Jérôme Desroziers
Promotion 2016

TELECOM Nancy deuxième année (2017-2018)

Visualisation 3D de données astronomiques dans un navigateur Web

Observatoire Astronomique de Strasbourg
11 Rue de l'Université
67000 Strasbourg

Responsable en entreprise : André SCHAAFF
Responsable enseignant : Hervé PANETTO

Résumé

J'ai réalisé ce stage à l'Observatoire Astronomique de Strasbourg dans le cadre de la validation de ma seconde année à Télécom Nancy. L'objectif de ce dernier était de finaliser le développement d'une application, utilisant le Javascript, permettant la visualisation en 3D de données issues de simulations astronomiques sur un navigateur web. Les problématiques sont multiples, notamment en ce qui concerne la taille des données manipulées, qui peuvent atteindre plusieurs Téraoctets et qui dans ce cas ne peuvent pas être chargées localement dans le navigateur.

Mots clés : stage, Big Data, Javascript, 3D, web, astronomie

Absract

I did this internship in the Astronomical Observatory of Strasbourg in order to validate my second year of studies at Telecom Nancy. Its purpose was to finalize the development of a web application, utilizing Javascript, allowing the 3D visualization of data coming from astronomical simulations on a web browser. Multiple problems arise, notably when the size of some simulations is taken into account, thoses weighting up to more than 1 Terabyte, which prevents them for being directly loaded in the browser.

Keywords : internship, Big Data, Javascript, 3D, web, astronomy

Déclaration sur l'honneur de non-plagiat

(à joindre obligatoirement au document rendu)

Je soussigné(e),

Nom, prénom :

Élève-ingénieur(e) régulièrement inscrit(e) en e année à TELECOM Nancy [1]

N° de carte d'étudiant(e) :

Année universitaire : 20 - 20

(Co-)Auteur(e) du document, mémoire, rapport ou code informatique intitulé :

Par la présente, je déclare m'être informé(e) sur les différentes formes de plagiat existantes et sur les techniques et normes de citation et référence.
Je déclare en outre que le travail rendu est un travail original, issu de ma réflexion personnelle, et qu'il a été rédigé entièrement par mes soins. J'affirme n'avoir ni contrefait, ni falsifié, ni copié tout ou partie de l'oeuvre d'autrui, en particulier texte ou code informatique, dans le but de me l'accaparer.
Je certifie donc que toutes formulations, idées, recherches, raisonnements, analyses, programmes, schémas ou autre créations, figurant dans le document et empruntés à un tiers, sont clairement signalés comme tels, selon les usages en vigueur.
Je suis conscient(e) que le fait de ne pas citer une source ou de ne pas la citer clairement et complètement est constitutif de plagiat, que le plagiat est considéré comme une faute grave au sein de l'Université, et qu'en cas de manquement aux règles en la matière, j'encourrais des poursuites non seulement devant la commission de discipline de l'établissement mais également devant les tribunaux de la République Française.

Fait à ....., le ..../...../.....

Signature :

1. **TELECOM Nancy :** Campus Aiguillettes • 193, avenue Paul Muller • CS 90172 • Villers-lès-Nancy
Tél. : +33 (0)3 83 68 26 00 • Fax. : +33 (0)3 83 68 26 09 • www.telecomnancy.eu • contact@telecomnancy.eu

Rapport de stage 2A

Visualisation 3D de données astronomiques dans un navigateur Web

Jérôme Desroziers

Année 2017-2018

Stage réalisé en 2018 à l'Observatoire Astronomique de Strasbourg dans le cadre de la validation de ma deuxième année à Télécom Nancy.

Jérôme Desroziers
18 Chemin du Grand Patis
54200, TOUL
06 48 24 88 03
jerome.desroziers.54@gmail.com

TELECOM Nancy
193 avenue Paul Muller,
CS 90172, VILLERS-LES-NANCY
03 83 68 26 00
contact@telecomnancy.eu

Observatoire Astronomique de
Strasbourg
11 rue de l'Université
7000 Strasbourg
03 68 85 24 50

Responsable en entreprise : André SCHAAFF
Responsable enseignant : Hervé PANETTO

Remerciements

Je tiens à remercier en premier lieu l'ensemble du personnel de l'Observatoire de Strasbourg pour leur accueil chaleureux, tout particulièrement André Schaaff, mon tuteur, et Sébastien Derriere, dont j'ai partagé le bureau durant les 2 mois et demi de stage et le mois de CDD qui l'a suivi.

Je tiens aussi à remercier les membres du CDS, notamment Frédéric Marin et François Xavier Pinault, pour les nombreux échanges que j'ai eu avec eux durant mon stage, ainsi que Thomas Boch et Gilles Landais.

Enfin, je remercie Pierre Alain Duc et Mark Allen, respectivement directeurs de l'Observatoire et du CDS, pour m'avoir donné l'opportunité de réaliser ce stage.

Table des matières

Introduction

L'Observatoire Astronomique de Strasbourg est une structure de recherche importante qui chaque année reçoit de nombreux stagiaires, notamment dans le domaine de l'informatique où ces derniers se voient confier des missions relevant le plus souvent de la R&D.

En 2016 et en 2017 j'ai eu l'opportunité d'y travailler dans le cadre de stages et de CDD, notamment sur le développement d'une application permettant la visualisation et la manipulation de données astronomiques en 3D, sur navigateur.

Plus particulièrement, cette application a pour ambition d'être applicable non seulement à des jeux de données chargés localement, mais aussi à des jeux de données provenant de simulations de très grande taille, de l'ordre du Téraoctet.

L'objectif de mon stage est de finaliser l'application afin qu'elle soit mise à disposition du public dans la fin de l'année. L'axe majeur sur lequel je vais devoir me concentrer est la gestion des grands jeux de données, point sur lequel j'ai commencé à travailler l'année dernière mais qui n'est pas fonctionnel au début de ce stage.

Dans ce rapport je vais dans un premier temps présenter l'Observatoire, son organisation et les services qu'il propose, puis dans un second temps exposer les problématiques associées à mon stage, son déroulement et son bilan.

Chapitre 1

Contexte du stage

1.1 Présentation de l'entreprise

1.1.1 Historique

Fondé en 1881 - après que l'Alsace-Lorraine aie été cédée à la Prusse - l'Observatoire Astronomique de Strasbourg se situe dans la partie historique de la ville et est entouré par le Jardin botanique de l'Université de Strasbourg (ci-dessous Fig. 1.1, une photographie de la grande coupole de l'Observatoire).

FIGURE 1.1 – Grande coupole de l'Observatoire

Bien que sa fonction première fut l'observation des étoiles, notamment grâce à la lunette contenue dans la grande coupole (la plus grande d'europe au moment de sa création), cette dernière n'est maintenant plus utilisée et la grande majorité des travaux actuellement réalisés à l'Observatoire on trait à l'exploitation / la diffusion de données astronomiques.

1.1.2 Structure

Vue d'ensemble

A ce jour, l'Observatoire dépend du CNRS ainsi que de l'Université de Strasbourg, et emploie 80 personnes environ (dont 11 astronomes), et son directeur depuis 2017 est Pierre Alain Duc.
Il est un acteur majeur de l'IVOA (International Virtual Observatory Alliance), un consortium dont l'objectif est de déterminer les standards utilisés en terme de création et de manipulation de données astronomiques.
Il est structuré en 2 équipes de recherche :

- L'équipe GALHECOS qui étudie la formation et l'évolution des galaxies et de notre Galaxie dans un contexte cosmologique, au travers de leurs populations stellaires, de la dynamique des étoiles et de la matière noire, et des effets de rétro-action liés notamment à l'activité de leur trou noir central. Elle s'intéresse aux sources galactiques et extragalactiques émettrices en rayons X, objets compacts (étoiles à neutron, naines blanches, etc.) et noyaux actifs de galaxies.
- Le Centre de Données astronomiques de Strasbourg (CDS), dont le directeur actuel est Mark Allen, qui consiste en un ensemble de services permettant d'accéder à un large éventail d'informations provenant d'études diverses liées à l'astronomie.

Services

Les services fournis par l'observatoire comprennent notamment trois services développés et maintenus par le CDS, en libre accès sur son site web :

- Aladin : qui est un atlas interactif du ciel, et permet de visualiser la voûte céleste grâce à des recoupements d'images provenant de diverses observations astronomiques. (logo : Fig. 1.2)

FIGURE 1.2 – Logo d'Aladin

- Simbad : qui est un service possédant une base de données centrée sur les objets astronomiques (comètes, nuages de gaz, etc...), et qui permet d'effectuer des requêtes sur cette base en se basant sur le nom, la position, ou certaines propriétés physiques de ces objets (logo : Fig. 1.3). Actuellement le nombre d'objets répertoriés avoisine 8,3 millions ; et le nombre d'identifiants qui leur est associé dépasse 23 millions.

FIGURE 1.3 – Logo de Simbad

- VizieR : qui est aussi un service axé sur une base de données, se distingue grandement de Simbad dans la mesure où il permet d'effectuer des requêtes non plus en se focalisant sur un objet précis mais en prenant comme base un ou plusieurs catalogues (comme le Sloan Digital

Sky Survey par exemple). L'avantage offert par rapport à Simbad est qu'en se focalisant sur un catalogue, toutes les données que l'on récupère sont au même format. Il faut en effet savoir que la même information, voir le même objet, peuvent porter plusieurs dizaines de noms si l'on désire couvrir l'ensemble des catalogues. A ce jour le nombre de catalogues répertoriés est légèrement inférieur à 15000. (logo : Fig. 1.4)

FIGURE 1.4 – Logo de VizieR

Stages

L'Observatoire (et plus particulièrement le CDS) accueille chaque année un certain nombre de stagiaires venant d'horizons divers (BTS, DUT, Ecoles d'Ingénieur), dont le responsable (en plus de leur responsable de stage) est André Schaaff, qui travaille au sein du CDS.
En tenant compte des différents éléments cités précédemment, ma place dans l'Observatoire en tant que stagiaire peut ainsi être résumée par l'organigramme ci-dessous (Fig. 1.5) :

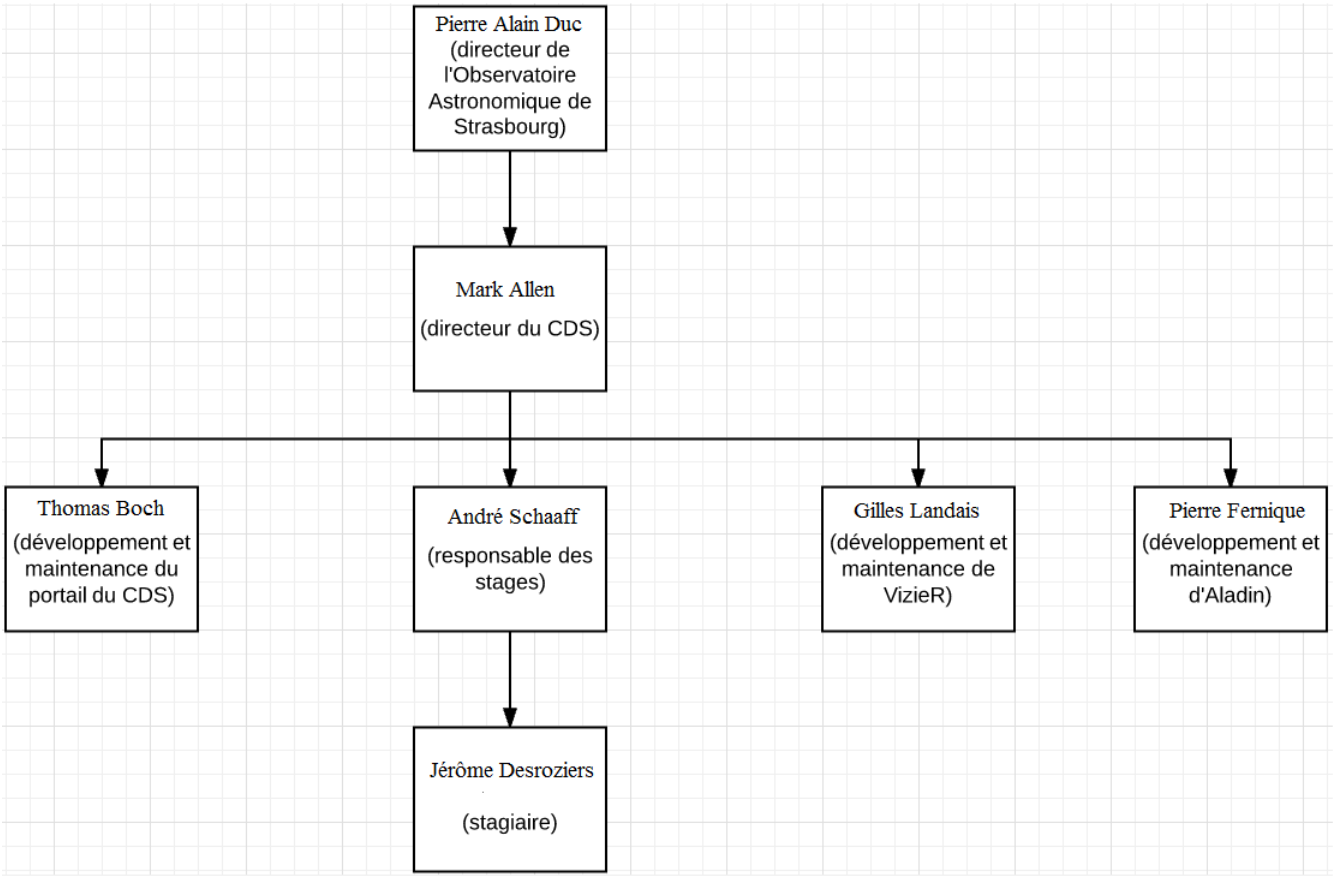


FIGURE 1.5 – Structure hiérarchique de l'Observatoire (simplifiée)

Chapitre 2

Problématique

2.1 Contexte du Stage

Dans le domaine de l'astronomie, il est très fréquent de recourir à des simulations pour tester un modèle (utilisant les lois de la relativité générale, etc...), par exemple dans le cadre de l'étude des trous noirs ou encore de la formation de galaxies.
Diverses problématiques leur sont associées, plus particulièrement l'extraction de données utilisables ainsi que leur visualisation.
Notamment, la taille des données générées pouvant rapidement dépasser l'ordre du Gigaoctet, il y a un réel besoin quant à leur visualisation ; à ce jour les outils dont disposent les astronomes reposent principalement sur des projections pour visualiser certaines zones des simulations, ou bien sont spécifiques à un type de donnée particulier.
C'est dans ce contexte que le CDS a initié en 2014 le développement d'une application ayant pour but la visualisation de données astronomiques (issues de simulations ou de missions telles que le SDSS)
Cette dernière prend la forme d'une applcation web, ce qui lui permet de ne nécéssiter aucun prérequis d'un point de vue utilisateur (ni même d'une connexion internet si le code client est possédé en local) ; qui s'appuie sur la librairie WebGL Three.js pour réaliser l'essentiel des calculs liés à l'affichage.
J'ai déjà participé au développement de cette application en 2016 (durant mon stage de fin de DUT et le CDD qui l'a suivi) et en 2017 (durant un CDD de 2 mois effectué à l'Observatoire pendant les grandes vacances), ce qui m'a donné une bonne connaissance quand à son fonctionnement et aux problématiques qui lui sont associées.
L'application, baptisée JASMINE (Javascript AStronomical data MINEr), peut être séparée en 2 parties, l'une orienté client et l'autre orienté serveur :

— La partie client (Fig. 2.1) est celle dont le développement -au début de ce stage- est le plus abouti, elle consiste en une interface accessible depuis un navigateur web et permet de manipuler des jeux de données chargés localement, qui sont traités par l'application via des scripts préalablement créés par l'utilisateur (le seul prérequis étant que le format des données fournies doit consister en une suite d'éléments de même structure composés de différent champs entiers ou flotants).
Développée en Javascript, est dotée de nombreux outils permettant de manipuler les données,

notamment :

— Des filtres (de couleur, d'intensité lumineuse) mettant en valeur pour un jeu de donnée un champ spécifique de ses éléments selon ses bornes

— Un outil de sélection pour observer une zone précise de l'espace

— Un gestionnaire d'animations, qui permet de chaîner plusieurs snapshots issus d'une même simulation

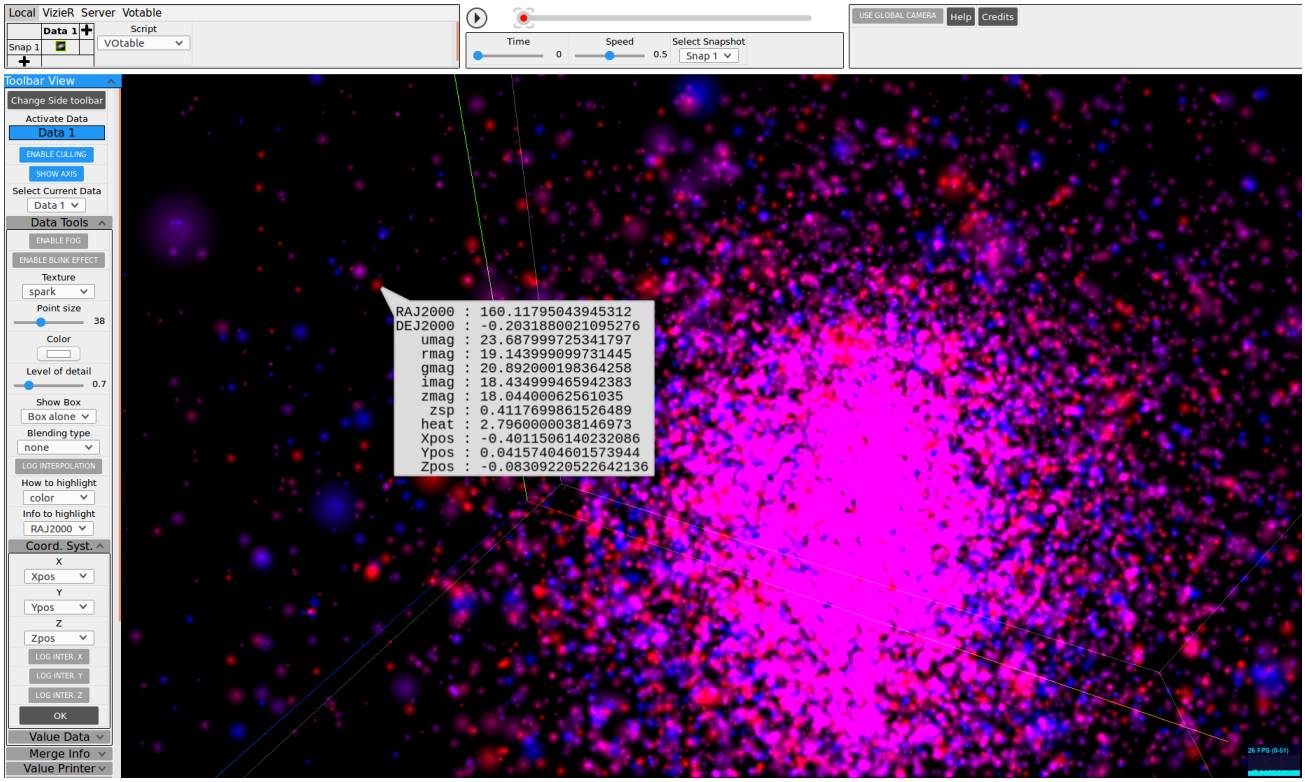


FIGURE 2.1 – Partie Client

La partie client utilise une représentation des données utilisant une structure appellée **Octree** (Fig. 2.2), très utilisée dès lors que l'on manipule des données spatiales. Un Octree est un arbre dont la représentation spatiale est un cube, dont les fils -appellés octans- correspondent chacun à un huitième du volume de leur parent, et ainsi de suite.

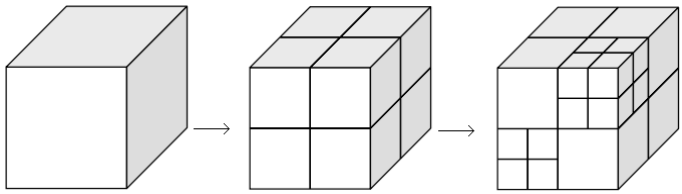


FIGURE 2.2 – Octree

Cette représentation est très utile car elle permet par exemple d'optimiser l'affichage des éléments chargés dans l'application, en ne retenant que ceux contenus dans le champ de vision de la caméra, ou encore si on veut sélectionner un élément unique pour afficher les données qui lui sont associées (via un lancer de rayon), de ne pas passer en revue l'ensemble des éléments chargés mais uniquement ceux contenus dans les octans traversés par le rayon.

— La partie serveur, elle aussi développée en Javascript et utilisant l'environnement Node.js.
Elle a pour but de permettre la visualisation de jeux de donnés de grande taille (jusqu'à l'ordre du Téraoctet), ce qui n'est pas possible avec la partie client seule du fait des limitations imposées par le navigateur (l'idée est de prétraiter les données sur le serveur et d'en extraire certaines parties à la demande du client).
C'est sur cette partie de l'application que va se concentrer l'essentiel de mon stage.

2.2 Objectifs du stage

Le but de mon stage est de finaliser le développement de l'application afin qu'elle soit mise à disposition du public (et notamment de la communauté astronomique), et plus précisément :

— Mettre en place la partie serveur de l'application, l'objectif étant qu'elle puisse traiter de gros jeux de données, dont la taille est de l'ordre du Téraoctet, avec les contraintes suivantes :
  — D'une part elle doit être capable d'en générer une base de données sur laquelle des requètes peuvent être effectuées à partir de coordonnées envoyées par le client.
  — D'autre part ces requètes doivent se réaliser en un temps raisonnable, même sur les plus grosses simulations (de l'ordre de la dizaine de secondes), afin de permettre une bonne expérience utilisateur.
  — Elle doit si possible être optimisée en utilisant le langage Web Assembly.
— Effectuer des correctifs et des améliorations sur la partie client.
— Développer un notebook Jupyter utilisant la partie client de l'application.
— Rédiger un article d'une dizaine de pages présentant l'application, et ayant vocation à être publié dans la revue Astronomy and Computing.

Chapitre 3

Réalisation et validation

Je vais maintenant présenter la façon dont les problématiques présentées ci-dessus ont été résolues.

3.1 Partie Serveur

3.1.1 Principe

La partie client de l'application n'est pas capable de charger d'importants jeux de données, notamment à cause des limitations du navigateur en terme de mémoire, mais aussi en terme de performances (le nombre de frames par seconde baissant significativement dès qu'on essaie de représenter plus de 10 milions de points à l'écran).
L'idée ici est donc, pour des simulations de grande taille, de charger les données associées sur un serveur, et de les convertir en un format supportant les requêtes (avec un temps de traitement relativement rapide), permettant un ciblage de zones spécifiques de la simulation.
Idéalement les résultats de ces requêtes doivent pouvoir prendre 2 formes : une normale qui permet d'extraire tout les points correspondant à la zone ciblée, et une forme dite **dégradée** qui doit seulement renvoyer une représentation globale des données, et qui doit être utilisée lorsque la zone ciblée est trop importante pour être renvoyée avec la première forme, par exemple si l'utilisateur veut afficher la simulation complète du côté client de l'application.

3.1.2 Problèmes

De nombreux problèmes se posent quant à la mise en place d'une telle architecture :

1. Si les requêtes sont faites de telle manière qu'elles ciblent une région précise de la simulation, alors le système de fichier généré par l'application doit être une version ordonnée dans l'espace de cette dernière.
2. Le programme doit aussi être paramétrable dans le sens où il doit permettre la génération d'un système de fichier à partir de n'importe quel type de simulation, à la condition que ses

éléments de base possèdent au moins 3 champs qui peuvent être utilisés afin de les ordonner dans l'espace.

3. Pour les simulations les plus lourdes (par exemple celles dont l'ensemble des fichiers pèse plus de 10Go), générer le système de fichiers associé ne peut pas être fait en une seule fois, à cause des limitations en terme de RAM de la machine utilisée. Il doit donc supporter des mises à jour de manière à ce que les données qu'il contient soient toujours ordonnées, sans avoir pour autant à le réécrire entièrement à chaque fois (sinon cela prend trop de temps).

3.1.3 Solution

Je vais maintenant aborder les choix qui ont été fait afin de résoudre les problèmes présentés ci-dessus :

1. **Représentation des données**
   Le choix a été fait de donner au système de fichiers la forme d'un Octree, ce qui va notamment permettre d'effectuer une sélection sur les données en se basant sur un code calculé grâce à ses octants.
   Une solution intuitive consisterait à créer un système de fichiers dont les noeuds seraient représentés par des répertoires et les feuilles par des fichiers, mais cette implémentation ne peut pas s'appliquer ici, notamment à cause du fait qu'une représentation en Octree d'une simulation contenant un nombre important d'éléments (de l'ordre du milliard) aura incidemment un nombre de feuilles lui aussi important. Assimiler ces feuilles à des fichiers rend alors toute mise à jour ou requête effectuée sur le système de fichiers extrêmement couteuse en terme de temps, du fait du nombre important d'ouvertures/fermetures qui en découle.
   La solution choisie consiste donc en une forme hybride de la solution intuitive : le haut de l'arborescence consiste en une suite de répertoires et de fichiers, ces derniers ayant une taille limite arbitraire de 200Mo. L'arborescence continue dans chaque fichier, et prend alors la forme d'une arborescence binaire où les noeuds et les feuilles sont représentés par une succession de champs de taille fixe (les feuilles binaires étant ici les feuilles réelles de l'Octree, et contiennent les points de la simulation).
   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.
   A chaque octan (et donc à chaque zone de l'espace) on peut donc associer un code calculé récursivement à partir de sa position par rapport à ses parents (Fig. 3.1), les 8 fils de la racine de l'arbre étant associées aux codes 000, ..., 111, et leurs propres fils aux codes 0000000, 0000001, ..., 111111, etc...

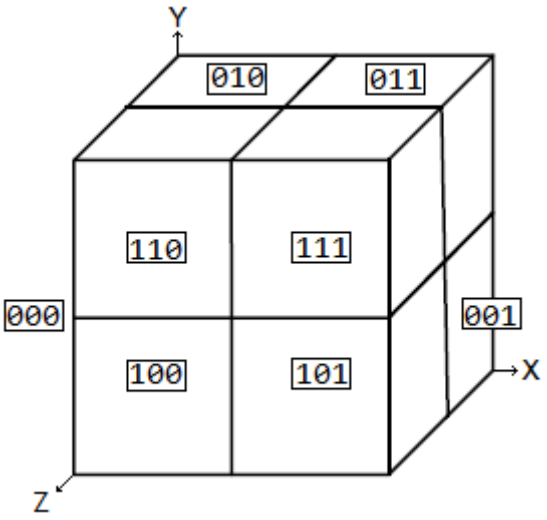


FIGURE 3.1 – Codes de l'Octree

Un noeud binaire (Fig. 3.2) est composé de 33 octets ; un champ de typage (noeud ou feuille) de 1 octet, et 8 champs d'adressage de 4 octets chacun pointant sur les fils du noeud, tandis qu'une feuille binaire (Fig. 3.3) consiste (dans une forme simplifiée) en un entête contenant 1 champ de typage de 1 octet, 1 champ de 2 octets représentant le nombre d'éléments contenus dans la feuille, suivi par une liste d'éléments, chacun consistant en une succession de champs.

	Code 000	.....	Code 111
Node Flag	Addr Son 1	.....	Addr Son 8
1 byte	8 * 4 bytes		

FIGURE 3.2 – Noeud binaire

Leaf Flag	Nb Elts N	Elt 1	.....	Elt N
1 byte	2 bytes	Elt Size * N bytes		

FIGURE 3.3 – Feuille binaire (simplifiée)

2. **Paramétrage du système de fichiers**
   Il a été décidé de stocker des informations générales sur le format de données de la simulation courante dans 2 structures :
   — La première contenant les informations générales telles que le nombre maximum d'éléments pouvant être contenus dans une feuille ainsi que les bornes de la simulation.
   — La seconde décrivant la structure d'un élément (l'ordre des champs, leur taille en octet, etc...)
3. **Création du système de fichiers**

Prérequis

Pour chaque simulation un reader spécifique doit être créé par l'utilisateur, de telle sorte qu'il effectue un stream sur chaque fichier qui lui est présenté et en extrait les éléments par paquets

devant peser 200Mo au maximum. Ces derniers sont alors ajoutés de façon séquentielle au système de fichier.

Traitement des données

Je vais maintenant présenter la façon dont a été implémentée la création du système de fichiers, qui peut être découpée en 2 phases distinctes : la phase de dite de **division** et la phase dite de **sauvegarde**.
Dans un premier temps penchons-nous sur le cas le plus simple, lorsque le système de fichiers est vide et que l'on reçoit le premier paquet de max. 200Mo d'éléments. A la fin de cette opération, le système de fichiers consistera en un seul fichier binaire d'une taille de max. 200Mo.

(a) **Initialisation**
Le programme initialise (en mémoire) un Octree vide dont les dimensions sont basées sur celles de la simulation.

(b) **Phase de division** Les points du paquet sont triés récursivement dans l'Octree, en divisant ce dernier jusqu'à ce qu'aucune de ses feuilles ne contienne plus d'éléments que le nombre spécifié dans la structure de configuration globale.

(c) **Phase de sauvegarde**
Une fois l'Octree rempli, son contenu est écrit sous forme binaire dans un buffer, qui contient l'ensemble de ses noeuds, feuilles et points, de façon totallement ordonnée (en ordre préfixe). Ce buffer est ensuite sauvegardé.

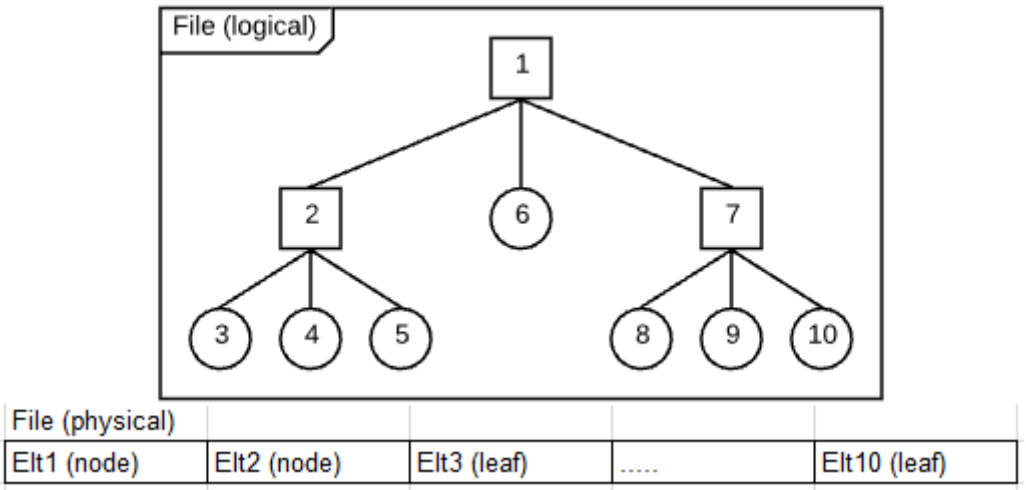


FIGURE 3.4 – Fichier ordonné

On peut observer ci-dessus (Fig. 3.4) un exemple d'un tel fichier, qui contient 3 noeuds et 7 feuilles (le nombre de fils par noeud a été réduit à 3 pour plus de simplicité). On peut voir dans sa représentation physique que ses éléments ont bien été écrits en ordre préfixe.

La cas général n'est pas aussi intuitif : le programme reçoit un paquet de points (de max. 200Mo) et doit l'ajouter au système de fichiers pré-existant.
Comme précédemment, il faut créer un Octree donnant une représentation spatiale des points

reçus, mais cette fois-ci le programme doit se baser sur le système de fichiers pour forcer l'Octree à imiter sa structure.
La phase de division est maintenant plus complexe, et est séparée en 2 sous phases, une phase globale qui consiste à parcourir l'arborescence physique du système de fichiers, et de nombreuses phases (qui seront appellées phases de division binaire), dont le but sera de parcourir chaque fichier binaire atteint (qui sera chargé dans un buffer).
A chaque phase de division binaire correspond une phase de sauvegarde :

(a) **Phase de division binaire**

Le fichier correspondant à la feuille courante de l'arborescence physique est chargée dans un buffer de 400Mo, de façon à éviter tout problème de débordement mémoire (comme la feuille à mettre à jour pèse 200Mo, dans l'éventualité où tout les points reçus se situent dans la zone précise associée à la feuille chargée, l'ensemble des nouveaux points rentrent dans le buffer).

Lorsqu'un fichier est chargé en mémoire un index dit des espaces libres est initialisé, dont les éléments prennent la forme suivante : {adresse, taille (en octets)}, auquel on ajoute la partie inutilisée en fin de buffer. L'Octree et la partie du buffer contenant les données sont ensuite parcourues simultanément.

Si il s'avère qu'une feuille de l'Octree correspond à un noeud du buffer, cette dernière sera divisée 'de force' (d'où le nom de la phase), jusqu'au point où à chaque feuille de l'Octree corresponde une feuille du buffer, les points de cette dernière étant ajoutés à la feuille de l'Octree. A ce stade, si une feuille appartenant à l'Octree dépasse la limite du nombre de points autorisée, elle est divisée, de façon 'naturelle' cette fois-ci.

En parallèle, l'emplacement des feuilles du buffer dont les points ont été ajoutés à l'Octree est ajouté à l'index des espaces libres.

A la fin de la phase de division binaire, l'Octree donne une représentation de toutes les modifications qui doivent être appliquées pour mettre à jour le système de fichiers, et l'index des espaces libres indique toutes les zones du buffer sur lesquelles il est possible de réécrire des données avant d'utiliser la partie inutilisée en fin de buffer.

(b) **Phase de sauvegarde**

Le buffer est mis à jour en comparant ses éléments à leur équivalent dans l'Octree, et en écrivant ces derniers à l'aide de l'index des espaces libres.

Au moment de sauvegarder une feuille, il se peut très bien qu'aucun des segments mémoire situés dans l'index des espaces libres n'aie une taille suffisante pour la contenir, et la feuille est alors scindée sur autant de segments que nécéssaire (Fig. 3.5), la fin de chaque segment contenant l'adresse du suivant.

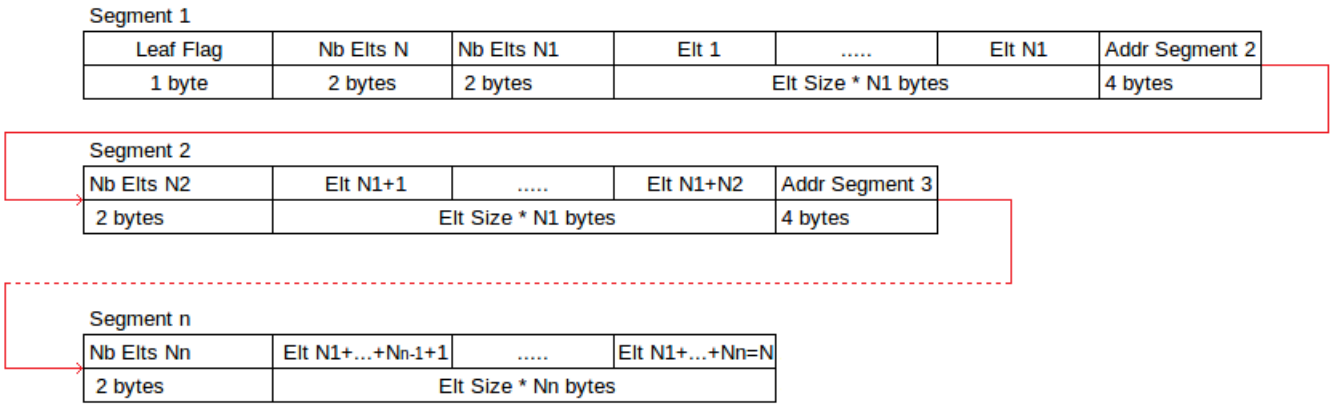


FIGURE 3.5 – Feuille (sur plusieurs segments)

A la fin de la phase de sauvegarde, le buffer est simplement sauvegardé (i.e. on écrase le fichier d'origine) si l'espace libre qui le termine n'a pas une taille inférieure à 200Mo. Dans le cas contraire le fichier correspondant est supprimé et est remplacé par un répertoire tandis que 8 sous-buffers sont extraits du noeud racine du buffer (le contenu de ces derniers sera alors complètement ordonné), et ce processus est répété jusqu'à ce que chaque buffer soit sauvegardé dans un fichier de maximum 200Mo.

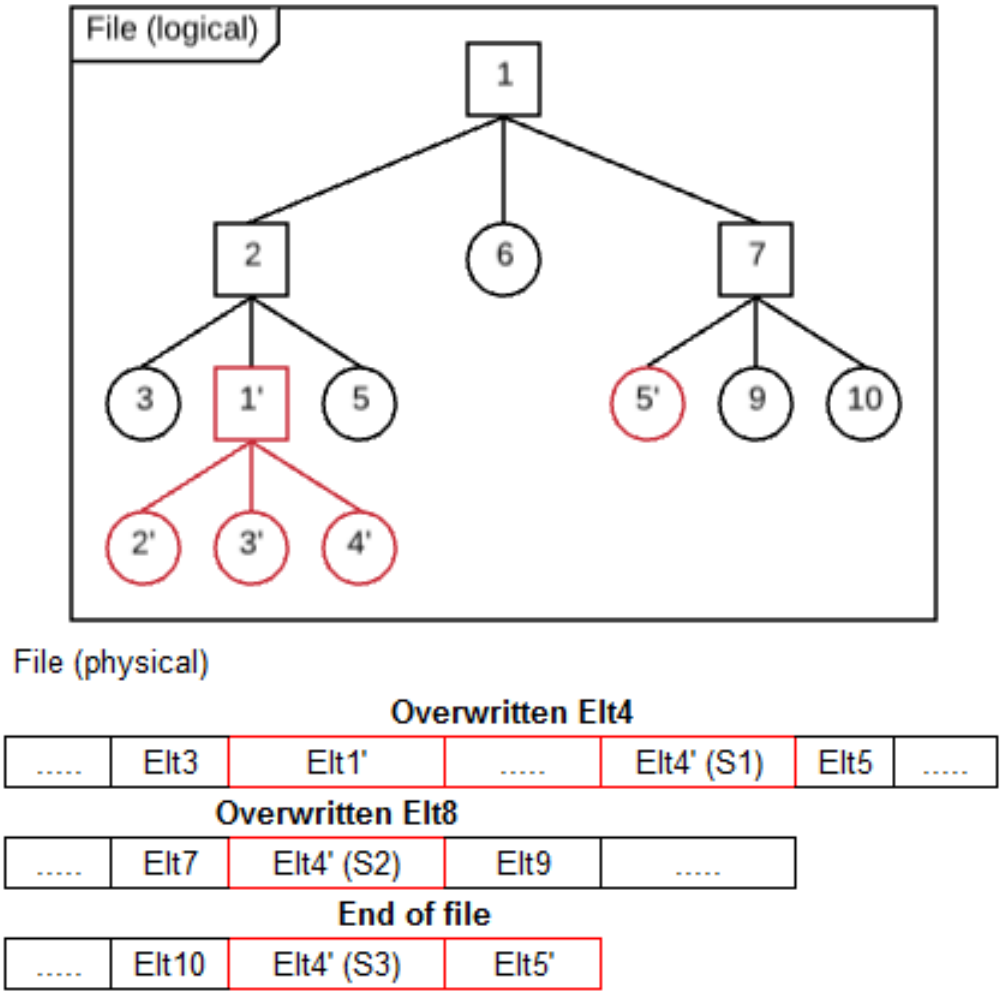


FIGURE 3.6 – Fichier après mise à jour

La Fig. 3.6 montre l'état dans lequel le fichier de la Fig. 3.4 devrait être après être passé au travers de ce processus.
Dans cet exemple les points reçus ont été ajoutés aux feuilles 4 et 8, et la feuille 4 a dû être subdivisée car son nombre d'éléments dépassait le seuil autorisé.
On suppose ici que la majorité des nouveaux points se situent dans la feuille 4', à un tel point que lorsqu'on regarde la représentation physique du fichier, on voit que l'espace qui était associé à la feuille 4 n'a pas été suffisant pour que la feuille 4' puisse y être entièrement écrite (ses autres segments étant localisés à l'emplacement où se situait la feuille 8, ainsi qu'en fin de fichier).
Les nouveaux éléments étant écrits en ordre préfixe, la nouvelle feuille 8 (la feuille 5') est la dernière à être ajoutée au fichier, i.e. sa localisation physique correspond à la fin du fichier, après le dernier segment de la feuille 4'.

3.1.4 Vue dégradée

Le rôle de la vue dégradée est d'offrir une représentation du système de fichier principal, et de reproduire sa structure sans toutefois peser autant, pour permettre l'observation des données de la simulation sur une échelle beaucoup plus étendue (i.e. la totalité de la simulation).
Là encore une idée intuitive (qui a été ma première implémentation des données dégradées) serait de directement stocker ces données dégradées dans les noeuds et les feuilles du système de fichiers principal ; par exemple en rajoutant des champs moyenne au début de chaque élément qui correspondraient à l'ensemble des données situées en dessous de ces derniers.
Mais le coût en terme d'ouvertures/fermetures de fichiers rend une telle implémentation impossible sur des jeux de données importants (comme les données seraient contenue dans les feuilles de l'arbre, charger une version dégradée de ce dernier reviendrait à devoir lire toutes ses feuilles pour en extraire l'information contenue dans les noeuds binaires).
L'implémentation choisie consiste donc, une fois le système de fichiers principal créé, à créer un second arbre qui va copier la structure du premier, à la différence que chacun de ses éléments va contenir des informations sur l'ensemble des éléments situés en dessous de lui dans la version non-dégradée de l'arbre (cf Figs 3.7 et 3.8).
Ces derniers ont la structure suivante :

— 1 champ poids, qui indique le nombre de fils réels de l'élément courant.
— N champs moyenne, N étant le nombre de champs que l'on trouve dans un élément donné de la simulation (champs x,y,z exclus), qui représentent chacun les moyennes pondérées des éléments fils réels de l'élément courant.

Node Flag	Weight	Avg Field 1	.....	Avg Field N	Addr Son 1	.....	Addr Son 8
1 byte	4 bytes	Size of Avg Field 1	.....	Size of Avg Field N	8 * 4 bytes		

FIGURE 3.7 – Noeud dégradé

Leaf Flag	Weight	Avg Field 1	.....	Avg Field N
1 byte	2 bytes	Size of Avg Field 1	.....	Size of Avg Field N

FIGURE 3.8 – Feuille dégradée

Notamment, les feuilles binaires de la représentation dégradée du système de fichiers se retrouvent réduites à 1 élément, avec 1 champ poids correspondant au nombre d'éléments dans la feuille originale et N champs moyenne correspondant à la moyenne des champs de ces derniers. Hormi cela, le nouveau système de fichiers dispose de la même structure que le premier, avec une partie physique et une autre binaire, et des fichiers de 200Mo pour faire la jonction entre les 2. La différence notable est que la partie physique de l'arborescence est beaucoup plus petite que celle de son équivalent non-dégradé, à cause de la taille des feuilles binaires qui s'est retrouvée réduite à 1 point (ce qui a pour conséquence le fait que les feuilles de l'arborescences physique -qui pèsent jusqu'à 200Mo- peuvent contenir des arbres d'une profondeur nettement plus importante que le système de fichier non-dégradé avant de subir une division).
Les champs moyenne de chaque élément dégradé sont générés en réalisant un parcours postfixe du système de fichiers non-dégradé.
Une **vue** dégradée de la simulation est alors définie par son niveau N et son origine O, et consiste en l'aggrégat de tout les points dégradés situés au niveau de profondeur N par rapport à l'origine O dans le système de fichiers dégradé. Par exemple une vue dégradée représentant l'ensemble de la simulation à un niveau 7 est définie par le noeud racine et l'ensemble des noeuds et feuilles situés 7 niveaux plus bas (ou moins si l'arbre s'arrête avant)
On peut aussi noter que les points dégradés ne possèdent pas de champs xyz, qui sont générés à la volée lors de la résolution des requêtes et qui correspondent au milieu de l'octant qui leur est associé.
L'algorithme en lui-même a été assez difficile à réaliser dans la mesure où l'arbre dégradé est construit à partir du premier, mais que la gestion de la mémoire (le fichier courant des données dégradées doit être scindé en 8 si son poids dépasse 200Mo) est indépendant de la position dans le buffer dégradé courant.
Par exemple lors de la génération de l'arbre principal, lorsqu'on mettait à jour une feuille physique, ce n'est qu'après lui avoir ajouté tout les éléments lui étant associés (donc après avoir parcouru entièrement son arborescence) qu'éventuellement on la scindait en 8 si son poids excédait 200Mo. Dans la génération de l'arbre dégradé on vérifie la taille du fichier courant à chaque fin de lecture d'un fichier issu de son équivalent non-dégradé, et très souvent il faut effectuer une division alors que l'élément dégradé pointé (dont l'équivalent non-dégradé est le fichier que l'on vient de lire) n'est pas la racine du fichier.
Il faut représenter la position de ce dernier dans une pile, qui après division d'une feuille physique du système de fichiers dégradé permet de retrouver le noeud binaire représentant la position de l'élément courant dans le parcours de l'arbre réel. De la même manière lorsqu'une telle division est réalisée, il faut prendre en compte les noeuds binaires situés après l'élément courant (en ordre postfixe) dont le contenu n'a pas encore été défini.
Afin de mieux illustrer cela, la Fig. 3.9 représente l'arborescence physique d'un système de fichier à partir duquel une représentation dégradée est en train d'être créee (là encore chaque noeud possède 3 fils pour plus de simplicité).
Le tracé en rouge représente les éléments de l'arbre qui ont déjà été parcourus.

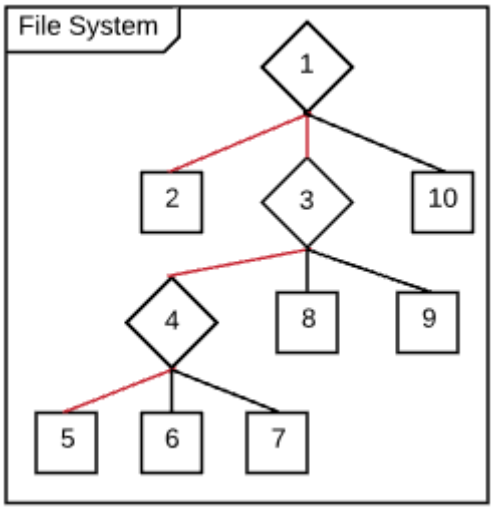


FIGURE 3.9 – Système de fichiers simplifié

Cette représentation dégradée (Fig. 3.10) n'est constitué que d'1 fichier dont les noeuds binaires 2 et 14 correspondent à l'arborescence contenue dans les fichiers 2 et 5 du système de fichier. On voit notamment que les fils des éléments d'indice 1, 12 et 13 n'ont pas tous été écrits, ce qui est normal compte tenu du fait que leurs équivalents non-dégradés n'ont pas encore été rencontrés (en ordre postfixe).

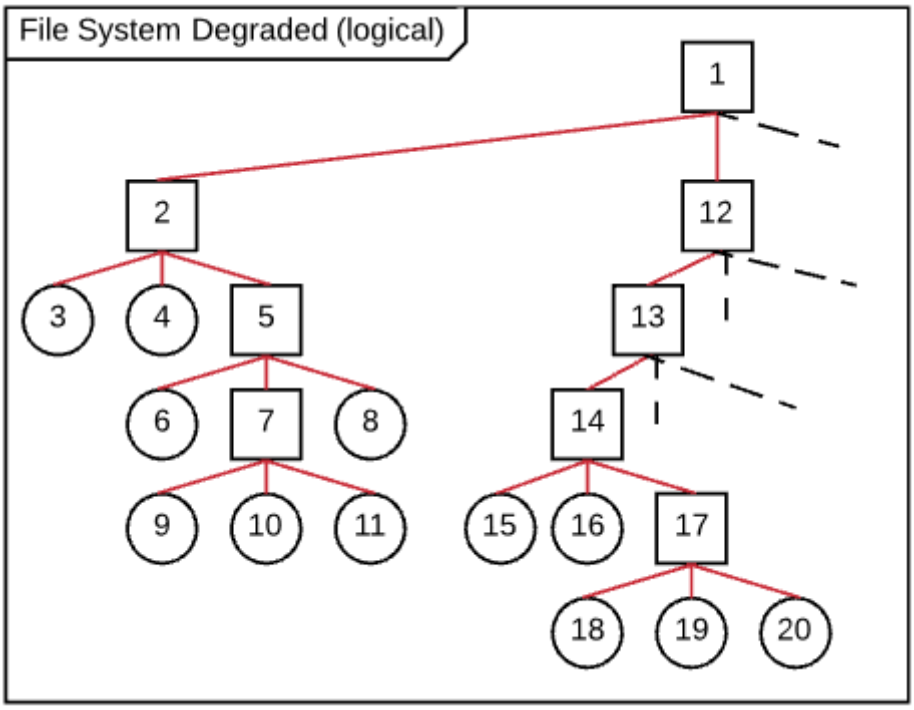


FIGURE 3.10 – Équivalent dégradé (pré-division)

Si on suppose qu'à la fin de la lecture du fichier 5 (non-dégradé) le fichier dégradé pèse plus de 200Mo, alors ce dernier équivaudra à Fig. 3.11 après division.
Sur cette dernière figure on voit clairement que l'écriture de l'arbre dégradé s'est arrêtée au noeud 14 du second fichier, et il est alors nécéssaire d'ouvrir ce dernier et le charger dans un buffer, en repointant sur l'élément 14, afin que la suite du parcours du système de fichier et de

sa version dégradée continuent de coïncider (le tracé en rouge représente ici le chemin qui a été stocké en pile avant la division).

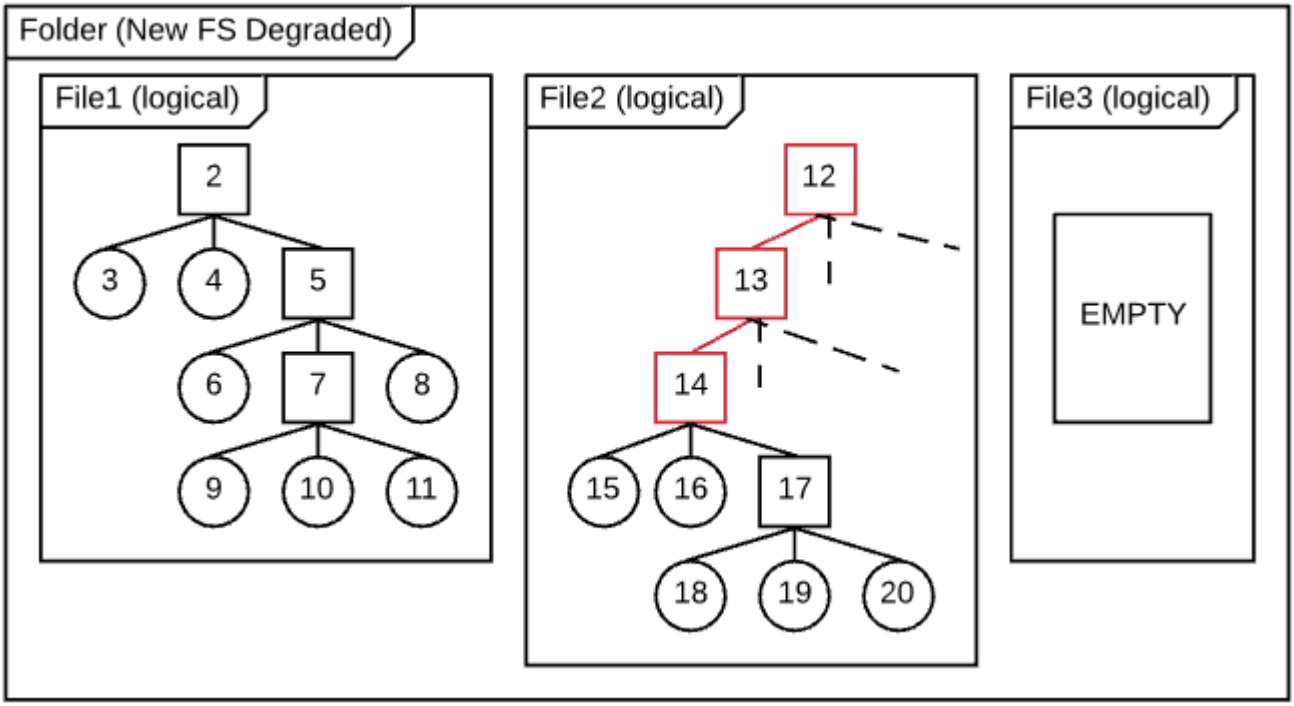


FIGURE 3.11 – Équivalent dégradé (post-division)

3.1.5 Gestion côté client

Je vais maintenant présenter la manière dont s'effectuent les interractions avec le serveur depuis le client.

Outil de zoom

Lors de mon stage en DUT (effectué en 2016), j'avais mis en place un ouil (Fig. 3.12) permettant, sur un jeu de données chargé localement, de sélectionner une zone d'intérêt via un cube de sélection dont la taille et la position peuvent être modifiés par l'utilisateur via un système de glisser-déposer.
Le principe est le suivant : lors que le mode de zoom est activé, l'écran principal de l'application est scindé en 2 vues, l'une dite **maître** (à droite sur la Fig. 3.12) et l'autre dite **esclave** : la vue maître contient les données globales de l'application (notamment les données chargées avant l'activation du mode de zoom) ainsi que l'outil de sélection, tandis que la vue esclave contient uniquement les points situés au sein de ce dernier.
La position de ces points est bien sûr normalisée, ce qui permet d'observer plus en détail les zones denses de certaines simulations, et il est possible leur appliquer des transformations (filtres, changement de système de coordonnées) sans altérer les données issues de la vue maître (sur lesquelles par exemple d'autre filtres peuvent être appliqués, etc...).

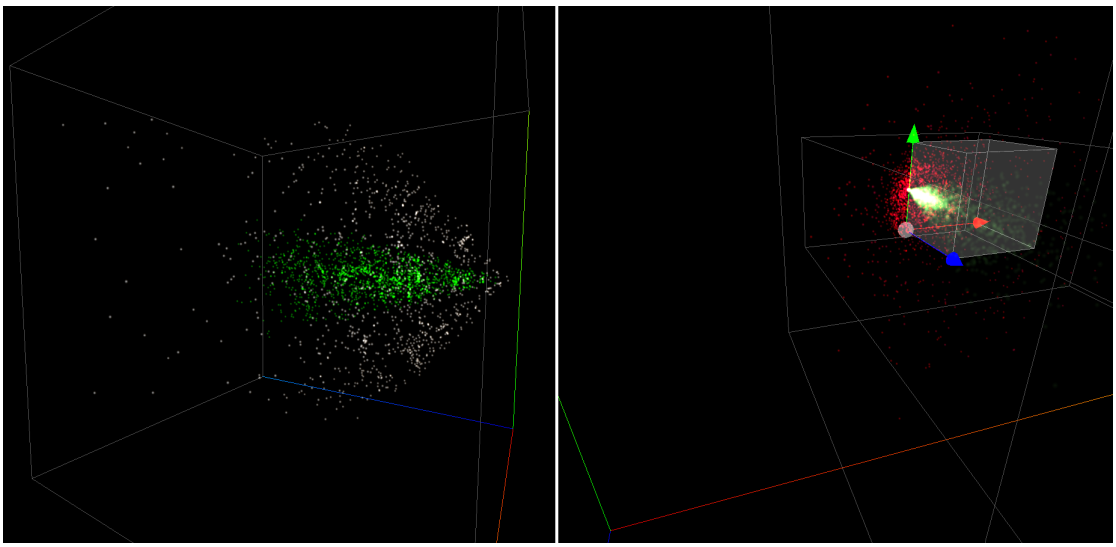


FIGURE 3.12 – Outil de zoom

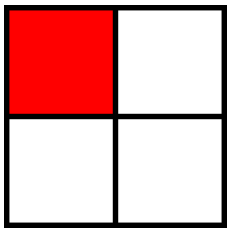
J'ai décidé d'utiliser cet outil pour gérer la sélection et l'affichage des données issues du serveur

Communication avec le serveur

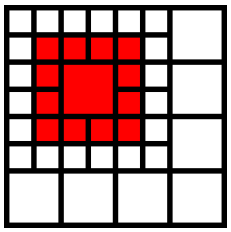
Les échanges entre le client et le serveur peuvent se séparer en 2 étapes :

(a) Les données dégradées correspondant à la simulation complète sont chargées dans l'application, la position des points dégradés ainsi que leurs valeurs moyennes permettant à l'utilisateur (via notamment l'utilisation de filtres) de sélectionner des régions d'intérêt avec l'outil de zoom.

(b) Comme les données dégradées affichées dans la partie client de l'application ont leurs positions normalisées à celles d'un cube de côté de longeur 1 (que l'on va assimuler à un Octree), il est facile, depuis un point ciblé de l'espace, de déterminer le code du noeud correspondant dans la partie serveur de l'application, pour une profondeur donnée.
Lorsque le mode de zoom est utilisé sur des données issues du serveur, ses déplacements sont faits de telle sorte qu'ils correspondent à des pas d'une longeur égale à l'inverse d'une puissance de 2. De cette manière si la zone ciblée par l'outil de sélection ne correspond pas à 1 noeud de l'Octree (mais par exemple à l'intersection de 8 noeuds), il est possible, via une fonction récursive, de calculer l'ensemble des codes correspondant à la zone ciblée.
Les Figs. 3.13a etd 3.13b illustrent ces situations, l'Octree étant ici remplacé par un Quadtree (son équivalent 2D) pour plus de simplicité.



(a) Requête à code unique



(b) Requête à codes multiples (13)

FIGURE 3.13 – Génération des codes côté client

Ces codes sont ensuites concaténés en une unique chaîne qui constitue la requête qui sera envoyée au serveur. Ce dernier renvoie les points correspondant aux positions ciblées, qui sont alors affichés exactement de la même manière que si ils résultaient d'une sélection sur des données locales, c'est à dire dans la vue esclave du mode de zoom.
Il se peut aussi que les codes envoyés par la partie client soient plus profonds que les données présentes sur le serveur ; il faut donc faire attention à ce que le serveur élimine les codes redondants afin de ne pas envoyer des duplicatas d'une même zone.

Un problème persiste : si l'ensemble des codes correspondant à la zone dégradée sélectionnée représente une quantitée trop importante de points à envoyer depuis le serveur, ces derniers prendront dans le meilleur des cas trop de temps à être collectés, et dans le pire des cas feront échouer la requête à cause d'un dépassement de RAM.
La solution consiste pour le serveur à tout d'abord utiliser le système de fichiers dégradé pour calculer, via le champ poids de ses éléments le nombre d'éléments réels associé à la requête (et leur poids total). Si il apparaît que les données brutes correspondantes sont trop volumineuses pour être traitées par le serveur, c'est leur version dégradée qui est envoyée à la place.
Dans l'éventualité où ce cas de figure se produit, il est possible d'écraser les données de la vue maître par celles de la vue esclave, de façon à récursivement effectuer des zooms sur la simulation.

3.2 Partie Client

Plusieurs améliorations ont été apportées à la partie client de l'application, la principale étant une refonte de la représentation interne des données chargées.
Originellement, l'application distinguait le type des données affichées selon qu'elles provenaient d'une simulation, de relevés réels, etc...
Au fur et à mesure que de nouvelles fonctionnalités ont été ajoutées à l'application ce typage des données rendait la maintenance de la partie client de plus en plus difficile, chaque type de donnée étant stockée à des niveaux différents par rapport à l'imbrication des classes constituant l'application.
Une des fonctionnalités ajoutée tardivement consistait en la possibilité de changer le système de coordonnées d'un jeu de données affiché. Et seules les données dont le type représentait des données 'réelles' géraient un redimentionnement automatique en fonction des autres jeux chargés. Par exemple si 2 jeux de données étaient affichés simultanément et que la même longeur d'onde était sélectionnée comme coordonnée des points pour l'axe y, la position des points de chaque jeu de données était recalculée de façon à avoir une représentation cohérente des données affichées.
L'objectif était d'étendue cette fonctionnalité à l'ensemble des types de l'application, ce qui aurait impliqué le fait que chaque type aurait ses propres fonctions de changement de système de coordonnées, modifiant ses propres buffers de données, etc.., avec pour conséquence une complexité accrue de l'application, avec par exemple une gestion de chaque type dans les classes associées à l'interface de changement de coordonnées, dans les classes gérant le chargement d'un nouveau fichier, etc... et aurait résulté en un antipattern plat de spaghetti.
J'ai donc décidé de supprimer ce typage des données de l'application, car l'ouvrir à la communauté en conservant son système originel de gestion de données l'aurait rendue impossible à mettre à jour sans en avoir une connaissance parfaite.

Un autre ajout concerne la mise en place d'une interfac permettant à l'utilisateur d'ajouter ses propres scripts de chargement à la volée (pour des données locales), sans avoir à modifier le code source de l'application.

3.3 Intégration dans un notebook Jupyter

3.3.1 Notebook Jupyter

Présenté simplement, un notebook Jupyter est une application web accessible depuis un serveur, dont le noyau est exécuté en Python.
Un notebook permet notamment d'écrire du code d'un langage donné (Python, Javascript, etc...) depuis une page web, ce dernier étant envoyé au serveur qui traite la requête et renvoie le résultat au client (une des application intéréssante est qu'il permet d'utiliser simultanément des librairies provenant de langages différents afin de combiner leurs résultats).
Une fonctionnalité avancée des notebook Jupyter est de permettre l'intégration de widgets Javascript.
Rendre l'application JASMINE compatible avec Jupyter offre plusieurs avantages :

— Une meilleure diffusion : l'application est disponible sous forme d'une librairie qu'il suffit d'importer pour pouvoir la tester sur son propre notebook.

— Le fait de pouvoir prédéfinir des commandes à exécuter sur le widget, ce qui permet de créer des fichiers d'exemple (Fig. 3.14) montrant les possibilités offertes par l'application, ou encore de réaliser facilement des démos lors de conférences.

FIGURE 3.14 – Widget Ipyjasmine

3.3.2 Implémentation

Afin de rendre possible l'intégration d'un widget Javascript dans un notebook, il est nécéssaire de concevoir les éléments suivants :

- Une classe Python regrouppant les différentes méthodes permettant d'interragir avec le widget depuis le notebook.
- Une classe Javascript qui va englober la librairie d'origine et qui va faire le lien avec la partie Python de l'application.

Le principal problème réside dans le fait qu'à moins d'avoir pensé l'application dès le départ pour qu'elle soit compatible avec un notebook, il y a des problèmes de compatibilité qu'il faut résoudre ; dans mon cas j'ai dû notamment :

- Gérer l'affichage fenêtré de l'application, qui se redimensionnait différemment depuis le notebook (en mode multivues seule 1 vue sur 2 était affichée)
- Mettre en place une gestion spécifique du chargement des fichiers depuis le notebook.
  En effet un fichier chargé depuis l'interface Jupyter l'est du côté Python de l'application, et son contenu est ensuite envoyé au wrapper Javascript qui le transmet finalement à la librairie. Comme Python traite différemment les fichiers selon qu'ils sont écrits en binaire ou non, ces derniers arrivent du côté Javascript dans un format différent de celui dans lequel ils auraient dû être (si ils avaient été chargés directement depuis l'application) et sont encodés sous la forme d'une chaîne en base 64 (qu'il faut recoder en binaire).
- Jupyter précompile la librairie et gère les dépendances par modules, or la partie client de l'application n'a pas été pensée être utilisée de cette manière ; la solution qui a été trouvée est de précompiler l'ensemble de ses classes dans 1 seul fichier qui est alors fourni au notebook.
  Mais même ainsi certaines librairies utilisées par l'application reconnaissent l'utilisation de modules et ne se chargent pas correctement, il a fallu englober l'ensemble des classes du fichier dans une fonction anonyme contenant un objet module 'null' pour outrepasser ce problème.

3.4 Rédaction d'un article présentant l'application

Une portion du stage a enfin été consacrée à la co-rédaction avec André d'un article présentant l'application, en anglais, et ayant vocation à être publiée dans la revue Astronomy and Computing.

Chapitre 4

Bilan

4.1 Difficultés rencontrées

4.1.1 Partie Serveur

La principale difficulté vient de la nature et de la taille des données manipulées.
En effet les simulations les plus denses étant écrites en binaire, il est difficile de déterminer d'où provient un bug lors de la création du système de fichiers, notamment dans le sens où ce dernier prend la forme d'erreurs d'écriture et n'est détecté qu'au moment où le fichier est lu.
Il peut par exemple consister en un décalage de 1 octet lors de l'écriture d'une feuille, ce qui a pour conséquence l'écrasement de l'entête de l'élément suivant et mène le plus souvent à une tentative de lecture d'un élément dont l'adresse est supérieure à la taille du buffer.
Cela devient d'autant plus problématique lorsque le bug survient à un stade avancé de la création du système de fichiers (et uniquement si la simulation est de grande taille), car il faut parfois attendre plusieurs minutes de traitement avant qu'il ne survienne.
Notamment lors de la conception de l'algorithme de création de l'arbre dégradé, un problème lié à la gestion de la pile pour l'emplacement du noeud courant après division du buffer ne survenait qu'après 9 minutes de traitement (sur le traitement d'un jeu de données de 600 Go).
Pour palier ce problème j'ai criblé le code de la partie serveur d'asserts (qui peuvent être désactivés au besoin) qui vérifient à chaque étape du processus de création de la base si il n'y a pas d'incohérence dans le format d'un point sélectionné, ou dans l'emplacement d'un champ d'adresse, etc... et fait en sorte de générer des fichiers de log à chaque génération/mise à jour d'un système de fichier.

4.1.2 Partie Client

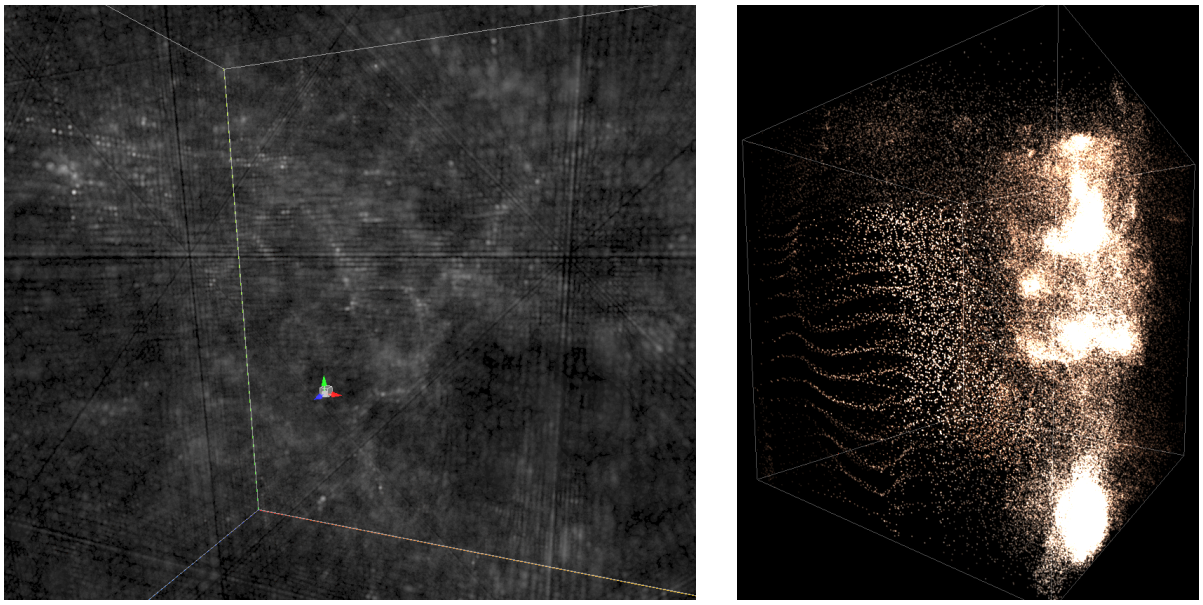
De façon similaire à la partie serveur, il peut être difficile de déterminer l'origine d'erreurs dans la partie client car ces dernières, avant d'être détectées, peuvent avoir été causées bien en amont des appels de fonction à cause d'un mauvais passage par référence, etc...
Cela est d'autant plus exacerbé par le fait que le Javascript est un langage interprété, et que l'application

possède une boucle d'update pour la mise à jour de l'affichage ce qui rend très difficile l'utilisation de points d'arrêt.
La solution trouvée a été de mettre en place un mode débug dans lequel chaque fonction appelée imprime son nom sur la console, ce qui facilite la remontée jusqu'à la source des erreurs.

4.2 Résultats obtenus

4.2.1 CoDa

Les capacités de la partie serveur de l'application ont été mises à l'épreuve en utilisant un snapshot de la simulation CoDa (contenant 4096^3 éléments), dont l'ensemble des fichiers pèse environ 4To, les données extraites consistant pour chaque élément en ses coordonnées xyz et la norme de son vecteur vitesse.
En utilisant un ordinateur possédant 8Go de RAM, la création du système de fichiers associé a pris 149 heures, avec 11 de plus qui ont été nécéssaires pour sa version dégradée.
Ces derniers pèsent respectivement 1,9To et 510Mo.
Concernant les performances des requêtes effectuées, il faut environ 40sec pour charger une vue dégradée de la simulation complète avec un niveau de profondeur de 7, et 15 pour un niveau 6.
En moyenne, 10sec sont nécéssaires pour réaliser une requête sur le système de fichier principal (pouvant retourner jusqu'à 2,5 milions de points).
Les images suivants (Figs. 4.1a, 4.1b, 4.2) ont été obtenues à partir de l'application :
La Fig. 4.1a représente la vue dégradée de la simulation complète, sur laquelle un filtre (de luminosité) a été appliqué en se basant sur le champ de poids de ses éléments.
La Fig. 4.1b montre les données réelles contenues par l'outil de sélection visible en Fig. 4.1a.



(a) Vue dégradée complète (lv7) (b) Zoom sur cible

FIGURE 4.1 – Utilisation de l'outil de zoom pour explorer CoDa

La Fig. 4.2 montre un autre ensemble de points issus des données réelles, cette fois avec un filtre coloré mettant en avant le champ vitesse des éléments (plus la couleur tends vers le rouge, plus la valeur est elevée).

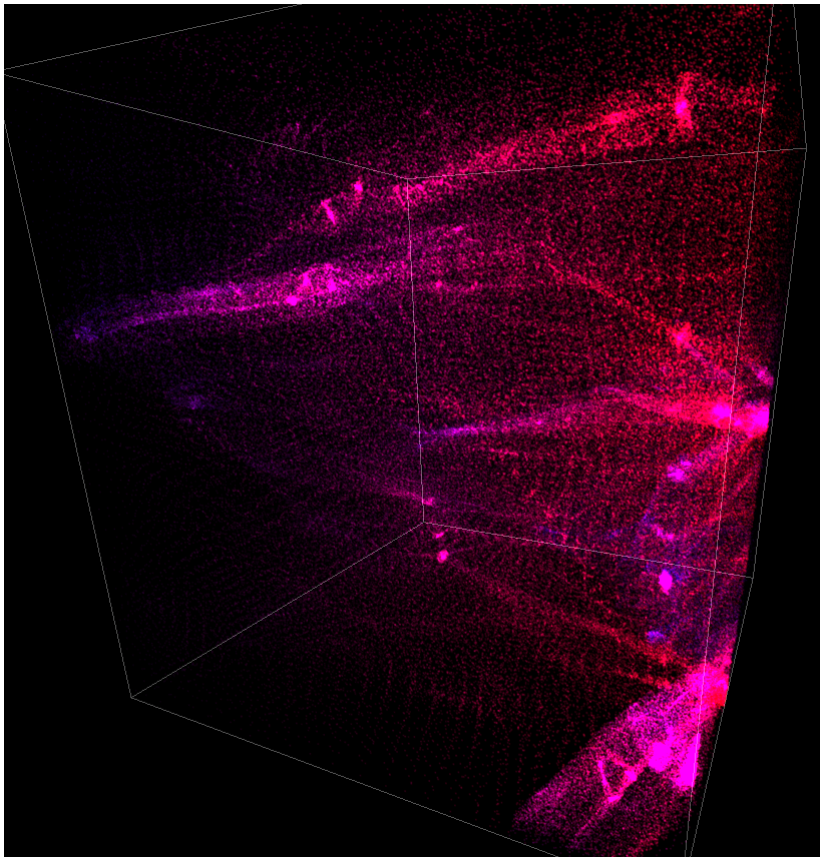
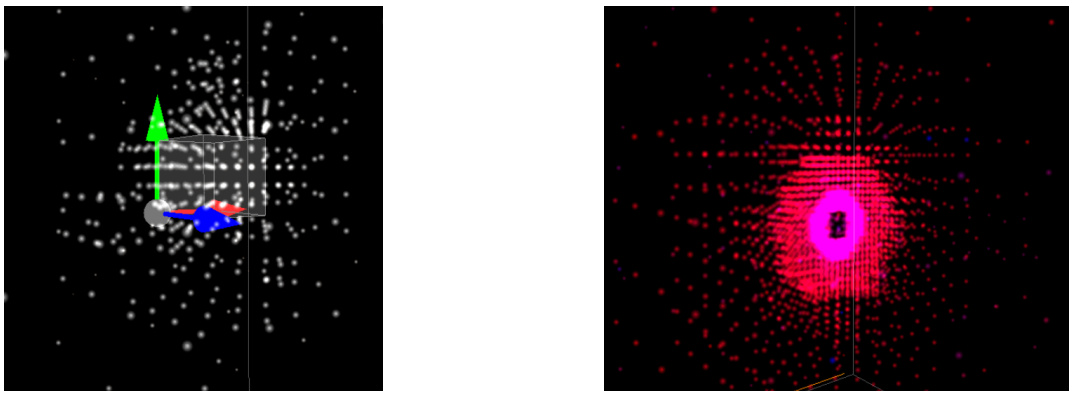


FIGURE 4.2 – Affichage du champ vitesse

4.2.2 Simulations de Frédéric Marin

La partie serveur de l'application a aussi pu être testée sur d'autres jeux de données, de structure complètement différente (bornes de la simulation, 8 champs par éléments) qui ont permis de valider la capacité du serveur à gérer d'autres formats de données. Ces dernières m'ont été fournies par Frédéric Marin, un astronome travaillant à l'Observatoire.
Les Figs 4.3a et 4.3b représentent le résultat de zooms récursifs effectués sur une simulation représentant le centre d'un trou noir, dont le centre était extrêmement dense.



(a) Vue dégradée complète (lv7) (b) Horizon du trou noir (dégradé récursif)

FIGURE 4.3 – Simulation de Frédéric Marin

4.3 Validation des objectifs

Le bilan du stage est très satisfaisant ; la plupart des objectifs ayant été atteints :

— La partie serveur est entièrement fonctionnelle et est suffisament robuste pour être utilisable sur des simulations de très grande taille (plusieurs To).

— La partie client a vu son format de données simplifié et il est beaucoup plus aisé d'y apporter des modifications.

— Une version notebook fonctionnelle de l'application a été mise en place.

Seule la partie concernant l'optimisation de la partie serveur grâce au Web Assembly n'a pas pu être abordée par manque de temps.

Conclusion

A l'issue de ce stage l'application est entièrement fonctionnelle et est prête pour sa diffusion.

Elle permet supporte désormais le traitement de simulations dont la taille est de l'ordre du Téra-octet, et est aisément paramétrable par l'utilisateur.

Du point de vue des compétences ce stage m'a beaucoup apporté, notamment en ce qui concerne les problématiques associées à la gestion de grands jeux de données, avec la prise en compte de contraintes importantes qui modifient radicalement la façon les traiter ; comme le fait d'utiliser un format de fichier binaire pour limiter au maximum l'espace disque utilisé, ainsi que la prise en compte des limitations de la RAM dans le développement des algorithmes.
J'ai également pu renforcer ma maîtrise du Javascript, plus particulièrement la manipulation des données binaires et la gestion des appels asynchrones.

Au cours de ce stage la première priorité était de livrer une application complète et fonctionnelle, mais il reste encore beaucoup de pistes à explorer quant à son optimisation (notamment du côté du Web Assembly)
Un autre point intéréssant concernant le futur de l'application est que les parties client et serveur peuvent être utilisées de façon complètement indépendantes ; on peut donc imaginer d'autres application à la génération du système de fichier.

Glossaire

CDS Centre de Données astronomiques de Strasbourg, labellisé depuis 2008 "Très Grande Infrastructure de Recherche" (TGIR) par le Ministère de l'Enseignement Supérieur et de la Recherche. 5–7

CoDa Cosmic Dawn, une simulation sur la formation des galaxies dans l'univers. 25

Jupyter Application web utilisée pour programmer dans des langages comme le Python ou le Javascript;qui permet de réaliser des notebooks, i.e. des programmes contenant à la fois du texte en markdown et du code . 1, 2, 9, 22, 23

Octree Arbre utilisé en représentation 3D, dont les fils -appelés octans- correspondent à 1 huitième du volume de leur parent. 8, 11, 13, 14, 20

SDSS Sloan Digital Sky Survey, catalogue d'objets astronomiques. 7

Table des figures

Bibliographie

[1] Coda. https://arxiv.org/abs/1511.00011.

[2] Documentation jupyter. http://jupyter.org/documentation.

[3] Documentation node.js. https://nodejs.org/api/.

[4] Documentation three.js. https://threejs.org/docs/index.html#manual/introduction/Creating-a-scene.

[5] Observatoire astronomique de strasbourg. https://fr.wikipedia.org/wiki/Observatoire_astronomique_de_Strasbourg.

[6] Octree. https://fr.wikipedia.org/wiki/Octree.

[7] Site du cds. http://cdsweb.u-strasbg.fr/about.