Rova RASOANAIVO
Promotion 2020

UNIVERSITÉ DE LORRAINE

Rapport de stage - DUT Informatique

IUT Saint-Dié-des-Vosges

Visualisation progressive en 3D de données astronomiques dans un navigateur

Observatoire astronomique de Strasbourg

CENTRE DE DONNÉES ASTRONOMIQUES DE STRASBOURG

Responsable en entreprise : André SCHAAFF
Responsable enseignant : Pierre-Frédéric VILLARD

Récapitulatif - Rapport de stage 2A

Contexte : Ce stage devait être réalisé à l'Observatoire Astronomique de Strasbourg dans le cadre de la validation de mon Diplôme Universitaire de Technologie (DUT) à l'IUT de Saint-Dié-des-Vosges. Cependant, à cause d'évènements extérieurs, il a été entièrement réalisé en télétravail depuis mon domicile.

Durée du stage : 10 semaines, du **6 Avril 2020 au 12 juin 2020**

Mots-clés : Visualisation 3D, Simulation, Three.js, Octree, Navigateur, WebAssembly, Performances

Matériel à disposition : Ordinateur portable sous Windows 10 avec 8Go de RAM et un GPU Nvidia GTX 1060 Max-Q 3Go

Maître de stage : André SCHAAFF - Ingénieur de recherche au CNRS-CDS

Tuteur enseignant : Pierre-Frédéric VILLARD - Enseignant-Chercheur LORIA

Pour plus d'informations, contactez les parties prenantes :

Rova RASOANAIVO 56 rue Pierre Evrat 88100, Saint-Dié-des-Vosges rasoanaivo.rova@gmail.com	IUT Saint-Dié-des-Vosges 11 rue de l'Université 88100, Saint-Dié-des-Vosges Tél : 03 72 74 95 00 iutsd-info-sec@univ-lorraine.fr	Observatoire Astronomique de Strasbourg 11 rue de l'Université 67000, Strasbourg Tél : 03 68 85 24 50

Remerciements

En premier lieu, je tiens à remercier M. André SCHAAF, mon maître de stage, qui m'a encadré et conseillé tout au long de ces 10 semaines. Je suis très reconnaissant de la confiance qu'il m'a accordée et le temps qu'il a consacré à répondre à toutes mes interrogations.

Je souhaite également remercier M. Pierre-Frédéric VILLARD, tuteur enseignant, pour le suivi pédagogique du stage.

Mes remerciements s'adressent également aux collaborateurs du Centre de Données Astronomiques de Strasbourg (CDS) pour l'accueil chaleureux et nos différents échanges autour des services du centre.

Enfin, je remercie tous mes proches pour leur soutien et leurs encouragements.

Table des Matières

Introduction

Ce rapport a pour but de retracer mon stage DUT de deuxième année à l'Institut Universitaire de Technologie de Saint-Dié-des-Vosges, qui devait normalement se dérouler au sein du Centre de Données astronomiques de Strasbourg. Ce centre est rattaché à l'Observatoire Astronomique de Strasbourg (ObAS), une unité mixte de recherche du CNRS Alsace, qui chaque année reçoit plusieurs stagiaires de tous niveaux (BAC+2 à BAC+5) afin de réaliser des missions souvent liées à la Recherche et au Développement en Informatique.

Le sujet du stage fut la *Visualisation progressive en 3D de données astronomiques dans un navigateur*. Je devais reprendre le développement d'une application existante créée par plusieurs stagiaires. Ce stage comporte plusieurs aspects : le rajout de nouvelles fonctionnalités côté client avec la librairie Three.js et la palette d'outils créée par Thibault BOUCHARD, un des stagiaires en 2016, mais aussi l'amélioration des performances grâce à l'implémentation de nouvelles technologies comme le WebAssembly.
Mon rôle était donc de peaufiner l'application afin qu'elle puisse être déployée au grand public.

J'ai été tout de suite séduit par ce sujet car la formation que je suis à l'IUT m'avait déjà introduit au WebGL et aux techniques d'animation 3D dans le navigateur. Cependant, le stage m'a poussé à approfondir d'autres domaines et à développer ma curiosité car l'application nécessite de connaître à la fois les techniques de programmation Web avancées et les rouages côté serveur.

Dans ce rapport, je vais, tout d'abord, vous présenter l'Observatoire. Ensuite, j'aborderai les objectifs du stage, sa réalisation et enfin, j'établirai un bilan.

1 L'Observatoire Astronomique

1.1 Présentation générale

L'Observatoire astronomique de Strasbourg est un Observatoire des Sciences de l'Univers, une école interne et UFR de l'Université de Strasbourg, ainsi qu'une Unité Mixte de Recherche entre l'Université et le CNRS. Il est dirigé par Pierre-Alain Duc depuis 2017.

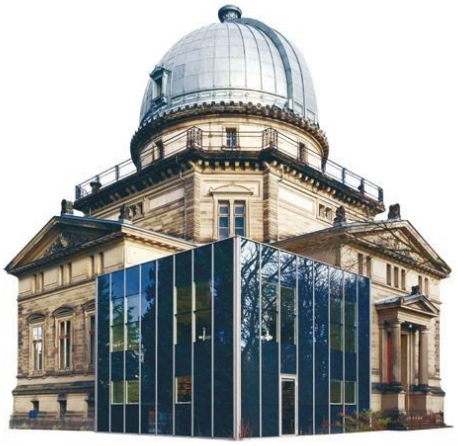


Figure 1 – Grande coupole de l'Observatoire *(topic-topos.com)*

Au départ, sa fonction première fut l'observation des étoiles grâce à la 3ème plus grande lunette astronomique de France dont elle disposait dans sa grande coupole. De nos jours, elle n'est plus utilisée et la majorité des travaux réalisés à l'Observatoire relèvent de l'exploitation et la diffusion de données astronomiques.

Ainsi, si l'on devait résumer ses principales missions :

- **la recherche** via ses différentes équipes de recherches :
  - L'équipe GALHECOS : «Galaxies, High Energy, Cosmology, Compact Objects & Stars» qui étudie la formation et l'évolution des galaxies et de notre Galaxie dans un contexte cosmologique.
  - Le Centre de Données Astronomiques de Strasbourg (CDS) qui collecte et maintient à jour les informations sur les objets astronomiques et les diffuse à la communauté scientifique. C'est avec cette équipe que j'ai réalisé mon stage.

- **l'enseignement** car il délivre le Master spécialité astrophysique de l'université de Strasbourg et forme à la Licence, à l'agrégation et voire même au CAPES.

- **l'observation astronomique** via la mise à disposition d'outils (cf. Partie 1.2) tels que Simbad, VizieR ou encore Aladin à la communauté scientifique internationale et au grand public.

- **la diffusion des connaissances** au grand public à travers le planétarium et l'organisation de divers projets comme *la restauration et la valorisation du globe de Coronelli*[1], dernièrement.

Voici les années clés de l'Observatoire :

1673 : Création du premier observatoire de Strasbourg sur l'une des tours de l'enceinte de la ville.

1828 : Création du second observatoire de Strasbourg sur le toit des bâtiments de l'Académie.

1881 : Inauguration de l'actuel observatoire.

1972 : Création du CDS (sous le nom Centre de Données Stellaires puis devient en 1982 le Centre de Données astronomiques de Strasbourg).

1981 : Création du planétarium dans les locaux de l'observatoire.

2008 : Le CDS est labellisé "Très Grande Infrastructure de Recherche".

1.2 Structure et services du CDS

Structure

Durant mon stage, j'ai été rattaché au CDS, sous la direction de M. André Schaaff. J'ai été rejoint par plusieurs stagiaires venant de formations différentes (MMI, INFO, etc.).
Les 38 personnes travaillant au CDS sont réparties selon 4 départements distincts :

- **La Direction** avec Mark Allen en tant que directeur général du centre.

[1] Projet de valorisation du globe de Coronelli : https://astro.unistra.fr/uploads/media/depliant-globe-coronelli-2-bd.pdf

- **Le service Documentation** avec les documentalistes qui enrichissent et maintiennent à jour les catalogues et les bases de données du CDS.
- **La direction Technique** dont la responsabilité revient à P. Fernique. Il s'assure que toute l'infrastructure technique (serveurs, ordinateurs, réseaux, sécurité informatique, etc.) fonctionne en permanence correctement.

- **Le service Développement** qui s'occupe de développer et entretenir les outils informatiques et les services cités auparavant.

Services

Comme dit précédemment, le CDS met à disposition, en libre accès sur son portail Web[2], trois services qu'elle développe et maintient :

Service	Logo	Description
Aladin		Aladin est un **atlas interactif** du ciel. Il permet aux utilisateurs de visualiser des images digitalisées et des relevés complets du ciel.
Simbad		Simbad est un service permettant de lancer des requêtes personnalisées (identifiant, position, propriétés physiques, etc.) sur une **base de données** contenant quasiment 11 millions d'objets référencés par plus de 35 millions d'identifiants.
VizieR		VizieR est une bibliothèque de **catalogues astronomiques** accessibles via de multiples interfaces. Des outils sont disponibles afin que l'utilisateur puisse récupérer les données sous le même format et selon des critères personnalisés. Un objet peut souvent être référencé par plusieurs catalogues sur les ~15000 catalogues présents.

[2] Lien vers le portail en ligne du CDS : https://cdsweb.u-strasbg.fr/index-fr.gml

2 Problématiques

2.1 Contexte du stage

Cette année, mon stage se porte sur l'**amélioration de l'existant**. Il s'agit d'une application Web de visualisation 3D dans le navigateur baptisée **Jasmine** (Javascript AStronomical data MINEr). Cette application permet à son utilisateur de visualiser des nuages de points dans l'espace. Ces points sont fournis par des jeux de données astronomiques, issues de simulations, de très grandes tailles, fournies par le CDS. Ces jeux de données décrivent une partie cubique de l'espace d'où l'appellation cube de données.
N'importe quel jeu de données contenant des coordonnées XYZ peut suffire afin de constituer un nuage de points. Cependant, en astronomie, nous avons souvent d'autres informations telles que leur masse, leur âge ou encore leur énergie. Ces jeux de données sont parfois binaires mais aussi lisibles par l'humain. Ainsi le rôle de Jasmine est de **lire**, de **traiter**, et enfin d'**afficher** ces nuages de points grâce au moteur de rendu de Three.js. L'application possède déjà plein de fonctionnalités avancées et l'interface de contrôle est conviviale. De plus, on ne fait pas que visualiser, on peut aussi **manipuler les points**. Par exemple, nous pouvons mettre en valeur certains points, zoomer, filtrer, animer, comparer plusieurs jeux de données, etc. La liste est longue. Les principaux critères ayant mené à la création de Jasmine en 2014 sont l'**accessibilité** de l'application via un simple navigateur (donc peu de prérequis) et le fait de pouvoir **se déplacer** dans le nuage de points. Les stagiaires me précédant ont déjà fait un gros travail sur ces points.

Mon objectif est donc de peaufiner l'application afin qu'elle puisse être déployée au grand public. C'est-à-dire, optimiser les performances, la préparer au déploiement, corriger les dysfonctionnements et proposer/rajouter des fonctionnalités si besoin. Ma mission doit aussi s'inscrire dans une démarche d'amélioration constante et de veille technologique. C'est pour cela que ce projet a trait à la recherche et au développement.

2.2 Présentation de Jasmine

Jasmine dispose de deux parties distinctes :

- **Côté client :**

C'est une interface dans le navigateur disposant d'une palette d'outils et d'une ou plusieurs vues sur les nuages de points. Cette partie est très fournie en fonctionnalités et est assez aboutie. Nous pouvons charger des jeux de données locaux via cette interface et les manipuler à notre guise. Ci-dessous une capture de l'application :

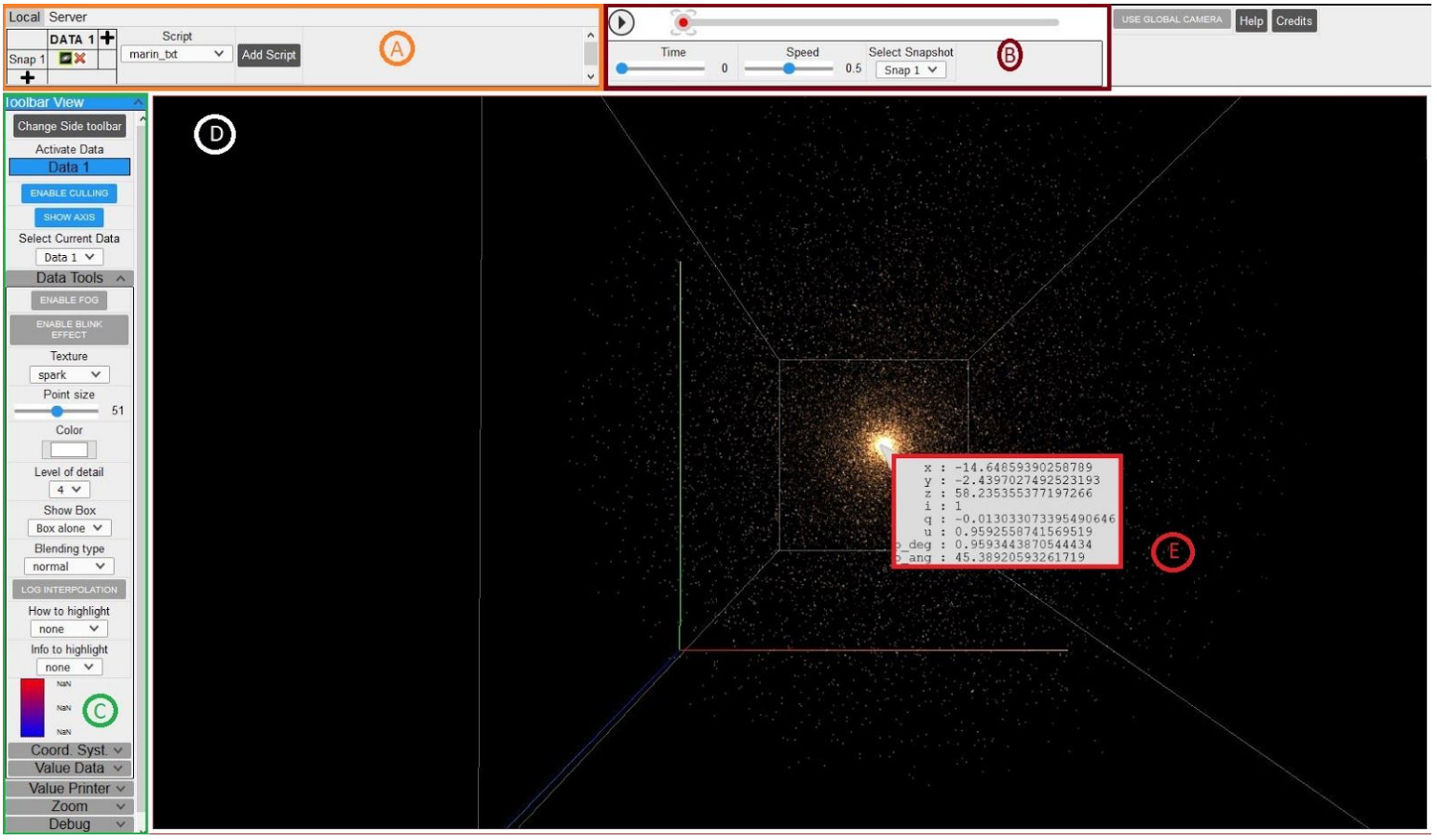


Fig 2 – Vue de la partie Client

Pour l'exemple, j'ai chargé une jeu de données dans l'application. On distingue ainsi 5 parties étiquetées de A à E :

A. Data Manager : c'est ici que l'on sélectionne le type du jeu de données (et par extension le script qui va le charger) et où l'on le charge dans l'application.
B. Timeline : cette timeline permet d'animer la simulation par interpolation si l'on dispose de plusieurs snapshots dans le temps de celle-ci.
C. Palette d'outils : ce sont les outils permettant de manipuler la vue, les données, etc.
D. Vue : elle contient la scène Three.js de la simulation du jeu de données. Elle permet de se déplacer et d'interagir avec la simulation.
E. Informations : ce billboard s'affiche lorsque l'on clique sur un point de la simulation. Il affiche ses informations fournies dans le jeu de données.

Cette partie client est entièrement développée en **HTML5** et **CSS** avec la simulation qui est codée avec **JavaScript**. Elle utilise quasiment toutes les techniques et optimisations que le **WebGL** et **Three.js** mettent à disposition. De plus, pour l'immersion de l'utilisateur, il est possible d'utiliser un **casque de réalité virtuelle**. Pour activer la **timeline** et faire les **simulations**, le principe des Shaders est utilisé afin d'envoyer les calculs d'**interpolation** vers le GPU.

Octree

La partie client utilise une représentation des données utilisant le principe de l'Octree. C'est une structure d'arbre 8-aire. Il permet une partition de l'espace de manière récursive. Ainsi, dans la simulation on peut par exemple optimiser l'affichage on ne calculant que les points dans le cône de vue de la caméra. On peut aussi optimiser la sélection des points en évitant de passer en revue l'ensemble des éléments chargés mais uniquement ceux contenus dans les octants traversés par le rayon de sélection. Il existe d'autres techniques liées à l'Octree mais ces deux ont été retenues pour l'application.

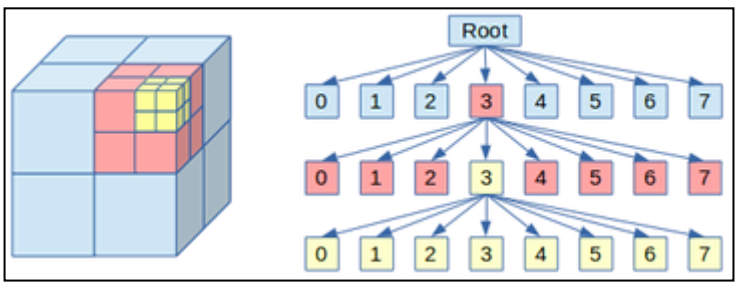


Fig 3 – Structure en Octree et son arbre associé

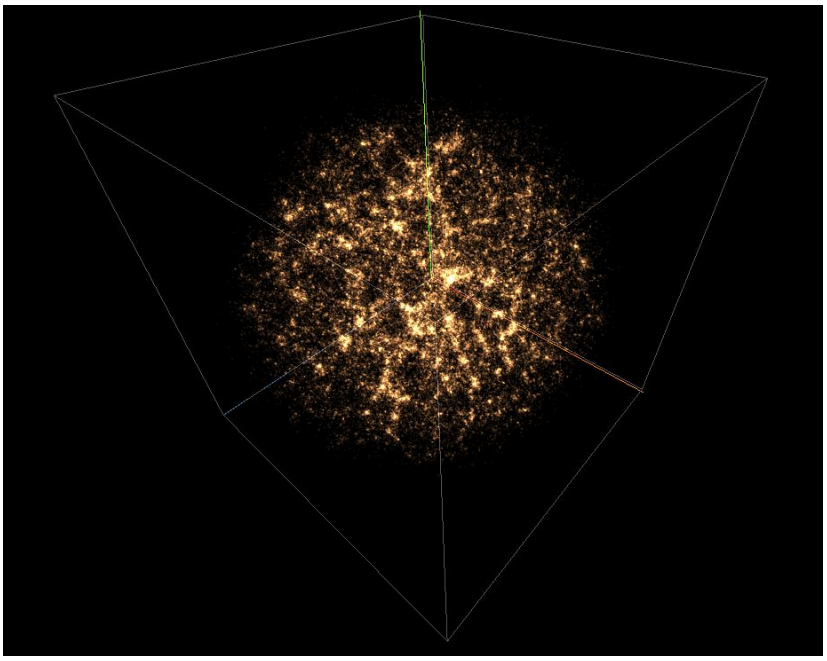


Fig 4 – Données issues du télescope Hubble

❖ **Côté serveur :**

Le côté serveur utilise node.js. Cette partie est utilisée par le client pour accéder à des jeux de données très volumineux (allant jusqu'au Téraoctet) que le navigateur ne peut charger localement du fait de ses limitations. Les données sont traitées en amont sur le serveur (partitionnement, indexation, etc.). Ainsi, l'utilisateur peut charger uniquement les parties qui l'intéressent. Le côté serveur partitionne également ses jeux de données en Octree. Ici, c'est le système de fichiers qui implémente l'Octree. Le haut de l'arborescence consiste en une suite de répertoires et de fichiers, ces derniers ayant une taille limite arbitraire de 200 Mo. Un noeud de l'arborescence physique est simplement représenté par un répertoire, et chacun de ses 8 fils peut soit être un autre répertoire, soit un fichier, chaque fils étant nommé via un code représentant sa position au sein de son parent.
Lorsque ce système de fichiers est généré et que le serveur est en marche, le client peut s'y connecter via le client et accéder à l'onglet *Serveur* pour télécharger les données.

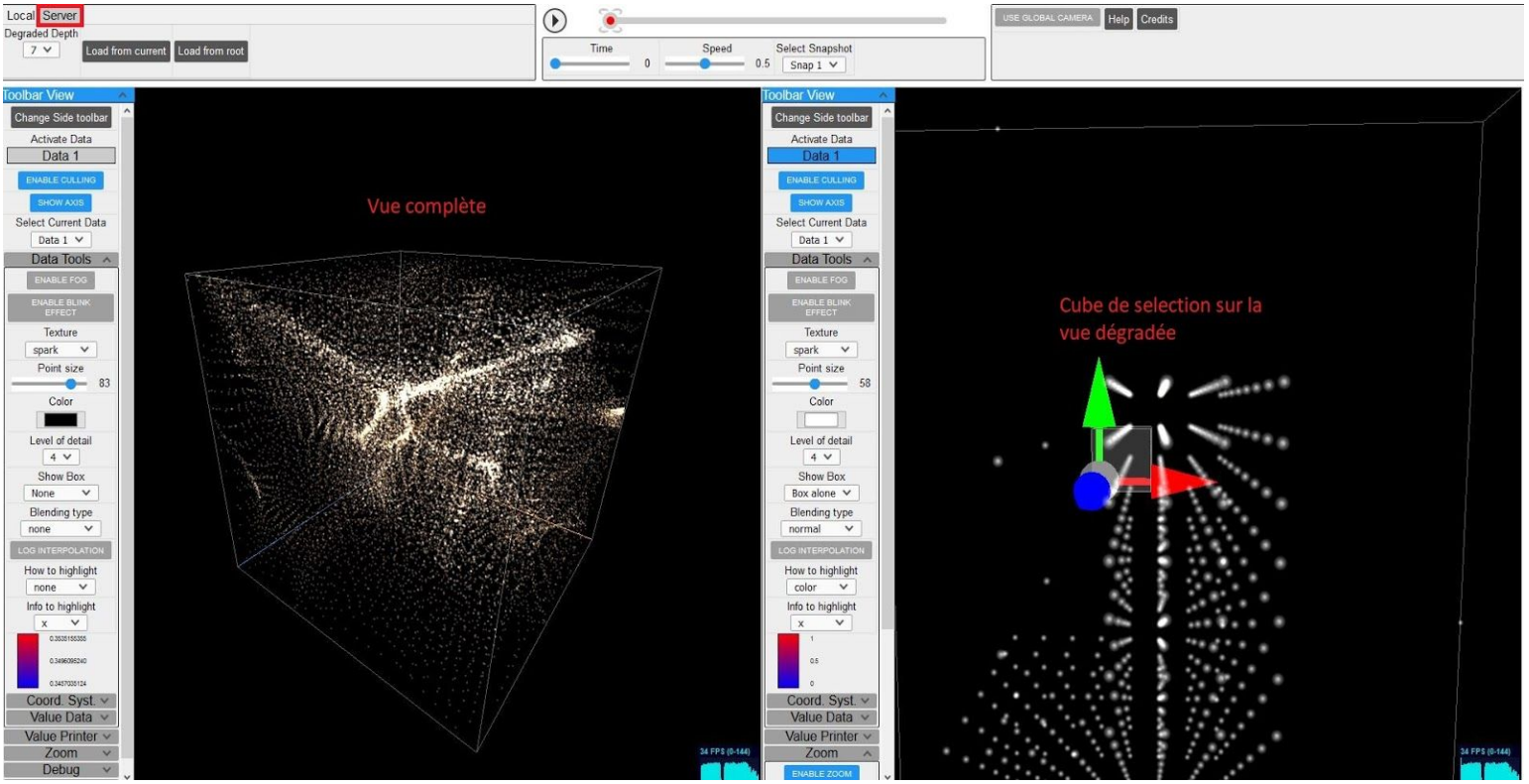


Fig 5 – Fonctionnement du mode Server

Comme vous pouvez le voir sur cette capture, la vue de droite représente une modélisation dégradée de toutes les données disponibles sur le serveur. En déplaçant le cube de sélection, l'utilisateur charge, sur la vue de gauche, depuis le serveur, les données correspondantes à l'octant sélectionné.

Cette partie a été développée par Jérôme Desroziers, stagiaire en 2017 et je vous invite à vous référer à son rapport de stage pour plus de détails.

2.3 État des lieux de l'application

Je vais citer dans cette partie les problèmes que rencontre l'application tant côté client que côté serveur. Je détaillerai dans la prochaine partie les techniques mises en oeuvre afin de les régler.

❖ Côté client :

-> La **génération de l'Octree est très lente** au chargement initial, car il faut que l'application traite chaque point du nuage. De plus, ce processus bloque le thread principal de JS tant qu'il n'est pas terminé car il est obligatoire afin d'afficher le nuage de points. Ainsi, avec un très gros jeu de données (coeur scanné 125*125*125 = 1 953 125 points) on a le résultat suivant :

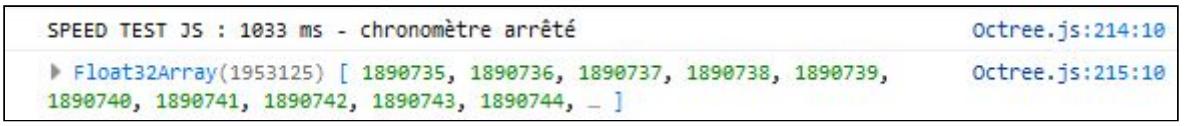


Fig 6 – Performances avec un algorithme JavaScript

En moyenne, on tourne autour de 1000 ms pour 125^3 mais rappelons-le, je fais ce test sur une machine puissante donc sur des machines moins puissantes, cela peut prendre plus de temps. De plus cette opération peut être amenée à être répétée dans le cas de la génération d'une vue dégradée. Potentiellement, le temps de chargement peut avoisiner les 2000 ms. Cependant, après ce chargement coûteux, l'application gagne grandement en performances.

-> L'application tient en **mémoire** les coordonnées des points tant qu'elles sont utilisées. Cependant, parfois, même si le jeu de données est désactivé dans la vue, il reste en mémoire. À cause des limitations de JavaScript nous ne pouvons pas gérer manuellement la libération de la mémoire. Et tant qu'il existe encore des références aux objets utilisés, ces derniers ne seront pas libérés. Cette procédure est faite automatiquement par le *Garbage Collector* ou ramasse-miettes en français. Ce procédé ne peut être qu'une approximation car savoir si tel ou tel fragment de mémoire est nécessaire est un problème indécidable (autrement dit, ce problème ne peut être résolu par un algorithme). Je devais donc trouver un moyen afin que l'application puisse avoir plus de contrôle sur ce point là.

-> **L'application manque de fonctionnalités** au niveau de la caméra. Certes, plusieurs caméras sont utilisées et des angles de vues différentes sont disponibles. Cependant il manque des outils permettant de personnaliser ces caméras (angle de vue, champ de vue FOV, type de caméra, etc.)

-> Il y a encore un **travail à faire sur l'ergonomie de l'interface**. En effet certaines fonctionnalités existent mais l'utilisateur n'est pas informé sur leur existence via l'interface. Les touches auxquelles elles sont liées ne sont pas du tout intuitives et il est souvent nécessaire d'aller ouvrir la documentation pour une action basique.

-> **L'application ne dispose pas de version déployable**. C'est-à-dire qu'une partie de la documentation se trouve encore dans le code source et peut se retrouver chez l'utilisateur. Ce n'est pas forcément voulu. De plus, la taille du code source peut encore être optimisée avec des techniques comme la Minification ou l'Uglification afin que les scripts (donc par extension les pages) se chargent plus vite dans le navigateur. Je détaillerai ces techniques plus tard.

❖ **Côté serveur :**

-> Mettre en place le côté serveur reste encore assez complexe à faire. La méthode n'est pas encore aboutie et n'est pas assez documentée. Cependant le serveur fonctionne très bien sous node.js et l'implémentation en Octree du système de fichiers est une réussite.

-> Il reste encore beaucoup de nettoyage et de "Refactoring" de code à faire même si la version de Jérôme Desroziers est déjà bien avancée.

2.3 Objectifs du stage

Pendant le stage j'ai eu carte blanche sur le choix des solutions appliquées tant qu'elles respectent toutefois quelques critères :

- La **portabilité** de l'application sur les différents navigateurs (HTML5, CSS3, JS8) et même Jupyter Notebook.
- L'usage de la mémoire et de la puissance de calcul doit être raisonnable et réfléchie car les futurs utilisateurs n'ont pas forcément une machine "puissante".
- Le code de l'application doit être **maintenable et documenté**.
- Utiliser le **GPU** pour les calculs lourds.
- Peser le pour ou le contre sur l'utilisation des **nouvelles technologies** existantes. La programmation se fait généralement avec les langages Web Par exemple, le WebAssembly ou WebGPU. Toujours se projeter vers une démarche de recherche et de développement.

À la fin du stage, la version devrait être mise à la disposition du public (notamment de la communauté astronomique). Toutefois, elle devra encore être mise à jour périodiquement, au gré des évolutions technologiques.

2.4 Planification du projet

❖ **Organisation :**

Afin de mener à bien mon projet, j'ai développé et utilisé une méthode de travail qui correspondait bien au format R&D du stage. J'ai consacré beaucoup de temps à la recherche, c'est-à-dire, à la lecture d'articles sur le sujet, de thèses, de la documentation, surtout les deux premières semaines. J'ai aussi beaucoup échangé avec des spécialistes sur des serveurs Discord sur la pertinence des solutions que j'allais implémenter. Par exemple, sur un serveur consacré à Three.js, j'ai pu discuter avec l'un des cocréateurs de cette librairie sur l'avenir de cette technologie face à l'arrivée de WebGPU. Ce n'est qu'un exemple parmi d'autres mais en bref, mon temps se partageait entre la recherche et l'implémentation des fonctionnalités (⅓ ; ⅔).

Pour organiser mes tâches, j'utilisai une simple To-Do List personnelle. La journée terminée, je détaillai sur un Wiki privé de l'Observatoire (un journal de bord à destination des stagiaires) les avancées et les remarques que j'ai pu avoir. Bien évidemment, je discutais régulièrement, en visio avec le tuteur, des idées que j'avais et du travail réalisé.

❖ **Outils utilisés :**

Système d'exploitation	**Editeur**	**Navigateur + debugger**	**Librairies principales**	**Langages**
Windows 10 	Visual Studio Code 	Firefox v77 	-Three.js (client)  -Node.js (serveur) 	**-HTML5** **-CSS3** **-JS8** **-C++** **-C**
Win10 fut mon OS principal mais à côté j'ai utilisé un sous-système Linux pour certains tests.	Simple, efficace. Des extensions permettent de documenter le code, faciliter le développement...	Ce navigateur me permet d'éviter les erreurs cross-origin[3].	Ce sont les principales librairies mais l'application utilise aussi dat.js, stats.js, etc.	C++ pour le WebAssembly. Le reste, classique pour du Web

[3] https://developer.mozilla.org/fr/docs/Web/HTTP/CORS

3 Réalisation du projet

3.1 Partie Client

3.1.1 Ajout de nouvelles fonctionnalités à la palette d'outils

<u>Caméra</u> :

Comme dit précédemment, il y a un travail à faire sur la caméra. Nous avons plusieurs caméras disponibles dans la scène mais il nous manque la possibilité de personnaliser ces derniers. La bibliothèque Three.js nous fournit deux types principaux de caméras : caméra perspective et caméra orthographique.

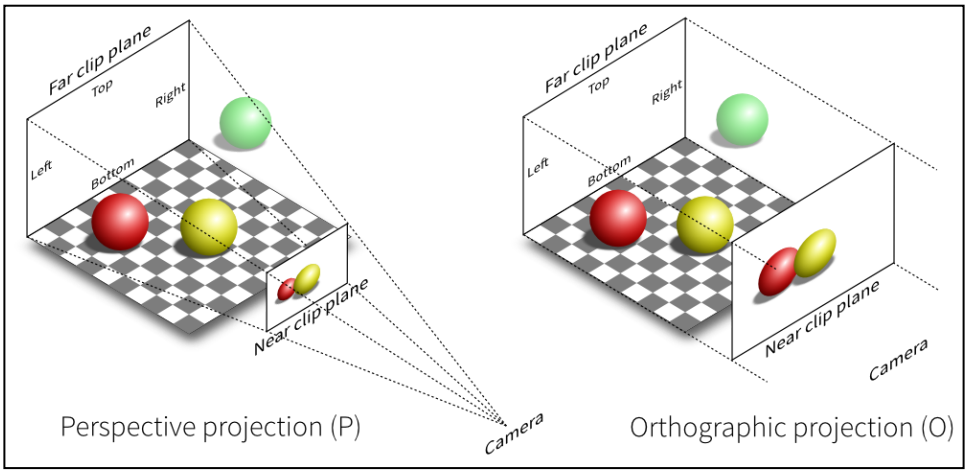
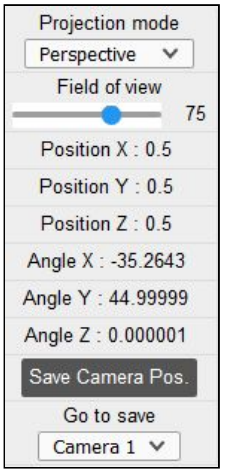


Fig 7 – Les deux types de projections

Il a fallu que je rajoute une section caméra dans la palette d'outils afin que l'on puisse choisir entre ces deux modes de projection. J'ai aussi rajouté la possibilité de changer la FOV de la caméra et ai corrigé un bug qui faisait que leur position soit faussée dans le débugger. Ainsi, l'outil amélioré se présente comme sur la droite (Fig 8 – Outil - Caméra).



On peut aussi changer le type de caméra en mode multivue. Par exemple, décider que l'affichage de gauche soit en perspective et celle de droite orthographique.

J'ai fait en sorte que le déplacement soit toujours possible à l'intérieur de la vue orthographique même si visuellement ce ne soit pas intuitif à cause de la préservation des angles et des distances.

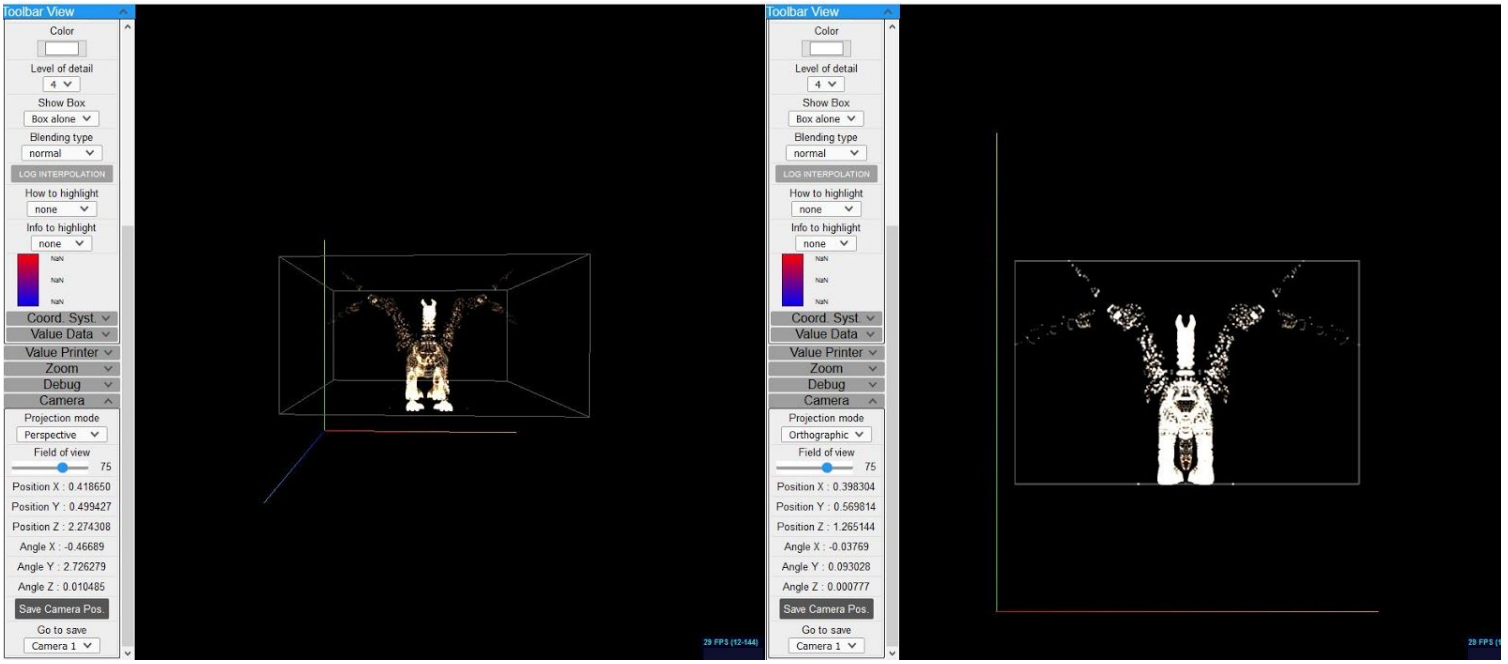


Fig 8 – Le résultat en multivue

<u>Data Manager</u> :

J'ai aussi étudié la manière dont l'utilisateur pouvait supprimer un jeu de données chargé dans l'application. Il existe déjà un outil qui permet de charger des séries de données et des snapshots associés. Cependant, celui-ci ne montre pas clairement comment on peut "désactiver" ces derniers. Dorénavant, il est possible de faire cela directement depuis l'interface.

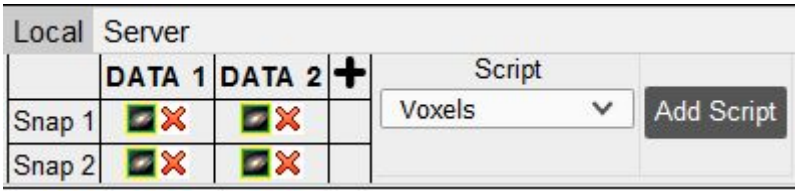


Fig 9 – Outil de chargement de données

J'ai aussi étudié la manière dont Javascript gère la mémoire lors de cette suppression. En réalité, on ne peut pas réellement contrôler le passage du garbage collector. Cependant, on peut essayer de déréférencer chaque objet utilisé lors de sa fin de vie. Pour cela, j'ai dû appliquer plusieurs techniques afin de déceler tout "cycle de références" ou encore mauvaises pratiques concernant les variables globales oubliées.

```
function foo(arg) {
 o = {}; b = {};
 o.a = b;
 b.a = o; //cycle - mauvaise pratique
}
```

```
**************************************************************
function foo(arg) {
    bar = "this is a hidden global variable"; // fuite mémoire
}

**************************************************************
```

Ainsi, j'ai dû m'assurer que ces problèmes ne puissent se produire dans le futur car comme l'application gère de grosses quantités de données, une simple erreur peut tout faire crasher. Dans l'application cela revient à affecter **null** aux attributs d'objets en fin de vie et à s'assurer qu'aucun cycle ne se promène dans l'application.

Au final, en 2020, JavaScript ne permet pas encore de contrôler le Garbage Collector même si l'on peut influencer son comportement.

3.1.2 Ajout d'un nouveau script de lecture de données Voxels

Lors de mon projet tutoré 4, durant ma formation DUT, j'ai eu l'occasion de développer un petit programme capable d'afficher dans le navigateur le nuage de points contenu dans un fichier Voxel. Ce fichier binaire contient le niveau de gris de chaque Voxel mais pas de coordonnées. Ces derniers forment une grille cubique dans laquelle on affiche uniquement les voxels dont le niveau de gris est différent de zéro.

Il se trouve que cette fonctionnalité peut être rajoutée dans Jasmine car le principe reste le même. Il me suffit ainsi de créer un script capable de parser les fichiers et retourner les buffers remplis dont l'application a besoin.

```
var position = new Float32Array(npart * 3);
    var colors = new Int32Array(npart * 3);
    var index = new Float32Array(npart);
    var info = new Array(npart);
....
    -------parsing + remplissage des buffers-----
   Cette partie dépend du format des fichiers (binaire, format, etc.)
....
return [{name: "index", value: index}, {name: "position", value:
position}, {name: 'info',value: info},{name: 'color',value: colors}];
```

Le résultat :

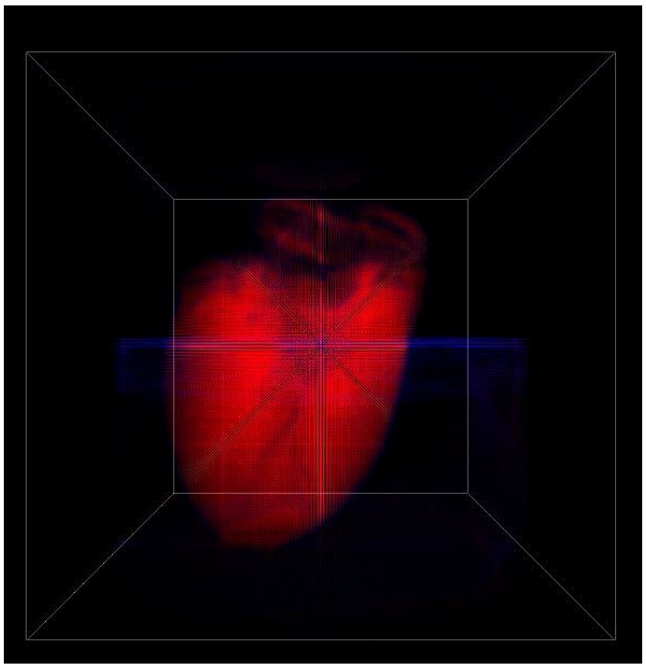


Fig 10 – Coeur scanné en 3D (source : M. Pierre-Frédéric VILLARD)

3.1.3 Version WebAssembly de la génération de l'Octree

À chaque chargement de fichier, l'application génère un octree modélisant l'ensemble de l'espace cubique afin de pouvoir contrôler le nombre de particules affichées à l'écran. Comme dit précédemment dans la liste des problèmes de l'application, la génération de l'octree après le chargement et le parsing (cf. 3.1.2) prend énormément de temps (cf. 2.3). Cela peut impacter sur le confort d'utilisation et la charge sur les ressources système n'est pas optimale avec JavaScript. Sans rentrer dans les détails de l'implémentation de cet octree, j'ai tout de suite vu une opportunité de l'implémenter dans un autre langage qui est très tourné vers les performances : C++. Les avantages à utiliser ce langage sont multiples :

- les performances
- la possibilité de gérer la mémoire car il n'y a pas de garbage collector
- un contrôle des erreurs accru
- le langage est très documenté sur Internet

De plus, lors de mes recherches, j'ai appris que l'on pouvait convertir du code C++ vers le WebAssembly, abrégé wasm, qui permet de le faire tourner dans le navigateur. Cela ne se limite pas au C++, mais l'on peut prendre du code C, Rust, AssemblyScript, etc. **Le code**

sera compilé et tournera à une vitesse quasi native sur le CPU. Il faut toutefois préciser que le WebAssembly n'est pas là pour remplacer JavaScript mais ils sont faits pour travailler ensemble. Un des projets récents (2019) utilisant cette technologie est Google Earth. Comment cela fonctionne-t-il ?

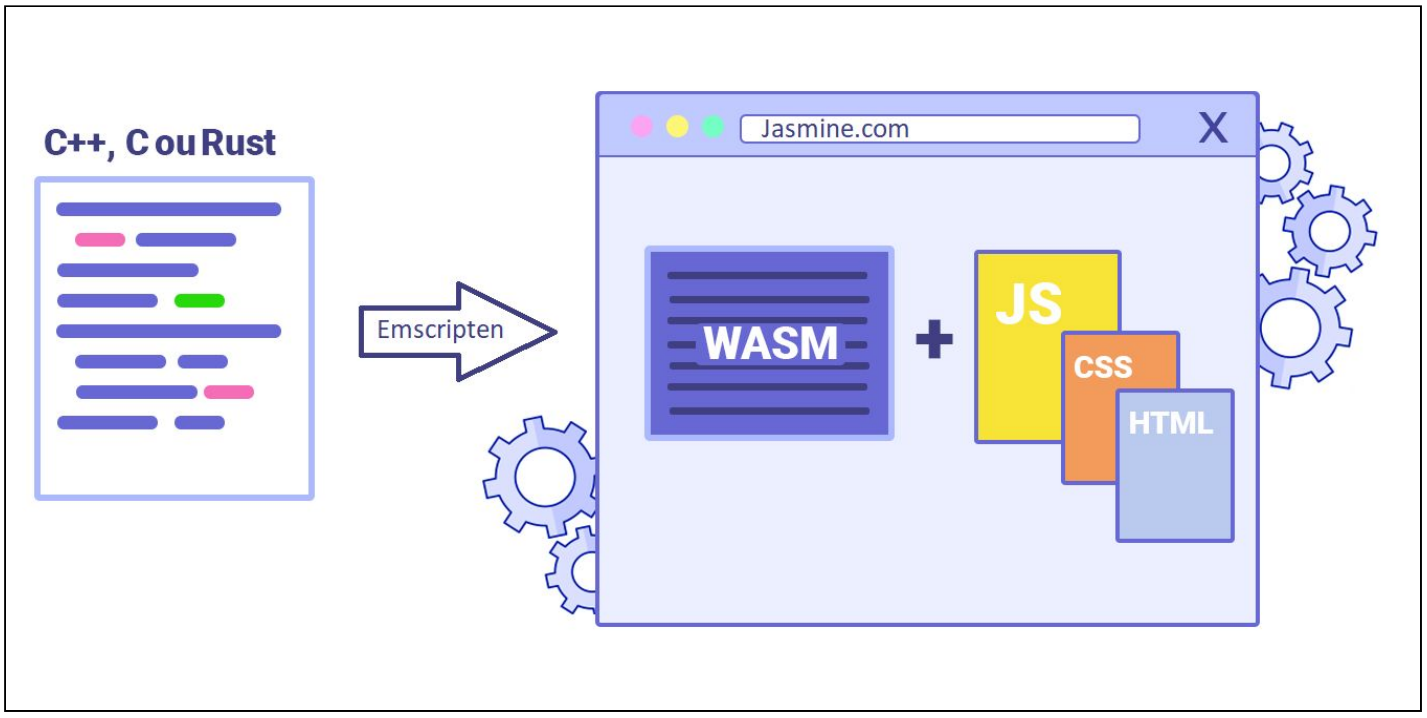


Fig 10 – Mécanisme derrière le WebAssembly

Le principe est "simple", d'abord, le développeur code avec son langage de prédilection. Ce code cible généralement les fonctionnalités intensives. Ensuite, le compilateur nommé Emscripten se charge de le traduire en bytecode WebAssembly. Le fichier produit .wasm sera chargé par le navigateur. Pour le moment, ce fichier ne peut pas encore être chargé tout seul dans le navigateur. Il faut pour cela charger un fichier .js, généré par Emscripten, qui se chargera de faire le pont (on appelle ceci la "glue code") entre les fonctions C++ et Javascript.

Qu'en est-il de la mémoire ? Ce contexte WebAssembly disposera de sa propre mémoire, dans laquelle on peut utiliser à notre convenance les fonctions de C++ tels que malloc() ou free(). Toutefois, le bytecode wasm tourne dans la même sandbox que JavaScript et l'on peut accéder aux données (lecture seule) dont dispose ce dernier mais pas aux fichiers du système.

Concrètement, comment cela se passe-t-il dans l'application?

1. Tout d'abord j'ai traduit en C++ l'algorithme de génération de l'Octree créé par Pierre Lespingal. Il a fallu réécrire toute la classe et utiliser des techniques propres au C++, surtout pour la génération de nombres aléatoires bien plus complexe que Math.random(). Il fallait aussi faire attention aux types des variables, du fait que le C++ soit fortement typé. En voici un extrait au niveau du constructeur :

```
SIMU.Octree = function () {
    this.child = [];
    this.box = null;
    this.hasChild = false;
    this.start = 0;
    this.count = 0;
    this.detailOrder = null;
}; // Javascript devient en C++:

class Octree
{
  public:
    std::vector<Octree> child;
    std::vector<float> box;
    bool hasChild = false;
    int start = 0;
    int count = 0;
    std::vector<float> detailOrder;
    Octree()
    {};
}
std::vector<T> a un comportement relativement proche aux array de JS
```

2. Il faut ensuite compiler ce code avec emscripten avec la commande suivante :

```
emcc Octree.cpp -o Octree.js -s WASM=1 -s
EXTRA_EXPORTED_RUNTIME_METHODS='["cwrap"]' -s ALLOW_MEMORY_GROWTH=1
//cwrap est la fonction qui appelle le C++ et ALLOW_MEMORY_GROWTH
permet de lever les limitations sur la mémoire du navigateur
```

3. J'importe le fichier Octree.js dans l'application et s'ensuit cette série d'étapes :

    a. Je "transforme" la fonction C++ en une fonction JS normale avec cwrap()

```
var js_wrapped_octree_function = Module.cwrap("createOctreeFromPos",
"null", ["number","number","number","number","number"]);
//param : (C++ function name, return type, args)
//type of number can be a pointer, an integer, or a float
```

Remarquez que c'est l'objet Module du fichier js glue code qui gère la transformation.

    b. Pour envoyer des données vers le contexte wasm, il faut tout d'abord allouer de la place dans ce dernier puis copier le tableau js vers le buffer de float de wasm :

```
        /******************ALLOCATING MEMORY*******************/
var position_ptr = Module._malloc(len_position * bytes_per_element);
//param : desired size in bytes = table length * 4 bytes for a float
//return : pointer to the first allocated byte

        /******************COPYING*******************************/
Module.HEAPF32.set(position_table, position_ptr / bytes_per_element);
```

4. Enfin, l'appel de la fonction C++ et la recopie des résultats :

```
//call to the C++ createFromOctree function
js_wrapped_octree_function(position_ptr,len_position,index_ptr,len_in
dex,result_ptr);

//writing memory back to js
var result = new
Float32Array(Module.HEAPF32.buffer,result_ptr,len_index);
```

Comme vous pouvez le voir, le mécanisme est encore assez complexe à mettre en place mais le gain en performances sont considérables. **Dans mes tests (peut varier en fonction du PC, du disque dur, etc.), pour générer l'octree d'un nuage de 125^3 (environ 2M) points, sur JavaScript, il fallait environ 1000ms. Dorénavant, ce processus passe à 400 ms en moyenne.** Ce processus peut encore être amélioré car dans mon implémentation, je fais des

copies obligatoires des données vers le contexte wasm, ce qui est coûteux en temps et en mémoire. En effet, actuellement, il est impossible d'éditer directement la mémoire de JavaScript depuis le wasm.

3.2 Partie Serveur

J'ai passé moins de temps sur cette partie car je n'avais pas forcément trouvé des idées à implémenter vers la fin du stage. Cependant, j'ai passé pas mal de temps à refaire la documentation et à mieux structurer le code afin qu'il soit lisible et maintenable.

3.2.1 Lecture des fichiers, refactoring, tests de déploiement

Dans l'état où la partie serveur était laissée, le côté serveur ne supportait qu'un seul type de fichier. Si l'on veut rajouter d'autres formats, il fallait reprendre les scripts de lecture déjà existants afin de créer des nouveaux Reader (parsers côté serveur). J'ai donc dû migrer tous les scripts côté client vers le côté serveur afin d'avoir plus de liberté dans le choix des formats.

Ensuite le code était un peu en "chantier", il fallait le nettoyer et faire un peu de refactoring. J'ai vu la possibilité d'implémenter ces Reader en WebAssembly côté serveur cependant, je n'ai pas eu le temps de le faire.

Enfin, j'ai pu essayer de déployer l'app node.js avec ngrok[4], juste pour essayer de voir comment l'application se comportait via Internet. Ce qui m'intéressait particulièrement, c'était de voir le temps de chargement de certains octants (fichiers de données sur le système de fichiers) qui font plus de 50 Mo. Au final, rien de spécial, le temps de téléchargement dépendait juste du débit descendant du client. Cependant, j'ai remarqué que les scripts mettaient plus de temps à charger vers le client. Il fallut mettre en oeuvre des techniques de "minification" afin de nettoyer le code de toute documentation inutile pour le client et ainsi réduire la taille de l'application.

Fig 11 – Code minifié : plus de sauts de lignes, noms de fonctions raccourcies, etc.

[4] Tunnel permettant d'exposer un serveur local sur Internet : https://ngrok.com/

4 Bilan

4.1 Difficultés rencontrées

<u>Côté client</u> :

-> Le WebAssembly est une technologie relativement récente et encore en expérimentation. De ce fait, la documentation en ligne n'est pas encore assez fournie et certains mécanismes encore trop complexes. Il m'a fallu chercher des informations sur des forums, serveurs discord, etc. Au départ, j'ai eu du mal à paramétrer le compilateur Emscripten et à comprendre comment les données passaient entre JS et WASM.

-> La version de Three.js utilisée dans l'application depuis sa création est la R71. Pour information, la dernière version de cette librairie est la R117 (juin 2020, donc 46 versions de retard). Ainsi, le problème rencontré fut que la documentation de la R71 n'était plus disponible en ligne. Certaines fonctions changent de nom, certaines disparaissent et d'autres s'ajoutent dans le lot. De ce fait, migrer l'application vers la dernière version prendrait plusieurs semaines à se faire. J'ai donc discuté avec les créateurs de Three.js et ils m'ont garanti que cela n'aura pas d'impacts sur l'application sur le long terme. Les anciennes versions fonctionnent aussi longtemps que la spécification WebGL reste supportée par les navigateurs. Il fallait que je m'adapte à cette contrainte en allant me documenter directement dans le code source de la version R71.

<u>Formation personnelle</u> :

-> Au début du stage, je n'avais que quelques notions en JavaScript et en C++, j'ai dû m'auto former sur ces langages afin d'avoir un niveau correct. J'ai donc suivi de mon côté des petits MOOCs, des tutoriels et des vidéos afin de redresser mon niveau.

-> J'avais abordé certaines notions comme l'octree et manipulé la librairie Three.js durant ma formation à l'IUT, cependant, leurs implémentations dans cette application sont d'un tout autre niveau. La relecture active de la documentation et des rapports laissés par les anciens stagiaires m'a aidé dans mon avancée.

Organisation :

-> Comme dit précédemment, je ne me suis penché sur la partie serveur que vers la fin du stage. J'ai effectué certaines recherches sur l'optimisation d'un système de fichiers en octree mais je n'ai pas implémenté de nouvelles solutions. Je me suis contenté d'améliorer ce qui existait déjà car la partie client m'a pris plus de temps que prévu. En effet, juste pour comprendre seul les rouages de la partie client, cela m'a pris 2 semaines au début du stage. Heureusement, mon tuteur était présent en visioconférence une fois par semaine pour m'accompagner. Le télétravail fut difficile les premières semaines mais je me suis adapté assez rapidement.

4.2 Validation des objectifs

Pour ma part, le bilan du stage est satisfaisant. J'ai pu atteindre la plupart de mes objectifs :

- prouver qu'il est possible de réduire le développement JavaScript en utilisant d'autres langages comme le C++.
- améliorer les performances de l'application
- améliorer l'interface
- l'application est déployable

4.3 Propositions de développement futur

- Améliorer la partie WebAssembly, voire même utiliser cette technologie côté serveur.

- Faire en sorte que Jasmine puisse directement faire des requêtes vers les services du CDS (VizieR, Aladin, etc.) comme le fait TOPCAT[5].

- Plus ambitieux, étudier la possibilité d'utiliser le WebGPU, un nouveau standard bien plus performant que le WebGL selon certains benchmarks[6], pour optimiser l'utilisation des shaders en ayant un accès plus direct aux ressources du GPU.

[5] Topcat : http://www.star.bris.ac.uk/~mbt/topcat/
[6] WebGPU : https://alain.xyz/blog/raw-webgpu ou
https://developers.google.com/web/updates/2019/08/get-started-with-gpu-compute-on-the-web

Conclusion

En conclusion, ce stage aura été une expérience très enrichissante. Il m'a permis de consolider mes compétences dans le domaine de l'Imagerie Numérique 3D et de confirmer mon intérêt pour les nouvelles technologies. De plus, l'aspect Recherche m'a laissé plus de liberté de réflexion au niveau des choix techniques. De ce fait, j'ai pu développer ma capacité à m'autoformer tout au long du stage.

Je suis satisfait des résultats obtenus mais cela n'aurait été possible sans l'accompagnement efficace de mon tuteur de stage. De nombreux ajouts de fonctionnalités ont été effectués et je pense que l'application a encore beaucoup de potentiel surtout pour la vulgarisation astronomique à destination du grand public. Il est probable que certains de mes choix techniques sont à revoir mais j'ai essayé de faire de mon mieux avec les compétences dont je disposais à ce moment là.

J'éprouve toutefois le regret de n'avoir pas pu passer tout ou partie du stage en présentiel au CDS mais je passerai surement rendre visite un de ces jours.
Ce que j'ai appris durant ce stage me servira surement pour la poursuite de mes études et ma vie professionnelle.

Glossaire

Three.js :

Bibliothèque JavaScript pour créer des scènes 3D en WebGL dans un navigateur web sans plugin.

GPU :

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

Octree :

Un octree est une structure de données de type arbre dans laquelle chaque nœud peut compter jusqu'à huit enfants. Les octrees sont le plus souvent utilisés pour partitionner un espace tridimensionnel en le subdivisant récursivement en huit octants.

Shader :

Programme informatique, écrit avec une syntaxe assembleur ou dans un langage de programmation de plus haut niveau (ex : GLSL), directement exécutable par la carte graphique et remplaçant certaines parties du pipeline habituel d'exécution.

File System :

Méthode de stockage des informations et d'organisation de fichiers, localisés par des chemins d'accès.

WebAssembly :

WebAssembly, abrégé wasm, est un standard du World Wide Web pour le développement d'applications. Il est conçu pour compléter JavaScript avec des performances supérieures. Le standard consiste en un bytecode, sa représentation textuelle et un environnement d'exécution dans un bac à sable compatible avec JavaScript. Il peut être exécuté dans un navigateur Web et en dehors.

Sitographie

Mémo Javascript :
https://javascript.info/

Tutoriel C++ :
https://openclassrooms.com/fr/courses/1894236-programmez-avec-le-langage-c

Potree, un exemple d'implémentation de l'Octree :
https://github.com/potree/potree/

Documentation officielle Emscripten :
https://emscripten.org/docs/getting_started/Tutorial.html#generating-html

Documentation officielle Three.js :
https://threejs.org/docs/

Tutoriel suivi pour le WebAssembly :
https://marcoselvatici.github.io/WASM_tutorial/

WebGPU :
https://gpuweb.github.io/gpuweb/

Site du CDS :
http://cdsweb.u-strasbg.fr/about