IUT ROBERT SCHUMAN
UNIVERSITE DE STRASBOURG
72 ROUTE DU RHIN
67400 ILLKIRCH-GRAFFENSTADEN
DEPARTEMENT INFORMATIQUE

Rapport de Stage

Evaluation et prototypage avancé de nouvelles technologies

Visualisation 3D dans votre navigateur web grâce au WebGL

AUTEUR : ARNAUD STEINMETZ
Elève en informatique à l'IUT Robert Schuman

MAITRE DE STAGE : ANDRE SCHAAFF
Ingénieur de recherche au CNRS

STRASBOURG, LE 19 JUIN 2015

REMERCIEMENTS

Tout d'abord, je tiens à remercier l'ensemble de l'équipe de l'Observatoire de Strasbourg, qui m'a permis de vivre une expérience professionnelle enrichissante et de passer mon stage dans de très bonnes conditions. Je n'ai pas vu passer ces 10 semaines de stage, tellement j'ai été captivé par un travail intéressant qui m'a permis de développer mes compétences.

Je tenais à remercier tout particulièrement mon maître de stage, Monsieur André SCHAAFF, pour m'avoir accordé sa confiance en m'offrant ce poste. Il a également su répondre à mes différents questionnements et n'a pas hésité à m'apporter son aide pour la confection et la rédaction de ce rapport.

J'adresse également mes remerciements à mon responsable de stage, Monsieur Pierre GANCARSKI, pour avoir effectué le suivi de mon stage. Et qui m'a procuré des conseils judicieux pour l'élaboration de mon rapport.

Mais nombreuses sont encore les personnes m'ayant aidé :

- Nicolas GILLET et Nicolas DEPARIS, pour les indications apportées concernant les données brutes fournies.
- Thomas BOCH, Gilles LANDAIS, pour la présentation des différents services
- L'équipe Skybot 3D : Jérôme BERTHIER et Jonathan NORMAND, pour leurs explications pratiques.
- L'équipe du Planétarium, pour la démonstration du rétroprojecteur 360°.
- Sandrine LANGENBACHER, pour la partie administrative.
- Ainsi que toutes les personnes présentes à ma pré-soutenance.

Réf. rapport :								Entreprise : Observatoire astronomique de Strasbourg
Etudiant : Arnaud STEINMETZ								

Mots clés de l'application :

Visualisation 3D

WebGL

Navigateur

Prototype

Performance

Temps réel

Three.js

Matériel/système informatiques utilisés :

Poste de travail sous le Linux Ubuntu 14.04.

16 Go de RAM

Pas de carte graphique

Logiciels utilisés *(comme support à l'analyse et/ou au développement, y compris les langages)* :

SublimText2 (éditeur de texte),
Google Chrome et Mozilla FireFox (navigateurs)

Enoncé du sujet : Evaluation et prototypage avancé de nouvelles technologies

Résumé :

Le sujet du stage, définit plus précisément durant l'entretien, était l'évaluation de la technologie du WebGL pour réaliser la modélisation de cubes de données en 3D dans un navigateur. L'intérêt de ce stage était de voir si le WebGL, en plus d'offrir une portabilité et un déploiement aisé, disposait de performances assez importantes en vue de représenter un grand nombre de données (étoiles, particules de matière noire, gaz),

Tout d'abord il a fallu comparer les librairies JavaScript permettant d'utiliser la technologie WebGL. Pour cela j'ai d'abord effectué une étude comparative entre Babylon.js et Three.js à l'issue de laquelle j'ai sélectionné Three.js comme étant la plus adaptée.

Ensuite j'ai mis en place différentes méthodes de lecture des données qui m'ont été fournies. Celles-ci provenant de sources différentes, elles respectaient également une syntaxe ainsi qu'un encodage distinct. Ceci impliquait donc la mise en place de plusieurs fonctions de lectures adaptées et optimisées pour chaque type de données.

J'ai par la suite étudié les différentes possibilités de représentation des données dans l'espace, pour trouver celle qui convenait le mieux.

Finalement j'ai étudié les possibilités d'animation des particules afin de mettre celles-ci en mouvement pour créer une véritable simulation 3D en temps réelle dans laquelle il est possible de se déplacer et de contrôler le temps pour observer le déroulement des événements. Le stage s'est conclu par une recherche sur les optimisations et fonctionnalités à implémenter par la suite.

TABLE DES MATIERES

INTRODUCTION

Dans l'optique de valider mon DUT Informatique, j'ai effectué un stage d'une durée de 10 semaines à l'Observatoire astronomique de Strasbourg. Le but de ce stage est de mettre en contact avec le monde du travail, d'avoir une première expérience professionnelle et de mettre en pratique toutes les connaissances acquises durant les 2 années de formation à l'IUT.

L'intitulé du stage était : « Evaluation et prototypage avancé de nouvelles technologies » et proposait d'effectuer des recherches sur 4 technologies différentes. L'une d'entre elles a particulièrement retenu mon attention : la modélisation 3D en WebGL.

L'enseignement dispensé durant le 4ème semestre avait introduit le sujet, mais la complexité et l'étendue des domaines de la 3D et de l'optimisation de l'exécution du code ont nécessitées de nombreux approfondissements et recherches personnelles au fil du développement. Malgré tout l'aspect Recherche et Développement a suscité ma curiosité et m'a toujours poussé à explorer les différentes manières d'appréhender la problématique afin de trouver des solutions toujours plus performantes.

Ce rapport a donc pour but de résumer et expliquer au mieux la manière dont s'est déroulé ce stage selon la structure suivante :

Dans un premier temps, je présenterai l'Observatoire Astronomique de Strasbourg, les différentes équipes et services qui le composent. Cela aura pour but de présenter le contexte dans lequel j'ai travaillé et permettra également de comprendre la démarche qui a donné naissance au sujet du stage.

Je détaillerai ensuite le déroulement du stage, en commençant par souligner les différentes applications déjà existantes. Suivra un état de l'art, qui permettra de mieux cerner le domaine et les outils qui pourront être utilisés. J'aborderai ensuite la partie développement qui a nécessité le plus de temps, et dans laquelle seront précisées les différentes démarches mises en place pour toujours optimiser les performances.

Et je terminerai par un bilan qui fera la liaison entre les prototypes réalisés et les propositions de développement à explorer dans le futur.

I) PRESENTATION DU CADRE

1) L'OBSERVATOIRE ASTRONOMIQUE DE STRASBOURG

1.1) PRESENTATION GENERALE

L'Observatoire astronomique de Strasbourg est un Observatoire des Sciences de l'Univers (OSU) ainsi qu'une Unité Mixte de Recherche de l'Université de Strasbourg et du CNRS. Il est actuellement dirigé par Hervé Wozniak.

Historiquement, Strasbourg a accueilli trois Observatoires astronomiques. Le premier a vu le jour en 1673 grâce à l'astronome Julius Reichlet et se situait sur les tours d'enceinte de la ville. Le second à quant à lui été construit en 1828 sur le site de l'Académie de Strasbourg. Ce n'est qu'en 1881 que l'Observatoire Astronomique de Strasbourg rejoint son emplacement actuel. Celui-ci fait partie, avec le jardin botanique voisin, des actions mises en œuvre par l'empereur Guillaume Ier pour faire de Strasbourg une vitrine suite à la guerre de 1870.

Actuellement, l'Observatoire astronomique de Strasbourg est composé de trois bâtiments dont le principal est celui de la Grande Coupole, visible ci-contre. Ce dernier contient la 3e plus grande lunette astronomique de France. Outre ses propres bâtiments, l'Observatoire héberge également le Planétarium de Strasbourg dont il a eu la responsabilité de 1986 à 2008. Celui-ci fait désormais partie du Jardin des Sciences de l'Université de Strasbourg.

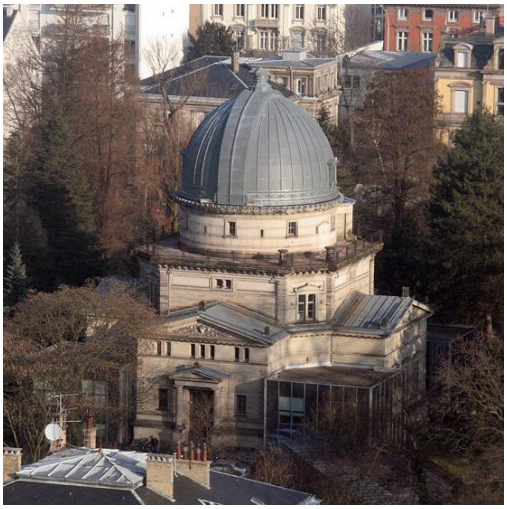


Figure 1 - Bâtiment de la Grande Coupole

L'Observatoire astronomique de Strasbourg est constitué de trois équipes de recherches — Galaxies, Hautes énergies et le CDS — et de deux services d'observation de l'Institut National des Sciences de l'Univers — SSC-XMM et le CDS.

Si à l'origine, l'Observatoire avait pour quasi unique vocation l'observation de comètes, de météorites et d'étoiles variables, ses missions sont bien plus diversifiées à l'heure actuelle. En effet, l'Observatoire astronomique de Strasbourg a pour mission de contribuer au progrès de la connaissance par l'acquisition de données d'observation, le développement de moyens appropriés ou encore l'élaboration des outils théoriques nécessaires. De plus, l'Observatoire est également chargé d'assurer la diffusion des connaissances, d'assurer la formation des étudiants et des personnels de recherche, de prendre part à des activités de coopération internationale, et bien plus encore.

1.2) LES EQUIPES DE RECHERCHES

EQUIPE DE RECHERCHE GALAXIES

Le champ d'action de l'équipe Galaxies regroupe tout ce qui concerne la formation des galaxies ainsi que l'étude et le recensement des populations stellaires qui composent ces galaxies. Plus particulièrement, cette équipe va s'intéresser à notre propre galaxie, la Voie Lactée, ainsi qu'à ses voisines du Groupe Local. Outre les recherches sur les populations stellaires, l'équipe Galaxies s'intéresse également à la dynamique gravitationnelle qui régit les étoiles et les matériaux à l'intérieur des galaxies. Le but de cela est de réunir suffisamment d'informations afin de pouvoir établir l'histoire précise de l'évolution d'un système stellaire et de reconstituer les événements clés dans la vie d'une galaxie.

En plus de cela, l'équipe Galaxies mène des recherches sur les amas stellaires — composants fondamentaux d'une galaxie — afin de mieux comprendre la formation de celles-ci.

Enfin, l'équipe s'implique également dans des missions satellitaires telles que Gaia, mission lancée en décembre 2013 par l'Agence Spatiale Européenne (ESA) ayant pour but de fournir des relevés sur la Voie Lactée.

EQUIPE DE RECHERCHE HAUTES ENERGIES

L'équipe Hautes énergies s'intéresse aux sources émettrices de rayon X, aux objets compacts — étoiles à neutrons par exemple — et aux noyaux actifs des galaxies. Elle est impliquée dans le Survey Science Center d'XMM (SSC-XMM), un consortium international de laboratoires sélectionnés par l'ESA qui est en charge de fournir les 8 catalogues complets d'objets observés par le satellite XMM-Newton. L'équipe Hautes énergies participe également à des projets communautaires tels que le projet européen Arches, en collaboration avec le CDS.

CENTRE DE DONNEES ASTRONOMIQUES DE STRASBOURG (CDS)

Le Centre de Données astronomiques de Strasbourg est à la fois un service d'observation et une équipe de recherche. Le CDS a entre autre développé des services d'accès aux données astronomiques — Simbad, VizieR — et de visualisation de ces mêmes données — Aladin — qui sont utilisés par l'ensemble de la communauté internationale. C'est aussi l'un des acteurs majeurs du développement de l'International Virtual Observatory Alliance (IVOA). En 2008, le CDS a été labellisé « Très Grande Infrastructure de Recherche », ce qui le place au même niveau que des infrastructures européennes comme l'European Southern Observatory (ESO).

2) LE CDS

2.1) PRESENTATION GENERALE

Crée en 1972, le CDS est d'abord connu sous le nom de Centre de Données Stellaires avant de devenir le Centre de Données astronomiques de Strasbourg en 1983. Hébergeant entre autres la base de données de référence pour l'identification d'objets astronomiques, ces services — Simbad, VizieR et Aladin — sont largement utilisés par la communauté astronomique internationale.

Figure 2 – Logo du CDS

Ses missions sont nombreuses et comprennent entre autres :

- le rassemblement des informations utiles concernant les objets astronomiques,
- la mise à jour de ces données,
- la distribution de ces données à la communauté internationale,
- la mise en place de recherche à propos de ces données.

Le CDS emploie actuellement 33 personnes réparties de la façon suivante : 8 chercheurs/astronomes-adjoints, 1 chercheur postdoctoral, 11 informaticiens, 10 documentalistes et 3 administratifs.

2.2) LES DIFFERENTS SERVICES DU CDS

SIMBAD

Simbad est la base de données de référence mondiale en ce qui concerne la nomenclature et la bibliographie d'objets astronomiques. Possédant l'équivalent de plus de 40 ans de données, Simbad permet aux astronomes de trouver facilement des informations sur plus de 8 millions d'objets grâce à plus de 300 000 références bibliographiques.

Simbad dispose également d'un résolveur de noms permettant de retrouver n'importe quel objet désigné par l'un des 22 millions d'identifiants que la base contient.

Simbad a reçu en moyenne 360 000 requêtes par jour pour l'année 2014, soit plus de 4 requêtes par secondes.

Figure 3 – Logo de Simbad

VIZIER

VizieR est une base de données regroupant des catalogues d'objets astronomiques. Ces catalogues sont constitués de données relevées durant des missions d'observation et sont ajoutés à VizieR par les documentalistes du CDS. A l'heure actuelle, la base est constituée de plus de 13 000 catalogues.

Interrogeable sur de nombreux critères (longueur d'ondes, nom de la mission, etc.), VizieR permet de rassembler et d'homogénéiser les données astronomiques afin de pouvoir les comparer et les exporter.

En 2014, le nombre de requêtes faites sur la base de données ont été en moyenne de 300 000 par jour, avec des pics à plus de 2 millions.

Figure 4 – Logo de VizieR

ALADIN

Aladin est un atlas interactif du ciel permettant de visualiser et de comparer des images du ciel. Il a été entièrement développé en Java par Pierre Fernique, Thomas Boch et François Bonnarel.

Aladin utilise des images provenant d'observatoires au sol ou spatiaux et génère des cartes du ciel en trois dimensions (technologie HEALPix). Ces images peuvent être issues de la base de données inclus à Aladin, d'autres bases telles que la NASA Extragalactical Database, ou encore provenir de l'utilisateur lui-même. A l'heure actuelle, Aladin est disponible sous quatre formes : une application téléchargeable, une applet Java, une version JavaScript et un simple previewer en ligne.

Figure 5 – Logo d'Aladin

2.3) INTERNATONAL VIRTUAL OBSERVATORY ALLIANCE

L'International Virtual Observatory Alliance (IVOA) est une organisation internationale regroupant 19 projets d'Observatoire Virtuel (VO) provenant de 17 pays différents : Arménie, Australie, Brésil, Canada, Chine, France, Allemagne, Hongrie, Inde, Italie, Japon, Russie, Ukraine, Espagne, Royaume-Uni, Argentine et États Unis.

Ce projet international a un but similaire au World Wide Web Consortium : mettre en place des standards et décider quels seront les travaux les plus importants à faire ainsi que les stratégies à adopter. Ceci permet de faciliter les échanges internationaux et la collaboration nécessaire à l'élaboration de standards d'interopérabilité et au déploiement d'outils, de systèmes et de structures organisés nécessaires pour permettre une utilisation internationale des archives astronomiques (images, catalogues, etc..). Telle est la mission confiée à l'IVOA lors de sa création en juin 2002.

Le CDS participe activement à ce projet notamment par l'implémentation des standards du VO dans les services cités précédemment : Aladin, Simbad et VizieR.

Figure 6 – Logo de l'IVOA

II) DEROULEMENT DU STAGE

1) MISE EN CONTEXTE

1.1) SUJET DU STAGE

Le travail demandé s'inscrit donc dans un cadre de veille technologique que suit le CDS depuis de nombreuses années afin d'améliorer en permanence la qualité de ses services.

L'intitulé exact du stage était « Evaluation et prototypage avancé de nouvelles technologies ».

Il était question de choisir entre 4 technologies différentes :

- L'automatisation du déploiement d'applications dans des conteneurs logiciels, avec Docker
- La visualisation 3D en utilisant WebGL, avec Babylon.js
- La découverte d'un framework JavaScript, avec Angular.js
- La construction de sites Web « plus » sémantiques, avec Semantic UI

J'ai finalement décidé durant l'entretien, en accord avec mon maître de stage, de me focaliser sur la visualisation 3D en utilisant WebGL. Ce choix fut motivé par le fait qu'une grande partie de ma seconde année à l'IUT eu pour sujet la géométrie dans l'espace et la modélisation en WebGL. Etant donné que cela m'avait beaucoup intéressé, j'ai saisi l'occasion d'approfondir mes connaissances en la matière.

Ce stage était plutôt axé sur l'évaluation des performances pouvant être atteintes avec le WebGL, et demandait la création de prototype pouvant servir de démonstration.

1.2) LES BESOINS

André SCHAAFF m'a donc expliqué ce qu'il attendait plus en détail. Ce qui intéressait le CDS était de modéliser des cubes de données pour représenter des résultats de simulations pouvant concerner plusieurs millions d'objets. Cela permettrait d'interpréter plus facilement les énormes jeux de données que possède le CDS.

Je me suis donc renseigné sur le type de rendu attendu en faisant quelques recherches. On peut donc voir ci-dessous 3 cubes de données.

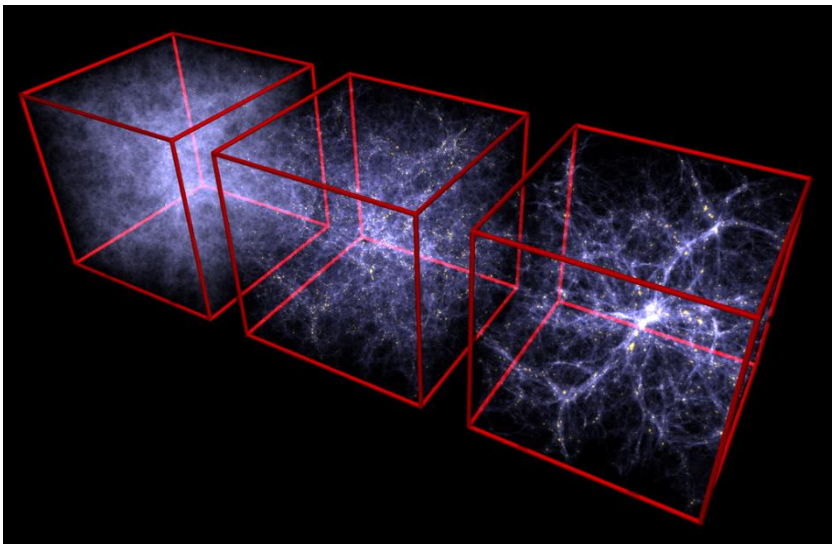


Figure 7 – Structure cosmique en formation dans un composant gazeux

1.3) L'EXISTANT

Toujours dans le but de saisir le rendu des prototypes que j'avais à créer, je me suis renseigné sur les différentes applications existantes actuellement.

SKYBOT3D

Il s'agit d'une extension du service SkyBoT (Sky Body Tracker) qui a été créée dans le but d'explorer le système solaire dans un environnement 3D. Ce logiciel est utilisé pour apprendre la structure des petits objets du système solaire ainsi que pour visualiser les différentes classes d'astéroïdes présentes. La précision fournie par Skybot3D en fait un outil parfaitement adapté aux scientifiques intéressés par un rapide aperçu du système solaire à une époque donnée. Ci-dessous se trouve un aperçu de ce à quoi ressemble Skybot3D.

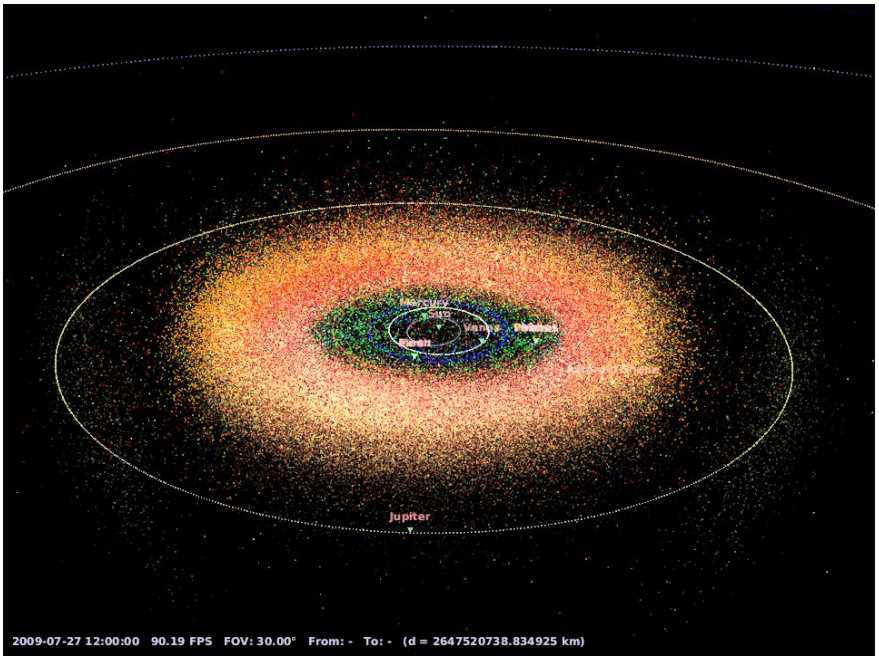


Figure 8 – Représentation du système solaire par Skybot3D

STELLARIUM 360

Cet outil est un logiciel pour planétarium open source. Il permet d'afficher un ciel réaliste en 3D, comme si vous le regardiez à l'œil nu, aux jumelles ou avec un télescope. Il est notamment utilisé par le planétarium de Strasbourg, où j'ai eu la chance de participer à une démonstration.

Ce logiciel est compatible avec les rétroprojecteurs capables de projeter une image à 360° sur voûte hémisphérique, ce qui permet donc une immersion toujours plus grande. Sur la figure 9 présente ci-contre, nous avons un exemple de rendu.

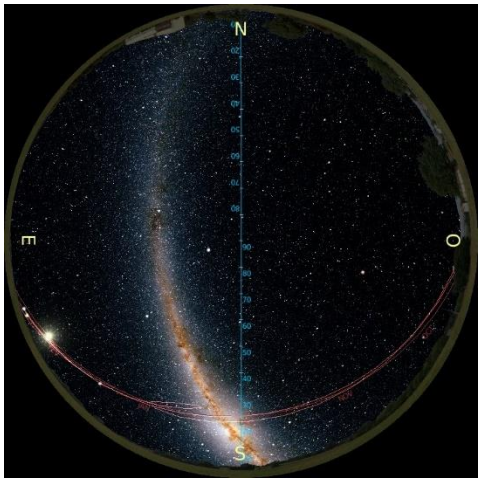


Figure 9 – Voie lactée par Stellarium 360

En 2014, un stagiaire avait réalisé un travail très similaire à ce qui m'a été demandé, seulement son étude était plus portée sur le développement pour la visualisation par l'utilisation de masque de réalité virtuelle. Mais il s'était également basé sur des cubes de données. Dans son cas, on peut voir sur la figure ci-dessous, que les données étaient représentées sous la forme de cubes texturés.

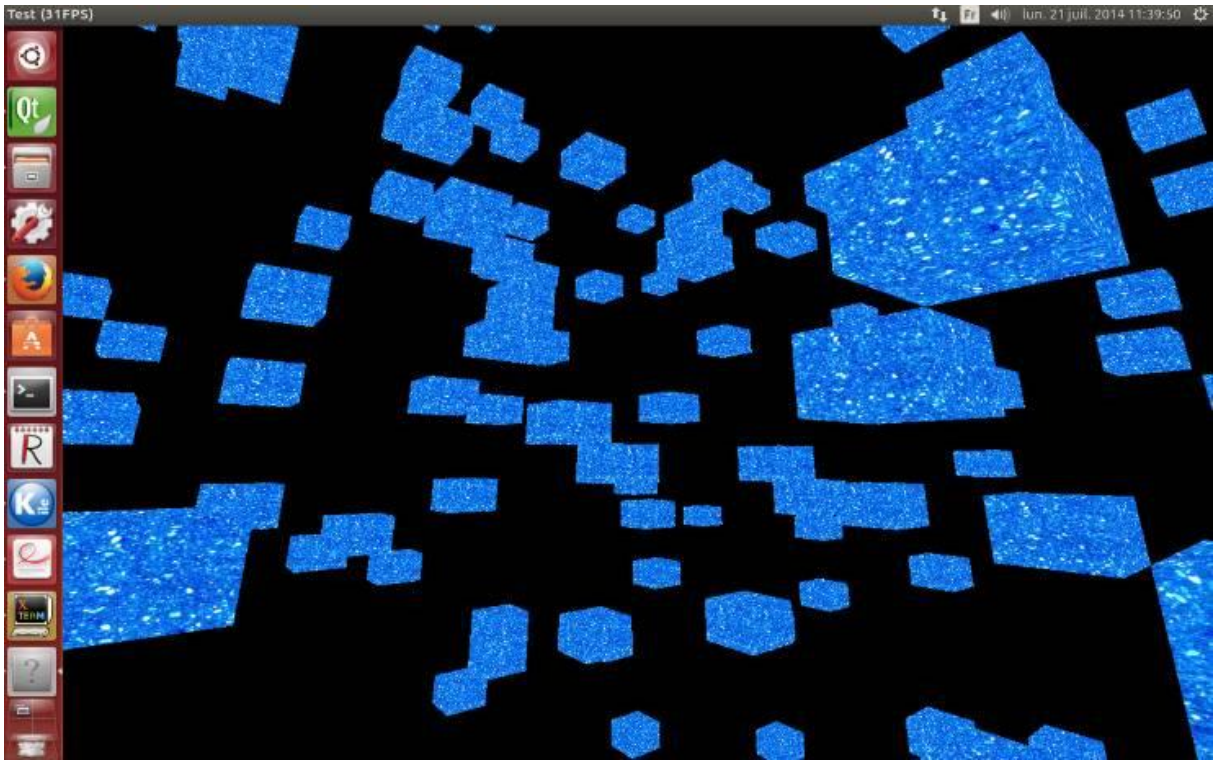


Figure 10 – Visualisation d'un jeu de données avec l'outil concerné

Tous ces exemples ont une chose en commun, ils ont tous été écrit grâce au langage C et à l'utilisation d'OpenGL. Or ce langage doit être compilé pour fonctionner, et doit se trouver directement sur la machine de l'utilisateur. Ce qui pose des problèmes quant à la portabilité de l'application. C'est pour cela que le CDS cherche à développer des prototypes similaires en WebGL pour justement pouvoir partager et utiliser plus facilement l'application. Car pour des personnes non initiées, la mise en place et la compilation d'une application en C peut très vite devenir un casse-tête.

2) OUTILS UTILISES

2.1) CODAGE

Durant le stage, j'ai disposé d'une machine sous le système d'exploitation Linux. Celle-ci possédait une mémoire vive de 16Go et ne possédait pas de carte graphique. A la place, la puce graphique du CPU était mise à contribution. Ce détail est important, car si l'utilisateur possède une carte graphique, les résultats seront bien meilleurs.

Pour développer en JavaScript, étant donné que le stage était principalement axé sur la recherche et le prototypage, je n'ai pas jugé nécessaire l'utilisation d'un IDE. J'ai donc opté pour SublimText2, qui est un éditeur de texte générique. A celui-ci, j'ai également ajouté l'utilisation d'un plugin connu sous le nom de « Emmet », qui fournit des fonctionnalités supplémentaires permettant de coder plus facilement.

2.2) ANALYSE ET MONITORING

Mon stage étant centré sur l'évaluation des performances des différents prototypes que je mettais en place, il a fallu trouver des moyens de visualiser le comportement de l'application.

Pour cela j'ai utilisé principalement les outils de développement mis à disposition par le navigateur.

- FireBug, qui est une extension pour Mozilla FireFox, qui permet de déboguer et contrôler le HTML, le CSS, le DOM et le JavaScript d'une page web.
- Les outils de développement Chrome :
    - Timeline, qui permet d'enregistrer pendant un court instant tous les appels de fonctions qui sont effectués. (cf. figure annexe 1).
    - Profiles, qui permet d'établir différents profils. Il permet notamment d'enregistrer les allocations effectuées sur le tas. (cf. figure annexe 2).
- Un Objet de Three.js appelé ''Stats'', permettant d'afficher le nombre de FPS de l'application en temps réel

J'ai également tenté de trouver des outils permettant d'observer l'utilisation des performances de la GPU, mais je n'ai malheureusement pu trouver que des utilitaires compatibles avec des IDE de développement en C.

2.3) DOCUMENTATION

Pour effectuer la documentation du code, et permettre la réutilisation de celui-ci, j'ai opté pour l'utilisation de JSDoc, qui permet d'annoter le code source en respectant une certaine syntaxe. Une fois les commentaires entièrement rédigés, il suffit de générer la documentation qui sera accessible à partir d'un fichier HTML, sous la même forme que pour un site Web.

J'ai également tout au long du stage, rédigé des comptes rendus sur mon avancée jour après jour dans le projet, en y indiquant les problèmes rencontrées ainsi que les démarches effectuées pour essayer de les résoudre. Ces comptes rendus sont disponibles sur l'intranet de L'Observatoire où chaque stagiaire disposait d'un Wiki pour y indiquer son avancement.

3) ETAT DE L'ART

3.1) WEBGL

WebGL est une spécification d'affichage 3D pour les navigateurs web. Elle permet d'utiliser le standard OpenGL au sein d'une application basée sur le standard HTML5 en s'aidant du langage JavaScript et de l'accélération matérielle (GPU) pour les calculs et le rendu 3D.

L'utilisation de cette spécification plutôt que l'OpenGL permet donc de déployer bien plus facilement les projets, en les hébergeant sur un serveur. Ce qui apporte plus de portabilité ainsi que d'accessibilité pour les utilisateurs.

Une autre question à se poser était pourquoi choisir WebGL par rapport à ses concurrents dans le domaine du Web, et notamment Flash. Cela fut très simple à déterminer.

En trois points essentiels, le WebGL :

- Fonctionne sans plugin (donc pas de mises à jour nécessaire, ni de connexion internet)
- Est basé sur HTML5 qui est un standard du W3C
- Intègre un moteur 3D très complet et bien plus fonctionnel que ses autres concurrents

3.2) THREE.JS ET BABYLON.JS

Pour développer de façon plus efficace, il fallait avoir recours à une bibliothèque JavaScript. Dans le sujet du stage, André SCHAAFF m'avait proposé d'utiliser Babylon.js, mais de mon côté j'en connaissais une autre appelé Three.js.

J'ai donc décidé de réaliser une comparaison entre ces deux bibliothèques pour m'assurer d'utiliser celle qui conviendrait le mieux à mes besoins. J'ai tout d'abord établi un tableau comparatif entre ces dernières.

	Three.js	**Babylon.js**
Date de sortie	Avril 2009	Eté 2013
Licence	Open Source	Open Source
Documentation/Tutoriels	Bien expliquée/Nombreux	Mal documenté/faibles
Github	~10 300 commits 443 contributeurs	~1500 commits 48 contributeurs
Stackoverflow	~6400 tags	46 tags

En premier lieu la chose importante à noter est le simple fait que ces deux bibliothèques soient Open Source, car pour rester dans l'optique de développement, de partage et d'ouverture du CDS, c'est un élément à ne pas négliger.

Ensuite on peut également constater l'ancienneté de Three.js, qui a été créée 4 ans avant Babylon.js, ce qui va expliquer certain défauts de la seconde bibliothèque.

Tout d'abord j'ai remarqué que pour Three.js, il y avait un nombre très important d'exemples et de démonstrations disponibles en ligne, ce qui est un grand avantage pour comprendre le fonctionnement d'une bibliothèque. De plus depuis 2012/2013, les contributeurs de Three.js ont passé beaucoup de temps à l'élaboration d'une documentation qui est désormais assez complète. A

l'opposé la documentation de Babylon.js laisse les utilisateurs dans le doute, lorsqu'ils recherchent à comprendre ce que fait exactement une méthode. Si l'on veut vraiment comprendre, il faut inspecter le code source, ce qui est très peu pratique et prend énormément de temps.

L'ancienneté de Three.js par rapport à Babylon.js s'illustre également par d'autres faits, comme le nombre de commits sur les codes sources qui sont hébergés sur GitHub. D'un côté on peut voir Three.js avec plus de 10300 commits et 443 contributeurs et de l'autre, 1500 commits avec 48 contributeurs.

Egalement sur Stackoverflow, qui est un forum d'entraide dans le domaine informatique et qui recense de nombreuses questions. On peut voir que l'on trouve 6400 questions comportant le tag Three.js, tandis qu'il n'y en a que 46 pour Babylon.js.

On peut donc noter une très grande différence de popularité ainsi qu'une facilité d'utilisation non équilibrée entre les deux bibliothèques.

Mais pour prendre une décision appropriée, il fallait évaluer les performances des deux Bibliothèques. J'ai d'abord effectué des démos pour chacune d'entre elles, pour comprendre le fonctionnement général, et les éléments de base de chaque bibliothèque. Cela m'a pris un temps assez conséquent, car durant les cours à l'IUT nous n'avions qu'aborder les bases de Three.js.

J'ai ensuite mis en place une démo effectuant le même traitement avec les deux bibliothèques pour voir laquelle des deux était la plus rapide dans la mise en place de la scène, et laquelle offrait les meilleures résultats en terme de Frame Par Seconde.

Three .js	Babylon.js
Chargement de 3282 éléments en 52 ms Avec 17 FPS	Chargement de 3282 éléments en 1177 ms Avec 5.5 FPS

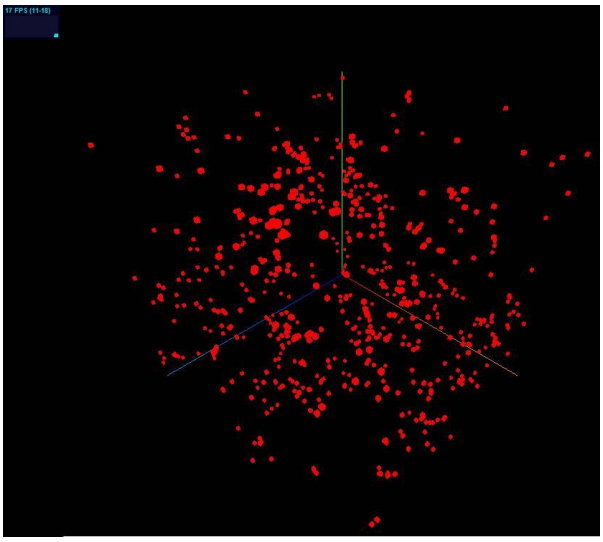
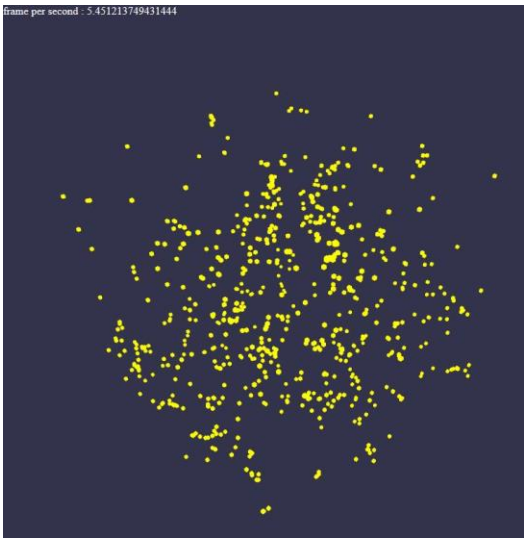

	

Figure 11 – Représentation Three.js	Figure 12 – Représentation Babylon.js

Dans l'exemple ci-dessus, j'ai créé une scène à laquelle j'ai ajouté un certain nombre d'éléments. On peut voir dans la version avec ThreeJS que le temps de chargement initial (le temps entre le chargement de la page et l'affichage des éléments dans la scène) est bien moindre comparé à celui de BabylonJS. Le même constat est fait du côté des FPS où ThreeJS triple les performances.

De ce fait j'ai finalement opté pour un développement de l'application avec Three.js. Cette dernière apportant une facilité de codage, une plus grande performance, et une éventuelle aide plus facilement accessible.

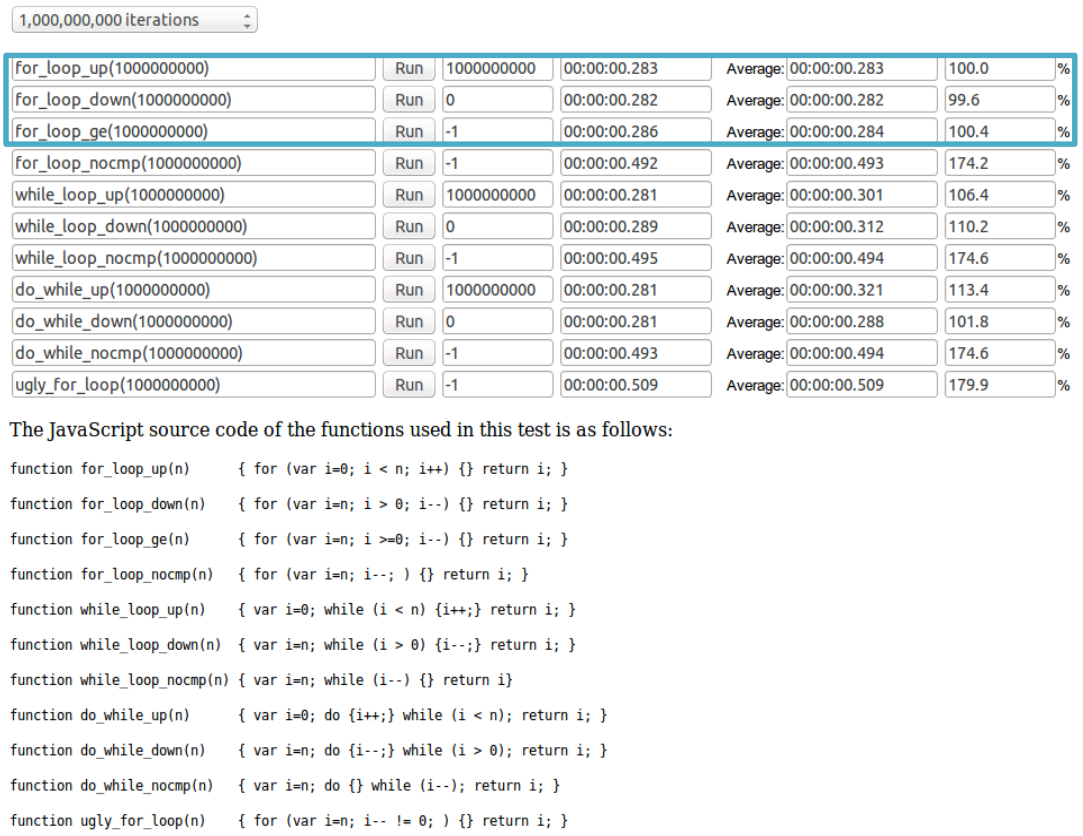
4) DEVELOPPEMENT

4.1) BONNES PRATIQUES JAVASCRIPT

Après avoir effectué les différentes comparaisons entre Three.js et Babylon.js, j'ai vite remarqué qu'il y aurait de nombreuses boucles dans le code. J'ai donc passé beaucoup de temps au préalable de la création des prototypes, à chercher des optimisations et bonnes pratiques à mettre en œuvre en JavaScript.

J'ai remarqué que pour les boucles itérant un grand nombre de fois, de petits changements permettaient d'obtenir des résultats bien plus convaincants.

Pour effectuer une itération, il est possible de s'y prendre de différentes manières, j'ai donc effectué des comparaisons entre les différentes méthodes, visible ci-dessous :



1,000,000,000 iterations

for_loop_up(1000000000)	Run	1000000000	00:00:00.283	Average:	00:00:00.283	100.0 %
for_loop_down(1000000000)	Run	0	00:00:00.282	Average:	00:00:00.282	99.6 %
for_loop_ge(1000000000)	Run	-1	00:00:00.286	Average:	00:00:00.284	100.4 %
for_loop_nocmp(1000000000)	Run	-1	00:00:00.492	Average:	00:00:00.493	174.2 %
while_loop_up(1000000000)	Run	1000000000	00:00:00.281	Average:	00:00:00.301	106.4 %
while_loop_down(1000000000)	Run	0	00:00:00.289	Average:	00:00:00.312	110.2 %
while_loop_nocmp(1000000000)	Run	-1	00:00:00.495	Average:	00:00:00.494	174.6 %
do_while_up(1000000000)	Run	1000000000	00:00:00.281	Average:	00:00:00.321	113.4 %
do_while_down(1000000000)	Run	0	00:00:00.281	Average:	00:00:00.288	101.8 %
do_while_nocmp(1000000000)	Run	-1	00:00:00.493	Average:	00:00:00.494	174.6 %
ugly_for_loop(1000000000)	Run	-1	00:00:00.509	Average:	00:00:00.509	179.9 %

The JavaScript source code of the functions used in this test is as follows:

```
function for_loop_up(n)      { for (var i=0; i < n; i++) {} return i; }
function for_loop_down(n)    { for (var i=n; i > 0; i--) {} return i; }
function for_loop_ge(n)      { for (var i=n; i >=0; i--) {} return i; }
function for_loop_nocmp(n)   { for (var i=n; i--; ) {} return i; }
function while_loop_up(n)    { var i=0; while (i < n) {i++;} return i; }
function while_loop_down(n)  { var i=n; while (i > 0) {i--;} return i; }
function while_loop_nocmp(n) { var i=n; while (i--) {} return i}
function do_while_up(n)      { var i=0; do {i++;} while (i < n); return i; }
function do_while_down(n)    { var i=n; do {i--;} while (i > 0); return i; }
function do_while_nocmp(n)   { var i=n; do {} while (i--); return i; }
function ugly_for_loop(n)    { for (var i=n; i-- != 0; ) {} return i; }
```

Figure 13 – Tests de performance d'itération

On peut remarquer qu'en moyenne le type de boucle le plus rapide est la boucle for, quel que soit le sens de l'itération. J'ai également constaté que stocker la valeur maximum (ou minimum selon le sens) dans une variable, permet d'obtenir des gains de performances de l'ordre de quelque ms. Ceci est dû au fait que la variable sera stockée en cache et donc plus rapide d'accès par rapport à un accès du type ''array.length''.

Lorsqu'il faut accéder à une variable globale à plusieurs reprises, il vaut mieux créer une variable locale pour y accéder plus rapidement. Un exemple est visible ci-dessous.

```
var blah = document.getElementById('myID'),
blah2 = document.getElementById('myID2');
```

to:

```
var doc = document,
blah = doc.getElementById('myID'),
blah2 = doc.getElementById('myID2');
```

Figure 14 – Déclaration de variable en local

La figure 15 ci-dessous nous montre également l'importance de déclarer des variables intermédiaires lorsque l'on recherche, à accéder plusieurs fois, à des attributs stockés en profondeur. On remarque qu'il y a une grande inégalité entre les différents navigateurs. Les courbes sont par contre à peu près similaires dans le cas de la lecture et de l'écriture.

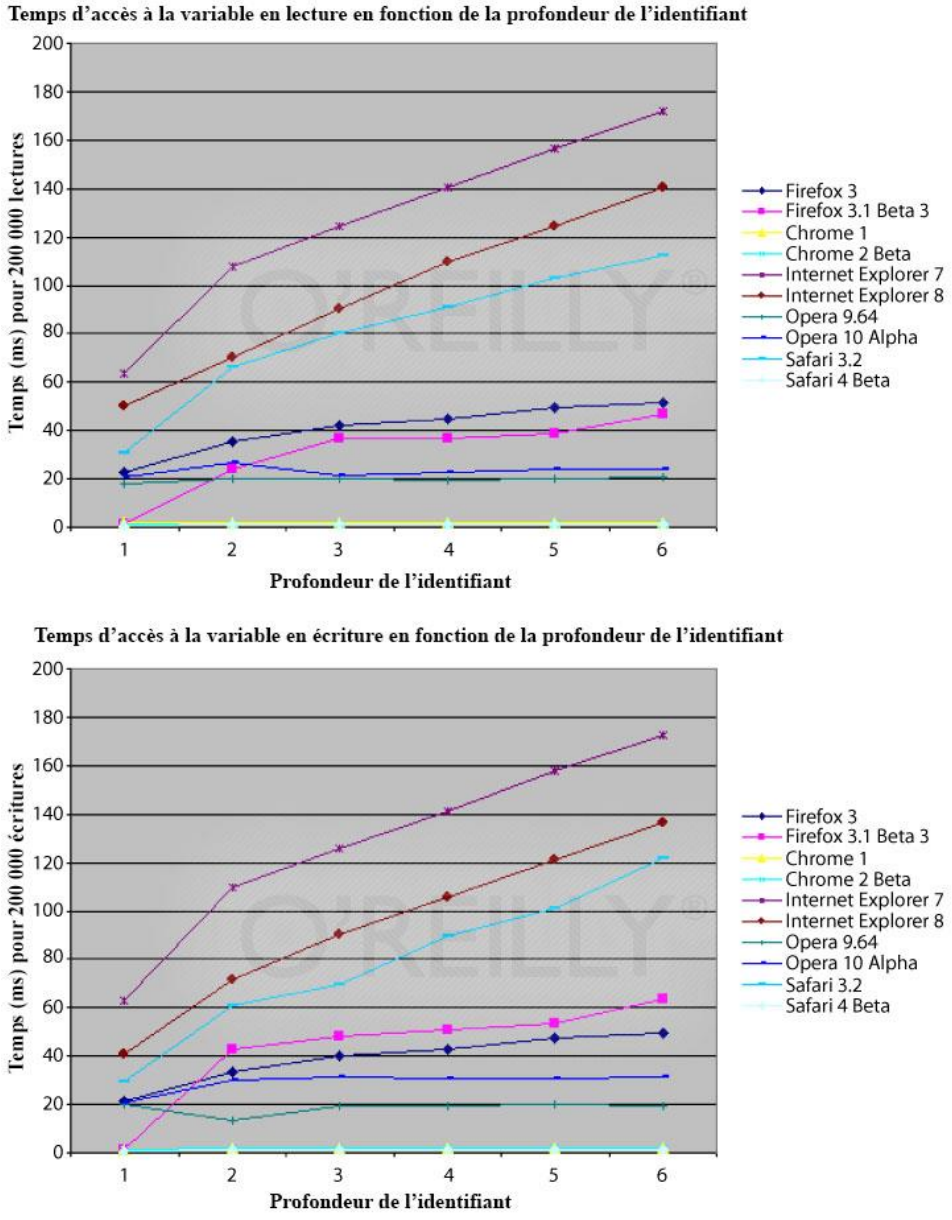


Figure 15 – Temps d'accès à la variable en fonction de la profondeur de l'attribut

Une dernière astuce que j'ai découverte, est l'utilisation de la valeur « null », pour aider à la gestion de la mémoire. Pour comprendre pourquoi, il faut savoir comment fonctionne la mémoire cache en JavaScript. Dans ce langage, il n'y a pas autant de possibilités pour gérer l'allocation de la mémoire comme en C. Le JavaScript utilise un « garbage collector » qui permet d'éliminer les variables non utilisées. Le programmeur n'a donc pas la main sur la mémoire. Néanmoins en affectant la valeur null à une variable dont on aura plus d'utilité, il est possible d'indiquer au « garbage collector » les variables qu'on ne voudra plus utiliser.

Toutes ces petites astuces mises bout à bout permettent d'augmenter la qualité d'écriture du code tout en améliorant sa performance générale. Le gain de temps est infime dans le cas de boucle n'itérant que très peu. Mais lorsque les volumes de données sont de plus en plus importants, cela permet de gagner de plus en plus de temps d'exécution.

4.2) ELEMENTS FONDAMENTAUX DE THREE.JS

Pour mieux comprendre le fonctionnement de la bibliothèque Three.js, cette partie sera consacrée aux différents composants entrant en jeu lors de la création d'un rendu 3D.

Tout d'abord il faut spécifier le type de caméra à utiliser, il en existe deux :

La caméra en perspective garde les mêmes propriétés que la manière dont on voit les objets dans la réalité. C'est-à-dire que les objets sont de plus en plus petits selon la distance à laquelle ils se trouvent dans la scène. Tandis que la caméra orthogonale (ou orthographique) ignore cet effet pour permettre d'effectuer des mesures de plus grande précision. La figure présente ci-dessous illustre bien cette différence.

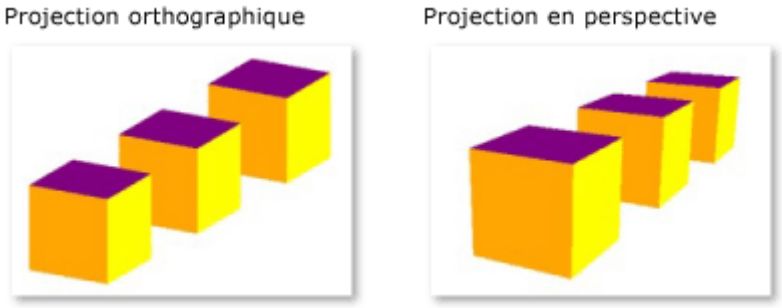


Figure 16 – Illustration de deux types de projections

Dans notre cas, il est important de garder la sensation de profondeur apportée par la caméra en perspective, car les simulations sont censées représenter le plus possible la réalité. Il a donc été nécessaire de garder un effet de profondeur pour le rendu.

A présent que nous avons la caméra qui donne un cadre à ce qui est vu à l'écran, il faut avoir une scène dans laquelle seront placés les étoiles, particules, ou tout autre élément. Pour cela il suffit de déclarer un objet de type « Scene ».

Pour effectuer le lien entre la caméra et la scène il faut utiliser un « renderer » (cf. figure annexe 3), qui correspond au rendu en français. Il calculera l'image 2D à afficher à l'écran à partir de la scène comportant la position des éléments dans un espace 3D et des paramètres de la caméra. Pour avoir un rafraichissement de l'image à l'écran, il faut ajouter une boucle de rendu qui sera appelée indéfiniment pour permettre d'avoir un rendu en temps réel. Cette boucle est communément appelée « render() » ou « animation() » et dans sa forme la plus simple, se divise en deux parties. Une partie s'occupe de rappeler la fonction « render() », ceci est effectué par l'utilisation de la

méthode « requestAnimationFrame() » qui permet d'indiquer au navigateur que l'on veut effectuer une animation et lui demande d'exécuter une fonction de mise à jour qui est passée en paramètre, en l'occurrence la fonction « render() ». Mais ce que « requestAnimationFrame() » apporte également, c'est lorsque le focus n'est pas sur la page, la boucle arrête d'être exécutée et l'animation se met en pause pour éviter une consommation de ressource inutile.

La seconde partie de la fonction de rendu concerne le « renderer », sur lequel on applique une méthode également appelée « render » mais qui est propre à l'objet « renderer ». Cette méthode nécessite deux paramètres d'entrée qui sont la caméra et la scène.

En plus de ces deux actions, on peut ajouter d'autres instructions à exécuter comme la mise à jour des contrôles utilisée pour le déplacement de la caméra. On peut également calculer le temps requis entre deux exécutions de la boucle pour effectuer un rafraichissement. Pour avoir un rendu en temps réel fluide, il faudrait que la boucle ne prenne que 17 ms à s'exécuter, il faut donc éviter de la surcharger.

Pour terminer cette section, je voudrais également mentionner l'utilisation de Dat-GUI (cf. figure annexe 4) qui est une interface utilisateur légère permettant d'interagir avec les variables de notre code JavaScript. Cette bibliothèque n'est pas directement liée à Three.js, mais est souvent utilisée en complément. Ce qui est très intéressant, c'est que l'interface permet une communication à double sens. C'est-à-dire qu'elle permet de modifier la valeur de variables. Mais également que lorsque la valeur d'une variable change durant l'exécution du code, l'interface prend en compte ce changement et l'indique à l'utilisateur. Cette fonctionnalité est utile dans le cas d'une variable représentant le temps qui se mettrait à jour dans le code, et qu'on pourrait remettre à 0 grâce à l'interface.

J'ai regroupé l'instanciation de tous ces objets dans une méthode appelée « init() », qui permet comme son nom l'indique de mettre en place tous les éléments nécessaires au bon fonctionnement de l'application.

4.3) UTILISATION DES DONNEES

Les bases étant posées, il faut à présent mettre les objets dans la scène. Pour cela la première chose à faire est de trouver un moyen efficace de lire les données stockées dans des fichiers pouvant aller de quelques milliers d'éléments (étoiles, particules, etc…), à plusieurs dizaine de millions. Ce grand volume de données à traiter a été un grand challenge.

Pour effectuer la lecture des fichiers, j'ai d'abord opté pour une méthode souvent utilisée en JavaScript : « XMLHttpRequest » qui est utilisée pour récupérer facilement des données via des requêtes HTTP. Mais ce protocole est plutôt utilisé pour des communications client-serveur, or les fichiers à ma disposition étaient disponibles uniquement localement. Cette méthode n'était donc pas adaptée et générait des problèmes tels que l'erreur « Cross-origin resource sharing » dans le navigateur Chrome. Cette erreur provient du fait que le navigateur bloque les requêtes HTTP effectuées en local par mesure de sécurité. Cette méthode pourrait par contre être utilisée par la suite quand l'application sera hébergée en ligne et que les fichiers contenant les données seront également stockés sur le serveur. Mais dans cette optique, il faudrait tout de même laisser la possibilité à l'utilisateur d'envoyer ses propres données.

J'ai donc opté pour l'utilisation de l'API JavaScript File qui permet de manipuler les fichiers dans les applications web côté clients. En plus de celle-ci, l'utilisation de l'API FileReader permet aux applications web de lire de manière asynchrone le contenu des fichiers stockés sur l'ordinateur de

l'utilisateur. Cette API permet donc de lire des objets « File » ou « Blob » qui représentent des données brutes.

Pour indiquer quels fichiers lire, il faut mettre une balise input de type file dans le code HTML comme ci-dessous. On peut également ajouter la propriété « multiple » pour laisser l'utilisateur charger plusieurs fichiers en simultané.

```
<input type="file" id="files" name="file" multiple />
```

Figure 17 – Balise permettant de sélectionner des fichiers

Ceci permet à l'utilisateur de sélectionner les fichiers à lire grâce à l'interprétation de la balise par le navigateur, comme présenté dans la figure suivante.

Figure 18 – Représentation de la balise input par le navigateur

Il faut ensuite indiquer à l'application les opérations à effectuer lorsque l'utilisateur valide sa sélection. Pour cela JavaScript nous permet d'affecter une fonction lorsqu'un événement se produit. En l'occurrence l'événement qui sera occasionné est appelé « change ». Pour cela j'ai créé une fonction appelée « initFileReading » (cf. figure 19) qui vérifie tout d'abord que le navigateur possède les différentes API nécessaires au bon fonctionnement de la lecture des fichiers. Ceci est indispensable car les utilisateurs utilisent parfois des versions antérieures de leur navigateur, il est donc nécessaire de le mettre à jour pour bénéficier de toutes les fonctionnalités. Après cette vérification je déclare la fonction « handleFileSelect » qui ne fait que parcourir tous les fichiers sélectionnés en appellant une autre fonction « readAdd » qui à son tour s'occupera de lire le fichier entré en paramètre. Et finalement, à la ligne 247 de la figure 19, je lie la fonction « handleFileSelect » à l'événement « change » de la balise qui permet de sélectionner les fichiers.

```
220 /**
221  * @author Arnaud Steinmetz <s.arnaud67@hotmail.fr>
222  *
223  * @description Function that will define what to do when opening a file.
224  *
225  */
226 var initFileReading = function() {
227     // Checking for the various File API support.
228     if (window.File && window.FileReader && window.FileList && window.Blob) {
229
230         /**
231          * @author Arnaud Steinmetz <s.arnaud67@hotmail.fr>
232          *
233          * Function that will handle the loading of the given file
234          * @param {Event} evt
235          */
236         function handleFileSelect(evt) {
237             var files = evt.target.files;
238             var nbFiles = files.length;
239
240             //Loop used to launch the reading of each file
241             for(var numFile = 0; numFile < nbFiles; numFile++)
242             {
243                 readAdd(files, numFile);
244             }
245         }
246         //Setting the event change on the file input to launch the function handleFileSelect
247         document.getElementById('files').addEventListener('change', handleFileSelect, false);
248
249     } else {
250       alert('The File APIs are not fully supported in this browser.');
251     }
252 }
```

Figure 19 – Fonction initFileReading

C'est donc la fonction « readAdd » qui réalise la lecture en utilisant l'API FileReader. Notons que l'API propose différentes méthodes pour effectuer cette opération :

- readAsArrayBuffer qui renvoie le résultat sous forme d'un ArrayBuffer
- readAsBinaryString qui renvoie le résultat sous forme de chaîne de caractères
- readAsText qui renvoie également le résultat sous forme de chaîne de caractères
- readAsDataURL qui renvoie le résultat sous forme d'une URL encodée en base64

Chacune de ces méthodes est spécifiques à certaines utilisations. « readAsDataURL » est souvent utilisé pour lire des images et permet d'afficher facilement des miniatures.

« readAsBinaryString » et « readAsText » sont utilisées si l'on veut récupérer directement le contenu des fichiers sous formes de string (chaînes de caractères) pour pouvoir directement l'exploiter.

« readAsArrayBuffer » quant à lui est utilisé pour lire des fichiers encodées en binaire. Pour accéder aux données, il faut soit utiliser un objet « DataView » soit déclarer un tableau du type correspondant à l'encodage initial.

Une problématique rencontrée durant le stage fut la très grande variété de formats des fichiers de données. Ce fut assez fastidieux de travailler avec des données provenant de trois sources différentes, et qui ne respectaient ni la même syntaxe ni le même encodage. Car il réaliser et optimiser un prototype pour chaque type de données.

Les premières données mises à ma disposition m'ont été fournies par André SCHAAFF. Celles-ci étaient issues d'une simulation appelée CODA (COsmic DAwn). Cette simulation a utilisé un code nommé RAMSES qui permet d'étudier la formation des grandes structures et des galaxies et qui a été exécuté sur TITAN, un supercalculateur américain (n°2 mondial). Les conditions initiales de la simulation ont été données de telle sorte à reproduire ce que les scientifiques pensent être des voies lactées. La taille totale de cette simulation était de 4096^3 cellules équivalentes à 64 mégaParsec (1 mégaParsec = 3.08 * 10^{22} mètres) ce qui est proche d'être la taille maximum.

Les fichiers avec lesquels j'ai travaillé étaient des extraits d'une taille de 256^3 cellules équivalentes à 4 mégaParsec. Les jeux de données reçus étaient séparés en 8, du fait du fonctionnement de l'algorithme qui a utilisé plusieurs processeurs pour calculer les résultats. Ces données étaient encodées sous la forme de string, et concernaient des étoiles. Comme on peut le voir sur la figure 20, la première ligne présente dans le fichier permet de dater l'échantillon de données. La deuxième indique le nom de chaque colonne.

```
1  Age of the universe in Myr    247.229670543717
2       Id   Mass(Msol)      x(Lbox)      y(Lbox)      z(Lbox)     Age(Myr)
3    10040  3.19393E+03  4.56792E-02  5.77574E-02  1.16433E-02  2.01123E+01  0.00000E+00
4    19692  3.19393E+03  3.79545E-02  4.76482E-02  9.89502E-03  7.18425E+00  0.00000E+00
```

Figure 20 – Extrait des données de CODA

Pour lire ces fichiers, j'ai utilisé la méthode binaryString. Lorsque la lecture est terminée, l'événement « onload » est lancé par le FileReader. A cet événement il suffit d'associer une fonction (cf. figure 21) qui parcourt toutes les lignes en ignorant les deux premières. Pour manipuler le résultat donné sous la forme d'une seule chaîne de caractères. Il a fallu utilisé la méthode « split », qui permet de scinder la chaîne, en indiquant un séparateur, et qui finalement retourne les résultats dans un tableau contenant les sous-chaînes

```
var data = file.result.split('\n');
var lines = data.length-1;
for (var i = 2; i<lines;i++) {
    var info = data[i].trim().split('  ');
}
```

Figure 21 – Extrait de la fonction de lecture pour les données de CODA

Pour charger des jeux de données d'environ 780 000 étoiles, il y avait une latence d'environ 1200 ms, et le nombre de FPS était aux alentours de 51 (cf. figure annexe 5), ce qui est très proche de la valeur optimale qui est 60. De plus considérant que d'après le rapport de l'année dernière de Philippe GAULTIER, son application ne parvenait à afficher qu'au maximum 32000 éléments pour un temps de chargement de 120 secondes. La faible performance de son application s'explique par les différents choix qu'il a pu faire pour représenter ses données et qui seront détaillés dans la partie suivante.

Le deuxième type de données rencontré, m'a été fourni par Nicolas DEPARIS, doctorant et membre de l'équipe Galaxies. Celles-ci provenaient d'un code similaire à celui de RAMSES appelé EMMA, qui a été exécutée par un supercalculateur français nommée CURIE. Dans ce cas les fichiers étaient séparés en 128 sous-parties, pour les mêmes raisons que pour CODA. Ce jeu de données contenait 3 types fichiers. Il y avait ceux comportant des particules de matière noire, des étoiles, ou encore des champs scalaires représentant du gaz. Et en tout, il y avait 10 extraits pris à des instants différents. Je n'ai que manipulé les fichiers contenant les données de matière noire et d'étoiles.

Pour lire ces fichiers il a fallu utiliser la méthode « readAsArrayBuffer ». Puis pour récupérer les données, il faut savoir que les éléments de base étaient de type Float32. Il faut donc créer un tableau en indiquant en paramètre le résultat obtenu à la fin de la lecture.

Chaque fichier débute par un nombre indiquant la quantité d'éléments présents, et un second dont j'ignore la signification. Après cela, dans le cas des particules de matière noire, les données se suivent selon le schéma suivant :

Case	0	1	2	3	4	5	6	7	8	9
Information correspondante	Position			vecteur vitesse			identifiant	Masse	Epot	ekin
	x	y	z	x	y	z				

Dans le cas des étoiles, il n'y a que l'âge qui se rajoute en plus du reste des informations.

On peut donc récupérer les données en utilisant une boucle comme illustré dans la figure 22.

```
var array = new Float32Array(file.result);
var nbElements = (array.length-2)/10;

for(var i = 0; i<nbElements;i++)
{
    var identifiant = array[8+i*10];
}
```

Figure 22 – Extrait de la fonction de lecture pour les données de particule d'EMMA

Dans cette boucle, chaque particule possède 10 informations associées, il faut donc multiplier l'itérateur par 10 pour qu'à chaque tour de boucle on accède aux données d'une nouvelle particule. A cela il suffit d'ajouter le chiffre correspondant à l'information à laquelle on veut accéder, sans oublier d'ajouter 2 pour ignorer les informations d'introduction inutilisées.

Le chargement de ces données (cf. figure annexe 6) était bien plus rapide que le chargement des données encodées en chaîne de caractères, ce qui était prévisible. Cette fois ci pour charger environ

2 millions d'éléments il faut environ 600 ms. De plus le nombre de FPS était d'environ 53, ce qui est assez performant pour un nombre d'éléments aussi important.

On remarque ainsi une grande différence de performance entre la lecture de chaîne de caractères et celle de données binaires bien plus performantes. Ceci s'explique très facilement par le fait que le processeur peut directement envoyer les données binaires à la carte graphique sans effectuer d'opérations intermédiaires car la GPU comprend directement de quoi il s'agit, tandis que dans le cas des chaînes de caractères la conversion prend un temps précieux.

Finalement, le troisième jeu de données concernaient skybot3D, un logiciel ayant pour ambition de fournir aux astronomes un outil utilitaire permettant de visualiser les objets présents dans le système solaire (étoiles, planètes, astéroïdes…). J'ai passé moins de temps sur ces données, car je les ai obtenues vers la 8ème semaine. De plus ce n'était pas le type de données initialement destinées à utiliser mon application de modélisation de par le fait qu'il ne s'agissait pas d'une simulation, mais il était tout de même intéressant de voir ce que cela pouvait donner.

Concernant le format, il s'agissait de JSON qui est un format de données textuelles dérivé de la notation des objets du langage JavaScript. Ayant d'autres fonctionnalités à mettre en place pour les autres prototypes, Je n'ai pas construit de fonction de lecture adaptée. J'ai par contre demandé à avoir le même jeu de données sous format de chaînes de caractères avec des séparateurs, pour pouvoir visualiser ce à quoi les données pouvaient ressembler lorsque je les visualisais en utilisant mon application (cf. figure annexe 7).

L'encodage et le format des données est donc un paramètre très important à prendre en compte. Surtout si comme dans le cas de l'Observatoire, les données peuvent provenir de nombreuses sources différentes. Il faudrait donc essayer de mettre en place un format de données standardisé, ou d'utiliser le format déjà en place pour les outils Aladin, VizieR et Simbad, ce qui permettrait même une certaine interopérabilité.

Un des derniers point à soulever est que dans le cas des données que m'a fournies André SCHAAFF, la taille des fichiers pouvaient dépassée les 180 Mo, ce qui posait des soucis à ma méthode de lecture. Car l'API fileReader place le résultat de la lecture en mémoire, en attendant qu'un traitement se fasse. Or au-delà de 180 Mo, l'application ne supportait pas cette charge et cessait donc de fonctionner.

Pour pallier à cela, il est possible de découper le fichier concerné en sous-partie appelée Blob grâce à l'API File. Cela se fait en spécifiant trois paramètres, l'octet de départ, l'octet d'arrêt, et le fichier source. En utilisant cette méthode, il est donc possible de découper les fichiers trop volumineux. A noter que la lecture d'un Blob se fait de la même manière que pour un fichier.

4.4) STRUCTURE ET REPRESENTATION DANS L'ESPACE

TYPE D'OBJET

Une fois les données chargées, il fallait trouver un moyen efficace de les représenter. Three.js proposait deux représentations pouvant s'avérer intéressantes :

- L'utilisation de Mesh qui est la représentation d'un objet tridimensionnel constitué de triangles.
- L'utilisation d'un objet appelé PointCloud (traduit en nuage de point).

Dans le premier cas, les objets sont constitués de sommets et d'arêtes, et les faces se composent de triangles. Pour un cube par exemple, il faut deux triangles pour représenter une face et donc 3 sommets par triangles. Ce qui fait un total de 2*6*3 = 36 sommets à manipuler pour la carte graphique (en réalité certain sommets sont communs entre les triangles, mais la GPU les manipule autant de fois qu'ils appartiennent à un triangle).

Dans le cas de l'objet PointCloud, la GPU reçoit les données concernant chaque objet, et les représente chacun avec un objet natif à Open_GL appelé « GL_POINTS ». Cette méthode ne nécessité qu'un sommet pour représenter un élément.

On remarque donc une grande différence dans la manière dont ces deux types d'objets sont manipulés avec la carte graphique. C'est en partie ce qui explique la faible performance de l'application de Philippe GAULTIER, car celui-ci utilisait des « mesh » pour représenter les données.

J'ai bien entendu fait des tests comparatif en utilisant ces deux méthodes avec le même jeu de donné contenant environ 30 000 éléments.

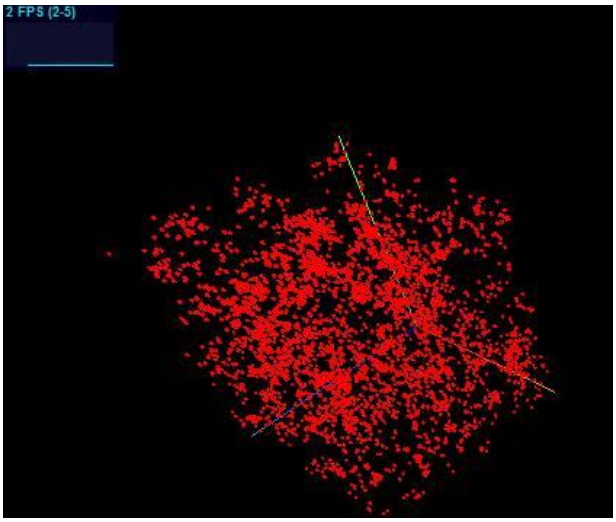


Figure 23 – Modélisation avec meshes

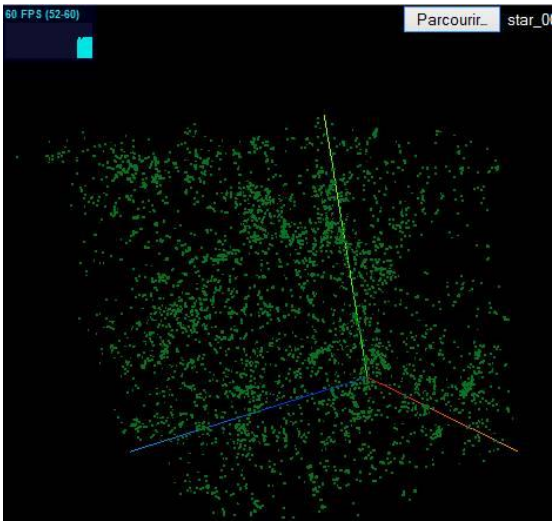


Figure 24 – Modélisation avec un PointCloud

Temps de chargement : 415 ms	Temps de chargement : 45 ms
Fluidité : 5 FPS	Fluidité : 60FPS

Le résultat de ce test n'a fait que conforter l'hypothèse avancée précédemment. Mon choix s'est donc naturellement porté vers l'utilisation de l'objet « PointCloud ».

STRUCTURES DANS LE CODE

Pour entrer plus en détail dans le fonctionnement de l'objet PointCloud et tenté d'en améliorer les performances, il faut se pencher sur les deux paramètres que prend le constructeurs en entrée.

Celui-ci nécessite deux choses :

- Un objet ''geometry'' qui contiendra toutes les données nécessaires pour décrire notre objet dans l'espace (dans notre cas, la position des différents points uniquement)
- Un objet ''material'' qui spécifiera l'apparence de l'objet (taille, couleur, texture, …). Lors de l'utilisation d'un PointCloud, le plus souvent on utilise l'objet ''PointCloudMaterial''.

Pour indiquer à mon PointCloud à quelle position il faut afficher les points. Il suffisait de parcourir les lignes du fichier et de créer pour chaque élément un Vector3 qui contiendrait les coordonnées et qui serait ajouté dans le tableau des sommets de l'objet geometry.

De base, un GL_POINTS est représenté de la taille d'un pixel quelle que soit sa distance par rapport à la caméra. Or Three.js offre possibilité de changer la taille de chaque point en ayant une notion de distance grâce à la propriété sizeAttenuation de l'objet PointCloudMaterial. C'est également ce qui explique que dans ma modélisation les étoiles soient représentées par un carré, car il s'agis d'un pixel dont la taille a été agrandie.

Après avoir réalisé ces premiers prototypes, j'ai cherché à simplifier la fonction de lecture. Mais j'ai également cherché s'il existait un moyen d'envoyer les données à la GPU de manière à ce qu'elle comprenne directement les opérations à effectuer. Car j'utilisais l'objet « geometry » qui envoie la position des sommets sous la forme d'un tableau de « Vector3 », chose que la GPU ne pouvait pas comprendre sans effectuer des opérations de conversion.

Après quelques recherches, j'ai trouvé que Three.js proposait l'utilisation de « BufferGeometry ». Cet objet était dit plus difficile à appréhender pour le programmeur, mais bien plus efficace en terme de performance. Par l'utilisation de cet objet, le traitement à effectuer pour passer les informations à la GPU est amoindri. Pour le chargement d'un jeu de données contenant environ 800 000 éléments. Le temps de chargement est passé de 2119 ms à 1300 ms. Et le nombre de FPS est passé de 30 à 45. Ce qui est un bon de performance plutôt important.

Comme l'on peut voir sur la capture d'écran ci-dessous, il faut créer un tableau de float32, qui contiendra les positions successives de tous les éléments de la scène et le remplir avec toutes les positions en parcourant les différentes lignes du fichier.

Ensuite il suffit d'ajouter ce tableau en tant qu'attribut de notre geometry en spécifiant le nombre de case concernant un élément. Dans notre cas il faut les positions en x, y et z, donc on spécifie que les informations se succèdent par groupe de 3.

```
var file = evt.target;
//Checking if the file has correctly been read
if (file.readyState == FileReader.DONE) {
    var geometry = new THREE.BufferGeometry();
    var vertices = new Float32Array(lines*3);

    var info;
    for (var i = 0; i<lines;i++){
        //Separating the different informations in the
        info = data[i+2].trim().split('  ');

        //Setting the position in the buffer
        vertices[i*3]=parseFloat(info[2]);      //x
        vertices[i*3+1]=parseFloat(info[3]);    //y
        vertices[i*3+2]=parseFloat(info[4]);    //z
    }
    geometry.addAttribute('position' , new THREE.BufferAttribute( vertices, 3 ));

    pointCloud = new THREE.PointCloud(geometry, material);
    pointCloud.name = fileName;
    scene.add(pointCloud);

    pointClouds.push(pointCloud);
}
file = null;
```

Figure 25 – Opération post-lecture version avec BufferGeometry

André SCHAAFF m'a également demandé s'il était possible de mettre une couleur pour chaque étoile en fonction de son âge. Pour cela, en utilisant la même manière qu'avec la position, il est possible de remplir un tableau qui contiendra les couleurs correspondantes à chaque point. Il ne faut pas oublier également de préciser à l'objet PointCloudMaterial que l'on envoie chaque couleur individuellement, et pour cela il faut mettre la valeur de l'attribut VertexColors à THREE.VertexColors.

La fonction de calcul que j'ai appliquée fait varier les couleurs du bleu pour les étoiles très jeunes vers le rouge pour les étoiles les plus âgées (cf. figure annexe 8)

Pour pouvoir toujours modifier davantage l'apparence de nos points il est possible de programmer un shader nous même en GLSL. L'objet PointCloudMaterial sera alors remplacé par un shaderMaterial, où l'on pourra reprogrammer les différentes opérations à effectuer sur chaque point.

Pour mieux comprendre ce qu'est un shader voici un schéma basique des opérations effectuées par l'ordinateur pour obtenir le rendu à l'écran.

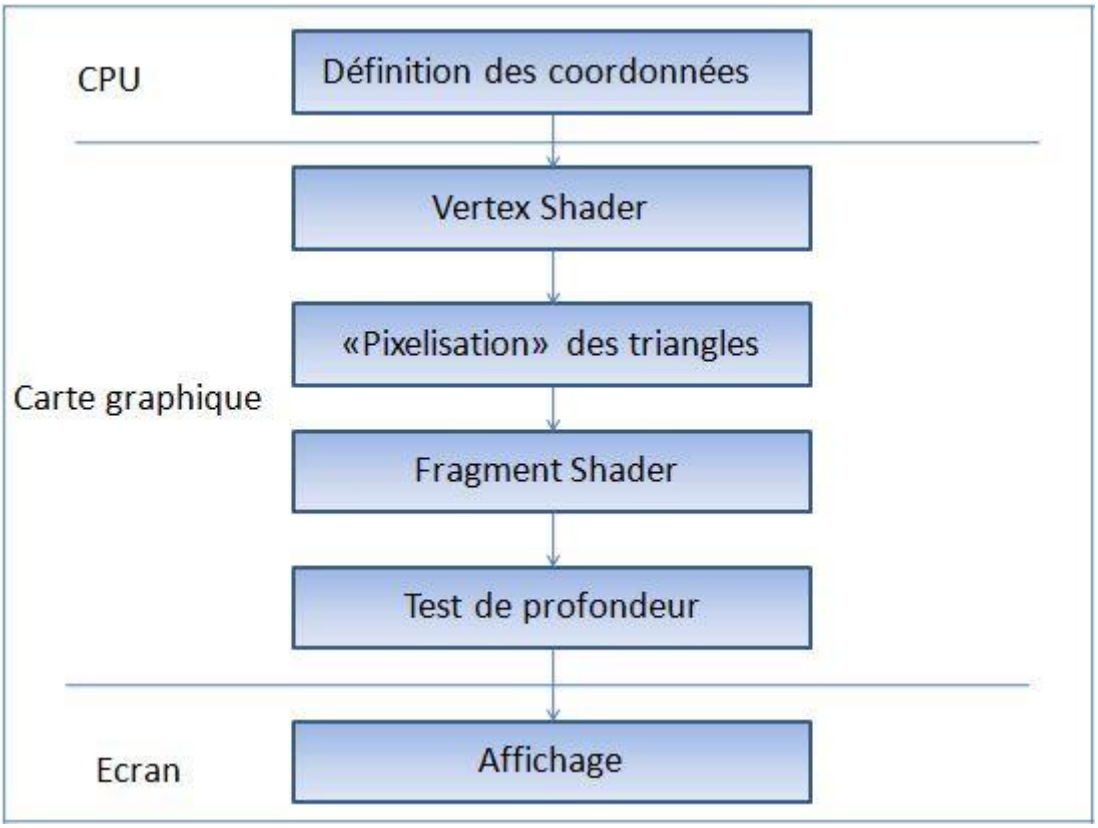


Figure 26 – Schéma simplifié présentant les opérations nécessaires à obtenir un rendu

Comme on peut le voir ci-dessus, les coordonnées sont envoyées par le CPU au GPU, qui va effectuer un premier traitement appelé Vertex Shader. Celui-ci est reprogrammable, et va effectuer une action pour chaque vertex (sommet). C'est celui-ci dont on tirera le plus d'utilité car nous cherchons justement à effectuer un traitement pour chacun de nos points.

Après ce traitement, il y a la ''pixelisation'' des triangles. C'est ce qui remplit les différentes faces des objets de type « mesh ». Or ici nous n'en n'utilisons pas, donc cela ne nous concerne pas.

Arrivé maintenant au deuxième type de shader programmable, le Fragment Shader, qui permet d'effectuer un traitement sur chaque pixel de la scène. Ce shader est plutôt utilisé dans le cas d'utilisation d'objets de type mesh.

C'est donc grâce à l'utilisation d'un shaderMaterial que j'ai pu charger une texture pour la représentation des étoiles dans les simulations provenant d'EMMA (cf. figure annexe 12 et 13).

Grâce aux shaders, il est donc possible d'agir sur un grand nombre de propriétés d'affichage. Il serait donc éventuellement de modifier la valeur d'autres attributs comme l'opacité ou la taille des points.

OPTIMISATION STRUCTURE DU CODE

Durant mes recherches d'optimisations j'ai également découvert la possibilité d'effectuer la mise en place d'une structure en Octree. Cette structure de données permet le découpage d'un espace tridimensionnel en le subdivisant récursivement en huit sous-cubes appelés octants.

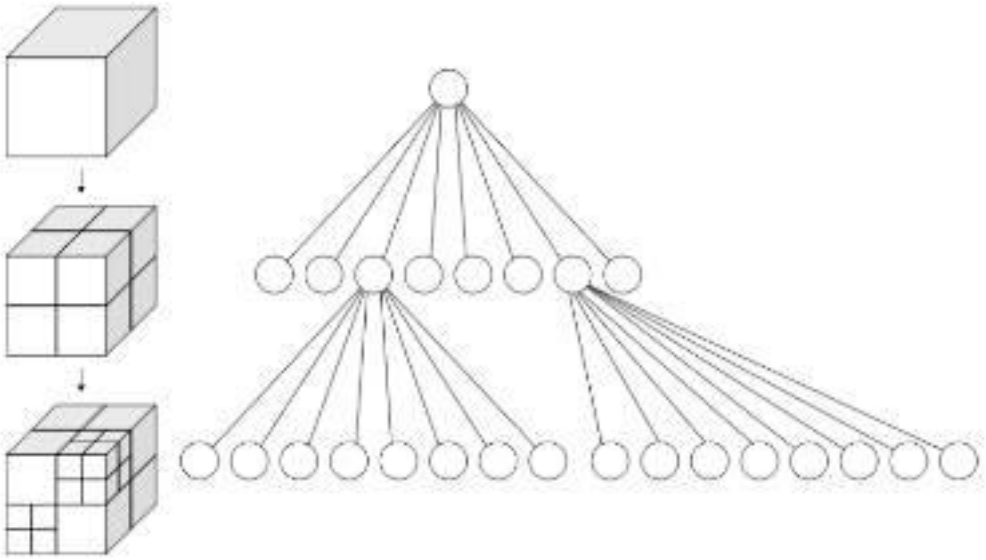


Figure 27 – Représentation schématique de la structure en octrees

Cela a pour conséquence de ralentir le temps de chargement initial, car il faut que l'application découpe l'espace en respectant cette structure. C'est notamment ce qui a permis à Philippe GAULTIER de pouvoir afficher 32 000 meshes avec 60 FPS, mais en contrepartie son temps de chargement initial était de 120 secondes.

Cette structure permettrait, dans le cas d'un chargement de jeux de données de plusieurs dizaine de millions de points, de n'effectuer qu'un chargement partiel des données proches de la caméra. Cela éviterait donc une consommation inutile de ressource pour les éléments situés trop loin de la caméra.

Lors de la création de mes différents prototypes, grâce à l'utilisation d'un PointCloud et aux autres optimisations effectuées, j'ai obtenu des résultats suffisamment concluant pour ne pas avoir à mettre en place cette structure. J'ai malgré tout effectué un grand nombre de recherche pour en comprendre le fonctionnement. Il existe de nombreuses implémentations différentes, qui apportent chacune leurs points positifs et négatifs. Cela sera une voie à explorer dans le cas d'une reprise des travaux.

4.5) FONCTIONNALITES

Pour qu'il y ait un réel intérêt à modéliser ces données, j'ai ajouté quelques fonctionnalités.

CONTROLES

Tout d'abord j'ai implémenté différentes manières de contrôler la caméra :

- OrbitControls
- FpControls
- EarthControls

OrbitControls permet grâce au clic gauche comme son nom l'indique de faire un rotation en orbit autour d'un point qui est placé de base en (0,0,0). Avec le clic droit, il est possible de déplacer le point de rotation, et la molette permet de zoomer.

Le second mode de déplacement correspond à un contrôle de caméra à la « first person », ce qui se traduit par première personne. Avec ce type de contrôle, la caméra effectue une rotation sur elle-même lorsque l'utilisateur utilise le clic gauche. Et l'on peut déplacer la caméra grâce aux touches directionnelles.

EarthControls quant à lui est une sorte d'amélioration d'OrbitControls, cette fois-ci les déplacements se font par rapport à un élément de la scène (étoiles, particules, ou autre…) Avec le clic droit on peut donc effectuer une rotation autour du point sélectionné. Le clic gauche permet de sélectionner un élément et de s'en approcher ou s'en éloigner.

Pour implémenter cette dernière méthode de déplacement, il a fallu trouver un moyen pour savoir sur quel élément l'utilisateur clic. Pour faire cela, la documentation officielle m'a été d'une très grande aide.

RAYCASTING

Pour pouvoir savoir où l'utilisateur clic, il faut faire un « raycasting ». Ceci est une technique de lancer de rayon qui trace une demi-droite ayant pour origine la position de la caméra et qui passe par le point de coordonnées (x,y) de l'écran sélectionné. Pour comprendre ce mécanisme voici ci-dessous une illustration.

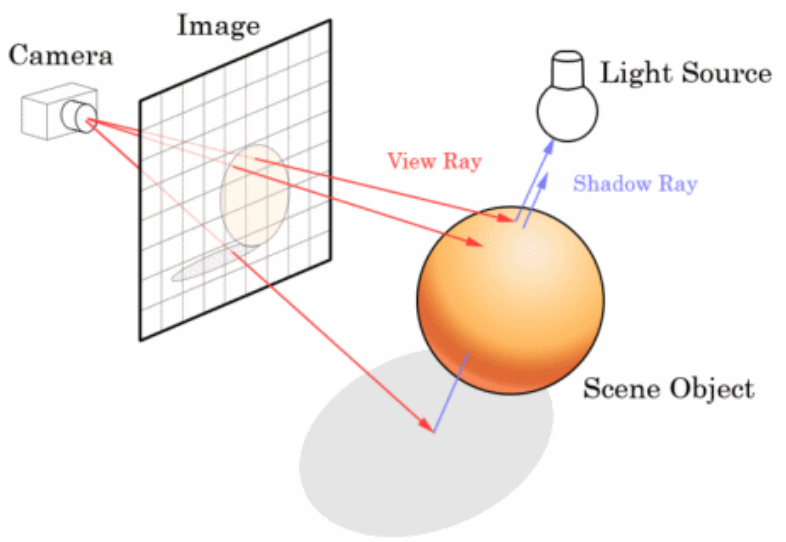


Figure 28 - Technique de raycasting

Grâce à la position de la caméra à la position où l'utilisateur a cliqué, il est donc possible de tracer une demi-droite fictive dans notre espace 3D. Il suffit alors de tester s'il y a une interaction avec un objet de la scène.

Pour réaliser cela dans notre application, il suffit de déclarer un objet « Raycaster » de Three.js qui facilite grandement la tâche. A cet objet, il suffit d'indiquer en paramètre la caméra et la position sur l'écran où l'utilisateur a cliqué. Puis on utilise une méthode propre au « Raycaster », appelé « intersectObjects » à laquelle on envoie également en paramètre un tableau contenant les objets qui vont être testés. Mais derrière cette facilité apparente se cache un souci de taille. Dans notre modélisation, nous n'avons en réalité qu'un seul objet qui se trouve être le PointCloud, et les différents éléments visibles de notre scène ne sont des attributs de cet objet. Pour résoudre ce problème, Three.js a directement implémenté deux fonctions « raycast » différentes que le « Raycaster » utilise. Dans le cas où l'objet à tester est un « PointCloud », alors le teste se fera en réalité sur les attributs de celui-ci. Or une chose que j'avais omise, c'est qu'il faut indiquer un seuil appelé « threshold » qui correspond à la distance à partir de laquelle les objets sont considérés ou non en intersection avec la demi-droite. La valeur par défaut étant de 1, et mes données étant placées dans un volume de taille 1, je recevais en retour de la fonction la globalité de mes points. J'ai donc expliqué mon problème sur Stackoverflow, un forum d'entraide où l'on m'a expliqué qu'il fallait donner une valeur au « threshold » du « Raycaster ». Pour trouver la valeur, il suffisait de diviser la taille correspondante au point par 2. Cette précaution prise, la fonction dont on peut voir une capture d'écran ci-dessous fonctionnait correctement.

```
function getMousePointCloudIntersection(mouse)
{
    if(pointClouds != null)
    {
        var raycaster = new THREE.Raycaster();
        raycaster.params.PointCloud.threshold = pointSize/2;
        raycaster.setFromCamera( mouse, camera );

        var intersections = raycaster.intersectObjects( pointClouds );
        intersection = ( intersections.length ) > 0 ? intersections[ 0 ] : null;
    }
    return intersection;
}
```

Figure 29 – Fonction permettant le raycasting

Le résultat de l'opération « raycaster.intersectObjects(pointClouds) », renvoie un tableau de tous les éléments intersectés classés par rapport à leur distance à la caméra. La ligne qui suit est une forme particulière de condition qui permet dans ce cas-ci de soit renvoyer le premier élément rencontré par la demi-droite, soit de renvoyer la valeur « null » dans le cas où il n'y avait aucun objet dans le tableau d'intersections.

La valeur retournée par la fonction dans le cas d'un succès est un objet ayant les propriétés suivantes

- distance : il s'agit de la distance de la caméra par rapport à l'intersection
- distanceToRay : il s'agit de la distance entre le centre de l'élément intersecté et l'intersection
- face : dans le cas d'un PointCloud cette valeur est « null »
- index : indice correspondant au numéro de l'attribut intersecté
- object : objet de l'élément intersecté, en l'occurrence notre PointCloud
- point : coordonnées de l'intersection dans les repères du monde

C'est donc cette méthode qui permet de localiser précisément l'élément sur lequel l'utilisateur a cliqué. Et grâce aux attributs object et index, il est possible d'accéder à l'élément en question en accédant aux buffers correspondant à l'opération à effectuer.

J'ai par exemple pensé à afficher des informations supplémentaires lors d'un clic sur une étoiles (cf. figure annexe 9), comme par exemple leur identifiant, masse, âge, ou toutes autres informations contenues dans les fichiers sources. Pour bien mettre en évidence le point sélectionné, lors du clic de l'utilisateur sur un élément, je le colore en blanc, tout en gardant la précédente couleur en mémoire pour remettre l'élément à sa couleur d'origine lors d'un clic sur une autre zone.

Le « Raycasting » m'a aussi permis de résoudre un problème rencontré durant la modélisation. Car certaines étoiles sont assez proches et forment un regroupement appelé galaxie, or les galaxies sont très espacés entre elles. Donc pour avoir une vue globale, il faut agrandir la taille des carrés qui représentent nos étoiles. Mais ceci provoque un effet de chevauchement, et l'on a du mal à distinguer les étoiles unes à unes (cf. figure annexe 10). J'ai donc mis en place un zoom lorsque l'utilisateur double clic sur une étoile qui fait en sorte de réduire la taille des points tout en s'approchant du point lui-même (cf. figure annexe 11), ce qui permet d'apercevoir des structures complexes entre les étoiles. Il est bien entendu encore possible d'affiner la vue grâce à l'interface qui permet d'agir sur la taille des étoiles.

MULTIVUE

Une autre fonctionnalité intéressante aurait été la création d'un mode multi-vue, qui permettrait de charger deux jeux de données représentant la même simulation à des instants différents. Cela pourrait permettre de voir aisément les différences entre deux époques de la simulation.

J'ai donc tenté de réaliser cela en créant deux renderer et deux scènes, qui accueilleront les deux vues. Et par question de simplification du code et d'optimisation des performances, j'ai pensé que l'utilisation de la même caméra dans pour effectuer les deux rendus serait le mieux. Car l'on souhaitait que les déplacements soient effectués en simultané sur les deux vues.

Mais lorsque j'ai implémenté cette solution, la console m'indiquait plusieurs warning me prévenant que la caméra était utilisée deux fois, ainsi que pleins d'autres erreurs annexes.

Le déplacement s'effectuait bien pour les deux scènes, mais le problème est survenu lors du chargement des données. Les deux jeux de données se chargeaient correctement, mais le rendu du second était mauvais. Car les points n'avaient pas la taille que je leur avais attribué, il était donc quasiment impossible de les distinguer. Leur taille était d'un pixel qu'importe la distance, comme si la valeur de « sizeAttenuation » était false.

J'ai donc abandonné l'idée de réparer cette erreur après de nombreuses heures de recherches infructueuses, pour me consacrer à d'autres fonctionnalités.

NOTION DE TEMPS

Avec les données concernant les particules de matière noire provenant d'EMMA, j'ai essayé de réaliser un déplacement des particules en temps réel en fonction du temps.

Pour cela Nicolas DEPARIS, m'avait fourni les positions des particules au départ, et à la fin de la simulation. Pour calculer le déplacement, donc charger les différents fichiers en stockant la position de départ et le vecteur vitesse calculé simplement en d'arrivée à la position de départ.

Pour effectuer le calcul de la position à un instant t, j'ai effectué le calcul suivant :

```
positionDepart + vecteurVitesse * t
```

En sachant que la variable t varie entre 0 et 1 et est incrémenté dans la boucle de rendu.

Sans me faire d'illusion j'ai d'abord tenté de réaliser ce calcul directement en JavaScript, et donc sur le CPU. Mais les performances était lamentables, les FPS chutaient jusqu'à 5.

J'ai donc pensé à utiliser un shader personnalisé (cf figures 30 et 31) qui permettrait de mettre à contribution la puissance de calcul du GPU.

```
var attributes = {
    direction: { type:'v3', value:null }
}

var uniforms = {
    t:          { type: 'f', value: 0.0},
    size:       { type: 'f', value: 0.1 },
    color:      { type: "c", value: new THREE.Color( 0xffffff ) }
};

var shaderMaterial = new THREE.ShaderMaterial( {
    attributes:     attributes,
    uniforms:       uniforms,
    vertexShader:   document.getElementById( 'vertexshader' ).textContent,
    fragmentShader: document.getElementById( 'fragmentshader' ).textContent,
});
```

Figure 30 – variables transmises à la GPU par le shaderMaterial

```
<script type="x-shader/x-vertex" id="vertexshader">
    uniform float size;
    uniform float t;
    attribute vec3 endPosition;

    void main() {

        vec4 mvPosition = modelViewMatrix * vec4( position+direction*t, 1.0 );

        gl_PointSize = size / length( mvPosition.xyz );

        gl_Position = projectionMatrix * mvPosition;

    }

</script>

<script type="x-shader/x-fragment" id="fragmentshader">
    uniform vec3 color;

    void main() {
        gl_FragColor = vec4(color,1.0);
    }

</script>
```

Figure 31 – contenu du fragment et vertex shader

Sur la figure 31, on peut voir les éléments envoyés au GPU, pour cette fonctionnalité les variables importantes sont la direction qui représente le vecteur vitesse. Cette variable doit être stockée en tant qu' « attributes » car c'est un tableau stockant la valeur des vecteurs vitesse pour chacune des 2 millions de particules.

La variable t est quant à elle envoyée en tant qu' « uniforms », car sa valeur est identique pour chaque particule. Dans la figure 31, on intervient dans le vertex shader pour changer la position de notre élément, grâce à la formule citée précédemment. L'utilisation du GPU permet de passer à 60 FPS ce qui est optimal. Ce résultat est confortant en sachant que l'on peut se déplacer en temps réel dans la scène tout en gardant ce taux de rafraichissement. De plus, grâce à l'interface, il est possible de changer la vitesse de l'animation ainsi que de remettre celle-ci à un moment précis. Des captures d'écran prises à des moments différents sont disponibles en annexe (cf. figure annexe 13 à 15).

Ces performances élevés présage la possibilité de mettre en place des animations plus complexes, comme le déplacement des astéroïdes pour une éventuelle version de skybot en WebGL. Mais dans ce cas, le calcul serait bien plus complexe et rien ne garantit de garder une telle fluidité. Il faut également prendre en compte que la puissance de l'ordinateur est un facteur déterminant, car sur mon ordinateur portable personnel, les FPS se situaient aux alentours de 52.

AUTRES POSSIBILITES

J'ai aussi pensé à deux autres fonctionnalités pouvant être intéressantes à implémenter qui seraient de :

- Nommer des galaxies avec une sélection du nom de la galaxie que l'on veut voir par l'intermédiaire de l'interface dat-GUI, qui déplacerait la caméra à l'endroit voulu.
- Relier l'application aux autres services du CDS, pour bénéficier de toujours plus de possibilités

III) BILAN

1) LES PROTOTYPES REPONDENT-ILS AUX ATTENTES

Pour résumé, j'ai effectué deux types de modélisation.

- La modélisation statique
- La modélisation animée

Pour la modélisation statique j'ai surtout travaillé avec les jeux de données des simulations EMMA et CODA.

Pour les simulations concernant EMMA, le temps de chargement pour la modélisation d'environ 2 000 000 de particules étaient de seulement 600 ms pour un rendu à 60 FPS.

Disposant d'importants jeux de données avec la simulation CODA, j'ai pu charger jusqu'à 26 millions de d'éléments pour un temps d'attente d'environ 30 secondes et une fluidité à 15 FPS. Ce qui est incroyablement plus efficace qu'avec l'application de Philippe GAULTIER qui se limitait à 32 000 éléments pour un temps de chargement de 120 secondes.

A titre de comparaison avec les données de la simulation d'EMMA, pour le même volume de donnée il faut environ 4 000 ms pour effectuer le chargement. Il est donc crucial de choisir un encodage binaire similaire aux fichiers provenant d'EMMA pour optimiser le fonctionnement de l'application.

Pour la modélisation animée, j'ai réalisé un prototype prenant en entrée les particules de matière noire d'EMMA. Et j'ai grandement été surpris par l'efficacité avec laquelle l'application gérait le calcul des positions car La fluidité restait à son maximum de 60 FPS.

Chose importante à préciser, est que les tests ont été effectués sur un machine dotée de 16Go de RAM, mais ne possédant pas de carte graphique (celle-ci n'utilisait que la puce graphique du CPU).

J'ai également effectué quelques essaies sur mon ordinateur portable, possédant une carte graphique GTX600M et 8 Go de RAM, les performances ne subissait qu'une perte d'environ 10% sur l'ensemble des prototypes effectuées.

2) PROPOSITION DE DEVELOPPEMENTS FUTUR

S'il y avait suite à ce projet, voici quelques améliorations simples pouvant être intéressantes à intégrer.

- Réalisation d'une caméra à 360° pour adapter l'affichage au dispositif du planétarium
- Définir un format de données binaire pour une utilisation plus générale
- Vérification du format de données. Si non respecté demander à l'utilisateur de modifier la fonction de lecture)

Une autre possibilité, plus complexe cette fois, serait d'utiliser des WebWorkers couplées aux octrees. Les WebWorkers sont pour le moment le seul moyen d'effectuer des traitements sur des threads séparés. Cela permettrait de créer un thread d'affichage, et d'autres threads effectuant des calculs. En utilisant cette technique, il est possible d'effectuer des calculs complexes sans pour autant bloquer la page web. Mais l'utilisation des WebWorkers est très peu intuitive et est très peu utilisée de nos jours, je n'ai donc pas trouvé toutes les informations nécessaires à la mise en place d'un prototype utilisant cette API.

Mais l'on pourrait imaginer une simulation qui lorsque la caméra se trouve trop loin d'un amas d'étoile, l'application chargerait une texture 2D de la galaxie correspondante. Puis lorsqu'on s'approche, il suffirait de charger les sous-cubes des octants pour affiner la précision et afficher les textures des étoiles ou astéroïdes correspondantes.

Une autre chose à prendre en compte est dans le navigateur Mozilla Nightly, de nouvelles API sont en cours de développement :

- Multi-processing : qui permettrait de paralléliser certaines opérations
- Mozilla Virtual Reality (MozVR) : qui est une implémentation réalisée avec Three.js permettant d'expérimenter la réalité virtuelle dans le navigateur.

3) CONCLUSION

Comme on peut le constater tout au long de ce rapport, j'ai été confronté à un important volume de données, provenant d'origines différentes. Cette variété a impliquée de nombreuses modifications entre les prototypes, ce qui s'est avéré d'une grande complexité. De plus les performances selon le type de données dépendaient grandement de leur encodage, il m'a donc fallu trouver des moyens optimisés à chaque modèle. Pour parvenir à mes fins, il a fallu me plonger en détail dans le fonctionnement du rendu 3D, qui n'avait été que superficiellement abordé en cours.

Ce sont notamment ces difficultés qui ont fait l'intérêt du stage. En effet j'ai pu approfondir de nombreuses notions et appréhender des technologies comme le WebGL que je ne maitrisais pas encore, et que j'ai trouvé passionnantes.

Le bon déroulement du stage a été possible grâce à l'encadrement efficace de mon maître de stage ainsi qu'à la disponibilité des collègues attentifs et à l'écoute qui me fournissaient des conseils avisés me permettant de bien structurer les bases et d'avancer vers l'objectif fixé. Mes recherches très enrichissantes sur le forum Stackoverflow m'ont également permis de solutionner nombre de problématiques rencontrées.

Cette expérience a été mon premier projet informatique de longue durée sur lequel j'ai pu travailler, et il m'a permis d'asseoir mes connaissances et d'améliorer mes compétences professionnelles. C'était également un premier contact bénéfique avec le monde du travail.

La bonne avancée de ce projet, permet d'ores et déjà aux deux étudiants ingénieurs de poursuivre le développement sur ces bases.

GLOSSAIRE

API

« Application Programming Interface », interface de programmation. Ensemble normalisé de classes et de fonctions qui sert de façade par laquelle un logiciel offre des services à d'autres logiciels

BIBLIOTHEQUE

En informatique, une bibliothèque logicielle est une collection de fonctions.

BUFFER

Terme anglais se traduisant par tampon. En informatique, un buffer est appelé mémoire tampon et désigne une zone de mémoire virtuelle utilisée pour stocker temporairement des données.

BUFFERGEOMETRY

Alternative plus efficace à la classe Geometry présente dans Three.js, puisqu'elle stocke toutes les données sous la forme de buffers.

COMMIT

Terme anglais désignant validation. Dans le cas de GitHub, un commit désigne la validation d'une modification apportée à un projet.

CPU

« Central Processing Unit », processeur. Composant de l'ordinateur qui exécute les instructions machine des programmes informatiques.

CSS

« Cascading Style Sheet », feuilles de styles en cascade. C'est un langage informatique qui décrit la présentation des documents HTML.

FPS

« Frame Per Second », mesure de la fluidité du rendu graphique en image par seconde

FRAME

Image rendue graphiquement par un programme, typiquement 60 fois par seconde.

FRAMEWORK

Ensemble cohérent de composants logiciels structurels.

GEOMETRY

Classe de three.js qui permet d'indiquer toutes les données nécessaires à décrire un modèle 3D.

GLSL

« OpenGL Shading Language ». Langage de programmation de shaders

GPU

« Graphics Processing Unit », processeur graphique. Circuit intégré présent sur une carte graphique et assurant les fonctions de calcul de l'affichage.

GUI

« Graphical User Interface », Interface graphique permettant un dialogue homme-machine, dans lequel les objets à manipuler sont dessinés sous la forme de pictogrammes à l'écran

HTML

« HyperText Markup Language ». Format de données conçu pour représenter les pages web.

HTTP

« Hypertext Transfer Protocol » est un protocole de communication client-serveur développé pour le World Wide Web

IDE

« Integrated Development Environment », Environnement de développement. Ensemble d'outils pour augmenter la productivité des programmeurs qui développent des logiciels.

JAVASCRIPT

Langage de programmation de scripts principalement employé dans les pages web interactives. C'est un langage orienté objet à prototype, c'est-à-dire que les bases du langage et ses principales interfaces sont fournies par des objets qui ne sont pas des instances de classes

JSON

« JavaScript Object Notation ». C'est un format de données textuelles dérivé de la notation des objets du langage JavaScript.

OCTREE

Structure de donnée de type arbre dans laquelle chaque nœud peut compter jusqu'à 8 enfants.

OCULUS RIFT

Masque de réalité virtuelle fournissant une expérience d'immersion inédite.

OCULUS VR

Entreprise de réalité virtuelle fabriquant l'Oculus Rift

PLUGIN

Module d'extension. Paquet qui complète un logiciel hôte pour lui apporter de nouvelles fonctionnalités.

SHADER

Programme informatique, utilisé en image de synthèse, pour paramétrer une partie du processus de rendu réalisé par une carte graphique ou un moteur de rendu logiciel. Ils peuvent permettre de décrire l'absorption et la diffusion de a lumière, la texture utiliser, les réflexions et réfractions, l'ombrage, le déplacement de primitives et des effets post-traitement.

VERTEX

Terme anglais signifiant sommet. (au pluriel : vertices)

WORLD WIDE WEB

Littéralement signifiant « toile mondiale », il permet de consulter avec un navigateur, des pages accessibles sur des sites. Ce n'est qu'une des multiples applications d'Internet.

WORLD WIDE WEB CONSORTIUM

Egalement connu sous le sigle W3C, le World Wide Web Consortium est un organisme de normalisation à but non lucratif chargé de promouvoir la compatibilité des technologies de World Wide Web.

BIBLIOGRAPHIE

Documentation officielle de Three.js :

http://threejs.org/docs/

Documentation officielle de Babylon.js :

http://doc.babylonjs.com/

Recensement d'exemples utilisant Three.js :

http://threejs.org/examples/

Stackoverflow :

http://stackoverflow.com/

Différentes implémentations des octrees :

https://geidav.wordpress.com/2014/07/18/advanced-octrees-1-preliminaries-insertion-strategies-and-max-tree-depth/

Tutoriel pour comprendre le fonctionnement de l'API FileReader :
http://www.html5rocks.com/en/tutorials/file/dndfiles/

Site de mozilla regroupant les différentes API :

https://developer.mozilla.org/fr/docs/Web/API

Site de Google regroupant des exemples d'utilisation de l'interface dat-GUI :

http://workshop.chromeexperiments.com/examples/gui/#1--Basic-Usage

Wiki interne du CDS détaillant mes démarches :

http://cds.u-strasbg.fr/twiki/bin/view/Stages/ArnaudSteinmetz

ANNEXES

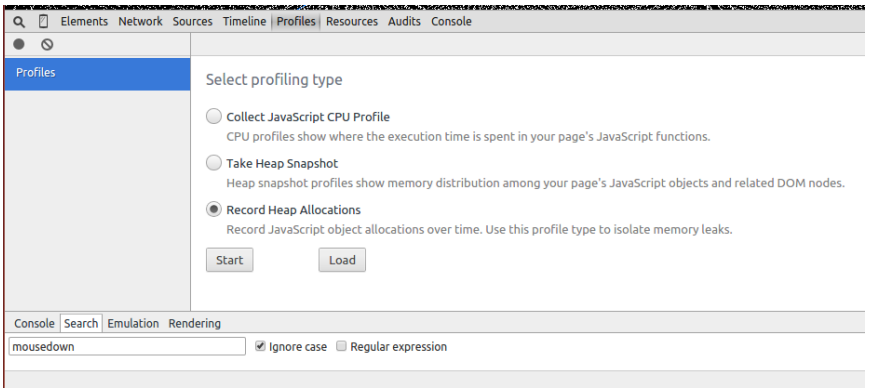


Figure annexe 1 – Outil de développement Chrome, Profiles

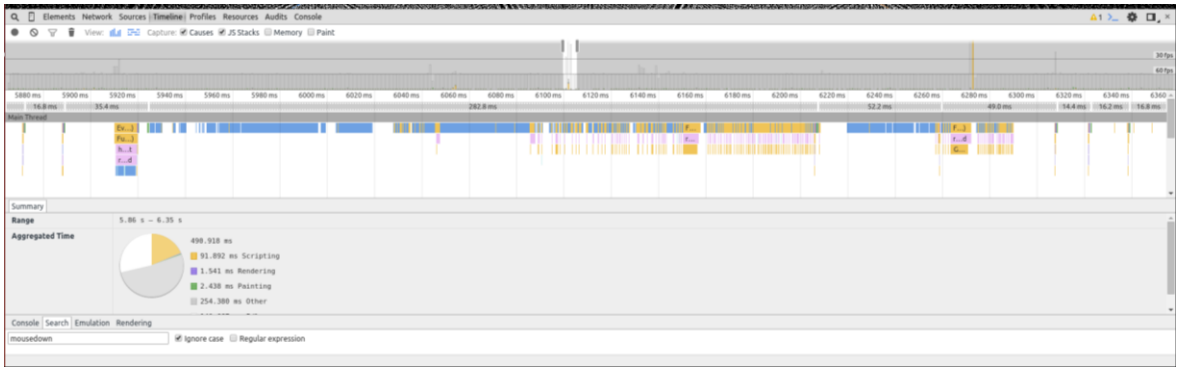


Figure annexe 2 – Outil de développement Chrome, Timeline

```
function render() {
    requestAnimationFrame(function (){
        render();
    });
    renderer.render( scene, camera );
    controls.update(clock.getDelta());
}
```

Figure annexe 3 – Boucle de rendu

Figure annexe 4 – Exemple d'interface dat-gui

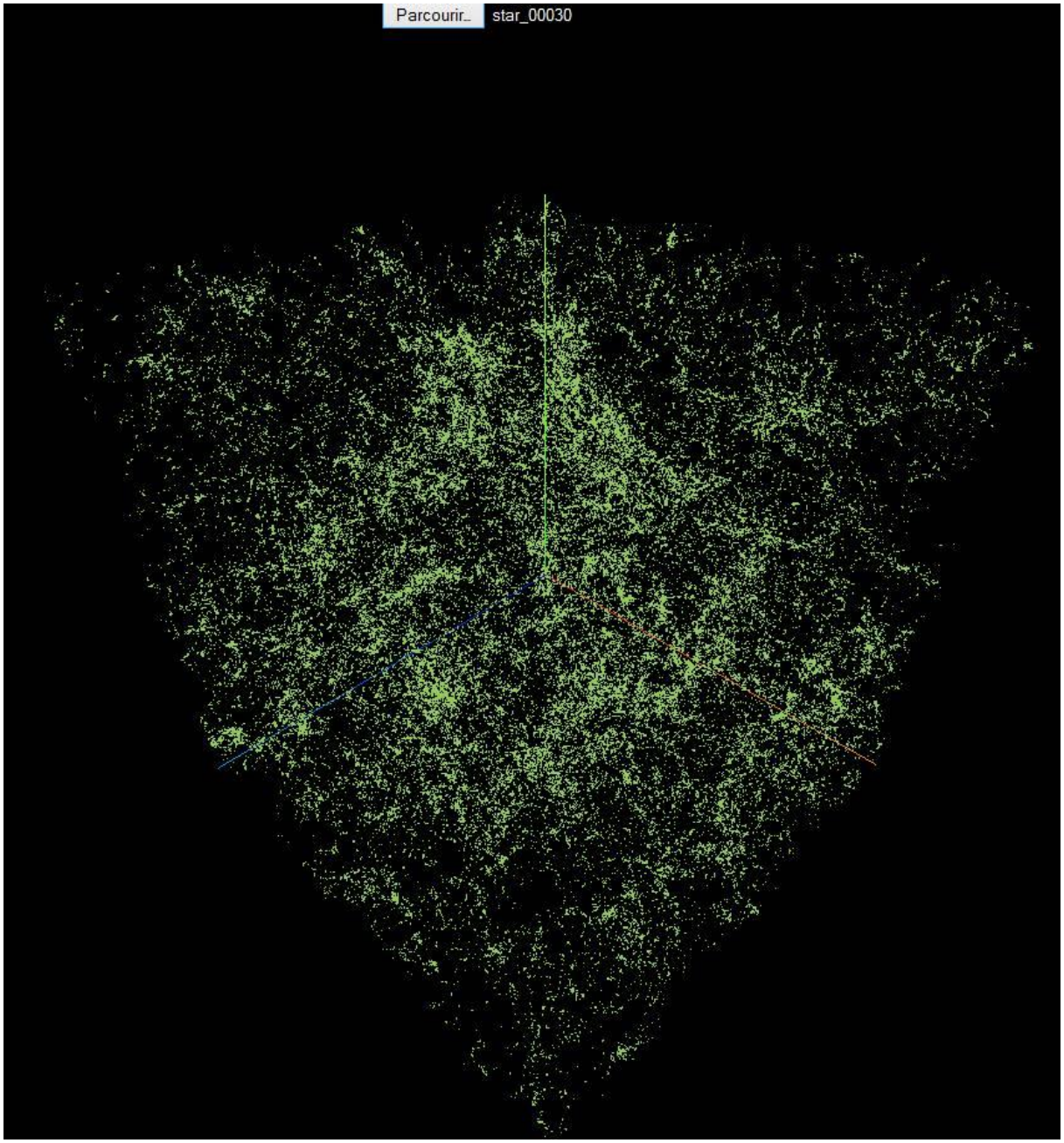


Figure annexe 5 – Simulation CODA modélisation d'étoiles

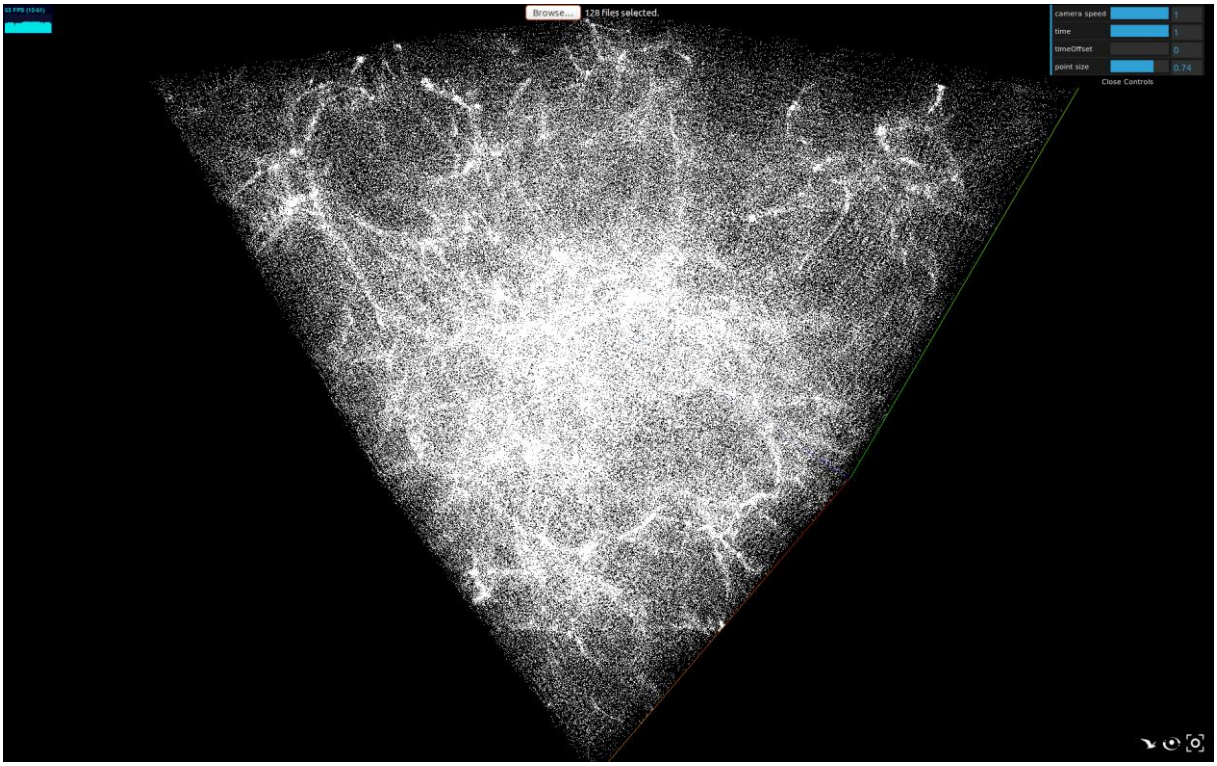


Figure annexe 6 – Modélisation de particules provenant de la simulation EMMA

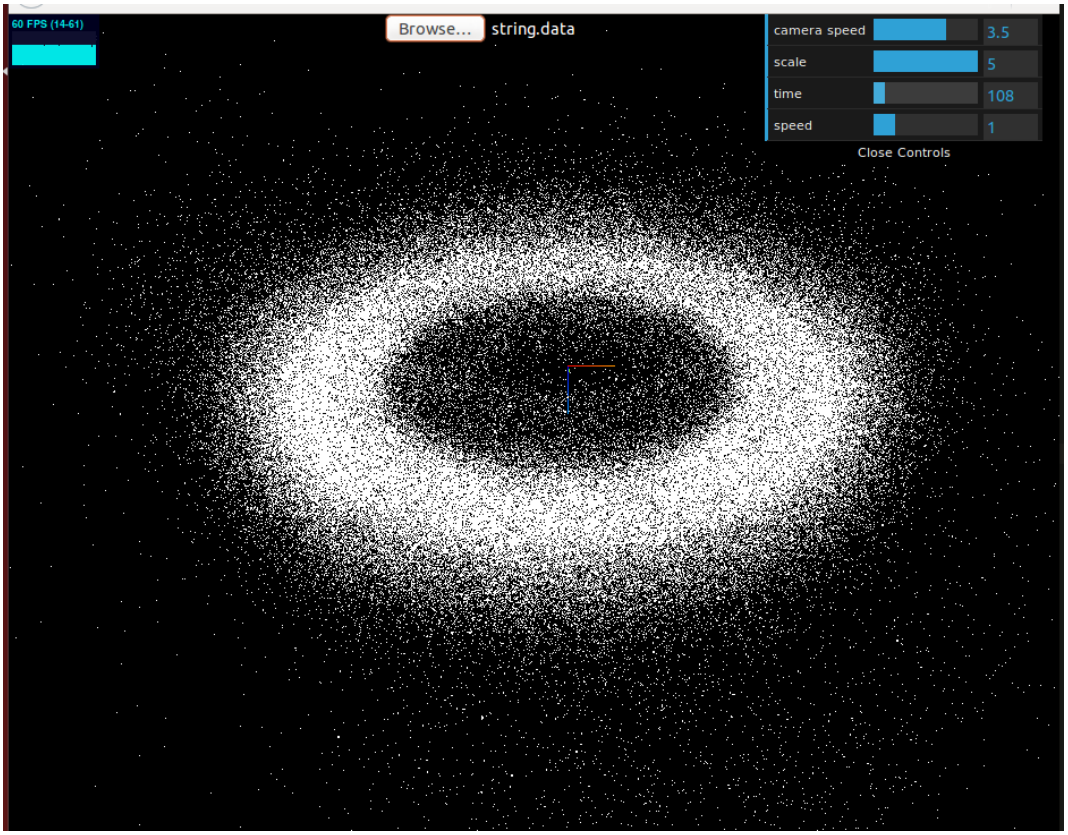


Figure annexe 7 – Modélisation des données de skybot3D

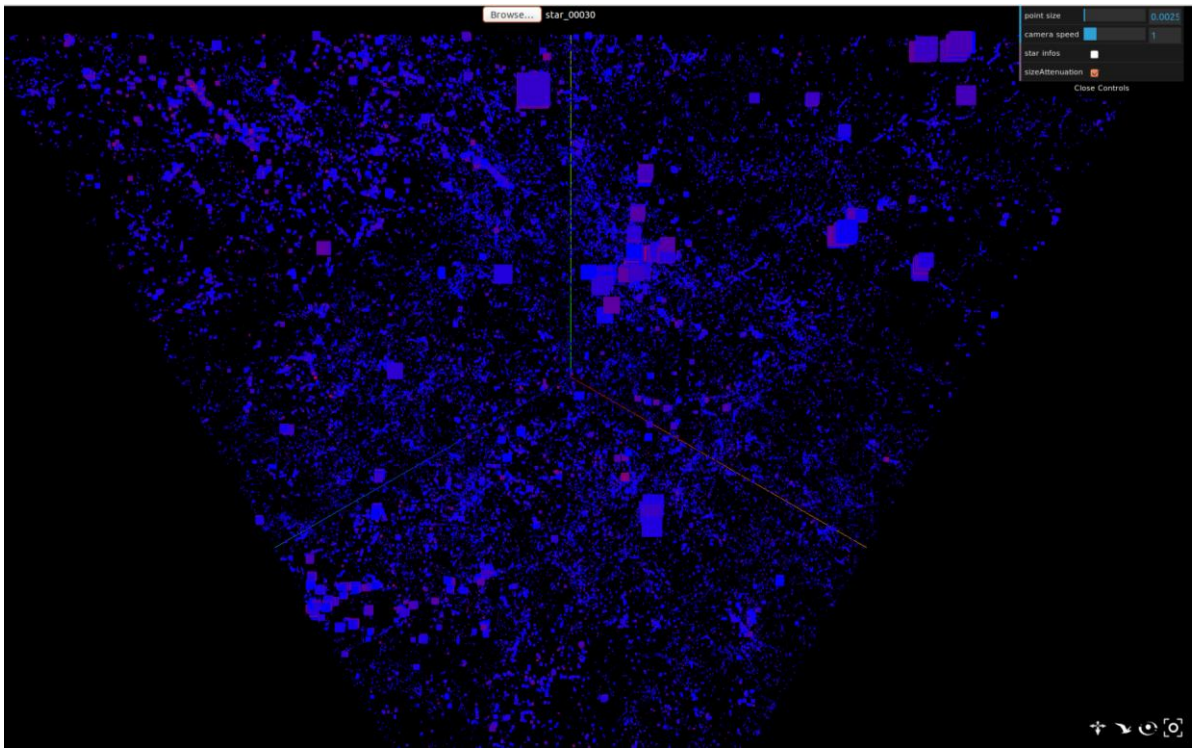


Figure annexe 8 – Simulation CODA avec couleur en fonction de l'âge

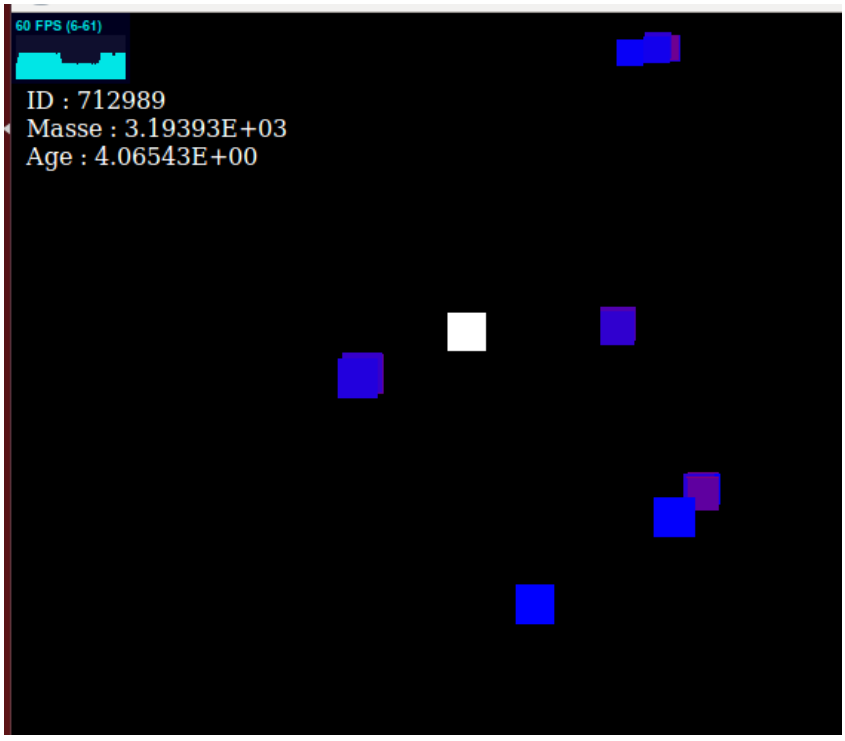


Figure annexe 9 – Simulation CODA, affichage d'information avec raycasting

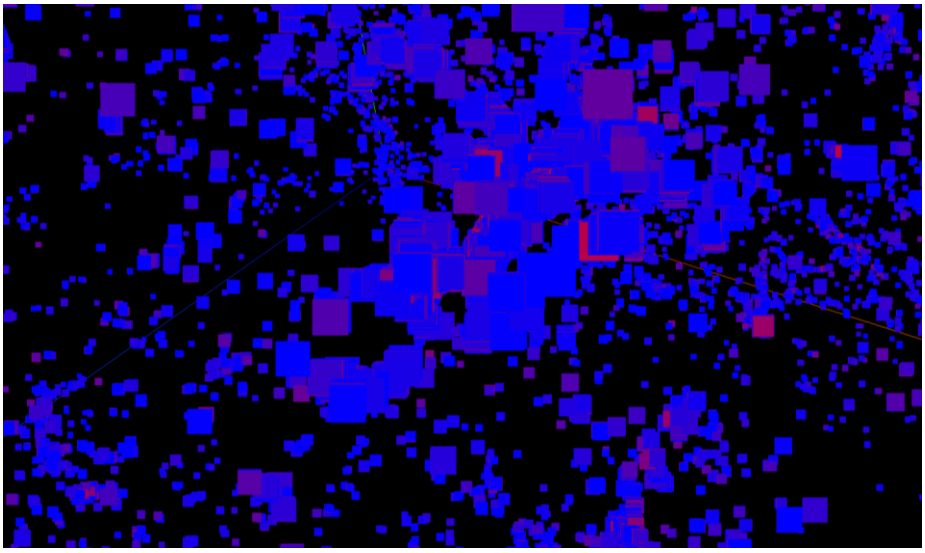


Figure annexe 10 – simulation CODA, agrégat d'étoiles

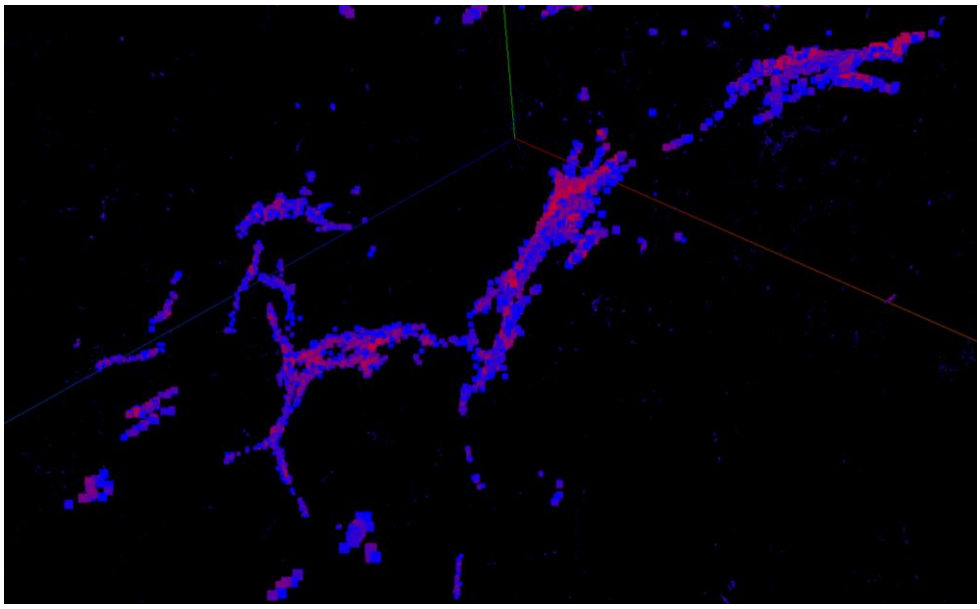


Figure annexe 11 – simulation CODA, zoom sur une structure d'étoiles

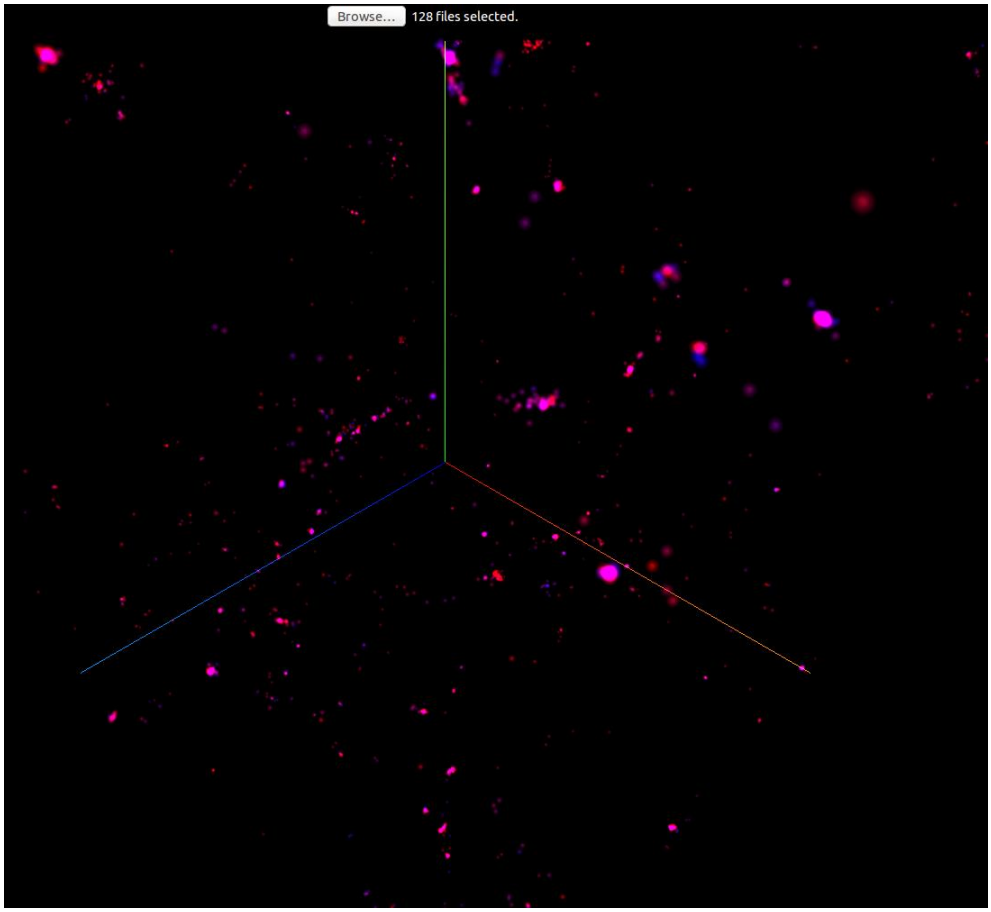


Figure annexe 12 – Simulation EMMA, étoiles avec textures

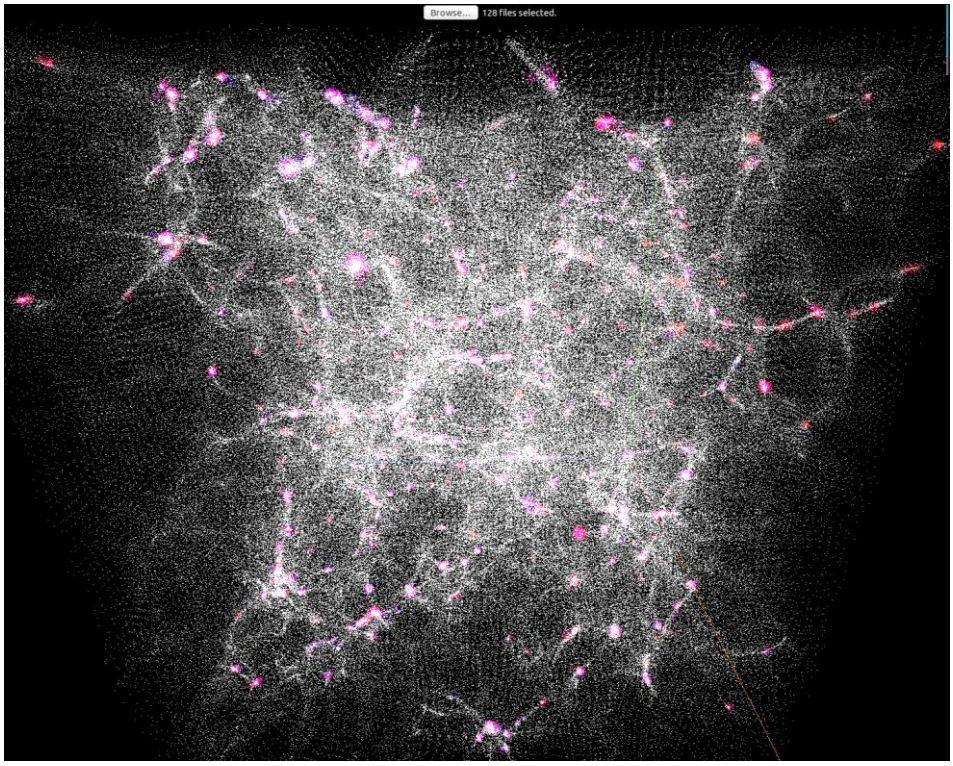


Figure annexe 13 – Simulation EMMA, avec étoiles et particules

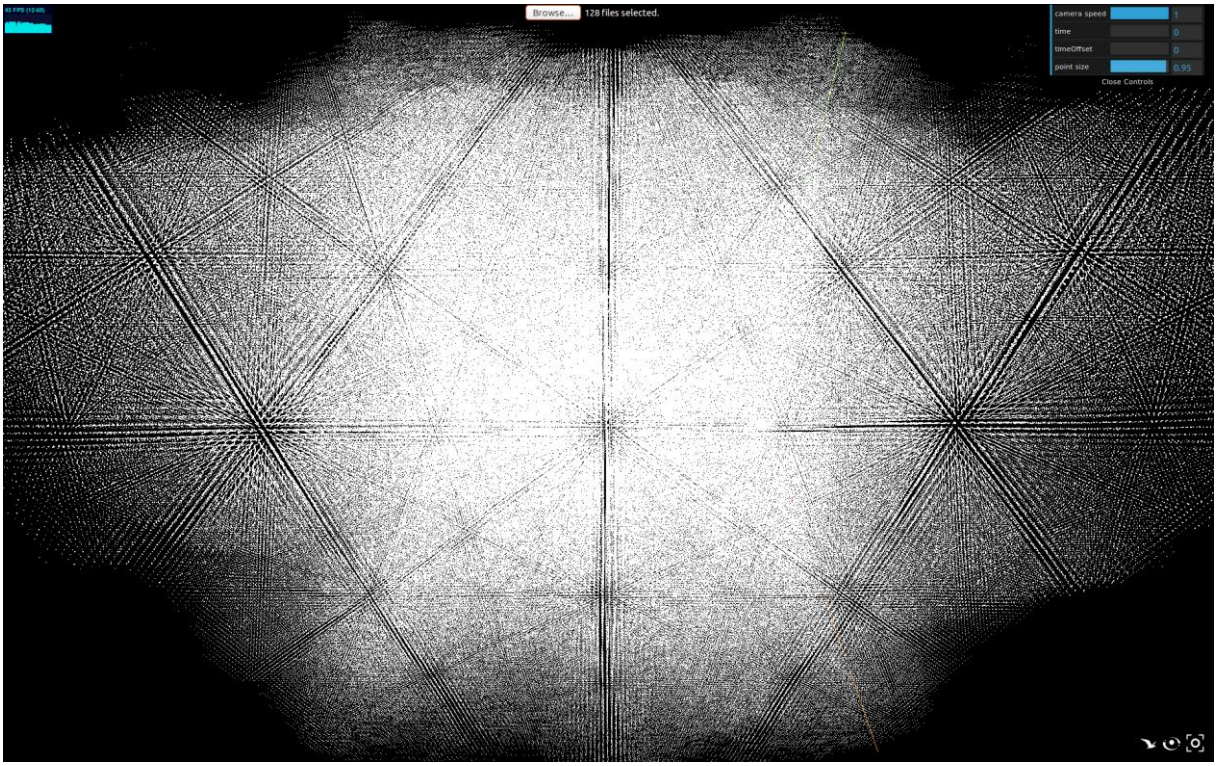


Figure annexe 14 – Simulation EMMA, particules en mouvement début

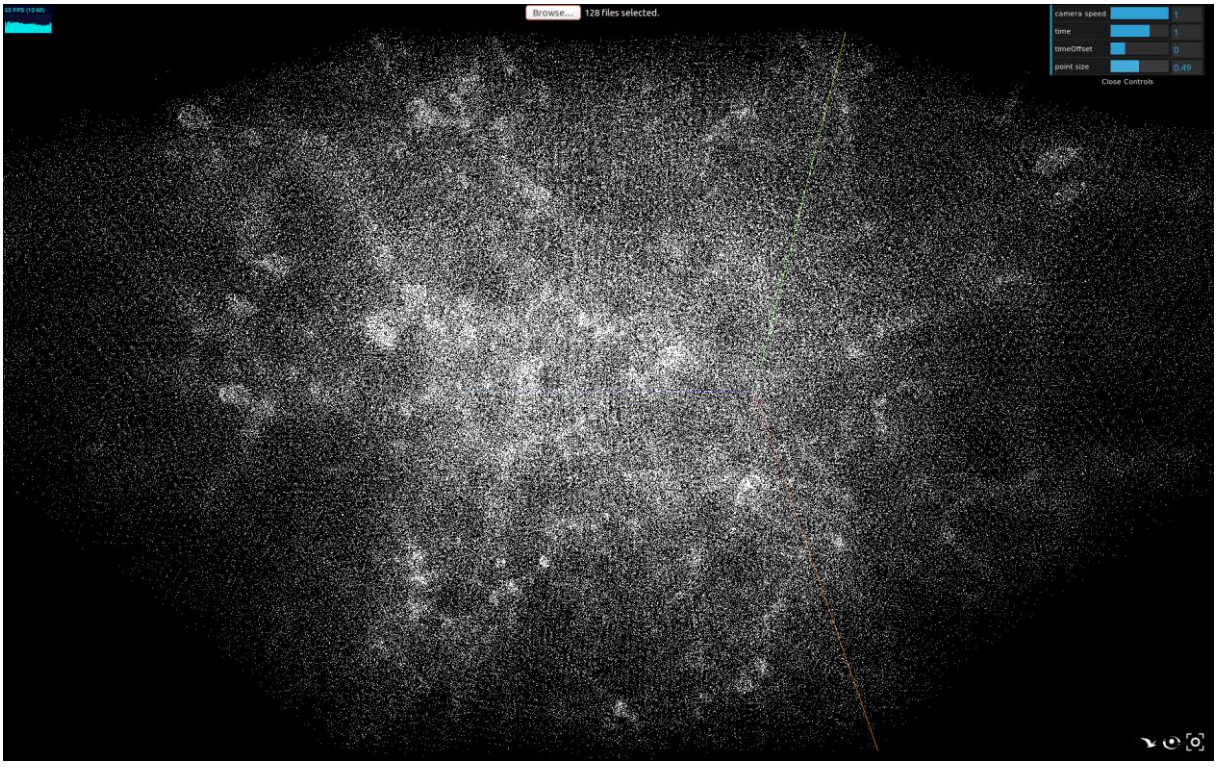


Figure annexe 15 – Simulation EMMA, particules en mouvement intermédiaire

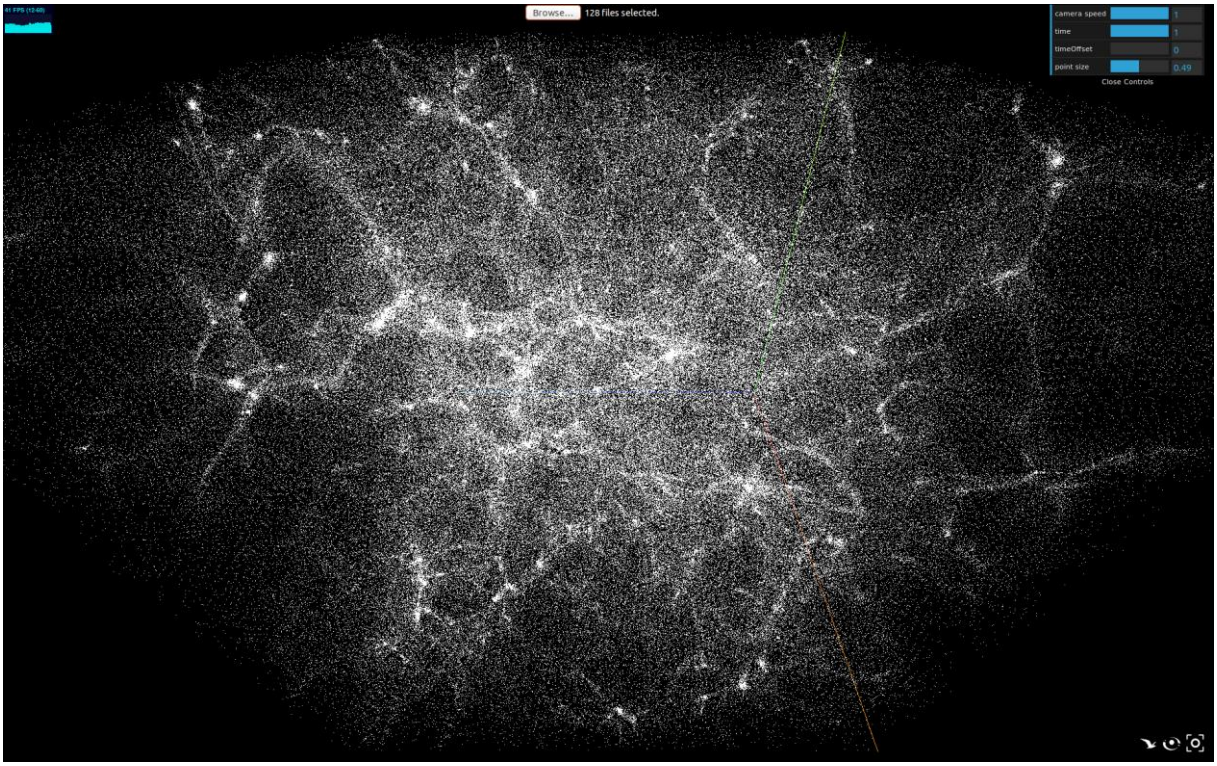


Figure annexe 15 – Simulation EMMA, particules en mouvement fin