingénieur) ont successivement travaillé à l'Observatoire de Strasbourg à la réalisation d'une application permettant de visualiser des données astronomiques (issues de simulations) sur un navigateur web.

L'objectif de mon stage est de reprendre le développement de cette application, afin d'en enrichir les fonctionnalités, mais aussi d'y apporter des corrections (optimisation, correctifs).

Le déroulement de mon travail fut donc découpé en plusieurs phases, qui peuvent se recouper en 4 catégories:

- Étude de l'application existante

- Amélioration de fonctionnalités existantes

- Mise en place de nouvelles fonctionnalités

- Tests et correctifs

Comme mon travail sur l'application m'a fait alterner ces phases de façon assez régulière, j'ai jugé qu'il était plus pertinent de rédiger mon compte-rendu en découpant ce dernier par catégories, plutôt que de les énumérer par ordre chronologique.

Néanmoins, un diagramme se trouve en annexe et permet d'avoir une meilleure visibilité sur la façon dont le stage s'est déroulé semaine après semaine.
(Note: d'après le diagramme, l'étude de l'existant ne couvre que le début de la première semaine, mais elle s'est en réalité étendue jusqu'à la 5eme semaine du stage et au delà à mesure que j'ai effectué les différentes corrections).

Présentation de l'entreprise

Fondé en 1881 - après que l'Alsace-Lorraine eut été cédée à la Prusse - l 'Observatoire Astronomique de Strasbourg se situe dans la partie historique de la ville et est entouré par le Jardin botanique de l'Université de Strasbourg (cf Figure 1, une photographie de la grande coupole de l'Observatoire).

Figure 1

L'Observatoire dépend du CNRS et emploie actuellement 80 personnes environ (dont 11 astronomes), et son directeur depuis 2009 est Hervé Wozniak.
Il est un acteur majeur de l'IVOA (International Virtual Observatory Alliance), un consortium dont l'objectif est de déterminer les standards utilisés en terme de création et de manipulation de données astronomiques.

Il est structuré en trois équipes de recherche:
- L'équipe Galaxies, qui se concentre sur l'étude de la formation et de l'évolution des galaxies
-L'équipe Hautes Énergies, qui s'intéresse aux sources émettrices en rayons X, aux objets compacts (comme les naines blanches) et aux noyaux de galaxies
-Le Centre de données astronomiques de Strasbourg (CDS), un ensemble de services

permettant d'accéder à un large éventail d'informations provenant d'études diverses liées à l'astronomie.

Ces services comprennent notamment 3 logiciels développés par l'Observatoire et libres d'accès sur son site web:

- Aladin: qui est un atlas interactif du ciel, permet de visualiser la voûte céleste grâce à des recoupements d'images provenant de diverses observations astronomiques.

- Simbad: qui est un service possédant une base de données centrée sur les objets astronomiques (comètes, nuages de gaz, etc...), et qui permet d'effectuer des requêtes sur cette base en se basant sur le nom, la position, ou certaines propriétés physiques de ces objets. Actuellement le nombre d'objets répertoriés avoisine 8,3 millions; et le nombre d'identifiants qui leur est associé dépasse 23 millions.

- VizieR: qui est aussi un service axé sur une base de données, se distingue grandement de Simbad dans la mesure où il permet d'effectuer des requêtes non plus en se focalisant sur un objet précis mais en prenant comme base un ou plusieurs catalogues (comme le Sloan Digital Sky Survey par exemple). L'avantage offert par rapport à Simbad est qu'en se focalisant sur un catalogue, toutes les données que l'on récupère sont au même format.
Il faut en effet savoir que la même information, voir le même objet, peuvent porter plusieurs dizaines de noms si l'on désire couvrir l'ensemble des catalogues.
A ce jour le nombre de catalogues répertoriés est d'environ 15000.

I) Étude de l'existant

La première partie de mon stage fut consacrée à l'étude de l'application existante, afin d'en comprendre le fonctionnement pour pouvoir y implémenter mes propres ajouts par la suite.

J'ai tout d'abord lu les rapports de stage et la documentation fournise par Pierre Lespingal et Arnaud Steinmet, puis ai utilisé l'application elle-même afin d'en comprendre l'utilisation.

A) Description de l'application

Cette application a pour but de réaliser l'affichage en 3D dans un navigateur Web de données issues de simulations astronomiques, qu'elle représente sous la forme de nuages de points. Elle permet notamment de se déplacer au sein de ces données, de leur appliquer des filtres de couleur pour mettre en valeur certains attributs et de réaliser des animations entre plusieurs jeux de données mis bouts à bouts.

Figure 2 - une capture d'écran de l'application

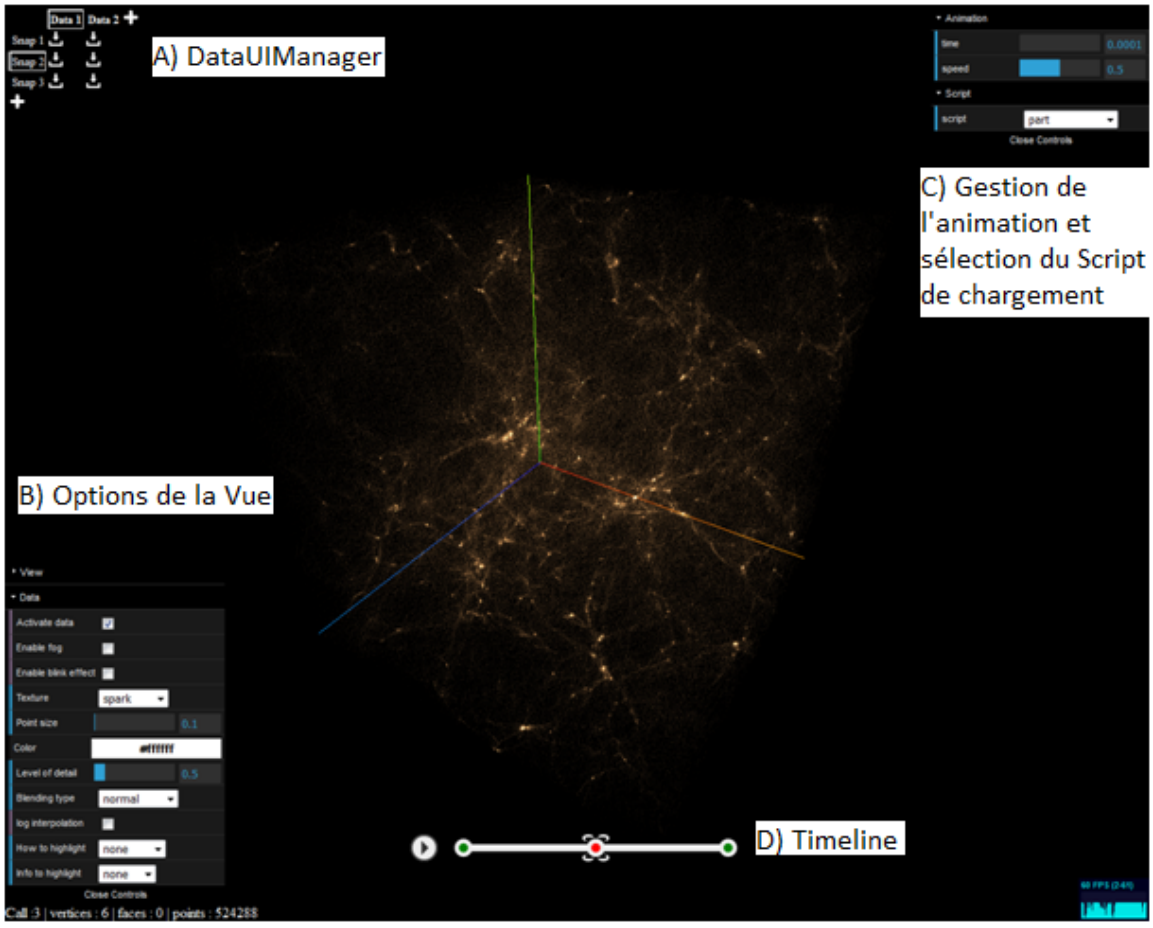


Figure 2

L'application, réalisée en JavaScript, s'utilise via un navigateur et offre diverses fonctionnalités (A, B, C et D se réfèrent à la Figure 2):

a) Charger des données en mémoire, grâce au menu situé en haut à gauche ainsi qu'au menu situé en haut à droite, en A).

b) Gérer l'affichage des données chargées, grâce au menu situé en bas à gauche, en B) (on peut par exemple gérer le niveau de détail des nuages de points, sur une échelle de 1 à 4; u niveau de détail de 1 n'affichant qu'un quart des points)

c) Réaliser une animation en assemblant à la suite les différents jeux de données chargés, grâce à la Timeline située en bas au centre, en D) ainsi qu'au menu situé en haut à droite, en C).

Elle utilise de nombreuses librairies, les trois plus utilisées étant dat.gui, async.js et three.js.

- dat.gui permet entre autre de gérer facilement les interactions entre l'interface de l'application et cette dernière (à la manière d'un modèle Vue-Contrôleur)

- async.js permet de réaliser n fois la même action en asynchrone, puis d'appeler une fonction donnée lorsque ces actions ont toutes été effectuées.

- three.js constitue quand à elle le noyau de l'application; cette librairie permet de gérer l'affichage et l'animation en temps réel d'objets 3D sur navigateur.
Son utilisation se base principalement sur 4 types d'objets:
-Mesh (défini à partir d'un objet Material et d'un objet Geometry) , Material représente la texture du Mesh (comment ce dernier réagit aux sources de lumière, etc...) et Geometry contient l'ensemble des vecteurs permettant de modéliser l'objet Mesh.
Détail important: les nuages de points affichés par l'application sont, au niveau de three.js, des objets un peu spéciaux dans le sens où ce sont des Mesh "faits à la main" et pas des objets préfabriqués par la librairie (comme des cubes ou des cylindres); pour les afficher on doit créer différent buffers manuellement et effectuer un appel à une fonction nommée drawCall(indice, nombre) qui affichera chaque point du nuage ayant un indice compris entre indice et indice+nombre-1.
-Raycaster (le principe du raycasting consiste à tracer une demi-droite dans l'objet Scene et à récupérer l'ensemble des objets intersectés par cette droite: il faut noter que tout les objets générés via three.js sont constitués de triangles, on calcule si la demi-droite traverse ou non

l'objet en réalisant un test sur tout les triangles composant cet objet)

-Scene (à ne pas confondre avec la classe Scene.js de l'application), dans lequel on place tout les éléments que l'on veut afficher

-Camera, qui permet de visualiser l'objet Scene selon un point de vue défini

B) Architecture de l'application

J'ai ensuite étudié les différentes classes composant l'application afin d'en comprendre le fonctionnement interne (Figure 3, un diagramme de classe de l'application).

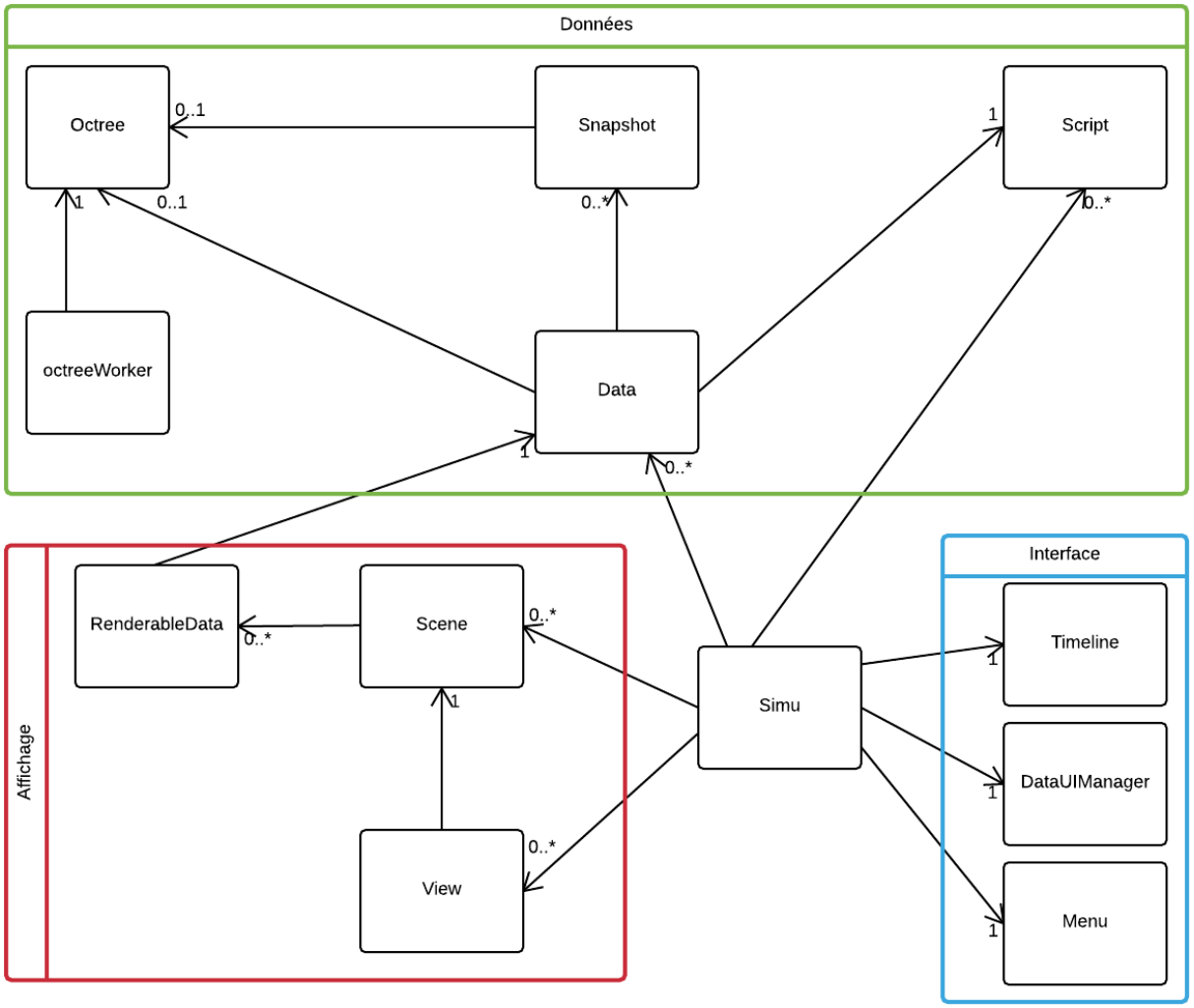


Figure 3

Ces classes peuvent elles-mêmes être divisées en 3 sous catégories:

1) Interface:

Cette catégorie regroupe les classes relatives à l'interface de l'application, et plus particulièrement les éléments de l'interface communs à toutes les Vues de l'application.

On objet "Vue" représente une perspective particulière du jeu de données courant (angle de caméra, options d'affichage sélectionnées, etc...).
Jusqu'à deux vues peuvent être affichées simultanément par l'application.

-Menu.js: cette classe gère l'affichage du menu de sélection du mode de Vue (Vue unique, Multivues, Oculus Rift, etc...). C'est ce menu qui apparaît au lancement de l'application et lorsque l'utilisateur presse Echap.

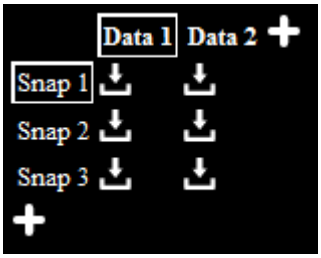


Figure 4

-DataUIManager.js: cette classe contient le code CSS relatif au menu de chargement des données (cf. Figure 4), ce dernier n'étant pas géré par la librairie dat.gui.
Il s'avère en effet que cette dernière est limitée quant à ses applications: elle permet de générer des éléments (boutons, formulaires...) auxquels on attache un actionListener, mais ne permet pas d'interagir avec ces éléments d'une autre manière .

-Timeline.js: cette classe gère l'affichage de la Timeline, qui permet de lancer / mettre en pause une animation (i.e. permet de gérer le déplacement du curseur selon la taille de la fenêtre et l'écoulement du temps, etc...)

2) Données:

Cette catégorie regroupe toutes les classes relatives au chargement des données et à leur manipulation en interne.

-Data.js: cette classe est utilisée pour la gestion du chargement des données (qui sera abordé par la suite dans le paragraphe).
Un objet Data correspond dans l'application à l'ensemble des données de simulation sur 1 élément (ensemble d'étoiles, nuage de gaz...), et est représenté par une colonne dans le DataUIManager.

-Snapshot.js: cette classe représente 1 élément à 1 période donnée, et est représenté par une case dans chaque colonne Data.
Grossièrement, chaque objet Data possède plusieurs Snapshots, chaque Snapshot contenant les données relatives à une simulation "photographiée" à un instant t.
Lorsqu'on déclenche une animation, on affiche successivement les données contenues dans chaque Snapshot (en ajoutant des transformations pour gérer les transitions).

-Octree.js: cette classe définit comment sont stockées les données après chargement via Data.js; chaque simulation peut être réduite à un ensemble de coordonnées (x,y,z) avec différentes informations associées à chaque point.
Afin d'optimiser notamment les calculs de raytracing (utilisés pour gérer la sélection d'un point avec la souris) et de niveau de détail lors de l'affichage, l'index de ces coordonnées est stocké sous la forme d'une structure de données récursive appelée Octree (Figure 5).
Un Octree est une structure arborescente et dans le cas de l'application elle contient 0 ou 8 fils.
La génération de l'Octree se fait de la façon suivante:
i) On regarde dans notre nuage de points si le nombre de points contenus dans l'Octree est inférieur à une limite N.
ii) Si i) est faux, alors on crée 8 Octree fils, chacun étant égal à 1/8ème de l'Octree d'origine, et on répète i)

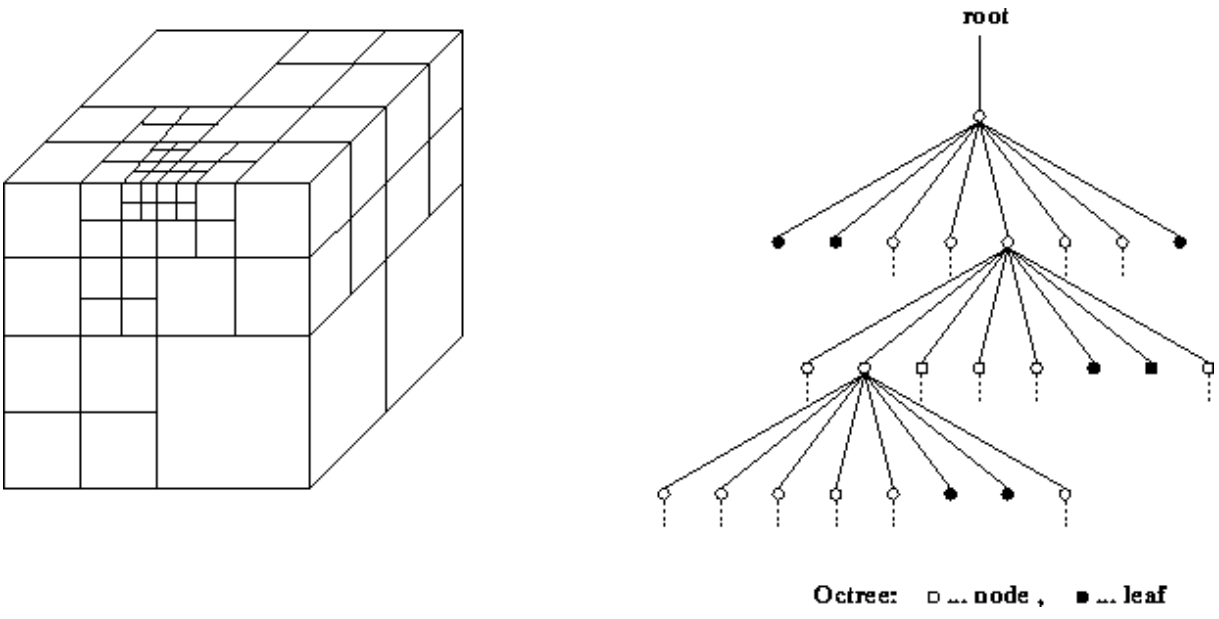


Figure 5

-OctreeWorker.js: cette classe est utilisée par Data.js lors du chargement des données, et utilise les "webWorkers" pour effectuer la mise en place de l'Octree.

-Script.js: cette classe contient les différent scripts utilisés pour lire les fichiers de simulation. En effet, chaque type de simulation a sa propre structure (et peut aussi être sous forme binaire ou non), et nécessite d'ajouter un nouveau script pour chaque type de fichier que l'on souhaite lire.

3) Affichage:

Cette catégorie regroupe les classes relatives au rendu des données.

-View.js: Comme mentionné précédemment, une Vue représente les données chargées dans l'application sous une certaine perspective. Par exemple le mode Multivues permet d'observer les données sous deux angles différents, avec des paramètres d'affichage différents ainsi que la possibilité de masquer certains jeux de données en parallèle sur une Vue donnée ou non. Chaque Vue possède ainsi son propre menu, son propre jeu de caméras, etc...
(Figure 6: exemple d'utilisation du mode Multivues; chaque vue affichant le même jeu de données)

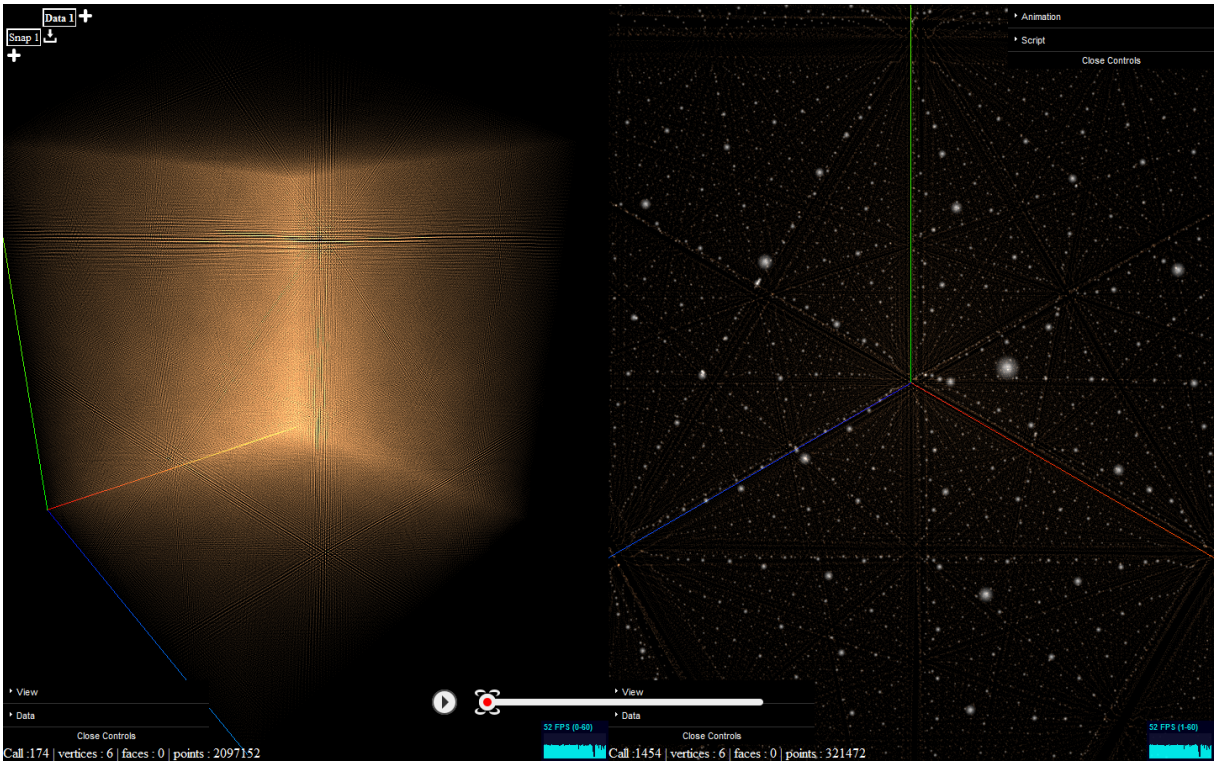


Figure 6

-Scene.js: chaque Vue possède un objet Scene qui permet de faire le lien entre les différents menus de la Vue et les données qu'elle contient.

L'objet Scene contient principalement les objets RenderableData, qui enrobent les objets Data communs à toutes les Vues.

-RenderableData.js: chaque RenderableData possède un objet Data, et a pour rôle d'effectuer l'affichage correspondant aux données qu'il contient selon les paramètres qui lui sont affectés par le biais de la Vue. C'est ici entre autres que sont gérés les appels à three.js ainsi que les calculs de raycasting et d'optimisation d 'affichage.
Attention: les objets Data sont communs à toutes les Vues, tandis que chaque Vue possède son propre jeu de RenderableDatas.

Enfin, la classe principale Simu.js permet de lier toutes les classes précédentes afin de faire fonctionner l'application.

C) Chargement des données

L'application étant assez riche et n'ayant pas la place pour décrire entièrement son fonctionnement, je vais tout de même aborder la manière dont est géré le chargement des données car j'en reparlerai assez souvent (notamment dans la partie traitant des correctifs).

a) Dans Simu.js: chaque bouton du DataUIManager possède un listener qui suite à un clic de l'utilisateur ouvre une fenêtre où ce dernier peut sélectionner les fichiers qu'il souhaite charger dans cet emplacement (la plupart des simulations sont composées non pas d'un unique fichier mais de plusieurs, de structure identique), après quoi un second listener appellera après validation des fichiers la fonction handleFileSelect de Data.js.

b) Dans Data.js: le chargement des données suit alors plusieurs étapes distinctes:
i) Tout d'abord, chaque fichier reçu par handleFileSelect est traité de façon asynchrone, i.e on va simultanément extraire les données contenues dans chaque fichier en leur appliquant le script courant de l'application grâce à la fonction readAdd.
ii) Lorsque le traitement de tout les fichiers est terminé, la fonction onEveryLoadEnd est alors appelée et ordonne l'exécution asynchrone de la fonction populateBuffer sur les groupes de donnés obtenus (autant que le nombre de fichiers de départ).
Le rôle de populateBuffer est d'organiser les données reçues et de les utiliser pour remplir les buffers de chargement de données. (par exemple, les données d1, d2..., dn contiennent chacunes un champ position de taille p1, p2..., pn; populateBuffer va initier la création du buffer position final de taille p1+p2+...+pn et va le remplir selon les champs indice i1, i2...in

uniques, tout en instanciant le buffer indice final, etc...)
iii) Après le peuplement des buffers, la fonction de création de l'Octree est appelée via l' utilisation de webWorkers (ici OctreeWorker.js).
Une fois l'Octree complet ce dernier est alors stocké dans le Snapshot correspondant, et un dernier traitement consistera à créer les buffers "departure" et "direction" d'après les données présentes (ou non) dans les Snapshots adjacents de la colonne Data.
Ces buffers seront utilisés lors de l'animation," departure" étant une sauvegarde du buffer "position" à l'instant t=0 et "direction" contenant le vecteur direction associé à chaque point.

II) Amélioration de l'existant

A) Amélioration de l'interface

a) La Vue active a désormais son menu en surbrillance. Cela sert notamment en Multivues, afin que l'utilisateur puisse savoir laquelle des deux Vues est verouillée.

b)Lorsqu'un emplacement du DataUIManager contient des données, il est représenté par une petite galaxie, qui remplace alors l'icône de chargement -cf. Figure 7- (auparavant aucune icône ne permettait de savoir si un emplacement contenait des données).
De plus, les emplacements non vides des colonnes Data actives sont en surbrillance (l'utilisateur a en effet la possibilité d'afficher/masquer les données contenues dans une colonne donnée).

Figure 7

c)En Multivues, lorsqu'on passe d'une Vue à une autre, les éléments actifs sur cette Vue se mettent dynamiquement en surbrillance sur le DataUIManager (idem pour les éléments inactifs et les emplacements vides).
De la même manière, le bouton "Activate Data" ne modifie l'affichage de l'icône associée aux données que si il appartient à la Vue courante (en effet, en Multivues l'utilisateur a la possibilité d'accéder aux options de n'importe laquelle des deux Vues, qu'elle soit sélectionnée ou non).
d) Création d'un bouton permettant d'activer / désactiver l'affichage du repère.

B) Amélioration des contrôles

a)Une des fonctionnalités d'origine permettait d'afficher les données selon un angle prédéfini en utilisant les touches 4,6 et 8 du pavé numérique. Cependant il était impossible de se déplacer depuis ces points; on peut désormais s'en servir comme points fixes pour réinitialiser la position de la caméra et continuer à déplacer avec.

b)Auparavant les touches de déplacement étaient ZQSD, et permettaient de se déplacer sur le plan horizontal par rapport à l'orientation de la caméra; l'ajout des touches A et E permet un déplacement vertical.

c) Lorsque l'on passe du mode Multivues à un mode de Vue unique, c'est la Vue précédemment active dans le mode Multivues qui est affichée (et non plus la Vue 0 par défaut).

d) Un clic dans le vide (i.e. dont la fonction de raycasting ne retourne aucun résultat) désactive désormais l'affichage des informations sur un point précédemment cliqué

C) Amélioration de la lecture des données

a) Ajout d'un script permettant la lecture de données VOTable (un standard de l'IVOA).
(V.O. pour Virtual Observatory, VOTable est un format, basé sur XML, employé couramment par les astronomes pour la représentation des tables de données astronomiques).
Il faut savoir que l'application utilise pour la lecture des fichiers un système de scripts.
Avant de charger un fichier l'utilisateur sélectionne le script correspondant au format dudit fichier, et c'est ce script qui en extrait les informations utiles (dans un objet littéral qui aura la même structure quelque soit le script utilisé)
Dans le cas des données VOTable, les coordonnées reçues sont sous la forme de coordonnées équatoriales auxquelles peut être associée une notion de distance; il faut donc les convertir en coordonnées (x,y,z) (si aucune information sur la distance n'est fournie, cette dernière est considérée comme égale à 1 pour toutes les données et on obtiendra un rendu sphérique)
Le calcul des positions (x,y,z) se fait de la manière suivante:
x = zsp * cos(RAJ2000) * cos(DEJ2000)
y = zsp * sin(RAJ2000) * cos(DEJ2000)
z = zsp * sin(DEJ2000)

Où:
-zsp représente le redshift (décalage vers le rouge); i.e. plus un objet astronomique est éloigné de la terre, plus son spectre visible tendra vers le rouge.
-RAJ2000 (pour Right Ascension) est l'angle mesuré sur l'équateur céleste à partir d'un point de référence.
-DEJ2000 (pour Déclinaison Equatoriale) est l'angle mesuré perpendiculairement entre l'équateur céleste et l'objet céleste observé.

(Note: on appelle équateur céleste lé projection de l'équateur terrestre sur la sphère céleste, et point vernal l'intersection entre l'équateur céleste et l'écliptique, qui représente l'orbite de la terre; cf Figure 8)

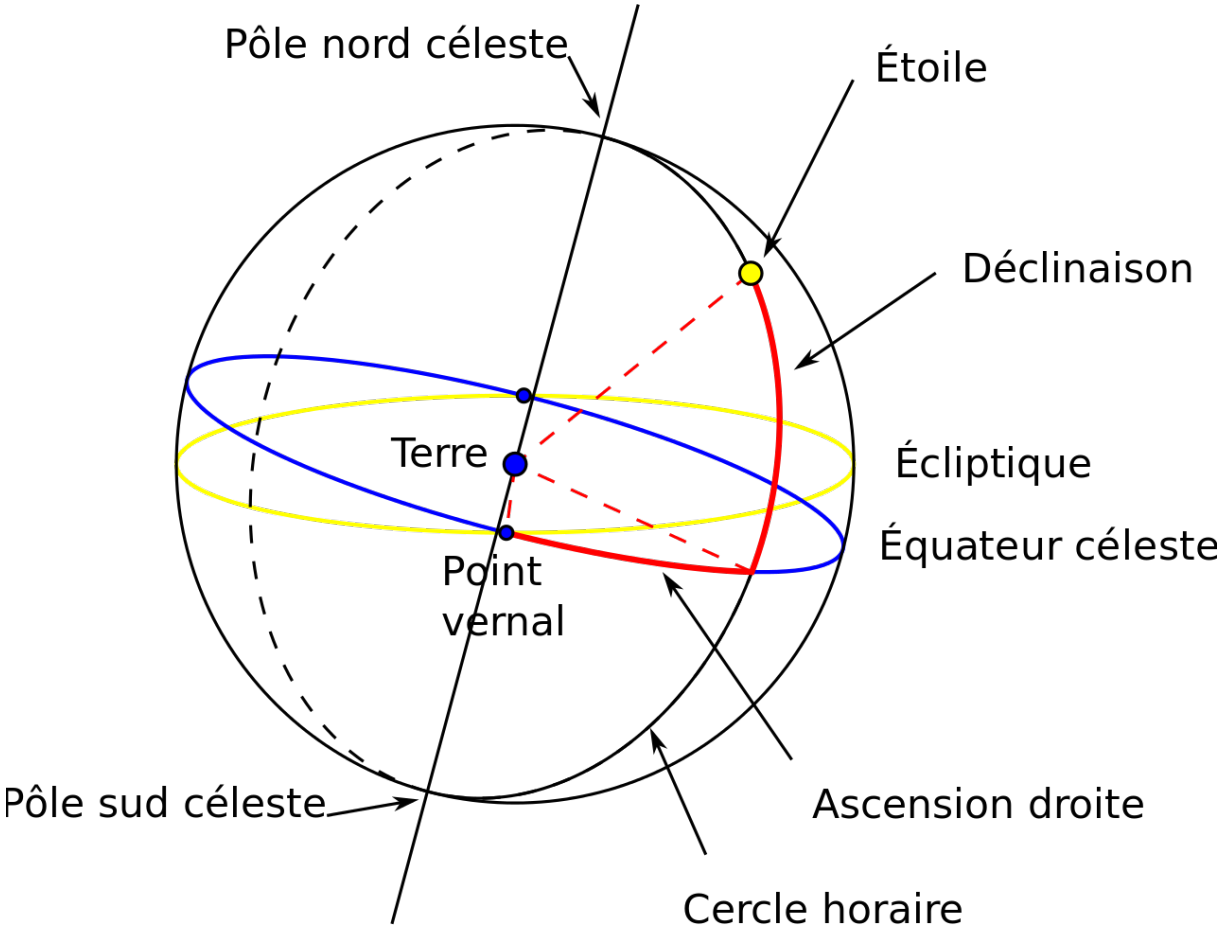


Figure 8

b) Amélioration du parseur VOTable (Javascript) du CDS.

Lors de l'implémentation de la visualisation de données réelles via l'application (c.f. troisième partie du rapport), nous avons opté pour un chargement, au format VOTable, des données VizieR.

Le format VOTable est l'un des formats sous lesquels on peut effectuer les requêtes, et le CDS dispose d'un parseur permettant de récupérer les informations qu'il contient (ce parseur a été réalisé en 2014 par le stagiaire Thomas Rolling, et est écrit en JavaScript).

Néanmoins ce parseur ne permet que de lire des données VOTables sous forme de fichiers (i.e. que l'on possède en local) mais ne permet pas de les lire lorsqu'on les reçoit directement sous la forme d'arbre XML (comme c'est le cas lorsqu'on effectue les requêtes sur VizieR depuis l'application).

Il a donc fallu lui ajouter un second constructeur acceptant en paramètre un arbre XML.

c)Refonte de la structure de données générée par les scripts:

Auparavant chaque résultat était généré sous la forme d'un tableau contenant des objets littéraux de la forme {name :NAME, value :VALUE}, deux d'entre eux étant obligatoirement de nom Index et Position et permettant de générer l'affichage des points. Les autres champs variaient selon les données récupérées par le script (infrarouge, masse, etc…)

Tout ces champs purement informatifs sont maintenant regroupés dans un unique objet {name :Info, value:[…]}.

De plus, un attribut nommé MetaType permet désormais de différentier les données chargées (selon qu'elles proviennent de simulations, de VizieR, et qu'elles sont en coordonnées sphériques ou non)

Dans la version actuelle de l'application, les objets retournés par un script sont tous de la forme: [Index, Position, Info, MetaType, Bornes], où Bornes correspond aux bornes min et max des coordonnées (x,y,z) du jeu de données local (la plupart des jeux de données sont répartis sur plusieurs fichiers, ces bornes ne sont donc représentatives que d'un fragment de l'ensemble des points).

III) Implémentation de nouvelles fonctionnalités

A) Mise en place du chargement et de l'affichage des données réelles

Le but de cette fonctionnalité est de permettre à l'utilisateur d'afficher dans l'application des données de relevés astronomiques issus de missions spatiales notamment (et plus précisément issues de catalogues de VizieR,et au format VOTable).

1) Interface

Il faut donc mettre en place une interface permettant de communiquer avec le service VizieR, de façon à ce que l'utilisateur puisse:

- Charger directement depuis le serveur des données VizieR au format VOTable en effectuant des requêtes ADQL (l'ADQL est le langage proche de SQL -avec des adaptations pour les coordonnées astronomiques- permettant d'envoyer des requêtes au service VizieR).
- Charger des données VOTable en local (via le DataUIManager)

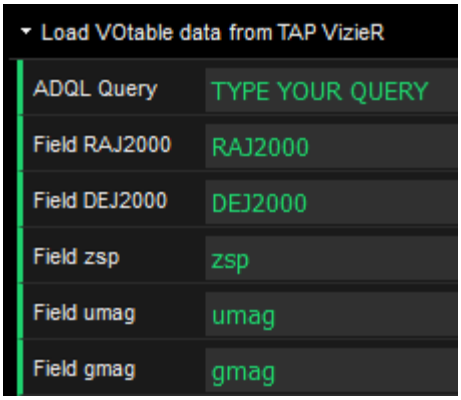


Figure 9

L'interface de chargement des données depuis VizieR -Figure 9- comporte plusieurs champs:

- Un champ pour la requête elle même.
- Trois champs pour récupérer l'identifiant des valeurs zsp, RAJ2000 et DEJZ000

Ces données sont indispensables car c'est grâce à elles que l'on peut déterminer les coordonnées (x,y,z) de chaque objet retourné par la requête (le détail des calculs est donné dans la partie 2, où l'on traite du script VOTable).

Une saisie de ces champs est demandée car, si dans un grand nombre de catalogues deVizieR ces valeurs portent bien ces noms, ce n'est pas une règle absolue et certains catalogues auront

par exemple zsp nommé z, etc... L'utilisateur a donc la possibilité de faire le lien.
Il faut noter que la saisie de zsp n'est pas obligatoire (c'est elle qui entre autre permet de déterminer la distance entre le point et l'origine) ; auquel cas l'affichage des données se fera sous forme sphérique c.f. Figure 10:

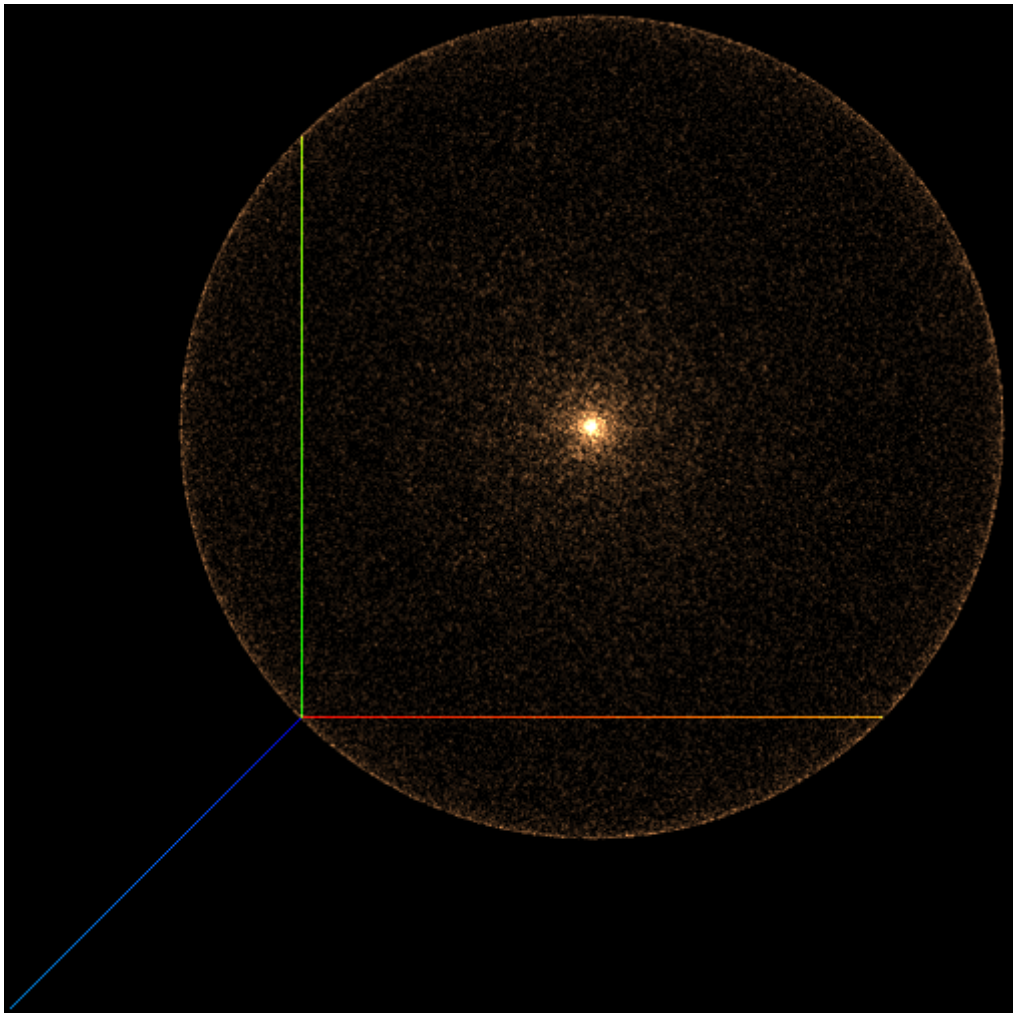


Figure 10

Bien évidemment les listeners clavier (mouvement de la caméra, etc...) sont désactivés lorsque l'utilisateur sélectionne les champs du formulaire de requête.

2) Lien avec le serveur VizieR

a) Structure des requêtes

Lorsque l'utilisateur appuie sur entrée, la requête saisie est alors envoyée au serveur via un

XMLHttpRequest de type Post (l'envoi de requêtes en cross-domain est possible sur le serveur de VizieR)

Basiquement, la préparation d'une requête XMLHttpRequest (abrégé xhr) nécessite 3 étapes:
i) xhr.open("POST", URL, true);
Cette première ligne permet de définir si la requête est de type GET ou POST, ainsi que l'URL de destination. (ici notre URL sera "http://tapvizier.u-strasbg.fr/TAPVizieR/tap/async")
ii) xhr.setRequestHeader("Content-Type", TYPE);
Cette seconde ligne est obligatoire dans le cas d'un post, et permet de transmettre au serveur le type d'information qu'on va lui envoyer ("application/x-www-form-urlencoded" dans le cas de VizieR)
iii) Enfin, on utilise
xhr.send("request=doQuery&format=VOTable&lang=adql&phase=RUN&runid=vQuery&query=" + request) pour envoyer la requête au serveur. (qui a au préalable été encodée au format URL i.e. certains caractères comme les espaces sont remplacés par % suivi d'un nombre hexadécimal)
b) Échanges avec le serveur
i)L'application envoie une requête POST au serveur, contenant la requête saisie par l'utilisateur.
ii)Le serveur retourne une url qui permet à l'application de vérifier périodiquement l'état de la requête sur le serveur.
(une vérification de l'état est effectuée toutes les deux secondes)
Les informations retournées par le serveur sont sous forme d'arbre xml, qu'il faut parcourir grâce à une fonction dédiée.
iii)Lorsque le serveur a fini de traiter la requête, il envoie l'url de téléchargement à l'application
iv) Traitement des données reçues par le script VOTable

3) Centralisation des données dans l'espace.

Jusqu'à maintenant, toutes les données chargées par l'application étaient des données de simulation, de bornes normées en [0,1]. Selon le type de fichiers chargés, ces données peuvent ne pas être normées et avoir des bornes extrêmes (où tous les points chargés ont leur x compris entre -5000 et -5000.005 par exemple). Il a donc fallu implémenter une méthode permettant de redimensionner ces données de façon à ce qu'elles soient comprises dans le cube de centre (0.5,0.5,0.5) et de côté 1:

a) On calcule le centre du polygone régulier englobant l'ensemble des points chargés
b) On calcule les bornes min et max (x,y,z) de ce polygone, et on sélectionne la longueur L de la différence la plus importante entre les bornes x,y ou z.
c) On translate chaque point par rapport à l'origine du repère et au centre du polygone
d) On divise la coordonnée de tout les points par L, pour avoir des données contenues dans un cube de côté 1. (c'est pour cette raison qu'on a effectué l'étape iii)
e) On translate chaque point par rapport au cube de centre (0.5,0.5,0.5)
Il est important de ne pas oublier que le chargement des données (de simulation notamment) est réalisé de façon asynchrone sur plusieurs fichiers. Il faut donc calculer dans le script les bornes x,y et z locales de chaque fichier, puis réaliser la centralisation dans Data en prenant les extrêmes de toutes ces bornes locales.

4) Homogénéisation des informations :

Il se trouve que la précédente version de l'application possédait une fonctionnalité permettant à l'utilisateur de mettre en valeur l'attribut de son choix, par exemple en lui appliquant un filtre coloré (les valeurs les plus hautes tendant vers le bleu, et les plus basses vers le rouge)
Plusieurs problèmes se posent néanmoins:
a) le calcul pour déterminer les bornes de couleurs pour la mise en valeur de l'attribut ciblé se fait en local pour chaque jeu de données. Ainsi si on superpose 2 jeux de données et qu'on veut mettre en valeur la température, les étoiles bleues de chaque jeu seront les plus chaudes, mais selon l'échelle propre à chaque jeu, et non pas via une échelle commune à ces deux jeux.
Solution : il a fallu mettre en place une structure de données permettant de gérer les bornes de chaque attribut :
On a ainsi une fonction d'homogénéisation qui
-Pour chaque Vue, calcule les bornes min et max globales à partir des bornes de chaque Data active
-Pour caque RenderableData ACTIVE, calcule les bornes min et max de chaque info...
-Pour chaque Snapshot
-Pour chaque info contenue dans le Snapshot (bornes min et max du Snapshot calculées lors du chargement VizieR)
Ce résultat est stocké dans la Vue sous la forme d'un tableau contenant des objets littéraux {nomInfo, min, max}
Cette fonction d'homogénéisation doit être appelée
-lorsqu'une colonne Data est activée/désactivée

-lorsque des données sont chargées/supprimées d'un emplacement du DataUIManager.
-lorsqu'on change de Snapshot dans la même colonne Data, ainsi que lorsqu'on ajoute une nouvelle ligne / colonne d'emplacement de données
Il est important de noter que chaque Vue possède ses propres bornes, i.e. en Multivues on peut donc avoir des jeux de couleurs différents selon les données affichées dans chaque Vue
Il n'y a enfin qu'à appeler la fonction de mise à jour de l'affichage à chaque fois que l'on appelle la fonction d'homogénéisation des informations.
b) Il n'est possible de mettre en valeur qu'un seul attribut à la fois (ce qui logique) or ces attributs sont nommés selon le nom qui leur a été donné dans la requête VizieR. Le problème est que 2 noms différents d'attributs (dans 2 catalogues par exemple) peuvent correspondre au même attribut dans l'absolu. Or, si les noms sont différents (car d'après Mr. Landais –qui gère les serveurs- il est impossible d'avoir accès à toutes les corrélations de noms d'attributs sur VizieR), il n'est pas possible pour l'application d'effectuer le lien entre ces deux attributs.
La solution qui a été trouvée consiste à mettre dans le menu de gestion des données un champ "merge" que l'utilisateur peut utiliser pour notifier à l'application que les 2 attributs déclarés dans le champ ont la même valeur sémantique (les valeurs à entrer sont de la forme nomAncien // nomNouveau), et à appeler la fonction d'homogénéisation après réalisation du merge.
Il est bien sûr impossible d'utiliser cette fonctionnalité pour donner à un attribut le même nom qu'un attribut déjà présent dans le même Snapshot.

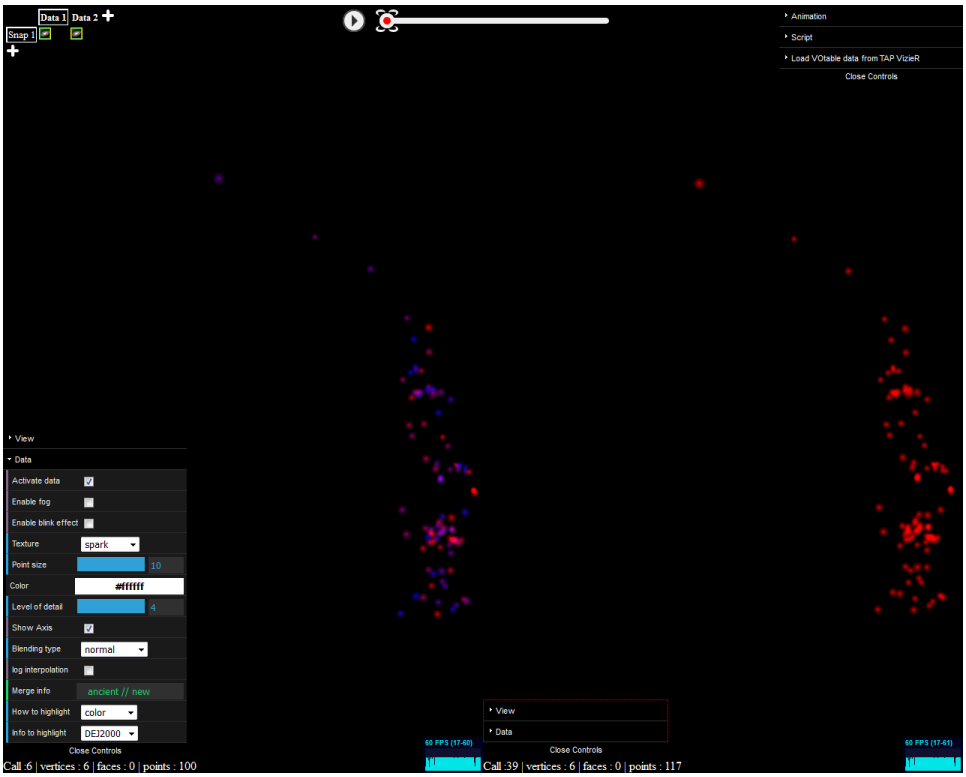


Figure 11

Sur la Figure 11(en Multivues), on peut observer une mise en pratique de cette homogénéisation.

On a chargé en mémoire deux jeux de données: un de 100 points, et un autre de 10000 points.

Sur les deux Vues on a mis en valeur la propriété DEJ2000 (par couleur).

Le fait est que sur la Vue de gauche, seul le jeu de 100 points est activé; on peut ainsi voir que le dégradé de couleurs couvre toute la gamme du bleu au rouge, contrairement à la Vue de droite qui, elle, a les 2 jeux d'activés: les 100 points appartenant au premier jeu, qui ont des valeurs DEJ2000 extrêmes par rapport au jeu de 10000 points, sont donc d'une couleur rouge vif.

5) Homogénéisation des positions dans l'espace

Problème : comme les données de simulation ne représentent rien de réel, et comme les données VOTable sphériques représentent la voûte céleste et ne contiennent donc aucune information sur la distance par rapport à l'origine, leur normalisation grâce au calcul vu en c) n'a rien de gênant.

Cette normalisation pose problème lorsque l'on examine les données VOTable 3D.

En effet si je charge un jeu de 10 points, ce dernier va être normalisé de façon à ce que la

distance maximale entre x,y, ou z soit égale à 1.
Mais si par la suite je charge un autre jeu de données (de 10000 points par exemple), ce dernier va aussi être normalisé dans un cube de taille 1: i l faut donc que la mise à l'échelle respecte la cohérence interne des données 3D. Il a donc fallu mettre en place une structure de données au fonctionnement similaire à celle réalisée dans le cadre de l'homogénéisation des informations.

a) Mise en place de la structure de données :

- Classe Script: l'objet Data renvoyé contient un nouveau champ "bornes" qui contient les bornes min et max des coordonnées x,y, et z de l'objet
- Snapshot: lors du chargement de données VOTables 3D dans un emplacement du DataUIManager les bornes calculées sont transmises au Snapshot.
- Data: lorsque des données sont chargées: après génération de l'objet Data par le script mais avant de peupler les buffers: on effectue une mise à jour des bornes de l'objet Data regroupant les Snapshots VOTable 3D ainsi que le Snapshot en train de charger le fichier (qui n'est pas encore considéré comme contenant des données)

Il faut bien entendu réaliser une mise à jour de ces bornes lorsque des données sont chargées/supprimées/masquées.

b) Mise en place de l'affichage

La mise en place de cette homogénéisation spatiale s'est heurtée à un problème de taille : comme on l'a vu dans la partie 1, chaque Vue possède ses propres RenderableDatas mais tous les RenderableData de l'application sont liés au même ensemble d'objets Data, commun à toutes les Vues. Le fait est que jusqu'à maintenant, bien que l'on puisse utiliser la Multivues pour observer 2 fois le même ensemble de points en mettant par exemple en valeur deux attributs différents, la position des points affichés dans chaque Vue était la même. Or si l'on veut avoir des données qui se mettent dynamiquement à l'échelle selon les autres données affichées dans la Vue, la position des points peut très bien varier d'une Vue à l'autre (selon que l'on choisit d'afficher des jeux de données différents selon la Vue).
L'idée qui a été trouvée consiste à ne pas modifier les coordonnées des points, mais à modifier leur affichage (i.e. comme chaque Vue possède ses propres RenderableDatas, même si les Datas sont communes à toutes les Vues, les RenderableDatas ne le sont pas)
Dans la fonction resetData de renderableData.js, qui génère l'affichage des données de l'application, lorsqu'on initialise le buffer de position du nuage de points, dans le cas où le Snapshot courant de l'objet Data est du type "VOTable_3D", au lieu de directement lui passer en paramètre l'attribut position du Snapshot, on en effectue une copie à laquelle on applique

l'algorithme d'homogénéisation spatiale(étendu aux bornes de la Vue) et on passe cette copie en paramètre à la place.
Le redimensionnement doit s'effectuer:
-après une suppression/insertion de données
-après un activation/désactivation d'affichage de Data
Désormais l'affichage des points de type VOTable_3D ne dépend plus (seulement) de l'objet Data et que la position des points affichés diffère de la position des points dans l'Octree de Data, les calculs de frustumCulling ainsi que de raycasting sur des données VOTable3D ne fonctionnent plus (on appelle frustumCulling le fait, en 3D, de ne réaliser l'affichage que de ce qui se trouve dans le champ de la caméra).
Il a donc fallu créer une structure de données intermédiaire telle que chaque RenderableData, en plus de posséder un objet Data contenant les données chargées dans leur état initial, aie accès à sa liste d'objets VOTable3D, qu'elle stocke donc dans un de ses attribut (créé à cet effet).
c)Cas particulier de l'animation concernant les données VOTable 3D:
Dû au fait que désormais selon la Vue, deux jeux de données de même origine peuvent afficher des points de coordonnées différentes, il a fallu modifier la mise à jour des buffers d'animation propre à ces jeux de données.
- Dans Data.js: lorsqu'on charge des données non-VOTable 3D, si les Snapshots précédents et/ou suivants sont de type VOTables 3D, il faut mettre à jour les buffers "departure" et "direction" situés dans le parent RenderableData (et non pas dans les Snapshots adjacents)
- Dans RenderableData.js: lorsqu'on homogénéise l'affichage des données VOTable 3D: après homogénéisation il faut mettre à jour les buffers "direction" et "departure" des Snapshots adjacents: il y a alors 2 cas à traiter:
i) le Snapshot contient des données VOTable 3D: il faut réaliser la mise à jour des buffers temporaires de RenderableData
ii) le Snapshot contient des données non VOTable 3D: il faut réaliser la mise à jour des buffers du Snapshot en attribut
iii) Comme chaque Snapshot contenant des données 3D aura son animation dépendante des buffers contenus dans RenderableData après homogénéisation; on est obligé de les stocker sous forme de tableau de buffer (et pas juste sous forme d'un buffer simple comme pour la position)
De plus, il faut -toujours dans le cas des données VOTable 3D- non plus homogénéiser un Snapshot donné quand celui est sélectionné pour l'affichage, mais les homogénéiser à l'avance

dès qu'une nouvelle donnée est insérée.
Originellement, le buffer direction de chaque Snapshot était mis à jour lorsqu'un nouveau jeu de données était inséré dans un emplacement adjacent. Le fait qu'avec les données VOTable 3D la position des éléments peut être soumise à des changements après avoir été définie (c.f. homogénéisation spatiale) nous oblige à, lors d'un changement de bornes globales:
i) mettre à jour les bornes de chaque RenderableData, puis, pour chaque RenderableData
ii) (dans une première boucle) on met à jour les données de chaque Snapshot dans renderable Data (uniquement la position)
iii) (dans une seconde boucle -les 2 boucles sont non-imbriquées-) on met à jour les buffers directions de chaque Snapshot dans RenderableData

B) Mise en place de l'affichage d'un cube entourant chaque jeu de données

Cette fonctionnalité a pour but de permettre une meilleure visibilité sur la répartition des données dans l'espace, et de dissocier deux jeux différents lorsqu'on les superpose (notamment dans le cas des données VOTable 3D).

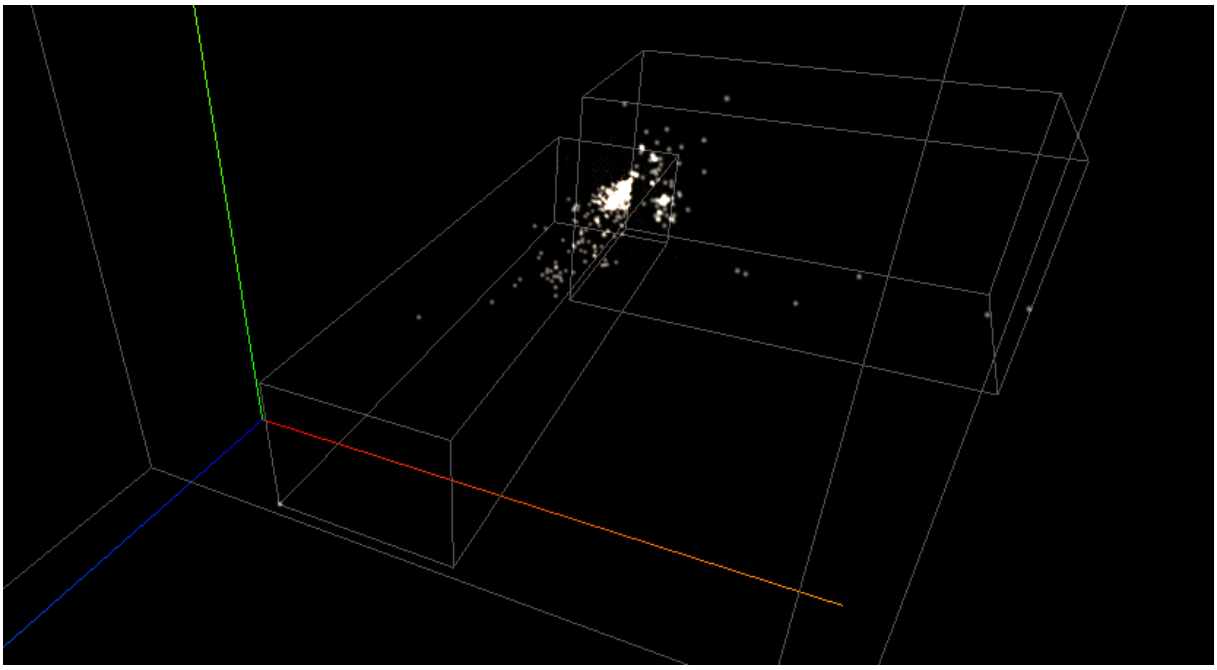


Figure 12

Cette option prend la forme d'une simple boîte à coche apparaissant sur le menu de chaque Vue et permet d'activer / désactiver l'affichage de la boîte des données courantes.
Les données relatives aux dimensions de la boîte sont facilement récupérables car elles sont

déjà calculées lors de la mise en place des bornes min et max de chaque jeu de données.
L'affichage des boîtes -Figure 12- se fait grâce à la librairie three.js: la structure est générée via l'objet BoxGeometry et l'affichage en fil de fer est obtenu en enrobant cet objet dans un EdgesHelper (les 2 objets doivent néanmoins être ajoutés à la scène)
Il faut noter que la boîte doit disparaître lors de la suppression des données, etc...
A ce niveau three.js ne permet justement pas de gérer le niveau d'opacité des éléments qu'on insère dans une scène; il faut donc systématiquement garder une référence des boîtes afin de pouvoir les extraire/réintroduire dans la scène.
Par ailleurs les bornes des boîtes sont contenues dans chaque Snapshot i.e. en Multivues les bornes sont "communes" à chaque Vue mais comme on stocke la référence de l'objet box propre à chaque Snapshot dans les RenderableDatas de chaque Vue, la modification des données relatives à la box dans le Snapshot n'impactent que la Vue courante.
Concernant les données VOTable 3D et leur homogénéisation dynamique: un appel à la fonction gérant l'affichage de la box est appelé à la fin de l'homogénéisation des données.

C) Mise en place du mode Widget:

Cet ajout doit permettre le fait d'intégrer l'application dans une page web, sous la forme d'une fenêtre -Figure 13- sur laquelle l'utilisateur peut cliquer pour accéder à l'application sous sa forme normale.

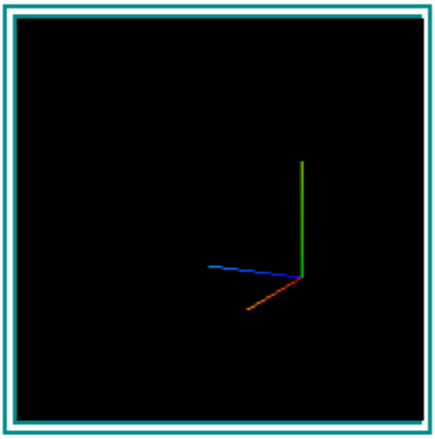


Figure 13

L'implémentation du mode widget a été faite de telle manière qu'aucune modification du code de l'application ne soit nécessaire pour passer d'un mode à l'autre:
Au niveau de HTML/CSS, on peut choisir au chargement de la page le mode:
a) normal: il doit juste y avoir la balise <div id:"app"> dans le body.
b) widget: en plus du contenu normal de la page, la balise "app" doit être contenue dans un élément ayant pour id "simuWidget" (et doté dans son CSS d'une width et height de mêmes valeurs)
Au niveau de l'application:
En mode normal, le seul changement significatif est que l'on affecte dans le CSS d'app l'attribut position sur "fixed".
En mode widget:
i) on affecte dans le css d'app l'attribut position sur "static"
ii) on affecte la propriété width de l'attribut container du dom à 0 -en mode normal cette dernière est égale à la taille de l'écran, et sert à corriger le problème de disparition de la deuxième Vue en Multivues (se reporter à la section des correctifs pour plus d'informations)-
iii) on définit un double égal à largeur de l'élément wrapper / largeur de la fenêtre, qui servira à conserver les proportions de la Vue lors du redimensionnement de la fenêtre en mode widget.
De plus afin de simplifier au maximum le lien entre l'application et une page web, le css de l'élément app est désormais défini via une fonction javascript, lors de l'initialisation.

D) Mise en place d'un outil de zoom

Le but de cette fonctionnalité est de permettre à l'utilisateur, sur un jeu de données ciblé, d'en sélectionner une zone précise afin de l'afficher dans une nouvelle Vue, où la zone ciblée est alors agrandie de façon à remplir tout le cube de données. (il est bien entendu possible d'effectuer toutes les opérations possibles sur les autres Vues sur ce jeu de données)

1) Mise en place de l'outil de sélection

Ce dernier prend la forme d'une boîte munie d'un repère, incluse dans le cube de données, qui apparaît lors de l'activation du mode de zoom (pour le modéliser on utilise les objets cylinder, cube et sphere de three.js)

Chaque axe du repère est muni d'une pointe; si l'utilisateur clique sur cette dernière et effectue un drag & drop, il déplacera le repère dans l'espace par rapport à l'axe qu'il aura sélectionné, déclenchant l'apparition de la droite selon laquelle on peut translater ce dernier -Figure.14- (si 2 pointes se superposent, la pointe sélectionnée sera la plus proche de l'utilisateur).

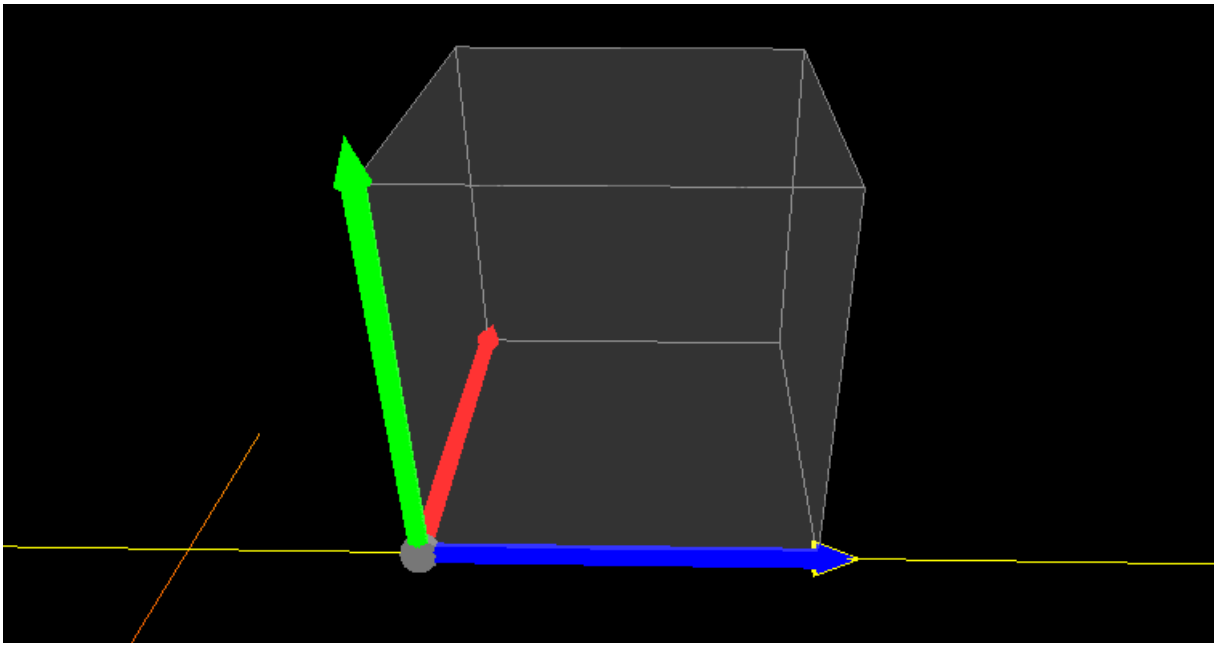


Figure 14

De la même manière, l'origine du repère, représenté par une sphère -Figure 15-, peut être utilisée pour modifier la taille de la boîte, en l'agrandissant à partir de son centre. (on utilise la propriété scale qui permet d'agrandir uniformément les objets; néanmoins pour les axes du repère comme les objets sont convertis sous forme de vecteurs après leur insertion dans la scène, il est impossible de modifier une de leur propriété seule -comme la longeur-, on est donc obligé de retirer les axes de la scène pour les remplacer par de nouveaux à chaque opération d'agrandissement)

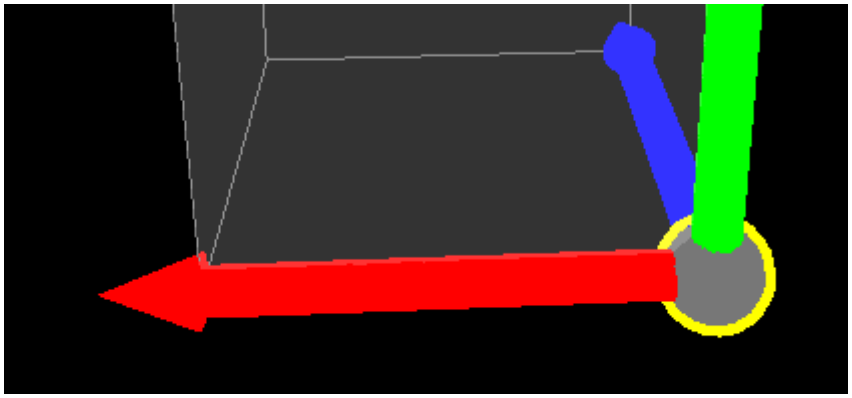


Figure 15

Bien sûr, les mouvements de la caméra sont désactivés durant le processus de drag & drop.
Il est impossible de déplacer l'outil en dehors des bornes du cube de données, et il est impossible de lui donner une taille débordant de ce dernier.
Le fait de pouvoir sélectionner des parties de l'outil via la souris utilise les fonctions de raycasting de three.js; on différentie les objets récupérés via le nom donné avant leur insertion dans la scène (aussi, l'appel à la fonction de raycasting se fait de pair avec la méthode qui gère le clic sur les points; si un élément de l'outil de zoom est sur la trajectoire du rayon, alors l'interaction avec ce dernier passe en priorité et les informations liées au point ne s'afficheront pas).

2) Gestion de l'activation / désactivation du mode zoom:

a) le fait de cocher/décocher le bouton de zoom fait désormais apparaître/disparaître l'outil de sélection

b) si l'utilisateur est en mode zoom et qu'il change de Snapshot/appuie sur echap/ajoute un

nouveau Snapshot: désactivation de l'outil de zoom
c) si l'utilisateur est en mode zoom et qu'il lance une animation/qu'il déplace le curseur sur la Timeline ou encore si il manipule la variable temps dans les options: désactivation de l'outil de zoom (le fait de lancer une animation n'est pas pris en compte si l'utilisateur est sur le dernier Snapshot)

3) Mise en place de la Vue esclave

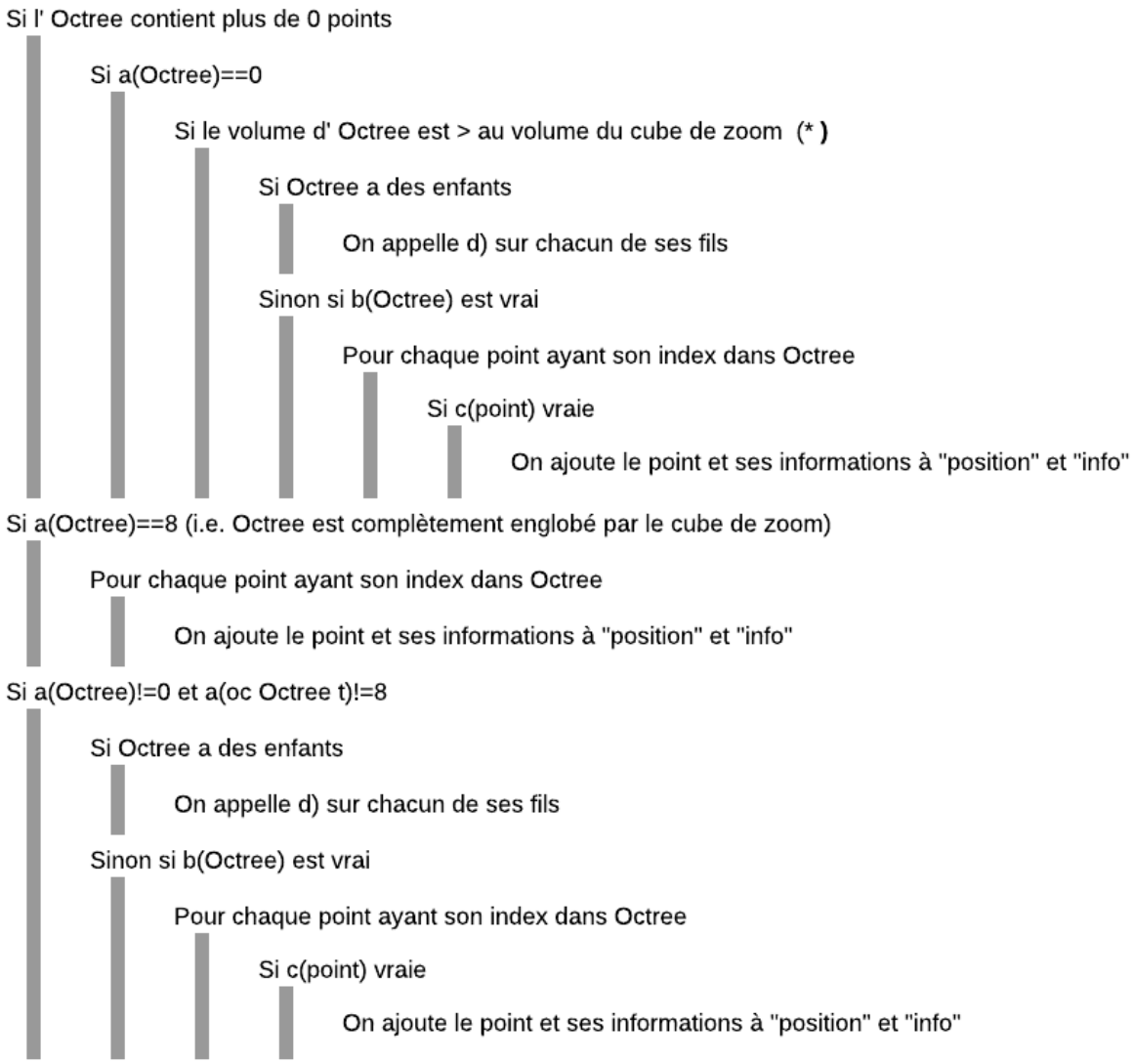
Comme la fonctionnalité de zoom est prévue pour être utilisée sur la Vue courante; il faut qu'elle soit utilisable en Vue simple mais aussi en Multivues.
Il a donc été décidé de faire en sorte que l'activation du zoom s'opère de la manière suivante:
i) la Vue sur laquelle on active le zoom -que l'on appellera la Vue maître- se retrouve munie de l'outil de sélection et se retrouve affichée à gauche de l'écran
ii) la seconde Vue -que l'on appellera la Vue esclave-, normalement utilisée en tant que seconde Vue lors du mode Multivues, se retrouve à droite de l'écran (et a son menu coloré en bleu)
Ce choix résulte du fait que la 2ème Vue de l'application (celle sur laquelle on n'applique pas le zoom) ne serait de toute façon pas affichée tant que le zoom est actif.
Plutôt que de créer/supprimer un nouvel objet Vue à chaque utilisation du zoom, on utilise cette seconde Vue comme support en la rendant dépendante de la première.
Ainsi son menu dispose de moins d'options -notamment la fonctionnalité de merge, ou encore le bouton d'activation du mode de zoom, car ces dernières sont gérées depuis la Vue maître-
Si l'utilisateur effectue l'une des actions qui désactive le mode de zoom (comme changer de Snapshot, presser echap, etc...) l'application revient à l'état précédant le déclenchement du mode de zoom.
(par exemple si l'utilisateur était en Vue simple, alors on repasse en Vue simple)
Pareillement le fait d'avoir le focus sur la Vue esclave ne la fera pas passer prioritaire au moment de désactiver le mode de zoom (il est aussi impossible d'insertion de nouvelles données tant que le zoom est actif) .
De la même manière, il faut faire en sorte que les données affichées par la Vue esclave avant l'activation du zoom lui soient restituées lorsque le mode de zoom est désactivé.
Ce qui implique que l'on doit effectuer une sauvegarde des options de la Vue, et les lui réappliquer lorsque cette dernière n'est plus esclave.

4) Implémentation de la récupération des points présents dans l'outil de zoom

La mise à jour des données de la Vue esclave doit être effectuée:
i) Lorsque la fonction de zoom est activée
ii) Lorsque l'utilisateur relâche la souris après avoir déplacé l'outil de zoom ou modifié sa taille
Cette mise à jour se déroule en 2 étapes:
a) Le stockage des points présents dans l'outil de zoom et leur formatage
Lorsque i) ou ii) se produisent, une fonction récupère le centre et la largeur de l'outil de sélection de la Vue maître, et les envoie à chaque RenderableData qu'elle possède, et réalise pour chacun d'entre eux un appel à la fonction de récupération/formatage des points.

Algorithme de la fonction de récupération des points:

I) Initialisation
1) On calcule les coordonnées des 8 points du cube de zoom, ainsi que ses bornes min et max xyz
2) On initialise les buffers "position" et "info" (où "info" contient les mêmes champs que les données courantes dans le Snapshot courant de l'objet Data lié au RenderableData)
3) 3 sous fonctions sont utilisées par l'algorithme principal:
a)une fonction permettant de déterminer si le cube de zoom contient (partiellement ou complètement) un Octree passé en paramètre (en se basant sur le nombre de points)
b) une fonction permettant de déterminer si un Octree passé en paramètre englobe complètement le cube de zoom
c) une fonction permettant de déterminer si un point donné est contenu par le cube de zoom
II) Algorithme principal
C'est une fonction récursive recevant un Octree en paramètre (à l'initialisation: l'Octree du RenderableData)



(*) Attention: on réalise la comparaison par volume car, si l'Octree est plus grand que le cube de zoom, alors si et seulement si son volume est plus grand que le cube de zoom il se peut qu'il englobe ce dernier.

De plus, durant une animation l'Octree est modifié (cf correction du problème de frustumCulling -voir dans la partie 4-) et les boîtes des fils de l'Octree ne sont plus nécessairement des cubes; il faut donc réaliser le calcul du volume à partir de la longueur maximale de la boîte.

II) Création de l'ensemble des buffers

Après le traitement des points par la fonction précédente, on repositionne chaque point récupéré afin qu'il soit à l'échelle du cube de données et non plus de l'outil de zoom.

On génère ensuite les buffers nécessaires à l'affichage des données par la Vue esclave (buffers departure, color,etc....)), ainsi que l'Octree qui servira de base pour les calculs de frustumCulling de la Vue esclave et on les stocke ensuite dans l'objet Data lié au

RenderableData traité (i.e. chaque objet Data contiendra les points englobés par l'outil de zoom issus des données initiales.

Il est très important de noter que les buffers doivent être sous la forme de Float32Array, qui sont des typedArray et dont la taille est définie lors de leur création (il est impossible de réaliser sur eux des opérations de type push), ce qui peut être contraignant lorsque l'on veut remplir les buffers au fur et à mesure que l'on parcours l'Octreee.
Il a enfin fallu modifier les différentes fonctions de RenderableData relatives à l'affichage des points de telle manière à ce qu'elles utilisent les buffers précédents lorsque la Vue est en mode esclave, et plus précisément la fonction rafraîchissant l'affichage ainsi que la fonction de frustumCulling (il faut utiliser l'Octree esclave pour gérer la détection des points).
L'outil de zoom permet donc de restituer dans la Vue esclave les différents éléments contenus dans le cube de zoom, qu'ils proviennent de différents jeux de donnés ou non -Figure 16-.

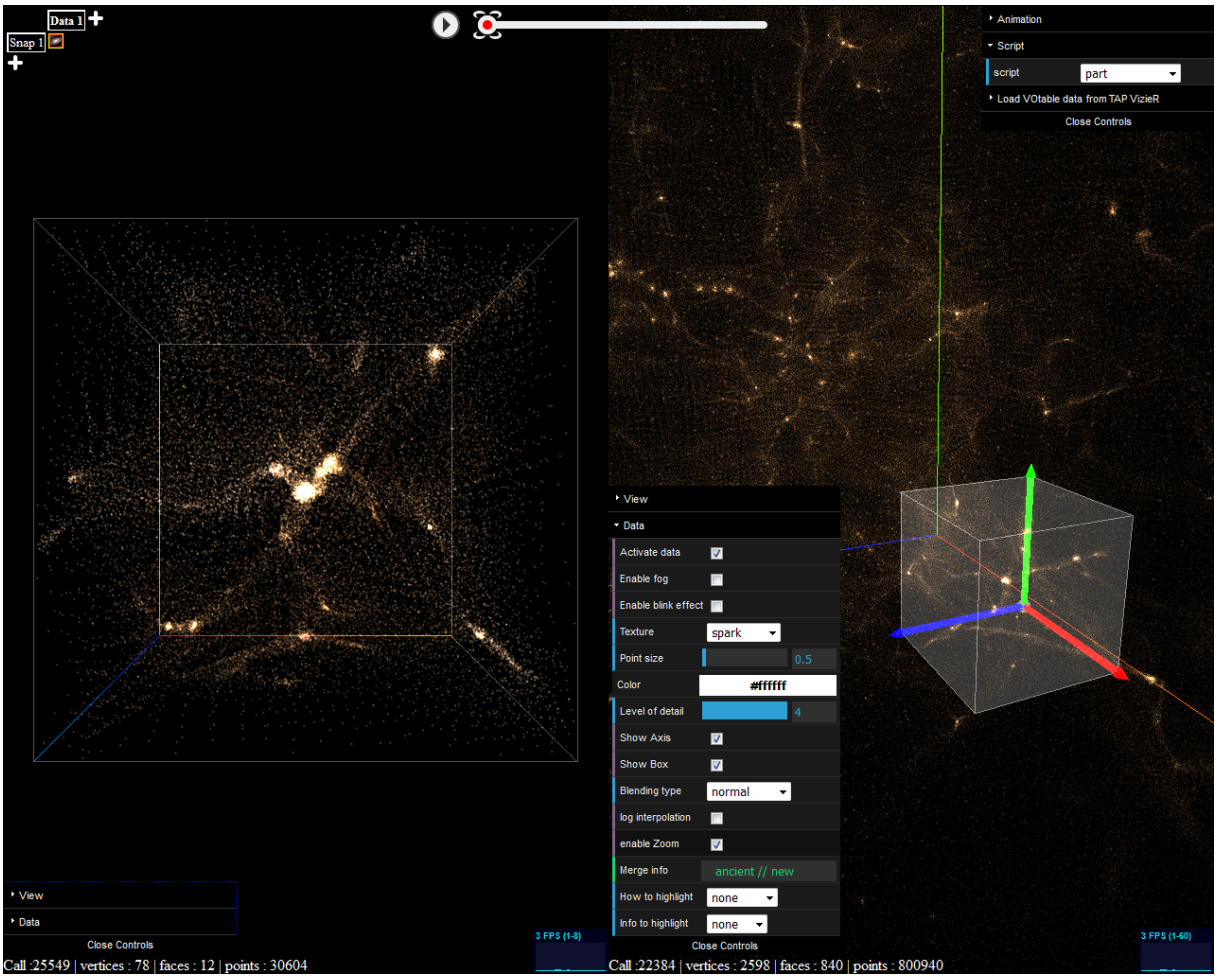


Figure 16

5) Fonctionnalité de focus sur un point avec l'outil de zoom

Une dernière fonctionnalité a été rajoutée et consiste à pouvoir, lorsque l'on est en mode zoom et que l'on a le focus sur la Vue maître, de centrer l'outil de zoom sur un point que l'on aura sélectionné au préalable.

Il faut prendre en compte le cas où le point est proche de la bordure du cube de données:
i) Si la distance entre le point et la bordure est supérieure ou égale à la moitié de la largeur minimale de l'outil de zoom, on redimensionne simplement l'outil de zoom avant de le centrer sur le point.
ii) Sinon on déplace l'outil de zoom de façon à ce qu'il soit le plus possible centré sur le point tout en étant collé à la bordure du cube de données

E) Mise en place de la fonctionnalité de changement de système de coordonnées

Sur n'importe quel jeu de données chargé, l'utilisateur peut choisir de changer son système de coordonnées via le menu Coord. Syst. de la vue.

Il sélectionne pour les champs x, y et z les informations correspondantes (magnitudes, position, etc...)

On peut ainsi passer d'un système de coordonnées

x: xPos, y: yPos, z: zPos à

x: rmag, y: gmag, z: imag

Important: l'un ou plusieurs des champs peut être set à 'none', i.e. selon le système de coordonnées choisi les données peuvent êtres affichées sur 3, 2 ou 1 dimensions.

A noter que pour permettre de repasser ensuite au système de coordonnées spatial (les 'vrais' xyz)

Les informations relatives aux champs xyz de chaque point ont été ajoutés aux champs info dans les différents scripts de l'application (ainsi lorsque les buffers position sont écrasés au changement de système de coordonnées, l'information xyz est conservée)

Prise en compte de l'animation (données non-VOTable 3d)

Il faut faire extrêmement attention lors de la création du nouvel octree, buffer position et buffer index: en effet lorsque l'utilisateur voudra repasser au système de coordonnées initial (xyz), il faut qu'il puisse réaliser une animation entre les != jeux de données chargés.

Il faut donc IMPERATIVEMENT, en plus d'effectuer le tri habituel des index (et de la position) lié à la création de l'octree, retrier les index par la suite en utilisant l'index des index défini au moment du chargement des données.

Prise en compte de l'homogénéisation (données VOTable 3d)

Tant qu'une donnée VOTable_3d est affichée, la modification de son système de coordonnées n'impacte pas le positionnement des autres données VOTable_3d (affichées sous coordonnées xyz)

Si cette donnée repasse au système de coordonnées xyz, alors elle se repositionne de façon homogénéisée // aux autres données VOTable_3d

En clair, la logique d'homogénéisation est la même quelque soit le système de coordonnées employé.

Prise en compte de l'outil de zoom

Les informations relatives au système de coordonnées de la vue maître sont transmises à la vue esclave, de plus lors du passage de la vue simple à la multivues, si aucune 2nd vue n'avait déjà été crée la copie des champs cooSys est effectuée

Au niveau des menus et afin de savoir quel système de coordonné est utilisé pour tel jeu de données,

l'information est stockée dans les renderableDatas correspondants (sous forme de tableau)

Important: l'information n'est PAS stockée dans les snapshots comme on pourrait le penser de prime abord: car les snapshots sont communs à toutes les vues, et chaque vue peut afficher le même jeu de données sous un système de coordonnées différent.

A noter que la répartition des couleurs (info to highlight), de la relation entre l'index des points et les infos correspondantes est conservée (pas de problème d'index, les données affichées restent cohérentes quelque soit le système de coordonnées utilisé)

IV) Correctifs

A) Correctifs au niveau de l'interface

a)Le redimensionnement de la page ne dérègle plus les rapports de distance entre les points d'arrêt (représentant les Snapshots) de la Timeline et le curseur de navigation; i.e. si on stoppe une animation à t=0,6, le redimensionnement la page n'aura pas pour conséquence un positionnement erroné de ce dernier.

b) Lorsque le curseur s'arrête sur le dernier point d'arrêt de la Timeline (i.e. il n'est normalement plus possible de déplacer le curseur au delà), il est désormais impossible de le déplacer en dehors de cette dernière en "trichant" sur la variable temps.

c) Correction du problème de white screen qui dans le mode Vue Multivues faisait disparaître la seconde Vue lorsqu'on redimensionnait la page à une largeur supérieure à celle qu'elle au moment de sa création.

d) Désactivation du focus sur les menus déroulants lorsque l'utilisateur cesse de parcourir ces derniers.

e)Correction d'un problème dans la fonctionnalité de mise en valeur d'un attribut: lorsqu'on sélectionnait un attribut pour le mettre en valeur, que l'on en changeait de Snapshot puis que l'on revenait sur le Snapshot initial: l'info sélectionné dans le menu (ainsi que l'affichage correspondant) s'était décalée d'un cran dans la liste d'attributs du menu.

B) Correctifs au niveau de l'affichage

a)Lorsqu'on charge un jeu de données: ce dernier s'affiche directement à l'écran à la fin du chargement (il n'y a plus besoin de passer par le menu de la Vue et de décocher puis recocher la case "Activate Data" pour rafraîchir l'affichage des données)

b)Correction du problème qui faisait que chaque jeu de donnée vide de l'application affichait de façon rémanente le dernier jeu de données chargé parcouru par l'utilisateur --> maintenant la sélection de données vides n'affiche rien (ce qui est cohérent avec l'état des données en haut à gauche)

c) Correction d'un problème qui causait une disparition des points à l'affichage (no1).
Il se trouve que c'est l'objet Data qui contient les buffers servant à l'affichage des données. Chaque Snapshot contient les données relatives aux données que l'on y a chargées, mais lorsque l'utilisateur clique sur un Snapshot pour afficher son contenu, c'est l'objet Data qui stocke dans ses attributs "currentIndex", "currentPosition"… les valeurs du Snapshot.
Il s'avère que ces attributs "current" n'étaient mis à jour que lors du chargement de nouvelles données, si bien que si par exemple on charge un jeu de 5 millions de points en S1, puis un jeu de 10000 points en S2, la taille des attributs "current" de Data passe respectivement de 5 millions à 10000. Mais si l'utilisateur désirait par la suite changer de Snapshot en passant de S2 à S1, la taille de l'attribut currentIndex n'était pas réinitialisée et l'application tentait d'afficher 5 millions de points tout en ne possédant que les 10000 premiers index.

d) Correction d'un problème qui causait une disparition des points à l'affichage (no2)
Il s'avère que le problème ne provenait d'une erreur assez importante: l'algorithme de redéfinition de l'index des points associés à un Snapshot donné permettant de gérer le niveau de détail ne prenait en compte que le cas strict où le nombre de points était égal à un multiple de 4. (c.f. les 4 niveaux de détail proposés par l'application)
I.e. lorsqu'on chargeait un jeu de données ne respectant pas cette condition, les 3/4 des points chargés se retrouvaient dotés d'un index à virgule dont la valeur ne correspondait plus a l'indice du buffer de position/4. De plus comme la division par 4 s'effectue non pas au niveau du buffer des index mais bel et bien sur les sous index obtenus à partir de chaque octant, même en ayant un nombre de point total divisible par 4, il est statiquement fort peu probable que chaque octant contienne lui aussi un nombre de points divisible lui aussi par 4.
Il faut savoir qu'en JavaScript les tableaux sont similaires à des objets dans le sens où t[0.75]=10 n'est pas une écriture incorrecte, ainsi il simple erreur de calcul d'index peut changer une instruction d'affectation à une case existante d'un tableau en instruction créant une nouvelle case ayant un numéro à virgule et recevant la valeur destinée à la case d'origine (qui, elle, aura sa valeur set à null).
Dans le cas de notre exemple, 3/4 des points se retrouvaient ainsi aux coordonnées (0,0,0) et on avait un affichage de détail de niveau 1 tout en utilisant les ressources pour un niveau 4. On peut ainsi voir dans la Figure 17 l'amas créé par l'ensemble des points ayant des coordonnées nulles (en bas à gauche de la capture).

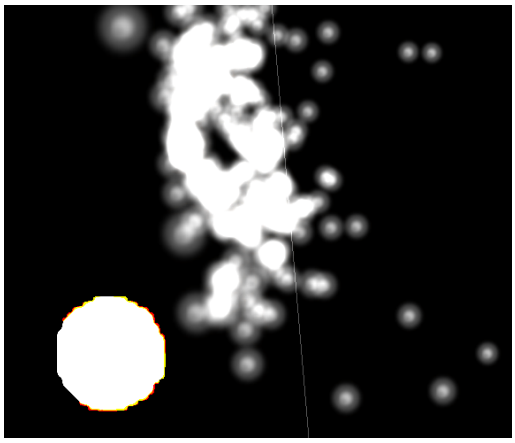


Figure 17

Il a donc fallu redéfinir l'ensemble des opérations relatives à l'affichage des points selon leur index et le niveau de détail (notamment dans la fonction de frustumCulling de RenderableData.js), sachant qu'il est impossible d'effectuer un trop grand nombre d'appel à la fonction drawCall de three.js sous peine de voir le nombre de fps passer à un ratio de 2 par seconde. (par exemple il n'est pas envisageable pour un niveau de détail de 2 et N points à afficher d'effectuer N/2 drawCall, il faut en effectuer le moins possible tout en possédant une indexation qui permette de gérer le niveau de détail)
Pour corriger cela, lorsqu'un Octree est entièrement situé dans le champ de la caméra, au lieu d'ajouter au drawCall son start et son count, on va aller chercher récursivement le start et le count de chacun des Octree feuilles qu'il contient.
Le problème qui aurait pu être posé par cette méthode est que, si le niveau de récursivité est trop grand, cela impacte les performances. Comme cette méthode fonctionnait sans affecter les fps sur les simulations part (contenant +2 millions de points) sachant que la taille des Octree est set depuis la VO à 1000; cette taille fixée à 1000 a été remplacée par une taille qui s'ajuste dynamiquement selon le nombre de points total dans l'Octree. Comme cette taille est proportionnelle au nombre de points, le rapport de taille entre les Octree feuilles et l'Octree principal est suffisamment élevé pour que l'homogénéisation du niveau de détail soit totalement uniforme pour l'utilisateur, tout en étant suffisamment bas pour que le nombre d'appel récursif ne dépasse jamais une certaine limite.
Le problème soulevé par cette correction est que le découpage des points lors de l'affichage

des données à un niveau de détail inférieur à 4 est trop régulier (cf. Figure 18)

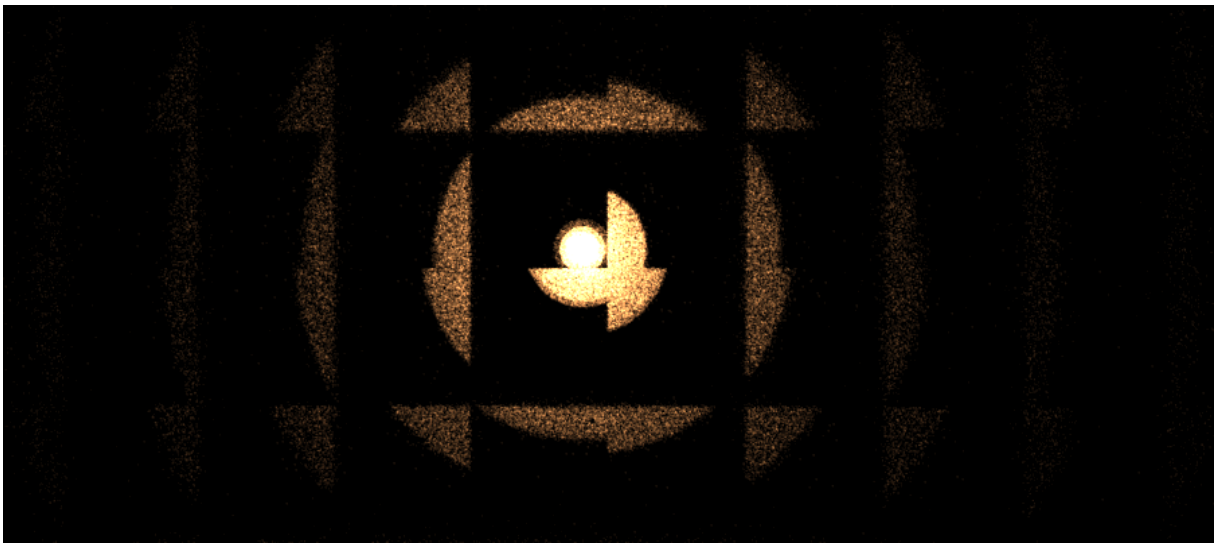


Figure 18

Une modification supplémentaire a donc été apportée:
Lors de la création de l'Octree chaque feuille se voit dotée d'un index de la forme [2,3,1,0] dont les éléments sont ordonnés de façon aléatoire, de telle sorte que lors de l'affichage la sélection des points par feuille se fasse de la manière suivante:
-pour chacun des 4 niveaux de détail n; pour i allant de 0 à n
-si index[i]=n avec i tel que i<niveau de détail actuel, alors on réalise le drawCall correspondant à l'affichage du quart d'index i correspondant
-sinon le quart n'est pas affiché

e) Correction du problème de déformation des données adjacentes lorsqu'on charge des données; et durant l'animation (cf. Figure 19).

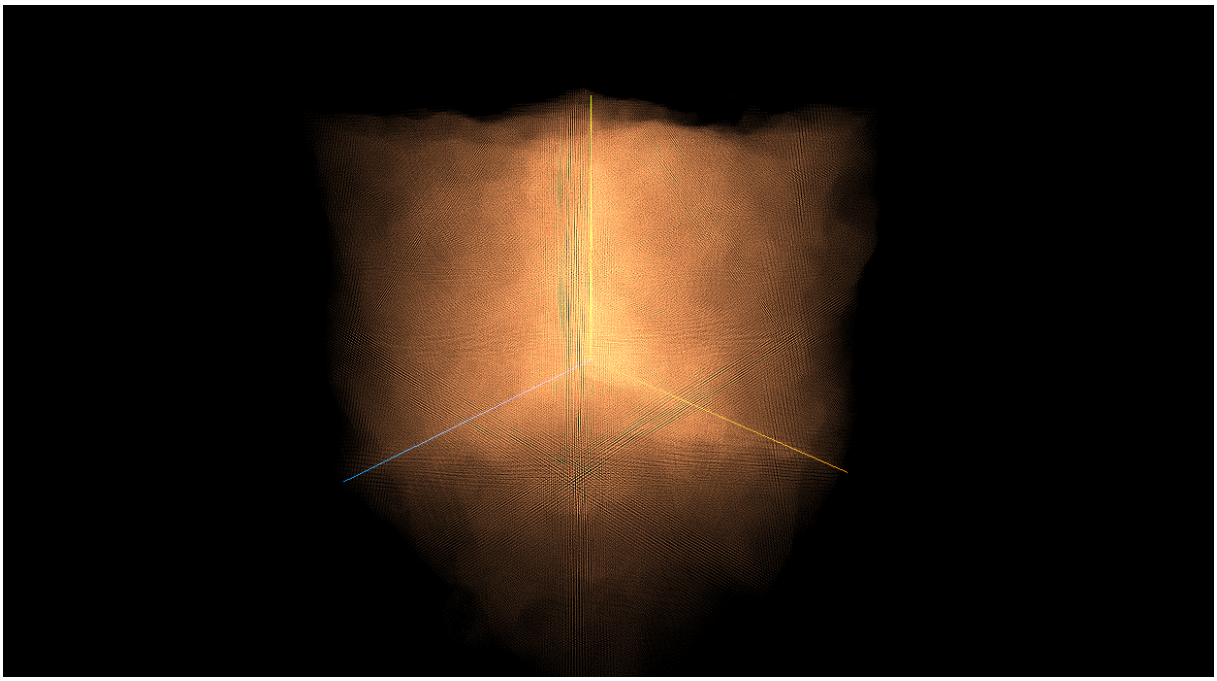


Figure 19

Cause: comme certains points des simulations sont cycliques (i.e. dans leur déplacement ils passent d'un côté du cube de données à un autre), la solution qui était mise en place depuis la V0 était, lors du chargement de nouvelles données (i.e. la mise à jour des buffers déplacement) de "repositionner" les points qui avaient à parcourir une distance supérieure à 0.5 de l'autre côté du cube (grâce à la méthode cyclicPosition de Snapshot).
Le problème est que ce déplacement des données n'était pas très fin (par exemple, si un point se trouvait en x=0.8, si suite au chargement d'un autre Snapshot sa destination avait un x=0.2, la position du point dans le premier Snapshot se retrouvait avec un x égal à 0.8-1= -0.2, i.e. il se trouve en dehors du cube de données, d'où le fait que ce processus ai été considéré comme une erreur)
La solution trouvée a été de ne pas modifier le buffer position des points de la simulation à t=0; mais plutôt d'effectuer dynamiquement le changement de coordonnées durant l'animation, seulement dès que l'une des coordonnées xyz devient >1.
Le positionnement des points à la pause durant une animation est géré par la méthode computePosition de Data; mais leur position durant l'animation est gérée par le shader parametric.animated.vertex, qu'il a fallu modifier.

f) Correction du problème d'animation étrange - qui cause une animation différente selon l'ordre selon lequel on charge les fichiers-: il est en fait dû à un peuplement erroné des buffers de déplacement (ce problème a mis du temps à être découvert car en général charger les données en commençant par le Snapshot le plus bas n'est pas intuitif).
En effet l'index des points que l'on récupère après chargement des fichiers est envoyé à l'Octree qui le réarrange selon l'ordonnancement des points dans l'espace. Comme on met à jour les buffers de déplacement selon la présence d'un autre Snapshot plus haut / plus bas, le problème est que comme l'agencement des données change d'un Snapshot à l'autre, la position des indices dans le buffer des index aura changé et on donnera aux points la direction d'un point d'un id différent dans l'autre Snapshot. La raison pour laquelle ce problème n'a pas été mis en évidence plus tôt est que le peuplement des buffers se fait de la manière suivante:
i) Des données sont présentes dans le Snapshot précédent
Alors on met à jour le buffer direction du Snapshot précédent à partir du Snapshot courant
ii) Des données sont présentes dans le Snapshot suivant
Alors on met à jour le buffer direction du Snapshot courant à partir du Snapshot suivant
Le fait est que cette mise à jour des buffers était appelée dans le Snapshot courant avant l'appel à la fonction d'homogénéisation de l'index pour le niveau de détail; ainsi dans le cas i) on a le buffer direction du Snapshot précédent qui récupère les bonnes directions, et ensuite l'index du Snapshot courant est trié selon le niveau de détail, mais dans le cas ii) le Snapshot suivant a déjà eu son index trié selon le niveau de détail et le Snapshot courant récupère les mauvais index.
Correction: création d'un buffer supplémentaire qui permet d'effectuer la correspondance entre les tableaux d'index (i.e. sa case 0 contient le no de case de l'index "0" dans le buffer des index qui permet à deux Snapshot de se transmettre leurs coordonnées sans erreurs.

g) Correction du problème des points qui disparaissent durant l'animation, lors de la transition d'un Snapshot à un autre:
L'origine du problème vient de la fonction qui permet de gérer l'affichage de manière optimisée.
Lors de la création de l'Octree, chaque feuille se voit attribuer un objet "box" qui contient ses bornes de définition.
Sauf que lorsqu'on lance une animation, les points vont se déplacer, mais l'Octree tel qu'il a été défini lors du chargement des données ne changera pas. Ce qui veut dire que si un point qui à t=0 n'est pas dans le champ de la caméra va à t=0.5 se retrouver à une position qui

devrait le faire apparaître dans le champ, le fait est qu'au niveau des calculs d'optimisation ce point sera considéré comme étant dans un octant dont les bornes (correspondant à celles de t=0) ne se situent pas dans le champ de vision. Le point ne sera donc pas affiché.
Et lorsque l'animation se terminera et que le changement d'Octree s'effectue: l'algorithme de calcul du niveau de détail prendra comme référence les box du nouvel Octree, et de ce fait les points qui précédemment n'étaient pas affichées auront leurs Octree avec des box de nouveau cohérentes, et seront affichés.
Le problème est qu'il n'est pas du tout optimisé de mettre à jour les box des Octree de façon dynamique lorsque l'animation est en cours (fonction récursive, gourmande), la solution qui a été trouvée est:
i) Lorsque l'animation est en cours, afficher la totalité des points sans calcul de précision (à cause de la position cyclique des points qui peut les faire passer d'une borne à l'autre du cube, il n'est même pas possible d'effectuer une optimisation grossière)
ii) Mettre en place une fonction de mise à jour de l'Octree, mais de l'appeler uniquement lors du changement de Snapshot ET lors de la mise en pause de l'animation. Ainsi il n'y a plus aucun problème d'affichage de points, et lors de la pause on optimise bel et bien l'affichage tout en affichant les points en tenant compte de leur position réelle dans le temps.

h) Correction d'un problème qui causait une disparition des points à l'affichage (no3)
Ce problème a été rencontré à de diverses reprises au cours du stage sans que je puisse en déterminer les causes et ce n'est qu'après la mise en place de l'outil de zoom (où j'ai trouvé par hasard un jeu de données où ce problème apparaissait systématiquement) qu'il a pu être résolu.
Les objets affichés par Three.js possèdent tous un attribut appelé BoundingSphere qui n'est jamais initialisé dans l'application. Cet attribut possède un centre et un rayon, et représente une sphère -invisible- qui fait office de "hitbox globale" pour l'objet, c'est à dire que si cette sphère n'est pas (partiellement ou non) dans le champ de la caméra, l'objet ne sera pas affiché (et cela indépendamment des appels à la fonction drawCall pour le calcul de frustumCulling)
Le problème est que cet attribut n'est jamais utilisé dans l'application et est défini par défaut comme étant nul. Néanmoins, dans certains cas, Three.js va décider arbitrairement d'attribuer un tel attribut aux nuages de points que l'on crée (avec des valeurs complètement fausses).
Par exemple, le jeu de données qui a permis de mettre ce problème en évidence (et dont les points étaient répartis dans l'ensemble du cube de données) se retrouvait doté d'une BoundingSphere de centre ~ (0.3, 0.5, 0.4) et de rayon 0.2, i.e. si l'utilisateur tentant de ne

visualiser que les données situées aux coins du cube de données, ces dernières disparaissaient. La solution mise en place pour résoudre ce problème consiste à réinitialiser la BoundingSphere de chaque nuage de point avant d'effectuer l'appel à la fonction d'affichage de ce dernier.

C) Correctifs au niveau de l'affichage

Détail relatif au nettoyage de la mémoire:
Soit obj un objet three.js ; AVANT d'utiliser obj.remove(); il faut lui appliquer obj.geometry.dispose(); et obj.texture.dispose(); sinon l'objet sera uniquement retiré de la scène et pas de la mémoire elle même.

Conclusion

En conclusion, ce stage aura été une expérience extrêmement enrichissante qui m'aura permis d'élargir mes connaissances dans de nombreux domaines, plus particulièrement en 3D: problématiques d'optimisation: frustum culling, raycasting, mais aussi en programmation JavaScript, que j'ai étudiée bien plus en profondeur que je ne l'avais fait au cours de mon DUT.

Néanmoins, bien que de nombreux ajouts aient été effectués au cours du stage, de nombreuses pistes n'ont pas été explorées, tant en terme de fonctionnalités (possibilité de mettre en valeur les zones denses des nuages de points, d'effectuer des zooms récursifs...) que d'optimisation (afficher de moins en moins de points plus la distance est grande entre le nuage de points et la caméra, gérer le découpage de l'Octree directement depuis le serveur...) et témoignent du potentiel de l'application.

En outre, les possibilités offertes par cette dernière ne se limitent pas au simple domaine de l'astronomie, et peut être utilisée pour afficher n'importe quel jeu de données tant que ces dernières sont présentes sous forme de nuage de point. Il suffit à l'utilisateur de créer le script correspondant pour la lecture des données (cf. Figure 20, où l'on a utilisé l'application pour afficher un foie humain à partir d'un fichier provenant d'une application médicale).

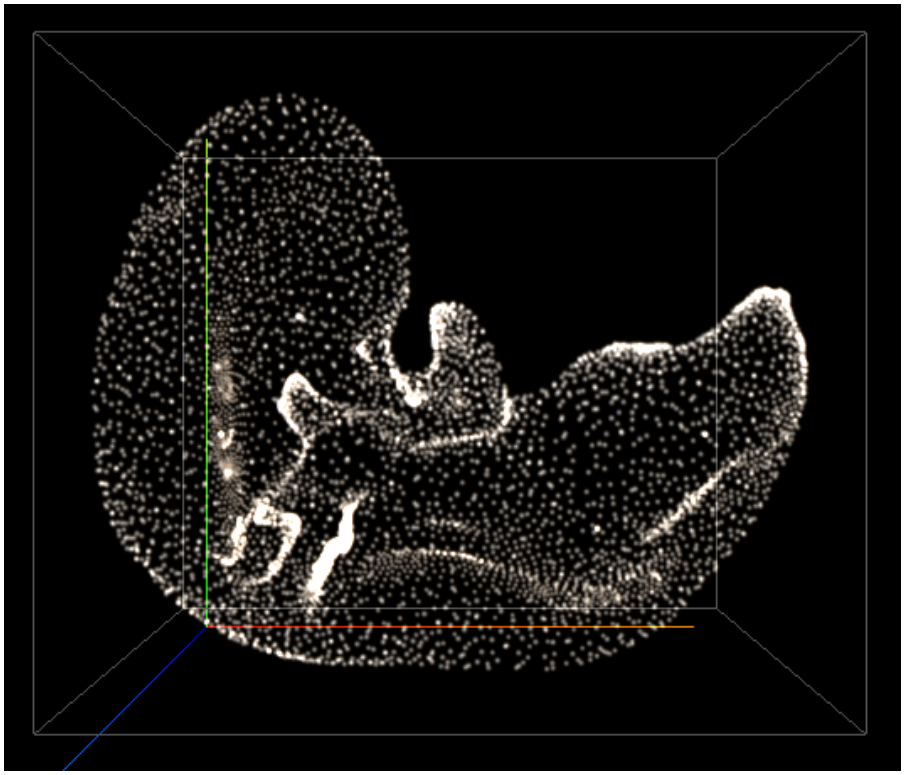
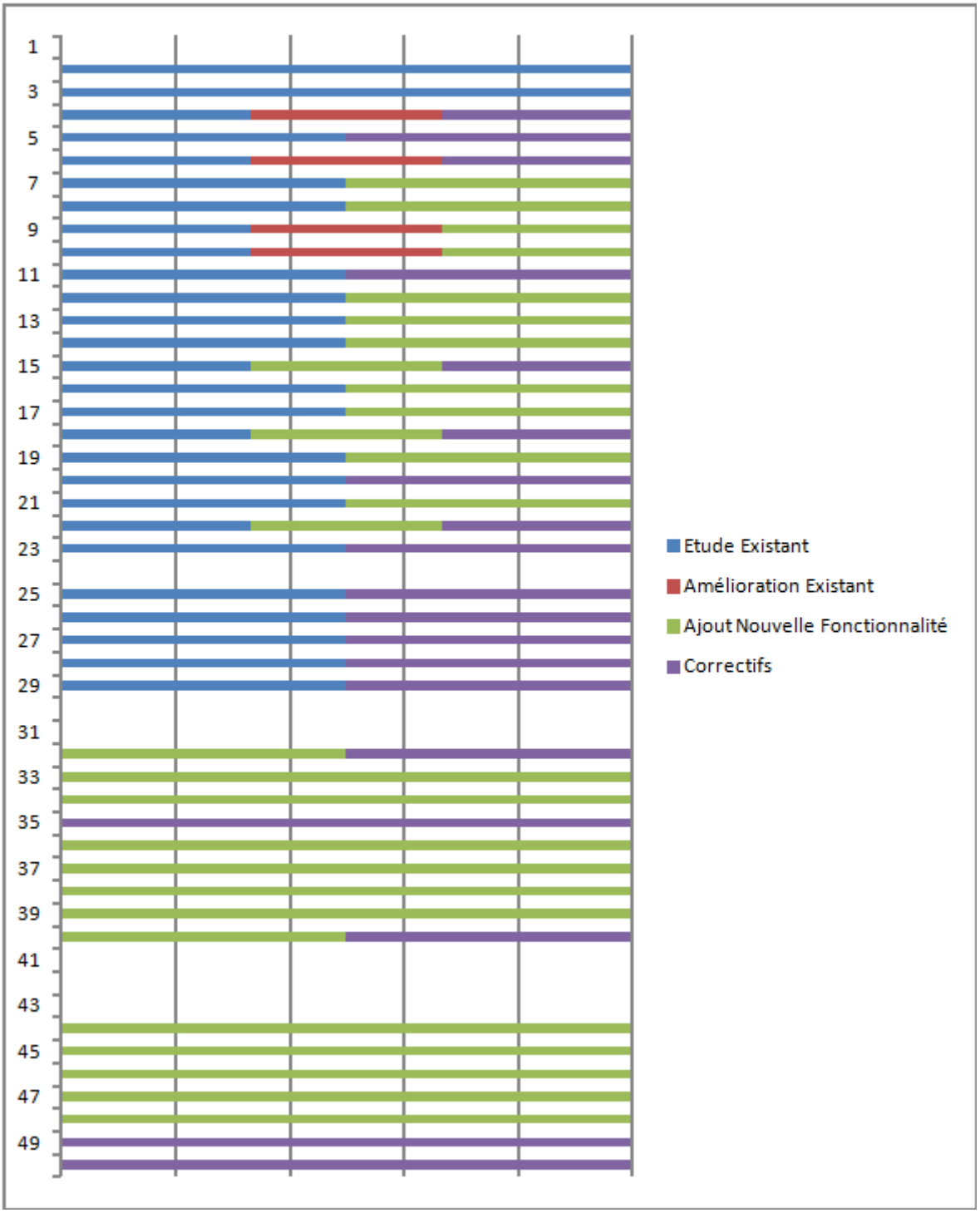


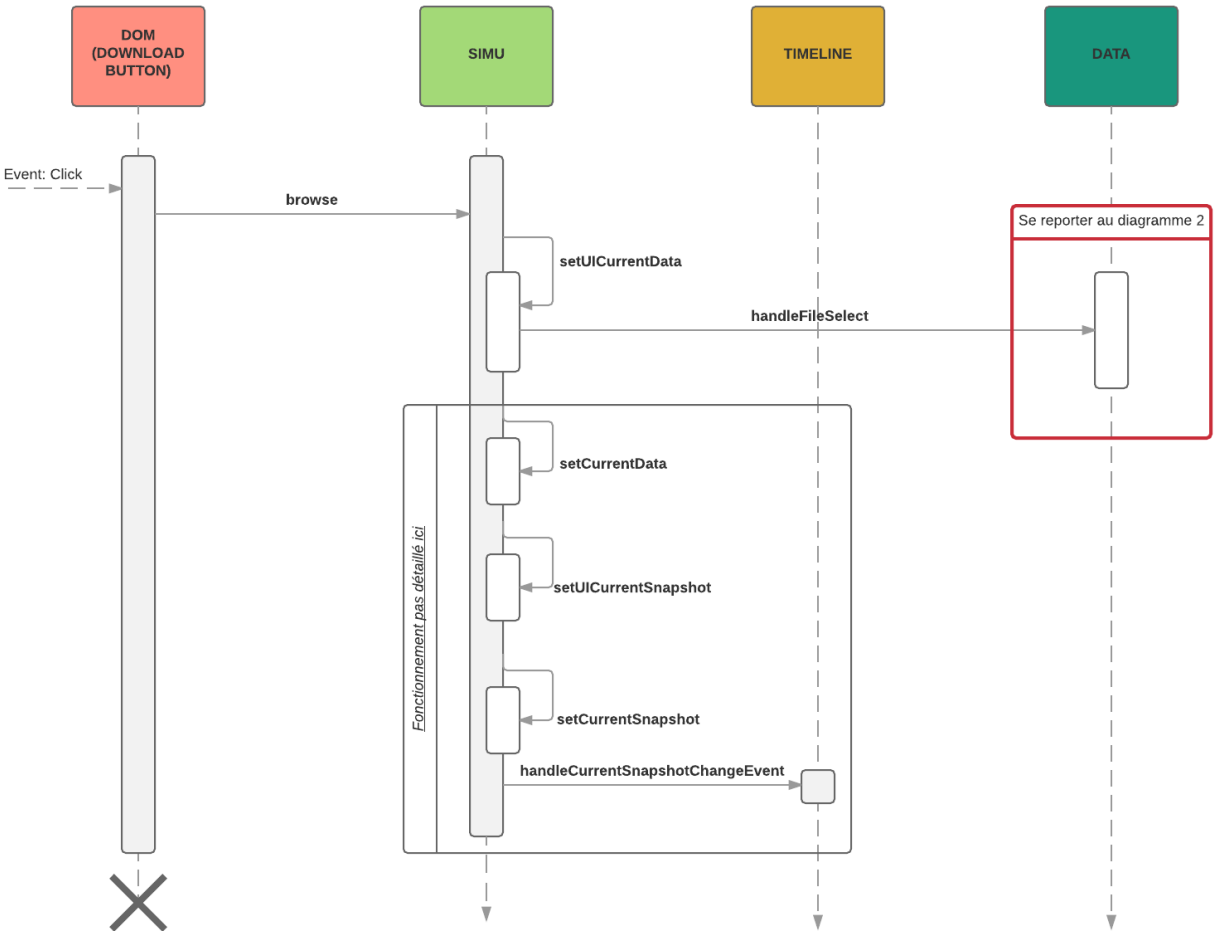
Figure 20

Annexes

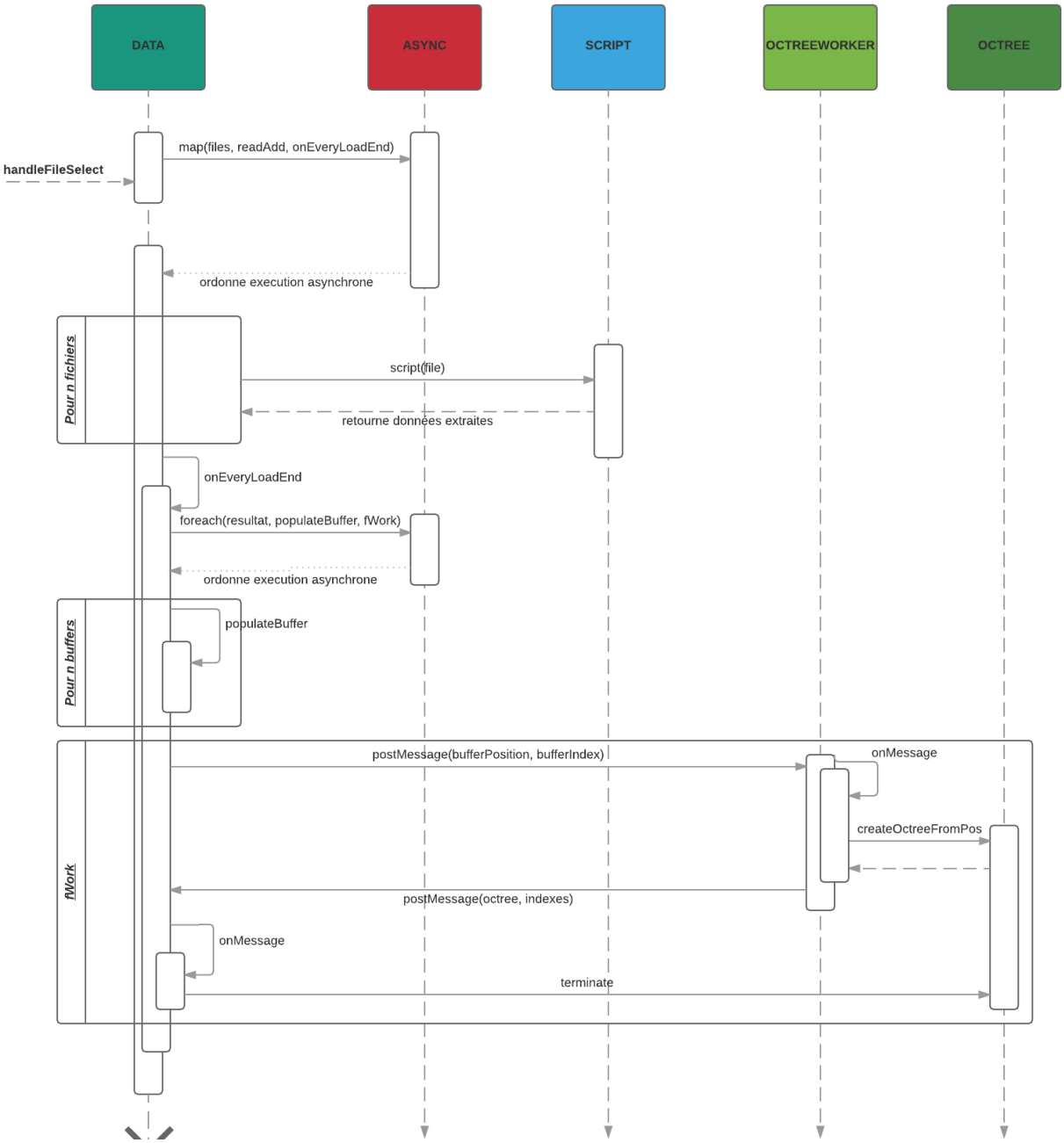
1) Déroulement du stage



2) Diagramme de séquence: chargement des données (no1)



3) Diagramme de séquence: chargement des données (no2)


DATA
ASYNC
SCRIPT
OCTREEWORKER
OCTREE
handleFileSelect
map(files, readAdd, onEveryLoadEnd)
ordonne execution asynchrone
Pour n fichiers
script(file)
retourne données extraites
onEveryLoadEnd
foreach(resultat, populateBuffer, fWork)
ordonne execution asynchrone
Pour n buffers
populateBuffer
fWork
postMessage(bufferPosition, bufferIndex)
onMessage
createOctreeFromPos
postMessage(octree, indexes)
onMessage
terminate

Sources:

Site web de tapVizieR:
http://tapvizier.u-strasbg.fr/adql/

Documentation sur three.js:
http://threejs.org/docs/index.html#Manual/Introduction/Creating_a_scene
https://stemkoski.github.io/Three.js/

Documentation sur dat.gui:
https://workshop.chromeexperiments.com/examples/gui/#1--Basic-Usage

Documentation sur les Webworkers:
https://developer.mozilla.org/fr/docs/Utilisation_des_web_workers

Documentation sur les calculs de Frustum Culling:
http://www.cescg.org/CESCG-2002/DSykoraJJelinek/

Documentation sur async.js:
http://www.sebastianseilund.com/nodejs-async-in-practice