Modélisation et calcul parallèle pour le Web SIG 3D

Fabien Cellier

UNIVERSITÉ CLAUDE BERNARD – LYON
ÉCOLE DOCTORALE MATHINFO

T H È S E

pour obtenir le titre de
DOCTEUR EN SCIENCES
de l'Université Claude Bernard - Lyon I

MENTION : INFORMATIQUE

Présentée et soutenue par

FABIEN CELLIER

MODÉLISATION ET CALCUL PARALLELE

POUR LE WEB SIG 3D

Thèse dirigée par Samir Akkouche
préparée au LIRIS (UMR 5205 CNRS) en partenariat avec ATOS Worldline
soutenue le 31/01/2014

JURY :

Rapporteurs :
Marc Daniel, PU, Université d'Aix-Marseille
Nicolas Paparoditis, DR, MATIS-IGN

Examinateurs :
Karim Houni, Docteur, Groupe SEB
Bruno Raffin, CR1 HDR, INRIA Grenoble

Encadrants :
Samir Akkouche, PU, Université Lyon 1
Pierre-Marie Gandoin, MCU, Université Lyon 2

Encadrants (invités) :
Aurélien Barbier-Accary, Docteur, Worldline
Raphaëlle Chaine, PU, Université Lyon 1

Université Claude Bernard Lyon 1

worldline
e-payment services

Remerciements

Je tiens à remercier toutes les personnes m'ayant soutenu et encouragé pendant ces trois ans de recherches et qui ont, à leur manière, permis à ce travail d'aboutir.

Tout d'abord, je tiens à remercier le jury. Je remercie M. Marc Daniel et M. Nicolas Paparoditis pour avoir accepté de rapporter ma thèse. Enfin, merci à M. Karim Houni et M. Bruno Raffin d'avoir accepté d'examiner mon travail.

Je tiens également à remercier tous ceux qui m'ont encadré durant ces 3 ans, Samir Akkouche, Aurélien Barbier-Accary, Pierre-Marie Gandoin et Raphaëlle Chaine pour les conseils, les encouragements et les remarques qui ont permis d'améliorer mon travail.

Je souhaite aussi exprimer ma reconnaissance envers Jessica De La Cruz ainsi que mes collègues de Worldline qui m'ont épaulé et dont les retours ont contribué à achever mes travaux.

J'adresse également une pensée aux amis du mercredi pour leur soutien, les franches rigolades et les soirées interminables. Un grand merci à eux.

Enfin, j'aimerais remercier mes parents, pour m'avoir toujours soutenu, quels que soient mes projets.

Table des matières

Résumé

Cette thèse est centrée sur l'affichage et la manipulation en temps interactif au sein d'un navigateur Internet de modèles 3D issus de Systèmes d'Informations Géographiques (*SIG*). Ses principales contributions sont la visualisation de terrains 3D haute résolution, la simplification de maillages irréguliers sur *GPU*, et la création d'une nouvelle *API* navigateur permettant de réaliser des traitements lourds et efficaces (parallélisme *GP/GPU*) sans compromettre la sécurité.

La première approche proposée pour la visualisation de modèles de terrain s'appuie sur les récents efforts des navigateurs pour devenir une plateforme versatile. Grâce aux nouvelles *API* 3D sans plugin, nous avons pu créer un client de visualisation de terrains "*streamés*" à travers *HTTP*. Celui-ci s'intègre parfaitement dans les écosystèmes *Web-SIG* actuels (*desktop* et mobile) par l'utilisation des protocoles standards du domaine (fournis par l'*OGC*, *Open Geospatial Consortium*). Ce prototype s'inscrit dans le cadre des partenariats industriels entre *ATOS Worldline* et ses clients *SIG*, et notamment l'*IGN* (*institut national de l'information géographique et forestière*) avec le *Géoportail* (http://www.geoportail.gouv.fr) et ses *API* cartographiques. La 3D dans les navigateurs possède ses propres défis, qui sont différents de ce que l'on connaît des applications lourdes : aux problèmes de transfert de données s'ajoutent les restrictions et contraintes du *JavaScript*. Ces contraintes, détaillées dans le paragraphe suivant, nous ont poussé à repenser les algorithmes de référence de visualisation de terrain afin de prendre en compte les spécificités dues aux navigateurs. Ainsi, nous avons su profiter de la latence du réseau pour gérer dynamiquement les liaisons entre les parties du maillage sans impacter significativement la vitesse du rendu.

Au-delà de la visualisation 3D, et bien que le langage *JavaScript* autorise le parallélisme de tâches, le parallélisme de données reste quasi inexistant au sein des navigateurs *Web*. Ce constat, couplé à la faiblesse de traitement du *JavaScript*, constituait un frein majeur dans notre objectif de définir une plateforme *SIG* complète et performante intégrée au navigateur. C'est pour cette raison que nous avons conçu et développé, à travers les *WebCLWorkers*, une *API Web* de calcul *GP/GPU* haute performance répondant aux critères de simplicité et de sécurité inhérents au *Web*. Contrairement à l'existant, qui se base sur des codes déjà précompilés ou met de côté les performances, nous avons tenté de trouver le bon compromis pour avoir un langage proche du script mais sécurisé et performant, en utilisant les *API OpenCL* comme moteur d'exécution. Notre proposition d'*API* a intéressé la fondation *Mozilla* qui nous a ensuite demandé de participer à l'élaboration du standard *WebCL* dans la cadre du groupe *Khronos*, (aux côtés de *Mozilla* mais aussi de *Samsung*, *Nokia*, *Google*, *AMD*, etc.).

Grâce aux nouvelles ressources de calcul ainsi obtenues, nous avons alors proposé un algorithme de simplification parallèle de maillages irréguliers. Alors que l'état de l'art repose essentiellement sur des grilles régulières pour le parallélisme (hors *Web*) ou sur la simplification via *clusterisation* et *kd-tree*, aucune solution ne permettait d'avoir à la fois une simplification parallèle et des modèles intermédiaires utilisables pour la visualisation progressive en utilisant des grilles irrégulières. Notre solution repose sur un algorithme en trois étapes utilisant des priorités implicites et des minima locaux afin de réaliser la

simplification, et dont le degré de parallélisme est linéairement lié au nombre de points et de triangles du maillage à traiter.

Au cours de cette thèse, nous avons donc mis en place une approche innovante pour la visualisation 3D *Web-SIG* sans *plugin*, en concevant des outils capables de conférer au navigateur une confortable puissance de calcul parallèle *GP/GPU*, et en proposant une méthode de simplification parallèle de maillages irréguliers permettant un affichage en niveaux de détails directement dans les navigateurs *Web*. Sur la base de ces premiers résultats, il devient possible de porter toute la richesse fonctionnelle des clients *SIG* sur *desktop* au sein des navigateurs *Web*, aussi bien sur *PC* que mobiles ou tablettes.

1. Introduction

Contenu

1.1. L'évolution des usages

Cette thèse traite de trois domaines différents : la modélisation 3D, les Systèmes d'Information Géographique (SIG) et les technologies du Web. Il nous semble intéressant de regarder l'histoire des deux premiers champs de recherche et la façon dont le troisième les a enrichis.

A notre époque, la 3D paraît être une évolution technologique récente. Cependant, le principe de glyphe (lunettes rouge et verte) date du XIX^{ème} siècle et servait initialement à donner du relief à certaines photographies. Il en est de même pour le principe des lunettes polarisées (technologie utilisée dans les cinémas) dont le brevet fut déposé en 1893. Le principe de peinture en perspective date, quant à lui, de la Renaissance, en se fondant sur la géométrie projective, considérée par certains comme le premier pas vers l'infographie tridimensionnelle. Alors qu'au début des années 1990, les ordinateurs amorçaient leur conquête des espaces professionnels et des foyers, il n'était alors possible d'afficher que quelques milliers de triangles seulement. C'est avec l'apparition de matériel spécialement conçu pour la représentation de modèles 3D que la visualisation est véritablement entrée dans l'esprit du grand public. Paradoxalement, malgré la puissance grandissante de ces nouvelles puces de calcul à mémoire embarquée, la conception de modèles de stockage et de visualisation est restée primordiale pour créer des mondes complets, du stockage de la géométrie et de la topologie jusqu'au transfert à partir d'une mémoire lente, qui peut être le disque dur ou le réseau, vers la carte graphique, tout en filtrant les données pertinentes afin de permettre à celle-ci de traiter les informations en un temps restreint compatible avec l'affichage.

Introduction

Bien que l'affichage d'un triangle, d'un cube en rotation, voire même d'un modèle 3D simple ne représente plus aucun challenge que ce soit pour la 3D ou le Web 3D, il n'en va pas de même pour des objets complexes dont la donnée peut représenter parfois plusieurs téraoctets. Dans ce cas, la géométrie algorithmique fournit des outils indispensables pour le partitionnement, la récupération, le *streaming* des données (dans les cas de stockage distant) ou leur mise à jour. La preuve en est l'utilisation intensive des octrees, ondelettes ou triangulations de Delaunay par exemple dans les solutions logicielles existantes.

Tout comme les travaux sur la 3D, les systèmes d'information géographique (SIG) possèdent une histoire remontant au XIX^{ème} siècle. La première application était française, avec une étude de Charles Picquet[1], réalisée en 1832 sur la dispersion du choléra dans Paris. Un an plus tard, Jon Snow utilisa un principe similaire et trouva le puits source de l'épidémie de choléra à Londres.

En 1960, le Canada utilisa des ordinateurs afin de stocker et manipuler des données représentant le monde qui nous entoure. De par la nature même des SIG qui répertorient ces données et cherchent à fournir des solutions de visualisation, il est naturel que les modèles classiques de la géométrie ait été progressivement intégrés à l'informatique. Pendant des décennies, les météorologues et géophysiciens ont utilisé des caractéristiques de la distribution d'informations géographiques pour leurs modèles et leurs prédictions. Ainsi, les modèles 3D ont été intégrés de façon naturelle, et c'est durant la maturation de la géographie dans le domaine de l'informatique dans le milieu des années 1990 que les SIG ont permis de croiser les données entre elles dans le but de trouver des coïncidences spatiales par juxtaposition.

Comme nous venons de le voir, les SIG ont pour but de répertorier, d'analyser et de visualiser le monde qui nous entoure. Il est donc logique que la modélisation géométrique soit un enjeu majeur pour le domaine tout comme l'infographie 3D.

Enfin, les technologies Internet, contrairement aux deux champs précédents, ne constitue pas seulement un domaine à part entière mais une évolution fondamentale pour l'ensemble des applications informatiques, en offrant aujourd'hui au grand public un accès rapide à une information détaillée. Alors que le SIG offre le stockage et la diffusion de l'information en s'appuyant sur la modélisation 3D, il était naturel qu'Internet complète ces services en offrant une diffusion à grande échelle des informations à travers le réseau.

C'est dans ce contexte que l'Open Geospatial Consortium[2] (OGC) a été créé, afin de standardiser les échanges des données géolocalisées.

[1] http://fr.wikipedia.org/wiki/Syst%C3%A8me_d'information_g%C3%A9ographique#cite_note-1
[2] http://www.opengeospatial.org/

Ces trois domaines de recherche reposent sur l'organisation de l'information en modèles et leur évolution est intimement liée aux capacités des matériels informatiques disponibles. Ainsi, puisque les architectures logicielles tendent à être de plus en plus massivement parallèles, il est naturel que les modèles et algorithmes qui supportent la visualisation 3D et les systèmes d'information évoluent eux aussi pour utiliser ces nouvelles ressources que sont le réseau ou les cartes graphiques pour la programmation générique.

1.2. Le contexte et la problématique

Le SIG 2D est présent sur Internet au travers des portails Web spécialisés dans la visualisation tels que Google Map[3] ou encore le Géoportail[4] qui est un portail de visualisation de cartes de l'IGN (Institut national de l'information géographique et forestière). Alors que le premier est un outil grand public de consultation, le second a pour vocation de devenir le socle de véritables systèmes d'information géographique complets en ligne. Moins de six mois après la sortie en 2007 de la deuxième version, de nombreuses informations étaient disponibles, telles que le cadastre ou encore l'hydrographie.

En 2007, dans le cadre de la réalisation du Géoportail v2, Worldline a cherché de nouvelles solutions pour la visualisation de données 3D, sans pour autant dénaturer le projet d'origine qui consistait initialement à proposer la visualisation des données de l'IGN dans un portail Web. De nombreux travaux sur la visualisation de terrains ont déjà été menés (*Clipmap* [LOSASSO ET AL. 2004], BDAM [CIGNONI ET AL. 2003] et variantes), et cependant, très peu de chercheurs avaient abordé le sujet du point de vue d'une application Web plongée dans un navigateur. De plus, le Géoportail est un site grand public qui se doit d'avoir une grande réactivité, et ce avec un nombre de serveurs limité afin de maîtriser les coûts de mise en place. Ainsi, les solutions développées doivent être adaptées à un transfert à partir de fichiers statiques ou tout du moins avec le moins de calculs possible coté serveur. Les consommations de bande passante doivent également être raisonnables pour que l'outil reste accessible au plus grand nombre. Enfin, l'application doit pouvoir s'adapter à la puissance du client de visualisation et à la bande passante, et doit autoriser l'ajout de nouveaux serveurs facilement. A l'heure où nous écrivons ces lignes, certains boîtiers Internet de fournisseurs d'accès et certains constructeurs tentent déjà de déporter tout le calcul côté serveur (images 3D, simulations physiques, etc.). Cette méthode possède le défaut majeur de nécessiter une importante puissance du côté du Cloud, qui impose la souscription d'un forfait par l'utilisateur final pour financer le *datacenter*. Par conséquent, les méthodes basées sur des calculs réalisés sur le serveur avant envoi aux clients ne répondaient ni au critère de bande passante pour les utilisateurs ayant une connectivité limitée, ni à la légèreté des serveurs pour la maîtrise des coûts.

[3] https://maps.google.fr/
[4] http://www.geoportail.gouv.fr/accueil

En 2009, une nouvelle technologie 3D est apparue : le *WebGL*[5], adaptation de l'OpenGL spécialement conçue pour les navigateurs et offrant de nouvelles possibilités. Néanmoins, cette technologie possède aussi ses limites et celles-ci doivent être prises en compte. Par exemple le *WebGL* limite le nombre de triangles, de textures et de fonctionnalités, afin de pouvoir être exécuté sur des appareils mobiles tels que les Smartphones ou les ordinateurs de bureau sans carte graphique avancée. De plus, cette technologie doit être utilisée conjointement au langage *JavaScript* qui possède lui aussi d'importantes limitations (pauses dues à la gestion de mémoire automatique, gestion des flottants non homogène, certains traitements de tableaux jusqu'à 200 fois moins rapides qu'en C, exécution « monothreadée », etc.). Enfin, l'utilisation de l'application dans un navigateur implique, pour des raisons de sécurité, de ne connaître que très peu d'informations à propos du matériel. Cela complique encore davantage l'adaptation de la qualité de rendu au client de visualisation, dont la puissance et les périphériques connectés peuvent varier significativement.

Dans le même temps, durant les dernières années, de nouvelles API et technologies sont apparues, le plus souvent sous la bannière de *HTML5*, provoquant une réduction considérable de l'écart entre les applications pour clients lourds (desktop) et celles pour clients légers (navigateurs Web). Pour autant, la puissance disponible dans le navigateur ne permet toujours pas d'envisager un SIG en ligne – ou WebSIG – complet. Pour s'en convaincre, il suffit d'essayer d'implémenter quelques cas d'utilisation classiques : à l'heure actuelle, il est impossible d'effectuer, directement dans le navigateur, un test de collisions en temps réel sur des centaines d'objets, ou de simuler une interaction de particules (n-body), ni même de visualiser un objet en adaptant le niveau de détail en temps réel en fonction du point de vue adopté. Il est par conséquent impossible d'envisager dans ces conditions un WebSIG avancé reposant uniquement sur des technologies Web.

1.3. Nos contributions

Ce n'est qu'après plusieurs mois de travail sur cette thèse que le *WebGL* est apparu, et sa publication changea radicalement nos hypothèses : en effet il permettait de passer de quelques dizaines de milliers de triangles sur une « rastérisation » manuelle à plusieurs centaines de milliers ou parfois quelques millions de triangles, avec en outre un support automatique des textures, et ce, même sur les smartphones de moyenne gamme. Nous avons donc étudié ces nouvelles possibilités techniques en parallèle de notre bibliographie sur les méthodes de visualisation 3D. Cette « double » étude a débouché sur la création d'*OpenScalesGL* [CELLIER 2012], une bibliothèque *JavaScript* de visualisation 3D SIG fonctionnant sur le navigateur d'un ordinateur comme celui d'un smartphone et ce avec un unique code source. Cet outil repose sur l'adaptation de la méthode des *Geometry Clipmaps*

[5] http://www.khronos.org/webgl/

[LOSASSO ET AL. 2004] et exploite les nouvelles possibilités offertes par le WebGL de *HTML5*. De plus, les protocoles utilisés sont les flux classiques du SIG pour la 2D, ce qui permet de ne demander aucun développement supplémentaire côté serveur, et de partager les données entre le portail 2D et sa version 3D. Contrairement à l'existant, nous avons pris en compte les particularités de la plate-forme – c'est-à-dire principalement les faiblesses du *JavaScript* dans le transfert de données et son caractère asynchrone – pour gérer la liaison des tuiles de façon dynamique.

Après cette première contribution, nous avons souhaité aller plus loin dans la qualité de rendu 3D des terrains en développant de nouveaux algorithmes pour la visualisation SIG au sein des navigateurs Web qui corrigeraient les défauts des *Geometry Clipmaps* [LOSASSO ET AL. 2004]. Dans le cadre de cette démarche, nous nous sommes heurté au manque de vélocité du *JavaScript*, aux pauses du *Garbage collector* et à la consommation mémoire, toutes trois excessives pour l'implémentation des meilleurs algorithmes de visualisation 3D, et limitant ainsi grandement nos possibilités. Nous avons donc décidé, en 2011, d'effectuer des travaux sur le calcul parallèle GP/GPU dans le navigateur. Nous avons ainsi proposé les WebCLWorkers , une API qui autorise l'utilisation du GPU pour des calculs génériques dans un navigateur Internet, sans en compromettre la sécurité. Dans certains cas, la vitesse d'exécution était jusqu'à 100 fois supérieure à celle du *JavaScript*. C'est grâce à ces travaux que nous avons été remarqué par Mozilla et que nous participons depuis au comité de standardisation du GP/GPU pour le navigateur au sein du consortium *Khronos*. Constitué, entre autres, de Mozilla, Samsung, Nokia, Qualcomm et AMD, ce consortium est connu notamment pour ses standards tels qu'*OpenCL* ou OpenGL. Le futur standard de GP/GPU dans les navigateurs s'appellera WebCL, à ne pas confondre avec le WebGL qui lui est dédié à la 3D.

Afin de profiter pleinement de la nouvelle puissance disponible dans le navigateur grâce aux WebCLWorkers , il est apparu nécessaire de paralléliser le plus possible les traitements. Toujours dans notre optique d'obtenir l'équivalent d'une visualisation SIG de qualité au sein d'un navigateur, nous avons alors élaboré un algorithme de simplification de modèles 3D dont le degré de parallélisme évolue linéairement avec le nombre de points et de triangles dans le maillage. De plus, ce principe de simplification est compatible avec les principes de partitionnement des méthodes BDAM [CIGNONI ET AL. 2003], ainsi qu'avec un *streaming* évolutif. Enfin, jusqu'à présent, aucun algorithme ne permettait à la fois une exécution « multithreadée », une simplification sur des grilles irrégulières et un *streaming* semi-continu, autrement dit, avec des niveaux de détails intermédiaires durant le transfert. Nous avons donc proposé une approche qui répond à tous ces critères et qui, de plus, ne nécessite aucun verrou logiciel, au sens mutex du parallélisme.

1.4. Construction de ce mémoire

Ce document est construit en trois parties contenant chacune un état de l'art. Dans un premier temps nous évoquerons les travaux effectués sur la visualisation de terrain 3D qui se basent sur les technologies déjà présentes dans les navigateurs et considérées aujourd'hui comme stables. Puis, à partir des manques que nous avons observés, nous traiterons la conception et le développement de nouvelles fonctionnalités augmentant la puissance des navigateurs. Enfin, nous terminerons par une méthode de simplification de maillages irréguliers exploitant la puissance des GPU mais aussi, plus généralement, le parallélisme des nouvelles unités de calculs CPU et GPU à travers un algorithme dit « lock free ».

2. Chapitre 1 : WebSIG 3D sans plugin : OpenScalesGL

Contenu

Dans cette première partie, nous commencerons par une description du contexte ainsi qu'une brève explication de l'architecture des cartes graphiques, avant de nous intéresser à l'état de l'art. En effet, la connaissance des restrictions technologiques actuelles et des contraintes imposées par le contexte applicatif permettent de mieux comprendre l'orientation de nos recherches et nos choix de développement.

2.1. Contexte

2.1.1. Contexte industriel

Géoportail est un portail Web permettant d'accéder à un très large ensemble des données géographiques du territoire français. Depuis octobre 2006, Worldline réalise et héberge les versions 2 et 3 du Géoportail pour le compte de l'IGN. La 3D a été proposée pour la première fois à l'été 2007 sur la base du logiciel Terra Explorer, inclus dans le navigateur à l'aide d'un plugin spécifique.

En 2009, Wordline et l'équipe GEOMOD du LIRIS ont choisi de mettre leurs compétences en commun afin d'imaginer le client SIG du futur pour la visualisation et la manipulation de modèles 3D dans le navigateur, et plus particulièrement le stockage, le *streaming* et la visualisation de terrain à partir d'un simple site Internet et le tout sans plugin. C'est dans ce contexte collaboratif que la thèse CIFRE développée dans ce mémoire a été initiée.

Parallèlement, l'appel d'offre pour la version 3 du Géoportail étant gagné début janvier 2011, Worldline se lance dans sa réalisation début février. La phase de réalisation concerne durant un an et demi une trentaine de personnes au sein de la « Business Unit » PST (Secteur Public et Transports) de Worldline, ainsi que quatre entreprises partenaires. Le projet est découpé en sous-parties et six équipes sont constituées pour chacune de ces sous-parties :

- le sous-système Portail regroupe les différents portails, les points d'accès privilégiés aux données, ainsi que les services de la plate-forme ;
- le sous-système API clientes regroupe l'ensemble des API clientes graphiques permettant la visualisation des données du Géoportail : API 2D (*JavaScript* et *Flash*), API 3D (*JavaScript*) ainsi que l'API d'accès aux services permettant l'encapsulation, d'un point de vue non graphique, des couches de communication avec les services de diffusion de la plate-forme ;
- le sous-système Contrôle des Accès regroupe les mécanismes d'identification, les autorisations, les alertes et les statistiques, mises en œuvre à chaque utilisation des API et services ;
- le sous-système Services regroupe l'ensemble des services de diffusion de données de la plate-forme, ainsi que les services orientés traitements. Ces services s'appuient sur des services communs transversaux tels que les services de découverte, de catalogage et d'auto-configuration ;
- le sous-système Entrepôt regroupe les problématiques de stockage, la gestion des traitements métiers au travers d'un orchestrateur ainsi que les mécanismes de publication et de transfert des résultats de ces traitements au sein de la plate-forme Géoportail ;

- Le sous-système Forge Kazan® s'occupe de la gestion des développements et déploiements, de l'environnement de développement et du portail.

L'ensemble du projet Géoportail 3 (appelée aussi GPP3) est sous la responsabilité d'Aurélien BARBIER-ACCARY, qui co-encadre cette thèse.

Pour développer cette troisième version du Géoportail, Worldline fait notamment appel à l'entreprise Diginext pour s'occuper de la partie 3D de la visualisation sur la base de son logiciel Virtual Geo, là encore avec une approche de client lourd intégré dans le navigateur via un plugin.

Durant cette thèse, nous nous sommes assuré que les choix technologiques utilisés dans le prototype seraient compatibles avec les futures possibilités des navigateurs. Ainsi, dans le cadre du Géoportail, il ne sera plus nécessaire d'installer un programme pour pouvoir profiter de toute la richesse des données du portail : la cartographie 3D sera parfaitement intégrée au Web sans interruption de l'expérience utilisateur.

2.1.2. Contexte technologique : de 2009 à 2013, le Web gagne la troisième dimension

Comme nous le verrons tout au long du mémoire, le Web est en constante évolution. Ces évolutions peuvent paraître très lentes d'un point de vue développeur, demandant parfois jusqu'à plusieurs mois ou années pour l'émergence et l'utilisation de nouvelles technologies, mais elles paraissent très rapides pour les entreprises et la recherche, car certaines pistes peuvent être complètement balayées en moins d'un an par l'apparition d'une nouvelle fonctionnalité ou API. Par ailleurs le véritable métronome des capacités du Web est en fait l'adoption, qui doit être très majoritaire, par le grand public, des versions des navigateurs qui permettent les nouveaux usages.

L'intérêt pour la 3D dans Internet n'est pas récent : c'est en 1990 que la technologie VRML (*Virtual Reality Modelling Language*) a été conçue et mise en avant. Grâce à cette dernière, il était possible non seulement d'afficher des objets 3D, mais aussi d'interagir avec l'utilisateur ou encore de gérer les animations. En tant que spécification, le VRML n'est qu'un fichier de description, au même titre que le HTML. La visualisation d'un modèle VRML nécessitait l'installation d'un plugin navigateur. A l'époque, la consommation de bande passante n'était pas adaptée à la taille du réseau (56kbits). Comme nous le verrons un peu plus loin, même si aujourd'hui le réseau n'est plus véritablement un facteur limitant, et que la compression intégrée au HTTP permet d'avoir des résultats convenables pour les fichiers WRL (fichier texte descriptif du VRML), notre vision mais également notre objectif était de pouvoir travailler sur le plus d'informations topologiques et géométriques possible pour savoir si, à terme, nous pourrions envisager la réalisation d'un logiciel SIG complet dans le navigateur.

En 2009, la 3D dans les navigateurs[6] nécessite un plugin tel qu'Unity[7] ou encore o3d[8]. Bien que quelques tests aient été réalisés afin d'avoir un rendu 3D sans passer par le GPU, les contraintes étaient telles que seuls 10 000 triangles texturés pouvaient être affichés à l'écran. Les meilleurs résultats étaient obtenus grâce à Flash qui intégrait des effets 3D dans sa version 10.0 (datant d'octobre 2008).

Les limitations des capacités 3D n'étaient pas le seul problème à résoudre : en effet, le *JavaScript* n'était pas aussi optimisé qu'aujourd'hui, et ses performances catastrophiques limitaient encore davantage les possibilités d'affichage 3D disponibles. Le simple parcours d'un arbre binaire était, dans ce cas précis, 50 fois plus lent qu'en C. Encore aujourd'hui, avec toutes les optimisations récentes, le *JavaScript* reste, dans le meilleur des cas 15 fois plus lent que le C (sans compter une consommation mémoire 3 fois supérieure en moyenne) sur des algorithmes classiques tels que le « k-nucleotide » (une comparaison de sous-séquences d'ADN), tout en ajoutant des pauses non prévisibles dues au « *garbage collector* ». Enfin, il n'existait en 2009 aucune solution 3D de rendu convenable en *JavaScript*.

En ce qui concerne le Flash, lors de notre premier prototype qui avait pour but de créer un maillage à partir d'un nuage de points en utilisant une triangulation implicite de Delaunay, nous nous sommes aperçu que la construction prenait plus de 1400 ms pour 3000 points avec l'algorithme de Ruppert[9], ce qui n'était pas acceptable en pratique : en effet, jusqu'en 2012, le Flash est « monothreadé » et tout traitement doit se terminer avant que l'affichage ne soit réalisé. Le fait d'avoir des pauses de plusieurs secondes lors des mises à jour du maillage pour quelques milliers de triangles n'était donc pas réaliste au moment où les dernières cartes graphiques étaient capables d'afficher plusieurs millions de triangles, y compris celles des Smartphones.

Dans le même temps, fin 2009, le consortium *Khronos* (Mozilla, Samsung, Nokia, Qualcomm et AMD, etc.) annonçait publiquement le début de son travail sur le standard *WebGL*. Néanmoins, aucune indication ne permettait de prédire le temps nécessaire à l'intégration de cette future technologie dans les principaux navigateurs. Les premières implémentations publiques n'ont été intégrées dans certains navigateurs qu'à partir d'octobre 2010 et les premiers pas officiels dans des navigateurs n'ont été réalisés qu'à partir de 2011. A ce jour, Internet Explorer de Microsoft ne supporte toujours pas cette technologie et ne commencera à le faire que dans sa future version 11. Devant l'incertitude autour de l'adoption de *WebGL*, nous avons préféré étudier toutes les solutions possibles, comme le prototype à base de Delaunay cité précédemment. Ces tests incluaient la représentation 3D classique avec les cartes graphiques aux rastérisations manuelles. De

[6] http://fr.wikipedia.org/wiki/WebGL#Utilisations
[7] http://unity3d.com/
[8] http://en.wikipedia.org/wiki/O3D
[9] Ruppert, J. (1995). A Delaunay refinement algorithm for quality 2-dimensional mesh generation. *Journal of algorithms*, *18*(3), 548-585.

même, il a fallu attendre mi-2010 pour enfin avoir la possibilité de manipuler des binaires dans les navigateurs avec *JavaScript* (jusque-là, seuls les fichiers textes pouvait être utilisés). Cette dernière évolution était un prérequis important pour la compression des données, mais aussi pour traiter efficacement des tableaux de données. En effet, avant l'apparition des « typedArray[10] », tout objet *JavaScript* ou tableau était simplement un tableau associatif dont l'identifiant pointait vers la valeur grâce à une clef de hachage. Ce mécanisme de stockage, bien que très puissant, avait un impact négatif important sur les performances du langage.

Ce constat d'évolution rapide d'Internet n'est pas limité à la 3D. Avec toutes les nouvelles API de l'*HTML5* qui sont en cours de standardisation comme par exemple le WebRTC, le navigateur est passé en quelques années d'un simple contexte d'exécution pour le langage *JavaScript* à un écosystème complet que l'on pourrait comparer à un système d'exploitation sommaire.

2.2. Fondamentaux des domaines Web, géographique et 3D

Comme mentionné précédemment, l'un de nos objectifs était de valider la possibilité de créer un client de visualisation 3D pour le SIG dans un navigateur Web, et ce sans plugin. Notre proposition se concrétisera au travers du projet OpenScalesGL [CELLIER 2012]. Néanmoins, avant d'aborder l'état de l'art puis notre contribution, il nous semble important de rappeler les fondements des différents domaines impliqués, tout au moins pour les parties utilisées comme socle de nos travaux.

2.2.1. Concepts fondamentaux du Web

Le « Cloud Computing » permet de déporter vers des serveurs distants des données ou des traitements auparavant présents sur les postes utilisateurs locaux, en utilisant notamment l'*HTML5* qui propose des nouvelles API et une autre utilisation du DOM[11].

Les langages tels que Java ou C++ sont toujours plus performants que les moteurs *JavaScript* des navigateurs. C'est pourquoi l'utilisation du Framework *WebGL*, portage d'OpenGL, proposant des performances similaires aux API disponibles dans les applications natives, permet d'apporter un complément intéressant. Ainsi, l'association *HMTL5/WebGL* permet la création d'applications Web présentant des fonctions similaires aux programmes classiques.

[10] http://www.Khronos.org/registry/typedarray/specs/latest/

[11] Le DOM (Document Object Model) permet aux programmes et scripts d'accéder dynamiquement au contenu, à la structure et au style d'un documents XML ou HTML, pour lecture ou modification.

OpenGL (Open Graphics Library) est une API utilisée pour la conception d'applications générant des images 2D et 3D. OpenGL permet notamment la gestion de la caméra, la rotation 3D des objets, le remplissage des faces, les textures, la lumière …
Une version nommée OpenGL ES a été conçue spécifiquement pour les applications embarquées. *WebGL* est un dérivé de cette version embarquée d'OpenGL, destinée aux navigateurs Web. Cela permet d'utiliser le standard OpenGL depuis le code *JavaScript* d'une page Web. On peut voir le lien entre ces trois versions d'OpenGL sur la figure suivante.

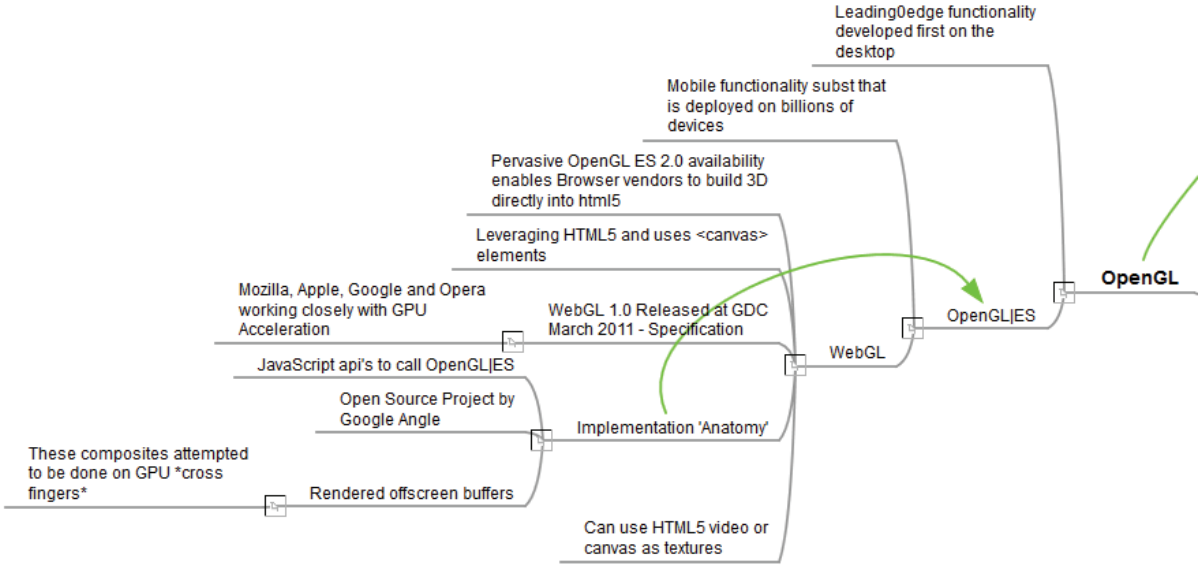


Figure 1- OpenGL, OpenGL/ES et WebGL

2.2.2. Concepts fondamentaux en transmission des données géographiques 2D et 3D

Pour les besoins de réalisation du client de visualisation 3D, l'IGN (Institut national de l'information géographique et forestière) a mis à notre disposition des photographies aériennes et des MNT (Modèles Numériques de Terrain). Ces données sont géo-référencées et accessibles par des protocoles de communication géographiques standards tels que WMS ou WMTS. Ces protocoles suivent des normes et des technologies précises, définies par l'Open Geospatial Consortium (OGC).

Avant de pouvoir afficher de la 3D, il est nécessaire d'interpréter les données fournies pour obtenir des élévations et des textures. En effet, l'IGN met à disposition des données géographiques dans plusieurs protocoles et selon plusieurs projections. Ces projections (passage d'une donnée 3D à une donnée 2D) engendrent des déformations contrôlées. A noter que dans le cadre d'OpenGL, il est aussi nécessaire d'effectuer cette conversion à travers une matrice de reprojection.

2.2.2.1. Représention de la terre : ellipsoïde de référence

Dans le domaine des SIG[12], l'un des objectifs importants de la cartographie est de représenter des éléments de la réalité avec le plus d'exactitude possible. Par nature, la surface terrestre (géoïde) est une forme géométriquement imparfaite. Elle est déformée, à cause de l'inégale répartition des masses à la surface de la Terre. Sa géométrie est donc complexe et ne peut être formulée mathématiquement de façon simple. De ce fait, on utilise l'ellipsoïde, qui est une surface géométrique permettant de représenter assez fidèlement la forme du géoïde. En cartographie, on utilise les systèmes géodésiques. Ceux-ci reposent sur deux types de référence : l'ellipsoïde de référence et le système de coordonnées géographiques. Pour représenter une portion de la surface terrestre, on distingue l'ellipsoïde global de l'ellipsoïde local, comme on peut le voir sur la figure 2 ci-dessous.

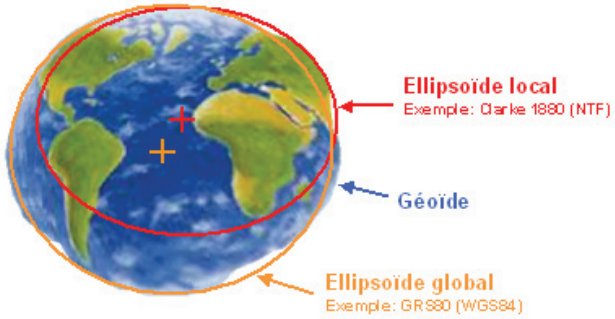


Figure 2- Représentation des deux types d'ellipsoïdes

Les ellipsoïdes globaux, dépendants des projections, sont utilisés pour des cartographies couvrant l'ensemble (ou une grande partie) de la surface terrestre. C'est le cas du système de positionnement GPS qui fournit, par défaut, le positionnement en coordonnées longitude/latitude.

Les ellipsoïdes locaux sont définis, quant à eux, de manière à "épouser" au mieux la forme du géoïde sur une zone restreinte de la surface terrestre (une région ou un pays). Dans ces cas-là, afin de mieux suivre la forme du géoïde, la forme de l'ellipsoïde local est modifiée et son centre est décalé par rapport au centre des ellipsoïdes globaux.

[12] Un système d'information géographique (SIG) est un système d'information permettant de créer, d'organiser et de présenter des données alphanumériques spatialement référencées, ainsi que de produire des plans et des cartes.

2.2.2.2. Système de coordonnées géographique : Latitude/Longitude

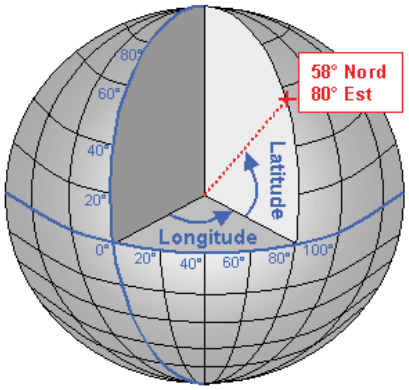


Figure 3 – Système de coordonnées géographiques

La localisation d'un élément à la surface de la Terre peut s'exprimer sous la forme de coordonnées géographiques. Les coordonnées sont alors déclinées à l'aide de deux valeurs angulaires : longitude et latitude. Ces deux angles peuvent être exprimés dans différentes unités mais les plus courantes sont les degrés décimaux et sexagésimaux.

Dans le système sphérique, les "lignes horizontales", sont des lignes de latitude égale ou des parallèles. La ligne de latitude qui sépare les pôles est appelée l'équateur.
Les "lignes verticales", sont des lignes de longitude égale ou des méridiens. La ligne de longitude zéro est appelée méridien principal.
Ces lignes ceinturent le globe et constituent un réseau quadrillé appelé un graticule.
L'origine de ce graticule (0,0) est définie d'après le point d'intersection de l'équateur et du méridien principal.

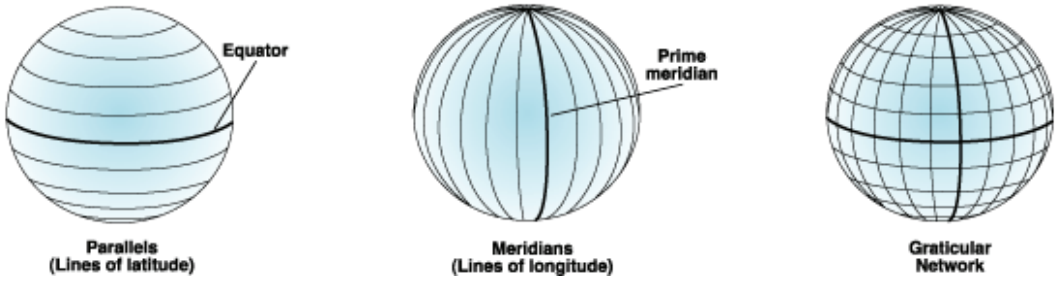


Figure 4 – Parallèles et méridiens constituant un graticule

Pour notre preuve de concept, nous avons décidé de représenter l'île de la Réunion. En effet, cela nous permet de travailler avec des données de taille adaptée, ni trop restreinte pour la démonstration, ni trop grande afin de pouvoir représenter la zone dans son intégralité. La Réunion présente également l'avantage de posséder des plaines et des reliefs montagneux variés, ce qui est très intéressant pour les problématiques liées au rendu 3D. Cette île ne possédant pas son propre ellipsoïde local, nous utilisons l'ellipsoïde global et le système géodésique WGS84.

Si on étale la Terre, on utilisera les coordonnées (exprimées en degrés décimaux) suivantes pour récupérer les données :

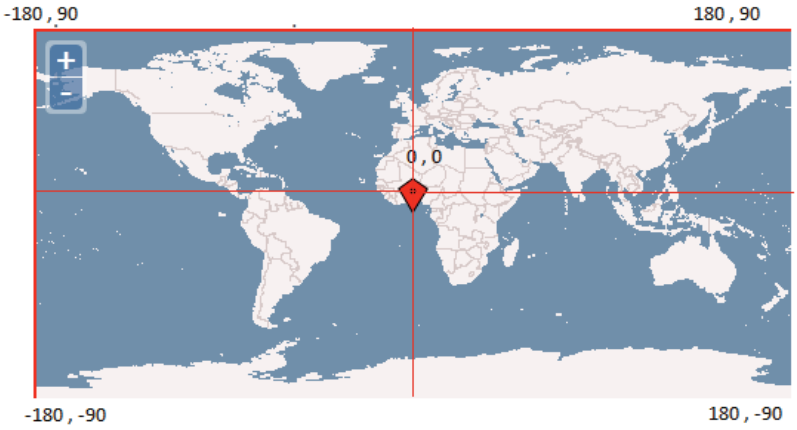


Figure 5 - Latitudes et longitudes dans le monde

2.2.2.3. Protocoles de communication géographiques standards WMS et WMTS

Il existe plusieurs protocoles de communication qui permettent d'obtenir des fonds de plan et des terrains rastérisés à partir de différents serveurs de données géolocalisées. Worldline souhaitait une démonstration fonctionnant avec les protocoles WMS et WMTS qui sont des standards décrits dans les spécifications maintenues par l'Open Geospatial Consortium. Le mode de récupération des données change d'un protocole à l'autre.

Les services WMS peuvent être appelés en utilisant un navigateur Web standard, au travers de demandes directement intégrées dans l'URL. Les informations contenues dans cet URL dépendent de l'opération que l'on veut effectuer. Dans notre cas, nous nous intéressons à l'opération « GetMap », qui permet au service WMS de renvoyer une image rastérisée ou bien une image vectorielle au format SVG. Dans cet URL, il faut indiquer :

- Le nom de la couche que l'on souhaite requêter (elle dépend des informations que l'on veut obtenir : routes, cadastres, photos aériennes, etc.)
- Le système de coordonnées utilisé : dans notre cas, il s'agit du WSG84
- La taille de l'image demandée en pixels
- L'emprise géographique désirée en degrés décimaux. Il faut indiquer les coordonnées du coin inférieur gauche (latitude 0, longitude 0) puis celles du coin supérieur droit (latitude 1, longitude 1). On nommera ces coordonnées respectivement (x0,y0) et (x1,y1). Ainsi, par exemple, pour récupérer une image représentant une vue éloignée de l'île de La Réunion comme celle de l'image ci-contre, on indiquera les coordonnées suivantes : 55.0,-21.5, 56.0,-20.5.

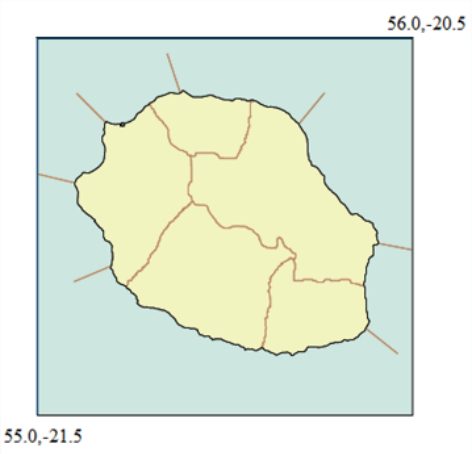


Figure 6 - Emprise de la Réunion

La norme WMTS est proche de celle du WMS, mais au lieu de générer une nouvelle image à chaque déplacement, comme dans le cas du WMS, le WMTS utilise des couches d'images multi-échelles et pré-calculées. La carte est découpée selon une grille prédéfinie pour apporter plus de performances et moins de requêtes réseau. Pour ce protocole-là, on utilisera l'opération « GetTiles » et on ne transmettra pas l'emprise mais le niveau de résolution, ainsi que la colonne et la ligne de la tuile (image) souhaitée. Ci-dessous, on trouve une pyramide d'images avec différents niveaux de résolution (figure 7), et pour chaque niveau de résolution, il y a un nombre minimum et maximum de lignes et de colonnes (figure 8).

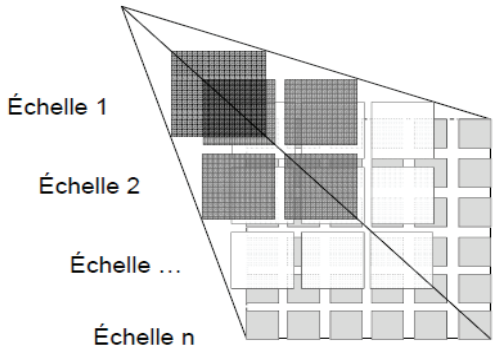


Figure 7 - Pyramide d'images

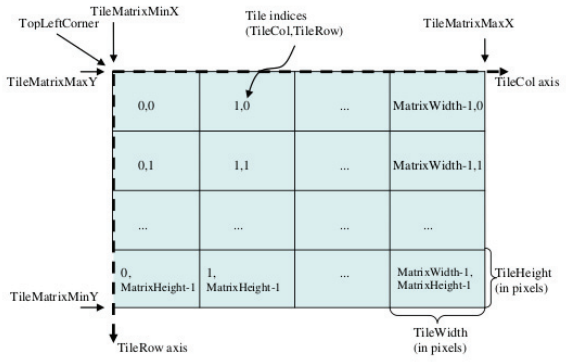


Figure 8 - Tuiles pour un niveau de résolution donné

Afin de récupérer les données images et les élévations (MNT), on utilisera donc l'un des deux protocoles WMS ou WMTS. Il y aura donc deux codes de requêtage différents. Cependant, afin de ne pas devoir modifier une partie trop importante du code, on transmettra uniquement une emprise (*bounding box* ou *bbox*) à la classe de requêtage, et on

recevra dans tous les cas une image correspondant à l'emprise. En WMS, la construction de l'URL sera donc simple, car il suffira simplement de changer les coordonnées de l'emprise pour chaque image. En revanche pour le protocole WMTS, il faudra auparavant calculer le niveau de résolution, la colonne et la ligne de l'image associée à l'emprise désirée. Ces calculs se font à l'aide de la taille de l'image et des niveaux de résolution disponibles selon la projection utilisée. Ces calculs ne seront pas détaillés ici car ils relèvent du domaine du SIG pur, nous les utilisons sous leur forme standard, et ils sont donc éloignés de notre problématique de recherche.

2.2.3. Concepts fondamentaux pour l'affichage temps réel en 3D

Afin d'obtenir des performances convenables pour notre prototype, nous avons dû minimiser les appels au *JavaScript* et déléguer le plus de calculs possible au GPU (carte graphique). Ainsi, avant d'expliquer le fonctionnement d'*OpenScalesGL* [CELLIER 2012], une rapide présentation du fonctionnement de la 3D est une étape nécessaire pour faciliter la compréhension.

2.2.3.1. Le pipeline graphique

La plupart de ce processus se passe au sein d'un GPU[13]. Ci-dessous se trouve un schéma du pipeline graphique qui représente la succession des opérations réalisées par une carte graphique.

[13] GPU (*Graphics Processing Unit*) est un circuit intégré présent sur une carte graphique et assurant les fonctions de calcul de l'affichage. Un processeur graphique a généralement une structure hautement parallèle qui le rend efficace pour une large palette de tâches graphiques comme le rendu 3D.

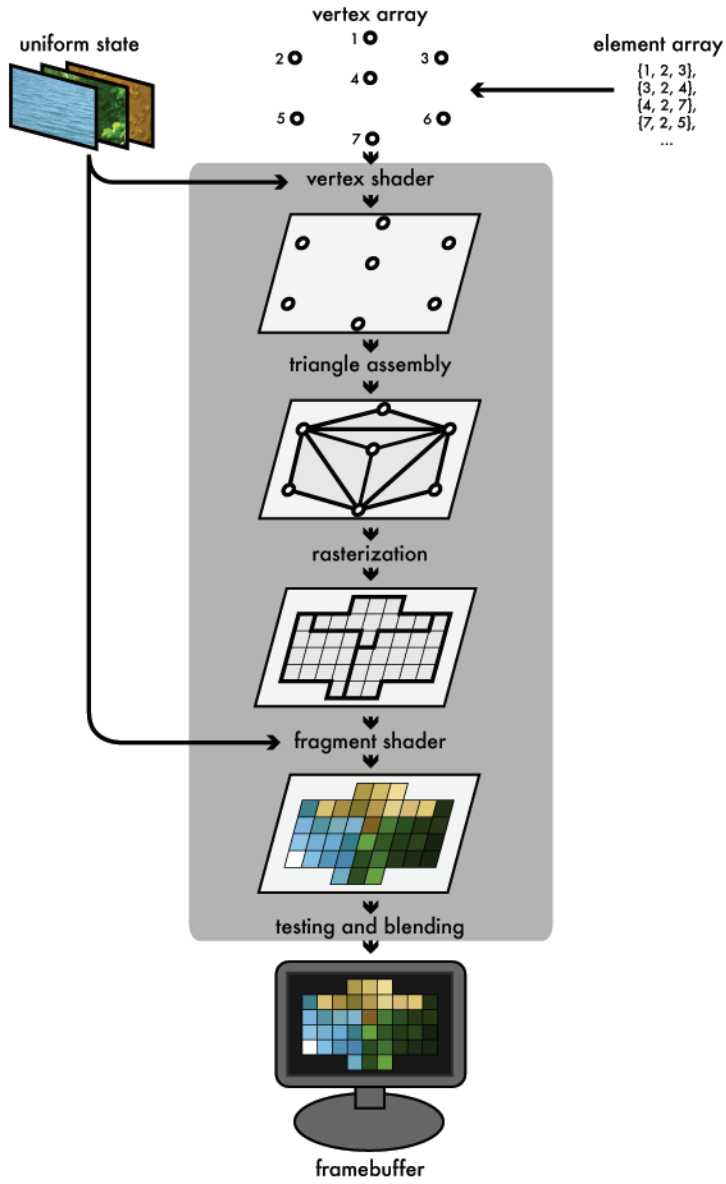


Figure 9 - Pipeline graphique simplifié

Le programme principal remplit des *buffers* de la mémoire gérés par OpenGL avec des « vertex arrays » (des tableaux de sommets). Un autre « buffer » de « element array » est rempli et celui-ci précise les triangles à former à partir des sommets. Ces sommets sont projetés dans l'espace écran par les *vertex shaders*, assemblés en triangles à l'aide des sommets fournis dans le *element array* et enfin "rasterisés" (ou pixelisés) en fragments. Finalement, ces fragments se voient assigner des couleurs par des *fragments shaders* et sont dessinés sur le *framebuffer*.

Au minimum, le *vertex shader* calcule la position des sommets projetés sur l'espace écran mais il peut également renvoyer les coordonnées de la texture. Le *vertex shader* et le *fragment shader* sont donc des programmes construits par le programmeur et envoyés au GPU par le programme principal.

2.2.3.2. Affichage de terrains texturés

Pour effectuer un rendu de terrain en 3 dimensions, on commence généralement par découper la surface initiale en rectangles (ou tuiles) qui forment donc une grille. Celle-ci peut être régulière ou irrégulière. Comme montré sur la figure 10 qui suit, à chaque cellule de cette grille (chaque tuile), on associe un MNT (modèle numérique de terrain, constitué d'élévations régulièrement ou irrégulièrement réparties sur la surface de la tuile) et une texture. Le MNT porte l'information de relief tandis que la texture contient l'information de couleur.

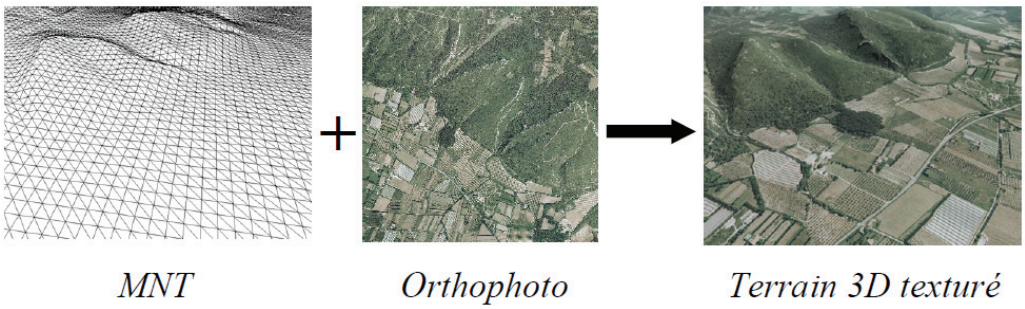


Figure 10 - Rendu d'un terrain 3D

Le rendu de terrain en trois dimensions n'est pas un domaine récent et beaucoup d'algorithmes sont disponibles. Nos contributions seront notamment liées à l'intégration de la dimension Web, tant pour les problématiques réseau que pour celles de l'affichage au sein d'un simple navigateur d'ordinateur ou de dispositif mobile.

2.3. État de l'art

S'il existe de très nombreux travaux sur la visualisation de terrains, la plupart d'entre eux ne sont pas compatibles avec l'utilisation de WebGL. En effet, beaucoup proposent des techniques intéressantes mais la version embarquée d'OpenGL que constitue WebGL limite considérablement les possibilités de calcul et d'affichage. Ainsi, nous avons exclu de cet état de l'art tous les articles traitant de la tessellation car celle-ci n'est pas réalisable avec WebGL, du fait de l'impossibilité d'ajouter de nouveaux triangles dans le pipeline d'affichage.

Le VRML des origines, par exemple, ne répondait pas à nos besoins pour plusieurs raisons, et notamment à cause de son format qui n'est pas compressé. Bien que le problème du réseau ou du stockage puisse paraître anecdotique à l'ère des smartphones capables de télécharger jusqu'à plusieurs mégabits par seconde et des terminaux dont les mémoires de masse se comptent en gigaoctets voire téraoctets, il reste un enjeu critique dans les sites à forte affluences tels que le Géoportail. Par nature, le site est composé de plusieurs téraoctets de données qui doivent être accessibles rapidement à un nombre important d'utilisateurs simultanés (20 000 pour le Géoportail actuel). Ces trois contraintes cumulées, à savoir la masse de données stockées, la vitesse de récupération et le nombre d'utilisateurs simultanés, imposent une réflexion sur la compression, le format de stockage, les mises à jour et la possibilité de *streaming*. Ainsi, contrôler le moteur même de rendu 3D, similaire à l'OpenGL, nous autorise à créer une solution complète répondant à nos besoins pour la visualisation continue de grands terrains avec des méthodes de tuilage (tiling), un stockage compatible avec les formats de cartographie 2D actuels préconisés par l'OGC, tout en conservant la possibilité de réaliser du *streaming*.

La meilleure manière de valider notre hypothèse de faisabilité dans ce contexte très contraint consistait donc à réaliser un prototype de visualisation capable de traitements lourds, comme par exemple la décompression temps réel des modèles de terrain avant leur affichage. Avant de présenter un horizon des connaissances dans les domaines qui nous intéressent dans le cadre de la conception de ce prototype, il est important de comprendre le contexte et les évolutions technologiques des navigateurs de ces quatre dernières années.

En 2009, Google propose une solution d'exécution de binaire dans une *sandbox* du nom de *Native Client* [YEE ET AL. 2009]. Bien que les performances soient encourageantes, cette solution n'a pas été retenue par les navigateurs pour plusieurs raisons : la première est le lien obligatoire entre le binaire et l'architecture qui demande à minima de recréer un binaire par type de processeur (x86, arm...). La seconde raison est que l'utilisation de code source offusqué (non lisible) dans les navigateurs était à contre-courant des règles implicites du Web. Celles-ci prônent en effet des codes ouverts et scriptés.

À cette même période, les « WebWorkers » commençaient à se stabiliser, permettant ainsi d'obtenir une technologie équivalente au *multi-process* avec des API de communication de type « envoi de message ». Tout comme « l'actor model », aucun état n'est partagé entre les WebWorkers. Cette API est particulièrement intéressante pour éviter les lenteurs dues aux entrées et sorties d'information par les caches ou le réseau. Néanmoins le *JavaScript* n'est toujours pas optimal pour certains traitements. Effectivement, le parcours d'un arbre binaire de plus de 1000 éléments est 50 fois moins rapide en *JavaScript* qu'en langage C optimisé, sans compter les problèmes de mémoire qui sont intrinsèques aux objets *JavaScript*.

C'est fin 2009 que le WebGL est créé à partir d'un projet Mozilla du nom de Canvas 3D. A l'époque, seul Mozilla supportait le projet. Après quelques mois, mi-2010, les problèmes de mémoire et d'efficacité des tableaux étaient de plus en plus visibles comme goulot d'étranglement pour le WebGL. C'est pour cette raison que les TypedArray[14] (tableaux binaires optimisés) ont été intégrés aux navigateurs. A noter que les expériences de Mozilla, au travers de LLJS[15], un JavaScript modifié pour réaliser de la programmation de bas niveau), ont montré que le problème de création d'objets, parcours et suppression rapide n'est toujours pas totalement réglé à ce jour pour ces structures.

Nous allons maintenant examiner les différentes solutions de visualisation et de *streaming* compatibles avec les contraintes de la thèse. Ces méthodes ont été reprises en grande majorité dans le rapport de R. Pajarola, et E.Gobbetti en 2007 [PAJAROLA ET AL. 2007]. Non seulement la plupart des articles importants qui traitent de la visualisation grâce à des grilles régulières et semi-régulières y sont décrits, mais aussi leurs usages dans des logiciels spécialisés.

[14] http://www.Khronos.org/registry/typedarray/specs/latest/
[15] http://lljs.org/

Historiquement, l'une des premières parutions modernes sur la visualisation de « soupe de triangles » est celle de M. Duchaineau [DUCHAINEAU ET AL. 1997]. Il y est décrit notamment comment utiliser une grille régulière pour la transformer en grille semi-régulière dont l'erreur associée à chaque triangle dépend du point de vue. Des limites, hautes et basses, sont définies pour la fusion et la séparation des points, en fonction de l'erreur observée. Par cette méthode, le nombre de triangles affiché est totalement maîtrisé et l'utilisateur peut donc imposer un nombre précis. En outre, afin d'optimiser la vitesse de traitement, une analyse de cohérence entre les images successives est réalisée.

En 2001, H. Hakl [HAKL 2001][16] décrit une méthode fondée sur ROAM[17], expliquée précédemment, mais qui utilise une structure géométrique alternative : l'arbre quaternaire à base de triangles. Il en résulte moins de fusions et divisions, tout en ajoutant la possibilité d'utiliser des chemins de triangles optimisés pour l'affichage. Cette méthode s'avère particulièrement efficace à une époque où la communication entre CPU et GPU était réduite en bande passante, et avec des latences importantes. L'algorithme repose sur quatre listes LIFO pour les buffers au lieu d'une seule file de priorité, et les jonctions entre les tuiles de triangles qui provoquent des discontinuités sur la surface sont éliminées par des ajouts de triangles de façon ad hoc par le parcours de l'arbre. Cependant, la communication CPU et GPU n'est plus aussi limitée aujourd'hui, invalidant ainsi l'une des hypothèses du papier.

Dans l'article « *Real-time Terrain Rendering using Smooth Hardware Optimized Level of Detail*» [LARSEN ET AL. 2003], une grille régulière est utilisée avec du géomorphing pour éviter les cassures entre les différents niveaux de résolution. De plus, chaque tuile est créée dans une liste et est fournie avec ses voisins, avec les niveaux de détail nécessaires au calcul par interpolation des points manquants lors d'un passage à un niveau de détail plus important. L'inconvénient de cette méthode est qu'elle suppose une grande quantité de communication entre le disque-dur et le GPU (en passant par la mémoire vive), à cause des besoins de la tuile principale mais aussi de ses voisins, et ce sur plusieurs niveaux de résolution.

Une approche originale a été décrite par P. Cignoni dans son papier « *Batched Dynamic Adaptive Meshes for High Performance Terrain Visualization* » [CIGNONI ET AL. 2003]. Partant du constat que le nombre de triangles n'était plus aussi limité en 2003, et que le transfert entre le CPU et la mémoire du GPU était beaucoup plus rapide, en termes de bande passante comme de latence, il devenait donc plus intéressant, pendant la visualisation, de traiter un ensemble de triangles plutôt que chacun d'entre eux individuellement. Ainsi, le papier revendique un modèle de communication entre le serveur et la carte graphique qui dépasse les performances des autres solutions. Il repose sur la fusion et division (appellé aussi « tessellation », tout en conservant les aspects irréguliers des maillages à l'intérieur des *patchs*. En revanche, le système de partitionnement du terrain

[16] Henri Hakl, University of Stellenbosch
[17] ROAMing Terrain: Real-time Optimally Adapting Meshes, 1997

global en *patchs*, fondé sur un arbre binaire avec des contraintes, nécessite un prétraitement coûteux pour produire des fichiers statiques du terrain représenté sous plusieurs niveaux de détail. Le point intéressant, dans notre contexte, est que même si le prétraitement de simplification demande plus de ressources que les autres solutions, une fois cette étape effectuée, le client ne nécessite que le transfert de fichiers statiques et par conséquent, dans le cas d'une utilisation réseau, peu de ressources côté serveur.

En 2006, le C-BDAM [GOBBETTI ET AL. 2006], évolution du BDAM [CIGNONI ET AL. 2003], reprend le même principe en remplaçant les maillages irréguliers inclus dans chaque *patch* par des grilles en quinconce. Il optimise également le remplacement d'un *patch* (ou feuille de l'arbre binaire) par un de ses enfants lors du raffinement par l'utilisation d'une ondelette. Le changement de méthode dans la simplification est plus rapide à calculer durant le prétraitement et plus simple à réaliser. En outre, il peut être effectué sur la carte graphique. Néanmoins, la nouvelle méthode ne peut être utilisée qu'avec des modèles projetables 2,5D (c'est à dire qu'il est impossible de gérer les surplombs, c'est-à-dire deux points ayant les mêmes coordonnées x et y avec un z différent).

Il existe d'autres méthodes qui autorisent la visualisation 3D sur des terminaux avec de faibles capacités de calculs. L'article « *Adaptive Streaming and Rendering of Large Terrains using Strip Masks* » [POUDEROUX ET AL. 2005] utilise des chemins de triangles pour la performance et règle les problèmes de jointure à l'aide d'une texture de fond projetée telle une ombre en dessous du terrain.

J. Schneider [SCHNEIDER ET AL. 2006] propose en 2006 de diviser le terrain en tuiles de 257 par 257 éléments. Afin de combler les trous pour le raccordement, un ruban de "*zero area polygons*" est défini. D'une part, cette approche améliore le stockage sur GPU[18] : en effet, seuls 9 bits sont nécessaires sur l'axe X, 9 pour Y et 14 pour le Z. D'autre part, elle permet une transmission progressive des sommets afin d'optimiser le cache du GPU, et une pagination de la mémoire faisant appel à la fois à un algorithme du dernier utilisé récemment (LRU : « *last recently used* ») et de la « *tighest fit strategy* » (TF).

Une autre solution de jointure entre les tuiles a été proposée dans « *Seamless Patches for GPU-Based Terrain Rendering* » [LIVNY ET AL. 2007]. L'idée principale est de mettre en place une rotation de 45° entre la grille des tuiles et celle provenant du modèle de terrain. Chaque tuile étant alors composée de 4 sous-triangles dont la résolution dépend de celle des voisins.

[18] GPU-Friendly High-Quality Terrain Rendering Journal of WSCG 2006

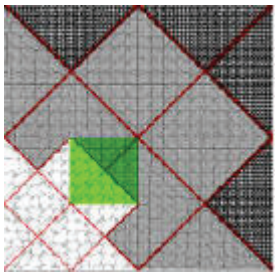


Figure 11 - Nous voyons en vert la donnée et en rouge le découpage du papier. Par cette méthode, il est possible de mettre à jour une tuile verte (ainsi composée de 4 méta-tuiles en rouges) par une mise à jour des 4 sous-triangles des méta-tuiles sans introduire de changement dans les autres. On évite ainsi toute anomalie (crack) dans le paysage.

Une nouvelle fois, l'utilisation de masques fixes ou indices fixes pour les élévations, comme paramètre des sommets, permet d'obtenir de meilleures performances de rendu. Cependant, les triangles composant les méta-tuiles doivent être construits à la volée en fonction du point de vue, ce qui implique une quantité de transfert de données importante, qui peut être facilement soutenue par un disque dur local, mais qui est irréaliste dans le cas d'un réseau classique de type xDSL.

Enfin, il existe une méthode de *raycasting* qui a été proposée par *C. Dick* [DICK ET AL. 2009][19]. Néanmoins, cette solution exige de nombreuses passes avec plusieurs algorithmes différents pour le rendu, comme par exemple pour exploiter l'occlusion. De plus, le papier est efficace uniquement lorsque la résolution est grande (1920*1080) avec une large quantité de données par pixel (soit plus de 64 sommets par pixel). Dans le cas contraire, les approches classiques et la rastérisation semblent être plus adaptées à notre contexte.

La plupart des articles mettant en œuvre la technique de *raycasting* [DICK ET AL. 2009] ou de raytracing sont également mis de côté, car leur utilisation requiert beaucoup de puissance et celle-ci ne sera pas disponible sur l'ensemble des supports que nous envisageons d'utiliser (les terminaux mobiles notamment). En effet, ces techniques nécessitent des dispositifs possédant une carte graphique relativement récente et puissante.

La méthode des *Clipmaps* de *Losasso et Hoppe* [LOSASSO ET AL. 2004] propose un rendu de terrain à partir de la manipulation de grilles fixes régulières hiérarchisées. Les auteurs décrivent une solution rapide et efficace qui permet de changer la résolution des données affichées selon la distance de la caméra à l'objet. Cette technique, appelée *Geometry Clipmap*, est spécialisée dans la génération de maillages très réguliers permettant une mise en cache sur le GPU. En effet, le problème majeur des algorithmes précédents est une mise en œuvre sur CPU, nécessitant d'importants transferts de données vers le matériel graphique à chaque image. Au contraire, cette dernière technique recourt à l'alimentation maximum du pipeline graphique (décrit en section 2.2.3.1) à chaque image, en fixant un nombre de triangles rendus et en les répartissant simplement au mieux, en fonction de la distance au point de vue. Des « anneaux » rectangulaires, contenant chacun un niveau de

[19] C. Dick, J. Krüger, R. Westermann (Munich, Utah)

détail, sont ainsi dessinés autour de l'utilisateur (ce qui correspond à l'emplacement de la caméra). Plus on s'éloigne de celui-ci, plus la résolution des données diminue. A l'intérieur d'un même niveau de résolution, les triangles sont répartis uniformément et ne tiennent pas compte du relief du terrain. Lors d'un déplacement de caméra, seules les nouvelles données sont transmises à la carte pour compléter chaque anneau. Les anciennes données ne sont donc pas redessinées, et les nouvelles sont quant à elles calculées et affichées. Cette méthode est intéressante pour le chargement des données car seules les dernières seront chargées en RAM, les autres y figurant déjà. On peut voir ci-dessous (figures 12 et 13) les anneaux rectangulaires et la pyramide de niveaux qui illustrent l'application des données à partir des différents niveaux de détail sur les grilles imbriquées formant les anneaux. C'est cette dernière solution qui constitue la base de notre approche, avec des adaptations et des simplifications : *Losasso et Hoppe* [LOSASSO ET AL. 2004] avaient introduit un bruit fractal afin de rendre le résultat visuellement plus réaliste. Cependant, dans notre cas, nous souhaitons avoir une visualisation la plus fidèle possible, tout en respectant les données de terrains réels (celles de l'IGN pour les territoires français).

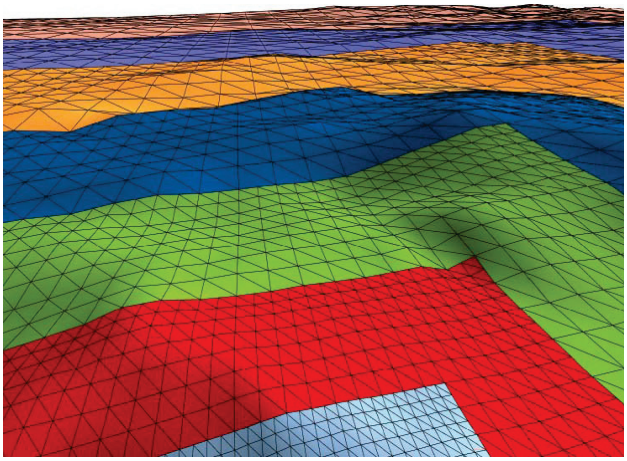


Figure 12 - Anneaux rectangulaires centrés sur le point de visualisation

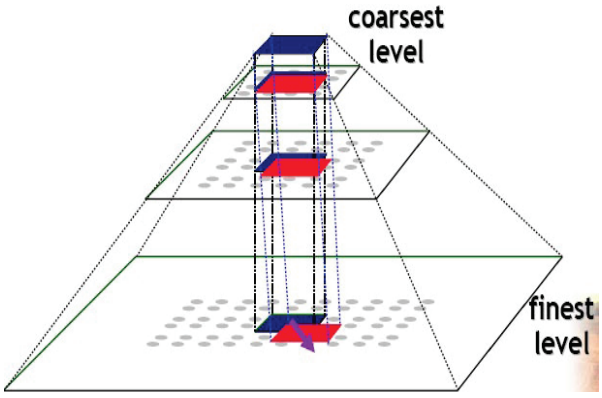


Figure 13 - Pyramide de niveaux

Si une transposition de ces travaux sous forme d'application Web est possible, il faut néanmoins les adapter pour tenir compte des contraintes inhérentes au *WebGL*, aux spécificités du *JavaScript* et des navigateurs, ainsi qu'à l'hétérogénéité des clients de visualisation et d'interaction, qui peuvent aussi bien être des ordinateurs que des tablettes ou smartphones.

Jusqu'à présent, nous avons abordé la visualisation de terrains du point de vue de la segmentation et de la hiérarchisation de l'information. Or, le problème soulevé par la visualisation 3D est double : outre l'étape indispensable consistant à segmenter l'information globale pour faciliter sa manipulation, il faut établir une méthode de simplification des *patchs* qui fournira les différents niveaux de détail. Nous complétons donc

maintenant notre état de l'art par une présentation rapide des méthodes de simplification les plus connues.

La simplification de maillage n'est pas un sujet récent et nous n'effectuerons pas ici un traitement exhaustif de la thématique. D'autre part, l'étude de certaines méthodes (contraction d'arête, clusterisation) est différée au chapitre 3.

Le problème peut être abordé comme l'approximation avec une erreur bornée d'une courbe constituée de n segments. L'algorithme le plus connu est celui de *Douglas-Peucker* [DOUGLAS-PEUCKER 1972] qui consiste, à chaque étape, en une tentative d'approximation d'une séquence de points par un segment tendu entre le premier et le dernier. Si le point le plus éloigné du segment est en dehors de la borne d'erreur, le segment et découpé en deux pour relier ce dernier, l'algorithme étant exécuté récursivement sur les nouveaux segments tant qu'il reste un sommet dont l'erreur dépasse la limite autorisée. Le problème majeur des techniques de simplification de courbes est la gestion de l'erreur du résultat qui doit être recalculée à chaque étape.

Comme nous l'avons exposé ci-dessus, la simplification peut être effectuée en ignorant certains sommets ou même triangles : *W. J. Shroeder et al*. [SHROEDER ET AL. 1992] nous expliquent comment réduire le nombre de triangles grâce à des opérations locales sur la géométrie et la topologie.

Il existe d'autres méthodes utilisant des outils mathématiques différents : par exemple, *Lounsbery et al*. [LOUNSBERY ET AL. 1994] proposent une représentation multi-résolution qui consiste en un maillage simple lié à des coefficients de correction locale nommés "*wavelets coefficients*" et qui représentent les détails présents dans l'objet à différentes résolutions. Néanmoins, ce principe ne peut s'appliquer que sur des maillages réguliers avec des triangles d'un type défini. Ce problème est résolu par *Eck et al*. [ECK ET AL. 1995] à l'aide d'une approximation du maillage d'entrée qui est alors arbitraire.

C'est d'ailleurs par une approximation aléatoire que *Turk* [TURK 1992] diminue le nombre de points décrivant la surface d'un objet 3D : à chaque étape, des points aléatoires sont choisis sur la surface, puis ceux qui ne sont pas assez uniformes sont retirés avant d'utiliser les autres pour générer des sommets pour la nouvelle triangulation (après une tessellation de certains polygones si nécessaire).

Les papiers précédents ne sont que des exemples traitant de simplification de maillages. Pour plus de détails, le lecteur pourra se reporter au *survey* détaillé de *Heckbert et Garland* [HECKBERT ET AL. 1997].[20] D'autre part, nous reviendrons sur ce sujet dans la dernière partie de ce mémoire.

[20] Survey of Polygonal Surface Simplification Algorithms, 1997

2.4. Notre proposition : OpenScalesGL pour ordinateurs et mobiles

Nous présentons ici notre contribution pour la visualisation. Celle-ci est basée sur un quadtree classique pour les niveaux de détail, dans lequel nous avons contraint les différences de niveaux entre voisins afin de résoudre les cassures (cracks) entre les tuiles à la volée.

2.4.1. Les algorithmes

Pour *OpenScalesGL* [CELLIER 2012], un quadtree est utilisé pour la segmentation de l'espace. Un quadtree est une structure de données de type arbre dans laquelle chaque nœud peut compter jusqu'à quatre fils. La figure 14 ci-dessous montre l'évolution du quadtree d'un niveau à l'autre.

Comme expliqué précédemment, à chaque cellule de ce quadtree on associe une texture et un MNT. Plus on descend dans les niveaux du quadtree, plus la résolution des données sera précise.

Comme mentionné dans les différents articles et le *survey* de *Pajarola* [PAJAROLA ET AL. 2007], il est intéressant de tenir compte de la distance de la caméra et de son orientation. En effet, le fait d'utiliser une résolution faible pour un objet éloigné et une résolution plus élevée pour un objet proche évite de stocker un surplus d'informations inutiles dans la carte mémoire.

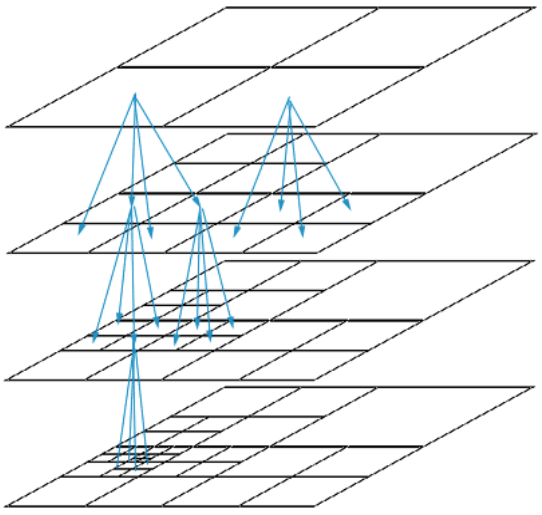


Figure 14 - Représentation d'un quadtree

La structure du quadtree, permet de gérer aisément la variation de résolution des données. Au lieu de créer les anneaux rectangulaires des *Clipmaps* [LOSASSO ET AL. 2004], on choisit de développer uniquement les cellules du quadtree qui se trouvent devant la caméra. Leur niveau de développement (ou raffinement) dépendra de leur distance à la caméra. Afin de faciliter les calculs, la distance est estimée à partir du centre de la cellule. La différence avec une grille régulière est qu'il n'est pas nécessaire que tous les nœuds du quadtree soient au même niveau (voir figure 15).

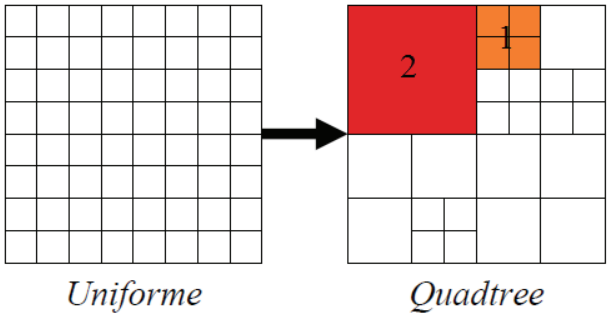


Figure 15 - Grille régulière et quadtree

Un exemple de ce que nous pourrions obtenir est représenté sur la figure 16. Au lancement de notre application, la caméra est éloignée de l'île de la Réunion (vue 1), une seule emprise (exprimée en degré décimaux) est donc envoyée pour obtenir une seule image en retour. Dans le cas de la vue 2, la caméra s'est rapprochée du terrain, le quadtree est donc descendu d'un niveau et s'est divisé en quatre fils. Ici, il y a alors quatre emprises qui sont envoyées, et quatre images sont ensuite affichées. Enfin, dans le dernier cas (vue 3), la caméra s'est de nouveau rapprochée mais s'est aussi orientée vers la droite. Ce sont uniquement les cellules devant la caméra qui se sont donc subdivisées afin d'obtenir des images (et des MNT) de meilleure résolution que celles de derrière, dont il n'est pas utile d'avoir tous les détails. Dans le cas 3, on peut voir trois niveaux de détails différents (rouge, bleu et vert).

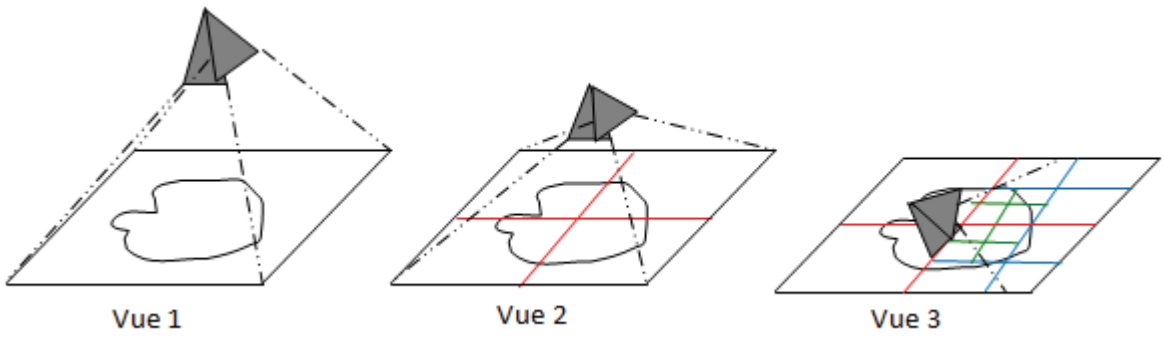
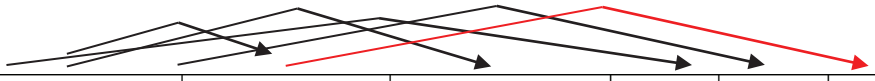


Figure 16 - Rapprochement de la caméra

Au niveau du développement, notre quadtree est représenté sous forme de tableau. A chaque fois que l'on développe une cellule en quatre fils, on ajoute quatre cases au tableau avec l'indice de la cellule parente, l'indice du premier enfant et les indices des voisins (droite, gauche, bas et haut). Le schéma ci-dessous illustre ce principe :



0	1	2	3	4	5	...	...
Cellule 1	**Cellule 1-1**	**Cellule 1-2**	**...**	**Cellule 1-4**	**Cellule 1-1-1**	**...**	**Cellule n-n-n**
Indice du 1er enfant :1	Indice du 1er enfant :5	Indice du 1er enfant :null	...	...			
Indice du parent: *null*	Indice du parent: *0*	Indice du parent: *0*					
Est parent: vrai	Est parent: vrai	Est parent: faux					
Indice voisins	Indice voisins	Indice voisins					

Figure 17 - Tableau contenant le quadtree

Maintenant que nous savons comment récupérer les textures et les MNT et comment afficher les données cellule par cellule, nous allons expliquer comment le rendu 3D est calculé dans les *shaders*.

Ainsi que nous l'avons fait observer lors de la description du pipeline graphique (cf. Section 2.2.3.1), il est nécessaire d'envoyer à notre *vertex shader* un tableau contenant les sommets (*vertex*) que nous souhaitons afficher. Cette étape a pour but de calculer leur position 3D (qui est pour l'instant inconnue). Pour cela, il nous faut indiquer leur position relative (quel est le premier sommet de l'image), et les sommets que nous souhaitons relier entre eux. Il faut savoir que le nombre de sommets par ligne doit être une puissance de 2 afin d'améliorer les performances et être compatible avec le plus de smartphones possible. Plus le nombre de sommets par ligne est élevé, plus le niveau de détail sera important. On doit retrouver ce nombre de sommets par ligne dans le MNT. En effet, ce dernier contient une matrice d'élévation de taille $n \times n$ où n désigne le nombre de sommets par ligne.

Quel que soit le nombre de sommets par ligne, notre objectif est de pouvoir tracer des triangles au sein d'une cellule du quadtree. En effet, la forme géométrique du triangle est la plus simple et la plus naturelle pour réaliser un affichage 3D.

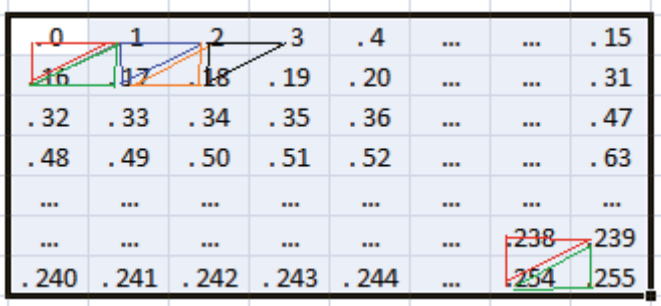


Figure 18 - Une cellule du quadtree avec 16 sommets par ligne

Par exemple, si l'on choisit d'avoir 16 sommets par ligne, notre cellule du quadtree ressemblera à la figure ci-contre (figure 18). Pour dessiner les triangles, il faut que l'on passe du tableau de sommets au *vertex shader*. Le premier est appelé tableau de position, dans ce tableau on indique uniquement l'ordre des sommets à traiter. Il s'agit donc d'un simple tableau allant de 0 à 255 (car on souhaite avoir 16x16=256 points).

Position:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...	...	...	253	254	255
Index:	0	1	16	16	1	17	1	2	17	17	2	18	2	3	18	...	238	239	254	254	239	255

Figure 19 - Tableaux de positions et d'indices à envoyer au vertex shader

Le deuxième tableau de sommets que nous devons fournir est le tableau d'indices (ci-dessus). Dans ce tableau se trouve la suite des sommets à relier pour former le maillage. Le GPU sait qu'il doit construire des triangles, il va donc traiter les sommets par paquets de trois et les relier entre eux. Si l'on souhaite dessiner le premier triangle rouge représenté sur la figure 18, il faudra donner les sommets : 0, 1, 16. Puis les sommets 16, 1, 17 pour tracer le triangle vert. Et ainsi de suite pour dessiner l'intégralité des triangles au sein de la cellule du quadtree. L'intérêt de ce masque « fixe » est de pouvoir donner uniquement les élévations à la carte graphique sans devoir répéter les positions relatives des triangles ou leur connectivité. Cela permet une utilisation efficace des caches du GPU.

A l'issue de cette opération, nous obtenons un maillage de triangles. Afin d'ajouter le relief et de représenter l'ensemble des cellules du quadtree les unes à côté des autres, il est nécessaire de calculer la position de chaque sommet dans l'espace. Pour cela, on envoie les données du MNT dans le *vertex shader* sous forme de tableau. Le tableau contenant les données du MNT comptera exactement le même nombre de points qu'il y a de sommets dans une cellule du quadtree. Si l'on reprend l'exemple précédent, on a 16 sommets par ligne, soit 256 sommets dans la cellule du quadtree et dans le tableau de données du MNT. Il faut préciser que le nombre de sommets par ligne peut être modifié par l'utilisateur (16, 32, …, 1024) et que par conséquent les algorithmes sont écrits de manière à prendre automatiquement en compte ce changement de paramètre.

C'est grâce aux *shaders* que l'on va pouvoir calculer la position 3D de chaque sommet, et assigner une couleur à chaque pixel. L'algorithme ci-dessous est exécuté pour chaque sommet :

1. Calcul de la position du sommet au sein de la cellule du quadtree (position relative)
2. Calcul des coordonnées x et y dans l'espace 3D à l'aide des coordonnées géographiques de l'emprise (en degrés décimaux) -> position absolue
3. Calcul des coordonnées de texture correspondantes
4. Envoi de la position 3D du sommet et des coordonnées de texture au *fragment shader*
5. Récupération de l'information de couleur aux coordonnées de texture calculées précédemment
6. Attribution de la couleur au pixel.

Après ces six étapes, la texture (orthophoto) que nous avons chargée est plaquée sur le MNT. La figure 20 ci-dessous montre l'analogie entre l'espace 3D (X,Y,Z) et l'espace texture(s,t).

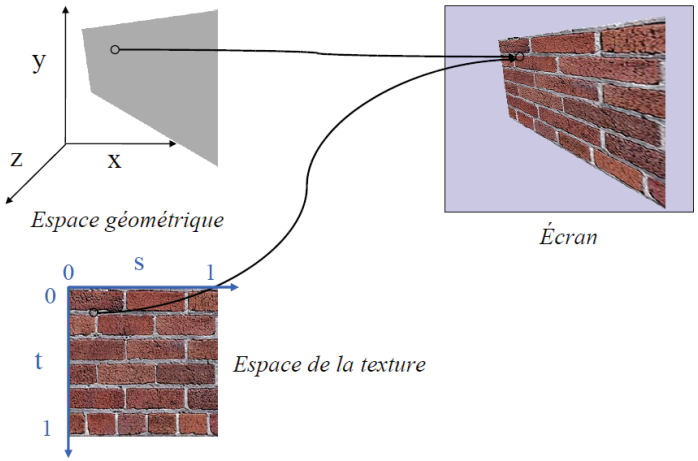


Figure 20 - Principe du plaquage d'une texture

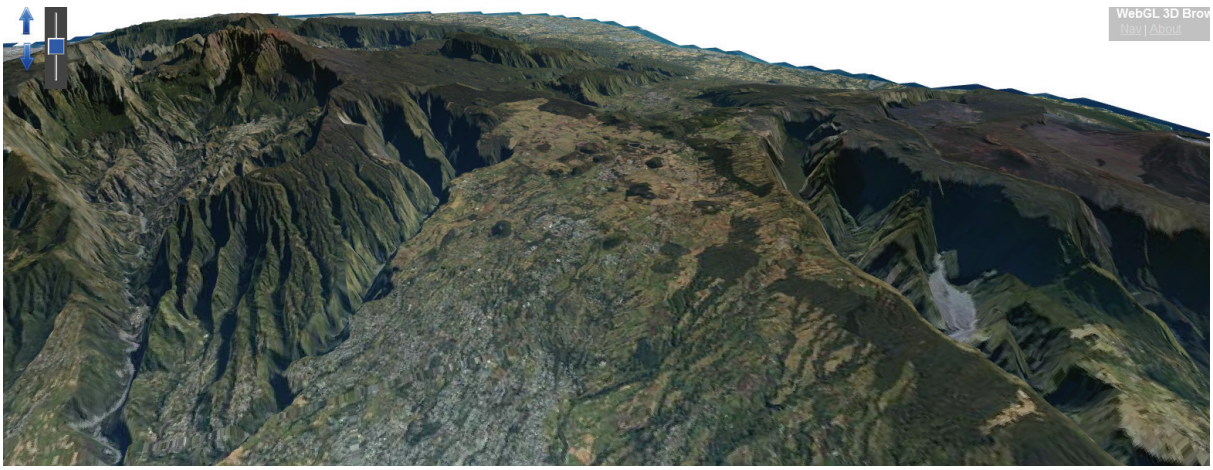


Figure 21 - Première version d'OpenScalesGL

Voici le pseudo-algorithme simplifié du processus de rendu d'un terrain 3D (figure 21) consistant à dessiner le contenu du quadtree :

- Si la caméra s'approche, le quadtree se subdivise en quatre fils (et ainsi de suite).
- Pour chaque cellule (enfant) du quadtree, on connait l'emprise géographique précise qui nous permet de requêter une image et un MNT.
- On dessine une à une chaque cellule en envoyant des tableaux de sommets dans les *shaders*.

Le rendu final du terrain est affiché dans la balise <canvas> de *HTML5*

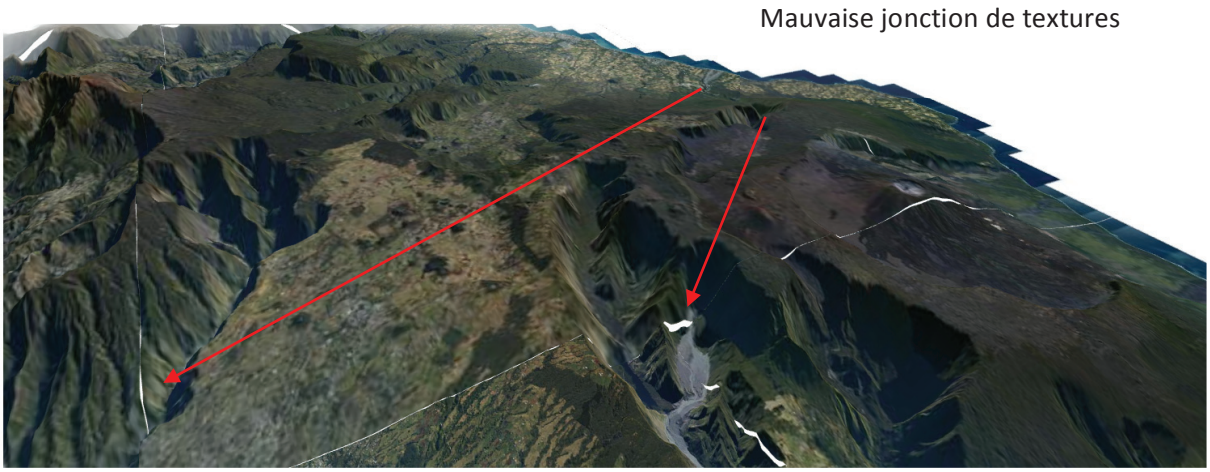


Figure 22 - Ecarts entre les textures

2.4.1.1. Jonction des cellules

Dans la méthode proposée ci-dessus, la liaison entre les textures n'a pas été prise en compte. En observant la manière dont nous avons construit nos triangles au sein des cellules du quadtree, nous voyons que l'écart que l'on voit sur la figure 22 est de la taille d'un pixel (cf. figure 23 et 24). En effet, si deux cellules adjacentes du quadtree ne possèdent pas exactement le même relief, il peut y avoir des écarts d'élévation visibles à l'œil nu. Sur les figures ci-dessous, on peut voir d'une part, la mise en évidence de l'écart entre deux cellules du quadtree situées côte à côte, et d'autre part l'écart dû aux élévations.

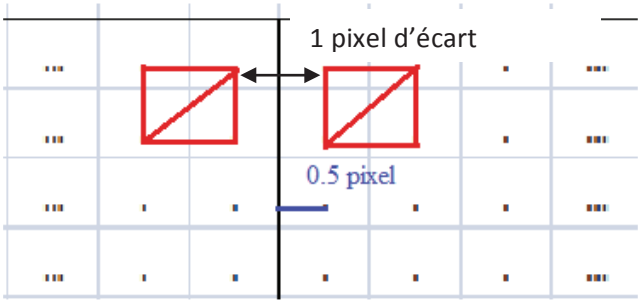


Figure 23 - Ecart entre deux cellules adjacentes du quadtree

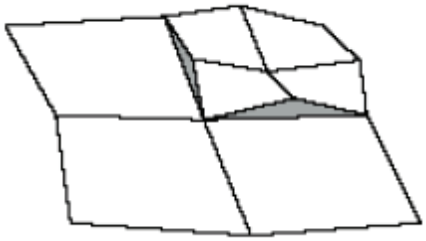


Figure 24 - Ecart dû aux élévations

Afin de pallier ce problème, une solution consiste à rajouter des *vertex buffers* à envoyer dans le *shader* afin de dessiner des triangles aux bordures des cellules. Pour cela, il faut vérifier, pour chaque cellule, la présence de cellules voisines. Afin d'éviter de faire des parcours inutiles du tableau représentant le quadtree, on tiendra compte des résultats obtenus au niveau de résolution inférieure c'est-à-dire celui des tuiles parentes. Ainsi si une cellule mère n'a pas de voisin de droite, les deux fils aux extrémités droites n'auront pas de voisins non plus à droite.

Il faut également faire attention à la différence de niveaux de résolution que l'on peut avoir entre deux cellules. L'algorithme que nous avons défini autorise un seul niveau de différence entre une cellule et sa voisine. Pour relier les bords des cellules, nous aurons donc seulement deux cas de figure (cf. figure 25).

Le cas n°1 concerne deux cellules adjacentes de même niveau (cellules 1 et 2 sur la figure) tandis que le cas n°2 gère deux cellules voisines présentant un niveau de différence (cellules 1 et 3 sur la figure). La construction des triangles rouges se fait de la même manière que celle des triangles bleus. On va stocker dans un *buffer* de positions deux lignes de longueur égale au nombre de sommets par ligne et un *buffer* d'indices contenant les indices des sommets à relier. Il faut également récupérer dans un autre *buffer* les élévations de la dernière ligne de la cellule 1 et celles de la première ligne de la cellule 2 (ou la dernière colonne de l'une et la première colonne de l'autre pour une

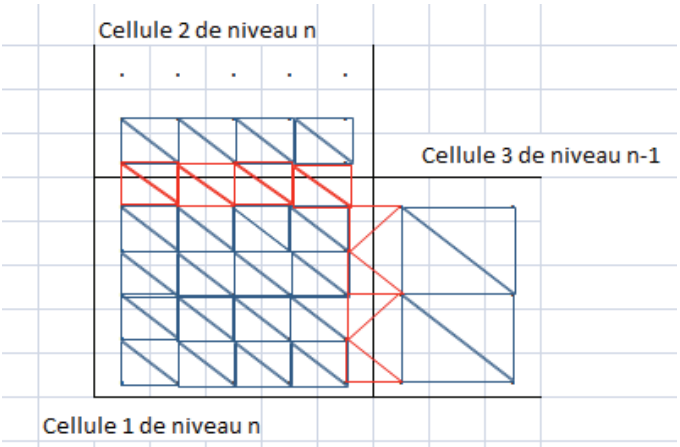


Figure 25 - Liens entre bords de cellules

frontière commune verticale).

Ensuite, une fois dans le *vertex shader*, tous les points suivent le même algorithme que précédemment. La seule différence entre ces deux cas de figure est le nombre de sommets que l'on va indiquer pour construire les triangles. Entre deux arêtes, on construit deux triangles (il faut donc 6 sommets) dans le cas n°1 et 3 triangles (soit 9 sommets) dans le cas n°2.

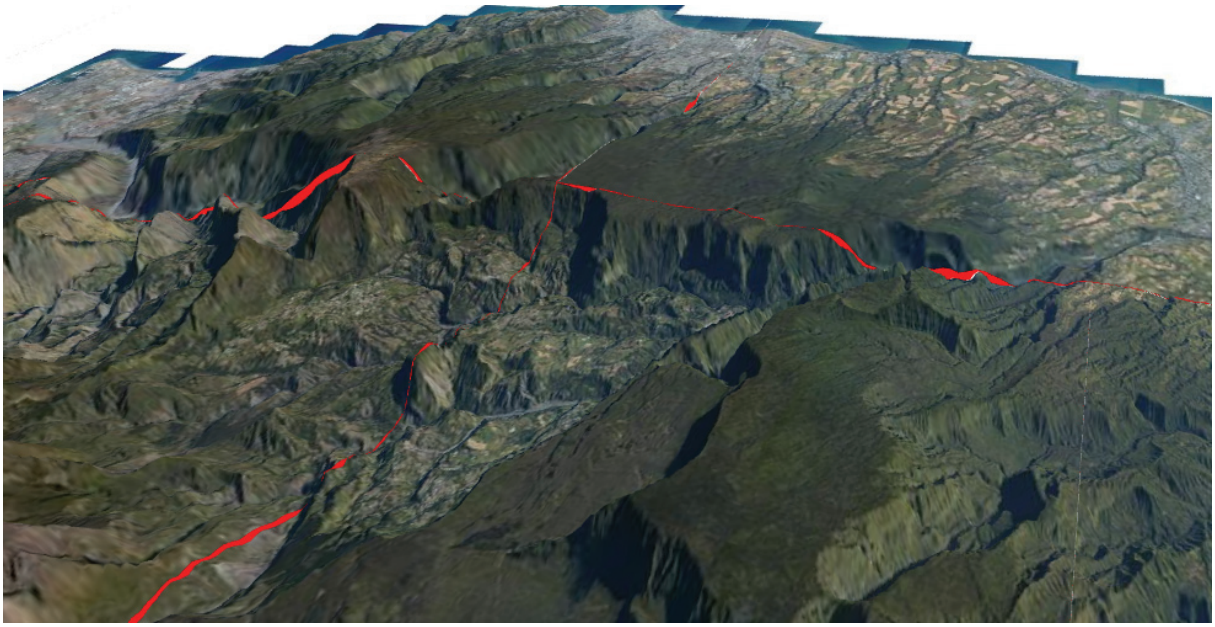


Figure 26 - Mise en évidence des jonctions entre les textures

Cette solution contraint le quadtree à n'avoir qu'un niveau de différence entre tuiles voisines, mais le même principe pourrait aussi être étendu à n niveaux de différence. Le principal défaut de cette approche est la consommation CPU induite. Parfaitement acceptable sur un ordinateur de bureau, elle peut provoquer des lenteurs sur certains smartphones.

Nous avons donc aussi testé une seconde solution originale, plus adaptée aux configurations légères, et dont l'idée nous est venue lors de l'étude du papier de *J. Pouderoux* [POUDEROUX ET AL. 2005] et de ses défauts. En effet, alors que la méthode crée une ombre complète de la texture pour combler les cassures, nous avons plutôt choisi d'étendre les tuiles. Cette approche permet d'éviter les sauts de couleur dès lors que la caméra est trop proche de la cassure et que sa direction suit l'horizon. Dans le cadre de configurations plus véloces, il est plus intéressant d'utiliser une des deux méthodes citées précédemment, telles que le *BDAM,* afin d'obtenir un résultat sans aucun défaut à la surface.

Dans le cadre de notre seconde solution originale, nous ajoutons une longueur de 2 sommets à chaque tuile. Ces sommets additionnels agissent comme des bordures, ayant les mêmes coordonnées en x et y que le sommet adjacent. Nous appliquons ensuite une pente à ces bordures. Ainsi, il y a forcément un point où se croisent les bords de deux tuiles

adjacentes : bien que la profondeur ne soit pas fidèle aux données originales au niveau de ce croisement, le phénomène de cassure disparaît et les tuiles apparaissent liées (cf. figure 27 ci-dessous).

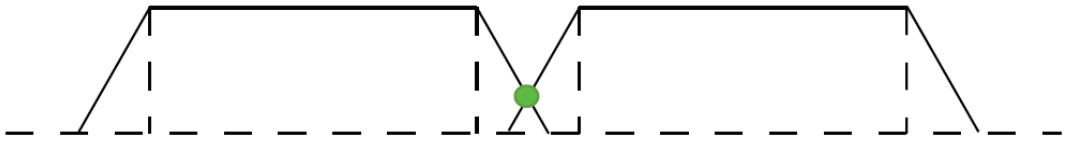


Figure 27 - Seconde solution par extension des tuiles

2.4.1.2. Résultats

Dans toute application 3D temps réel, le rendu se fait sous forme de boucles d'affichage. A chaque mouvement ou modification d'affichage, un nouveau rendu est réalisé. Ces boucles peuvent être très rapides et sont généralement invisibles à l'œil nu. A chaque itération, un rendu complet du quadtree est effectué.

30 tuiles visibles dans l'arbre. Chaque tuile possède 16k points et une texture 4 fois plus grande	**Temps du dessin du quadtree en ms**	**Temps du dessin de la cellule en ms (appel au GPU en asynchrone)**	**Temps de l'affichage global en ms (temps du GPU pour créer la frame complète)**
Version sans prise en compte des jonctions	2,71	0,087	18,76
Version avec prise en compte des jonctions	3,86	0,072	40,0

Figure 28 - Résultats expérimentaux

Le tableau (figure 28 ci-dessus) regroupe les résultats expérimentaux obtenus avec un ordinateur équipé d'un CPU intel i7-880 et d'une carte graphique ATI HD 6970, le tout avec 16 Go de RAM. Ces résultats mettent en évidence que la solution utilisée est limitée par le GPU et non par le CPU. C'est pour cette raison que le temps d'appel des fonctions pour l'affichage (3^{e} colonne) est bien plus rapide que le temps GPU nécessaire à l'affichage (4^{e} colonne). On observe aussi que le temps de rendu d'une cellule est plus court avec la gestion des jonctions que sans. L'explication se trouve dans le résultat visible : en comblant les trous, la carte graphique peut optimiser les calculs en éliminant les triangles non visibles lors de la rastérisation.

Ces résultats ne signifient pas que le *JavaScript* est un facteur limitant pour des traitements lourds, tels que le calcul d'une triangulation de Delaunay, mais seulement qu'il est possible d'utiliser la totalité de la puissance du GPU si les problèmes de transfert réseau sont réglés en amont et si le modèle de données est bien adapté.

Il faut noter qu' *OpenScalesGL* a bénéficié de nombreuses optimisations que nous ne détaillerons pas ici. En effet, elles ont été abandonnées pour la plupart car certaines évolutions récentes des moteurs *JavaScript* les ont rendus obsolètes. Par exemple, notre premier prototype utilisait un système de cache pour éviter les pauses du ramasse-miettes (*garbage collector*), système aujourd'hui inutile avec la création du *garbage collector* incrémental intégré en 2011 dans Chrome et en 2012 dans Firefox[21].

[21] http://blog.mozilla.org/*JavaScript*/2012/08/28/incremental-gc-in-firefox-16/

3. Chapitre 2 : WebCL : Calcul parallèle haute performance pour le Web

Contenu

3.1. Introduction

Au cours de nos premières expérimentations sur la visualisation 3D en utilisant Flash ou WebGL, nous nous sommes rendu compte que les principales difficultés liées au portage de logiciels d'ordinateurs vers des clients Web légers résidaient, d'une part dans les performances limitées du langage *JavaScript*, et d'autre part dans les temps de transfert relativement élevés entre ce code *JavaScript* et la mémoire du GPU. C'est pour cette raison que nous avons choisi d'étendre les capacités du navigateur en proposant les WebCLWorkers, une API permettant l'exécution sur GPU et CPU de scripts compilés à la volée. Bien que notre objectif initial consistait à calculer des simplifications de scènes 3D en temps réel, l'outil développé peut parfaitement être utilisé dans de toutes autres situations. Enfin, nous avons souhaité ajouter ces nouvelles fonctionnalités en limitant au maximum les difficultés d'apprentissage, afin de faciliter l'adoption de l'outil par la communauté des développeurs Web, généralement peu familiers des API de calculs parallèles.

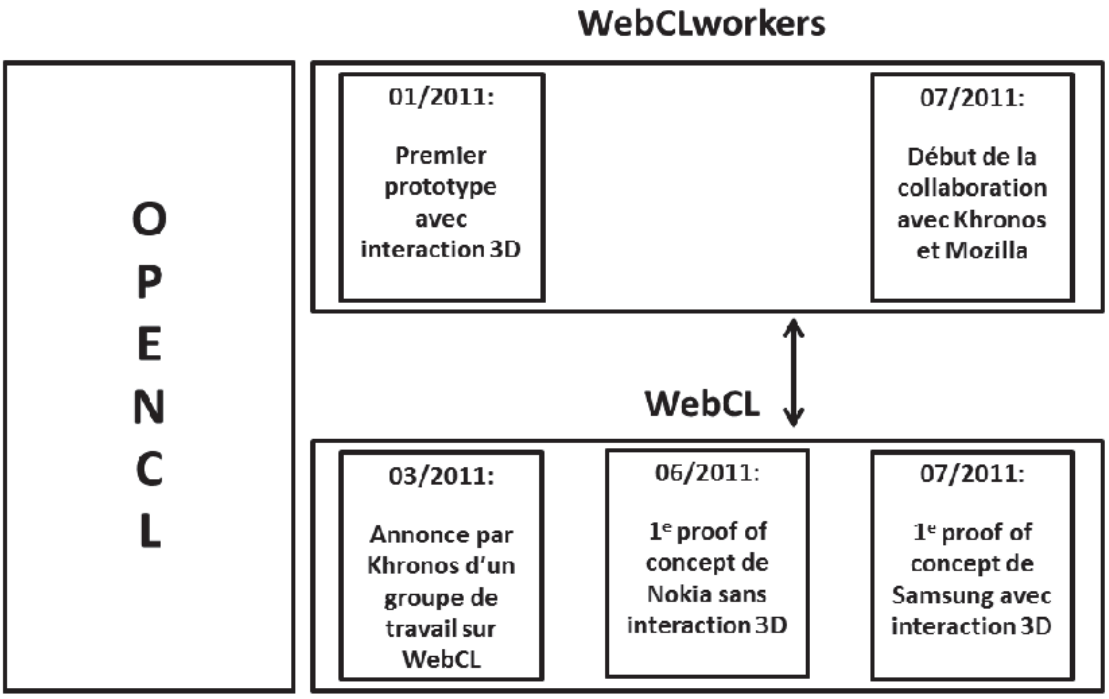


Figure 29 - Notre travail a été réalisé avant la décision du consortium Khronos de créer un groupe de travail sur ce qui deviendra le standard WebCL. Nous avons ensuite choisi d'intégrer l'équipe de Khronos et de partager avec elle les fruits de notre travail.

Ces dernières années, les possibilités offertes par les navigateurs n'ont eu de cesse d'augmenter, avec la mise à disposition de nouvelles API, si bien qu'il est maintenant possible de considérer les environnements d'exécution *JavaScript* comme des systèmes d'exploitation à part entière, contraints certes, mais pas davantage que dans le contexte déjà connu de l'embarqué. Cette comparaison est devenue possible grâce aux nouvelles

fonctionnalités *HTML5* telles que la persistance (*localStorage* et *sessionStorage*[22]) ou encore le mode offline offrant des applications riches fonctionnant sans recourir au réseau[23]. Dans ce mode, l'application va chercher les données à l'intérieur de son cache et les resynchronise dès que possible. Même si le cache est limité, il est possible de contourner cette limitation par l'utilisation de fichiers locaux, accessibles dans les pages Web par un simple glisser-déposer (drag and drop)[24]. Le réseau a lui aussi évolué via les WebSockets[25] qui permettent une communication asynchrone avec les serveurs. Naturellement, les ressources disponibles pour l'exécution ou le stockage restent limitées pour chaque page Internet, et les optimisations demeurent indispensables pour produire des applications offrant le même niveau de qualité que les logiciels conçus pour le Desktop.

C'est dans cette perspective d'optimisation, et afin de rendre possible des traitements lourds tels que le rendu 3D, audio ou vidéo[26], que de nouveaux types de données binaires ont été créés par le consortium *Khronos*[27]. Ces types de données, nommés *typedArrays*[28] et *arrayBuffers*, étaient à l'origine destinés à *WebGL*, l'API *HTML5* native dédiée au rendu 3D au sein des navigateurs. Mais leur efficacité et les besoins croissants en performance ont provoqué la généralisation de ces structures à des domaines applicatifs dépassant celui du rendu 3D, comme celui des *WebSockets* par exemple, ou, plus généralement, pour toute application ayant besoin de manipuler efficacement des structures binaires[29].

Beaucoup d'applications effectuent des traitements dont le volume et la lourdeur peuvent rapidement nuire à la réactivité de l'interface graphique, ce qui oblige à utiliser des *threads* afin d'éviter tout blocage. Par ailleurs, les fréquences d'horloge des unités de calcul ont tendance à rester stables, tandis que leur nombre de cœurs d'exécution augmente, ce qui plaide également en faveur du « multithreading ». C'est dans cette gestion des *threads*, devenue incontournable, que les *WebWorkers*[30], interviennent, pour ouvrir et simplifier le champ des possibles.

[22] http://dev.w3.org/html5/webstorage/
[23] http://www.w3.org/TR/offline-webapps/
[24] http://gmailblog.blogspot.fr/2010/04/drag-and-drop-attachments-onto-messages.html
[25] http://tools.ietf.org/html/rfc6455 et http://www.w3.org/TR/websockets/
[26] http://www.w3.org/html/wg/drafts/html/master/single-page.html#htmlmediaelement et http://www.w3.org/TR/html-media-capture/
[27] http://www.khronos.org/
[28] http://www.Khronos.org/registry/typedarray/specs/latest/
[29] http://www.w3.org/TR/websockets/#dom-websocket-binarytype
[30] http://www.whatwg.org/specs/web-apps/current-work/multipage/workers.html

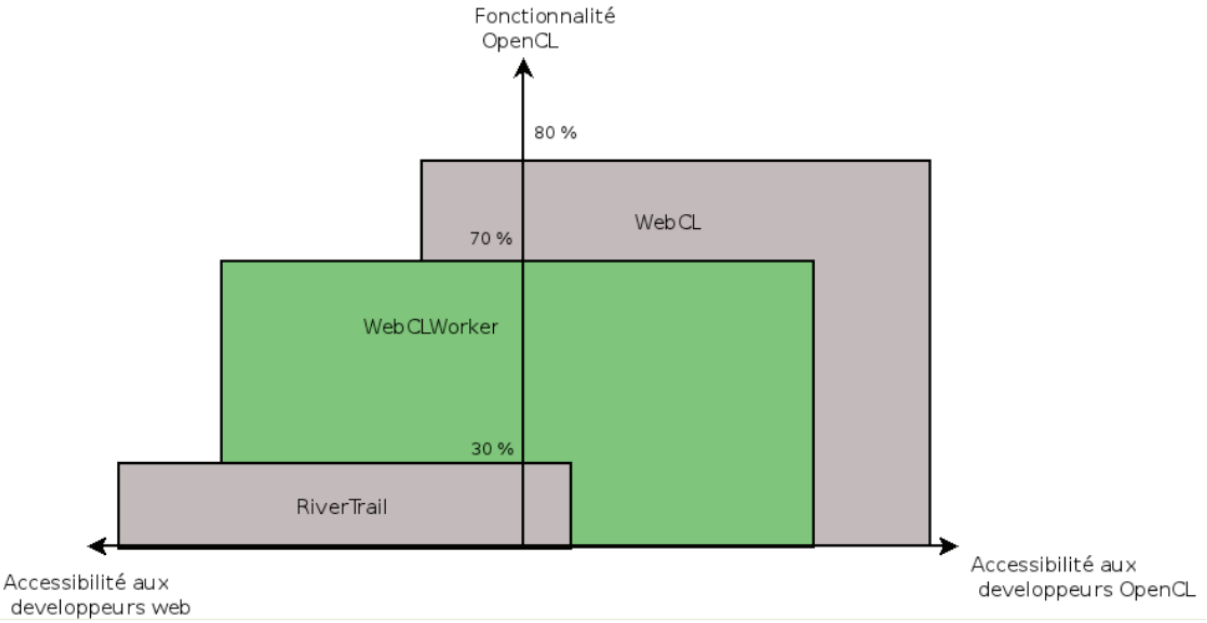


Figure 30 - Les WebCLWorkers offrent aux développeurs Web un accès simple et rapide à la plupart des concepts d'OpenCL en privilégiant l'accessibilité à la complétude. WebCL, qui est en phase de réflexion et sera proposé par le groupe Khronos, privilégie la complétude et se destine aux développeurs bas niveau. Nous travaillons actuellement de concert afin de définir un socle commun. Une fois ce socle technique défini, les temps d'exécution, les empreintes mémoire et les temps de transfert de données seront sensiblement les mêmes. Concernant Rivertrail (une proposition d'Intel pour permettre le parallélisme dans *JavaScript*), les possibilités d'optimisation offertes aux développeurs restent très limitées.

Comme précisé plus haut, le monde de la 3D n'est pas en reste. Il s'est lui aussi étendu au Web avec l'arrivé de *WebGL* et de la version 11 de Flash. Une preuve de l'adoption rapide du nouveau standard de *Khronos* par la communauté est donnée par l'apparition de « *frameworks* » tels que *SpiderGL* [DI BENEDETTO ET AL. 2010] offrant une couche haut niveau et des outils dédiés pour la visualisation en temps réel d'objets 3D au sein d'un navigateur.

Ainsi, comme le montre cette rapide évocation des dernières évolutions technologiques du Web, l'écart entre les applications pour clients lourds (desktop) et les applications pour clients légers (navigateurs Web) s'est considérablement réduit. Pourtant, il reste encore aujourd'hui une importante différence qui distingue les 2 types de plates-formes : il s'agit de la gestion du GPU. En effet, depuis quelques années, la puissance croissante des *GPU* les ont porté progressivement vers d'autres types de tâches que l'affichage 2D et 3D. Ils sont devenus aujourd'hui programmables, offrant ainsi des capacités proches des *CPU* tout en restant différents dans leur architecture, comme le montre la figure 31. En quelques années, des API de calcul parallèle comme *CUDA* ou *OpenCL* se sont imposées. Elles permettent d'utiliser la puissance du *GPU* pour des tâches diverses dépassant le simple cadre de la « rasterisation ». Or, si le Web commence à tirer parti efficacement des *GPU* pour l'affichage 3D avec *WebGL*, l'utilisation efficace de cette ressource et le partage de tâches généralistes entre le *CPU* et la carte graphique est pour l'instant impossible au sein d'un navigateur Web. Ainsi, les applications récentes,

particulièrement gourmandes en ressources et nécessitant l'exploitation du *GPU* en renfort du *CPU* restent inaccessibles aux clients légers.

A l'heure actuelle, il est impossible d'effectuer dans un navigateur Web et en temps réel ou interactif un test de collision sur des centaines d'objets, de simuler une interaction de particules (n-body) ou de visualiser un objet en adaptant son niveau de détail suivant le point de vue, comme le proposent par exemple – hors Web – les travaux récents de *Hu et al.* [HU ET AL. 2009], qui suggèrent une utilisation exclusive du GPU.

Pour parvenir aux résultats présentés dans cette thèse, nous sommes parti de la problématique particulière de la visualisation 3D sur client léger, problématique qui nous a conduit à identifier puis à développer les dernières fonctionnalités manquantes aux navigateurs pour le portage des applications les plus coûteuses en termes de calcul. Notre objectif initial était donc de développer un outil permettant d'implémenter un visualiseur de scènes 3D capable de calculer à la volée les différents niveaux de détail à afficher. Pour cela, nous nous sommes appuyés sur les API fournies par *HTML5* et avons mis au point les *WebCLWorkers*, dont la portée a finalement dépassé le simple cadre du rendu 3D temps réel.

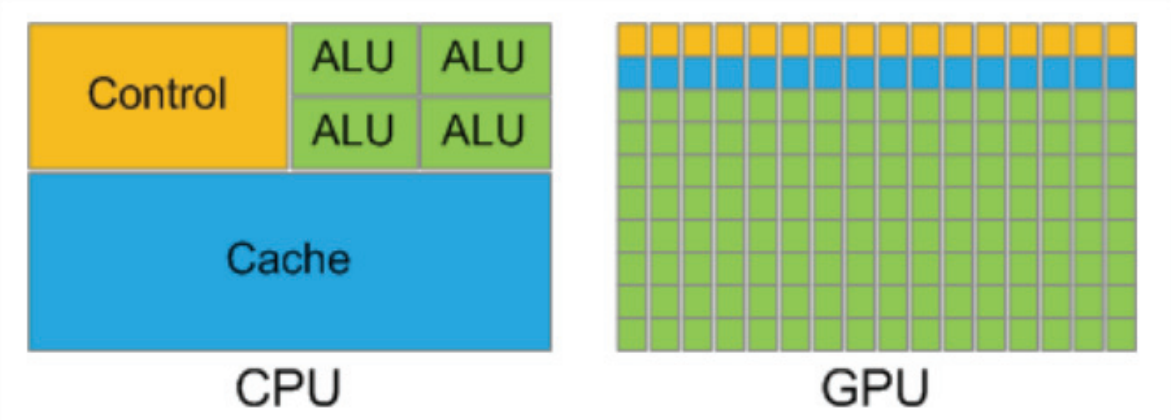


Figure 31 - A l'inverse du CPU, l'architecture du GPU repose sur un grand nombre de cœurs avec des mémoires caches partagées par groupes de cœurs.

Comme le montre la figure 31, le *CPU* possède relativement peu de cœurs d'exécution ainsi qu'un cache global, alors que le *GPU* est plus efficace lorsque l'on sépare le travail sur de nombreux cœurs et que l'on utilise uniquement des caches locaux. Ainsi, de notre point de vue, les dernières étapes nécessaires pour profiter, sur clients légers, d'applications riches comparables à celles qui tournent aujourd'hui sur clients lourds sont :

– Le parallélisme sur les données : la multiplication des cœurs dans les *GPU* et les *CPU* plutôt que l'augmentation de la fréquence d'horloge impose de revoir la manière dont sont conçus les algorithmes. Le parallélisme est développé depuis des années, en utilisant un processus par aspect fonctionnel (par exemple, un premier *thread* gère les entrées et sorties tandis qu'un second thread gère l'affichage). Cet aspect de la programmation concurrentielle a été formalisé mathématiquement en 1973 par le

« modèle d'acteur » [HEWITT 1973]. L'idée étant que chaque composant logiciel est acteur (de façon analogue aux objets) et réagit à la réception d'un message asynchrone d'une manière qui lui est propre, c'est-à-dire par la création d'un nouvel acteur, l'envoi de message à d'autres acteurs, ou simplement la déconnexion de sa boîte (ou de ses boîtes) de réception. Afin d'éviter tout problème d'accès concurrent, il existe deux types d'acteurs : les premiers sont « stateless » et peuvent recevoir et traiter plusieurs messages de façon simultanée ; les autres sont « statefull » avec la particularité de ne traiter qu'un message à la fois et de ne permettre à personne d'autre qu'eux-mêmes de modifier leur état. Par ailleurs, une contrainte supplémentaire est imposée aux deux types d'acteurs : tous les messages (envoyés et reçus) sont immutables, c'est-à-dire qu'ils ne peuvent pas être modifiés durant l'exécution du programme. Ce type de parallélisme est d'ores et déjà envisageable dans un navigateur Web avec les *WebWorkers*, l'API *HTML5* conçue pour autoriser l'exécution de scripts *JavaScript* en tâche de fond. Cependant, ces *WebWorkers* ne permettent pas le traitement parallèle sur un même jeu de données. Indépendamment de cette contrainte propre à *HTML5*, il est difficile d'obtenir un parallélisme de données efficace avec le modèle des acteurs, à cause du caractère immutable des messages qu'ils manipulent, qui impose bien souvent une duplication des données.

– La gestion des ressources et l'adaptation au client : suivant l'hôte de l'application, la puissance disponible peut varier. Certains ordinateurs pourront posséder des *CPU* très puissants sans carte graphique, d'autres deux *GPU* avec plusieurs gigaoctets de mémoire, etc. C'est pourquoi, afin d'exploiter au mieux tout le matériel disponible (en particulier les unités de calcul comme le *CPU* ou le *GPU*), il est nécessaire de connaître ses caractéristiques et de distribuer les tâches en conséquence. En outre, suivant le nombre de *threads* et les contraintes de synchronisation, certains traitements exploiteront mieux le *CPU* alors que d'autres seront plus adaptés au *GPU*.

C'est sur la base de ces observations que nous avons créé les *WebCLWorkers*, premier outil Web de programmation généraliste du *GPU*, mais qui permet également une meilleure efficacité dans l'utilisation des *CPU*, du fait de l'utilisation possible de pointeurs. Nous avons choisi l'API *OpenCL* comme point de départ car elle présente l'avantage sur sa concurrente *CUDA* d'utiliser un langage compilé à la volée qui peut parfaitement s'intégrer à un site Internet.

Notons que la solution de Google sur le sujet, le plugin Native Client [YEE ET AL. 2009], consiste à compiler des applications en code machine et à les exécuter dans la *sandbox* du navigateur, c'est-à-dire en imposant des contrôles sur les accès aux ressources pour des raisons de sécurité. Cette approche nécessite d'avoir un binaire pour chaque type de processeur, et bien qu'elle puisse répondre à de nombreuses problématiques en offrant une

grande flexibilité, nous sommes convaincu qu'il est possible de répondre à la question du GP/GPU sans recourir à des binaires et en conservant une approche plus traditionnelle et compatible avec le fonctionnement actuel d'Internet. Nous souhaitions en particulier éviter le remplacement pur et simple du *JavaScript* par un autre langage (le C++ en l'occurrence), et préférons offrir des outils de programmation du *GPU* utilisables optionnellement pour les portions de code critiques du point de vue des performances. En effet, nos *WebCLWorker*s sont destinés à être utilisés avec des scripts publiés au niveau du DOM (Document Object Model), de la même manière que les *WebWorkers* ou que les *shaders* de *WebGL*, deux solutions de l'API standard *HTML5*.

Intel a également proposé, à travers son projet RiverTrail[31], d'ajouter du parallélisme au langage *JavaScript*. Cependant, l'API proposée est centrée sur l'intégration dans le code *JavaScript* et ne permet pas de contrôler le dispositif (*device*[32], c'est-à-dire l'unité de calcul) qui sera utilisé, le type de synchronisation entre les exécutions, ou encore la gestion précise de la mémoire. Ainsi, nous sommes convaincus que notre travail n'entre pas en concurrence avec celui d'Intel, mais permet de le compléter. D'ailleurs, à l'heure où nous écrivons ces lignes, Rivertrail peut parfaitement être implémenté au-dessus des *WebCLWorkers* ou avec *WebCL*.

Afin de démontrer la validité et l'efficacité de notre approche, nous avons enrichi le navigateur Firefox puis soumis nos résultats au consortium *Khronos* en septembre 2011, à l'occasion d'un meeting sur *WebCL*. Nous travaillons désormais étroitement avec *Khronos* sur le futur standard WebCL.

Notre travail se subdivise en deux parties : un « *framework* » qui nécessite des modifications du code source de Firefox (*patch* d'environ 6000 lignes), et une version open source qui est une réécriture permettant d'utiliser les *WebCLWorkers* comme un simple plug-in du navigateur. Actuellement, la version open source impose d'appliquer un autre *patch* de 200 lignes pour pouvoir interagir avec *WebGL*. Les modifications apportées par ce *patch* n'étant pas spécifiques à notre travail, une discussion avec les développeurs de Firefox est en cours pour les intégrer dans une prochaine version du navigateur. C'est pourquoi certaines parties décrites ici ne sont pas encore disponibles en open source.

Fin Mars 2011*, Khronos* annonçait un groupe de travail sur le développement d'un outil de *GP/GPU* baptisé *WebCL*. C'est pourquoi, ayant conçu et réalisé la toute première implémentation utilisant *OpenCL* à partir d'un navigateur Internet, nous avons fait converger un socle commun entre *WebCL* et les *WebCLWorkers*, tout en apportant à la communauté

[31] https://github.com/RiverTrail/RiverTrail/wiki

[32] Dans cette thèse nous utiliserons le mot « device » dans sa version anglaise pour désigner une unité de calcul. Ce choix a été fait en accord avec le vocabulaire de référence du domaine de la haute performance. En effet, que ce soit dans les articles ou dans les API de référence comme OpenCL ou Cuda, les mots « devices », « context », « buffer » et « commandQueue » sont devenus incontournables et avec une connotation précise liée au domaine, d'où notre choix de conserver ces termes. « buffer » et « comandQueue » sont devenus incontournables et avec une connotation précise liée au domaine, d'où notre choix de conserver ces termes.

l'expérience que nous avons acquise sur le sujet. De plus, les *WebCLWorkers* devraient constituer au final une couche de plus haut niveau que *WebCL,* offrant au développeur Web non spécialiste une API plus simple et plus accessible pour des performances comparables.

Grâce à l'avance que nous avons prise dans le domaine, nous participons aujourd'hui activement à l'intégration de *WebCL* dans Firefox et partageons notre expérience et notre point de vue avec les initiateurs de *WebCL* au sein du consortium *Khronos* (principalement Nokia et Samsung). Même si les propositions de « drafts » de *WebCL* seront vraisemblablement très proches des *WebCLWorkers* sur le fonctionnement, il existe une différence fondamentale entre les deux approches : le but du *WebCL* de *Khronos* est de fournir une API la plus conforme possible à *OpenCL* afin que les développeurs familiers de cette technologie ne soient pas perturbés lors du passage vers la programmation Web ; Dans le cas des *WebCLWorkers*, nous avons choisi de porter *OpenCL* dans Firefox et d'y adjoindre une couche de plus haut niveau offrant une interface aussi proche que possible de l'API de parallélisme amenée à devenir un standard pour les développeurs Web, à savoir les *WebWorkers* de *HTML5*.

Enfin, étant donné que les couches basses de *WebCL* et des *WebCLWorkers* seront très proches, nous travaillons actuellement avec Mozilla afin que ce socle commun, provenant de nos développements, soit intégré directement dans Firefox[33].

3.2. Limitations du *HTML5/JavaScript*

Malgré les progrès réalisés récemment par les derniers moteurs *JavaScript*, ces derniers continuent d'afficher des performances sensiblement inférieures à celle du langage C, même si ce dernier est « monothreadé »[34]. C'est pourquoi diverses méthodes d'optimisation ont été développées, comme l'utilisation d'un compilateur "Just In Time"[35], afin de s'approcher des performances des langages natifs et ce, quel que soit le type d'algorithme utilisé (parcours d'arbres binaires, recherche d'expression régulière, etc.). Néanmoins, malgré ces avancées, il ne sera jamais possible, de par la structure même du *JavaScript*, d'écrire des algorithmes bas niveau aussi optimisés qu'avec du langage C. Cela constitue un obstacle majeur au développement d'applications pour lesquelles les performances sont cruciales, comme c'est le cas pour notre objectif de visualisation de modèles de terrain 3D en temps réel.

Un autre verrou de *HTML5* est l'impossibilité de traiter des données en parallèle à l'intérieur d'un même code *JavaScript* : les *WebWorkers* précédemment évoqués peuvent s'apparenter à des processus systèmes (au sens POSIX du terme) ayant chacun leur propre zone mémoire et communicant à travers des messages de type « mailbox ». Ainsi, la

[33] https://bugzilla.mozilla.org/show_bug.cgi?id=664147
[34] http://benchmarksgame.alioth.debian.org/u64/benchmark.php?test=all&lang=v8&lang2=gpp&data=u64
[35] http://en.wikipedia.org/wiki/Just-in-time_compilation

consommation mémoire de chacun des *WebWorkers* n'est pas négligeable, et le transfert de données utilise une sérialisation *JavaScript* qui peut, dans le cas de grandes quantités de données, avoir des répercussions significatives sur les performances.

Il existe des solutions en dehors du Web qui permettent d'instancier des millions de *workers*, comme le font par exemple les langages *Scala* ou *Erlang*[36] avec leurs modèles d'acteurs [HEWITT 1973]. Mais ce modèle, souvent désigné par « SHARE NOTHING », n'est pas adapté aux architectures de type *GPU* et en particulier aux algorithmes parallèles évoqués précédemment. En effet, les GPU sont composés d'un nombre conséquent de processeurs (parfois plus de mille) mais dont la vitesse d'exécution séquentielle est plus lente que les processeurs CPU classiques.

C'est pourquoi la multiplication des cœurs a fait apparaître un nouveau type de parallélisme, se basant sur les données, qui n'est aujourd'hui exploité que par les « *frameworks* » de *GP/GPU* – hors Web – comme *CUDA* ou *OpenCL*. Ces solutions permettent en particulier d'éviter le va-et-vient entre les différents *CPU* et/ou *GPU* en conservant les données dans les zones mémoire associées à l'unité de calcul. Une conséquence immédiate de l'adaptation de cette méthode aux technologies du Web serait de pouvoir interagir directement avec le « canvas », qu'il soit 2D ou 3D (*WebGL*), sans devoir préalablement récupérer les données au niveau du *JavaScript*. Les *WebCLWorkers* que nous proposons, inspirés par *OpenCL*, exploitent ce nouveau type de parallélisme et offrent ainsi la possibilité de traiter les données 3D sur le *GPU,* tout en autorisant simultanément leur affichage dans les pages HTML.

Notre solution ne remplace pas *JavaScript* mais permet de compléter ses possibilités, et les apports sont suffisants pour justifier l'utilisation, non pas d'une évolution du *JavaScript*, mais d'un tout nouveau langage utilisé dans les WebCLWorkers (à l'instar des *shaders* de *WebGL*) : que nous présentons dans la partie suivante.

Dans cette proposition, notre approche a été de limiter l'apprentissage des développeurs venant d'*OpenCL* tout en minimisant les impacts pour les développeurs web familiers des *WebWorkers JavaScript*.

On pourrait objecter qu'il serait possible de simuler une partie du parallélisme apporté par notre solution en utilisant l'API *WebGL* avec des textures : l'idée consisterait à utiliser les *shaders* pour les traitements et les textures pour lire et écrire les résultats sans que les données ne soient transférées hors du *GPU*. Cependant, cette solution ne prendrait pas en compte les différentes ressources disponibles sur l'ordinateur et interdirait l'utilisation d'algorithmes nécessitant des parties de synchronisation. Nous pensons par exemple, dans le domaine de la visualisation 3D, à la simplification côté *GPU* proposée par *Hu et al.* [HU ET AL. 2009], ou encore au *Map And Reduce*[DEAN ET AL. 2004] sur GPU [ELTEIR ET AL. 2011] optimisant le traitement des données 3D. Ces 2 méthodes seraient impossibles à

[36] http://savanne.be/articles/concurrency-in-erlang-scala/

implémenter par cette technique dans la mesure où elles utilisent des données de taille variable en entrée et en sortie et nécessitent des points de synchronisation dans leur exécution. Par ailleurs, il est souhaitable, dans de nombreux cas, de pouvoir traiter les données sans bloquer leur visualisation comme le ferait un traitement *WebGL*. Enfin, les *WebCLWorkers* apportent la possibilité, au sein d'un navigateur Web, de répartir un traitement entre différents types de processeurs, *GPU* et *CPU*, tout comme le permet actuellement *OpenCL*.

3.3. Les WebCLWorkers

Bien que notre travail ait été validé sur le navigateur Firefox proposé par Mozilla, il n'existe à notre connaissance aucune difficulté pour adapter à tout autre navigateur les principes développés dans cette section.

3.3.1. Principes de la méthode

Le but des *WebCLWorkers* est de s'approcher le plus possible, en termes de fonctionnement et d'utilisation, des *WebWorkers* apparus dans *HTML5*, en tentant de conserver les notions d'*OpenCL* et la plupart des possibilités offertes par la programmation générique sur *GPU* (*GP/GPU*). Comme décrit dans la figure 32, la notion de *worker* correspond à la fusion d'un contexte et d'une queue de tâches à exécuter. Ainsi, un *WebCLWorker* peut s'apparenter à un *WebWorker* qui s'exécuterait sur une ressource choisie par le développeur.

Comme le montre la figure 32, nous avons légèrement simplifié l'API *OpenCL* afin de rester aussi proche que possible de l'API *HTML5* actuelle. Alors que les notions de mémoire et de *device* nous paraissent claires, le contexte n'est pas une notion évidente pour un développeur Web classique. C'est pourquoi nous avons choisi de l'intégrer dans l'objet *WebCLWorker*, tout comme la file d'exécution des tâches. L'objet résultant devient ainsi un équivalent des *WebWorkers* (si l'on omet bien sûr la manière dont les tâches sont traitées). Une tâche est définie par un kernel *OpenCL* auquel on adjoint toutes les options de configuration nécessaires à son exécution, c'est-à-dire le nombre de *threads* qui vont être utilisés ou les paramètres d'appel de la fonction. Ces *threads* sont définis par des *WorkItems* et des *WorkGroups*. Les *WorkItems* peuvent être apparentés aux *threads* et sont réunis en *WorkGroups*. Les *WorkGroups* sont des ensembles de *WorkItems* qui seront exécutés sur la même unité de calcul (les plates-formes *OpenCL* définissent l'unité de calcul, qui peut être un *CPU* ou un « core » de *CPU*, etc.). Les unités de calcul peuvent, quant à elles, exécuter plusieurs *WorkGroups* simultanément. Cette nuance peut être importante pour l'optimisation des traitements et la synchronisation des données pendant l'exécution des programmes.

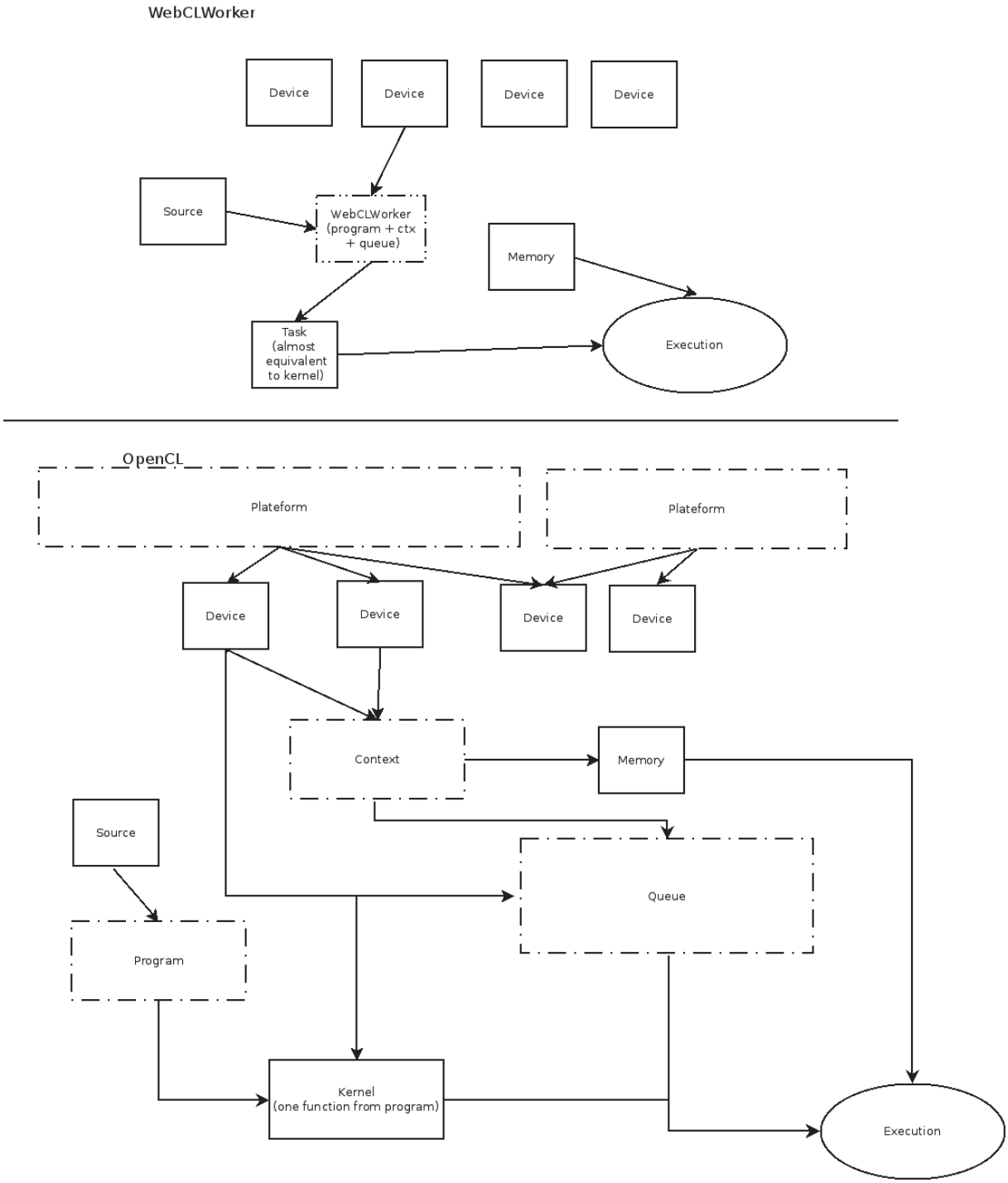


Figure 32 - En limitant le contexte à un seul device (contrairement à WebCL, dont la seule restriction par rapport à OpenCL est que les différents devices appartiennent à une même plate-forme), les WebCLWorkers constituent une simplification d'OpenCL, grâce à la fusion des contextes, programme et queue d'exécution.

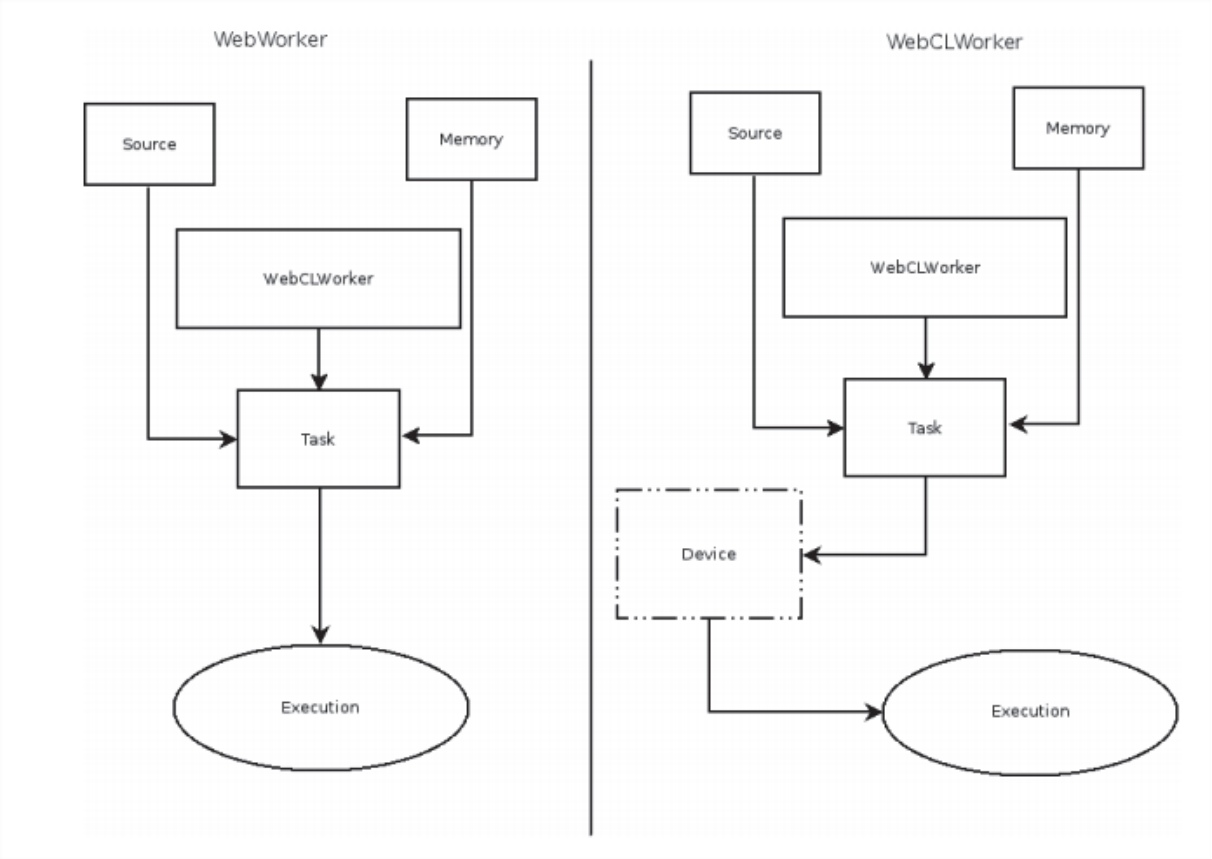


Figure 33 - Comme le montre la figure, à part la notion de device, les WebWorkers et les WebCLWorkers restent très proches dans leur architecture et leur utilisation. La notion de device apporte le choix de l'exécution sur CPU ou GPU.

L'annexe 2 donne un exemple concret d'utilisation des *WebCLWorkers* pour une multiplication de matrice. Elle permet de comparer la simplicité et la longueur du code nécessaire à la programmation de cette tâche dans les 3 API : *WebCL* (prototype actuel de Samsung), les *WebCLWorkers* et les *WebWorkers* de *HTML5*.

Les *buffers* utilisés sont représentés par les *WebCLMem* qui en pratique peuvent être des *buffers WebGL* ou simplement des données provenant du code *JavaScript* comme des images ou des tableaux natifs *typedArray.*

Sous Firefox, l'interaction optimisée entre des éléments *JavaScript* d'une page et les composants natifs *typedArray*, utilise un mécanisme baptisé *quickstub*, qui n'est malheureusement pas accessible aux extensions. Sans ce mécanisme, il est impossible d'effectuer des appels aux fonctions des *WebCLWorkers* en moins de 3 ms. En effet, le typage dynamique de *JavaScript* impose une vérification systématique de chaque élément de tableau, ce qui pénalise l'échange de données entre *JavaScript* et C++. Pour éviter ce surcoût, lorsqu'un appel de fonction contient un tableau, nous avons mis en place des listes containers dont tous les éléments sont de même type.

Les *WebCLTasks* sont l'équivalent des *kernel OpenCL* avec toutes les données nécessaires pour lancer une exécution sur le *device*. Ainsi, ils permettent au *JavaScript* d'injecter tous les paramètres nécessaires à l'exécution tels que les données utilisées, les variables pour le traitement, mais aussi le nombre de fois que la fonction du *script CL* du *WebCLWorker* doit être appelée simultanément.

Comme le montrent les figures 32 et 33, nous avons dû ajouter aux *WebWorkers HTML5* les notions de *device* et de plate-forme. De même, la notion de tâche a fait son apparition pour permettre de configurer les données et le nombre de *threads* utilisés pendant l'exécution : cet ajout reflète la capacité des *devices* à traiter les données de façon parallèle.

Enfin, l'utilisation des événements à travers le DOM permet de limiter la quantité de *JavaScript* nécessaire à la synchronisation, tout en restant proche des *WebWorkers*. Toujours dans une optique de simplicité pour le développeur, et pour se conformer au fonctionnement des *WebWorkers*, il a été choisi de ne partager aucune zone mémoire entre les *workers* et le *JavaScript*. L'avantage principal de notre solution est de pouvoir répartir le travail entre le *CPU* et le *GPU*, en exploitant efficacement toutes les ressources disponibles.

3.3.2. Gestion des traitements de données

L'un des intérêts des *WebWorkers* est le travail asynchrone et la notification des résultats à travers les événements, libérant le *JavaScript* pour effectuer d'autres tâches. À noter que dans les versions récentes de Firefox et de Chrome, chaque onglet possède un contexte d'exécution de *JavaScript* et que chacun de ces contextes s'exécute dans son propre *thread* ou processus. Dans les versions antérieures à la version 8 de Firefox, tous les *JavaScript*s s'exécutent simultanément dans le *thread* principal, c'est-à-dire celui qui s'occupe aussi de l'affichage de l'application.

Ainsi, à cause de ces contraintes et de la gestion des événements et des données accessibles au code *JavaScript*, il a été nécessaire de dis*patch*er les événements par le *thread* principal ou celui associé à la page Web suivant la version de Firefox (en se basant sur la méthode NS_Dis*patch*ToMainThread dans les versions de Firefox inférieure à 8 ou sur NS_GetCurrentThread dans les versions plus récentes, afin de trouver le bon *thread* pour l'envoi d'événements). L'exécution de certaines actions, uniquement dans le *thread* de la page, présente à la fois l'avantage de pouvoir s'assurer qu'aucun traitement ne sera fait de façon concurrente mais aussi d'éviter ainsi tout problème de blocage ou d'erreur de traitement des variables.

Bien que cela puisse, en théorie, provoquer des saccades lors de la navigation, nous n'avons pas observé de ralentissement lors de nos tests, pour une utilisation normale. Enfin, dans les prochaines versions de Firefox, l'affichage, les traitements *WebGL* et les traitements *WebCL* se feront dans le même processus, séparé du reste de l'application, mais dans des *threads* indépendants. L'interaction avec le code *JavaScript* se fera, quant à elle, à travers

des classes proxy qui gèreront l'aspect asynchrone. Ce modèle, semblable au modèle d'acteur cité précédemment, est parfaitement adapté pour ce cas d'utilisation : chaque processus ou *thread* doit travailler de façon autonome sur de longs traitements ayant leur propre « stack » (*JavaScript*) et ce, sans que les opérations à effectuer ne soient les mêmes.

Ainsi, dans le cas de notre exemple de visualisateur 3D, les opérations s'exécutant de façon asynchrone, elles permettent de laisser le *JavaScript* traiter le choix des simplifications et de ne lancer de nouveaux traitements que lorsque le *device* a terminé les précédents. Ainsi, si la caméra de la scène se déplace rapidement d'un point A à un point B, le *JavaScript* pourra choisir a priori de réaliser uniquement le raffinement des points A et B, sans se préoccuper des étapes intermédiaires, mais s'il reçoit des événements entretemps (à mi-chemin entre A et B par exemple), il pourra alors choisir de lancer un raffinement intermédiaire.

3.3.3. Gestion de l'échange des données

L'utilisation de tableaux à partir du code *JavaScript* peut rapidement devenir problématique, et ce pour plusieurs raisons. La première est une conséquence du typage implicite du langage *JavaScript* : chaque élément d'un tableau doit être vérifié et « casté » avant de pouvoir être utilisé, ce qui peut rapidement pénaliser les performances. Heureusement, *WebGL* a apporté récemment un nouveau type de données, les *arrayBuffer*, qui ne sont autre qu'une zone mémoire pouvant être interprétée de différentes manières à l'aide de « viewers », les *typedArrays*[37]. Cependant, cette zone mémoire ne peut pas être utilisée par des *threads* concurrents. Or, une possibilité offerte par *OpenCL* est d'utiliser directement une zone mémoire de l'hôte, ce qui peut s'avérer très utile lors de traitements volumineux par le *CPU*, comme par exemple un tri de tableau. Une solution serait de créer un *wrapper* bloquant l'accès à la zone depuis le contexte *JavaScript* pendant le traitement par le *device*, en utilisant les notifications envoyées par *OpenCL*. Cependant, on ne pourrait alors pas empêcher l'utilisation d'une zone mémoire simultanément par plusieurs *workers*, à moins de mettre en place des vérifications ordonnançant les tâches sur le principe des *mutex*. Pour l'instant, nous n'avons pas trouvé de solution s'adaptant aux API *OpenCL* sans introduire un risque d'inter-blocage inacceptable dans le cas de la manipulation de données (contrairement à ce qui a été permis pour les événements, étant donné que cette possibilité était déjà présente en utilisant simplement *JavaScript* et le DOM). Nous avons donc décidé que les *WebCLWorkers* ne travailleraient que sur des objets non directement disponibles depuis le contexte *JavaScript*. Pour des raisons de performances, il pourrait être envisagé d'utiliser un mécanisme qui, lors de l'appel d'une fonction, basculerait la zone mémoire entre le *JavaScript* et chacun des *WebCLWorkers*, la référence de l'objet devenant nulle dans le contexte où ce dernier a disparu. Cette solution permettrait d'économiser l'espace nécessaire à une copie, tout en diminuant les temps de transfert. Dans les toutes dernières

[37] http://www.Khronos.org/registry/typedarray/specs/latest/

évolutions de *HTML5* et des *typedArrays*, une discussion sur l'échange efficace de données sans copie a fait son apparition (fin juin 2011). La solution retenue serait semblable à notre méthode : un échange sans copie entre les différentes zones mémoire *JavaScript* pour les *WebWorkers*, mais aussi *WebCL*.

L'avantage de notre proposition par rapport à celle de l'actuelle spécification *HTML5*, est qu'elle permet d'éviter la création d'un grand nombre d'objets qui ralentissent le code *JavaScript* et provoquent parfois des latences. En effet, afin de libérer la mémoire, le ramasse-miettes *(garbage collector)* doit tester chacun des objets. De plus, en imposant un *swap* de la mémoire plutôt que son invalidation, nous gagnons encore en rapidité d'exécution en évitant le parcours de tous les objets attachés à la zone mémoire.

Pour notre test, nous avons choisi de prendre comme étalon un transfert de 180 Mo par seconde, ce qui correspond à l'affichage d'une vidéo HD (de 1920 par 1080 pixels) en 24 bits de profondeur par couleur, à 30 images par secondes.

Dans les autres cas, c'est-à-dire lorsqu'*OpenCL* travaille sur des données qu'il a préalablement copiées dans un cache, la solution retenue est l'utilisation d'un *wrapper* local contenant une copie locale immutable et bloquée tant que la synchronisation avec le *device* n'a pas été effectuée. En cas de tentative d'accès en lecture ou écriture à l'objet bloqué, le navigateur renvoie immédiatement une erreur et ce, tant qu'il n'a pas reçu de notification indiquant que le *device* a terminé ses opérations. L'utilisateur a aussi la possibilité de transférer les données en mode synchrone. Dans ce cas, toute méthode appelée ne libérera le contrôle de l'exécution du programme qu'à la fin du transfert.

Pour notre exemple de représentation de scènes 3D volumineuses, un tableau binaire permet de stocker des structures de type *octree* optimisées pour la recherche, tout en réduisant l'empreinte mémoire. De plus, en optimisant le transfert des données binaires entre le code *JavaScript* et les *WebCLWorkers*, le traitement peut être partagé efficacement entre plusieurs *devices*, et effectué partiellement par le *JavaScript*.

Nombre de morceaux utilisés	Transfert par copie (*clone memory* de Firefox 10)	Transfert par Swap
1 (180 Mo)	100 ms	6 ms
10 (18 Mo)	88 ms	6 ms
30 (6 Mo)	90 ms (+/- 5%)	6 ms
180 (1 Mo)	130 ms (+/- 15%)	6 ms

Figure 34 - Transfert de données correspondant au décodage d'une vidéo HD.

On remarque que le temps consacré au simple transfert est d'environ 100 ms pour le *JavaScript* dans le meilleur des cas (performances stables avec des morceaux entre 5 et 20 Mo), alors que le principe de *swap* utilise seulement 6 ms quelle que soit la quantité de données (évitant en outre le gel des calculs durant les *callocs* des zones mémoire). Les variations pour la dernière expérience (paquets de 1 Mo) peuvent s'expliquer par les appels au ramasse-miettes qui peuvent varier d'une exécution à l'autre.

3.3.4. Sécurité

La question de la sécurité est primordiale dans les usages Web au sein de navigateurs, et elle est susceptible d'impacter les choix d'implémentation et de performance pour les solutions de calcul parallèle. Nous lui consacrons donc cette importante partie qui présente le contexte, l'état de l'art, et la suite de nos propositions.

3.3.4.1. Introduction

L'API de protection des *WebCLworkers* est similaire à celle du *WebCL* et les travaux réalisés sont partagés avec le groupe de standardisation afin de constituer le socle d'une implémentation industrielle financée par *Khronos* et réalisée par Vincit. C'est pour cette raison que dans la suite, nous ne ferons pas la distinction entre *WebCL* et *WebCLWorkers.*

Avec les nouvelles évolutions du Web, il sera bientôt possible d'utiliser des API de calcul hautes performances sur *GPU* et *CPU* comme *OpenCL* directement dans le navigateur. Or, ce type d'API de bas niveau soulève la question de la sécurité au sens large. De plus, ce thème devient particulièrement sensible pour les navigateurs, qui ont la responsabilité d'exécuter du code non vérifié et potentiellement dangereux et ce, sans intervention ni supervision de l'internaute. C'est pour cette raison que la nouvelle API de calcul intensif – *WebCL* ou *WebCLWorker* – doit fournir une protection capable d'assurer un niveau de sécurité suffisant pour une utilisation au sein d'un navigateur Internet, sans pour autant introduire de ralentissement significatif dans la vitesse des traitements *OpenCL*. Pour cela, nous avons mis en place une solution de transcompilation ajoutant les vérifications statiques et dynamiques nécessaires pour assurer un fonctionnement à la fois sûr et performant. Pour les lecteurs familiers de la modélisation 3D et qui ne sont pas coutumiers des problématiques de programmation pure, la transcompilation est l'action de transformer un code source vers un autre code source (éventuellement au sein d'un même langage de programmation). Nous reviendrons en détail sur la transcompilation en section 3.3.4.3.

3.3.4.2. Etat de l'art

A notre connaissance, il n'existe pas de travaux spécifiques à la sécurité dans *OpenCL.* Cependant, des travaux pour l'exécution de code binaire, ou byte code, au sein des navigateurs ont été menés chez Google à travers la réalisation de « Native Client » [YEE ET AL. 2009].

Cette approche a été étendue pour fonctionner sur des architectures ARM et x86_64 avec du code portable issu d'une représentation intermédiaire bas niveau, LLVM IR, dont la dernière étape de compilation est réalisée par le navigateur.

Le principe de cette approche est d'utiliser deux *sandboxes*. La première, classique et externe, exécutant le code dans un processus séparé avec des droits réduits, et la deuxième,

plus légère, exécutant le code binaire modifié en insérant des sauts (jmp assembleur) qui interagissent uniquement avec des instructions sûres (comme par exemple des fonctions fournies par l'API de Native Client ou la zone mémoire de données de l'application) .

Il existe dans la littérature d'autres travaux sur la protection de code C qui pourraient être adaptés à *WebCL*. On peut citer par exemple SafeC [AUSTIN ET AL. 1994], MSCC [XU ET AL. 2004], ou encore *SoftBound* [NAGARAKATTE ET AL. 2009] et CETS [NAGARAKATTE ET AL. 2010] qui offrent en grande partie le type de sécurité requise pour le *WebCL*. *SoftBound* [NAGARAKATTE ET AL. 2009] présente l'avantage d'implémenter la protection de la mémoire en moins de 7k lignes et CETS [NAGARAKATTE ET AL. 2010] quant à lui, ajoute la protection des « dangling pointers », c'est-à-dire la gestion d'une zone mémoire réclamée alors qu'il existe toujours des pointeurs qui lui sont attachés.

Cependant, *SafeC* [AUSTIN ET AL. 1994], *SoftBound* [NAGARAKATTE ET AL. 2009] (+ *CETS* [NAGARAKATTE ET AL. 2010]) et *Native Client* [YEE ET AL. 2009] présentent un inconvénient majeur : la protection de la mémoire s'effectue sur la représentation du code en LLVM IR. Or, afin de réaliser correctement la protection sans se priver de certaines fonctionnalités du langage, il est nécessaire de connaître certaines informations, telles que la taille des pointeurs et leur alignement, qui sont spécifiques au *device* (unité de calcul) qui exécutera le code. Cependant, il n'est pas envisageable de gérer tous les *devices* dans le navigateur, et par ailleurs, nous souhaitons prendre en compte le plus de drivers *OpenCL* possible avec un seul langage de programmation. C'est pour cette raison qu'il nous semble préférable que la protection soit capable de produire du code *OpenCL/C* valide et sécurisé. De plus, dans le cas de Native Client [YEE ET AL. 2009], une partie de la protection repose sur des fonctionnalités très proches du matériel (comme le MMU par exemple), alors que dans le contexte du *WebCL*, il est difficile de faire ce genre de supposition. La problématique pour SoftBound [NAGARAKATTE ET AL. 2009] est différente et il est possible de porter cette solution avec les drivers utilisant SPIR, le code binaire intermédiaire d'*OpenCL,* actuellement en cours de standardisation, et inspiré de LLVM IR. Cependant, SPIR n'a pas pour vocation première d'être utilisé avec toutes les architectures possibles, contrairement à *WebCL*.

De plus, bien que ces approches fonctionnent dans le navigateur, comme le prouve Native Client [YEE ET AL. 2009] dans Chrome, elles ne correspondent pas au mode opératoire de WebCL. En effet, elles se basent sur des programmes déjà compilés alors que WebCL doit travailler sur les sources pour valider ces dernières. L'objectif étant une application Web, il est non seulement nécessaire que l'application produise le même résultat sur tous les navigateurs (et *devices* d'exécution) mais aussi que la syntaxe et le code source du programme soit exactement les mêmes pour tous. Bien que cette contrainte ne soit pas directement liée à la sécurité, elle doit obligatoirement être prise en compte pour permettre l'adoption du WebCL par Mozilla, Google et Opera.

Enfin, comme énoncé précédemment, tous ces travaux ne sont pas spécifiques au contexte *WebCL(Workers)/OpenCL* et par conséquent, ils ne peuvent tirer profit des spécificités fonctionnelles de ce dernier.

3.3.4.3. Principe : la transcompilation comme machine virtuelle (VM) légère dédiée à l'exécution de code non sécurisé

Dans nos travaux, nous nous sommes appuyés sur une adaptation de SoftBound [NAGARAKATTE ET AL. 2009], de façon à ce que la sécurité porte sur l'Abstract Syntax Tree (AST) produit par *Clang* et non sur LLVM IR. Nous avons aussi établi de nouvelles extensions telles que l'initialisation de la mémoire, afin d'éviter que les restes d'informations d'autres programmes puissent être récupérés par des attaquants ou programmeurs mal intentionnés. Une autre caractéristique importante de notre travail est que nous avons considéré le *JavaScript* comme un modèle lorsque cela paraissait pertinent. En effet, le *JavaScript* (ou la VM JS) est considéré comme sûr, de notre point de vue, grâce à la *sandbox* mise en place pour le code exécuté. Tout comme pour *OpenCL*, ce code peut être traduit en natif par le *JavaScript* grâce aux utilisations du « Just In Time » *compiler* ou JIT.

Dans notre cas, nous utilisons un transcompilateur qui transforme le code *OpenCL* non sécurisé en code *OpenCL/C* de confiance (« secured »). Nous avons aussi développé un prototype utilisant un sous-ensemble *JavaScript* transcrit en *OpenCL/C sécurisé*. Le mécanisme est inspiré de Rpython[38], une spécification d'un sous-ensemble de Python autorisant le typage statique mais surtout de pouvoir transcompiler le code en C et obtenir des performances comparables à ce dernier. Cette seconde approche est effectuée en parallèle pour garder une certaine neutralité dans notre démarche vis-à-vis de la technologie. En effet, la première est fortement liée au compilateur Clang et c'est pour cette raison qu'il nous paraissait intéressant d'utiliser un transcompilateur JavaScript comme seconde approche.

Bien que ces approches ne soient pas effectuées à travers un JIT, les étapes de recompilation et de validation associées à notre « générateur » de code assurent que le résultat peut être aussi sécurisé que *JavaScript*. En effet, seul le code que nous produisons nous-même sera utilisé et compilé dans le driver *OpenCL,* recréant une *sandbox* légère directement dans le code source.

Nous considérons notre générateur de code comme sûr pour plusieurs raisons. La première est qu'il ajoute directement dans le code C les tests dynamiques lorsque cela est nécessaire (et, dans un futur proche, des vérifications statiques ainsi que certaines optimisations). De plus, afin d'assurer une plus grande sécurité, le transcompilateur échoue plutôt que de produire une sortie contenant des portions de code C qu'il n'est pas capable d'interpréter correctement. Cela permet d'une part d'augmenter la sécurité et d'autre part

[38] http://dl.acm.org/citation.cfm?id=1297091

de valider que le code respecte rigoureusement les spécifications du langage que nous avons fixé comme entrée de notre solution, en empêchant du code ou une extension spécifique à une plateforme ou un matériel d'être exécuté, point très appréciable dans le contexte d'applications Internet.

De plus, notre transcompilateur permet de gommer tout comportement spécifique à un *CPU*, *GPU* ou un système d'exploitation, finalisant la VM légère et réduisant la surface d'attaque. Enfin, cela nous rapproche d'un comportement unique dans tous les navigateurs, quel que soit le dispositif utilisé (ordinateur ou smartphone), ce qui constitue l'une des caractéristiques fondamentales du Web.

Enfin l'utilisation du transcompilateur permet de produire du code *OpenCL/C* compatible avec tous les drivers, que ceux-ci implémentent ou pas les extensions de sécurité.

De surcroît, utiliser les nouvelles extensions spécifiques à la sécurité dans *OpenCL*, comme l'initialisation des mémoires, permet des gains de performance importants, grâce aux fonctionnalités des matériels inaccessibles en dehors des drivers. On peut aussi noter que nos travaux sont utilisables dans un contexte différent de celui du *WebCL*, pour toute API utilisant *OpenCL* et souhaitant produire un code *OpenCL/C sécurisé*.

3.3.4.4. Exemples d'attaques

Une caractéristique inhérente à toutes les API disponibles sur le navigateur est la prise en compte de la sécurité. En effet, il est crucial qu'aucune information ne puisse être dérobée à l'insu de l'utilisateur, que personne ne puisse prendre le contrôle de la machine ou que l'accès à une simple page ne provoque jamais un état instable du système. C'est l'une des principales différences entre *OpenCL* et le *WebCL*. Alors que le premier considère que les programmes exécutés sont de confiance, comme tous les programmes natifs, la deuxième API présume que toutes les instructions proviennent d'une source non fiable (site Internet) et par conséquent le *WebCL* doit ajouter les protections nécessaires pour assurer la sûreté de l'utilisateur.

Nous ne décrivons pas ici toutes les attaques possibles mais donnons seulement quelques exemples précis qui nous paraissent significatifs. Ils nous permettent d'exposer les éléments principaux qui ont guidé cette réflexion sur la sécurité, et devraient aider à la compréhension de la solution proposée.

La première sécurité qu'il convient de mettre en place est la résistance aux attaques DOS (déni de service). Un autre principe général de sécurité consiste à limiter la quantité d'informations disponible pour réduire au maximum la surface d'attaque. Or, un risque induit par notre outil est que nous exposons toutes les informations habituellement accessibles à *OpenCL*, ce qui autorise le développeur à connaître le nom et les

caractéristiques du matériel client, en contradiction avec le principe évoqué plus haut. Pourtant, cette possibilité est intéressante du point de vue de l'utilisateur/développeur du site car cela lui permet d'adapter ses traitements au matériel afin d'optimiser le rendu et les performances. Mais cela permet aussi à une personne mal intentionnée de connaître suffisamment d'informations sur l'ordinateur pour exploiter d'éventuels trous de sécurité connus et liés à un matériel précis. Nous pensons qu'avec le temps, les drivers *OpenCL* seront suffisamment stables pour que le risque soit limité, et que l'expérience permettra d'identifier les paramètres les plus importants afin d'offusquer seulement ceux qui n'apportent qu'un gain négligeable au regard du risque encouru. Dans les cas pratiques que nous avons étudiés, c'est-à-dire le calcul de niveaux de détail pour la visualisation 3D en temps réel et la simulation de particules, les seuls paramètres qui ne semblent pas importants pour déployer l'application sont des paramètres de bas niveau, comme la version des drivers utilisés pour le *device* ou l'alignement mémoire. En concertation avec le *WebCL Working Group*, il a été décidé de maintenir la possibilité d'interroger les extensions disponibles pour les langages *CL* et les *devices*, afin d'éviter la sous-exploitation de ces derniers (même s'il est vrai par ailleurs que le développement dédié à des architectures spécifiques n'est généralement pas souhaitable dans le cadre d'applications Web). Ce choix se fonde sur les décisions prises par *Khronos* de permettre des fonctionnalités d'interrogation dans le *WebGL*.

3.3.4.4.1. Déni de service

Nous commencerons par l'attaque par déni de service (DOS) qui peut facilement être réalisée en exécutant des calculs intensifs sur le *device*. Il est important de noter qu'il est quasiment impossible de différencier un calcul intensif réalisé dans le cadre d'une application légère, d'un autre dont le but est de consommer toutes les ressources disponibles de l'hôte. C'est d'autant plus vrai sur notre API haute performance dans le navigateur.

Pour notre réflexion, nous séparerons les *devices* selon que l'unité de calcul est préemptive (qui peut être interrompue) et capable de changer de contexte d'exécution (fonctionnement multitâche) ou pas. Concrètement, les *CPU*, qui possèdent ces deux capacités seront distingués des *GPU* qui n'en possèdent aucune.

Dans le cas du *CPU*, le *DOS* n'est pas vraiment impactant pour plusieurs raisons. La première est que la préemption des CPU permet d'arrêter les processus trop gourmands ; la deuxième est liée au fonctionnement multitâche et à la gestion des priorités des processus, qui autorisent à exécuter certains calculs intensifs en tâche de fond sans que l'utilisateur ne soit impacté. Du point de vue du navigateur, cela ne diffère pas des *WebWorkers* du *HTML5*.

Pour le *GPU* le problème est légèrement différent car il n'est pas possible de supprimer l'exécution d'une tâche avant que celle-ci ne soit terminée. En revanche il est possible d'utiliser le même type de protection que pour l'API *WebGL*, dérivée d'*OpenGL* sur

les *GPU* : nous considérons que même s'il est envisageable que certains traitements nécessitent plus de deux secondes sur le *GPU*, il n'est pas acceptable de bloquer l'affichage de l'ordinateur sur une période aussi longue. Ce blocage provient du fait qu'un *GPU* classique n'est pas capable de mettre en pause un traitement pour en effectuer un autre prioritaire. Afin de régler ce problème, une extension a été mise en place et est implémentée par la plupart des *GPU*. Cette extension permet de fixer un temps de traitement maximum avant que le *GPU* ne soit rebouté et le contexte *OpenGL* associé invalidé. Nous avons soumis des demandes d'extensions similaires pour le *WebCL.*

Nous souhaitons aussi attirer l'attention sur la nouvelle génération de *GPU* haut de gamme qui offre la possibilité de changer de contexte d'exécution, ce qui permet d'allonger le temps de traitement puisque l'affichage de l'ordinateur sera seulement ralenti, comme pour toute exécution de plusieurs applications ayant une consommation de ressources cumulée importante.

Enfin, il est possible de réaliser une attaque par déni de service en utilisant des codes vérolés (provoquant une *segmentation fault* par exemple), mais ce type de comportement est protégé par la sécurité dédiée au vol de données, que nous allons maintenant exposer.

3.3.4.4.2. Programme non conforme

Afin d'obtenir un niveau de sécurité satisfaisant, il est nécessaire que toutes les informations accessibles par le contexte ne puissent provenir que du contexte actuel ou d'une valeur prédéfinie.

La première attaque consiste à accéder à des zones mémoires qui ne devraient pas être utilisées par la tâche en cours d'exécution, afin d'y insérer du code malicieux ou plus simplement d'y voler des données. La protection typique est la création d'une *sandbox* pour l'exécution du programme, comme le fait par exemple Native Client [YEE ET AL. 2009]. Bien que cette protection soit efficace, elle n'est pas envisageable dans notre cas pour plusieurs raisons : la première est que la *sandbox* est très proche du hardware et utilise des fonctionnalités spécifiques à chaque unité de calcul (comme le bit d'exécution ou le *segment memory* du x86), alors que dans notre cas, nous souhaiterions une solution générique non dépendante du matériel. De plus, Native Client [YEE ET AL. 2009] permet l'exécution de code binaire non vérifié, alors que nous souhaitons l'exécution d'un code source non vérifié que nous compilons dans le navigateur. Nous pouvons faire confiance à notre compilateur car il fait partie intégrante de l'OS et par conséquent, si ce dernier était compromis, alors il serait beaucoup plus facile pour l'attaquant d'utiliser d'autres méthodes pour prendre le contrôle de l'ordinateur. Si nous devions réaliser une analogie avec le *JavaScript*, nous compilerions tout le code avec le JIT avant l'exécution de ce dernier ; et tout comme le *JavaScript*, il est possible, durant la compilation, d'ajouter les éléments de sécurité. Enfin, il faut noter que Native Client [YEE ET AL. 2009] permet d'écrire des programmes en C/C++ alors que le code utilisé dans les programmes *OpenCL* est déjà une version limitée du C (absence d'allocations

dynamiques, absence de récursion, etc.), ce qui est plus simple à analyser (à *parser*) que le C++ et le *JavaScript*. Ainsi, notre solution peut être vue comme une adaptation de SoftBound [NAGARAKATTE ET AL. 2009] reposant sur une représentation intermédiaire de plus haut niveau que LLVM IR.

Enfin, nous souhaitons mettre en place une sécurité qui soit la plus légère possible en mémoire et en temps d'exécution, et dont il soit facile de connaître à l'avance le surcoût maximal, tout en restant compatible avec la plupart des drivers *OpenCL 1.2* (cette version a été choisie car elle permet de régler des problème dans l'alignement mémoire des structures, à la différence de la version 1.1). En effet, une des vocations de *WebCL* est de porter la haute performance au sein des navigateurs pour les développeurs *OpenCL*, bien que *WebCL* devrait aussi bien pouvoir être utilisée par les développeurs *JavaScript*. Par conséquent, nous souhaitions conserver l'un des avantages du C qui est la proximité entre le code source et le code compilé ("*close to the metal*"). En autorisant le développeur à connaître le code généré, hors optimisation, c'est-à-dire le pire des cas, nous permettons aux utilisateurs de *WebCL* d'optimiser leur code tout comme du code C.

Les échanges d'information non autorisées peuvent être réalisés de plusieurs manières. Les plus conventionnelles sont les attaques de type *stack smashing*, c'est-à-dire l'utilisation erronée des pointeurs comme par exemple les "*out of bound accesses*", c'est-à-dire l'accès à des valeurs non initialisées ou encore le pointage sur des variables dont la durée de vie (ou *scope*) est plus petite que celle du pointeur (ainsi la variable est libérée mais le pointeur est encore présent et référence une zone mémoire non valide).

L'exemple le plus caractéristique de notre approche est, de notre point de vue, la protection contre le surpassement de la pile (*stack*). Il existe déjà des protections connues, mais qui ont un impact significatif sur l'exécution du programme. Cependant, dans le cas d'*OpenCL* et de *WebCL*, les tableaux dynamiques et les appels récursifs étant interdits, il devient possible d'analyser statiquement la consommation maximale utilisée. Bien que cette méthode ne soit pas sans défaut (possibilité de faux positifs), le cas est très rarement rencontré en pratique, rendant les faux positifs de dépassement de mémoire quasi inexistants, ce qui, de plus, n'ajoute aucun surcoût à l'exécution. D'autres contraintes consistent à connaître la taille de la pile pour le programme ou encore à définir un minimum commun pour tous les *devices* compatibles avec le *WebCL*. Pour que cette solution soit la plus efficace possible, il est nécessaire qu'elle soit effectuée directement dans le driver. En effet, la taille de la mémoire de la pile dépend aussi du nombre de *work-items* exécutés simultanément et cette information n'est détenue que par le driver lui-même.

En ce qui concerne les accès en dehors de la mémoire allouée, comme évoqué précédemment, nous adaptons les solutions issues de la littérature, comme SoftBound [NAGARAKATTE ET AL. 2009], pour les rendre applicables dans l'AST. Nous ajoutons aussi une contrainte supplémentaire sur l'initialisation des variables, point particulièrement important dans le cadre d'une utilisation Web. Nous avons aussi proposé une extension

OpenCL s'occupant de la mise à zéro par le matériel de la mémoire privée et locale. Ce choix est motivé par l'inaccessibilité de certaines fonctionnalités bas niveau et par le fait que seul le driver possède tous les éléments pour savoir quand la réinitialisation de la mémoire est nécessaire. En effet, en présence d'une telle extension, il deviendrait possible d'éviter l'initialisation de la mémoire lorsque les données qu'elle contient sont issues du même programme, c'est-à-dire du même contexte *OpenCL* (les contextes *OpenCL* sont utilisés pour gérer et regrouper les zones mémoires et les programmes). Par exemple, si deux tâches utilisent la même mémoire partagée ou locale, que ces deux tâches proviennent d'un seul contexte *WebCL* et qu'aucune autre exécution d'un contexte étranger n'a pu y accéder, alors il n'est pas nécessaire de la réinitialiser. Ainsi, seul le driver est en possession des informations qui permettraient d'économiser éventuellement cette réinitialisation avant chaque exécution.

Il existe aussi des méthodes moins classiques comme cette attaque, issue du *WebGL,* qui consistait à récupérer une image d'un autre site (potentiellement sécurisé) en exécutant un *shader* dont le temps d'exécution dépendait de la valeur des pixels. Même si le *WebGL* ne permettait pas directement l'affichage de l'image pour des raisons de sécurité, l'attaquant pouvait accéder à son contenu à l'occasion d'un traitement.

3.3.4.5. Notre solution : implémentations

Dans nos premier travaux et ceux repris par le groupe de travail WebCL, nous utilisons un code *OpenCL/C* et *Clang*. La méthode est implémentée par Vincit qui réalise avec notre aide un programme de sécurité pour le *WebCL* dans les navigateurs, financé par le consortium Khronos.

Notre implémentation repose sur *l'Abstract Syntax Tree* (AST) pour produire du code sécurisé à partir du code C original. Pour cela, nous utilisons un code qui génère le code source sécurisé à partir de l'arbre de *Clang* qui est lui-même retransformé en code C. cette solution a été choisie pour rester le plus proche possible de l'esprit de la bibliothèque fournie par *Clang*, bien qu'il soit également possible d'intégrer les vérifications nécessaires par un remplacement de nœuds de l'AST (simple réécriture de nœuds grâce à l'API *rewritter* fournie par *Clang*, puis recréation de l'AST afin de s'assurer de l'intégrité des modifications apportées). Nous avons cependant choisi de réécrire complétement la fonctionnalité de transformation de l'AST en code C (appelée *prettyPrint* dans *Clang*) afin de complexifier l'insertion de code malicieux dans le code utilisé par le driver. En effet, le code initial de réécriture n'avait pas été pensé pour la sécurité et il y avait donc des risques que le code produit par le transcompilateur ne soit pas totalement sécurisé (l'intégrité n'étant alors pas assurée). Au contraire, avec notre solution, l'oubli provoque une erreur dans la chaîne de compilation plutôt que de produire un code dangereux.

Pour mettre en place cette protection, nous avons choisi de réaliser un transcompilateur du langage *OpenCL/C* vers *OpenCL/C sécurisé* où les pointeurs sont

remplacés par des structures opaques (cet outil est réalisé grâce à *Clang* et les en-têtes *OpenCL* de *liblclc*). Chaque pointeur est attaché à une zone mémoire précise (avec une adresse de début et une longueur), et aucun pointeur ne peut se voir associer une valeur arbitraire (comme dans l'instruction *int* a = 5;*)

Nous limitons aussi les possibilités offertes par le langage au niveau des structures : éviter les effets de bord lors des *casts* ou simplement faire en sorte que les structures ne soient trop complexes. Nous avons limité les structures aux types basiques de l'*OpenCL*, c'est-à-dire aux structures ne pouvant contenir que 64 variables. Ces contraintes permettent de restreindre les zones mémoires "de déchet" en fonction de l'architecture, sans entamer significativement les possibilités.

Enfin, la taille de la pile (*stack*) peut aussi être problématique si elle devient trop grande (*stack smashing*). Nous effectuons donc une validation statique du code à partir de l'AST fourni par *Clang,* par un simple parcours en profondeur : la branche ayant la taille mémoire la plus grande permet de déterminer le pire cas de consommation de mémoire et de vérifier que ce dernier ne soit pas problématique. La taille de la pile est limitée pour nos programmes à une valeur choisie arbitrairement pour fonctionner sur les matériels mis à notre disposition. Une étude plus complète permettrait d'obtenir une valeur minimum commune à tous les *devices* utilisables et de fixer un minimum pour la compatibilité avec *WebCL*. Cette analyse statique est rendue possible par les restrictions d'*OpenCL* par rapport à du code C classique, telles que l'interdiction d'utiliser des tableaux dont la taille n'est pas définie par une constante dans les sources, ou encore l'interdiction d'écrire des appels récursifs.

OpenCL possède plusieurs zones mémoires : la zone globale qui est partagée par tous les *work-groups* (que l'on peut considérer comme des processus), la zone locale qui est partagée par tous les *work-items* (que l'on peut assimiler à des *threads* légers) et la zone privée, mémoire par défaut, qui est assignée uniquement à un *work-item*. Ce dernier point ne s'applique qu'aux mémoires "private" d'*OpenCL* étant donné que tous les autres types de mémoire ont une durée de vie supérieure à l'exécution de la tâche.

Dans notre implémentation, nous avons défini trois zones poubelles : une pour chaque zone mémoire. Lorsqu'un pointeur utilise une zone trop petite, même pour un unique élément, le pointeur est remplacé par la zone poubelle correspondante.

Le cas de « *dangling pointers »,* quant à lui, est corrigé grâce à la génération de code suivant le principe SSA (*Static Single Assignment*) par le transcompilateur.

Dans le cas de la version industrielle avec Vincit et financée par *Khronos*, le choix retenu est légèrement différent : une zone mémoire fixe est définie et chaque variable incrémente la position du pointeur dans cette zone (comme l'allocateur de mémoire décrit

par *K&R* [KERNIGHAN ET AL. 1988]). Avec ce principe, toute utilisation mémoire sera faite sur la zone définie au préalable.

Un autre point important est qu'*OpenCL* et *WebCL* utilisent les tableaux de façon intensive. Par conséquent, il est très important de réduire les vérifications par des optimisations statiques, plus que dans des programmes classiques. Or, le langage C n'a pas de variable associée à la taille des pointeurs, alors que nous en aurions besoin ici pour des raisons de sécurité. Ainsi, lorsque l'utilisateur veut itérer sur tous les éléments d'un tableau, nous fournissons une nouvelle méthode (*arraysizeof*) qui permet au développeur de le notifier au compilateur, qui pourra alors détecter et effectuer plus aisément les optimisations statiques.

Enfin, le transcompilateur doit lui aussi être sécurisé, pour éviter toute attaque dans le *parser* ou la génération de code. Par conséquent, le transcompilateur ne peut être utilisé que dans une *sandbox*.

C'est à partir de cette dernière idée que nous avons réalisé un deuxième prototype dans lequel nous utilisons un sous-ensemble *JavaScript* comme code intermédiaire (tout comme l'est LLVM IR). Ce sous-ensemble a été revu pour être proche d'ASM.JS lors de sa sortie (notre prototype ayant été démarré avant l'annonce d'ASM.JS). ASM.JS est un projet de Mozilla dont le but est de définir un sous-ensemble *JavaScript* très efficace à utiliser, similaire à Rpython[39]. Cependant, l'utilisation d'une zone mémoire fixe pour tous les traitements et l'absence de tableau dans les paramètres des fonctions le rendait quasi inutilisable dans notre cas. C'est pour cette raison que nous l'avons étendu, afin de permettre d'ajouter des méta-informations comme la position mémoire (privée ou globale) ou encore l'utilisation des tableaux dans les paramètres, la déclaration de pointeurs mémoire stockés comme référence. Nous avons également restreint certaines fonctionnalités absentes dans le *GPU* (comme la récursivité). Il en résulte un code bas niveau beaucoup plus simple à sécuriser que le C (pas de structures, de pointeurs, etc.) mais qui reste compatible avec de nombreuses plateformes puisque le code sera retranscrit en *OpenCL/C*.

3.3.4.6. Résultats

Exemple de code:

```
// seulement l'API d'asm.js API, postMessage et onMessage sont
fournis par le worker tout comme la mémoire globale
"use pasm";
"use GPU";


//var testnull;
```

[39] http://dl.acm.org/citation.cfm?id=1297091

```
var priv = new Int8Array(16 );// la mémoire privée est seulement
accessible par le thread


// nous connaissons l'accessibilité et une copie n'est pas
nécessaire
var sharedMem = new Int8Array(16| 0);// la mémoire locale est
paratagée entre les threads
//dans le device
// nous pouvons accéder à la mémoire partagée grâce à l'API fournie
dans le worker  (mémoire transactionnelle)
// et persistente

var double = 0.0;
var simple = 0;

var num = 5e7;

var multi = 2, dd = 5;

function kernel_testArray(myArray) {
    myArray = new Int32Array(myArray);
    return myArray.buffer;
}

function priv_test(arrayOrScalar  ) {
    // Au début pour aider, besoin de déterminer si scalaire
    arrayOrScalar = arrayOrScalar | 0;


    switch (arrayOrScalar) {
        case 2 :
            break;
        default:

    }

    if (true) {
        arrayOrScalar++;
        arrayOrScalar = 7;
    }
    else
        return +7;

    if(5){
        return +5;
    }
```

```
    do{
        arrayOrScalar++;
    }while(arrayOrScalar<5);

    while(arrayOrScalar>7) {
        arrayOrScalar--;
    }

    var i = +0;

    for (i = 0; i < 5; ++i, i += 7) {
        arrayOrScalar += priv[8 >>> 5];
    }

    return +0;

}

function kernel_test(array, scalar) {
    array = new Int16Array(array);// aide pour le type
    // array peut être typedArray mais pas ArrayBuffer
    scalar = +scalar;
    var otherStuff = 0 | 0;
    var id = getItemId() | 0;
    var thirdStuff=
            array[id] += id;

    testArray(array.buffer);//l'appel peut seulement utiliser
ArrayBuffer

    return array.buffer;
}
```

Les résultats expérimentaux sont obtenus sur les drivers OpenCL natifs de MAC OSX 10.8 tournant avec un processeur Intel Core i7 (2,4Ghz), et un AMD Radeon HD 6770M. Ils nous montrent la moyenne d'exécution pour 150 séries de tests sur des codes instrumentés ou non, sur des matrices de tailles A(400 x 800), B(400 x 400), C(400 x 800). La déviation standard était inférieure à 1.

GPU Standard :
40,97 Gflop/s
Temps moyen d'exécution : 0,04960s
Temps total d'exécution : 0,05s

GPU Instrumenté :
23,10 Gflop/s,

Temps moyen d'exécution : 0,08818s
Temps total d'exécution : 0,08865s

GPU instrumenté: Avec mémoire initialisée
22,90 Gflop/s,
Temps moyen d'exécution : 0,08850s soit 1,78x plus lent
Temps total d'exécution : 0,08885s

CPU Standard:
1,71 Gflop/s,
Temps total d'exécution : 4,5s

CPU Instrumenté :
1.486 Gflop/s,
Temps total d'exécution : 5,1s, soit 1,133x plus lent

CPU instrumenté: Avec mémoire initialisée
1,2190 Gflop/s,
Temps total d'exécution : 6,3s, soit 1,4x plus lent

Comme nous le voyons, le surcoût de la sécurité dans ce cas précis se situe entre 14 et 40%, ce qui est plutôt encourageant puisque notre transcompilateur ne possède pas d'optimisation spécifique.

3.3.5. Intégration dans Firefox

Nous donnons dans cette partie les informations nécessaires à la réalisation du plugin que nous avons programmé. Pour chaque onglet du navigateur, un objet de type WebCLWorkerFactory est intégré au *window* du DOM associé à la page Internet. Cet objet contient une méthode qui permet d'instancier un *WebCLWorker* à partir de l'url du code que le *worker* devra exécuter.

Pour comprendre comment les interactions avec le *JavaScript* sont gérées, nous allons esquisser brièvement le fonctionnement de Firefox sans avoir la prétention d'en donner une description exhaustive. Tout d'abord, Firefox repose sur des composants *XPCOM*. Ces composants sont des classes qui possèdent des interfaces générées à partir de fichiers textes nommés xpidl. Ces fichiers textes permettent de définir les interactions entre les différents *XPCOM*, qui peuvent être implémentés dans des langages différents (C++, *JavaScript*, Python).

```
Xpidl:
[scriptable, uuid(cd247fc5-892e-4ee9-9bf2-15a22ec9a08a)]
interface nsIEcho~: nsISupports {
```

```
string echo(in string echo);
}
```

Ainsi, le composant qui implémente cette interface pourra être appelé indifféremment par le *JavaScript* :

```
var myEcho = nsEcho.echo("hello world");
alert(myEcho); // Affiche hello world
```

ou par le C++ :

```
string helloWorld;
nsresult rv = nsEcho.Echo("hello world", &helloWorld);
// Les methodes natives Xpidl sont en lettres capitales
// Vérifie si la methode a été effectuée correctement
if(rv == NS_SUCESS)
cout << helloWorld; // Affiche helloWorld
```

A noter que pour le C++, la variable qui doit contenir le résultat est passée en pointeur et le retour de la fonction correspond au statut de ce résultat. En revanche, en *JavaScript*, une erreur d'exécution envoie simplement une exception. Cette différence de traitement est un héritage du navigateur Netscape (ancêtre de Firefox) qui devait tourner sur des plates-formes ne gérant pas les exceptions C++.

L'interaction entre les codes natifs et les codes interprétés comme le *JavaScript* est traitée par une classe du nom de *xpconnect*. Cette classe gère la sécurité et fait la correspondance entre les objets *JavaScript* et leur équivalent en *XPCOM*, lorsque ces derniers existent. Ainsi, tout objet du DOM accessible en *JavaScript* est implémenté en C++ et possède une interface *XPCOM* définissant son comportement.

Pour qu'un composant puisse être utilisé avec le *JavaScript* et dans une page Internet directement, il doit posséder un lien vers une classe *ClassInfo* qui renseignera *xpconnect* sur les méthodes accessibles. Ainsi, une méthode *long add(long a,long b)* dans le *IDL (Interface Description Language)* sera transformée en C++ en méthode *NSMethodImpl Add(long a,long b, long *result)*, et en supposant que l'objet *JavaScript exemple* soit une instance de l'interface définie par l'idl et *exempleNatif* le composant *XPCOM* correspondant, le code *JavaScript* exemple.add(5,6) sera transcrit par *xpconnect* en exempleNatif.Add(5,6). A noter que dans un composant *XPCOM*, le résultat de l'opération est toujours livré sous forme de pointeur, afin que le retour de la méthode NSMethodImpl renseigne sur l'état d'exécution de la fonction et les erreurs éventuellement rencontrées.

Alors que les interactions avec canvas2D peuvent être implémentées directement et simplement en écrivant un idl dont la signature utilise l'interface *xpcom* du canvas, les typedArray, eux, ne sont pas traduits par *xpconnect* car ils n'ont pas d'équivalent *xpcom* ou

dans le langage IDL. Il est alors nécessaire d'utiliser les fonctions bas niveau du JavaScript pour récupérer la donnée capturée.

Comme expliqué plus haut, il existe un problème de performances pour le passage de données entre le *JavaScript* et le code natif. Pour résoudre les lenteurs des vérifications faites par *xpconnect* pour trouver la bonne méthode dans l'interface décrite dans l'idl, il existe une solution du nom de *quickstub* qui n'est accessible que pour les API intégrées à Firefox. Cette solution donne accès aux fonctions bas niveaux du *JavaScript* dans le code natif et doit ainsi permettre des comportements plus avancés que ne le permet les idl, comme par exemple la surcharge de fonctions, la modification de la valeur d'un paramètre lors d'un appel ou encore la récupération de l'objet appelant. Grâce au *quickstub*, nous pouvons réduire la latence de l'appel aux fonctions *WebCL* et *WebCLWorker* en dessous de 1 ms tout en permettant de transformer des tableaux C++ en *JavaScript* et vice versa.

Enfin, le *JavaScript* est un langage à ramasse-miettes. Par conséquent, pour ne pas provoquer de fuites mémoire ou de problèmes de stabilité du fait des traitements asynchrones entre le C++ et le *JavaScript*, il a été nécessaire d'attacher les événements et certains objets mémoire à une racine des objets *JavaScript*. De cette manière, on évite que les objets ne soient détruits (du fait de leur absence côté *JavaScript*) avant que le traitement côté C++ ne soit complètement terminé.

Nombre d'objets	Tests de collision en *JavaScript*	Tests de collision en *WebCLWorkers*
5	22 à 25 fps	20 à 23 fps
80	12 fps	20 à 22 fps
100	8 fps	20 à 22 fps

Figure 35 - Grâce à l'utilisation de CPU multi-coeurs et les optimisations du code des WebCLWorkers, il devient possible d'effectuer des tests de collision sur plus de 100 objets en temps réel avec un algorithme naïf non optimisé. Au-dessus de 100 objets nous avons observé une instabilité du driver OpenCL utilisé (CLEP).A l'heure où nous écrivons ces lignes, Mozilla vient de publier (Août 2013) une version optimisée d'un code de collision qui ne fonctionne que sous Firefox et qui peut gérer plusieurs centaines d'objets par seconde. Cette solution utilise asm.js mais ne règle cependant toujours pas le problème de transfert entre la carte graphique et le GPU, ni même le parallélisme de données.

3.4. Résultats et limitations

3.4.1. Résultats expérimentaux

Nous avons choisi de tester les *WebCLWorkers* avec les classiques algorithmes de particules et de collisions, car ces derniers sont populaires et présentent le même type d'exigences que ceux que l'on rencontre dans notre méthode de visualisation de terrain en 3D : parcours de tableaux et interactions entre les éléments en fonction de leur distance. Les différents résultats figurant dans cette section montrent qu'avec les *WebCLWorkers*, il devient possible de réaliser au sein d'un navigateur du traitement parallèle de données, efficacement et sur le *device* le plus approprié, CPU ou GPU. En plus de rendre possible la programmation GP/GPU, notre API permet en effet de tirer profit des CPU multi-cœurs actuels, et elle est également compatible avec un affichage en temps réel (2D ou 3D) des données traitées, comme le montre une application graphique gérant les collisions entre plus de 10000 particules sans que le CPU ne soit sollicité autrement que pour la boucle d'événements : 50 fps pour un CPU à moins de 1% de charge sur un PC standard équipé d'un processeur Intel Core2 9400 associé à une carte graphique NVidia FX Quadro 2700M (cf figure 38).

Jusqu'à présent, réaliser des tests de collision en *JavaScript* pouvait s'avérer très coûteux et difficilement exploitable en pratique (http://www.glge.org/demos/trimeshdemo/). Avec notre solution, les détections peuvent se faire en temps réel sur un nombre important d'objets, sans même utiliser la puissance du GPU, comme le montrent les résultats expérimentaux du tableau 35. Le ralentissement des fps, même lorsque le nombre d'opérations de collision est réduit, s'explique par le fait que notre solution est actuellement implémentée sous forme d'extension Firefox, ce qui provoque des latences dans le transfert des données entre les *WebCLWorkers* et le code *JavaScript* (environ 3ms par appel de fonction). Ce problème disparaît lorsque l'échange des données est effectué par une méthode spécifique à Firefox et qui n'est accessible que par les API directement intégrées dans ce navigateur (« quickstub »). Autrement dit, le problème sera résolu par l'intégration des *WebCLWorkers* dans Firefox.

Comme nous l'avons déjà souligné, les *WebCLWorkers* permettent de traiter les données 3D sur le GPU tout en gérant leur affichage dans les pages HTML, rendant ainsi réalisables les jeux vidéo les plus gourmands en ressources. Pour prouver que notre méthode permet la programmation GP/GPU sans impacter l'affichage, nous avons réalisé un test de collision entre des particules et des sphères (les collisions entre particules n'étant pas gérées). Nous avons obtenu 100 frames par seconde pour 10 sphères et 64000 particules (cf. figure 38).

Nb de particules	Temps de calcul des interactions *JavaScript* (ms)	Temps inter. WebCLWorkers	Nb op. JS (GFLOPS)	Nb op. GPU WebCLWorkers
1024	80	2	0,5	14
2048	340	4	0,5	18
4096	1400 (+/-10%)	9	0,5	23
8192	5400 (+/-10%)	34	non mes.	47

Figure 36 - En exécutant une version parallèle de l'oscillateur sur GPU avec l'API WebCLWorkers, les temps d'exécution sont divisés par 50 par rapport au code JavaScript (pour 4096 particules). Il devient ainsi possible de visualiser plus de 8000 particules avec des performances raisonnables.

Nous fournissons un troisième exemple d'application avec la simulation de N particules qui s'attirent et se repoussent alternativement en fonction de leur distance (oscillateur de particules, cf. figure 38).

Le tableau 36 montre que l'utilisation des *WebCLWorkers*, exploités par une version parallèle de l'oscillateur de particules, induisent un gain d'un facteur 50 en calcul brut sur les performances *JavaScript*. Indépendamment de ce gain, même si le *JavaScript* avait des performances équivalentes aux langages natifs, il existerait encore un goulot d'étranglement lié au transfert des données traitées vers la carte graphique pour l'affichage. Or, ce problème disparaît lorsque les traitements interviennent directement sur la carte graphique. Si l'on prend l'exemple d'une scène 3D pour laquelle on calculerait à la volée (par simplification ou raffinement) le niveau de détail à visualiser, le calcul de la scène devient inutile dès lors que le temps nécessaire au transfert des données vers la carte graphique est lui-même déjà trop long pour obtenir un affichage en temps réel (sans même se préoccuper du temps de rendu).

3.4.2. Limitations

Même si la solution que nous avons proposée fonctionne sous Windows, Linux et MacOSX, elle nécessite de configurer Firefox pour qu'il utilise OpenGL comme moteur de rendu *WebGL*. Sans cela, il n'est pas encore possible de faire interagir automatiquement les *WebCLWorkers* avec le rendu 3D ou les *buffers* et textures provenant de *WebGL*. Cette contrainte, présente uniquement sur les environnements Windows, provient de l'utilisation de la bibliothèque ANGLE par Firefox lorsqu'OpenGL n'est pas disponible. Cette bibliothèque

a pour fonction de transmettre les appels OpenGL à DirectX, mais elle ne permet pas au développeur de connaître la version de DirectX disponible. Or, *OpenCL* ne peut interagir qu'avec DirectX 10 ou supérieur.

De plus, les *WebCLWorkers* réduisent la possibilité de partager de la mémoire entre plusieurs *devices*. En effet, en supprimant la notion de contexte partagé, on supprime aussi la mémoire commune et l'on oblige donc le développeur à copier explicitement les données. Cependant nous pensons que cette dernière limitation est peu contraignante pour au moins deux raisons :

– Dans le cas où le contexte est partagé entre un CPU et un GPU, le driver OpenGL effectuera la copie de la même manière que le ferait l'utilisateur, ainsi la différence de performance reste minime ;

– Pour les autres cas, c'est-à-dire lorsque l'ordinateur possède plusieurs CPU et GPU (attention on parle ici de plusieurs puces distinctes et non de cœurs d'exécution), le contexte pourrait en effet partager la mémoire entre les *devices*. Cependant, comme la queue d'exécution est attachée à un seul *device* (tout comme l'exécution), l'utilisateur devra quand même s'occuper de la synchronisation des données sur les *devices*. Ainsi, la seule différence se situera au niveau de la performance lors de la synchronisation. Nous considérons que ce cas reste très rare, et sera probablement très peu utilisé étant donné que la cible de l'outil est l'utilisation au sein d'un navigateur. De plus, la prise en compte de ce cas impliquerait de complexifier l'utilisation des *WebCLWorkers*, ce qui est contraire à notre objectif initial d'accessibilité.

3.5. Conclusion et perspectives

Grâce aux *WebCLWorkers*, nous avons réussi à implémenter des simulations 3D en temps réel qui seraient irréalisables avec le seul langage *JavaScript*. Cette évolution permet d'envisager des traitements lourds réalisés par les clients légers et donc la migration d'une nouvelle gamme d'applications vers le *cloud.*

Afin de faciliter encore davantage l'utilisation des *WebCLWorkers* par les développeurs web, nous travaillons actuellement sur la possibilité optionnelle d'écrire les scripts exécutés directement en *JavaScript*. En effet, en se limitant à une sous-partie du *JavaScript* et en imposant certaines restrictions (comme par exemple, des règles précises pour la gestion des pointeurs et de la mémoire), il est possible de compiler ce langage de la même manière que le langage C. Ainsi, même si les performances risquent d'être légèrement inférieures à ce que l'on obtient actuellement, cela faciliterait encore la prise en main de notre outil et réduirait son coût d'apprentissage.

Nous poursuivons désormais nos travaux en collaboration étroite avec le groupe Khronos dans le cadre des spécifications du futur standard WebCL.

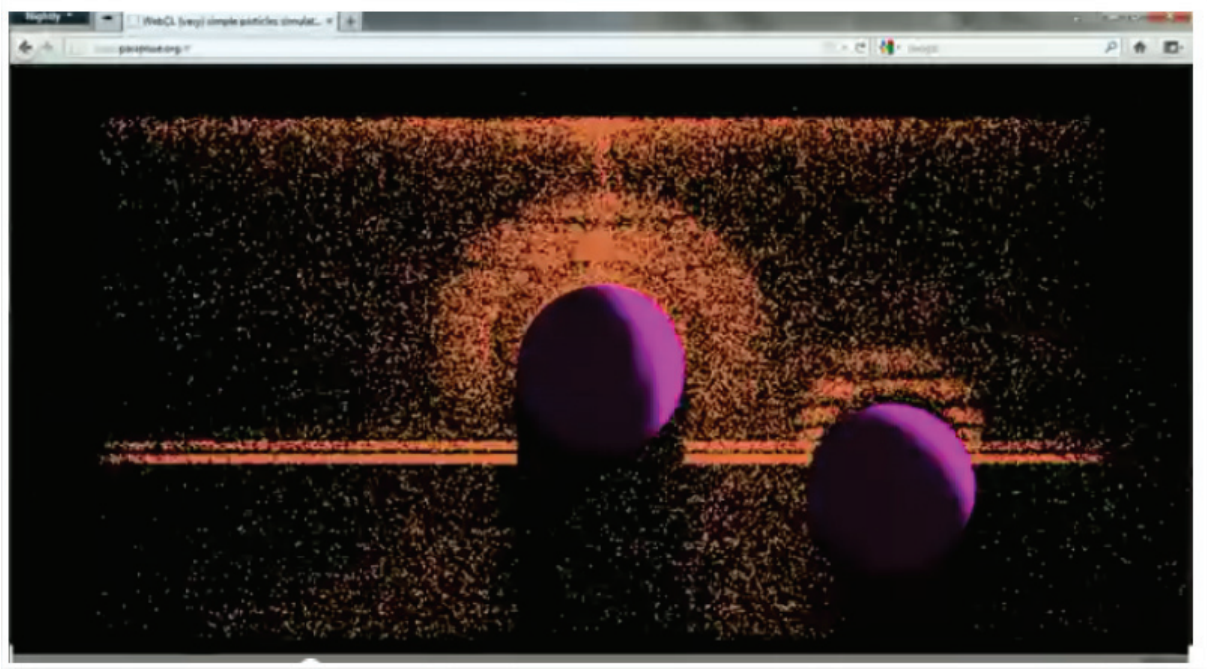


Figure 37 - Tests de collision entre 64000 particules et des sphères

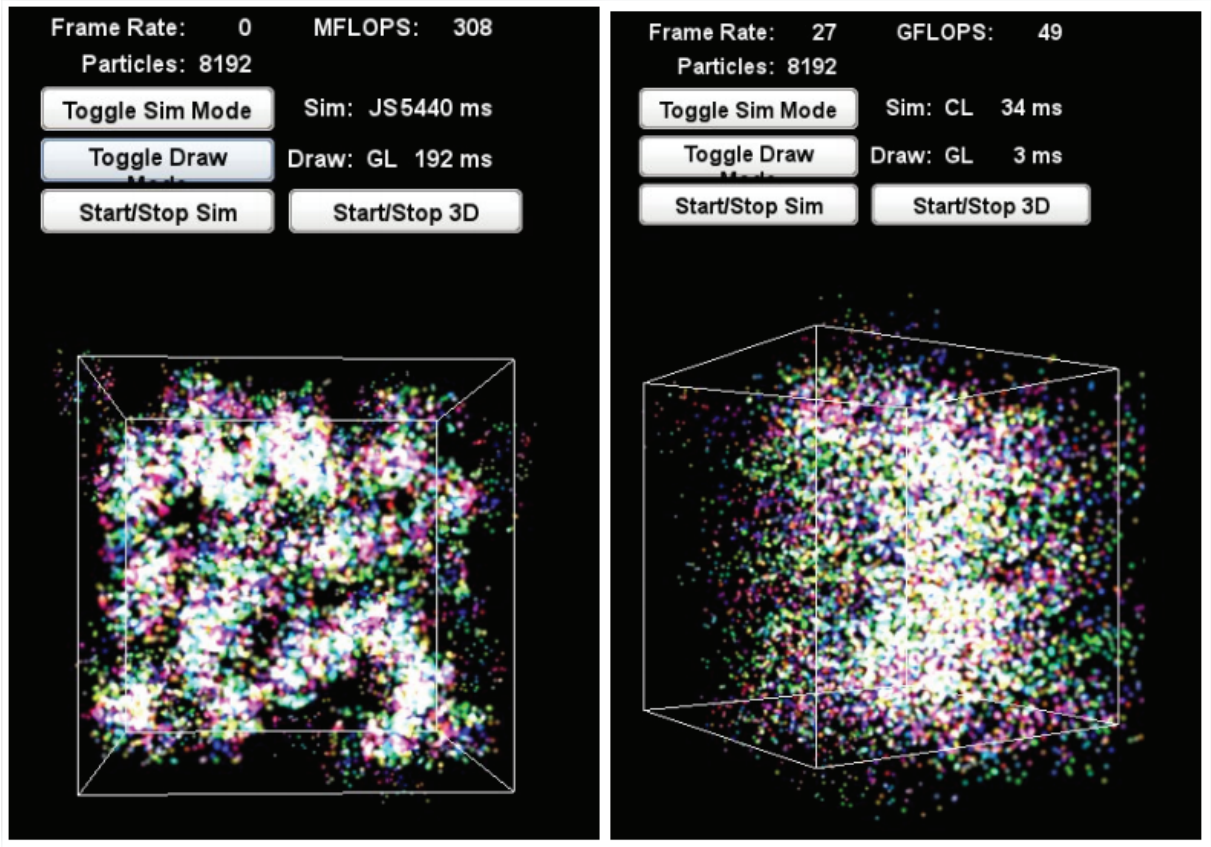


Figure 38 - Oscillateur de particules : à gauche, résultats du code JavaScript, à droite, en utilisant les WebCLWorkers.

4. Chapitre 3 : Simplification parallèle de données 3D

Contenu

4.1. Introduction et état de l'art

La visualisation de données géométriques est un domaine de l'informatique graphique particulièrement actif, du fait de ses nombreuses applications industrielles. Parmi celles-ci, la visualisation de modèles numériques de terrain (MNT) occupe une place significative, comme en témoigne l'abondante littérature consacrée au sujet depuis les années 1980, ainsi que la popularité d'applications telles que *Google Earth*. Les récentes évolutions technologiques, puissance considérable des cartes graphiques d'une part, langages et *frameworks* de développement web d'autre part, offrent de toutes nouvelles perspectives pour la visualisation de terrains au sein des navigateurs Web.

En effet, le passage au *Cloud Computing* et l'adoption de la technologie *HTML5* permettent désormais d'envisager les navigateurs Internet comme de véritables systèmes d'exploitation. Cependant, les performances des moteurs *JavaScript* des navigateurs sont toujours inférieures aux langages tels que Java ou C++. C'est pourquoi l'arrivée dans les navigateurs des *frameworks WebGL* et WebCL – à la conception duquel nous avons activement participé, cf chapitre 2 –, portages respectifs d'OpenGL et du *framework* GP/GPU *OpenCL* avec des performances comparables à leurs modèles d'origine, constitue un complément technologique fondamental. En effet, la combinaison *HTML5/WebGL/WebCL* rend possible la création d'applications Web offrant pratiquement les mêmes fonctionnalités que les programmes *desktop* classiques.

Nous rappelons que l'objectif de cette thèse est de développer une nouvelle méthode et des outils contribuant à la création d'un système d'information géographique 3D exécutable dans un navigateur Web. Plus concrètement, il s'agit de proposer à terme à l'IGN (Institut national de l'information géographique et forestière) une solution s'intégrant parfaitement au Géoportail (http://www.geoportail.gouv.fr), leur site actuel de visualisation de données géographiques. Leur solution s'apparente à *Google Earth* mais propose une visualisation de données plus diverses et détaillées.

C'est dans ce contexte que nous proposons ici une nouvelle méthode de simplification parallèle et de compression pour la visualisation de données 3D « streamées » issues de systèmes d'information géographiques. Les bases de cette méthode reposent sur les dernières avancées, à la fois matérielles (nos algorithmes sont parallèles de façon à exploiter l'architecture des derniers CPU et GPU) et logicielles (ils utilisent les API *HTML5* intégrées dans les navigateurs Web modernes). Parmi les avantages de notre méthode, nous pouvons citer entre autres la gestion des maillages irréguliers, la gestion de bâtiments ou d'autres modèles 3D intégrés au terrain, une prise en compte des problèmes de charge serveur avec la possibilité d'utiliser des fichiers statiques pour le *streaming*, et surtout, un nombre d'opérations parallèles proportionnel à la taille du maillage.

S'il existe de très nombreux travaux sur la visualisation de terrains, la plupart d'entre eux ne considèrent qu'un seul aspect de nos besoins, à savoir, soit la qualité de la visualisation, soit la compression des données, soit le transfert efficace par le réseau. Nous évoquerons ici les travaux que nous considérons comme les plus significatifs par rapport à nos besoins, et nous invitons le lecteur à se reporter au *survey* de Pajarola [Pajarola 2007] afin d'avoir une vision plus complète du domaine (même s'il ne fait que survoler les aspects *streaming* et compression).

Les *Geometry Clipmaps* [LOSASSO 2004] reposent sur l'organisation hiérarchique et la manipulation d'une grille fixe dont le centre dépend du point de vue. Il serait éventuellement possible d'utiliser cette méthode pour faire du *streaming* en ajoutant des ondelettes pour la compression (une méthode de compression d'image traditionnelle) et en exploitant les *geometry images* [GU 2002], qui permettent le stockage d'éléments 3D dans une image. Malgré cela, la méthode ne répond pas à nos besoins initiaux pour les raisons suivantes : tout d'abord, la grille étant semi-régulière, il y a un nombre constant de triangles à l'écran, et ces triangles sont répartis de manière uniforme, et non en fonction de la distribution de l'information. De plus, même si cette méthode est très rapide et efficace, la résolution des données affichées ne dépend que de la distance de la caméra à l'objet, ce qui empêche de fixer à l'écran une erreur visible inférieure à un seuil *s* donné, lorsque le matériel de rendu ne permet pas d'afficher plusieurs triangles dans la grille formée par s. Par exemple si l'on souhaite une erreur inférieure à la taille du pixel, il serait nécessaire, avec cette méthode, d'afficher plusieurs triangles par pixel.

Le *BDAM* [CIGNONI ET AL. 2003] présente une autre méthode fondée sur un découpage de l'espace en arbre binaire, avec des contraintes sur cet arbre permettant d'éviter la création de cassures. Ce découpage génère des *patchs*, c'est-à-dire des zones contenant des triangles, et la méthode simplifie des paires de *patchs* en effectuant des fusions d'arêtes qui reposent sur la métrique *QEM* [GARLAND 1997]. Malheureusement, aucune solution n'est proposée pour le *streaming* des données à travers le réseau : chaque raffinement de donnée impose une reconstruction complète du modèle, ce qui est adapté aux performances d'un disque dur sur une machine locale, mais inapproprié pour une diffusion à travers le réseau.

L'idée de l'arbre binaire avec contraintes a été ensuite reprise dans le *C-BDAM* [GOBBETTI ET AL. 2006], dans lequel une compression par une ondelette de *Neuville* est substituée à la simplification par fusion d'arêtes. Les points intéressants du principe d'ondelette sont l'efficacité de sa compression et sa possibilité intrinsèque de visualisation multi-résolution. Afin de lever la limitation du *C-BDAM* [GOBBETTI ET AL. 2006] à la 2,5D (et permettre ainsi, par exemple, l'intégration de bâtiments dans le modèle), il serait envisageable de remplacer l'ondelette de *Neuville* par celle de *Lounsbery* (ondelette 2D surfacique). Néanmoins, bien que les ondelettes et les grilles semi-régulières soient bien adaptées lorsque la représentation est composée principalement de basses fréquences, elles s'avèrent beaucoup moins efficaces en présence de très hautes fréquences.

Tous ces travaux peuvent être aujourd'hui implémentés sous forme d'applications Web, comme nous l'avons montré avec la réalisation d'*OpenScalesGL* [CELLIER 2012] (cf chapitre 1), qui est un prototype de visualiseur de terrains tournant aussi bien dans un navigateur, sur desktop comme sur smartphone, et fondé sur une méthode dérivée des *Geometry Clipmaps* de *Losasso et Hoppe* [LOSASSO ET AL. 2004] . Bien sûr, cela nécessite tout de même quelques adaptations pour tenir compte des contraintes inhérentes au *WebGL*, aux spécificités du *JavaScript* et des navigateurs et à l'hétérogénéité des clients. Le problème principal de ces solutions est qu'elles ne gèrent pas les grilles irrégulières, et ne permettent donc pas une modélisation efficace de données contenant de fortes variations de densité, comme par exemple un terrain avec routes ou bâtiments.

Dans ce chapitre, nous proposons de coupler le découpage en arbre binaire des méthodes *BDAM* [CIGNONI ET AL. 2003] et *C-BDAM* [GOBBETTI ET AL. 2006] à une méthode originale de simplification parallèle qui peut être effectuée sur *GPU* et n'impose pas l'utilisation d'une grille régulière. La simplification des triangles que nous proposons repose (comme celle du *BDAM* [CIGNONI ET AL. 2003]), sur la métrique d'erreur quadratique, ou *QEM* [GARLAND 1997]. La distance entre le modèle simplifié et l'original est estimée par la somme des distances quadratiques d'un point P aux plans qui lui sont attachés, à savoir les plans formés par les triangles du modèle original contenant P. Cette métrique a la particularité d'être relativement légère en mémoire et rapide à calculer car la distance quadratique à un ensemble de plans peut être décrite par une simple matrice triangulaire Q.

Lindström et al. ont montré [LINDSTROM ET AL. 1998] qu'il était possible d'améliorer ces résultats grâce à deux modifications : lors de la fusion de deux points, il suffit de prendre en compte les plans du maillage actuel (au lieu des plans du maillage original). De plus, en ajoutant au processus de simplification des contraintes garantissant un volume constant, le rendu obtenu est plus fidèle à l'original (d'après la métrique de *Hausdorff*). En revanche, cette simplification est beaucoup plus coûteuse en termes de complexité et reste, tout comme le *QEM* [GARLAND 1997], séquentielle.

La problématique d'une modification de maillage réalisée par le *GPU* a été abordée par *Hu et al.* [HU ET AL. 2009]. En revanche, ces travaux ne portent non pas sur le calcul à la volée d'un nouveau maillage mais sur la visualisation de maillages intermédiaires dépendants du point de vue : le contenu de ces maillages intermédiaires est pré-calculé lors d'une phase de *preprocessing*, et l'algorithme de rendu les charge ensuite dans la mémoire du *GPU*.

D'autres méthodes, plus récentes, utilisent le *raycasting* [DICK ET AL. 2009], afin de visualiser des terrains [DICK 2009] ou des bâtiments [CIGNONI 2007]. Cependant elles requièrent encore beaucoup de puissance côté client et ne sont adaptées qu'aux surfaces 2,5D et aux objets parallélépipédiques tels que les immeubles.

Dans la continuité de ses travaux initiaux, *Garland* a proposé d'introduire du parallélisme avec le *QEM* [GARLAND 2002], grâce à une simplification mixte qui consiste à fusionner en parallèle les points qui se trouvent dans une même cellule d'un octree, avant d'opérer une simplification séquentielle avec la méthode classique de son premier article pour affiner les dernières simplifications réalisées en parallèle. Cependant, bien qu'elle soit parallèle et compatible avec le GPU, cette méthode présente une incompatibilité avec le *streaming* de données tel que nous le souhaitons. En effet, elle construit et encode le passage d'un maillage fin à un maillage simplifié sans en détailler les étapes intermédiaires, ce qui va à l'encontre même de la notion de *streaming*. En effet, celui-ci repose sur le codage et la transmission des différences entre deux versions successives d'un même maillage. Une solution que nous avons envisagée est de transmettre au client uniquement le maillage sous forme de grille régulière (octree), puis de laisser le client effectuer les dernières fusions avec l'algorithme classique séquentiel de simplification. Cependant, cela interdirait la prise en charge des triangulations irrégulières, ce qui n'est pas satisfaisant dans notre contexte applicatif.

Dans ce chapitre, nous proposons de coupler le découpage du modèle de terrain original en arbre binaire des méthodes *BDAM* [CIGNONI ET AL. 2003] et *C-BDAM* [GOBBETTI ET AL. 2006] à une méthode originale de simplification parallèle qui peut être effectuée sur *GPU* sans qu'une grille régulière ne soit obligatoire. Notre solution améliore ainsi la vitesse de simplification du *BDAM* [CIGNONI ET AL. 2003], ajoute une fonctionnalité de compression compatible avec la visualisation *out-of-core* et le *streaming* réseau, et offre la possibilité de gérer de véritables objets en 3 dimensions (et non plus seulement en 2,5D).

4.2. Simplification parallèle

Tout comme le *BDAM* [CIGNONI ET AL. 2003] (ainsi que de nombreuses autres méthodes de visualisation 3D), nous avons choisi d'effectuer une simplification du terrain par fusions successives de ses arêtes (cf. figure 39). Nous avons opté pour l'exploitation de la métrique d'erreur quadratique, ou *QEM* [GARLAND 1997] en déclinant, à partir de l'algorithme original, une version parallèle permettant d'exploiter au mieux les nouvelles architectures *CPU* et *GPU* dont la tendance, depuis plusieurs années, est à la multiplication des cœurs d'exécution.

La métrique *QEM* [GARLAND 1997] est représentée par la distance quadratique d'un point P à tous les plans adjacents à P dans le maillage original M. Dans la définition de *Garland*, un plan R de M est adjacent à un point P d'une version simplifiée de M si R contient un triangle T dont l'un des sommets appartient à l'ensemble des points ayant formé P par fusions successives.

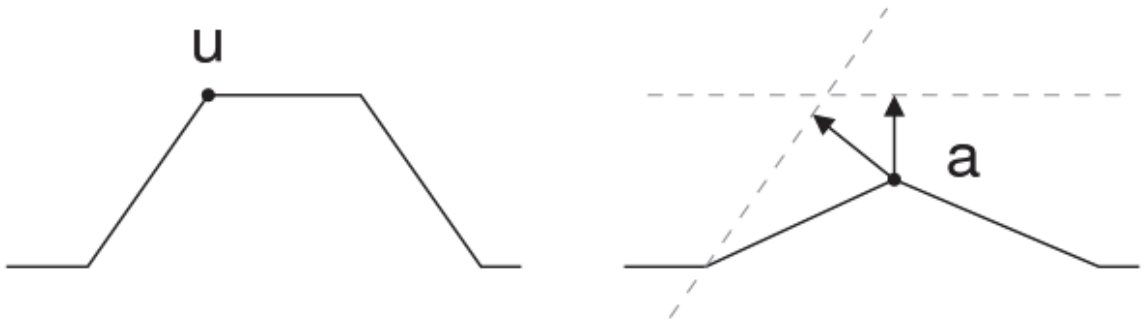


Figure 39 - Si u se retrouve en a, l'erreur associée à u est la somme du carré des distances aux plans adjacents à u, c'est-à-dire les plans en pointillés

Le premier avantage de cette métrique est que toutes les informations nécessaires au traitement d'un point P, c'est-à-dire la distance à l'ensemble des plans formés par les triangles adjacents à P (au sens de *Garland*), peuvent être stockées sous la forme d'une matrice triangulaire Q_P de taille 4 x 4, soit 10 coefficients par point seulement. Ceci est vrai à n'importe quel niveau de la hiérarchie grâce à la propriété remarquable suivante : le résultat de la fusion des points P et P' de matrices respectives Q_P et $Q_{P'}$ est un point P'' tel que $Q_{P''} = Q_P + Q_{P'}$. De par sa nature, la *QEM* [GARLAND 1997] ne dépend pas de l'ordre de fusion des points. Cela facilite grandement le parallélisme, dans la mesure où l'impact d'une fusion d'arête sous-optimale peut être borné. En effet, grâce à la métrique utilisée, le point résultant d'une fusion qui engendre des erreurs importantes sera naturellement pénalisé pour de futures fusions par rapport à des points qui proviennent de fusions ajoutant peu d'erreur au maillage. Cette propriété est fondamentale car le parallélisme que nous allons introduire nécessite d'effectuer la simplification par des fusions dont l'erreur n'est pas

toujours minimale sur l'ensemble du maillage mais seulement localement, comme nous le verrons un peu plus loin.

Il est à noter que *Lindström* [LINDSTROM ET AL. 1998], comme évoqué dans la section précédente, a proposé une méthode permettant d'obtenir des maillages simplifiés de meilleure qualité, mais qui n'est pas adaptée au parallélisme. En effet, cette solution repose sur une conservation du volume entre les différentes versions du maillage traité, ce qui implique en particulier d'intégrer dans l'erreur la différence de volume entre un modèle et sa version simplifiée. Ainsi, un même modèle simplifié pourra avoir des erreurs différentes suivant l'ordre des fusions de sommets, ce qui est contraire aux propriétés nécessaires au parallélisme. Il serait envisageable de modifier la méthode de *Lindström* [LINDSTROM ET AL. 1998] pour que les résultats soient indépendants de l'ordre de fusion, mais cela nécessiterait de stocker des informations supplémentaires sur chaque arête, sous la forme d'une matrice Q' similaire à Q. Cela engendrerait un coût supplémentaire important en mémoire vive, et rendrait la méthode impraticable pour de très gros maillages.

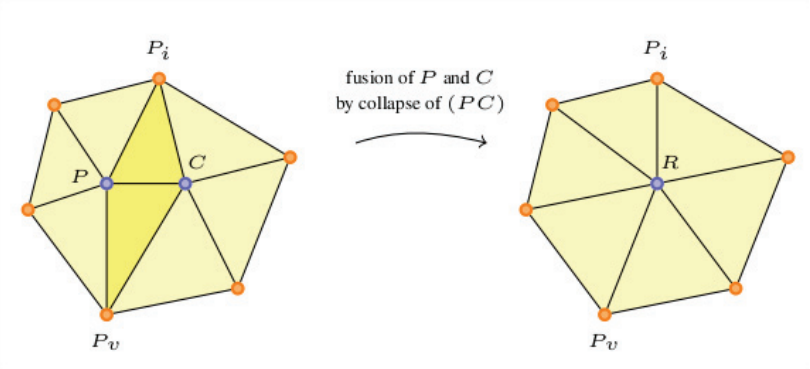


Figure 40 - Exemple de fusion inversible entre P et C. On définit le 1-ring de la fusion comme l'union des 1-rings de P et de C privée de P et C (soit les sommets en orange sur la figure).

La solution que nous proposons ici est issue de deux constats : d'abord, et ainsi que nous l'avons déjà mentionné, l'ordre de fusion des points n'influence pas la métrique d'erreur ; ensuite, l'influence d'une fusion d'arête ne dépasse pas le 1-ring de la fusion, à savoir les voisins immédiats des deux points fusionnés.

Pour réaliser cette fusion parallèle, nous utiliserons 4 tableaux dont les éléments respectifs seront des instances des structures de données suivantes :

```
typedef struct { // 1. sommets ////////////////////////////
  float x, y, z; // coordonnées

  // candidat à la fusion:
  unsigned int candidateEdge;

  // un triangle incident au point, pour atteindre 1-ring:
  unsigned int triangle;
```

```
  // valence initiale (un int pour le padding):
  unsigned int valence;

  // QEM [GARLAND 1997] coefficients:
  float a, b, c, d;
  float e, f, g;
  float h, i;
  float k;
} Vertex; // 16x4 octets par point

typedef struct { // 2. Index ////////////////////////////
  unsigned int sommet; // sommet id

  // le sommet qui stocke la description de la division:
  unsigned int child;

  // pour garder le même ordre pendant la compression et
décompression:
  unsigned int parent;

  unsigned int padding; // uniquement pour le padding
} Index; // 4x4 octets par point

typedef struct { // 3. Triangle ////////////////////////
  unsigned int a, b, c; // identifiants de sommets
  // arête ids (edgeX est opposé au sommet X):
  unsigned int edgeA, edgeB, edgeC;

  unsigned int padA, padB; // uniquement pour le padding
} Triangle; // environ 8x4x2 octets par point

typedef struct { // 4. Arête ////////////////////////////
  unsigned int trgA, trgB; // triangles adjacents

  float error; // erreur générée par cette contradiction

  // Position du point restant:
  float rX, rY, rZ;

  unsigned int padA, padB; // uniquement pour le padding
} Edge; // environ 8x4x3 octets par point
```

Les coordonnées des sommets sont stockées avec une référence à l'arête incidente qui pourrait constituer une candidate pour une future fusion. Les sommets, arêtes et triangles correspondent également à la description de la surface avec un accès direct au 1-ring de chaque sommet. Nous avons également ajouté une structure d'indices qui permet de reconstruire l'arbre binaire généré par les fusions et de travailler sur la base de pointeurs

plutôt que sur les instances des sommets. Il est à noter que les structures de données ont été conçues de façon à manipuler des maillages hétérogènes, avec ou sans bord.

Le champ *valence* de la structure *Vertex* est utilisé lorsque le point provient d'une fusion de sommets (par contraction d'arête). Il permet de stocker la valence du sommet, c'est-à-dire son nombre de voisins, au moment de son apparition comme résultat de la fusion. Ainsi que nous le verrons dans la section suivante, cette valeur est fondamentale pour la décompression.

Le tableau d'indices possède une double utilité. Il permet en premier lieu d'échanger la place des éléments sans devoir recopier toutes les informations associées à un point, ce qui sera utile lors de l'étape de tri détaillée plus bas. En outre, il décrit, par l'intermédiaire du champ *child*, la structure d'arbre binaire associée à l'algorithme de simplification. Notons qu'un seul champ *child* est nécessaire, car le second enfant est stocké dans la structure *Vertex* du point fusionné non prioritaire (le point C dans notre notation), avec les autres informations nécessaires à la décompression et détaillées dans la section suivante. Enfin, il permet, par l'intermédiaire du champ *parent* de distinguer facilement les points supprimés des points conservés.

La structure *Edge* permet de mémoriser l'erreur associée à chaque contraction d'arête, ainsi que la position du point résultant de l'opération. Elle accélère la recherche du meilleur candidat à une fusion donnée en évitant de recalculer systématiquement chaque matrice QP. Cette structure évite également de tester à nouveau chaque arête après chaque fusion de points. La taille des structures a été complétée de façon à atteindre le plus petit multiple de 16 octets, et garantir ainsi un alignement adapté à notre architecture matérielle, les *GPU*, qui fonctionnent sur 4 tableaux de flottants pour les applications 3D. Un autre point à noter est l'absence de pointeur dans les structures utilisées : l'objectif est d'être conforme à la future norme *WebCL* en cours de standardisation, qui interdit l'usage des pointeurs pour des raisons de sécurité.

Munis de ces 4 tableaux, nous proposons l'algorithme de simplification suivant :

– Initialisation :
  1. En parallèle, pour chaque point, on calcule la matrice QP initiale qui décrit l'équation de distance quadratique aux plans incidents à P.
  2. On fixe S, le seuil d'erreur maximum autorisé pour les fusions (S est défini par l'utilisateur).
– Tant que le nombre de points du modèle est supérieur au nombre de points désiré :

  Si le nombre de fusions est 0 : on augmente le seuil S

  Sinon :

  1. Simplification :

(a) En parallèle, pour chaque point P : on stocke l'identifiant de l'arête E incidente à P dont la contraction génèrerait l'erreur minimale (ou bien "infini" si cette erreur minimale est supérieure à S)

(b) En parallèle, pour chaque point P : Si P et C sont chacun candidat l'un de l'autre pour une fusion, ils passent en état "souhaite fusionner"

(c) En parallèle, pour chaque point P : Si P souhaite fusionner (avec C) ET aucun voisin de P (autre que C) ne souhaite fusionner avec une erreur inférieure ET P est prioritaire sur C, c'est-à-dire id(P) < id(C) :

i. On calcule la position du point R, issu de la fusion

ii. Les coordonnées de P sont remplacées par celles de R

iii. Q_P est mise à jour par $Q_P = Q_P + Q_C$

iv. Les informations permettant l'inversion de la fusion sont stockées dans Q_C (*cf* remarque 1)

v. On parcourt le 1-ring de la fusion : pour chaque triangle T du 1-ring :

- Si T possède un point unique qui a fusionné : on met à jour les indices de T,
- Si T possède deux sommets qui ont fusionné : on supprime T en mettant ses indices de sommets à -1 (*cf* remarque 2)
- Sinon, aucun traitement effectué.

vi. On parcourt le 1-ring de la fusion pour mettre à jour les arêtes

vii. On met à jour le tableau d'indices, à savoir le champ *child* de l'indice pointant sur P et le champ *parent* de l'indice pointant sur C

2. Tri : on trie en parallèle (par tri bitonic [BATCHER 1968]) le tableau des indices de sommets, afin de garantir un état identique lors de la simplification et du raffinement.

Remarques :

1. Pour pouvoir inverser la fusion, il est nécessaire de stocker les informations suivantes : les positions de P et de C, leurs valences de *split*, les indices de Pi et Pv qui forment avec P et C les deux triangles qui disparaissent (cf. figure 40), ainsi qu'un bit de configuration. De plus, il est nécessaire de stocker la position dans le tableau d'indices de l'élément pointant sur le point P. Nous verrons en détail dans la section suivante comment ces informations sont utilisées pour réaliser la séparation (ou *split*), c'est-à-dire la réciproque de la fusion. De plus, puisque le point C est retiré du maillage, ses structures de données correspondant aux sommets et arêtes peuvent être utilisées librement au cours de ce stockage.

2. Notons également que si le triangle T possède un sommet qui a fusionné, alors tous ses sommets appartiennent au 1-ring de la fusion complétée par le point C. Or, comme le 1-ring d'une fusion est bloqué par l'algorithme et ne peut donc pas fusionner, il est impossible que deux sommets d'un même triangle fusionnent séparément, chacun de son côté. Les 2 points ont donc nécessairement fusionné ensemble et le triangle T disparaît.

L'étape de tri est indispensable pour garantir un état identique au niveau du codeur et du décodeur. Elle permet en outre de séparer les points conservés des points supprimés après simplification. Pour cela, le tableau d'indices (c'est-à-dire de structures *Index*) est trié par l'indice du *parent* (c'est-à-dire le point R, résultat de la fusion, mais parent pour le split), puis par l'indice du sommet lui-même. A la fin du tri parallèle, il existe un seul élément dans le tableau qui ne possède pas de *parent* (autrement dit, qui n'a pas encore été fusionné) et dont l'élément suivant dans le tableau représente un point supprimé. On appellera cet élément le pivot du tableau. À noter que le fonctionnement intrinsèque du tri bitonic [BATCHER 1968], qui traite des couples d'éléments en parallèle, permet d'économiser une passe sur le tableau d'indices. Effectivement, la position du pivot, et donc le nombre de points encore présents dans le maillage à l'issue de la simplification, est détectable sans surcoût pendant la dernière étape du tri.

On notera que l'algorithme détaillé ci-dessus n'est rien d'autre qu'une écriture parallèle du processus suivant : pour chaque point P, on cherche dans son 2-ring privé de ses bords, l'arête d'erreur e minimale, avec e < S. Il est d'ailleurs possible de regrouper les étapes (a) (b) et (c) de l'algorithme en une seule. Il suffit pour cela de s'assurer qu'aucun voisin du 1-ring de P n'a de possibilité apparente de fusion avec une erreur inférieure. Dans le cas où une telle fusion concurrente pourrait exister, il n'est pas possible de vérifier que les points P et C' sont bien candidats réciproques, puisque l'étape (b) n'a pas été réalisée au préalable. Il peut donc arriver dans ce cas que certaines fusions soient bloquées inutilement, dégradant de fait très légèrement les performances du parallélisme sans pour autant mettre en cause la stabilité de l'algorithme qui est garantie par la notion de priorité.

On peut également remarquer que tout le travail de simplification est effectué par le processus attaché au point P alors qu'à première vue, il pourrait paraître judicieux de laisser des calculs sur le processus attaché au point C (le candidat à la fusion non prioritaire, qui est donc supprimé à l'issue de la fusion). Cependant, en présence d'un premier parallélisme au niveau des points, il ne nous a pas paru opportun d'ajouter de la complexité à l'algorithme en découpant la mise à jour en deux traitements asynchrones.

Nous avons remarqué expérimentalement que la simplification d'une surface plane s'appuyant sur une grille régulière était seulement 3 fois plus rapide que l'implémentation séquentielle du *QEM* [GARLAND 1997]. Cela s'explique par le fait qu'un point P possède dans ce cas plusieurs candidats équivalents pour la fusion (autrement dit portant la même erreur). Or, un choix arbitraire ou aléatoire risque d'empêcher certaines fusions pourtant légitimes, comme le montre la configuration de la figure 41. Dans de nombreuses méthodes, en cas de candidats de priorité égale, il est de coutume de choisir celui qui permettra d'obtenir les triangles de meilleure qualité. Cependant, ce critère nécessiterait de calculer toutes les fusions et de plus, il ne permettrait pas de trancher dans le cas particulier d'une grille régulière. Nous avons donc plutôt choisi de mettre en place une sous-grille qui, en priorisant les fusions au sein d'une même cellule, augmente artificiellement le nombre de

configurations analogues à celle de la figure 41 : chaque point choisira son candidat en fonction de l'erreur associée, puis, pour départager les ex-aequo, il choisira prioritairement un point qui se situe dans la même cellule de la sous-grille que lui, et dont les coordonnées sont minimales. Ainsi, lors de la fusion parallèle d'un plan, par exemple, chaque point aura tendance à choisir son voisin en haut à gauche, sauf le point se situant à l'extrémité d'une cellule de la sous-grille. En effet, un point se situant au bord de la cellule (D sur la figure 41) choisira nécessairement le candidat qui l'a lui-même choisi (C en l'occurrence), et la fusion pourra avoir lieu. Ainsi, en définissant cette sous-grille, on augmente le nombre de fusions parallèles : on passe d'une fusion unique pour l'ensemble de la région (le point à l'extrémité de la région), à une fusion par cellule de la sous-grille (le point à l'extrémité de la cellule).

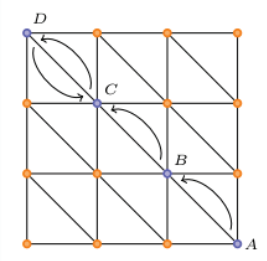


Figure 41 - Dans cette grille régulière planaire, fusionner B et A produit la même erreur que de fusionner B et C. Dans ce cas, on peut par exemple imposer de toujours sélectionner le candidat de coordonnées minimales : B choisit de fusionner avec C et A choisit de fusionner avec B. Dans cette configuration, seuls C et D fusionnent car D est contraint par sa position au sommet de la grille. Pour limiter ces cas de désaccord qui ralentissent la simplification, nous avons mis en place une sous-grille qui augmente artificiellement le nombre de points contraints.

Notons ici que cette parallélisation implique des fusions fondées sur un minimum local et engendre donc des résultats sous-optimaux. En effet, pour une même erreur maximale (le seuil S), nous obtiendrons un maillage simplifié contenant plus de points qu'en utilisant des minima globaux. Cependant, ainsi que nous l'avons évoqué précédemment, la métrique *QEM* [GARLAND 1997] n'est pas sensible à l'ordre des fusions, et les erreurs des fusions sous-optimales ne se cumulent pas. Ainsi, notre simplification parallèle permet d'obtenir un résultat qui possède seulement 10 à 15% de points supplémentaires par rapport à la méthode classique, pour des temps d'exécution jusqu'à 30 fois inférieurs à ceux obtenus par l'implémentation de référence, sur un ordinateur équipé d'un processeur Intel Core 2 duo 9200 (2,8 Ghz, 2 coeurs) et d'une carte graphique Nvidia Quadro FX 2700M.

Dans certains cas, il peut être souhaitable d'interrompre le processus de simplification lorsque le nombre de points souhaité est atteint. Or, connaître le nombre de points supprimés à partir du tableau d'indices trié est équivalent à connaître la position du premier point supprimé dans ce tableau (celui qui suit immédiatement le point pivot défini plus haut), d'où l'importance d'effectuer le tri du tableau d'indices à la fin de chaque étape de simplification.

L'un des inconvénients de notre simplification est qu'elle doit être effectuée simultanément sur toutes les mailles. En effet, une modification sur un sommet va introduire un nouveau minima qui a de fortes probabilités de modifier l'arbre généré par l'algorithme.

Or, certains maillages ont trop de simplexes pour être traités directement en mémoire, et c'est pourquoi nous utilisons la même segmentation en arbres binaires que celle introduite par le *BDAM* [CIGNONI ET AL. 2003].

Au final, grâce au tri bitonic[40][BATCHER 1968], la complexité de notre solution est $n\,log^2 n$, où n désigne le nombre de sommets du maillage. Ainsi, chaque tâche parallèle possède une complexité de $n\,log^2 n$ lorsque celle de l'algorithme séquentiel est égale à la taille de la mémoire utilisée par la simplification, soit $n\,log n$. Il serait possible d'obtenir de meilleures performances en remplaçant le tri bitonic [BATCHER 1968] par un tri radix[41]. Cependant, étant donné que ce dernier n'est pas stable, l'algorithme ne serait alors plus adapté au *streaming*.

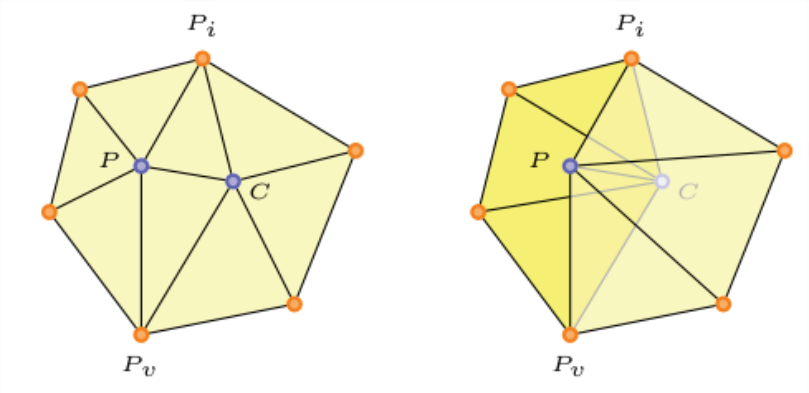


Figure 42 - Deux possibilités de split d'un point. Sans le bit d'attachement à P et C, il est impossible de choisir entre ces deux configurations lors de la phase de raffinement. Si les maillages traités sont des variétés (manifolds), seuls ces 2 cas sont possibles.

La figure 43 ci-après montre le résultat obtenu par l'implémentation originale du *QEM* [GARLAND 1997], comparé à notre algorithme de parallélisation. Les différences visuelles entre les deux versions de ce modèle 3D ne sont pas significatives. Expérimentalement, nous avons remarqué que c'est le cas pour la plupart des modèles surfaciques continus d'ordre 1.

[40] http://en.wikipedia.org/wiki/Bitonic
[41] http://en.wikipedia.org/wiki/Radix_sort

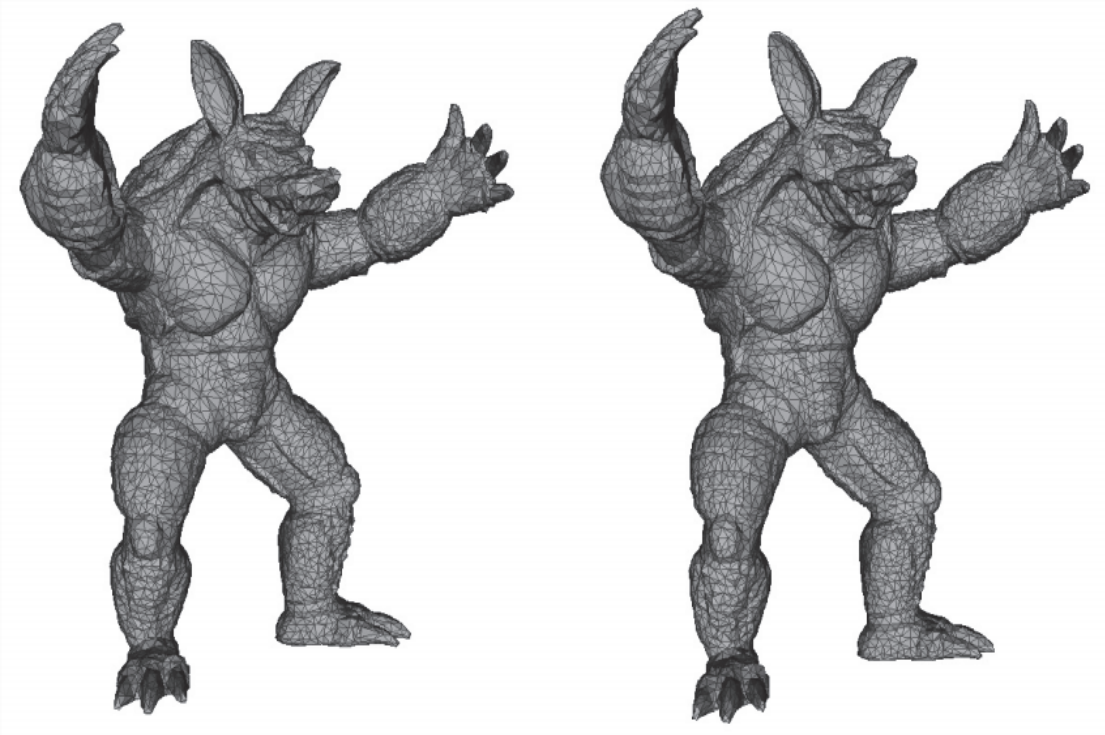


Figure 43 - A gauche, le maillage 3D obtenu par l'algorithme de simplification séquentiel et à droite, par notre méthode de parallélisation.

La figure 44 illustre l'influence du seuil d'erreur sur le maillage simplifié résultant. En effet, ce paramètre définit le taux maximum d'erreur toléré durant la simplification, ce qui permet à l'utilisateur de spécifier le niveau de précision du maillage obtenu. Dans ce cas précis, le modèle est un MNT de 2 millions sommets et la simplification a été stoppée à 500 000 sommets. En haut de la figure, aucune erreur maximale n'a été définie, et le résultat peut être obtenu à partir de seulement 10 étapes de simplification successives. En bas de la figure, le seuil a été défini à zéro et l'algorithme nécessite quant à lui 18 itérations. Nous pouvons remarquer que dans ce dernier cas, l'algorithme abandonne presque toutes les fusions en dehors de la mer de façon à préserver la précision du modèle. Par conséquent et ainsi que nous pouvons l'observer, le seuil est utile notamment si l'on souhaite augmenter la précision du modèle simplifié, au détriment du nombre d'itérations durant le calcul. En effet, en définissant l'erreur maximale à une valeur plus élevée, nous obtenons plus de régularité dans la densité des points et une meilleure efficacité du parallélisme durant la simplification.

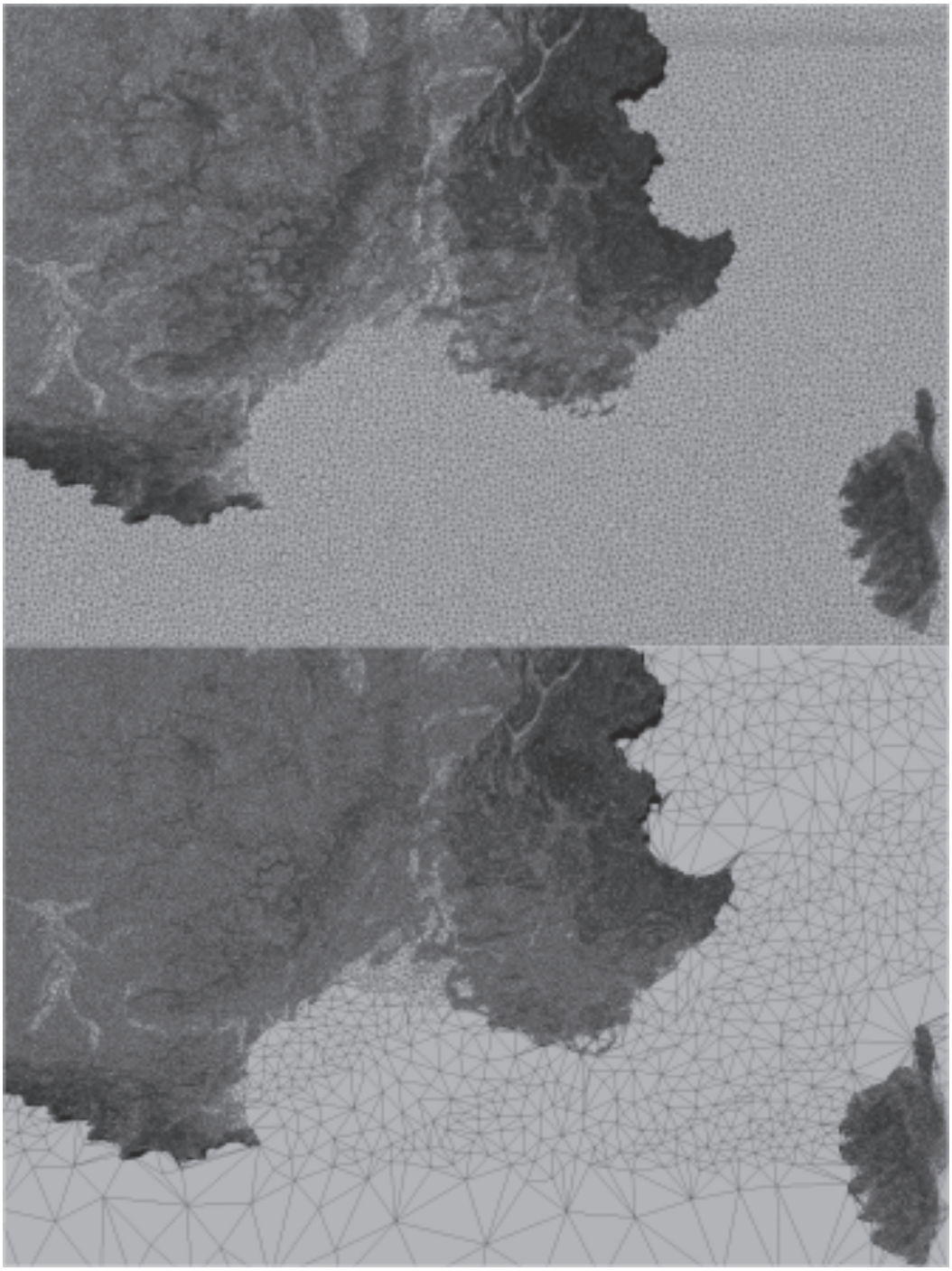


Figure 44 - impact du seuil d'erreur sur une simplification allant de 2 millions de sommets à 500 K

4.3. Compression et *streaming*

L'algorithme de simplification parallèle que nous venons de présenter est un élément essentiel de notre méthode de compression et de visualisation progressive de terrain. Il doit maintenant être inséré dans un processus de compression réversible et compatible avec la notion de *streaming*. De plus, et pour des raisons applicatives, il est impératif que le processus de raffinement bénéficie lui aussi du parallélisme. En effet, tandis que la

simplification est généralement effectuée une fois seulement et peut donc être réalisée sur des clients lourds, le raffinement sera quant à lui habituellement exécuté par les navigateurs Web.

Comme démontré par *Hoppe* [HOPPE 1996], la simplification par contraction d'arête est inversible. Ainsi il est toujours possible de retrouver le maillage original à partir d'un maillage simplifié par la méthode précédente. Cependant, afin de préserver la connectivité du maillage original, deux contraintes doivent être respectées : i) L'ordre de fusion des points lors de la simplification doit être strictement inversé lors du raffinement (c'est-à-dire la visualisation sur le client) ; ii) La séparation (ou *split*) de R ne peut être effectuée que si tous les voisins de ce point (c'est-à-dire son 1-ring), sont les mêmes qu'au moment où R a été créé, à l'issue de la fusion correspondante.

Dans notre cas, l'unique opération effectuée lors du raffinement est le *split*. C'est pourquoi un point ne peut jamais voir sa valence (son nombre de voisins) diminuer mais seulement augmenter, lorsque ses voisins effectuent un split à leur tour. Par conséquent, seule la valence v_R du point au moment de son apparition (à l'issue d'une fusion) est nécessaire au décodeur : pendant la décompression, un point sera candidat au *split* lorsque sa valence atteindra la valeurv_R. Bien que notre algorithme permette d'obtenir autant d'états intermédiaires que nous le souhaitons, nous avons choisi de passer d'une représentation grossière à une représentation plus fine par paliers, en doublant approximativement le nombre de points à chaque étape, abandonnant ainsi la possibilité d'un raffinement continu du modèle. Cela permet d'augmenter le volume des informations à transmettre et donc le taux de compression. Bien sûr, ces séparations de points par lots ne sont possibles qu'à condition de respecter les deux contraintes évoquées précédemment.

L'avantage d'utiliser la valence est qu'elle permet au visualiseur d'identifier de façon parallèle les points qui sont candidats au split : ce sont tous ceux dont la valeur actuelle de la valence est égale à la valence v_R lors de leur apparition en tant que point résultant de la fusion. Notons que la distribution des valences, dont les déviations par rapport à la médiane suivent celles d'une Gaussienne, est particulièrement bien adaptée à une compression entropique.

Cependant, pour rétablir la connectivité originale entre les points lors du *split*, il est également nécessaire de connaître les deux arêtes confondues par la fusion des deux sommets (représentées par (R_{Pv}) et (R_{Pi}) sur la figure 40 au début de ce chapitre), ainsi que l'appartenance de P et C aux triangles du 1-ring, pour pouvoir lever l'ambiguïté exposée dans la figure 42.

En ce qui concerne la géométrie du modèle, les positions des deux points P et C issus du split seront exprimées à partir de la position du point R résultant de la fusion. Il est d'ailleurs possible de contraindre la position du point R à l'ensemble formé par les deux points fusionnés P et C et de leur milieu. Au-delà des bénéfices au niveau de la compression

entropique, nous avons observé que dans le cas des bâtiments ou d'autres formes géométriques simples, cette contrainte n'engendre que très peu d'erreurs. Bien sûr, cette remarque est moins pertinente pour les terrains, pour lesquels on préférera transmettre moins de points mais en revanche mieux placés. Par conséquent, dans ce dernier cas, il est préférable de choisir un placement optimal pour réduire le nombre de points transmis et donc le nombre de triangles à afficher pour une même erreur et une taille constante de données transférées. Le choix d'utiliser le placement optimal ou contraint dépendra donc de la donnée à simplifier. Nous avons choisi de fixer le type de placement pour l'ensemble de la simplification.

En définitive, pour pouvoir inverser chaque fusion, l'algorithme de décompression a besoin des informations suivantes :

- Lors de la décompression, P et C seront générés à partir de R. Il est donc naturel de transmettre les positions de ces deux points à partir de celle de leur parent commun. On stockera donc les distances respectives de P et C à R : 2 fois 3 flottants pour un placement optimal, ou bien, en cas de placement contraint, 3 flottants (x, y, z) plus un bit définissant si R est le milieu de (PC) ou bien confondu avec l'un des deux points ;
- De manière analogue, pour coder les valences respectives pour lesquelles les deux nouveaux points P et C seront à leur tour candidats au split, on stockera les écarts respectifs à la valence de R, soit deux entiers DVP et DVC ;
- Les identifiants des arêtes Pi et Pv parmi le 1-ring de la fusion ;
- La configuration du 1-ring (cf. figure 42), codée sur 1 bit.

Nous rappelons que dans notre implémentation, toutes ces informations sont écrites dans l'espace mémoire précédemment dédié à la matrice QC du point supprimé. En effet, lors de la fusion, le point P d'identifiant minimal est conservé et contient le résultat de la fusion (que nous avons appelé R). L'autre point, C, est supprimé et contient les informations nécessaires au décodeur pout réaliser le *split* de R. Ces informations pourraient être réunies dans la structure *Vertex*, mais pour économiser quelques transtypages (*casts*), nous avons choisi de répartir ces informations entre les structures *Vertex* et *Edge* de la manière suivante :

```
typedef struct { // 1. Sommet ////////////////////////////
  // différence entre R et P (pour récupérer la position P):
  float x, y, z;

  // edge id, fournit un espace de stockage supplémentaire:
  unsigned int candidateEdge;

  // index of the previous child of P:
  unsigned int triangle;
```

```
  unsigned int valence; // non utilisé

  // anciens coefficients QEM [GARLAND 1997]:
  // a pour spécifier la configuration
  // b stocke l'erreur de l'ancien id utilisé pour le nouveau
sommet
  float a, b, c, d;
  float e, f, g;
  float h, i;
  float k;
} Vertex;

typedef struct { // 2. Edge ////////////////////////////
  // différences entres les valences  de séparations :
  // trgA stores v(P) - v(R) and trgB stores v(C) - v(R)
  unsigned int trgA, trgB;

  float error; // non utilisée

  // différence entre R et C (pour récupérer la position C):

  float rX, rY, rZ;

  // les indices des arêtes Pv et Pi dabns le 1-ring:
  unsigned int padA, padB;
} Edge;
```

Naturellement, pour que le *streaming* fonctionne, il est nécessaire que l'ordonnancement des points durant le raffinement côté client soit identique à celui utilisé durant la simplification. Afin de garantir cette cohérence, nous avons choisi de réordonner les points après chaque phase de simplification. Une fois le tableau trié, les points supprimés sont situés à la fin du tableau. Ainsi, nous pouvons réitérer la méthode de simplification avec tous les éléments du tableau pour les indices allant de 1 à p, p étant la position du pivot, c'est-à-dire du dernier point conservé par le lot de simplification.

Lorsque les fusions sont terminées, un deuxième traitement est effectué sur le tableau d'indices afin de créer les fichiers binaires utiles à la décompression du maillage, qui sont à leur tour comprimés par un algorithme entropique. Cette seconde étape possède un double objectif :

- Spécifier les informations que le client ne peut pas déduire seul : identifier les points qui possèdent leur valence de split mais dont le split n'est pas souhaitable car il introduirait trop peu d'information. Un bit par candidat est nécessaire, dans le but de spécifier si le split doit avoir lieu ou non.
- Définir la finesse des étapes de décompression : en effet, il est possible de réaliser une décompression continue ou, au contraire, de définir des étapes

effectuant davantage de *splits*. Limiter le nombre d'étapes permet d'augmenter le nombre d'informations envoyées sur le réseau pour chaque requête (ou stockées dans un fichier), et augmente donc l'efficacité de la compression.

Après la décompression entropique par laquelle débute chaque étape de visualisation, il est possible de faire la mise à jour du maillage en parallèle. Néanmoins, avec un CPU Core 2 duo 9200M, une carte graphique Nvidia 2700M, et des *buffers* de triangles d'une taille de 32k points (taille maximale pour avoir une bonne compatibilité avec le *WebGL* dont la limite actuelle se situe à 64 points), il est apparu qu'effectuer les opérations de manière séquentielle en C++ constituait la solution la plus efficace en termes de rapidité et de gestion de la mémoire du GPU. Cette observation provient des limitations actuelles des GPU dans la vitesse de transfert des données entre CPU et GPU. Cependant, dans le cas des navigateurs Web, et avec l'arrivée du *WebCL*, la version parallèle de la mise à jour devient indispensable. Elle permet en effet de pallier les faibles performances du *JavaScript* par rapport aux langages natifs tels que le C, même sur des GPU de puissance modérée.

L'algorithme de décompression utilisé par le client lors du raffinement du maillage s'écrira donc finalement comme suit :

1. En parallèle, pour chaque point : si la valence du point est égale à sa valence de *split* :

– On ajoute le point au buffer des candidats. L'ajout étant parallèle, le buffer des candidats ne sera pas ordonné. Cependant même si l'ajout se fait de façon parallèle, connaître et assurer l'unicité du pointeur de la zone mémoire correspondant au candidat nécessite d'utiliser une fonction synchrone (atomic_inc en *OpenCL*).

2. On trie en parallèle les candidats suivant leurs indices (tri bitonic) pour s'assurer que leur ordre soit le même que lors de la phase de simplification.

3. En parallèle, pour chaque point R candidat au split (c'est-à-dire dont la valence est égale à celle observée après la fusion) : Si R porte une erreur trop grande, le point doit être « *splité* »:

– On ajoute les deux points P et C provenant du *split* avec les informations sur leur valence de *split*
– On ajoute les nouveaux triangles dans le modèle : (Pi PC) et (Pv PC)
– On parcourt les triangles du 1-ring de la fusion afin d'y remplacer R par P ou C

Actuellement, nous nous heurtons au même problème que *Hu et al.* [HU ET AL. 2009], c'est-à-dire qu'en pratique, l'ajout du point dans le buffer des candidats est effectué de manière séquentielle. Cependant, tout comme *Hu et al.*, nous restons persuadés qu'avec le temps, les mécanismes fournis par les cartes graphiques seront de plus en plus performants et permettront de lever ce problème particulier. Il est d'ailleurs intéressant de noter que

notre méthode de compression pourrait aussi être utilisée dans le cadre des travaux de Hu et al. pour permettre une visualisation *out-of-core* progressive par le réseau.

En termes de compression, les résultats expérimentaux obtenus par notre méthode concernant les coûts de connectivité sont présentés dans le tableau 45. Lorsque l'on compare ces résultats à ceux de l'état de l'art, une marge de progrès est encore possible : la meilleure méthode de compression de maillage permet d'obtenir autour de 15 bits par sommet comprenant l'information géométrique, comme évoqué par exemple dans les résultats de comparaison de *Jamin* et al. [JAMIN ET AL. 2009]. Néanmoins, ces méthodes sont centrées sur la problématique de compression : l'utilisation d'un algorithme de visualisation reposant sur des chemins de triangles est bien moins efficace au regard du temps d'exécution et ne permet pas la multi-résolution. Inversement, la méthode de compression qui prend en compte la qualité de visualisation obtient un taux autour de 30 bits par sommets, ce qui est comparable à nos résultats.

4.4. Conclusion et perspectives

Nous avons présenté dans ce chapitre une méthode de simplification parallèle de maillages irréguliers compatible avec de la compression et du *streaming*. Cette méthode, utilisée conjointement avec le découpage en *patchs* de la méthode *BDAM* [CIGNONI ET AL. 2003], fournit une solution de visualisation complète offrant un nombre paramétrable d'états intermédiaires, particulièrement adaptée aux systèmes d'information géographique, et utilisable aussi bien pour des applications desktop classiques qu'au sein d'un navigateur Web. Cette solution reposant sur des considérations locales, elle permet de réduire la quantité d'informations nécessaires à la décompression en ne transmettant qu'une structure compacte lors du *streaming* et en laissant le client reconstruire le modèle original de façon parallèle une fois le flux décomprimé.

Parmi les perspectives d'amélioration de notre travail, la parallélisation du *streaming* occupe une place importante. En effet, notre solution passant par un codage entropique, il est nécessaire lors de la lecture d'un *patch* sur le réseau, de décomprimer le flux de façon séquentielle, ce qui constitue aujourd'hui un goulet d'étranglement pour la méthode. Il serait intéressant de travailler sur la possibilité de paralléliser cette décompression, voire le transfert des informations lui-même.

Une autre possibilité serait de remplacer la compression entropique du protocole HTTP par un encodeur arithmétique couplé avec un modeleur statistique adapté à notre format de données de sortie. Néanmoins, il faut s'assurer au préalable qu'un tel encodeur n'ait pas d'impact sur le temps de décompression.

Dans tous nos traitements, nous avons pris le parti d'utiliser la segmentation issue de la méthode *BDAM* [CIGNONI ET AL. 2003] pour découper le modèle en sous-parties (*patchs*)

facilement stockables et transmissibles sur le réseau. Le défaut majeur de ce principe de tuilage est d'interdire certaines fusions pour éviter la création de cassures dans le maillage. Par conséquent, le choix de la taille et de la position des *patchs* influence sensiblement la simplification du modèle. Nous investiguons actuellement la possibilité d'effectuer une segmentation du modèle respectueuse de la topologie et qui n'influence pas ou peu les choix de fusion des points.

Dans un futur proche, nous souhaitons également aborder le problème de la gestion efficace des textures des objets 3D modélisés afin de l'intégrer efficacement à notre méthode actuelle, à la fois en termes de compression et de *streaming*.

Enfin, la solution présentée dans ce mémoire n'est pas compatible aujourd'hui avec les plus légers des terminaux mobiles (smartphones dont les CPU ont des capacités très limitées ou qui ne possèdent pas de GPU). Cependant, nous restons persuadés que cette incompatibilité n'est que très provisoire au vu de l'évolution rapide des technologies de matériel mobile.

Modèles	Taille des informations de connectivité	Taille par sommet
France (résolution 1km)	1,90 Mo	15,2 bits
Asian Dragon	2,08 Mo	16,6 bits
Armadillo	1,97 Mo	15,8 bits

Figure 45 - Résultats expérimentaux selon la taille de connectivité du maillage (c'est-à-dire la séquence de code dans son intégralité et sans les informations de placement des points). Pour chaque modèle[42], 5 niveaux de détails ont été codés : 1M, 500k, 250k, 125k et 62500 sommets.

[42] http://graphics.stanford.edu/data/3Dscanrep/

5. Conclusion

5.1. Synthèse

Dans ce mémoire, nous avons commencé par identifier les limites des solutions actuelles, qu'elles soient technologiques (état de l'art des outils Web) ou algorithmiques (état de l'art des méthodes de visualisation 3D de modèles de terrain). Dans un premier temps, nous avons proposé une solution, *OpenScalesGL* [CELLIER 2012], dérivée d'une méthode classique de visualisation 3D, adaptée aux nouveaux défis liés au contexte d'exécution qu'est le navigateur et tirant parti de la toute nouvelle (à l'époque !) API WebGL. Puis, dans la perspective d'améliorer notre visualisation SIG, nous sommes partis à la recherche de puissance de calcul dans le navigateur.

En proposant les WebCLWorkers, nous avons introduit la puissance du GPU au sein des navigateurs Web, élargissant la portée de nos travaux au-delà de la seule problématique du rendu 3D. Cet outil apporte en effet aux navigateurs une puissance de calcul inédite, en compensant la lenteur du moteur *JavaScript* par la puissance des GPU (et des CPU multi-cœur), désormais accessible à travers un langage certes moins soupl que *JavaScript*, mais optimisé pour la mémoire et les temps d'exécution sans pour autant compromettre la sécurité.

Néanmoins, pour pouvoir exploiter les WebCLWorkers de manière optimale, il était nécessaire que les algorithmes de visualisation SIG tirent parti du parallélisme dit « de données ». C'est pourquoi notre troisième contribution a consisté à développer une méthode de simplification parallèle compatible avec la compression et le *streaming* de données.

Au cours de cette thèse – et ce fut pour nous l'un des principaux défis à relever – nous avons littéralement assisté à l'évolution de la sphère Internet et de ses technologies. D'abord avec l'émergence de solutions propriétaires sous forme de plug-in – Flash a été un temps un acteur majeur de la 3D dans le web – et leur déclin progressif. Puis avec l'apparition d'API spécialement conçues pour la visualisation haute performance comme le *WebGL* de la norme *HTML5*. Ces dernières évolutions n'ont pas été limitées à quelques *bindings* vers OpenGL ou DirectX mais ont mis en place de tous nouveaux concepts ou des améliorations radicales de l'existant. Les plus notables sont les « typedArray » ou « ArrayBuffer » qui permettent de gérer le binaire dans le navigateur, avec un coût en performance et en mémoire modéré pour le premier et insignifiant pour le deuxième. Nous pouvons également évoquer l'évolution du *garbage collector* incrémental, qui évite la plupart des pauses durant l'exécution par rapport à son homologue classique. Nous ne citerons pas ici toutes les (r)évolutions du Web, mais ce changement de contexte perpétuel

a participé à la difficulté de l'étude, changeant ainsi constamment les données et les suppositions.

Néanmoins, nous avons fait en sorte de nous adapter à ces fluctuations et même d'en tirer parti avec *OpenScalesGL* [CELLIER 2012], dont le code a été affiné au fur et à mesure des versions de navigateur, nettoyant le code inutile au fil du temps (le cache mémoire, par exemple, n'est plus nécessaire pour éviter les gels de l'affichage avec le nouveau *garbage collector*). Nous avons ainsi prouvé qu'il était possible d'obtenir un client de visualisation 3D portable dans le navigateur, rivalisant avec les clients lourds et qui soit même capable de fonctionner sur téléphone portable. La gestion des jointures des tuiles est même traitée dynamiquement, avec un impact négligeable sur les performances. Rappelons que le point le plus problématique étant devenu la communication réseau au niveau du débit (et parfois de façon moindre, la latence), il a été nécessaire d'effectuer plusieurs adaptions à l'existant et des remises en question concernant des choix de méthode, notamment sur le fait que le stockage des données est relativement lent mais suffisamment rapide pour supprimer toute donnée qui n'est pas en mémoire dans la carte graphique (ce qui n'est pas le cas avec un accès Internet). Par ailleurs, cette première contribution ne s'arrête pas à la visualisation mais s'étend à l'utilisation de standards et de normes provenant de l'OGC[43], conçues pour la 2D et utilisables directement dans la 3D sans aucun changement côté serveur. *OpenScalesGL* prouve donc que les navigateurs ont aussi suffisamment de puissance pour compenser les calculs nécessaires à la correction de la 2D vers la 3D dans le cadre de la visualisation de terrains disponibles sur plusieurs téraoctets de données.

Malgré toutes ces évolutions, les ressources disponibles dans le navigateur restaient insuffisantes pour la gestion de traitements plus lourds comme par exemple la simplification de maillage à la volée ou encore la reprojection de maillage irrégulier ou régulier mais non « rasterisée ». De plus, si l'on regarde les dernières évolutions dans l'état de l'art de la visualisation 3D, beaucoup de nouvelles techniques font appel à des algorithmes complexes qu'il est très difficile de reproduire dans *WebGL*. L'exemple le plus flagrant est sans doute l'utilisation de la tessellation à la volée pour gérer les niveaux de détails de la géométrie et de la connectivité voire parfois de la texture. C'est dans ce contexte que nous avons abandonné un temps nos recherches dans le domaine de la 3D pour nous intéresser au portage des techniques de GP/GPU (programmation générique sur GPU) dans les navigateurs Web. Ainsi, en proposant les *WebCLWorkers*, notre approche a consisté à aller chercher la puissance de calcul dans le parallélisme, en mettant en place un partage fin des ressources disponibles (CPU ou GPU). Au-delà d'apporter cette puissance au navigateur, nous avons étudié ses impacts sur la sécurité, et avons défini les prérequis matériels minimum. Notre solution permet des gains sensibles dans la vitesse d'exécution, des facteurs autour de 50 ayant été obtenus expérimentalement. Alors que jusqu'à présent la sécurité se concentrait surtout sur des codes bas niveaux, notre approche fondée sur des

[43] Open Geospatial Consortium

changements opérant de source à source permet désormais de profiter de toute la puissance des drivers GP/GPU des différents fabricants en montrant le résultat de la transcompilation au développeur pour que ce dernier garde un contrôle proche du C et puisse ainsi optimiser son code. En découpant le partage de calculs sur des devices sans autoriser le multi-devices, nous avons choisi de favoriser la simplicité d'utilisation, qui n'est pas la priorité de la future norme WebCL à l'édification de laquelle nous continuons de participer aujourd'hui.

En effet, un point important de cette seconde contribution est la collaboration avec le consortium Khronos. Nous avons en effet pu échanger directement avec de grands constructeurs et mettre en lumière les limites des modèles actuels. Nous avons aussi rédigé, en collaborations avec Samsung, des extensions pour les drivers facilitant grandement la mise en œuvre de la sécurité, et préparons actuellement une publication commune en lien avec nos travaux et la future norme WebCL.

Cette nouvelle puissance ainsi mise à disposition du navigateur s'accompagne cependant d'une contrainte de taille : le parallélisme. En revenant à notre problématique initiale de visualisation, il devenait donc nécessaire d'adapter les algorithmes existant pour les rendre parallélisables – ce qui va d'ailleurs dans le sens de l'évolution des unités de calcul observée depuis une dizaine d'années, avec la multiplication des cœurs et la stagnation des cadences. Indépendamment de cette contrainte de parallélisation, nous souhaitions développer une méthode de rendu en niveaux de détail permettant de conserver l'aspect irrégulier des maillages affichés. Nous avons ainsi choisi d'utiliser l'erreur quadratique décrite par Garland pour définir un algorithme de simplification qui puisse être exécuté sur *GPU* grâce à nos *WebCLWorkers* ou à la future API *WebCL*.

L'un des principaux avantages de cette méthode, qui constitue notre troisième contribution, est que le nombre de traitements parallèles est lié au nombre de points et de triangles du maillage à traiter, puisque chaque triangle et chaque point représentent un traitement parallèle. De surcroît, la parallélisation est particulièrement efficace car elle n'implique pas l'utilisation de verrous (ou *mutex*, on parle dans ce cas d'algorithme *lockless*). Autre avantage, la simplification des minimas locaux couplés à l'erreur quadratique présente l'avantage de ne pas perpétuer l'erreur d'un mauvais choix de fusion sur tous ses descendants. En effet, l'erreur n'est pas liée à l'ordre des fusions, ce qui permet de tolérer un ordre sous-optimal du point de vue de la qualité du maillage simplifié. Expérimentalement, pour une erreur de simplification donnée, nous avons observé une augmentation du nombre de points inférieure à 20% supérieurs en passant de la simplification itérative à la simplification parallèle. Enfin, nous avons résolu les problèmes relatifs aux grilles régulières et aux choix multiples par la mise en place de priorités implicites, tout comme si nous nous trouvions dans des grilles régulières, ce qui permet de tirer un meilleur parti du parallélisme.

Après cette synthèse, nous pouvons poursuivre la réflexion en nous interrogeant sur les voies ouvertes par ce travail et les possibilités d'amélioration désormais envisageables.

5.2. Perspectives

Dans cette thèse, nous avons identifié chaque point important qui semblait manquer à nos besoins pour la réalisation d'un client SIG complet dans un navigateur Web et sans plugin. Puis, pour chaque point, nous avons proposé une ou plusieurs solutions. Maintenant que ces besoins sont satisfaits, il s'agit de réaliser un site capable d'assurer la reprojection de données « raster » et vectorielles, mais permettant également à l'utilisateur de covisualiser des données distinctes provenant de plusieurs serveurs, laissant le client gérer les reprojections compatibles. Il serait également intéressant de définir des modèles multi-résolution spécialement adaptés aux différentes projections mais aussi au *streaming* et pourquoi pas aux traitements de simulation aussi bien dans le domaine de la physique (écoulement de l'eau, propagation de sons) que dans celui de l'éclairement des bâtiments.

Nous l'avons vu, le transfert et le traitement de données est au centre des dernières limites des navigateurs Web. Bien que le réseau ait atteint aujourd'hui un débit et une couverture confortables, réduire le trafic peut s'avérer un enjeu important pour les entreprises. Cela permettrait en effet d'économiser de la bande passante et par conséquent de limiter les coûts de fonctionnement. Quoi qu'il en soit, la compression sera toujours utilisée, soit explicitement, soit par le mécanisme même du protocole http, dont les algorithmes de compression actuels sont itératifs. Pour cette raison, la conception d'algorithmes de compression parallèles (ou plus précisément, compatibles avec une décompression parallèle) nous paraît être une piste prometteuse qui pourrait générer des gains importants, tant en termes de coût que de performance.

Enfin, dans le cadre de la segmentation des terrains avant simplification, nous réfléchissons également à une technique de découpage qui serait moins contraignante que celle proposée par la méthode *BDAM* [CIGNONI ET AL. 2003], c'est-à-dire dont l'influence serait moindre sur la simplification du modèle. Cela permettrait, en relâchant toute contrainte sur l'algorithme de simplification, d'obtenir des modèles simplifiés offrant une meilleure fidélité aux terrains originaux.

Annexes

Annexe 1 : Attestation des contributions de Fabien Cellier à la définition du standard WebCL (Khronos)

Dears Raphaelle Chaine, Pierre-Marie Gandoin and Aurelien Barbier-Accary,

It is my pleasure to provide an attestation of the contributions of Fabien Cellier, towards the WebCL standard, which is currently being defined.

Fabien Cellier has been actively participating in, and contributing to the Khronos WebCL Working Group and the WebCL Standard in progress, since 2011. He has been a critical contributor and resource to the WebCL effort. He actively contributed to the following:

1. OpenCL Security Extensions submitted by WebCL Working group.
2. WebCL Kernel Validator Request for Quotations.
3. WebCL Kernel Validator design and implementation in JavaScript, which is currently work in progress.
4. Comparison between WebCL and Parallel JavaScript (RiverTrail in Firefox), to show the value of WebCL.
5. He has provided input to the WebCL/OpenCL security white paper (work in progress).
6. Active contribution to review and resolution of WebCL Bugzilla entries.
7. Contribution to WebCL API design.

The WebCL working group considers Fabien Celiier a valuable and critical resource, and we value his continued contributions. We appreciate the continued support from his managers and employing organization, to encourage and enable Fabien to contribute towards the WebCL standard. WebCL allows compute intensive Web applications to gain direct access to parallel hardware for acceleration, with native performance in addition to portability across heterogeneous platforms. We are nearing finalization of the WebCL standard, during which time his continued contribution to the standard in progress will allow it to be successfully ratified and released.

Please let me know if you desire any additional information.

Sincerely,

Tasneem Brutch, Ph.D.
WebCL Working Group Chair
Senior Manager
Samsung Electronics
Cell: 408-712-7858

Annexe 2 : Mise en œuvre d'une multiplication de matrice avec WebCL (prototype Nokia), les WebCLWorkers et les WebWorkers.

Commentaires	WebCL (prototype de nokia)	WebCLWorker	WebWorker
n de plateforme contenant les devices n'existe que pour l'OpenCL/WebCL	Platform = WebCL.getPlatforms()[0] ;		
urce correspond à une chaîne de caractères décrivant la multiplication de matrice. uit être une "string" ou un fichier sur le réseau, dans notre cas c'est un fichier.	source = "matrice.cl " ;		source = "matrice.js" ;
sit le device sur lequel l'exécution sera réalisée	devices = platform.getDevices(CL.TYPE_ALL) ;//renvoie un tablea de device	devices = webCLWorker.devices ; //le tableau peut être filtrer par type avec la fonction getByType de WebCLWorker	
ion de contexte dans le webCL est une zone qui peut être partagée entre plusieurs devices d'une même platforme	ctx = WebCL.createContext(devices.Platform) ;		
ème, la notion de programme correspond à la rce et doit être associée dans un contexte. us, le programme doit être compilé avec une mmande de facon explicite alors que les CLWorkers le font de manière asynchrone et automatique	program = ctx.createProgramWithSource (source) ; program.build (/*device sur les on souhaites compiler*/ devices)		
es workers (pour l'OpenCL cela correpsond à la ı d'execution). A noter que préciser le device le choisir où va être exécuté le programme. Les rkers ne peuvent être executés que sur le CPU,	Worker1 = ctx.createCommand(devices[0]/* GPU dans notre cas d'étude*/) ; Worker2 = ctx.createCommand(devices[1] /*CPU*/) ;	Worker1 = new WebCLWorker(devices[0], source) ;	Worker1 = new Worker(source) ;
ère le kernel, c'est-à-dire la fonction qui va être cutée. Pour les webCLWorkers, toutes les ions sont faites par la même fonction et c'est dernière qui doit ensuite choisir suivant les arguments	kernel = program.createKernel(" multiply"/* , [optional devices*/) ;	task = worker.getTask("multiply") ;	
ise la mémoire qui correspond à la matrice : nts de int 32, soit une matrice 4*4	Matrice = new BufferArray(16 * 4) ;		
e les tableaux dans la zone mémoire associée evice (pour les WebCLWorkers, ce n'est pas gatoire car il peuvent être copiés à la volée. ent, si on souhaite utiliser les mêmes données leux exécutions, il faut alors utiliser le même principe que le WebCL) s, alors que la mémoire est automatiquement d'un device pour une même contexte à l'autre VebCL, cela doit être fait explicitement pour les Workers. Cela évite de masquer le mécanisme qui peut fortement impacter les performances.	mem = ctx.createBuffer(webCL.CL_MEM_READ_WRITE,4*16); Worker1.enqueueWriteBuffer(mem,false/*synchrone?*/,0 /*offset*/,16 /*length*/,Matrice);// écriture synchrone vers le device	mem = worker.getCLBuffer(worker.MEM_READ_WRITE, 4*16, Matrice);	
initialise les paramètres pour l'exécution	kernel.setArg(0,mem) ;	task.setArgs(mem/*,....*/) ;//un tableau d'arguments directement enregistrer sur la signature de la fonction //task.setArgs(Matrice fonctionne aussi)	
a l'exécution du programme qui calcule le carré trice (les paramètres correspondent au nombre ocessus" et de "threads" par processus qui sont s). La fonction alert sert alors de callback au résultat.	worker1.enqueueNDRangeKernel(kernel,1/* dimension de l'execution*/,0,16/* nombre d'execution totale en parllèle*/,8/* execution dans une même processus/workgroup?/,alert); //show popup with ""	Task.callback(alert/*,[optional boolean send result with callback*/) ; task.lauch(16) ; Show popup with ""	worker1.onMessage = alert Worker1.postMessage(Matrice) ; //show popu with the result
on sur le CPU du résulat du GPU sans passer veScript (le transfert se fait automatiquement, ne faut pas oublier que ce transfert peut être couteux	worker2.enqueueNDRangeKernel(kernel,,0,16/* nombre d'execution totale en parllèle*/,8/* execution dans une même processus/workgroup?/,alert);		
du résultat (pour le WebCL le worker n'a pas d'importance)	worker2.enqueueReadBuffer(mem,false, 0,16, Matrice) ;	worker1.enqueueReadBuffer(mem,false, 0,16, Matrice) ; //or if sendResultWithCallback is set Task.callback(function(event) { Matrice = event ; }, true) ;	/* le callback dans onmessage donne la réponse en paramètre*/ Worker1.onmessage = function(event) { Matrice = event.data ; }

Bibliographie

[AUSTIN ET AL. 1994] Austin T. M., Breach S. E., & Sohi G. S. Efficient Detection of All Pointer and Array Access Errors. In *Proceedings of the SIGPLAN 1994 Conference on Programming Language Design and Implementation*, June 1994

[BATCHER 1968] BATCHER, K. E. 1968. Sorting networks and their applications. In *Proceedings of the April 30–May 2*, 1968*, spring joint computer conference,* AFIPS '68 (Spring), pages 307–314, New York, NY, USA, 1968. ACM.

[BETTIO ET AL. 2007] BETTIO, F., GOBETTI E., MARTON F., & PINTORE, G., 2007. High-Quality networked terrain rendering from compressed bitstreams. In *Proceedings of the twelfth international conference on 3D web technology*, ACM, New York, NY, USA, Web3D '07, 37-44.

[CELLIER 2012] CELLIER F. 2012. OpenScalesGL : http://openscales.org/news/openscalesgl-announce.html, 2012.

[CIGNONI ET AL. 2003] CIGNONI P., GANOVELLI, F., GOBBETTI E., MARTON, F., PONCHIO, F., & SCOPIGNO, R. 2003. BDAM – Batched Dynamic Adaptive Meshes for High Performance Terrain Visualization. *Computer Graphics Forum*, vol. 22, no. 3, pages 505–514, (September) Proc. Eurographics 2003.

[CIGNONI ET AL. 2007] CIGNONI, P., DI BENEDETTO M., GANOVELLI, F., GOBBETTI E., MARTON, F. & SCOPIGNO, R. 2007. Ray-Casted BlockMaps for Large Urban Models Streaming and Visualization, Sept. 2007.

[COMMOWICK ET AL. 2007] COMMOWICK O., & MALANDAIN G., 2007. Efficient Selection of the Most Similar Image in a Database for Critical Structures Segmentation. In *Proceedings of the 10th Int. Conf. on Medical Image Computing and Computer-Assisted Intervention* - MICCAI 2007, Part II, volume 4792 of LNCS, pages 203–210. Springer Verlag, 2007.

[DEAN ET AL. 2004] DAEN J., GHERMAWAT S. 2004 MapReduce: Simplified Data Processing on Large Clusters from *Sixth Symposium on Operating System Design and Implementation*

[DI BENEDETTO 2010] DI BENEDETTO M., PNCHIO F., GANOVELLI F., SCOPIGNO R. 2010 *SpiderGL: A JavaScript 3D Graphics Library for Next-Generation WWW* , Web3D 2010. 15th Conference on 3D Web technology

[DICK ET AL. 2009] DICK C., KRUEGER J., & WESTERMANN, R. 2009. GPU for Scalable Terrain Rendering. In *Proceedings of Eurographics 2009 - Areas Papers*, pages 43–50, 2009.

Bibliographie

[DOUGLAS-PEUCKER, 1972] DOUGLAS-PEUCKER, 1972. Urs Ramer. An iterative procedure for the polygonal approximation of plane curves. 1972. *Computer Graphics and Image Processing*, 1:244–256.

[DUCHAINEAU ET AL. 1997] DUCHAINEAU M., WOLINSKY M., SIGETI D.E., MILLER M.C., ALDRICH C. & MINEEV-WEINSTEIN M.B., 1997. ROAMing Terrain: Real-time Optimally Adapting Meshes from IEEE Visualization '97 Proceedings

[ECK ET AL. 1995] ECK M., DEROSE T., DUCHAMP T., HOPPE H., LOUNSBERY M., & STUETZLE W., 1995. Multiresolution Analysis of Arbitrary Meshes, in *Proceeding SIGGRAPH '95 Proceedings of the 22nd annual conference on Computer graphics and interactive techniques* Pages 173 – 182.

[ELTEIR ET AL. 2011] ELTEIR M. , HESHAN L., WU-CHUN F., SCOGLAND T. StreamMR: An Optimized MapReduce Framework for AMD GPU from *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference,* pages 364-371

[GARLAND ET AL. 1997] GARLAND M., & HECKBERT P.S., Surface simplification using quadric error metrics. In Proceedings of the 24th annual conference on Computer graphics and interactive techniques, SIGGRAPH ’97, pages 209–216, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[GARLAND ET AL. 2002] GARLAND M., & SHAFFER, E., 2002. A multiphase approach to efficient surface simplification. In *Proceedings of the conference on Visualization ’02, VIS ’02*, pages 117–124, Washing-ton, DC, USA, 2002. IEEE Computer Society.

[GOBBETTI ET AL. 2006] GOBBETTI E., MARTON F., CIGNONI P., DI BENEDETTO M., & GANOVELLI F., 2006. C-BDAM - Compressed Batched Dynamic Adaptive Meshes for Terrain Rendering, september 2006. To appear in *Eurographics 2006 conference proceedings*.

[GU ET AL. 2002] GU X., GORTLER S.J. & HOPPE H., 2002. *Geometry images*. ACM Trans. Graph., vol. 21, no. 3, pages 355–361, (Juillet) 2002.

[GUIMOND ET AL. 2000] GUIMOND A., J. MEUNIER & THIRION J.P., 2000. *Average Brain Models: A Convergence Study*. Computer Vision and Image Understanding, vol. 77, no. 2, pages 192–210, 2000.

[HAKL 2001] HAKL H., 2001. Diamond Terrain Algorithm

[HECKBERT ET AL. 1997] HECKBERT P.S., & GARLAND M., Survey of Polygonal Surface Simplification Algorithms, 1997. SIGGRAPH ’97.

[HEWITT ET AL. 1973] HEWITT C; BISHOP P, STEIGER R.,*1973 A Universal Modular Actor Formalism for Artificial Intelligence*. IJCAI.

[HOPPE ET AL. 1993] HOPPE H., DEROSE T., DUCHAMP T., MCDONALD J., & STUETZLE W. Mesh Optimization, *SIGGRAPH 93,* 19-26.

[HOPPE 1996] HOPPE H., 1996. *Progressive Meshes*. ACM Press/ACM SIGGRAPH 96. New York, H. Rushmeier, Ed., pages 99–108.

[HOPPE 1998] HOPPE H., 1998. Efficient implementation of progressive meshes, *Computer & graphics*, Vol.22, No.1, 1998, pages 27-36.

[HU ET AL. 2009] HU L., SANDER P.V., & HOPPE H., 2009. Parallel view-dependent refinement of progressive meshes. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, ACM, New York, NY, USA, I3D '09, pages 169–176.

[JAMIN ET AL. 2009] JAMIN, C., G ANDOIN, P.-M., AND AKKOUCHE, S. 2009. CHuMI Viewer: Compressive Huge Mesh Interactive Viewer. *Computer & Graphics* 33, 4 (Aug.)

[LARSEN ET AL. 2003] LARSEN B.D. & CHRISTENSEN N.J., 2003. Real-time Terrain Rendering using Smooth Hardware Optimized Level of Detail. *Journal of WSCG*, vol. 11(2), pp. 282-9, 2003.

[LINDSTROM ET AL. 1998] LINDSTROM P., & Greg TURK G., 1998. Fast and memory efficient polygonal simplification. In *Proceedings of the conference on Visualization '98*, IEEE Computer Society Press, Los Alamitos, CA, USA,VIS '98, pages 279–286.

[LIVNY ET AL. 2007] LIVNY Y., KOGAN Z., & EL-SANA J., 2007. Seamless Patches for GPU-Based Terrain Rendering, WSCG 2007.

[LOSASSO ET AL. 2004] LOSASSO F., & HOPPE H., 2004. Geometry clipmaps : terrain rendering using nested regular grids. *ACM Trans. Graph.*, vol. 23, no. 3, pages 769–776, (Août) 2004.

[LOUNSBERY ET AL. 1994] LOUNSBERY M., DEROSE T., & WARREN J., 1994. Multiresolution analysis for surfaces of arbitrary topological type. Submited for publication. Preliminary version available as *Technical Report 93-10-05b, Department of Computer Science and Engineering*, University of Washington, January, 1994.

[KERNIGHAN ET AL. 1988] KERNIGHAN B.W., & RITCHIE D.M., 1988. The C programming language. Page 272. Upper Saddle River, NJ, États-Unis, March, 1988.

[NAGARAKATTE ET AL. 2009] NAGARAKATTE S., ZHAO J., MILO M.K. MARTIN, & ZDANCEWIC S., SoftBound: highly compatible and complete spatial memory safety for c. *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation.* Pages 245-258. ACM New York, NY, USA ©2009

Bibliographie

[NAGARAKATTE ET AL. 2010] NAGARAKATTE S., ZHAO J., MILO M.K. MARTIN, & ZDANCEWIC S., CETS: Compiler Enforced Temporal Safety for C. *Proceedings of the International Symposium on Memory Management,* Toronto, Canada, June 2010.

[OAKES 1999] OAKES D., 1999. Direct Calculation of the Information Matrix via the EM Algorithm. J. R. Statistical Society, vol. 61, no. 2, pages 479–482, 1999.

[PAJAROLA ET AL. 2007] PAJAROLA R., & GOBBETTI E., 2007. Survey of semi-regular Multi-resolution models for interactive terrain rendering. *Vis. Comput*., vol. 23, no. 8, pages 583–605, (Juillet) 2007.

[POUDEROUX ET AL. 2005] POUDEROUX J., & MARVIE J.E., 2005. Adaptive Streaming and Rendering of Large Terrains using Strip Masks, *Proceedings of ACM GRAPHITE* 2005, page 299--306 – 2005.

[ROSSIGNAC ET AL. 1993] ROSSIGNAC J., & BORREL P., 1993. Multi-resolution 3D approximations for rendering complex scenes. *Modeling in Computer Graphics, 1993.* Pages 455-465.

[SCHNEIDER ET AL. 2006] SCHNEIDER J. & WESTERMANN R., 2006. GPU-Friendly High-Quality Terrain Rendering, *Journal of WSCG* 2006

[SCHROEDER ET AL. 1992] SCHROEDER W.J., ZARGE J., & LORENSEN W., Decimation of Triangle Meshes, SIGGRAPH 92, 65-70.

[TURK 1992] TURK G., 1992. Re-Tiling Polygonal Surfaces, *SIGGRAPH 92*, 55-64.

[YEE ET AL. 2009] YEE B., SEHR D., DARDYK G., CHEN B., MUTH R., ORMANDY T., OKASAKA S., NARULA N., & FULLAGAR N., 2009. Native Client: A *sandbox* for Portable, Untrusted x86 Native Code. From *IEEE Symposium on Security and Privacy (Oakland'09*), IEEE, IEEE, 3 Park Avenue, 17th Floor, New York, NY 10016 (2009)

[XU ET AL. 2004] XU W., DUVARNEY D. C., & SEKAR R., 2004. An Efficient and BackwardsCompatible Transformation to Ensure Memory Safety of C Programs. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (FSE), 2004