RAPPORT DE STAGE DE 3A

Systèmes de navigation 3D et mécanismes de recommandations

Thomas FORGIONE
3IN

Tuteur : M. Vincent CHARVILLAT

16 Mars 2015 — 25 Septembre 2015

Table des matières

Préambule

Ce stage de fin d'études s'est déroulé dans le laboratoire de recherche de l'IRIT, dans l'équipe VORTEX.

Ce stage a commencé en F215, salle dans laquelle il y avait Thierry Malon, un de mes collègues de projet long qui travaillait sur le projet Popart avec Simone Gasparini, et Bastien Durix, qui jouait le rôle du client de notre projet long.

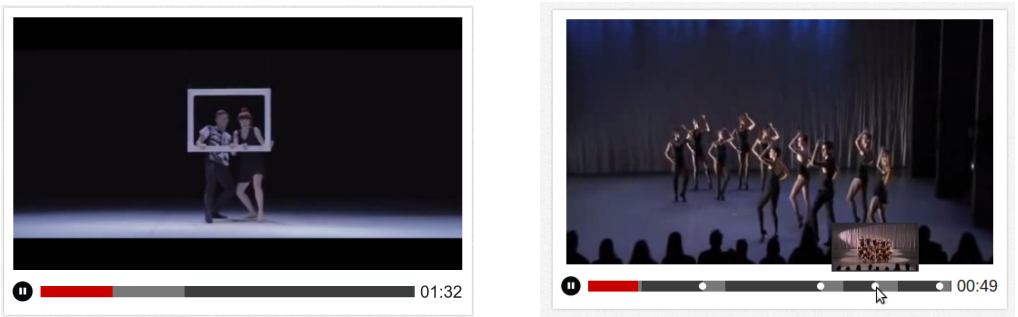
Quelques semaines plus tard, Émilie Jalras, autre collègue de notre projet long, est arrivée pour commencer son stage avec Sylvie Chambon sur les superpixels. Émilie n'ayant que Windows sur son ordinateur, il lui était impossible d'y développer du logiciel, et puisqu'aucune machine n'était disponible dans la salle F215, sa migration était inévitable. Par solidarité *stagiariale*, Thierry et moi avons donc migré vers la salle F207, où nous avons passé le reste de notre stage.

Première partie
Introduction

1. Contexte

Ce travail s'inscrit dans la continuité des travaux de Vincent Charvillat et d'Axel Carlier sur la prédictabilité du comportement d'un utilisateur en interaction avec un lecteur vidéo. L'objectif était alors de proposer un modèle de préchargement plus intelligent que le préchargement linéaire qui est actuellement implémenté sur les sites de streaming vidéo comme YouTube.

L'idée était de suggérer à l'utilisateur des instants potentiellement intéressants dans la vidéo de sorte à biaiser son comportement. Lorsqu'un utilisateur parcourt une vidéo, il peut sauter des passages et aller directement à d'autres. De nombreux travaux montrent qu'en ajoutant des recommandations sur une barre de navigation, l'utilisateur aura tendance à cliquer sur ces zones intéressantes plutôt qu'ailleurs. Plutôt que de précharger toute la vidéo linéairement, on peut donc précharger les zones intéressantes ce qui va réduire la latence lorsque l'utilisateur va sauter une partie pour aller vers une zone recommandée.



(a) Sans recommandations (b) Avec recommandations

Figure 1 – Des lecteurs vidéo avec et sans recommandations

Dans la figure 1b, les points blancs représentent des débuts de passages intéressants. Les zones gris clair représentent les passages chargés, et les zones gris foncé les passages qu'il reste à charger.

2. Objectifs

Ce projet a pour but d'appliquer les idées citées précédemment dans le contexte de la navigation et du streaming de modèles 3D. Nous avons donc développé une interface permettant de naviguer dans une scène 3D de manière plus simple que les interfaces actuellement existantes (et qui sont, le plus souvent, utilisées dans le monde du jeux vidéo). En effet, de nombreuses personnes n'ont pas l'habitude de jouer aux jeux vidéo, et peinent à utiliser les interfaces actuelles.

L'objectif est de faciliter la navigation grâce à un système de recommandations qui permettent à l'utilisateur de se déplacer facilement d'un point de vue intéressant à un autre, quitte à devoir utiliser les interactions classiques s'il désire un point de vue plus précis. En supposant que l'utilisateur soit susceptible de sauter vers les vues recommandées, on peut précharger des parties d'un modèle plutôt que d'autres de sorte à éviter que l'utilisateur se retrouve dans une zone de la scène qui n'est pas encore chargée.

Dans ce travail, on ne s'attachera pas à calculer des recommandations : elles seront supposées connues. Dans la pratique, elle seront définies manuellement.

3. Description globale du projet

Pendant ce projet, nous avons développé une interface qui permet de naviguer dans une scène en 3D. Cette interface sera complétée d'une tâche à réaliser (collecter des pièces rouges). Pour aider l'utilisateur à naviguer dans la scène, des mécanismes de recommandations seront introduits, et nous allons chercher à savoir à quel point ces mécanismes aident les utilisateurs.

Nous présenterons ensuite un mécanisme de chargement intelligent de la scène 3D. Nous ne parlerons pas de pré-chargement puisque ce projet n'est pas arrivé jusqu'au point où nous serions capable de prédire le comportement de l'utilisateur.

4. Présentation

Nous allons dans un premier temps parler des technologies que nous avons testées et utilisées. Nous verrons ensuite l'architeture du programme, qui peut paraître un peu complexe. Dans la partie V, nous présenterons l'interface que nous avons développée. Nous verrons comment nous avons testé cette interface dans la partie VI, et dans la partie suivante, nous verrons comment nous avons fait du streaming de modèle 3D.

Deuxième partie
Gestion de projet

1. Début du stage

Encadré par Vincent Charvillat et Géraldine Morin, ce stage a commencé par une phase de découverte du sujet, qui n'était alors pas clairement fixée : l'idée d'utiliser des recommandations pour influencer l'utilisateur afin d'être capable de prévoir ses interactions et de s'en servir pour réduire la latence était clairement présente, mais l'interface pour y parvenir était encore à définir et à développer. Il y avait en fait deux options pour le contexte de ce travail : la vidéo, ou la 3D.

Le stage a donc commencé par une phase bibliographique afin d'étudier l'état de l'art en termes de recommandations, préchargement, et d'interface utilisateur de manière générale. Au même moment, Vincent Charvillat et Axel Carlier étaient à Singapour, en train de finaliser un article, et j'ai pu leur apporter une petite aide :

— dans un premier temps, j'ai établi des profils de bande-passante lors de téléchargements. Pour ce faire, j'ai utilisé le programme `curl` qui affiche la vitesse instantanée de téléchargement chaque seconde ;
— dans un second temps, j'ai pu nettoyer une partie d'un code PHP servant à remplir les lignes d'une table d'une base de données. Le code était alors sensible à l'injection SQL.

Au bout de quelques semaines, j'ai décidé de faire le choix de travailler pour des contenus 3D et j'ai commencé à découvrir les multiples façons de faire des interfaces 3D via HTML et JavaScript.

1.1. Communication

Les communications intra-IRIT se faisaient principalement par mail. Lorsqu'il y avait plus de choses à dire, mais que cela ne nécessitait pas une réunion, les encadrants venaient me voir dans mon bureau pour discuter, notamment quand il y avait des nouveautés à faire ou faites dans le programme.

Des réunions étaient organisées lorsqu'elles étaient nécessaires, soit environ toutes les deux semaines. Ces réunions étaient souvent faites via vidéo-conférence, en présence de Wei Tsang Ooi, collaborateur de NUS (National University of Singapore), et je m'occupais de faire les compte-rendus des réunions par mail.

1.2. Tests

Le site a été déployé à l'adresse `http://3dinterface.no-ip.org/`. Les différentes choses à faire tester aux encadrants se sont retrouvées sur des pages à cette adresse.

2. Planning

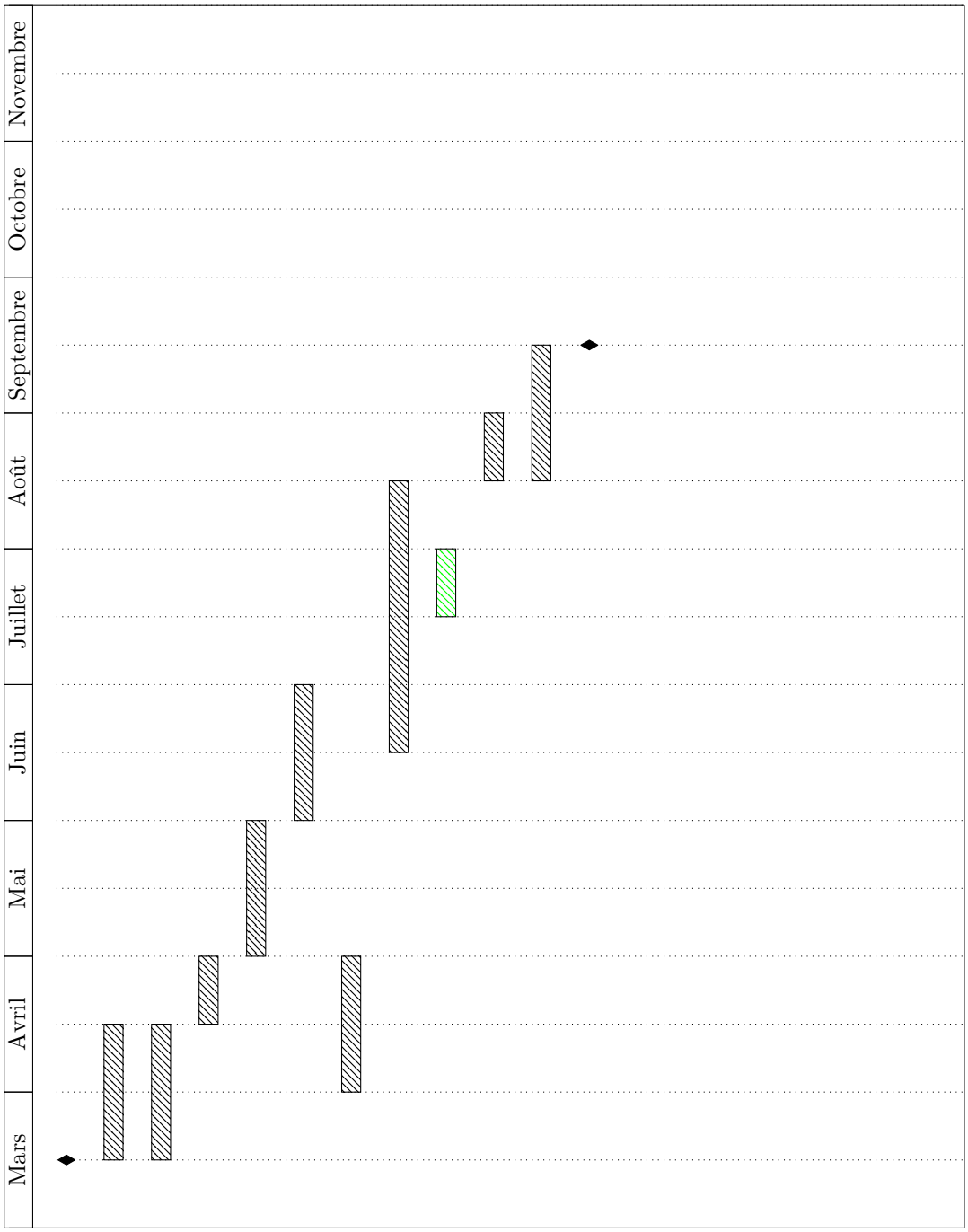


FIGURE 2 – Planning du projet

Troisième partie
Choix des technologies et prise en main

La première phase de stage était de choisir les technologies qui allaient être utilisées par la suite. Nous cherchions des technologies permettant la visualisation 3D sur un navigateur web afin de pouvoir faire une étude utilisateur simplement.

1. Côté client

Pour le côté client, il y avait plusieurs possibilités :
— WebGL, la spécification des fonctions permettant la 3D dans le navigateur
— Du code C++ compilé en JavaScript grâce à Emscripten
— N'importe quel moteur graphique qui puisse exporter vers JavaScript
— Une librairie JavaScript facilitant l'utilisation de WebGL

1.1. WebGL

WebGL est basé sur OpenGL ES (*Open Graphic Library for Embedded Device*). Cela signifie que de nombreuses fonctions présentes dans OpenGL rendant son utilisation plus simple ne sont pas disponibles dans OpenGL ES pour des raisons de performance. L'utilisation de WebGL devient donc assez complexe, et le simple dessin d'un cube tournant avec une lumière et une caméra devient très complexe. La contrepartie de WebGL est que toutes les fonctions élémentaires sont disponibles, et donc, il n'y a pas de limite imposée par un *framework* : tout devient possible, mais il faut le faire soi-même.

1.2. C++ vers JavaScript

Le code compilé de C++ et transformé en JavaScript avec Emscripten à ses inconvénients : comme WebGL ne supporte que OpenGL ES, il faut rédiger le code en utilisant OpenGL ES en C++ [1], et les contraintes de la sous-section précédentes sont toujours présentes.

Cette technique permet de bénéficier des performances (à l'exécution) du C++, et est très utile notamment dans le cas de portage de programmes déjà existants.

Un des inconvénients de passer par du C++ est que le binaire produit à la sortie sera relativement lourd (l'objectif du C++ est de faire le plus de calcul possible à la compilation, et les résultats de ces calculs seront donc inscrits dans le binaire) et le temps de chargement par le navigateur sera long.

1.3. Moteur graphique

Pour les moteurs graphiques, il y a de nombreux choix possibles. J'ai plutôt cherché des moteurs libres, et Vincent Charvillat m'a parlé de Minko. C'est un moteur graphique qui utilise Emscripten pour exporter vers JavaScript.

Le principal avantage d'utiliser un moteur graphique est qu'il permet d'avoir de très bons rendus assez facilement : toute la partie complexe du rendu est déjà implémentée.

1. En fait, certaines fonctions de OpenGL fonctionnent avec Emscripten : ce sont elles qui sont équivalentes en OpenGL et en WebGL

Son inconvénient est qu'il est assez lourd, long à prendre en main, et le programme JavaScript sera lui aussi lourd puisqu'il contiendra à la fois notre code et la librairie Minko compilée vers JavaScript. Les temps de chargement étant trop grand, nous nous sommes tournés vers la dernière option.

1.4. Three.js

Three.js est une librairie écrite en JavaScript qui utilise les fonctions de WebGL dans des classes qui permettent une utilisation plus simple de WebGL, tout en gardant sa puissance. Nous avons opté pour cette librairie pour développer notre interface.

Pour des raisons de simplicité, nous avons décidé de développer le code client pour Google Chrome et Firefox, les autres navigateurs ne sont donc pas (officiellement) supportés.

2. Côté serveur

Dans un premier temps, seul le côté client était pris en compte. Les programmes étaient écrits en JavaScript et ne nécessitaient pas de serveur. Quand les problématiques de dynamicité [2] sont arrivées, il a fallu choisir une technologie pour le côté serveur, et là, de nombreux langages étaient utilisables.

Plusieurs langages et framework ont été testée :
- le PHP : pratique pour des applications très petites, mais le langage est peu confortable dès que la taille du serveur grandit.
- les scripts CGI [3] (en python) : assez pratique pour des petites applications. Au final, cela ressemble un peu à du PHP mais dans un langage un peu plus confortable.
- Django (framework web en python) : très pratique, notamment pour des grosses applications, mais il a l'inconvénient d'être assez coûteux en mémoire vive (à ce moment, le serveur était hébergé sur une machine qui n'avait que 512 Mo de mémoire vive, et les temps de génération des réponses était de l'ordre de la dizaine de secondes).
- NodeJs : permet d'écrire le code serveur en JavaScript. Il est un peu moins pratique que Django (même si avec un peu de travail, on arrive à obtenir une architecture y ressemblant), mais il a l'avantage d'être très utilisé, et il existe donc de nombreuses librairies qu'on peut y ajouter.

J'ai fini par choisir NodeJs, notamment pour l'existance d'une librairie nommée Socket.IO, qui permet l'utilisation facile des sockets (côté serveur, et côté client, c'est-à-dire les WebSockets du navigateur web). Pour des raisons pratiques, le serveur a été herbergé sur un cloud gratuit (OpenShift).

3. Base de données

Pour le système de gestion de base de données, nous avons choisi Postgres (qui est libre et qui a largement fait ses preuves). OpenShift propose d'héberger lui-même la base de données, mais la version gratuite ne proposant qu'1 Go d'espace de stockage, nous avons préféré l'héberger nous-même.

2. On oppose les sites dynamiques (par exemple les réseaux sociaux) aux sites statiques (par exemple les blogs) : dans un site statique, l'utilisateur ne peut pas interagir avec le site. Le site se contente de délivrer une information.

3. Un script CGI est un programme dont l'exécution imprime une page web (ou autre) sur la sortie standard. Ils sont très simples à utiliser, puisqu'un simple *Hello world* en C peut devenir un script CGI

4. Développement, debug et déploiement

Pour éviter d'avoir des fichiers trop longs, nous avons choisi de séparer les sources dans de nombreux fichiers de taille plus petite, et de les fusionner automatiquement. Pour le développement, ils seront simplement concaténés grâce à un script développé spécialement pour cela, qui s'utilise de la même façon que Closure Compiler, qui sera utilisé pour la fusion au moment du le déploiement (ce dernier permet non seulement la fusion des fichiers mais aussi la minification [4] (effacement des commentaires et des retours à la ligne, simplifications des noms de variables et plus [5]). Pour le développement, on a utilisé nodemon et inotify, qui permettent de relancer le serveur local lorsqu'une modification est détectée (la fusion des fichiers est donc réeffectuée).

En ce qui concerne le versionnage des fichiers, nous avons utilisé Git avec deux dépôts :
— le premier, hébergé sur Github, sert au développement, et contient les fichiers fractionnés ainsi que les outils permettant la génération des fichiers fusionnés.
— le deuxième, hebergé chez OpenShift, qui contient la version finale du programme, permet de déployer le code du serveur quand les modifications sont propagées jusqu'à celui-ci.

Pour nous aider au debug, nous avons utilisé JSHint qui nous aide à détecter les erreurs potentielles liées aux subtilités du langage.

5. Documentation

En plus des rapports, deux documentations sont présentes.

5.1. *Github Wiki*

Github permet la création de Wiki pour chaque *dépôt* et nous nous en sommes servi pour de la documentation de haut niveau : il ne présente que des aspects théoriques de ce qui a été réalisé pendant ce projet.

5.2. L3D

Pour de la documentation de plus bas niveau (comment chaque classe ou méthode fonctionne) nous avons utilisé JSDoc (équivalent de javadoc mais pour JavaScript) et nous générons automatiquement des pages html pour avoir une documentation lisible et à jour sans avoir à parcourir le code.

4. la minification sert notamment à réduire la taille du script : n'oublions pas que nous parlons de serveur web, et il est donc intéressant de réduire la taille des programmes de sorte à les charger plus rapidement
5. en JavaScript, il est plus court d'écrire `!0` pour `true` par exemple.

6. Familiarisation

Pour me familiariser avec les technologies et librairies, j'ai développé quelques applications simples.

6.1. Bouncing Cube

Ceci est une première application de test de la librairie graphique. C'est une des premières applications qui a été faite, et elle m'a surtout servi à me familiariser avec la technologie, et notamment avec les clics sur les objets. C'est un simple cube, qui rebondit sur le sol et finit par s'arrêter. En cliquant sur le cube, il saute à nouveau.

6.2. Multisphere

Ceci est une des premières applications faites : elle m'a permi de tester l'affichage et le masquage des objets dans la librairie graphique. L'objectif était de raffiner le maillage au fur et à mesure en utilisant des modèles diffférents (plusieurs modèles sont chargés et on passe de l'un à l'autre en cliquant sur l'interface).

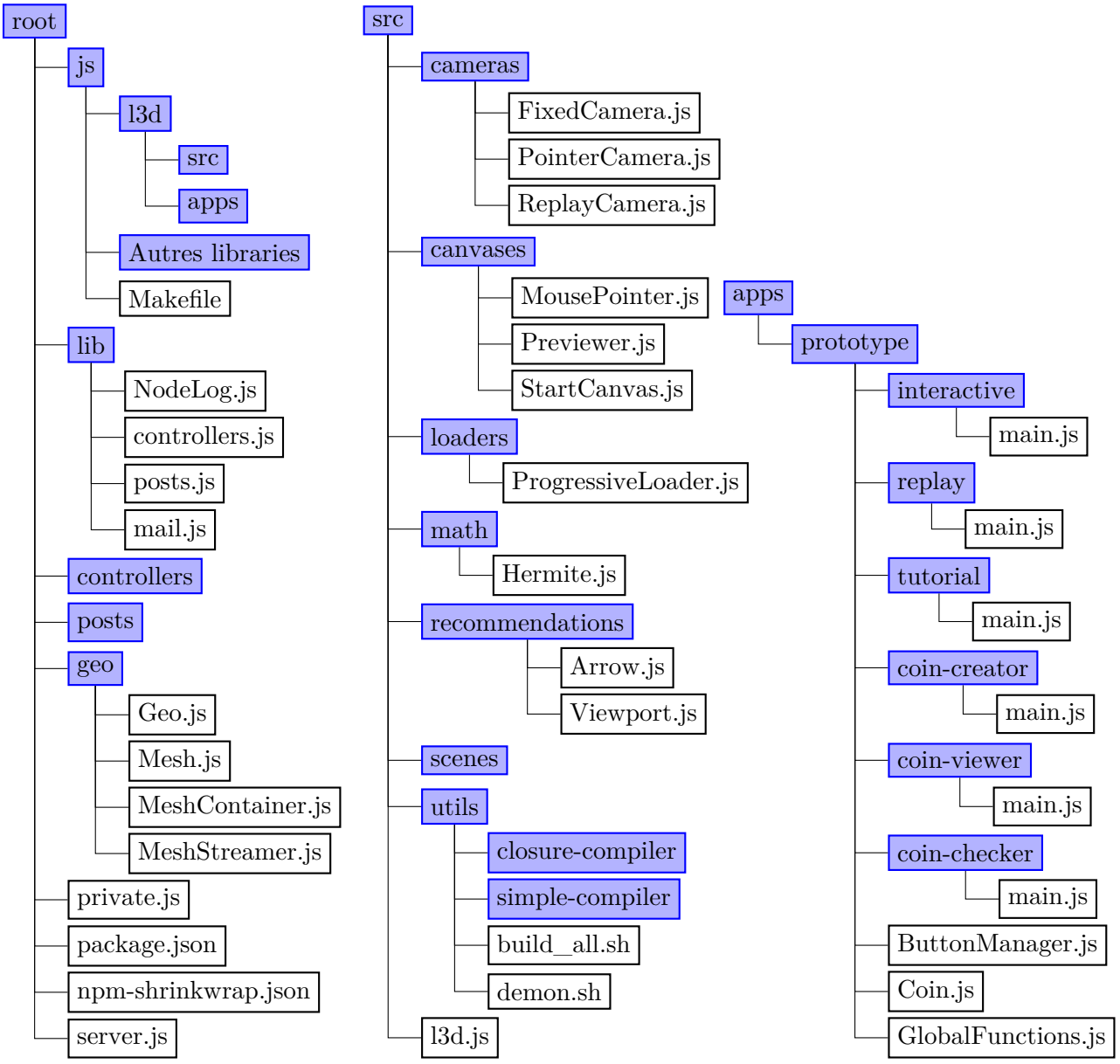
6.3. Stream-demo

Ceci est la première application que j'ai développée et qui utilise les sockets : c'est une version simplifiée du `ProgressiveLoader` que l'on détaillera dans la section VII.

Quatrième partie
Architecture du programme

Comme dit précédemment, le programme se décompose en un côté serveur et un côté client. Le cas du *streaming* sera traité à part (dans la partie VII, puisqu'il est comporte des parties à la fois sur le client et le serveur) et nous ne parlerons ici que du serveur, puis du code client.

Voici une *simplification* de l'arborescence de la version de développement :



(a) Vue globale (b) Détail des sources de L3D (c) Détail des applications

FIGURE 3 – Arborescence du *dépôt* de développement

1. Code serveur

Le code serveur est situé à la racine du projet : le fichier `server.js` est en fait le programme principal du serveur.

1.1. Dépendances

Le programme principal utilise de nombreuses libraires : les dépendances de notre serveur sont définies dans le fichier `package.json`. L'inconvénient d'avoir de nombreuses librairies en même temps est la compatibilité des versions : en effet, une des dépendances nécessite une version particulière d'une autre librairie. Chaque dépendance est une librairie, qui a elle même ses dépendances, etc...

J'ai recontré ce problème au cours du projet : une librairie utilisait une version précise d'une dépendance qui en nécessitait une autre. Nous avons donc utilisé une fonctionnalité de *npm* (le programme qui permet d'installer des librairies dans un serveur NodeJs) qui s'appelle *shrinkwrap* et qui permet de figer l'arbre de dépendance d'une application NodeJs.

Les *packages* que nous utilisons sont les suivants :
— *express* : un framework web pour NodeJs qui permet de gérer facilement les urls, les requêtes, les réponses, etc...
— *jade* : un moteur de template pour simplifier la génération des pages HTML
— *pg* : une librairie permettant la connexion à la base de données
— *body-parser* : une librairie permettant de traiter simplement les paramètres passés aux requêtes
— *cookie-parser* et *cookie-session* : une librairie gérant les sessions sous forme de cookies
— *socket.io* : une librairie permettant d'utiliser facilement les sockets (côté serveur et client)
— *serve-favicon* : une librairie pour choisir facilement l'icône du site
— *emailjs* : une librairie permettant de se connecter à une adresse e-mail

1.2. Modèle, vue, contrôleur

Pour ce projet, nous avons adopté une version simplifiée du design-pattern *modèle-vue-contrôleur* : en JavaScript, nos modèles seront des objets simples (en JavaScript, un objet n'est qu'une liste de paires *clé-valeur*), et le modèle sera limité à l'exécution des requêtes SQL.

Les contrôleurs sont chargés au démarrage par le fichier `controllers.js`, qui parcourt les dossiers qui sont contenus dans le dossier `controllers`. Dans chacun de ces dossiers, deux fichiers et un dossier sont présents :
— `index.js` qui contient des fonctions (contrôleurs) qui répondent à des requêtes
— `urls.js` qui contient les urls existantes et les fonctions auxquelles elles vont être associées
— `views`, dossier qui contient les vues qui vont être utilisées par les contrôleurs définis dans `index.js`

La même technique à été appliquée pour le dossier `posts` et le fichier `posts.js`, à la différence près que les requêtes traitées sont des requêtes POST et non pas des requêtes GET : elles servent principalement à stocker des informations dans la base de donneés.

2. Code client

Le code client est séparé en trois parties :
— une partie dans le répertoire `src` contenant de nombreuses fonctions et classes
— une partie dans le répertoire `apps` contenant des applications que nous avons développées
— des autres librairies développées par des tiers, dans le répertoire `js`
Le `Makefile` présent dans le dossier `js` est celui qui concatènera nos sources, les *minifieras* et génèrera les scripts que nous utiliserons par la suite.

Dans la suite, nous allons seulement parler de L3D, puisque c'est le code que nous avons développé.

2.1. Les sources

L3D est composée de plusieurs classes :
— les caméras, permettant de choisir des caméras avec des mouvements particuliers
— les canvas, qui permettent d'afficher des informations supplémentaires à l'écran
— les loaders, qui permettent de charger des modèles de manière différente de celles proposées par Three.js
— les classes mathématiques, comme les polynômes de Hermite
— les recommandations, notamment les flèches et les *viewports*

Elle contient aussi quelques fonctions qui permettent de créer les scènes que nous avons utilisées, et de les initialiser correctement.

Dans le répertoire `utils`, il y a plusieurs outils pratiques pour le développement et le déploiement : c'est là qu'est rangé le *minifier* de Google, Closure-Compiler, et une version simplifiée, le Simple-Compiler, qui utilise les mêmes paramètres mais se contente de concaténer le code.

Pour L3D, le premier fichier qui sera dans la version finale est `l3d.js`. Il contient simplement l'initialisation du *namespace* `L3D`, auquel toutes les classes et fonctions appartiendront.

2.2. Les applications

Les applications sont principalement composées de programmes principaux, qui utilisent les classes de L3D, ainsi, elles ne sont pas fusionnées avec L3D, et laissées dans le namespace global.

2.2.1. Interactive

Ceci est l'interface principale, où l'utilisateur doit rechercher les pièces. Nous en parlerons plus dans la partie V.

2.2.2. Replay

C'est l'interface qui crée une `ReplayCamera` et permet de visionner une expérience qui a été faite dans le passé.

2.2.3. Tutorial

C'est le didacticiel de l'application : il possède notamment une classe qui copie la caméra principale tout en permettant de vérifier que les interactions sont faites comme il faut.

2.2.4. Coin-creator, Coin-editor et Coin-checker

Ce sont des outils de développement qui seront détaillés dans la section VI.2.2.

Cinquième partie
L'interface

1. Interactions élémentaires

La première interface, sans recommandations, a été pensée pour être la plus simple possible. L'utilisateur contrôle une caméra qui se déplace librement dans une scène 3D. Elle est contrôlée par un ensemble de paramètres décrit dans la figure 4.

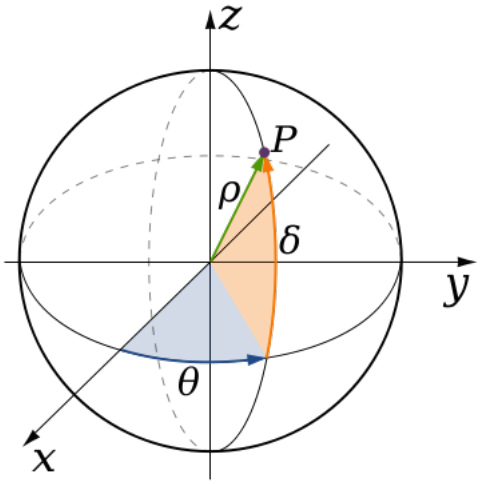


FIGURE 4 – Les paramètres du contrôleur. [6]

La translation de la caméra est contrôlée par le clavier : les touches Z, Q, S, et D servent respectivement à avancer, aller à gauche, reculer et aller à droite (de même que les touches fléchées). Concrètement, *avancer* revient à translater l'origine de la caméra O suivant le vecteur $\overrightarrow{OP}$, et *aller à gauche* correspond à translater O suivant le vecteur $\vec{z} \wedge \overrightarrow{OP}$, c'est à dire la normale du plan (OPz).

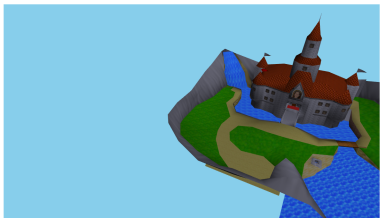
La rotation de la caméra (variation des angles θ et δ) de plusieurs manières :
— via le pavé numérique (2, 4, 6, et 8 pour tourner respectivement vers le bas, vers la gauche, vers la droite et vers le haut)
— via la souris, comme *drag-n-drop*, en cliquant un point de la scène et en le déplaçant (le mouvement de la caméra sera alors opposé au mouvement de la souris)
— via la souris, en mode *pointer-lock*, comme dans un jeu vidéo de tir

Par exemple, on peut tourner vers la gauche (c'est-à-dire diminuer θ) en :
— cliquant un point et en le déplaçant vers la droite (en mode *drag-n-drop*)
— déplaçant la souris vers la gauche (en mode *pointer-lock*)
— en appuyant sur la touche 4 du pavé numérique

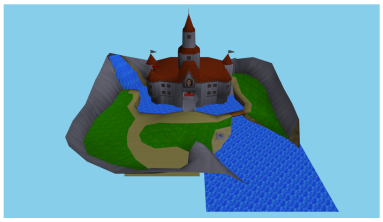
6. Contenu soumis à la licence CC-BY-SA 3.0 (http://creativecommons.org/licenses/by-sa/3.0/deed.fr) Source : Article Coordonnées sphériques de Wikipédia en français (http://fr.wikipedia.org/wiki/Coordonn%C3%A9es_sph%C3%A9riques).

Voici un petit tableau qui permet de montrer comment la caméra tourne autour de son centre en fonction des interactions.

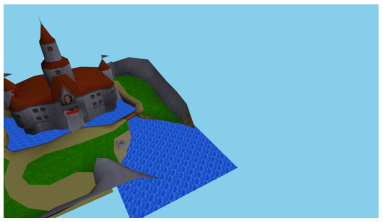
(a) Vers le haut ($\delta \nearrow$)



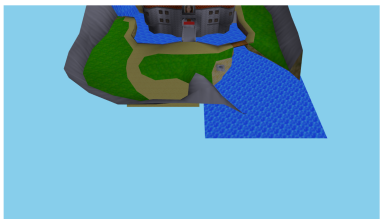
(b) Vers la gauche ($\theta \searrow$)



(c) Angle initial



(d) Vers la droite ($\theta \nearrow$)



(e) Vers le bas ($\delta \searrow$)

FIGURE 5 – Les différentes rotations possibles

Ces techniques de navigation 3D restent quand même complexes à utiliser, surtout pour quelqu'un qui n'est pas habitué à jouer aux jeux vidéo. Nous allons donc ensuite voir comment nous pouvons essayer de faciliter la navigation pour des utilisateurs non-initiés.

2. Les recommandations

Les recommandations sont là pour suggérer des points de vue à l'utilisateur. Elles permettent d'aider la navigation. Elles sont affichées sous forme d'objets 3D ajoutés à la scène. Deux affichages ont été testés.

2.1. Les *viewports*

Les *viewports* sont les affichages les plus simples : ils représentent une caméra, avec son centre optique et son plan image.

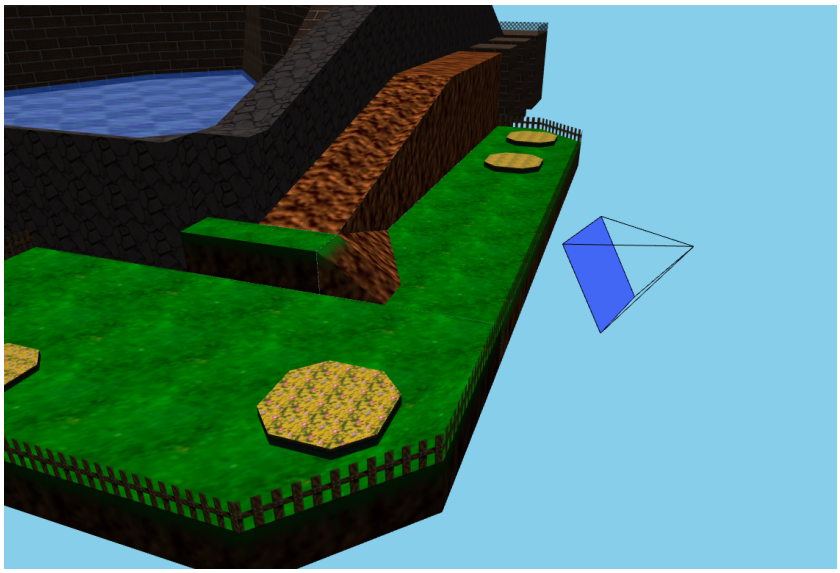


FIGURE 6 – Une recommandation *viewport*

Cette façon d'afficher une recommandation a l'avantage d'être simple, de ne pas beaucoup masquer le reste des modèles et suggère assez bien l'idée d'un *point de vue recommandé*, mais elle a l'inconvénient d'être ambiguë à cause de la perspective (dans cette image, il peut être difficile de savoir si le point de vue est dirigé vers le modèle ou vers nous).

2.2. Les flèches

2.2.1. Principe

Les flèches sont supposées être plus intuitives pour un utilisateur qui n'a pas l'habitude des *viewports* précédemment utilisés. Plutôt que de suggérer un point de vue, elles suggèrent le mouvement qui va mener à ce point de vue.

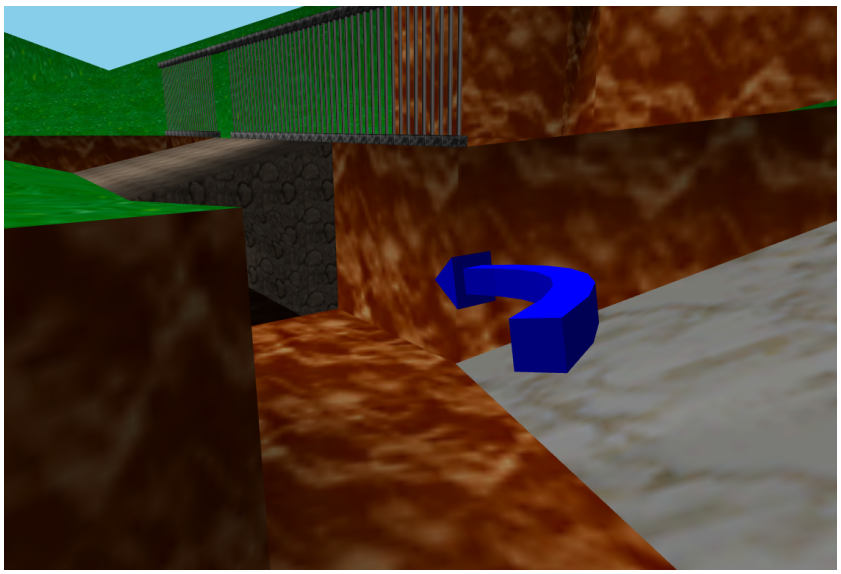


FIGURE 7 – Une recommandation flèche

2.2.2. Courbure de la flèche

Pour le dessin des flèches, plusieurs choses sont à prendre en compte :
— il faut éviter que la flèche soit trop *présente* à l'écran et qu'elle occulte trop le reste de la scène
— il faut que la flèche soit dans un plan qui ne soit pas orthogonal au plan image de la caméra. En effet, si la flèche est dans un plan orthogonal au plan image de la caméra, elle se projettera comme un segment et on ne pourra pas voir la courbure de la flèche.

Pour trouver la courbure de la flèche, nous posons C le centre de la caméra, R le centre de la recommandation, et R' le vecteur qui donne la direction de la recommandation. Nous cherchons ensuite un polynôme

$$\begin{array}{rccc} f & : & [0,1] & \to & \mathbb{R}^3 \\ & & t & \mapsto & (x,y,z) \end{array} \qquad \text{tel que} \qquad \left\{ \begin{array}{lcl} f(0) & = & C \\ f(1) & = & R \\ f'(1) & = & \lambda R' \text{ avec } \lambda \in \mathbb{R}^+ \end{array} \right.$$

Le problème de ce polynôme est qu'il ne vérifie aucune des contraintes énoncées précédemment : puisque $f(0) = C$, l'extrémité de la flèche est dans la caméra, et va donc masquer toute la scène, et la courbe va en plus se projeter sur une ligne sur l'écran.

Pour solutionner le premier problème, nous nous contenterons d'afficher seulement la flèche pour des valeurs du paramètres $t \in [0.5, 1]$ (c'est-à-dire qu'on n'affichera que la moitié de la flèche la plus lointaine de la caméra).

Pour solutionner le deuxième problème, nous allons translater le centre de la caméra vers le bas et le côté de la direction de la recommandation, pour accentuer la courbure de la flèche, et nous résolvons plutôt le système suivant :

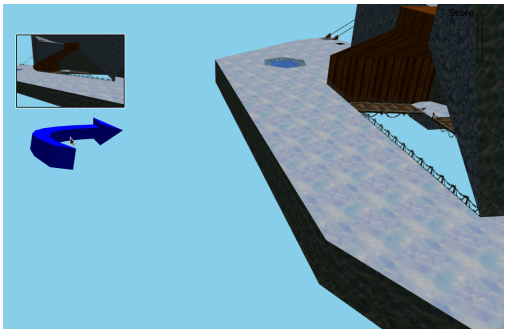
$$\left\{ \begin{array}{lcl} f(0) & = & C - e_z + \lambda R' \\ f(1) & = & R \\ f'(1) & = & \lambda R' \end{array} \right. \qquad \text{avec} \quad \lambda \in \mathbb{R}^+$$

2.3. Les interactions

2.3.1. Au survol

Cette fonctionnalité est inspirée des récents lecteurs vidéo sur le web. Lorsque l'on regarde une vidéo, on a la barre de *seeking* en bas et passer le curseur sur cette barre affiche l'image de la vidéo à l'instant visé. Nous avons simplement adapté cette technique à nos recommandations : lorsque le curseur survole une recommandation, une prévisualisation est affichée dans un petit rectangle au voisinage du curseur.

(a) Une prévisualisation sur Youtube



(b) Une prévisualisation dans notre système

2.3.2. Au clic

Lors d'un clic sur une recommandation, la caméra suit un mouvement fluide jusqu'au point de vue recommandé. La trajectoire est définie par un polynôme interpolant tel que :

— la position initiale est la position de la caméra
— la position finale est la position de la recommandation
— la dérivée de la trajectoire à l'instant final est la direction de la recommandation

Ce mouvement fluide est là pour ne pas dérouter l'utilisateur qui pourrait *se perdre* si jamais il était téléporté.

De plus, les recommandations se comportent comme des liens hypertextes : elles sont bleues si elles n'ont jamais été cliquées, et deviennent violettes si l'utilisateur les a déjà consommées. Ceci est fait pour qu'un utilisateur puisse savoir par où il est passé, et ce qui lui reste encore à visiter.

3. Autres éléments de navigation

Pour faciliter la navigation, quelques autres éléments de navigation sont présents.

FIGURE 9 – Les différents éléments de l'interface

1. *Reset camera* : pour chaque scène, une position initiale est définie. Cliquer sur ce bouton ramène la caméra à sa position initiale.
2. *Previous* : à chaque clic sur une recommandation, les positions intiale et finale sont sauvegardées. Cliquer sur ce bouton ramène à la position précédente.
3. *Next* : cliquer sur ce bouton ramène à la position suivante.
4. *Pointer lock* : permet de passer du mode *pointer-lock* au mode *drag-n-drop* et vice-versa.
5. *Music* : un lecteur qui contrôle une petite musique qui permet de se mettre dans l'ambiance de la scène.
6. *Coin gauge* : une jauge qui représente l'avancement de la récupération des pièces.
7. *FPS counter* : indique la période de rafraîchissement du rendu.

Sixième partie
Évaluation des interfaces

Pour tester le comportement des utilisateurs face aux recommandations, nous ne pouvons pas nous contenter d'observer le comportement des utilisateurs puisqu'ils ne sont pas nécessairement intéressé par nos scènes. Ils se contenteraient probablement de se promener un peu dans la scène puis partiraient.

Pour palier à ce problème, nous avons ajouté des *pièces* à chercher dans la scène. L'objectif est basé sur l'hypothèse que si un utilisateur a ramassé toutes les pièces rouges, il aura parcouru l'intégralité de la scène.

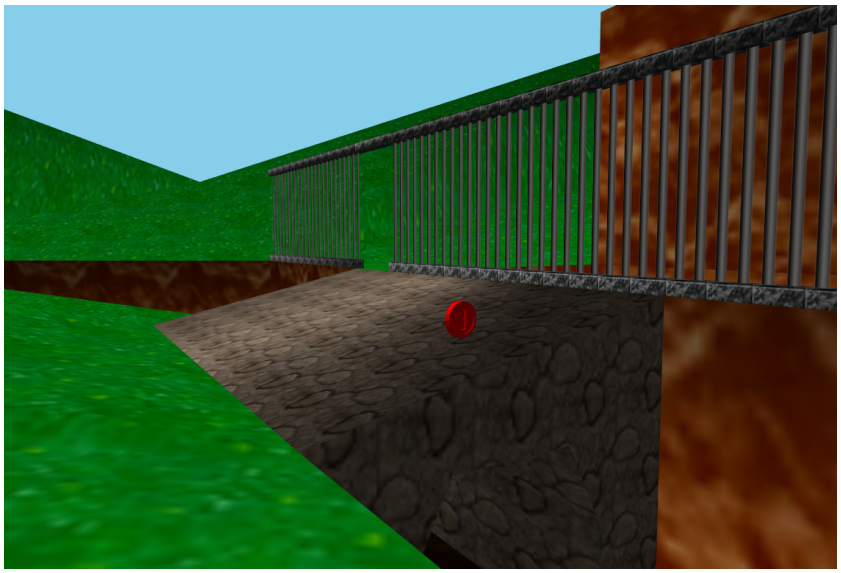


FIGURE 10 – Une pièce rouge

Pour éviter de biaiser les utilisateurs, il est nécessaire d'éviter la dépendance entre les recommandations et les pièces. En effet, si les recommandations mènent toujours aux pièces, l'utilisateur va croire que les recommandations sont là pour trouver les pièces, alors qu'elles ne doivent être là que pour aider la navigation. Pour cela, nous avons mis en place un système de tirage aléatoire des pièces parmi plusieurs positions possibles (ceci sera détaillé dans la section VI.2.2).

1. Déroulement de l'expérience

1.1. Première page

La première page présente rapidement l'expérience. Elle vérifie aussi le navigateur : si le client est sur Google Chrome ou Firefox, un lien apparaîtra pour passer à la suite, sinon, un message d'erreur s'affichera.

1.2. Identification

Cette page nous permet d'en savoir un peu plus sur l'utilisateur : nous allons demander l'age, le sexe et les habitudes en terme de jeux vidéo de l'utilisateur (nous avons fait l'hypothèse que la capacité des utilisateurs à manier notre interface allait dépendre fortement de leur habitude aux jeux vidéo). Nous leur demandons notamment de noter leurs capacités en terme de jeux vidéo (entre 1 et 5 étoiles).

1.3. Didacticiel

Ensuite, nous demandons à l'utilisateur de suivre un didacticiel : c'est une expérience normale mais guidée. Des messages indiquant les interactions à faire aideront l'utilisateur à s'habituer à cette interface. On y présente les moyens de déplacer la caméra, puis les recommandations et les différents boutons de l'interface.

1.4. Les trois vraies expériences

C'est ensuite que la *vraie* partie de l'étude utilisateur commence : à 3 reprises, l'utilisateur doit remplir dans une scène avec certaines pièces rouges à trouver, et un certain style de recommandations [7] pour l'aider. Nous expliquerons dans la sous-section VI.2.1 comment le choix de la scène et du style de recommandation sera fait.

Dans chacune des expériences, l'utilisateur devra chercher des pièces rouges, en s'aidant (ou pas) des recommandations. L'expérience se terminera soit quand l'utilisateur aura trouvé les huit pièces rouges, soit une minute après avoir trouvé la $6^{\text{ème}}$ pièce rouge. Pour éviter que l'utilisateur soit *frustré* de ne pas avoir trouvé toutes les pièces, nous n'indiquons pas clairement le nombre de pièces qu'il a, ou qu'il lui reste à trouver. Simplement, une *vague* idée de sa progression (voir figure 9 - élément 6).

1.5. Le *feedback*

Après avoir fini ces expériences, l'utilisateur se retrouvera sur un formulaire lui demandant son avis quand à la difficulté de l'interface, et l'utilité des recommandations pour se déplacer dans la scène.

2. Génération des expériences

2.1. Choix de la scène et du style de recommandations

Il y a trois scènes disponibles, et sur chaque scène, nous avons placé des pièces dans de nombreuses positions. Lorsqu'un utilisateur commence une expérience, l'algorithme de selection fonctionne de façon à faire des expériences sur les mêmes scènes avec les mêmes dispositions de pièces mais des styles de recommandations différents pour des utilisateurs de même niveau, de sorte à pouvoir comparer l'efficacité des recommandations.

Parmi les paramètres qui varient, nous avons :
— le niveau du joueur
— la scène
— la disposition des pièces rouges
— le type de recommandations

Le système que nous avons fait cherche les joueurs qui ont le même niveau pour leur donner les mêmes scènes avec la même disposition de pièces mais avec des styles de recommandations différents, de sorte à pouvoir les comparer.

2.2. Positions possibles des pièces

Pour choisir les positions possibles des pièces dans chaque scène, trois outils ont été développés.

7. aucune recommandation sera considéré comme un style de recommandation, de sorte à comparer la présence à l'absence de recommandation pour la navigation

2.2.1. Coin-creator

Cette interface permet de naviguer dans une des scènes possibles, et de cliquer sur une paroi pour créer une pièce. Cliquer sur un pièce la détruit, et un bouton permet d'envoyer les pièces par mail à mon adresse.

J'ai ainsi envoyé ce mail aux encadrants qui ont eux même pu créer les pièces, et j'ai ensuite fait un programme qui rassemble les pièces et élimine les pièces trop proches les unes des autres.

2.2.2. Coin-editor

Cette interface a été conçue pour corriger les imprécisions provenant de l'interface décrite précédemment. En effet, parfois, une pièce peut intersecter un mur. Coin-editor permet de cliquer sur une pièce pour pouvoir accéder à ses propriétés depuis la console JavaScript, et modifier sa position, puis renvoyer un mail avec les positions corrigées.

2.2.3. Coin-checker

Cette interface a été conçue lorsque nous nous sommes aperçus que certaines pièces étaient passées de l'autre côté du modèle. Il était ainsi difficile de les trouver et cette interface permet de chercher les pièces parmi toutes celles qui existent. On peut cliquer sur les pièces pour les récupérer et un compteur indique le nombre de pièces restantes.

3. Collecte des informations

Ces expériences n'auront d'interêt que si nous sommes capables de les analyser par la suite et de comprendre comment les utilisateurs interagissent avec les recommandations. Nous avons donc mis en place un système de collecte des actions de l'utilisateur basé sur les XmlHttpRequests de JavaScript.

Côté serveur, il y a quelques urls qui permettent d'enregistrer des informations dans des tables prévues à cet effet. Chaque évènement contient la date à laquelle il a été envoyé par le client ainsi que l'id du client et de l'expérience qu'il est en train de faire (ces deux derniers sont stockés dans la session sur le serveur). Les évènements enregistrés sont les suivants :

ArrowClicked : créé quand l'utilisateur clique une recommandation, accompagné de l'id de la recommandation cliquée.

CoinClicked : créé quand l'utilisateur récupère une pièce rouge.

KeyboardEvent : créé quand l'utilisateur appuie ou relâche une touche du clavier, accompagné de la position courante de la caméra et d'un booléen indiquant si elle a été appuyée ou relâchée. Dans le cas du *drag-n-drop* ou du *pointer lock*, aucun évènement (autre que les *mouse-move* qui ne sont pas stockés en base de données parce qu'ils sont trop nombreux) n'est envoyé. Ainsi, pendant toute la durée du mouvement de l'orientation de la caméra, on n'aura aucune information. Pour palier à ce problème, nous enverrons régulièrement des *KeyboardEvent* factices pour sauvegarder l'état de la caméra.

ResetClicked : créé quand l'utilisateur réinitialise la position de la caméra.

PreviousNextClicked : créé quand l'utilisateur clique sur les boutons précédente ou suivante, on stockera la position finale en base de données.

Hovered : créé quand l'utilisateur survole ou sort d'une recommandation avec le curseur.

PointerLocked : dans le cas où l'utilisateur utilise l'option *pointer lock*, il sera créé au moment où le pointeur sera capturé et où il sera libéré.

SwitchedLockOption : créé quand l'utilisateur change d'option entre *pointer locked* et *drag-n-drop*.

FpsCounter : chaque seconde, cet évènement est créé pour connaître le *framerate* du client, de sorte à savoir si les performances de sa machine ont pu lui poser problème dans ces expériences.

4. Replay

Afin de nous assurer que toutes les informations nécessaires étaient bel et bien récupérées en base de données, nous avons mis en place un programme permettant de rejouer les expériences stockées.

Pour le replay, nous avons simplement considéré les évènements `ArrowClicked`, `CoinClicked`, `KeyboardEvent`, `ResetClicked` et `PreviousNextClicked`, les autres évènements n'ayant pas d'influence sur la position de la caméra.

La génération du chemin de la caméra est faite de manière assez simpliste. En fait, nous accordons peu d'importance aux touches appuyées, mais plutôt aux positions de la caméra au moment de ces évènements. Nous interpolons entre les `KeyboardEvent`, le `ResetClicked` réinitialise la position de la caméra brutalement et les autres évènements utilisent les polynômes de Hermite comme pour le suivi des recommandations lors d'une expérience.

5. Déploiement

Pour l'instant, cette étude a été faite par des collègues (étudiants, chercheurs...), mais nous pensons ensuite le déployer vers une plateforme de crowd-sourcing (MicroWokers) où nous paierons les utilisateurs afin d'avoir plus de résultats à analyser.

Septième partie
Streaming de modèle 3D

Le but ultime de ce projet est de biaiser l'utilisateur avec les recommandations de sorte à être capable de prévoir ses déplacements futurs, et ainsi de précharger les parties du modèle qui vont être vues. Cette section présente le travail qui a été réalisé dans le domaine du chargement de modèle.

Évidemment, cette partie est celle qui commence après la fin de la première partie : il faut non seulement connaître l'influence des recommandations sur l'utilisateur, mais aussi être capable de prévoir le comportement de l'utilisateur, et enfin de s'en servir pour précharger les bonnes parties du modèle. Tout ceci n'étant pas encore possible, le travail qui a été fait est une version simplifiée : il n'y aura aucune prévision du comportement ici.

Introduction

Notre problématique ici est de transférer des modèles 3D sur le réseau. Les modèles sont stockés sur le serveur au format `.obj` et sont constitués :

- des matériaux (`usemtl`) : cela définit le matériau utilisé pour les faces qui vont suivre. Un matériau est notamment défini par ses constantes de réflexions optiques, ses textures, etc...
- de sommets (*vertices*) : des points 3D
- de coordonnées de textures : des points en 2D qui référencent un point d'une image
- de normales : des vecteurs en 3D
- de faces : une liste de 3 ou 4 sommets (représentés par leurs indices), avec éventuellement leurs coordonnées de textures et/ou normales

La seule contrainte d'ordre des éléments est qu'une face qui contient des sommets, coordonnées de textures ou normales n'arrive pas avant ses sommets, coordonnées de textures ou normales dans le fichier : si l'on parcourt les lignes du fichier, on doit déjà avoir toutes les informations sur la face.

Généralement, le fichier `.obj` vient souvent avec un fichier `.mtl` qui contient la définition des matériaux (leurs noms, leurs textures, leurs constantes...).

1. Architecture

Puisque cette partie traite à la fois du client et du serveur, les fichiers peuvent sembler éparpillés. Les fichiers concernés sont les fichiers du répertoire `geo` pour le serveur et le fichier `ProgressiveLoader.js` qui est le client.

1.1. Serveur

Nous avons rappelé que les modèles au format `.obj` contenaient plusieurs *sous-modèles* avec des matériaux différents, le serveur connait les classes suivantes :

- la classe `Mesh`, qui représente un *sous-modèle*, avec ses faces, ses coordonnées de textures, ses normales, son matériau
- la classe `MeshContainer`, qui représente un modèle au format `.obj`, en tant que liste de `Mesh`
- la classe `MeshStreamer`, qui permettra d'envoyer un modèle au client

Pour avoir une bonne performance, il faut éviter de re-parser les fichiers `.obj` et de recréer les `MeshContainer` à chaque requête.

Dans le fichier `MeshContainer.js`, une liste de modèle à charger (ainsi que le chemin permettant d'accéder aux fichiers) permet au serveur de créer les `MeshContainer` au démarrage, et ainsi, le `MeshStreamer` n'a qu'à prendre une référence vers ce `MeshContainer` (qui existe déjà, ce qui réduit la latence).

1.2. Client

Le client de la partie *streaming* de ce projet est contenue dans la classe `ProgressiveLoader`. Celle-ci crée des modèles 3D vides, puis établit la connexion avec le serveur et remplit les modèles au fur et à mesure que les informations arrivent du serveur. Dans les sections suivantes, nous allons montrer les différentes stratégies de *streaming* que nous avons mises en place.

2. Streaming linéaire

La première étape de cette fonctionnalité a été de faire un système client-serveur permettant le streaming de modèle 3D. En effet, les *loaders* de modèles présents dans la libraire Three.js ne permettent pas le chargement progressif : ils se content d'envoyer une requête vers le fichier contenant le modèle et à créer un modèle une fois que le fichier est chargé complètement.

Pour commencer, nous avons donc utilisé Socket.IO, une librairie permettant de gérer les sockets facilement avec JavaScript et Nodejs, pour faire une première version simplifiée du streaming : on travaillait sur un modèle ne contenant que des sommets et des faces (donc pas de textures ni de normales) et le protocole fonctionnait ainsi :

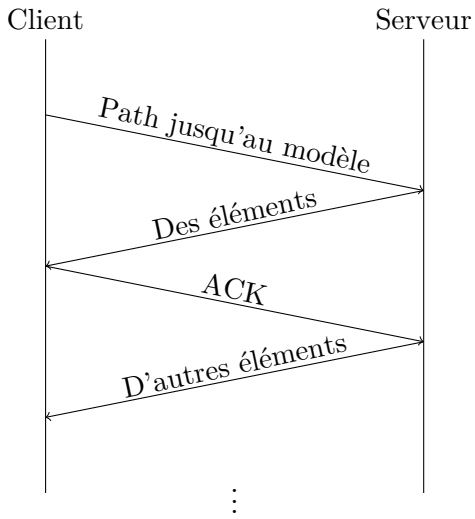


FIGURE 11 – Transmission élémentaire

Le serveur envoyait alors les éléments dans l'ordre dans lequel ils étaient présents dans le fichier du modèle 3D. Le gros inconvénient de cette méthode est que souvent, les sommets sont présents au début du fichier, et les faces vers la fin : nous recevons donc des informations de sommets au début qui ne nous permettent rien d'afficher, puis toutes les faces d'un coup, ce qui fait que le streaming n'est pas aussi progressif que l'on souhaiterait.

Dans le cas d'un modèle sans coordonnées de textures et sans normales, la seule condition pour pouvoir afficher une face est d'avoir envoyé les sommets qui la composent. On peut donc améliorer la fluidité en réarrangeant le fichier `.obj` : il suffit de faire apparaître les faces dès que les sommets sont disponibles.

Dans le cas où l'on souhaite gérer les textures et les normales, utiliser cette technique est beaucoup plus compliqué puisqu'il faut aussi décider de l'ordre relatif entre les sommets, coordonnées de textures et normales : en effet, pour envoyer une face le plus tôt possible, il faut que toutes ces composantes soient envoyés, et il est donc nécessaire de mélanger les sommets, coordonnées de texture et normales.

3. Streaming linéaire amélioré

La remarque précédente conduit directement à cette méthode. Le principe reste le même que celui précédent (figure 11), mais les éléments seront envoyés différemment : on va en fait parcourir les faces directement.

Le serveur va garder en mémoire ce qui a déjà été envoyé et ce qui ne l'a pas été, et envoyer les faces dans l'ordre : si certains éléments des faces n'ont pas encore été envoyés, on les enverra juste avant d'envoyer la face en question.

On peut de cette façon à la fois gérer les coordonnées de texture et les normales, tout en envoyant les faces le plus tôt possible (on peut aussi noter que si certains sommets / coordonnées de textures / normales sont inutilisés dans les faces, ils ne seront pas envoyés et on s'évite donc de transférer des données inutiles).

C'est dans cette version que nous avons commencé à nous intéresser à la façon de gérer les matériaux. Dans Three.js, un objet 3D est lié à un materiau, et nous sommes donc obligés de créer autant d'objets que de materiaux. Pour cela, nous avons légèrement modifié notre protocole :

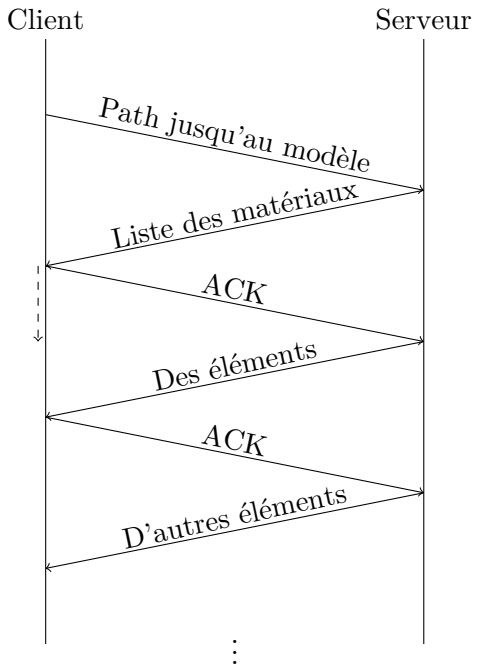


FIGURE 12 – Transmission avec gestion des matériaux

Les directives de matériau à utiliser (`usemtl`) seront ignorées, et les faces seront envoyées avec l'indice de l'objet auquel elles appartiennent, ce qui permettra au client de savoir dans quel objet (et donc avec quel matériau) elles doivent être ajoutées.

4. Streaming intelligent

C'est la dernière version du streaming qui a été faite sur ce projet. À chaque transfert, le client envoie sa position au serveur (ainsi que les plans définissant son *frustum*[8]) et le serveur va parcourir les faces du modèle en cherchant celles qui apparaissent dans le *frustum*. On évite ainsi d'envoyer les faces du modèle qui sont derrière la caméra et que l'utilisateur ne voit pas.

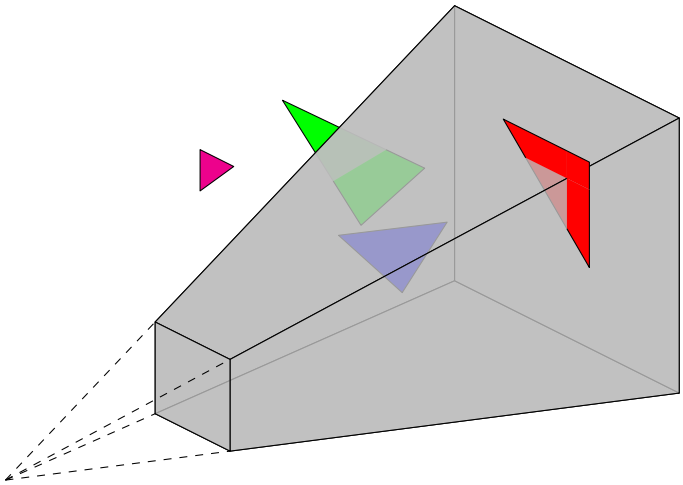


FIGURE 13 – Le frustum de la camera et différents objets

Dans cette version, on considère qu'une face apparaît dans le *frustum* si un de ses sommets y appartient. Évidemment, une face très grande pourrait appraître dans le *frustum* sans qu'aucun de ses sommets n'y soit, mais nous n'avons pas traité ce cas particulier ici (dans la figure 13, seuls les triangles verts et bleus seront envoyés, bien que le triangle rouge devrait être envoyé lui aussi).

Bien sûr, s'il n'y a plus de faces dans le *frustum* de la caméra, on va envoyer les faces en suivant la méthode de la section précédente.

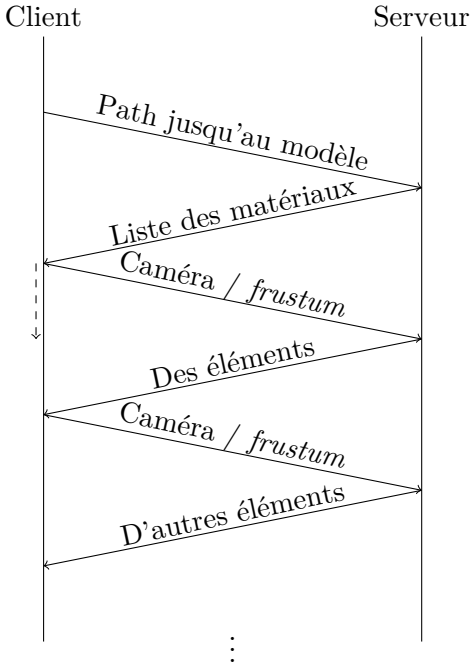


FIGURE 14 – Version finale

8. les bords du champ de vision de la caméra

Remerciements

Je tiens particulièrement à remercier

— Vicent Charvillat, pour m'avoir donné l'occasion de faire un stage intéressant, et pour toute l'aide qu'il a pu me fournir
— Geraldine Morin, pour sa disponibilité
— Axel Carlier, pour son aide et ses conseils sur les points plus techniques
— Wei Tsang Ooi, pour ses suggestions souvent pertinentes

mais je me dois aussi de remercier

— Thierry Malon et Émilie Jalras, pour avoir joué le rôle de canard en plastique [9] et plus encore,
— Julien Fayer, qui nous a accompagné durant les derniers mois de nos stages
— Bastien Durix,
— Vincent Angladon, pour ses visites régulières
— David Courtinot, qui m'a fourni un soutien psychologique lorsque JavaScript était méchant avec moi

ainsi que tous ceux qui ont participé à l'étude utilisateur.

9. *La méthode du canard en plastique consiste à expliquer méticuleusement le code source que l'on a écrit à un collègue, à un simple passant, ou même à un objet inanimé comme un canard en plastique. Le simple fait d'exprimer ses pensées à voix haute est censé aider à trouver les erreurs de programmation. Comme les réactions de l'interlocuteur ou son niveau de compréhension du problème n'ont aucune importance dans ce processus, on peut le remplacer par un canard en plastique.* — Contenu soumis à la licence CC-BY-SA 3.0 (http://creativecommons.org/licenses/by-sa/3.0/deed.fr) Source : Article Méthode du canard en plastique de Wikipédia en français (http://fr.wikipedia.org/wiki/M%C3%A9thode_du_canard_en_plastique)

Conclusion

En conclusion, pendant ce stage, nous avons tenté de développé une interface permettant à des utilisateurs non-expérimentés de parcourir des scènes 3D. Ce système peut s'ouvrir sur d'autres problématiques : nous pourrions par exemple utiliser ce genre de recommandation 3D sur des systèmes qui représentent des scènes grâce à des images 2D (comme Google Maps par exemple).

Une autre optique qui pourrait être utile à la suite de ce projet est de générer les recommandations automatiquement à partir d'interactions d'utilisateurs. On pourrait penser à un algorithme qui détecte les zones que les utilisateurs aiment regarder, et placer une recommandation à cet endroit pour les futurs utilisateurs (ce qui a déjà été fait dans l'équipe, par Thi Phuong Nghiem en 2012).

Peu avant ce stage, nous avons eu le projet long qui nous a permis d'expérimenter les projets d'envergure en groupe, et la façon dont on devait les gérer.

Ce stage, pendant lequel j'ai développé le programme (*plus ou moins*) seul m'a permis de compléter les expériences que j'avais eu lors du projet long. En effet, la plupart des projets étudiants à l'ENSEEIHT sont de très petite taille (si l'on devait passer 35h par semaine sur ces projets, je crois qu'il n'y en aurait pas un qui durerait plus de trois jours), et on ne se rend en fait pas compte que certaines des techniques de gestion de projet que nous avons dû mettre en application pendant le projet long sont tout aussi importantes dans le cas d'un projet seul.

Dès nos premiers cours à l'ENSEEIHT, on nous enseignait l'importance des commentaires qui rendaient compréhensible un programme à quelqu'un d'autre, ou parfois, à soi-même mais dans le futur. Mais sur un projet de quelques jours, il est évident que l'on comprendra toujours ce qu'on a fait avant. Cependant dès que le projet devient suffisament important, on se rend compte qu'il est impossible de maîtriser parfaitement le code d'un bout à l'autre, et que c'est là que les commentaires prennent tout leur sens.

Ce projet a été très agréable pour moi, puisqu'il m'a permis de me consacrer à temps plein, et pendant une longue durée, à un programme qui part de zéro. Lors d'un projet en groupe, ou d'un sous-projet qui permet de constituer une brique d'un plus gros bâtiment, on peut ressentir la fierté d'avoir participé à un si grand projet, mais il y a aussi cette sensation qu'au final, l'impact que l'on a eu sur ce projet n'est pas si grande que ça. Lors d'un projet fait seul (ou en petite équipe), et qui part de zéro, on voit le programme grandir au fur et à mesure, ses fonctionnalités s'améliorer, ses bugs se corriger et de nouveaux apparaître, et regarder le projet une fois terminé est nettement plus flatteur pour l'ego.

Ce projet n'est en fait pas encore fini : je vais être embauché par l'IRIT en tant qu'ingénieur de recherche pendant encore quelques mois, le temps de finir les parties qui ne sont pas encore tout à fait finies, avec pour but une éventuelle soumission à la conférence MMSys, fin Novembre.