OpenGL distribué

Équipe Resedas
Samuel Thibault

3 juin au 12 juillet 2002

LORIA, Technopôle de Nancy-Brabois Campus Scientifique
B.P. 239 54506 VANDOEUVRE-lès-NANCY CEDEX
Téléphone : (33) 03 83 59 30 00 — Télécopie : (33) 03 83 27 83 19

Remerciements

J'ai été encadré par Olivier Festor, Eric Fleury et Alain Filbois lors de mon stage au LORIA du 3 juin au 12 juillet 2002.

Je tiens à remercier :

- Xavier Cavin pour son intéressant sujet de stage et son encadrement par mail,
- Olivier Festor un des co-responsables de stage pour son encadrement,
- Eric Fleury un des co-responsables de stage pour ses idées intéressantes,
- Alain Filbois un des co-responsables de stage pour m'avoir éclairci sur les tenants et les aboutissants de VTHD
- Bruno Levy pour ses conseils à propos d'OpenGL et son logiciel Graphite,
- Greg Humphreys pour son logiciel aussi labyrinthique qu'intéressant.
- Josianne Reffort pour son aide administrative,
- Isabelle Astic pour son accueil chaleureux,
- toute l'équipe de Resedas,
- Mme Bernard pour mon emménagement,
- le serveur du restaurant du Loria pour son éternelle bonne humeur,
- et le Caméo pour son excellente programmation.

Introduction

L'action VSP (Visualization Service Provider) vise à fournir un moyen d'affichage de maquettes 3D à distance pour l'industrie. La quantité de données échangées étant importante, cette action s'incrit dans le projet VTHD / VTHD++ (plate-forme d'expérimentation IP/WDM Vraiment Très Haut Débit), qui prépare l'Internet de nouvelle génération où pourront se déployer les services interactifs du futur, grâce aux avantages du nouveau protocole IPv6 qui remplace peu à peu IPv4.

Chromium [3] est un logiciel de rendu OpenGL distribué écrit par Greg Humphreys (Stanford University) qui permet un affichage 3D dont le calcul est réparti sur un cluster de machines. Par nature donc, il est prévu pour fonctionner sur un réseau local où les délais de transmission et les pertes sont faibles ; le choix du protocole IP s'est donc au départ tourné vers TCP.

L'objectif de ce stage est donc d'adapter Chromium aux nouvelles conditions de VTHD, où UDP pourrait être plus adapté que TCP.

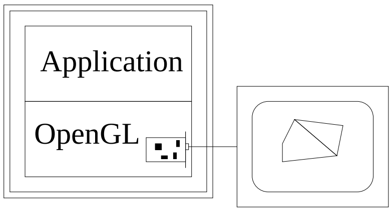
Table des matières

1 Chromium

Chromium se compose de plusieurs composants logiciels, notamment Appfaker, Tilesort, Packer et Unpacker, qui simulent des composants OpenGL. Expliquons tout d'abord comment ceux-ci s'articulent.

1.1 Principe général de fonctionnement

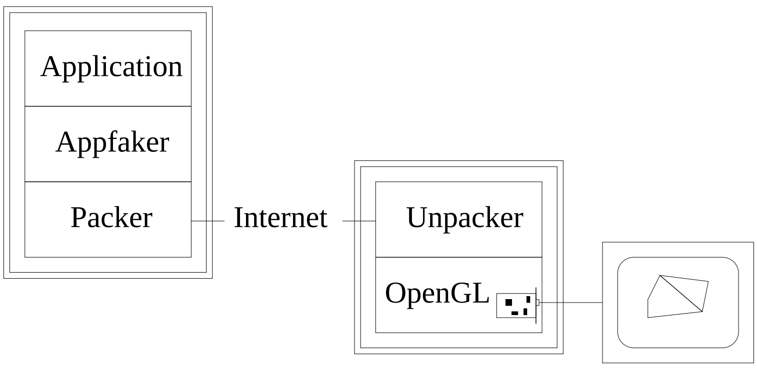
OpenGL est une bibliothèque d'affichage 3D. Cela signifie que les applications utilisent ses services pour afficher des objets 3D à l'écran :



L'application indique précisément à OpenGL ce qu'elle désire afficher : elle peut lui demander de changer de couleur, puis dessiner un triangle 3D (dont elle donne les coordonnées des sommets), puis tourner la figure, dessiner un autre triangle 3D, changer de couleur, etc... Ce sont les **appels OpenGL**. OpenGL utilise alors la carte graphique de l'ordinateur (représentée sur le schéma) pour effectuer le rendu. Les cartes graphiques récentes sont capables d'effectuer elles-mêmes une certaine partie du travail (calcul de coordonnées 3D par exemple), accélérant ainsi le tout.

Tout ceci se passe sur la machine (représentée par un double cadre) sur laquelle l'utilisateur travaille.

Chromium propose de distribuer le calcul OpenGL du rendu, c'est-à-dire de pouvoir dans un premier temps avoir une situation comme celle-ci :

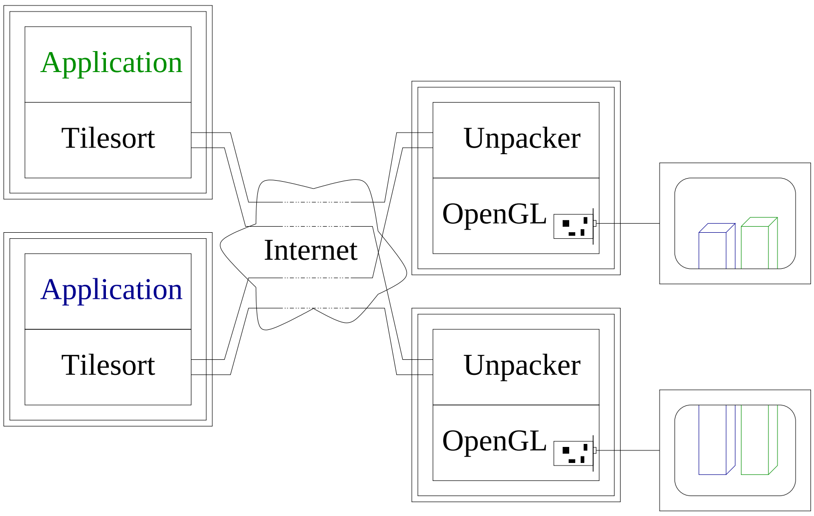


L'utilisateur est devant l'écran de la machine de droite, mais l'application tourne sur une autre machine connectée à Internet. Celle-ci croit utiliser OpenGL, mais en fait, c'est **Appfaker** (Trompe-application) qu'elle utilise, qui transmet les appels OpenGL au **Packer** (Empaqueteur) qui code ceux-ci sous forme d'un numéro de code d'opération (**opcode**) et de données pour pouvoir les envoyer sur Internet. Par exemple, l'appel « changer de couleur » aura un numéro, et la donnée sera le code de la nouvelle couleur choisie. Le Packer transmet alors ces appels codés à la machine de l'utilisateur, via Internet. Là, un **Unpacker** décode les appels et les transmet à OpenGL, comme s'il était la vraie application. OpenGL affiche donc les objets 3D sur l'écran de l'utilisateur.

Application et OpenGL sont tous deux trompés : l'affichage ne se fait pas sur la machine de gauche mais à distance, sur la machine de droite !

Ce mode de fonctionnement peut être utile si l'application ne peut tourner que sur la machine distante (si le système est différent, ou qu'elle est plus puissante que la machine locale).

Dans un deuxième temps, on peut adapter les applications à Chromium, pour qu'elles puissent réaliser un affichage distribué :



L'utilisateur est devant deux machines, dont les écrans sont posés l'un au-dessus de l'autre. Les deux applications discutent directement avec un **Tilesort**, mais lui transmettent des appels OpenGL comme si c'était OpenGL qu'elles utilisaient directement. Cependant, elles utilisent en plus des fonctions de synchronisation pour se mettre d'accord sur ce qu'il faut afficher. (Ici, chacune s'occupe de l'affichage d'une tour)

Le Tilesort s'occupe alors d'envoyer exactement les appels dont a besoin chaque machine de droite pour que l'affichage soit réparti entre les deux écrans. Par exemple, le segment qui forme la base de la tour bleue n'est envoyé par le tilesort de l'application bleue qu'à la machine du bas.

Ce mode de fonctionnement est utile à la fois quand on veut pouvoir afficher un rendu 3D sur un mur d'écrans, ou quand l'application peut être répartie sur plusieurs ordinateurs. Cela peut également être utile si l'on veut rendre plusieurs fois le même affichage sur différents écrans.

Ces deux exemples de fonctionnement ne sont pas exhaustifs, on peut imaginer tout arrangement de machines, avec d'éventuels traitements supplémentaires sur une machine avant retransmission sur une autre, etc...

1.2 Quelques mots sur les réseaux

Pour que deux machines puissent communiquer, il est nécessaire de décrire de quelle manière elles vont le faire : un *protocole* décrit sous quelle forme sont échangées les données, avec quels contrôles. On utilise en fait un empilement de protocoles, chacun utilisant les services du protocole juste inférieur, et fournissant des services plus évolués au protocole supérieur. Tout en haut, l'application utilise alors les services d'un ou de plusieurs des protocoles supérieurs :

Chromium		Application
UDP	TCP	Transport
IP		Réseau
VTHD		Physique

VTHD (Vraiment Très Haut Débit) est le réseau physique qu'on aimerait utiliser. La connection VTHD du Loria n'étant pas encore disponible, les tests on été faits sur le réseau IPv6 expérimental du Loria ;

IP (Internet Protocol) est le protocole qui fait la base d'Internet. Il décrit comment les machines s'envoient des paquets de données, mais sans préciser comment ces données doivent être organisées, c'est le protocole de transport qui doit le faire ;

TCP (Transfert Control Protocol) est l'un des protocoles de transport que l'on peut utiliser au-dessus d'IP, qui fournit un moyen sûr d'envoyer des données : les données seront reçues, et dans le même ordre où elles ont été envoyées. C'est ce protocole qu'utilisait Chromium jusqu'à ce jour ;

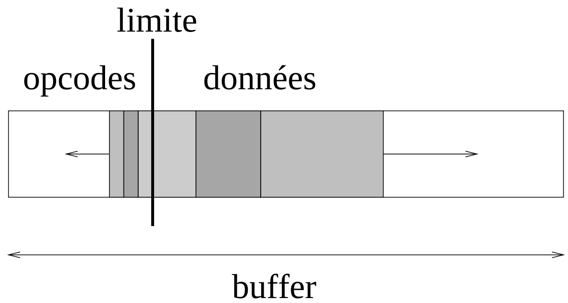
UDP (User Datagram Protocol) est un autre protocole de transport que l'on peut utiliser au-dessus d'IP, qui fournit un moyen simple d'envoyer des données : l'application envoie des paquets, qui sont reçus par l'autre ordinateur. Cependant, aucune garantie de réception ni d'ordre n'est assuré, c'est l'application qui doit compenser les pertes ou désordres. C'est ce protocole qu'on aimerait que Chromium utilise le plus possible à la place de TCP.

Les paquets transmis ne doivent pas être trop gros : la **MTU** (Maximum Transmission Unit) est la limite au-delà de laquelle IP est obligé de découper en morceaux les paquets que le protocole de transport lui fournit, pour que ces morceaux puissent eux passer sur le réseau physique. C'est une limite qui est propre au type de réseau physique utilisé, par exemple 1500 octets dans le cas d'un réseau Ethernet tel que celui utilisé.

TCP s'adapte automatiquement à la MTU en découpant lui-même de manière intelligente les données, tandis qu'UDP ne le fait pas : c'est à l'application de faire attention à la taille des paquets qu'elle confie à UDP.

1.3 En pratique

Voici comment fonctionne le Packer (voir [1] pour plus de détails) :



Il utilise un buffer où une limite opcodes/données est fixée à l'avance (1/5 d'opcodes pour 4/5 de données par défaut). Au départ, on a un paquet de longueur nulle, centré sur cette limite. Lorsque l'application donne un appel OpenGL au Packer, celui-ci note le numéro d'opcode à gauche, et les données correspondantes à droite, faisant ainsi grandir le paquet à la fois vers la gauche et la droite[1]. Si l'on risque de dépasser la zone de données d'un côté ou de l'autre, on « flushe » d'abord le tampon, c'est-à-dire qu'on ajoute au début du paquet le nombre d'opcodes, et on envoie le tout via Internet à la machine sur laquelle tourne l'Unpacker.

Grâce au nombre d'opcodes, l'Unpacker retrouve la limite opcodes / données, et peut alors relire les appels OpenGL avec leurs données, dans l'ordre où ils ont été stockés.

2 Modifications

Les modifications que j'ai apportées avaient pour objectif d'utiliser au mieux Chromium sur VTHD / VTHD++ ; elles ont été dirigées principalement dans trois directions.

1. Ce fonctionnement peut paraître étrange, mais permet d'aligner les données sur les limites de mots matériels, accélérant alors leur écriture puis lecture.

2.1 IPv6

Tout d'abord, adapter Chromium à la version 6 d'IP (IPv6). En effet, Internet utilise pour l'instant la version 4 d'IP, mais la version 6 se met peu à peu en place, notamment sur VTHD++. Cette nouvelle version adapte IP aux nouvelles conditions d'Internet : nombreuses machines, simplification des en-têtes pour un traitement plus rapide des paquets, multicast (envoi simultané à plusieurs machines),...

Il a donc fallu reprogrammer la partie réseau de Chromium pour qu'elle utilise les nouvelles fonctions d'IPv6[2] à la place de celles d'IPv4, notamment utiliser la nouvelle fonction `getaddrinfo`, et remplacer les appels à `gethostbyaddr` par `getnameinfo`. Ces nouvelles fonctions supportent à la fois IPv6 et IPv4, selon la disponibilité (voir [4]).

2.2 UDP vs TCP

Ensuite, l'un des gros enjeux du stage a été de faire en sorte que Chromium remplace le plus possible ses appels à TCP par des appels à UDP.

Chromium		Application
UDP	TCP	Transport
IP		Réseau
VTHD		Physique

En effet, TCP est pratique car il est fiable, et plutôt efficace sur un réseau local, c'est pourquoi le choix a été donné en sa faveur par Greg Humphreys.

Pour obtenir cette fiabilité, TCP, du côté récepteur, renvoie un acquittement à l'émetteur lorsqu'il reçoit correctement des données. Si un paquet est perdu, et même si les paquets suivants ne sont pas perdus, le récepteur n'acquitte pas le paquet perdu, et aucune donnée n'est fournie à Chromium, puisqu'elles sont sensées lui être fournies dans l'ordre. Du côté émetteur, la perte n'est détectée que par l'absence d'acquittement de la part du récepteur, au bout d'un certain délai. Le paquet non acquitté est alors retransmis, en espérant qu'il ne sera pas lui aussi perdu, mais bien reçu, si bien que le récepteur pourra acquitter toutes les données déjà reçues, et les fournir à Chromium.

Il apparaît ainsi un délai entre le moment où l'émetteur envoie un paquet et celui où il reçoit l'acquittement de la part du récepteur. Sur un réseau local, il est très court, mais sur un réseau plus étendu comme VTHD, où les paquets comme les acquittements doivent traverser des routeurs avant

2. IPv6 étant encore peu développé, il a fallu recompiler la version 2.2 de Python pour activer à la main le support d'IPv6.

d'arriver à destination, il devient plus long, et les pertes de paquets sont par ailleurs plus fréquentes.

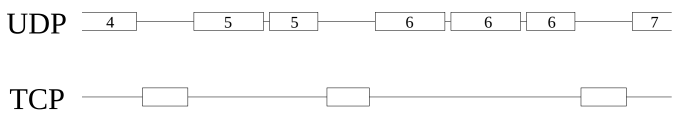
Remplacer TCP par UDP revient à supprimer les acquittements, quitte à ce que toutes les données ne soient pas forcément reçues ou du moins pas forcément dans l'ordre. Ainsi, on utilise au mieux le réseau puisque l'émetteur n'a pas à se soucier des pertes et envoie simplement les données dès qu'elles sont fournies par Chromium.

2.2.1 UDP seulement

Un premier essai a d'abord consisté à remplacer purement et simplement TCP par UDP. Ceci marche presque bien sur un réseau local, où les pertes sont rares, mais je me suis rapidement rendu compte que ce n'était pas viable, et conduisait parfois à un crash ! En effet, certains appels OpenGL ne doivent pas être perdus, tels que « ouvrir une fenêtre » ou « rendre cette fenêtre active », ainsi que ceux qui doivent retourner une valeur, ou encore les appels spéciaux de synchronisation.

2.2.2 UDP/TCP

On a donc choisi d'utiliser à la fois UDP et TCP :



Les appels qui ne doivent pas être perdus passent par TCP, tandis que le reste passe par UDP.

Il faut cependant synchroniser les deux moyens de communication, pour que les appels passant par le « canal » UDP soient correctement entrelacés avec ceux passant par le canal TCP. Pour cela, on ajoute un numéro de séquence aux paquets UDP, qui est incrémenté à chaque fois que des données sont envoyées via TCP. Le récepteur essaie d'abord de lire un paquet UDP, et s'il constate que le numéro de séquence a augmenté, il lit les données TCP, incrémente le numéro de séquence, et peut alors traiter ce paquet UDP.

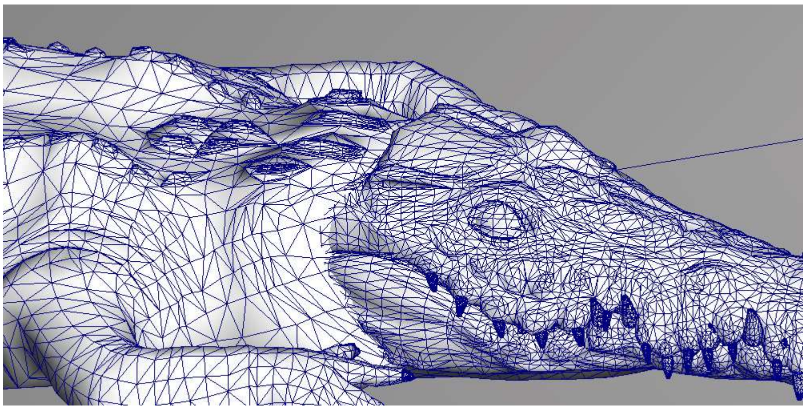
Plutôt que de recenser de manière exhaustive les appels OpenGL qui ne doivent pas être perdus, on se contente de recenser ceux que l'on peut se permettre de perdre. Finalement, on s'est contenté de ne faire passer que les blocs Begin/End par UDP.

En effet, les blocs Begin/End regroupent les appels OpenGL qui décrivent la géométrie (voir [5], 2.6 Begin/End Paradigm), et représentent ainsi 80 à 99% de la masse des données OpenGL, selon le type d'application.

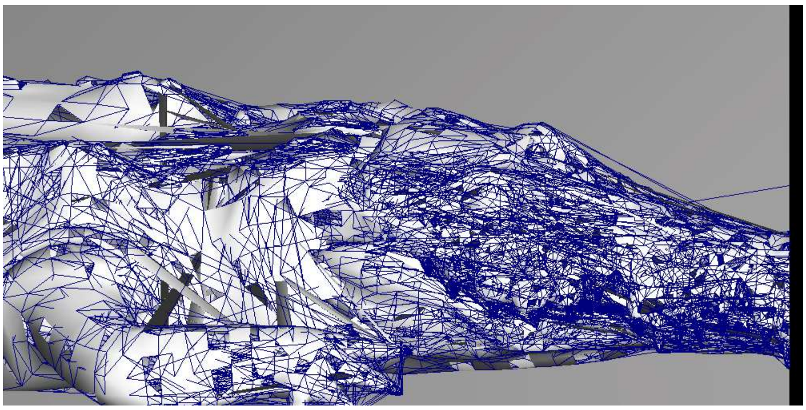
Un **bloc Begin/End** est une suite d'appels OpenGL entourés des appels Begin et End. Seuls quelques types d'appels sont autorisés au sein d'un tel bloc: ils se limitent à la description des points de l'objet 3D que l'on veut dessiner avec leurs caractéristiques (couleur, texture, orientation pour l'éclairage). Le bloc même a un type, selon que l'on veut que ses points soient considérés comme les points d'une ligne brisée, de lignes séparées, de triangles séparés, ou encore d'un polygone...

2.2.3 Pertes de géométrie

Si des blocs Begin/End sont perdus, on obtient des « trous » dans le rendu, ce qui resterait tolérable, mais d'autres problèmes apparaissent : la figure 1(b) montre des triangles étirés, des traits invraisemblables, quelque peu désordonnés.



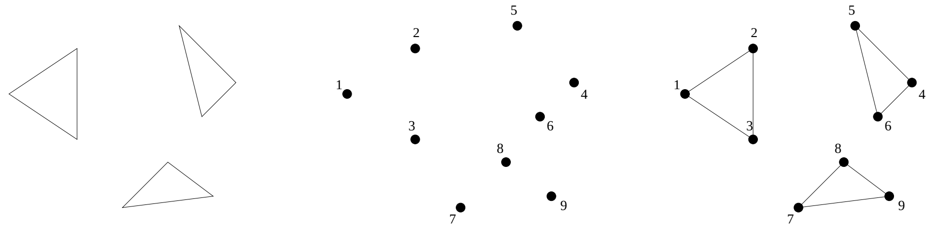
(a) sans perte



(b) avec pertes

FIG. 1 – *Premier résultat*

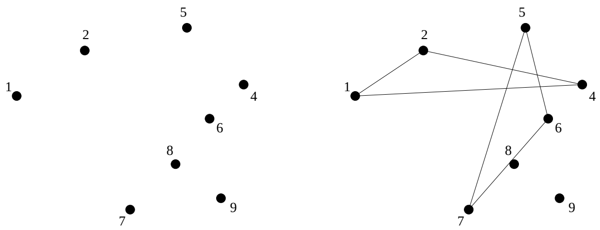
En effet, si l'application veut afficher trois triangles tels que ceux de la figure 2(a), elle utilisera certainement un seul bloc de type TRIANGLES contenant les points de la figure 2(b), dans cet ordre afin qu'ils soient reliés correctement comme sur la figure 2(c).



(a) voulu (b) envoyé (c) final

FIG. 2 – *Sans perte*

Cependant, des objets 3D tels que la figure 1(b) utilisent de nombreux points, et les blocs Begin/End sont alors plus long que la MTU (Maximum Transmission Unit), et sont donc découpés en morceaux avant d'être transmis. Si une partie d'un bloc Begin/End est perdue, on peut se retrouver dans une situation telle que celle (simplifiée) de la figure 3(a): un point et un seul a été perdu pendant la transmission. Le regroupement par trois pour former les triangles est alors incorrect, les deux derniers points sont ignorés, et le résultat figure 3(b) est complètement incorrect et ressemble beaucoup à ce que l'on avait pu observer à la figure 1(b).

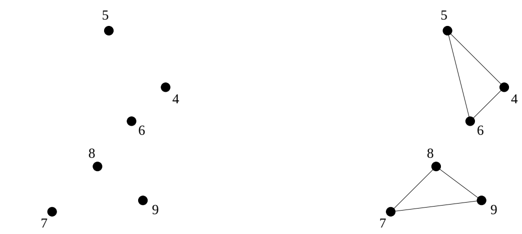


(a) reçu (b) final

FIG. 3 – *Perte d'un point*

Pour éviter cela, on regroupe les points par trois (dans l'exemple des triangles) dans les paquets UDP, afin que les pertes se fassent aussi par multiples de trois, comme sur la figure 4(a). Ainsi, on obtient le résultat 4(b) avec un trou, certes, mais qui reste cohérent.

Avec cette correction, on obtient un résultat bien meilleur (figure 5), où l'on voit uniquement d'inévitables « trous » dûs aux pertes (de l'ordre de 10% ici).



(a) reçu (b) final

FIG. 4 – *Perte d'un triplet de points*

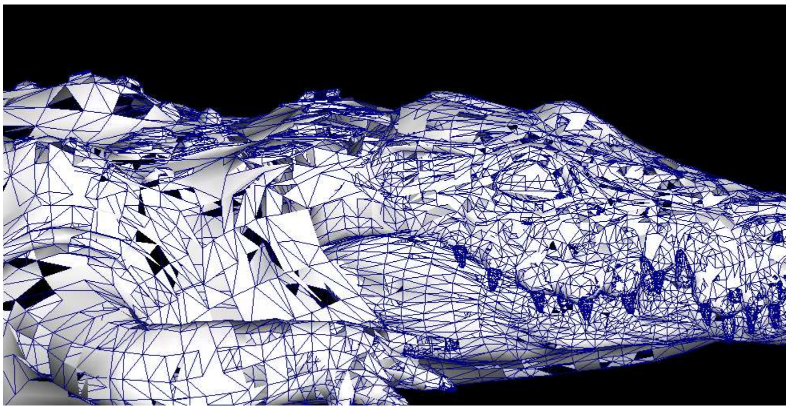


FIG. 5 – *Meilleur résultat*

Évidemment, il faut tout autant faire attention aux blocs Begin/End de type LINES (segments séparés) ou QUADS (quadrilatères).

2.3 Découverte de MTU

Enfin, un autre problème généré par un réseau étendu tel que VTHD est que la MTU devient la **Path MTU**, c'est-à-dire que la taille maximale des paquets que l'on peut se permettre d'envoyer est le minimum des MTU des réseaux traversés. Si un paquet arrive sur un réseau dont la MTU se révèle plus faible que celle des réseaux précédemment traversés, il risque fort d'être alors fragmenté pour pouvoir passer.

Déterminer la taille maximale de paquet que l'on peut se permettre n'est donc pas simple (on ne connait généralement pas la Path MTU à l'avance). Le RFC 1191 [7] décrit un algorithme pour s'approcher rapidement de l'optimum: on indique sur le paquet que l'on ne veut pas qu'il soit fragmenté, et s'il se retrouve sur un réseau dont la MTU est plus petite que la taille du paquet (nécessitant donc une fragmentation), un message indiquant cette MTU est retourné à l'envoyeur. Au bout de quelques essais, on finit ainsi par obtenir la Path MTU. Celle-ci peut par ailleurs encore diminuer, si l'organisation d'Internet change, et que l'on passe alors par des réseaux de MTU encore plus faible. De même, il faudrait de temps en temps essayer de réaugmenter la Path MTU, au cas où la nouvelle organisation permettrait de ne passer que par des réseaux de MTU plus grande.

Maintenant, il faut que Chromium s'adapte automatiquement à la Path MTU, au fur et à mesure que celle-ci est découverte, et ce n'est pas simple.

En effet, comme nous l'avons vu en 1.3 page 7, Chromium « flushe » ses buffers lorsque ceux-ci débordent, or la taille de ceux-ci est fixée comme étant égale à la MTU que l'utilisateur a entré à la main. De plus, si l'on n'utilise que peu d'opcodes avec beaucoup de données, le buffer déborde à droite [3], donc est flushé, alors que la MTU n'est pas complètement remplie, ce qui est peu efficace.

Pour améliorer cela, j'ai décidé d'agrandir la taille des buffers jusqu'à ce que la partie « données » ait la taille de la Path MTU (voir figure 6). En ne laissant alors qu'un test qui flushe simplement lorsque l'ensemble opcodes + données risque de dépasser la Path MTU, on est sûr

- de ne pas dépasser la Path MTU, le paquet pourra donc passer sur le réseau ;
- de remplir le plus possible la Path MTU puisque c'est le seul test.

3. Pour un opcode (un octet), il y a toujours au moins 4 octets de données ; la répartition 1/5 d'opcodes pour 4/5 de données assure donc que le buffer ne peut pas déborder à gauche.

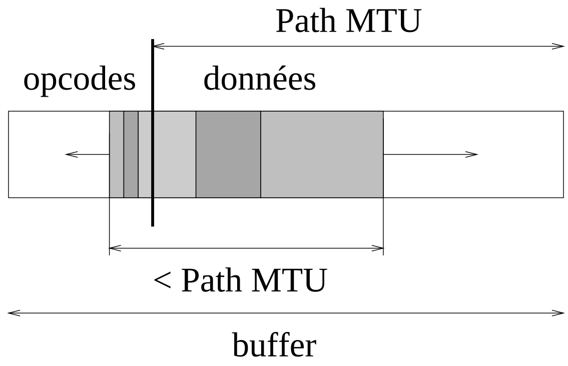


Fig. 6 – *Nouvelles limites*

Ainsi, les tests de Chromium sont automatiquement adaptés à la nouvelle Path MTU [4].

Évidemment, le temps que la Path MTU soit découverte, des paquets sont perdus et le rendu a alors un gros trou, mais ce n'est que très passager (de même si la Path MTU diminue soudainement).

3 Résultats

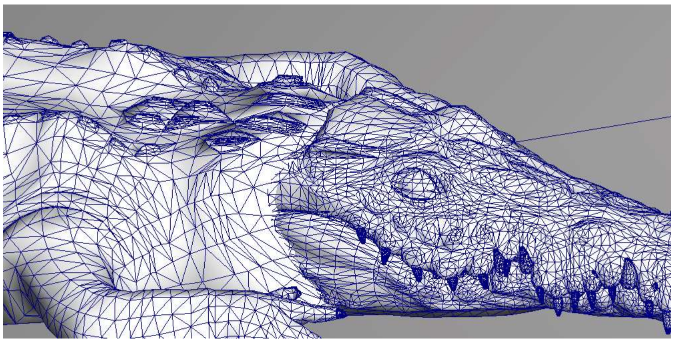
J'ai effectué quelques tests sur un réseau local entre deux machines : l'application tourne sur Bouh tandis que l'affichage se fait sur Canardo (qui a une bien meilleure carte graphique que Bouh). Pour varier les types d'application, j'en ai testé trois :

- Graphite est un logiciel d'affichage/traitement d'objets 3D, il utilise donc énormément de géométrie sous forme d'un rendu statique (Figure 7(a)) ;
- Anneaux est un logiciel de test du Tilesort : 4 applications produisent chacune un anneau, les quatre anneaux s'embriquent et sont affichés en 2 morceaux sur l'écran de Canardo, dont on peut voir la séparation sur la figure 7(b) ; on simule ainsi l'affichage sur deux écran tel qu'expliqué en 1.1 ;
- ChromiumBSU est un clone du jeu R-Type, animé, utilisant des textures (Figure 7(c)).

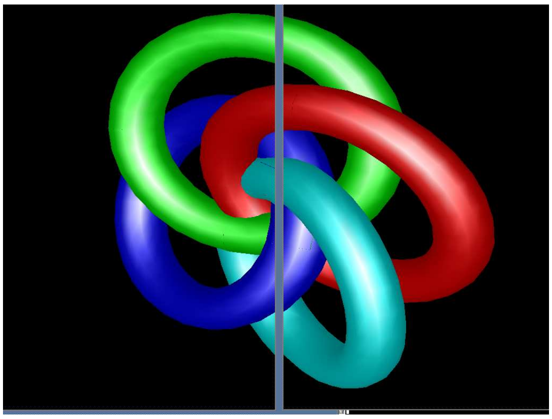
Pour chaque application, on mesure le nombre d'images produites par seconde (en fps : frame per second) [5]. Dans le cas de Chromium, on obtient

4. Les buffers restent de la même taille : il serait trop coûteux de les retailler, car ils sont utilisés par différents morceaux de Chromium, il faudrait « aller les chercher ».

5. Dans le cas de Graphite, on force un rafraîchissement incessant du rendu.



(a) Graphite



(b) Anneaux

(c) ChromiumBSU

Fig. 7 – *Applications testées*

presque toujours 50fps (le maximum), sauf lorsque le réseau est un peu encombré (lag). C'est pourquoi il n'y a pas de mesure précise, mais seulement une très vague mesure de la jouabilité.

Les tests ont été réalisés dans différentes conditions de réseau, pour voir l'impact qu'elles ont sur le comportement des solutions « tout TCP » et « UDP / TCP ».

3.1 100Mbps

Tout d'abord, on utilise toute la puissance du réseau local : 100Mbps (100BaseTX)

	Graphite	Anneaux	ChromiumBSU
TCP	2.2fps	$\sim$ 110fps	jouable
UDP/TCP	2.7fps	$\sim$ 110fps	jouable

On perçoit effectivement un petit gain dans le cas de Graphite, mais les performances sont semblables dans le cas des anneaux. En effet, ceux-ci ont une géométrie très simple, et la synchronisation entre les quatre applications se fait par TCP (sécurité oblige).

3.2 10Mbps

Pour diminuer la bande passante disponible, j'ai réglé la carte réseau de Bouh sur 10Mbps (10BaseT)

	Graphite	Anneaux	ChromiumBSU
TCP	0.28fps	$\sim$24fps	jouable
UDP/TCP	0.36fps	$\sim$39fps	jouable

On constate alors un léger gain à la fois avec Graphite et avec les anneaux.

3.3 Wireless

Enfin, on a remplacé la carte réseau interne de Bouh par une carte sans fil à 11Mbps. Cela permet de tester les conditions de fortes pertes, où TCP pert beaucoup en performances, et n'offre alors qu'un débit de 200Ko/s sur un transfert de fichier par exemple.

	Graphite	Anneaux	ChromiumBSU
TCP	0.066fps	$\sim$5.5fps	jouable, même si laggue un peu
UDP/TCP	0.37fps mais qualité médiocre	$\sim$8fps	non jouable : trop de pertes

On constate alors un gros gain avec Graphite, car TCP est obligé de retransmettre beaucoup de paquets pour compenser les fortes pertes. Un gain beaucoup moindre avec les anneaux est toujours dû au fait que ceux-ci utilisent beaucoup de synchronisation par TCP.

Cependant, le résultat à l'écran est déplorable si l'on utilise UDP/TCP, car les fortes pertes qui grèvent les performances de TCP sont alors directement visibles à l'écran !

4 Bilan

4.1 Performances

Utiliser UDP permet d'obtenir de meilleurs performances si l'on n'est pas trop regardant du côté de la qualité de rendu.

Cependant, sur un réseau local, les résultats sont plutôt mitigés, car celui-ci transmet les données presque plus vite que ce que peuvent gérer les machines, si bien que celles-ci sont bien souvent saturées. De plus, le délai de transmission étant très faible, les retransmissions de TCP se font rapidement, il n'est donc que très peu pénalisé par d'éventuelles pertes. Il faudrait donc maintenant tester sur un plus grand réseau, VTHD par exemple (lorsqu'il sera disponible), pour vérifier que TCP est alors beaucoup plus pénalisé qu'UDP.

Les performances dépendent grandement du type d'application : une application telle que Graphite profite beaucoup d'UDP car elle affiche surtout de la géométrie ; au contraire, les anneaux utilisent une synchronisation forte que l'on est obligé de faire passer par TCP.

Les défauts de rendu (trous) constatés si des paquets UDP sont perdus sont amoindris lorsque l'application est animée comme les anneaux car l'oeil compense alors en mélangeant les images successives dont les trous ont très peu de chances de se retrouver au même endroit. Ceci se fait d'autant mieux que l'on a plus d'images par seconde avec UDP. Au contraire, dans le cas de Graphite qui a un affichage statique, les trous restent visibles tant que l'on ne rafraîchit pas l'affichage !

4.2 Et encore ?...

Quelques idées, envisagées dès le départ ou apparues au cours du stage, restent encore à développer :

- Utiliser le multicast UDP d'IPv6 lorsque l'on veut afficher sur plusieurs écrans le même rendu. Ce n'est a priori pas simple car il faut tout de même gérer soi-même la diffusion TCP.
- À cause de la MTU, on ne peut envoyer les données via UDP que par petits paquets (bien souvent 1.5Ko), alors que l'on peut fournir les données à TCP par tranches de 16Ko par exemple, ce qui représente bien

moins d'appels au système. Il faudrait par exemple stocker temporairement plusieurs paquets UDP pour les donner ensemble au système grâce à `sendmsg` (au lieu de les donner un par un avec `send`). De même en réception.

- Il serait intéressant de pouvoir contrôler le flux de sortie UDP, afin d'éviter une situation où une machine rapide envoie ses données trop rapidement pour le réseau ou pour la machine qui les reçoit, ce qui entraîne de grosses pertes (le traffic est « tronqué » dans le meilleur des cas).

4.3 Conclusion

Ce stage m'a fait découvrir ce qu'est se plonger dans un gros projet encore en développement non stable, ainsi que la programmation d'IPv6 et le langage Python. J'ai également pu avoir un apperçu du fonctionnement d'OpenGL.

A Technique

A.1 Machines utilisées

A.1.1 Bouh

CPU/RAM : PIII 800Mhz 128Mo
Vidéo : ATI Rage Mobility AGP 2x
Ethernet : Intel 82557 (Pro100) (10BaseT/100BaseTX)
Wireless : Cisco Aironet 340 series (11Mbps)
OS : Debian 3.0
compilateur C : gcc 2.95.4

A.1.2 Canardo

CPU/RAM : PIII 550Mhz 256Mo
Vidéo : Matrox G400 AGP
Ethernet : 3c905B (100BaseTX)
OS : RedHat 7.2
compilateur C : gcc 2.96

A.1.3 Routeur

Cisco Catalyst 3500 series XL

A.2 Logiciels utilisés

- Chromium beta du 7 avril 2002
- Python 2.2 recompilé avec le support ipv6
- Snapshot de Graphite du 18 juin 2002
- ChromiumBSU 0.9.12

Références

[1] Greg Humphreys, Ian Buck, Matthew Eldridge, Pat Hanrahan, *Distributed Rendering for Scalable Displays*, IEEE Supercomputing 2000

[2] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, James T. Klosowski, *Chromium: A Stream-Processing Framework for Interactive Rendering on Clusters*, Stanford University, Lawrence Livermore National Laboratory, IBM T.J. Watson Research Center, SIGGRAPH 2002

[3] *Chromium*, documentation et code source, `http://chromium.sourceforge.net/`

[4] Gisèle Cizault, *IPv6 Théorie et Pratique 3e édition*, O'Reilly (2002)

[5] Mark Segal, Kirt Akeley, *The OpenGL Graphics System: A Specification (Version 1.1)*, `http://www.opengl.org/developers/documentation/Specs/glspec1.1/glspec.html`

[6] *Manuel du programmeur Linux :* `ip`, `udp`, `getaddrinfo`, `freeaddrinfo`, `socket`, `getsockopt`, `setsockopt`, `bind`, `listen`, `accept`, `connect`, `send`, `recv`

[7] J.C. Mogul, S.E. Deering, 1990, *Path MTU discovery*, RFC 1191

[8] P. Richard, P. Kynikos, 1991, *Gigabit network economics and paradigm shifts*, RFC 1214