III

FlowVR Render

La phase de rendu d'une application de réalité virtuelle nécessite l'intégration des informations concernant tous les objets de l'environnement virtuel. Même si le reste de l'application est construit de façon modulaire, l'ajout d'un nouveau type d'objet virtuel requiert fréquemment des modifications dans le code de rendu. Dans cette partie une architecture de rendu distribué est présentée, permettant de modulariser la phase de rendu pour minimiser ces modifications, tout en optimisant les performances dans le cadre d'une exécution sur grappe pilotant plusieurs projecteurs.

Notre approche est basée sur la définition d'un nouveau modèle de description de la scène 3D, exploitant les dernières avancées des cartes graphiques, les *shaders*, pour alléger les transferts réseau et permettre un implantation efficaces des opérations de fusion et découpage des données. Les performances obtenues sont comparées à un outil existant, et de nombreuses applications exploitant cette nouvelle architectures sont présentées.

Ce travail a été publié lors de la conférence IEEE Visualization 2005 [12].

Contexte et problématique 10

"ALL OUR KNOWLEDGE HAS ITS ORIGINS IN OUR PERCEPTIONS."
Leonardo da Vinci

Un système de rendu distribué est primordial pour pouvoir augmenter la surface et la résolution d'affichage, ainsi que la taille de l'environnement virtuel. Plusieurs machines sont mises à contribution pour calculer chacune une partie de l'image finale et/ou des objets dans l'environnement virtuel. Cette parallélisation peut intervenir à plusieurs niveaux dans le pipeline de l'application (chapitre 4). Nous nous intéressons ici au cas ou cette parallélisation intervient entre la génération de la scène 3D et le rendu lui même.

Traditionnellement, les systèmes de rendu distribué se basent sur une API graphique existante (OpenGL ou graphe de scène) et envoient les données correspondantes sur le réseau. L'avantage de cette approche est le support relativement aisé des applications existantes utilisant déjà cette API. En revanche l'algorithme de distribution est limité par la sémantique de l'API utilisée ainsi que le niveau d'information disponible.

Les approches basées sur les graphes de scènes [82, 123, 154] ont à leur disposition plus d'informations, mais doivent gérer les interdépendances entre les objets de la scène dues à la hiérarchie du graphe de scène. Une telle hiérarchie est utile pour gérer les différentes parties de la scène et faire des traitements en passant à l'échelle sur les grandes scènes (en construisant une hiérarchie de boites englobantes et ainsi rejeter facilement les parties non visibles). En revanche, cette hiérarchie est plus difficile à construire et à justifier pour les scènes très dynamiques, ou utilisant beaucoup les *shaders*. Les shaders sont de petits programmes exécutés par la carte graphique et contrôlant les transformations effectuées sur les sommets des objets ainsi que le calcul de la couleur des pixels, remplaçant ainsi l'étape traditionnelle de transformation des sommets par une pile de matrices. Avec un shader rien ne garantit qu'un objet va appliquer sa matrice de transformation tel quel, ni utiliser celle du noeud père dans le graphe de scène.

Les travaux liés aux API de rendu (section 2.5 page 13) et au rendu distribué (section 4.1 page 28) ont déjà été présentés, nous allons rappeler dans ce chapitre les concepts

utilisés par OpenGL, Chromium et les shaders afin de présenter le contexte et la problématique de notre travail.

10.1 OpenGL

OpenGL [159] est actuellement l'API de rendu prédominante, étant disponible pour la plupart des plateformes et des cartes graphiques. Elle utilise un modèle assez bas niveau basé sur une machine à état spécifiant comment les objets doivent être affichés (transformations, lumières, matériaux, transparence, textures, etc), ainsi qu'une description des objets eux mêmes selon un mode *immédiat*, c'est-à-dire ou chaque primitive est décrite séquentiellement pour chaque nouvelle image. Depuis l'introduction d'OpenGL en 1992, les machines et les cartes graphiques ont évolué de façon significative. Pour suivre cette évolution un mécanisme d'extensions est utilisé, permettant aux fabricants de cartes graphiques d'exposer des fonctionnalités additionnelles, correspondant aux nouvelles possibilités des cartes graphiques. Périodiquement une nouvelle version d'OpenGL est créée par l'*Architecture Review Board* (*ARB*), standardisant les extensions les plus utiles. La dernière version, OpenGL 2.0, est disponible depuis septembre 2004 et ajoute le support de fonctionnalités tel que les textures rectangulaires ou les langages de shaders de haut niveau.

L'utilisation du mode immédiat pur requiert une retransmission de toutes les primitives graphiques à chaque image, et ne peut donc pas bénéficier de la cohérence entre les images. Cela introduit un important surcoût au niveau du CPU, du GPU (*Graphics Processing Unit* – le processeur de la carte graphique), et du bus de communication entre les deux. Ce problème s'est intensifié avec l'évolution des performances des cartes graphiques. Pour réduire ce goulet d'étranglement, OpenGL a introduit plusieurs mécanismes optimisant les transferts de données vers la carte graphique. Il y a entre autres le mécanisme originel des *display-lists*, les *compiled vertex arrays*, et les récents *vertex/pixel buffer objects* d'OpenGL 2.0.

Chaque nouvelle version d'OpenGL est compatible avec toutes les versions précédentes. De ce fait, il existe de nombreuses fonctionnalités obsolètes ou redondantes, comme les multiples façons de spécifier les données des sommets, ou encore toute la machine à état liée au calcul de l'apparence des objets rendue obsolète par la programmabilité des shaders [19]. Ceci augmente significativement la complexité des implantations d'OpenGL. Un effort pour épurer les commandes et les états obsolètes du coeur d'OpenGL a été proposé pour OpenGL 2.0. Cette proposition n'a pas été acceptée, mais a abouti à une API dérivée OpenGL ES [28], dédiée aux systèmes embarqués. Cette API est utilisée dans de nombreux produits comme la future console Sony PlayStation 3.

10.2 Chromium

Chromium [84] est un outil permettant de découper et recombiner des flux de commandes OpenGL afin d'implanter un mécanisme de rendu distribué basé sur le principe du *sort-first* (section 4.1.2 page 29). Les primitives graphiques de la scène à visualiser sont produites par une application exécutée sur une ou plusieurs machines. Ces primitives sont ensuite triées et acheminées vers d'autres machines effectuant le rendu lui-même.

L'utilisation d'OpenGL permet le support de nombreuses applications sans nécessiter de modifications, Chromium remplaçant le driver OpenGL du point de vue de l'application. Ceci permet l'utilisation directe d'applications propriétaires sur un mur d'image. Chromium intercepte les commandes OpenGL émises par l'application et utilise un réseau de filtres implantant un système de caches avancés et de compression pour optimiser les communications.

Pour supporter des optimisations ou des schémas de rendu avancés, Chromium défini un certain nombre d'extensions spécifiques. Ainsi, l'utilisation parallèle de plusieurs applications clientes se répartissant le calcul de la scène est possible grâce à des extensions permettant de spécifier l'ordre de recombinaison des différents flux. Cette extension utilise un mécanisme de barrières et de sémaphores permettant d'indiquer les sections de commandes atomiques et l'ordre possible de recombinaison entre celles-ci. D'autres extensions permettent de réduire le surcoût lié à Chromium, comme la transmission des boites englobantes associées aux objets, permettant aux filtres de tri de calculer rapidement leur visibilité. Toutefois l'utilisation de ces extensions requiert une adaptation spécifique du code de l'application.

Utiliser OpenGL comme base de distribution rend difficiles les opérations de découpage et recombinaison de flux graphiques qui servent de base au rendu distribué. En particulier, les commandes OpenGL partageant un état commun, il faut faire attention de bien respecter les modifications faites à cet état. Le mode immédiat d'OpenGL impose de retransmettre beaucoup d'informations à chaque image. Ces retransmissions deviennent rapidement un goulot d'étranglement, même en rendu local. Plusieurs optimisations ont été développées et intégrées à OpenGL. Tous ces mécanismes doivent être supportés par le système de rendu distribué, ce qui augmente beaucoup la complexité de son implantation.

10.3 Shaders

Des shaders procéduraux sont très souvent utilisés pour le rendu logiciel hors ligne [80] pour spécifier l'apparence visuelle des objets. Les premières cartes graphiques étaient limitées à des calculs fixés, configurés au travers d'un ensemble de paramètres. Les nouvelles générations de carte graphique sont maintenant capable d'exécuter des shaders complètement programmables [133, 142]. Les modèles les plus récents [100] permettent

l'utilisation de langages de haut niveau [110, 153] incluant des appels de fonctions, des boucles, des structures conditionnelles, et des calculs et données utilisant une précision de 32 bits en virgule flottante.

Les shaders apporte une flexibilité et une précision additionnelle qui permet aux cartes graphiques d'exécuter des algorithmes jusqu'alors réservé à une exécution sur CPU. Ceci concerne par exemple les modèles d'éclairage de haute qualité (évaluation par pixel, ombres douces), la colorisation dessinée [69], ou le rendu volumique [151]. Ce déplacement de certains calculs du CPU vers le GPU permet parfois de réduire les transferts de données entre-eux. Par exemple en visualisation une *fonction de transfert* est souvent appliquée aux données brutes pour obtenir une représentation visuelle (couleur). Si ce calcul est effectué par le CPU, alors le résultat doit être retransmis à chaque fois que la fonction de transfert est modifiée. L'utilisation d'un shader pour appliquer la fonction de transfert permet de ne transmettre qu'une seule fois les données brutes. Un changement de la fonction n'entraîne alors qu'une modification des paramètres du shader.

10.4 Bilan

En utilisant les fonctions d'optimisation des transmissions de données comme les *buffers objects* d'OpenGL et les shaders matériels, les applications peuvent exploiter pleinement les dernières avancées des cartes graphiques. Cependant, la machine à état d'OpenGL et les nombreuses fonctions redondantes encore supportées augmentent considérablement la complexité et bride l'efficacité de toute implantation, en particulier les systèmes distribués.

L'utilisation de shaders permet de répartir les calculs entre les CPU et les GPU. En conséquence le niveau des informations envoyées à la carte graphique est modifié, celle-ci pouvant désormais traiter des données génériques en plus de simple données graphiques (couleurs, positions). D'autre part, le changement des paramètres des calculs implanté par des shaders ne nécessite plus la retransmission de toutes les autres données. Dans un contexte distribué ou les primitives graphiques sont produites sur des machines distinctes, et parfois distantes, des machines de rendu ceci peut dramatiquement changer les performances de la chaîne d'affichage.

Basé sur ces constatations, nous proposons un nouveau protocole de rendu distribué *sort-first*, reposant sur une utilisation omniprésente de shaders et dédié au rendu parallèle haute-performance.

Description d'environnements virtuels distribués 11

Dans ce chapitre, nous présentons une nouvelle approche de rendu distribué, basé sur un modèle de description de la scène spécialement conçu pour une construction la plus modulaire possible de la tâche de rendu, en utilisant les possibilités offertes par les shaders des dernières générations de cartes graphiques. Bien que les idées utilisées soient transposables à d'autres environnements, FlowVR est utilisé comme cadre pour la conception et l'implantation de ce système de rendu distribué, que nous appellerons *FlowVR Render*.

Après avoir défini les principes fondamentaux, nous nous attacherons aux éléments permettant de décrire la scène 3D et de transmettre cette description sur le réseau.

11.1 Principes fondamentaux

Nous définissons un modèle de description de la scène 3D. Cette description est générée par des modules appelés *viewers*, et envoyée aux modules appelés *renderers* effectuant le rendu sur les cartes graphiques.

Le modèle de FlowVR Render est basé sur les principes suivants :

– Chaque objet de la scène est décrit comme un ensemble de groupes de polygones (*batches*). Chaque groupe est indépendant des autres et son apparence visuelle est décrite à l'aide de shaders. Un groupe de polygones est appelé *primitive* dans le modèle.

– Les primitives ne sont pas ordonnées sauf dans certains cas particuliers (objets transparents, éléments d'interface utilisateur) spécifiés par le *viewer*. Ceci permet

aux *renderers* d'optimiser l'ordre local de rendu, réduisant par exemple les changements de textures et de shaders.
- Des informations de haut niveau comme la boite englobante ou les changements depuis la dernière image peuvent être spécifiés directement par les *viewers*, permettant de réduire les surcoûts de traitement et de transmission réseau. Pour la plupart des applications ces informations sont directement disponibles.
- Les shaders ne nécessitant qu'un petit nombre de paramètres, il n'est pas nécessaire de maintenir un état commun à toutes les primitives, ce qui simplifie beaucoup le protocole de transmission. Les opérations de découpage et recombinaison de plusieurs flux graphiques sont peu coûteuses du fait de l'indépendance des primitives.
- Les informations communiquées sont strictement unidirectionnelles, depuis les *viewers* jusqu'aux *renderers*. Ceci évite les allers-retours coûteux et permet de désynchroniser l'exécution de chaque module (par exemple envoyer les données vers un fichier pour les relire plus tard).
- Les shaders permettent d'utiliser facilement toute la puissance des dernières cartes graphiques. Certains calculs comme l'application de fonctions de transfert pour obtenir des informations de couleur à partir de données brutes peuvent être effectués par la carte graphique, libérant ainsi le CPU pour d'autres tâches.

11.2 Primitives graphiques

Comme indiqué dans la section précédente, notre modèle se base sur une scène 3D formée d'un ensemble de *primitives*. Chaque primitive décrit un groupe de polygones en spécifiant l'ensemble des *ressources* (coordonnées, textures, shaders) à utiliser. Les propriétés associées aux primitives et ressources sont présentées sur la figure 11.1.

Pour éviter la duplication des ressources utilisées par plusieurs primitives, ainsi que des ressources et primitives inchangées entre chaque image, il est nécessaire de pouvoir identifier les ressources et primitives envoyées et ainsi pouvoir les réutiliser. On associe donc à chaque ressource et primitive un identifiant unique *ID*. Chaque primitive spécifie les identifiants des ressources qu'elle utilise. Ceci introduit une dépendance unidirectionnelle entre les primitives et les ressources, mais pas d'interdépendance entre les primitives elles-mêmes.

Le système doit garantir que les identifiants utilisés soient globalement uniques pour toute l'application distribuée. Il existe plusieurs méthodes pour le faire, en particulier utiliser un serveur centralisé de nommage, ou alors inclure dans l'identifiant des informations locales à chaque machine garantissant sont unicité. La première méthode permet de générer des identifiants de petite taille, mais introduit un point de contention qui limite le passage à l'échelle du système. L'implantation de FlowVR Render utilise donc la seconde approche. Des identifiants sur 64 bits sont générés en combinant un compteur atomique local à chaque machine avec l'adresse IP de la machine.

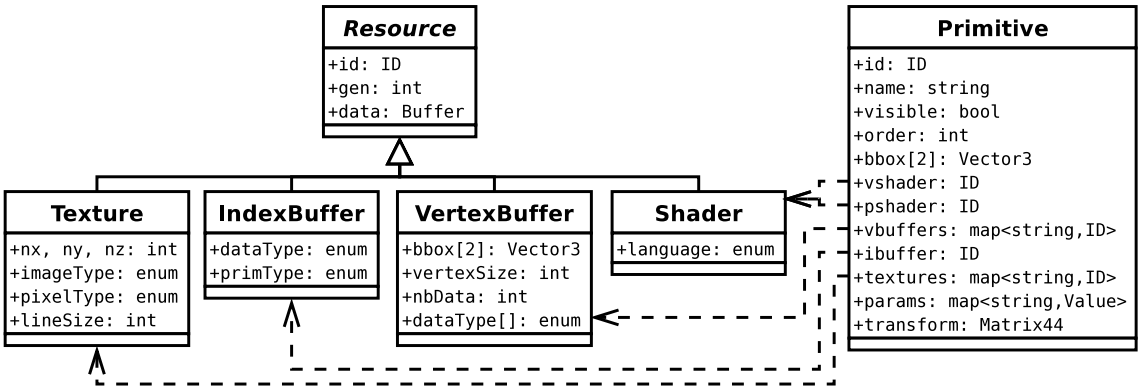


Figure 11.1 *Schéma UML des objets utilisés dans la description de la scène graphique.*

11.2.1 Ressource

En plus de son identifiant, chaque ressource possède un nombre *gen* spécifiant la génération actuelle. Ce nombre doit être incrémenté à chaque fois que la ressource est modifiée, et permet de savoir quelle donnée est la plus récente. Un numéro de génération égal à -1 signifie que la ressource ne sera a priori pas modifiée et que donc le module de rendu peut en optimiser son stockage (transférer les données directement dans la mémoire de la carte graphique par exemple). Ceci n'est toutefois qu'une indication, il est tout à fait possible de modifier la ressource plus tard, en réponse à un réglage de l'utilisateur par exemple. Le but de cette distinction est de fournir une heuristique simple pour déterminer ou stocker chaque ressource pour le rendu. Cela peut aussi être utile pour le filtrage des transferts réseaux. Une donnée statique peut être transmise qu'elle soit visible ou non, étant donné qu'elle va être utilisée pendant une longue période, alors qu'une donnée plus dynamique mérite d'être filtrée vu qu'elle est retransmise souvent et donc utilise une part importante de la bande passante.

Les données elles-mêmes de la ressource sont stockées dans un buffer FlowVR, qui est donc en mémoire partagée, ce qui évite de les dupliquer lorsque plusieurs modules ou filtres les utilisent. Le reste des propriétés des ressources dépendent du type de la ressource, qui peut être une *Texture*, un *Index Buffer*, un *Vertex Buffer*, ou un *Shader*.

11.2.1.1 Texture

Une texture stocke une table de valeurs 1D, 2D ou 3D, utilisée généralement pour plaquer une image sur les objets de la scène, mais peut aussi servir à stocker d'autres informations (normal maps, voxels, *Look Up Tables*) utilisées par le shader. Chaque texture possède une taille *nx*, *ny* et *nz* selon les axes X, Y et Z. Cette taille est égale à 0 si la texture est de dimension inférieure.

Les valeurs stockées à l'intérieur de chaque pixel de la texture sont spécifiées selon un

type de donnée spécifié dans *pixelType* en utilisant un type énuméré regroupant les types de base (*byte*, *int*, *float*, ...), certains types spéciaux (comme *null* spécifiant que toutes les données sont nulles), ainsi que de petits vecteurs ou matrices de taille fixe. Par exemple, une image couleur utilise généralement un vecteur de 3 octets pour chaque pixel.

Les données des pixels peuvent être interprétées de manière différente selon la convention utilisée, comme par exemple *RGB* spécifiant que la première valeur représente la composante rouge, la deuxième représente le vert, et la troisième le bleu, contrairement à la convention *BGR* qui spécifie l'ordre opposé. Cette convention est indiquée par *imageType*. Enfin les données sont stockées en mémoire de manière contiguë par ligne puis par plan. Il arrive souvent que le début de chaque ligne soit aligné sur un bloc de n octets. Pour prendre en compte cet alignement la variable *lineSize* indique le décalage entre chaque début de ligne.

11.2.1.2 Vertex Buffer

Les sommets qui composent les polygones à afficher sont spécifiés dans des ressources de type *vertex buffer*. Chaque sommet contient plusieurs données appelées attributs. Le nombre de ces attributs est spécifié par la variable *nbData*, et leur type est stocké dans le tableau *dataType*. Ce tableau utilise le même type énuméré que celui utilisé précédemment pour les pixels des textures. La taille des données de chaque sommet est spécifiée dans la variable *vertexSize*. Elle permet aussi de connaître le nombre de sommets contenus dans le buffer en divisant la taille totale des données par la taille de chaque sommet.

Un vertex buffer peut stocker des données indiquant la position 3D des sommets. Cette information à une incidence sur la visibilité des polygones et peut être utilisée pour filtrer les communications réseaux. Pour faciliter ce filtrage la boite englobante des sommets contenus dans le buffer est indiquée dans *bbox*. Dans le cas ou le buffer ne contient pas de donnée de positions mais uniquement d'autres attributs (couleurs, vecteurs normaux), alors cette boite est spécifiée comme vide.

11.2.1.3 Index Buffer

Une fois les attributs des sommets spécifiés, il est nécessaire de les relier pour former des points, lignes, ou polygones. Pour cela on utilise un *index buffer* qui contient les indices des sommets à utiliser. Ces indices sont stockés dans le buffer de donnée de la ressource en utilisant le format spécifié par *dataType* (généralement des entiers sur 8, 16 ou 32 bits en fonction du nombre de sommets). Dans certains cas les sommets sont utilisés exactement dans le même ordre qu'ils sont stockés dans les vertex buffers. Il n'est alors pas nécessaire de stocker les indices, et *dataType* contient alors *null*.

Le type de primitive formé par les sommets est spécifié dans la variable *primType* et correspond aux primitives classiques utilisées par OpenGL (figure 11.2).

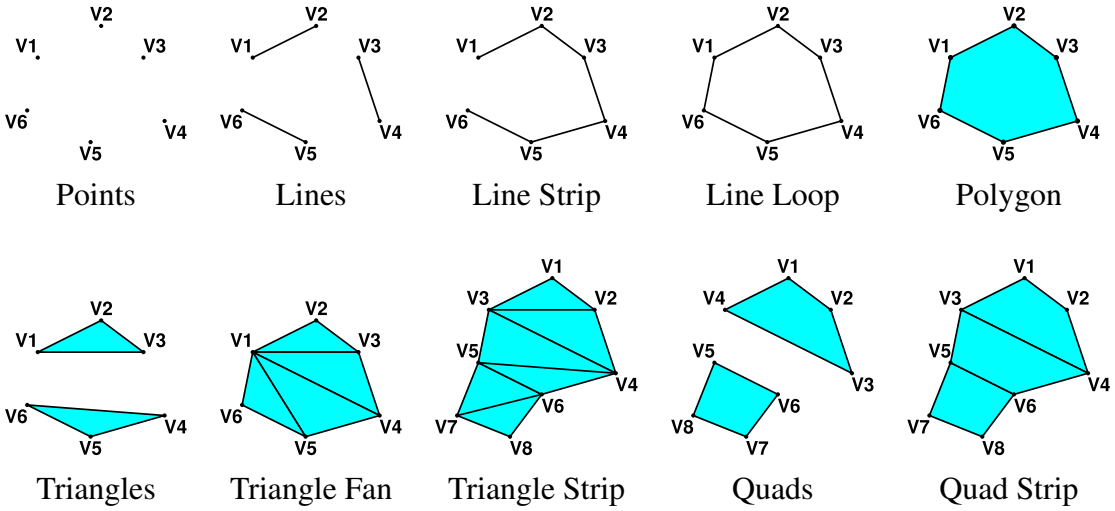


Figure 11.2 *Types de primitives (identique à OpenGL). Points, Lines et Triangles sont les plus utilisés. Les autres servent à optimiser le nombre d'indices à transmettre.*

11.2.1.4 Shader

Le dernier type de ressource concerne les shaders. Un shader est un petit programme qui utilise les données provenant des textures, des attributs des sommets, ainsi que d'autres paramètres spécifiques à l'objet à afficher (matrices de transformations, propriétés de la surface, ...) pour calculer son apparence visuelle finale. Ce calcul se fait en deux étapes :

vertex shader : utilise les attributs de chaque sommet pour en calculer sa projection sur l'écran ainsi que les valeurs à interpoler sur les polygones.

pixel shader : utilise les valeurs calculées par le vertex shader ainsi que les textures pour produire la couleur de l'objet à chaque pixel. Cette couleur peut contenir un coefficient de transparence, et sera dans ce cas combinée avec la couleur des autres objets de la scène pour obtenir la couleur du pixel final.

Il existe plusieurs langages permettant de spécifier ces shaders, notamment Cg [110] et GLSL [153], deux langages proches de C développés par NVIDIA et 3DLabs respectivement, ainsi que d'autres langages assembleurs plus bas niveau. Pour ne pas dépendre d'un seul langage, il est possible de spécifier quel est celui qu'on utilise via la variable *language*. En revanche l'implantation du moteur de rendu ou le driver de la carte graphique peut ne supporter que certains langages. Par exemple, l'implantation actuelle de FlowVR Render n'implante que les shaders Cg, principalement du au fait que des drivers supportant OpenGL 2.0 et GLSL ne sont disponibles que depuis très récemment sous Linux.

Les données d'un shader correspondent uniquement à son code source. Celui-ci est en général très cours (quelques lignes). Il ne semble donc pas très utile de le placer dans

une ressource externe aux primitives. Cependant, un shader implique un coût important de compilation sur la carte graphique, et il est aussi bénéfique de limiter les changements de shaders entre les différents objets de la scène. De ce fait il est important qu'un même shader puisse être explicitement partagé par plusieurs primitives.

11.2.2 Primitive

Une primitive rassemble toutes les données utilisées pour afficher un groupe de polygones. Ces données proviennent de ressources externes spécifiées par leurs identifiants, comme le vertex et pixel shader (*vshader* et *pshader*), l'index buffer (*ibuffer*), ainsi que les textures et les attributs des sommets (*textures* et *vbuffers*). Ces deux dernières données sont reliées à une variable utilisée par les shaders en spécifiant le nom de cette variable telle qu'elle apparaît dans le code source du shader. Des paramètres additionnels utilisés par les shaders ou par OpenGL pour régler par exemple la transparence, les tests de profondeurs ou de stencil, sont spécifiés dans la variable *params*.

Par défaut, les primitives de la scène sont considérées comme non ordonnées et le module de rendu est libre de les afficher dans n'importe quelle ordre. Toutefois pour certaines primitives, notamment utilisant de la transparence ou implantant un mécanisme particulier comme des ombres stencil, il est nécessaire d'imposer un ordre particulier. Pour cela on utilise la valeur de *order* comme une clé de tri. Les primitives sont triées par ordre croissant avant l'affichage. Les primitives utilisant la même valeur *order*, comme c'est le cas si la valeur par défaut 0 n'est pas modifiée, sont affichées dans un ordre non spécifié. Ce système permet de définir un ordre de précédance entre les primitives, sans toutefois connaître précisément les primitives envoyées par les différents viewers de l'application. Par exemple la plupart des applications actuelles utilisent par convention -10 pour le fond de la scène (*skybox*), 0 pour les objets courants, 10 pour les objets transparents, et 20 pour les éléments d'interface utilisateur (*overlays*).

Un paramètre particulier *transform* correspond à la matrice de transformation de l'objet dans la scène. Bien que cette information puisse être spécifiée comme un paramètre normal des shaders, il est souvent utile de le traiter de manière particulière. Dans le cas ou le shader applique cette matrice aux coordonnées des sommets, elle permet de calculer la boite englobante de la primitive dans la scène en appliquant la matrice *transform* à la boite englobante des attributs des sommets (section 11.2.1.2). Dans les autres cas, c'est-à-dire si le shader utilisé effectue d'autres transformations, alors le viewer doit spécifier la boite englobante de la primitive dans la variable *bbox*. Ce mécanisme permet de n'avoir à spécifier explicitement la boite englobante des primitives que rarement, même quand on change la matrice de transformation.

Enfin, une dernière information facultative *name* spécifie un nom textuel pour la primitive. Ce nom n'est pas obligatoirement unique et sert surtout pour aider l'utilisateur à reconnaître les primitives dans les messages d'erreurs ou visuellement dans un mode d'affichage de contrôle. Il peut aussi être utilisé par des filtres comme la capture de la

scène pour produire des fichiers X3D par exemple.

Un type de paramètres particuliers dans la scène concerne la gestion de la caméra. Pour les contrôler on utilise une primitive spéciale dont l'identifiant est *ID_CAMERA*. Cette primitive contient la matrice de transformation de la caméra, ainsi que d'autres paramètres tels que la focale. Si aucun viewer ne spécifie ces paramètres, les renderers utilisent les boites englobantes des primitives pour positionner la caméra de manière à voir toute la scène. L'utilisateur peut alors utiliser la souris ou le clavier pour se déplacer dans la scène. Ce contrôle par défaut de la caméra est désactivé dès qu'un viewer spécifie les paramètres de la caméra, ce qui permet d'implanter d'autres modes de déplacement (utilisant des traqueurs 3D, prenant en compte la topologie du terrain virtuel, etc).

11.3 Protocole de communication

La description des primitives graphiques de la scène doit être transmise depuis les viewers, responsables des différents objets de la scène, jusqu'aux renderers, qui vont chacun calculer une partie de l'image finale. Cette transmission doit utiliser un protocole efficace en bande passante, étant donné que les viewers et les renderers sont souvent placés sur des machines différentes, et doit en même temps être facile à traiter par les filtres intermédiaires qui vont être responsables du filtrage et de la combinaison de ces données. Etant donné qu'une partie souvent importante des données de la scène sont identiques d'une image à l'autre il semble bénéfique de décrire les changements dans la scène plutôt que toute la scène à chaque image. En relation avec les données décrites dans les sections précédentes, il y a plusieurs changements possibles :

Ajout d'une primitive, potentiellement en recopiant les données à partir d'une primitive existante.

Suppression d'une primitive.

Modification d'un paramètre d'une primitive, en indiquant quel paramètre, sa nouvelle valeur, ainsi que le nom associé (pour les valeurs de *vbuffers*, *textures*, et *params*).

Ajout d'une ressource, spécifiant le type de ressource, les paramètres associés ainsi que les données. Ceci peut correspondre à la création d'une nouvelle ressource ou à la mise à jour des données d'une ressource existante.

Suppression d'une ressource.

Modification d'une ressource, permettant de n'envoyer qu'une partie des données d'une ressource, dans le cas ou le reste est statique.

A chaque itération, les viewers envoient un message contenant l'ensemble des modifications à appliquer pour l'image suivante. Chacune de ces modifications est décrite dans un morceau (*chunk*) de ce message. Ces chunks sont mis bout à bout et suivent le format décrit dans la figure 11.3.

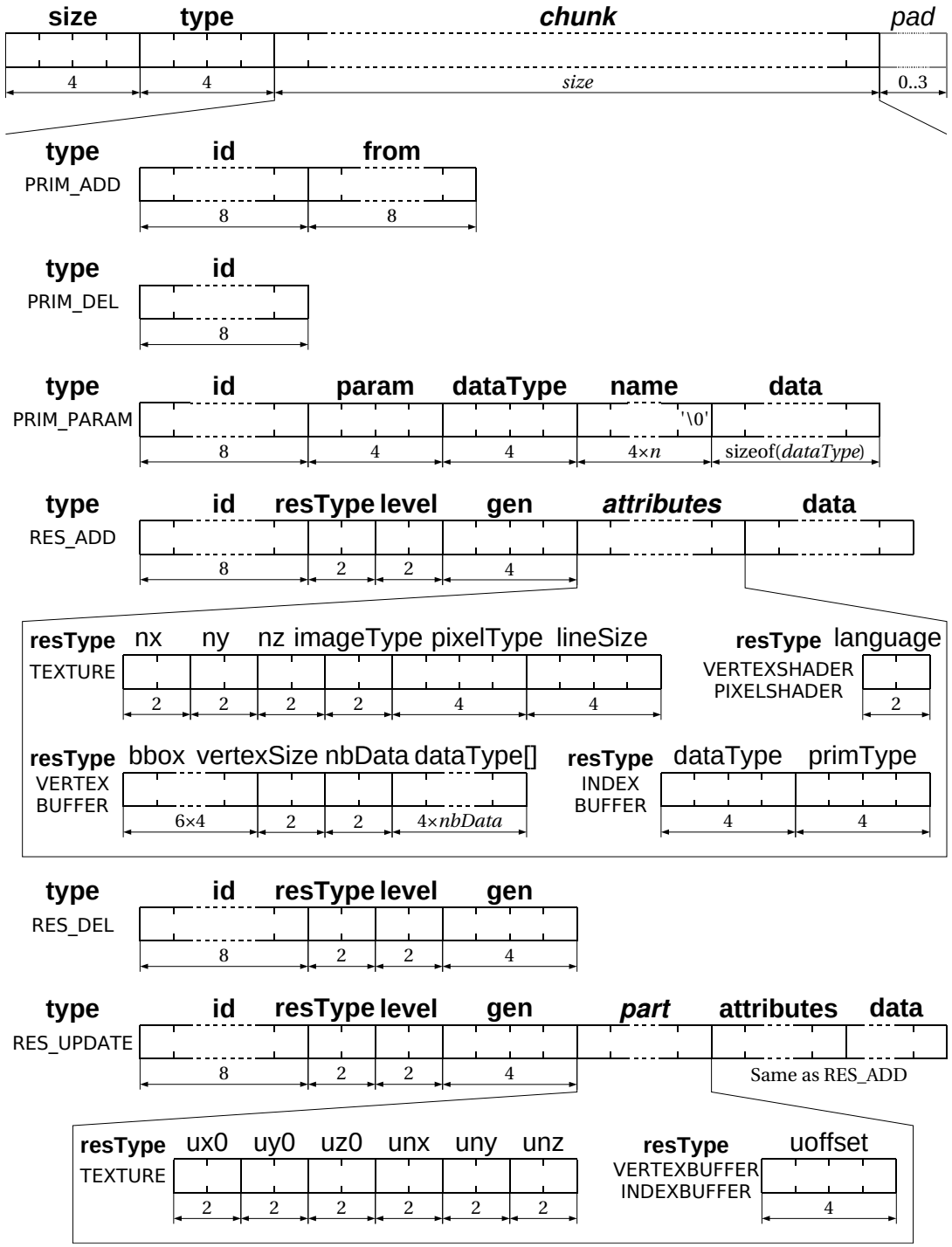
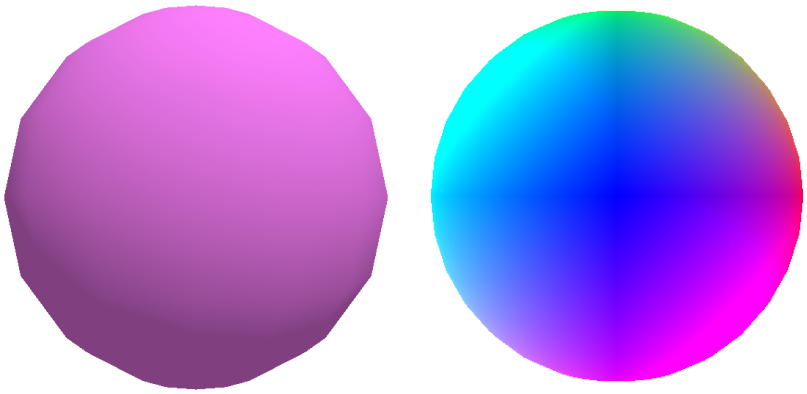


Figure 11.3 *Format des chunks contenant les mises à jours de la scène 3D. Les données correspondant aux attributs des ressources et des primitives (figure 11.1). Level est prévu pour gérer les niveaux de détails.*



(a) (b)

Figure 11.4 *Sphère affichée en utilisant (a) un shader de type Phong ou (b) un shader visualisant les composantes du vecteur normal.*

Comme il n'y a que 6 types de chunks différents les messages sont très faciles à décoder et traiter. De plus, il suffit de mettre bout à bout les différents messages pour combiner plusieurs flux. Enfin, laisser les primitives d'un viewer tel quel équivaut à envoyer un message vide, ce qui permet de désynchroniser la vitesse d'affichage de certains viewers plus lents. En effet il suffit de récupérer à chaque image les messages disponibles, et utiliser un message vide pour le cas ou il n'y a pas de nouveau message.

11.4 API

Un module peut tout à fait générer les messages de description de la scène en utilisant le format décrit dans la section précédente, mais cela l'oblige à travailler à un niveau très bas. Pour permettre de programmer les viewers plus facilement, une classe *ChunkWriter* est utilisée pour construire ces messages. Elle offre une API procédurale permettant de créer chacun des types de chunks et ensuite d'envoyer le message résultant sur un port FlowVR. Des fonctions de plus haut niveau très souvent utilisées permettent d'envoyer des modèles 3D de base (cube, sphère, cylindre), ou encore de charger à partir de fichiers les textures et les shaders.

11.5 Exemple

Voici un exemple d'un viewer simple envoyant une sphère et la faisant tournée sur elle-même. Le résultat est visible sur la figure 11.4. Cet exemple utilise *ChunkWriter* pour envoyer le modèle 3D de la sphère et un shader Cg pour implanter un calcul de lumière de type Phong (i.e. interpolation de vecteur normal et calcul de la lumière à chaque pixel). En

changeant le shader il est possible de changer complètement l'apparence visuelle, comme le montre la deuxième sphère ou des couleurs visualisent le vecteur normal de la surface.

Code du viewer

```
// Basic FlowVR Render example viewer module
// Simple sphere
#include <flowvr/module.h>
#include <flowvr/render/chunkwriter.h>
#include <stdlib.h>
using namespace flowvr::render;

int main(int argc, char** argv)
{
  // FlowVR Render output port
  SceneOutputPort pOut("scene");

  // Initialize FlowVR
  std::vector<flowvr::Port*> ports;
  ports.push_back(&pOut);
  flowvr::ModuleAPI* module = flowvr::initModule(ports);
  if (module == NULL) return 1;

  // Helper class to construct primitives update chunks
  ChunkWriter scene;

  // Get IDs for all our primitives and resources
  ID idPrim = module->generateID();
  ID idVB = module->generateID();
  ID idIB = module->generateID();
  ID idVS = module->generateID();
  ID idPS = module->generateID();

  // Create vertex+index buffers for a sphere
  scene.addMeshSphere(idVB, idIB);

  // Load custom shaders
  scene.loadVertexShader(idVS, "shaders/sphere_v.cg");
  scene.loadPixelShader(idPS, "shaders/sphere_p.cg");

  // Create a new primitive
  scene.addPrimitive(idPrim, "Sphere");
```

```
  // Set shaders
  scene.addParamID(idPrim, VSHADER,"",idVS);
  scene.addParamID(idPrim, PSHADER,"",idPS);

  // Set vertex buffers
  scene.addParamID(idPrim, VBUFFER_ID, "position", idVB);
  scene.addParamID(idPrim, VBUFFER_ID, "normal", idVB);
  scene.addParam(idPrim, VBUFFER_NUMDATA, "normal", int(1));

  // Set index buffer
  scene.addParamID(idPrim, IBUFFER_ID, "", idIB);

  // Set shaders parameters
  scene.addParam(idPrim, PARAMVSHADER, "color", Vec4f(1,0.5,1,1));
  Vec3f light(1,3,2); light.normalize();
  scene.addParam(idPrim, PARAMPSHADER, "lightdir", light);

  // Send initial scene
  scene.put(&pOut);

  // Main FlowVR loop. Contains the animations of the scene
  int it=0;
  while (module->wait())
  {
    // Update scene
    scene.rotatePrimitive(idPrim, it);
    // Send message
    scene.put(&pOut);
    ++it;
  }

  module->close();
  return 0;
}
```

Schémas de rendu parallèle 12

Ce chapitre présente la conception du graphe de flux de données reliant les modules *viewers* produisant la description de la scène 3D et les modules *renderers* en charge du rendu, au travers de plusieurs cas typiques de parallélisation de ces modules. La plupart de ces schémas existent déjà dans d'autres systèmes (section 4.1 page 28). Nous nous concentrerons alors sur les éléments nécessaire à FlowVR Render pour les implanter.

12.1 Exemple

Pour illustrer notre propos nous utiliserons l'exemple des applications présentées dans la section 8.4 d'interaction entre des simulations physiques et une reconstruction 3D de l'utilisateur dans un environnement virtuel utilisant la plateforme GrImage. Plusieurs viewers sont utilisés :

Sky : texture cubique contenant l'image du ciel et du paysage lointain.

Terrain : génère un terrain à partir d'une matrice de hauteur et d'une texture. Calcule aussi les mouvements de la caméra lorsqu'il est relié aux boutons d'un Pad.

Model : modèle 3D calculé par la reconstruction multi-caméras.

Octree : cubes constituant l'octree calculé par le module de sculpture.

Fluid : résultats de la simulation de fluide 2D.

Hair : résultats de la simulation de cheveux.

Rigid : ensemble d'objets rigides.

Video : images provenant d'une des caméras envoyées sous forme de texture.

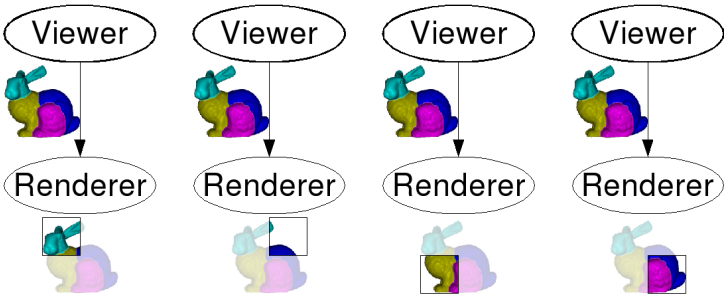


Figure 12.1 *Duplication d'un viewer sur chaque machine. Le rendu se fait localement en fonction de la position de chaque vidéo-projecteur.*

12.2 Duplication

Lorsqu'un module viewer est instancié sur chaque machine et produit la même description de scène il est alors dit *répliqué*. Cela correspond à une stratégie de parallélisation en amont dans le graphe de l'application (diffusion des données d'entrée par exemple). Dans ce cas il suffit de connecter localement chaque viewer au renderer de la machine, qui utilisera la configuration du projecteur local pour calculer une partie de l'image, comme présenté sur la figure 12.1. Cette approche est identique à celle utilisée par Net Juggler (chapitre 5).

Comme les données de la scène ne passent pas par le réseau ce schéma a une très bonne extensibilité en terme de dynamicité de la scène. En revanche la taille totale de la scène est limitée par les capacités de rendu et de mémoire de chaque machine.

Le modèle de FlowVR Render n'est ici pas indispensable, les viewers pourraient directement dialoguer avec le driver de la carte graphique. Toutefois la présence du modèle permet d'ajouter de manière transparente des éléments qui peuvent utiliser voir modifier les données de la scène. Il est par exemple facile d'ajouter un filtre récupérant des informations statistiques sur la scène (nombre de polygones, images par seconde, etc), ou modifiant certains aspects visuels comme l'utilisation de rendu proche du dessin [69, 116, 117], ou séparant les étages d'un bâtiment [125].

Dans les applications GrImage, ce schéma est utilisé pour le viewer **Sky** étant donné qu'il envoi de grandes textures (25 Mo). Comme il n'y a aucune animation cette duplication ne nécessite aucune synchronisation. Le viewer **Video** est aussi répliqué car il génère une quantité de données importante (une caméra en 640×480 à 30 Hz génère 27 Mo/s). Pour transmettre un maximum d'images un mécanisme de compression est utilisé et les viewers **Video** décodent localement les textures (section 13.2.1 page 127).

12.3 Répartition des données vers plusieurs noeuds de rendu (*sort-first*)

L'ensemble des approches présentées dans cette section sont classifiées dans les techniques de rendu parallèle *sort-first* [118], car la diffusion des données se fait *avant* la phase de rendu elle-même.

Ce deuxième cas classique de rendu parallèle concerne l'envoi via le réseau des primitives graphiques. Le module viewer est localisé sur une machine, et les données produites doivent être transmises à toutes les machines de rendu.

En fonction de la taille de ces données et du nombre de machines plusieurs approches sont possibles. Il est par exemple possible d'envoyer directement à chaque machine toutes les données (*flat broadcast*). La bande passante utilisée par la machine émettrice devient vite le goulot d'étranglement, étant donnée que son utilisation dépend du produit de la taille des données par le nombre de machines de rendu.

Une autre approche consiste à envoyer les données de manière récursive par un arbre de diffusion. Chaque machine retransmet les données à n autres machines jusqu'à ce que toutes les machines obtiennent les données (n*-ary broadcast tree*). Avec ce schéma la bande passante utilisée sur chaque machine ne dépend plus du nombre total de machines, en revanche la latence de la transmission augmente avec le logarithme du nombre de machines. Quand la plupart des objets de la scène sont statiques, ce schéma est souvent le plus efficace.

Une optimisation importante des approches précédentes consiste à n'envoyer à chaque machine que les données qui sont visibles dans la partie d'image affichée par la machine (*frustum culling*). Ce filtrage peut s'effectuer à plusieurs niveaux :

- au niveau des messages : un message de description de scène est transmis à une machine si au moins l'un des chunks qu'il contient a une influence sur l'image affichée par cette machine ;
- au niveau des primitives : on supprime des messages tous les chunks qui n'ont pas d'influence sur une machine donnée ;
- à l'intérieur de chaque primitive : on supprime les polygones non visibles, ou les zones non utilisées des textures (niveaux de mipmap détaillés pour les objets lointains par exemple).

Plus on utilise un filtrage fin, plus on diminuera les données transmises sur le réseau. En revanche, la complexité en terme d'exécution comme de développement augmente considérablement. Par exemple, si on reste au niveau des messages ou des chunks on peut se contenter de regarder les "meta-informations" (boite englobante, matrice de transformation) de chaque primitive, alors qu'un filtrage plus fin requiert de décoder toutes les données. De plus, le fait qu'une donnée ne sert pas pour l'image courante ne garantit pas qu'elle ne sera pas nécessaire dans le futur, si l'utilisateur bouge la caméra par exemple. Le filtre doit alors le détecter et envoyer les données, ce qui implique qu'il doit stocker la

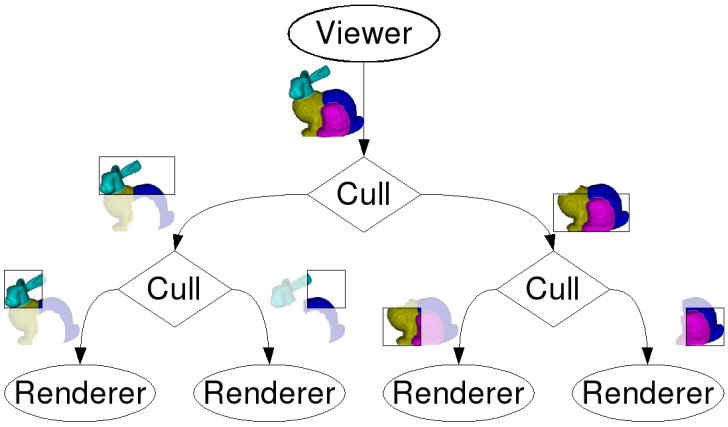


Figure 12.2 *Arbre de filtrage des données de la scène pour les diffuser sur les machines de rendu en fonction de la visibilité des différentes primitives sur chaque machine.*

dernière version de toutes les informations qu'il ne retransmet pas. Enfin, de la même façon que pour les schémas de diffusions, le filtrage peut être fait localement sur la machine émettrice, qui concentre donc tout le surcoût en terme de traitements et bande passante, ou récursivement en utilisant plusieurs niveaux de filtrage. Il est par exemple possible de filtrer les données en groupant les machines par lignes, puis de filtrer à l'intérieur de chacune de ces lignes. Ce schéma correspond à la figure 12.2. Sur cette figure, le filtrage se fait au niveau des primitives (les différentes parties du lapin).

Comme cela sera détaillé pour l'exemple de rendu volumique dans la section 13.1.3 (page 123), l'utilisation de shaders permet de diminuer le trafic nécessaire à la transmission des mises à jour de la scène. En effet, beaucoup d'animations (*keyframes*, *morphing*, *skinning*) ou d'extractions de données (conversions colorimétriques, fonctions de transfert, coordonnées de mapping) peuvent être calculées par les shaders appliqués aux sommets ou aux pixels. En conséquence il n'est souvent nécessaire de mettre à jour que les paramètres (matrices, coefficients, tables) utilisés par les shaders plutôt que tout le modèle 3D.

La plupart des viewers des applications GrImage utilisent un arbre de diffusion sans aucun filtrage des données. **Terrain** et **Rigid** ne mettent à jour que quelques matrices de transformations à chaque itération ; alors que **Model**, **Octree** et **Hair** mettent à jour le modèle 3D lui-même mais cela ne représente que quelques kilo-octets par secondes. Le dernier viewer, **Fluid**, utilise une seule texture. Un filtre de culling permettrait de gagner en performance seulement si le filtrage est effectué à l'intérieur des primitives et des ressources, ce qui n'est pas actuellement implanté de façon suffisamment générique.

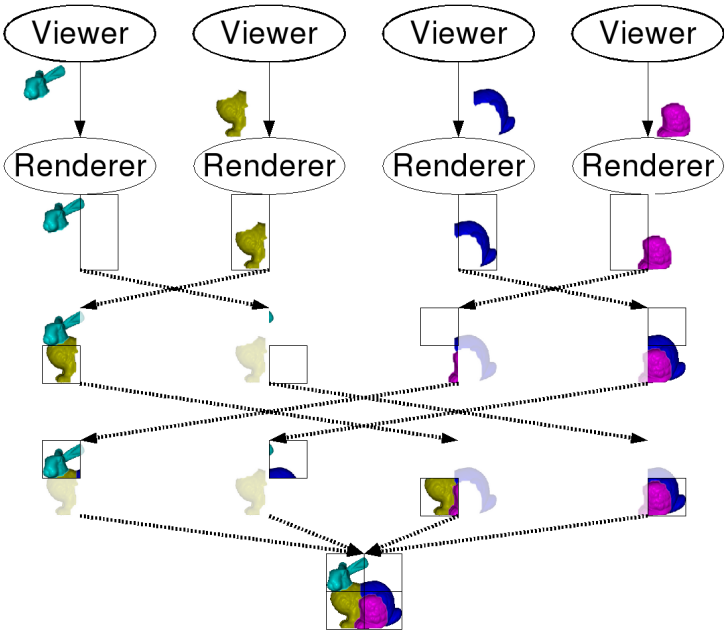


Figure 12.3 *Redistribution des pixels des images (flèches pointillées grasses).*

12.4 Rendu local et recomposition des pixels (*sort-last*)

Le dernier schéma classique de rendu parallèle concerne la recombinaison des pixels des images après le rendu. Cette méthode est surtout utilisée quand on souhaite accélérer le rendu d'une image de faible résolution, étant donné que le surcoût dépend directement de celle-ci. Elle est primordiale pour pouvoir visualiser les très grandes scènes, dont la taille dépasse la quantité de mémoire de chaque machine. Dans ce schéma plusieurs viewers, instanciés sur des machines distinctes, calculent chacun une partie de la scène, qui est ensuite rendue par un renderer local. Une fois ce rendu effectué, les images doivent être recombinées pour obtenir l'image de la scène complète. Plusieurs algorithmes existent, l'un des plus efficace est le *Binary Swap* [108]. C'est un algorithme en plusieurs étapes. A chaque étape les machines échangent deux-à-deux une partie de l'image, jusqu'à ce que chaque machine contienne une zone de l'image finale (figure 12.3). Ces zones peuvent ensuite être envoyées à une certaine machine ou affichées localement.

Cette approche n'est pas directement du ressort de FlowVR Render, étant donné qu'elle est basée sur un transport de pixels et non de primitives graphiques. En revanche, FlowVR Render permet de le faire de manière transparente pour les applications. Il est même possible d'utiliser un schéma de redistribution des primitives graphiques en amont des renderers, de manière similaire à la figure 12.2. Dans ce cas le filtrage des données

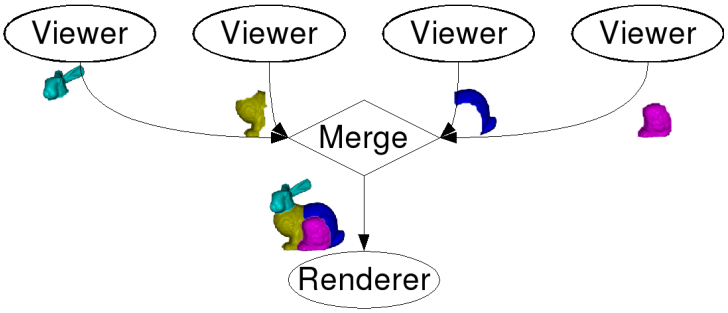


Figure 12.4 *Combiner plusieurs viewers revient à concaténer les messages produits par chacun d'eux.*

ne se fait pas par rapport à la zone visible sur chaque machine, mais plus pour un souci d'équilibrage de charge (i.e. envoyer approximativement le même nombre de polygones à toutes les machines).

12.5 Description parallèle utilisant plusieurs viewers

Combiner plusieurs modules viewers dans une même scène est une fonctionnalité très importante. Elle permet de distribuer les calculs de génération des primitives, par exemple dans le cas d'une extraction d'iso-surface, mais aussi de modulariser la description de la scène. Cette modularité est très utile pour une application basée sur du couplage de code, par exemple pour visualiser deux simulations qui ont chacune leur module de visualisation préexistant. C'est ce qui permet de combiner tous les viewers composant l'environnement virtuel sur GrImage.

Une autre utilisation importante de cette configuration concerne l'intégration de plusieurs schémas de rendu parallèle pour chaque partie de la scène. Par exemple, certains objets générant beaucoup de données peuvent être générés localement en dupliquant les viewers associés, tandis que d'autres viewers plus coûteux en calculs ou nécessitant des données d'entrée volumineuses peuvent être répartis sur d'autres machines avec diffusion par le réseau de leurs résultats. Ensuite, quelle que soit la provenance de chaque flux de données ils sont combinés avant d'être envoyés au renderer. Du fait de l'indépendance des primitives de la scène cette recombinaison est extrêmement simple : il suffit en effet de mettre bout-à-bout les messages de chacun des flux. C'est ce que fait le filtre *Merge* de FlowVR (figure 12.4).

12.6 Asynchronisme

L'utilisation de plusieurs viewers permet de coupler facilement des objets différents dans une même scène, avec toutefois la contrainte que tous les viewers doivent se mettre à jour simultanément. En utilisant un schéma d'échantillonnage avant le *Merge* on peut autoriser des fréquences de mises à jour différentes pour chaque viewer, et entre les viewers et les renderers. Cela permet une meilleure scalabilité et réactivité du rendu du fait du relâchement des synchronisations entre les différents modules. Cette désynchronisation est très simple à implanter du fait de l'aspect unidirectionnel des communications entre les viewers et les renderers. Il peut même être poussé à l'extrême pour exécuter les viewers et les renderers séparément en utilisant un fichier pour stocker les messages.

Quasiment toutes les applications utilisant FlowVR Render se servent de ce mécanisme de désynchronisation (voir par exemple la figure 8.8 indiquant les différentes fréquences de rafraîchissement dans l'application GrImage). Cela permet par exemple de bouger le point de vue de manière fluide, sans attendre la mise à jour du viewer, ceci même dans le cas ou un seul viewer est présent. En revanche certains problèmes d'incohérences apparaissent dans le cas multi-viewers. En effet certains groupes de viewers, comme par exemple ceux implantant une extraction d'iso-surface parallèle, doivent être mis à jour de manière synchrone pour ne pas faire apparaître les frontières.

Un problème similaire se pose au niveau des différents renderers. En effet, les différents projecteurs composant une seule image du point de vue de l'utilisateur doivent souvent se rafraîchir simultanément pour donner l'impression d'un seul grand affichage (*swaplock*). En effet, si une machine est plus rapide que les autres ou à une charge de calcul plus faible elle risque de mettre à jour son affichage constamment avant les autres ce qui risque de casser la cohérence globale. En revanche si deux affichages ne sont pas accolés il est souvent bénéfique de laisser leurs rafraîchissements se faire aussi vite que possible, afin de ne pas les ralentir inutilement. Cela permet d'avoir une console de contrôle distincte de l'environnement immersif sans affecter les performances de celui-ci.

Nous devons donc introduire des contraintes de rafraîchissement synchrone ou non pour les différents viewers et renderers. Heureusement ces politiques de couplages s'implantent facilement dans le graphe de l'application FlowVR en regroupant les synchroniseurs d'échantillonnage des données des différentes connexions concernées.

Ce mécanisme permet à FlowVR Render d'être en quelque sorte l'équivalent pour la 3D du protocole X pour les interfaces 2D. En effet, un serveur X Window permet à plusieurs clients distants de mettre à jour de manière asynchrone un ensemble de primitives (fenêtres) à l'intérieur d'une même surface d'affichage (2D dans ce cas). Le système est basé sur un protocole réseau gérant un certain nombre de fenêtres et de ressources (bitmaps, polices de caractères) en utilisant des identifiants uniques pour tous les clients.

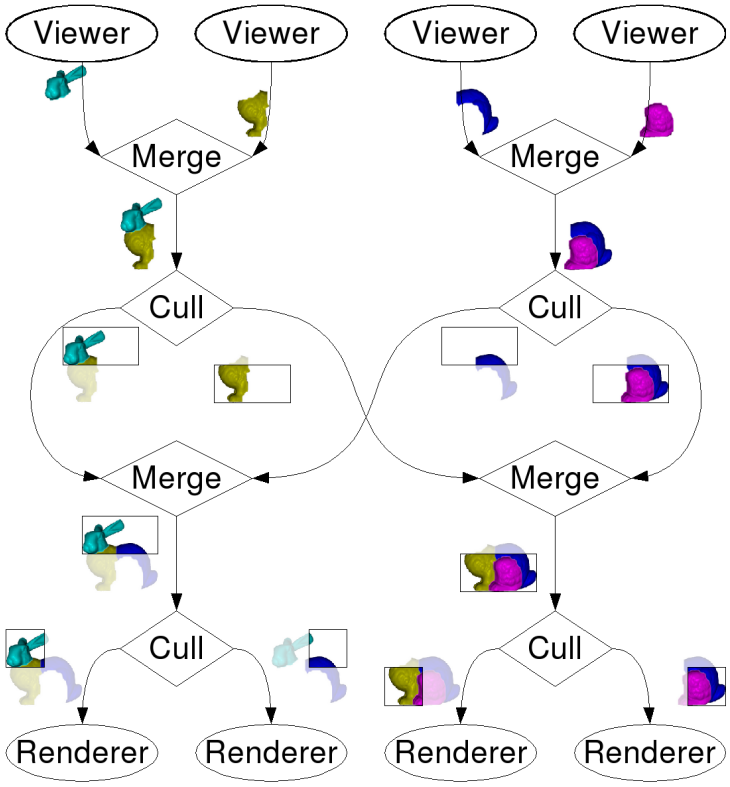


Figure 12.5 *Intercallage d'étapes de fusion et de découpage pour implanter efficacement l'envoi des données de plusieurs viewers vers plusieurs renderers.*

12.7 Passage à l'échelle

Pour une grande application avec de nombreux viewers et renderers, plusieurs schémas sont possibles pour redistribuer les données. En effet, il est possible d'effectuer une première phase de fusion (section 12.5), puis une phase de découpage (section 12.3), ou alors d'inverser l'ordre de ces deux phases. Cependant, une stratégie intermédiaire parfois plus efficace consiste à mélanger ces phases en une suite d'étapes de fusion et de découpages partiels (figure 12.5), similaire au principe du *Binary Swap* [108] utilisé en rendu *sort-last* (section 12.4). A chaque étape les primitives de chaque couple de machines de tri sont fusionnées puis redécoupées par zone. En utilisant N machines de tri, il faut $\log N$ étapes pour compléter l'algorithme. On peut noter que le cas $N = 1$ correspond à l'algorithme centralisé (fusion puis découpage). Ainsi la figure 12.5 utilise 2 machines de tri, et donc une seule étape de l'algorithme, alors qu'un cas plus complexe avec 4 machines de tri et 8 renderers est présenté sur la figure 12.6.

Pour comparer ces stratégies, il est possible d'estimer le volume des transmissions réseau en terme de messages et de bande passante. Soit V le nombre de viewers et R le nombre de renderers. La bande passante utilisée dépend de la finesse du découpage

des primitives. Ainsi des primitives très petites seront en moyenne visible sur un seul renderer, alors qu'au contraire de larges primitives devront être communiquées à tous. Ces deux extrêmes permettent de borner la bande passante utilisée.

En utilisant la stratégie de centralisation des données, à chaque image tous les viewers envoient un message à la machine de tri, qui retransmet ensuite un message à tous les renderers, soit $V + R$ messages au total. En ce qui concerne la bande passante utilisée, chaque primitive est envoyée une fois du viewer vers la machine de tri, puis elle est retransmise vers un seul renderer dans le meilleur cas, et vers tous dans le pire cas, soit un nombre d'envois compris entre 2 et $R + 1$.

La stratégie opposée de découpage puis fusion implique une connexion directe de chaque viewer vers chaque renderer, soit $V \times R$ messages. Les primitives sont envoyées directement vers les renderers concernés, sont entre 1 et R envois.

Enfin la stratégie intermédiaire nécessite $V + N \log N + R$ messages. Pour la bande passante, dans le meilleur cas une primitive a une chance sur deux d'être envoyée sur le réseau lors de chaque étape, soit un total d'envois de $2 + \frac{1}{2} \log N$ (en comptant les envois avant et après le tri), alors que dans le pire cas ce schéma est équivalent à une diffusion de chaque primitive, soit $R + N$ envois.

En conclusion, on remarque que la stratégie inspirée du *Binary Swap* permet d'éviter une centralisation des données ainsi qu'une explosion du nombre de messages envoyés sur le réseau (qui atteint le carré du nombre de modules si les viewers sont directement reliés aux renderers). En configurant le nombre de machines de tri, il est possible de contrôler l'équilibre entre la centralisation du réseau et le surcoût en terme de bande passante. En effet, ce surcoût est compris entre $\log N$ et N envois, ou N et le nombre de machines de tri.

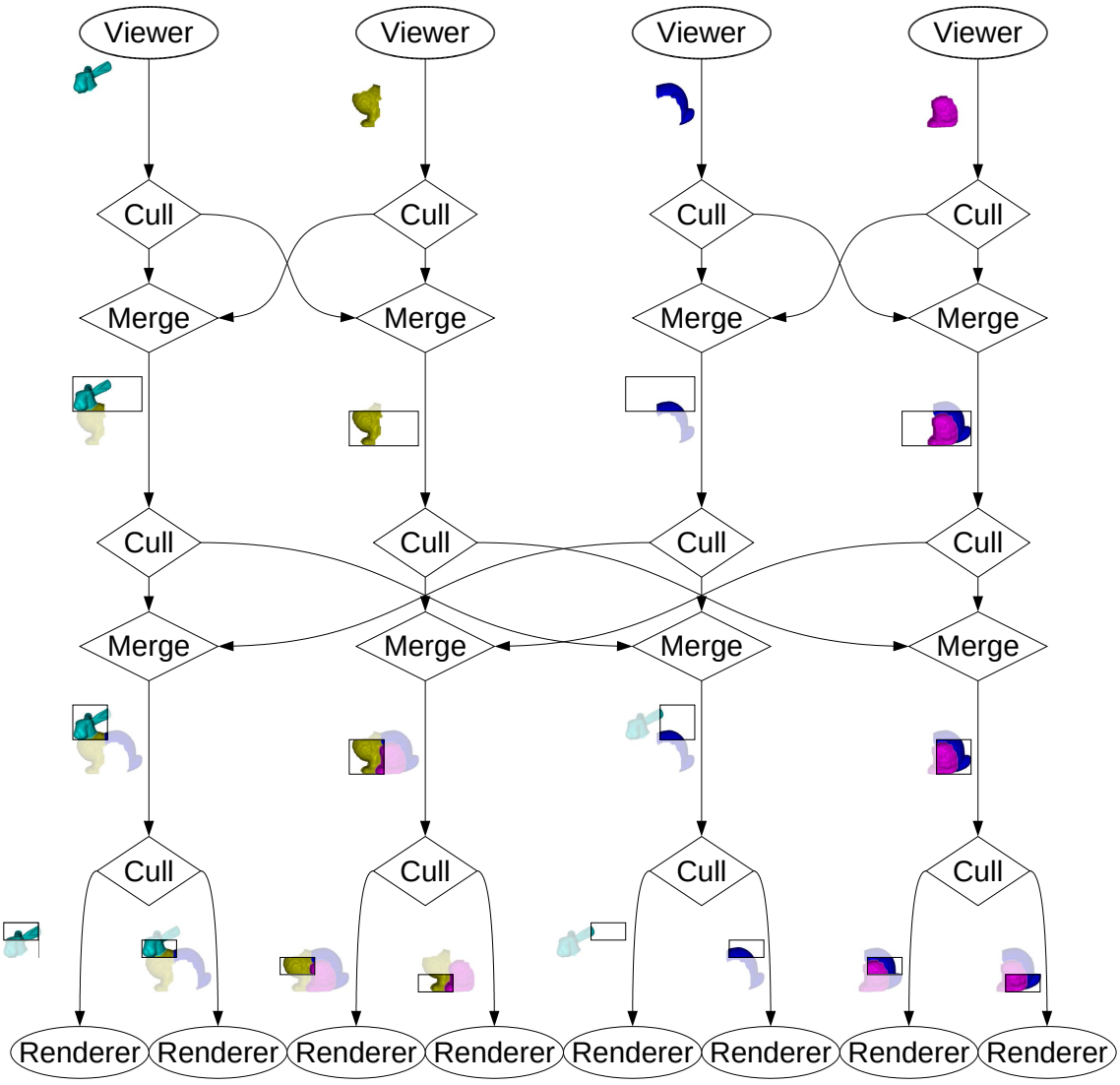


Figure 12.6 *Redistribution entre 4 viewers et 8 renderers utilisant 4 machines de tri et 2 étapes de découpage–fusion.*

Applications 13

13.1 Visualisation scientifique

L'un des premiers domaines d'application des techniques de rendu parallèle concerne la visualisation de gros volumes de données scientifiques, comme des résultats de simulations ou d'instruments de mesures. Nous avons donc expérimenté l'adéquation de FlowVR Render à ce type d'applications, en particulier en comparant avec l'approche existante utilisant Chromium [85]. Ces expérimentations sont aussi présentées dans le premier article publié sur FlowVR Render [12].

13.1.1 Couplage avec VTK

Les applications de visualisation scientifique reposent le plus souvent sur des librairies implantant un ensemble de filtres applicables sur les données pour analyser la partie désirée, ainsi qu'un environnement permettant de relier ces filtres entre-eux et générer l'affichage lui-même (section 2.4). Pour nos expérimentations nous avons choisi VTK [157] car c'est une librairie open-source avec une communauté d'utilisateurs active et offrant les fonctionnalités requises (en particulier le support de la parallélisation du filtrage des données [3]).

Nous avons développé une librairie qui remplace les classes de rendu de VTK de manière transparente. Le design de VTK fait qu'il y a peu de telles classes : une par type de données à afficher entre les images, les modèles 3D (composés de points, de lignes et de triangles) ou les volumes. L'implantation de ces classes fournie avec VTK utilise des commandes OpenGL en mode immédiat ou utilisant des *display lists*. Elle est relative-

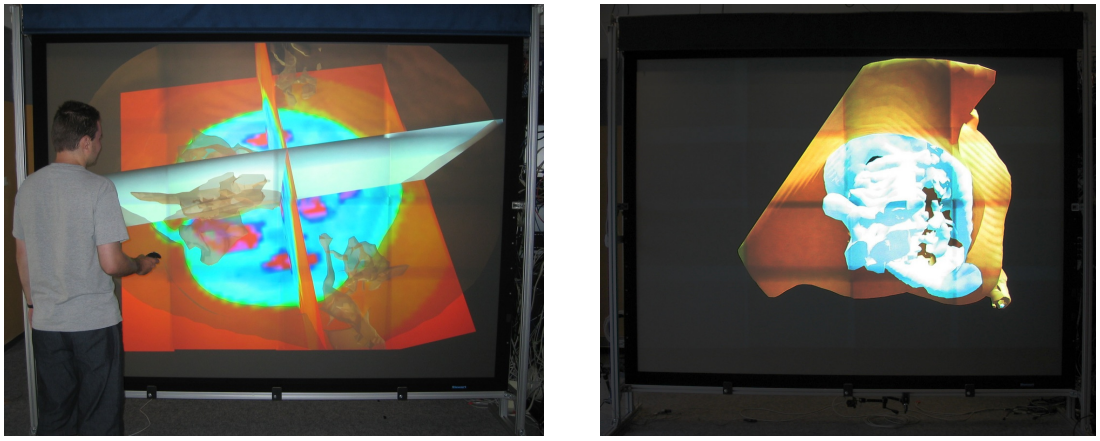


Figure 13.1 *Une des applications d'exemple de VTK affichée sur le mur d'image grâce à FlowVR Render.*

ment complexe pour gérer toutes les combinaisons possibles de données présentes sur les sommets, ainsi que pour supporter les anciens drivers OpenGL. La nouvelle implantation utilisant FlowVR Render est plus simple du fait que bien souvent il suffit d'encapsuler les données brutes dans des ressources FlowVR Render et sélectionner le bon shader. Des applications VTK existantes peuvent alors être affichées sur le mur d'image sans nécessiter de modification (figure 13.1).

13.1.2 Extraction de données

Pour comparer les performances de FlowVR Render par rapport à Chromium nous avons utilisé une application parallèle d'extraction d'iso-surface. Les données utilisées proviennent d'une simulation de fluide 3D de $132 \times 132 \times 66$ cellules pour 900 pas de temps (figure 13.2). L'application extrait et affiche de manière interactive une iso-surface (contenant approximativement 100000 triangles) pour chaque pas de temps. Cette extraction est parallélisée en découpant les données en blocs, chacun assigné à un viewer.

La figure 13.3 présente les mesures de performance de cette application utilisant soit Chromium soit FlowVR Render en fonction de la taille du mur d'image (nombre de renderers) ainsi que du nombre de viewers calculant l'iso-surface. FlowVR Render est plus performant que Chromium et offre un meilleur passage à l'échelle, à la fois en terme de nombre de renderers comme de viewers. FlowVR Render atteint 12 images par secondes avec 16 viewers et 16 renderers pour afficher le résultat sur le mur d'images 4×4 . Les performances de Chromium sont certainement affectées par le surcoût important lié aux opérations de culling et de fusion sur les flux d'opérations OpenGL.

L'utilisation des shaders pour calculer l'éclairage à chaque pixel permet d'augmenter significativement la qualité du rendu visuel. Par rapport au calcul OpenGL classique évalué uniquement aux sommets, les shaders permettent de cacher le découpage en triangle

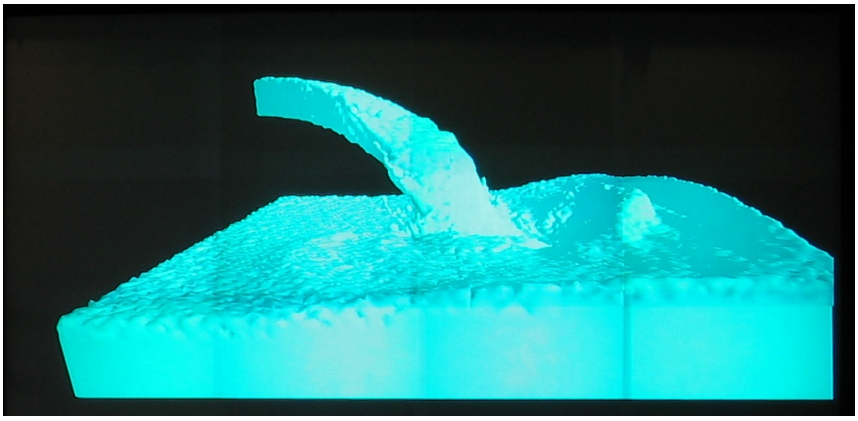


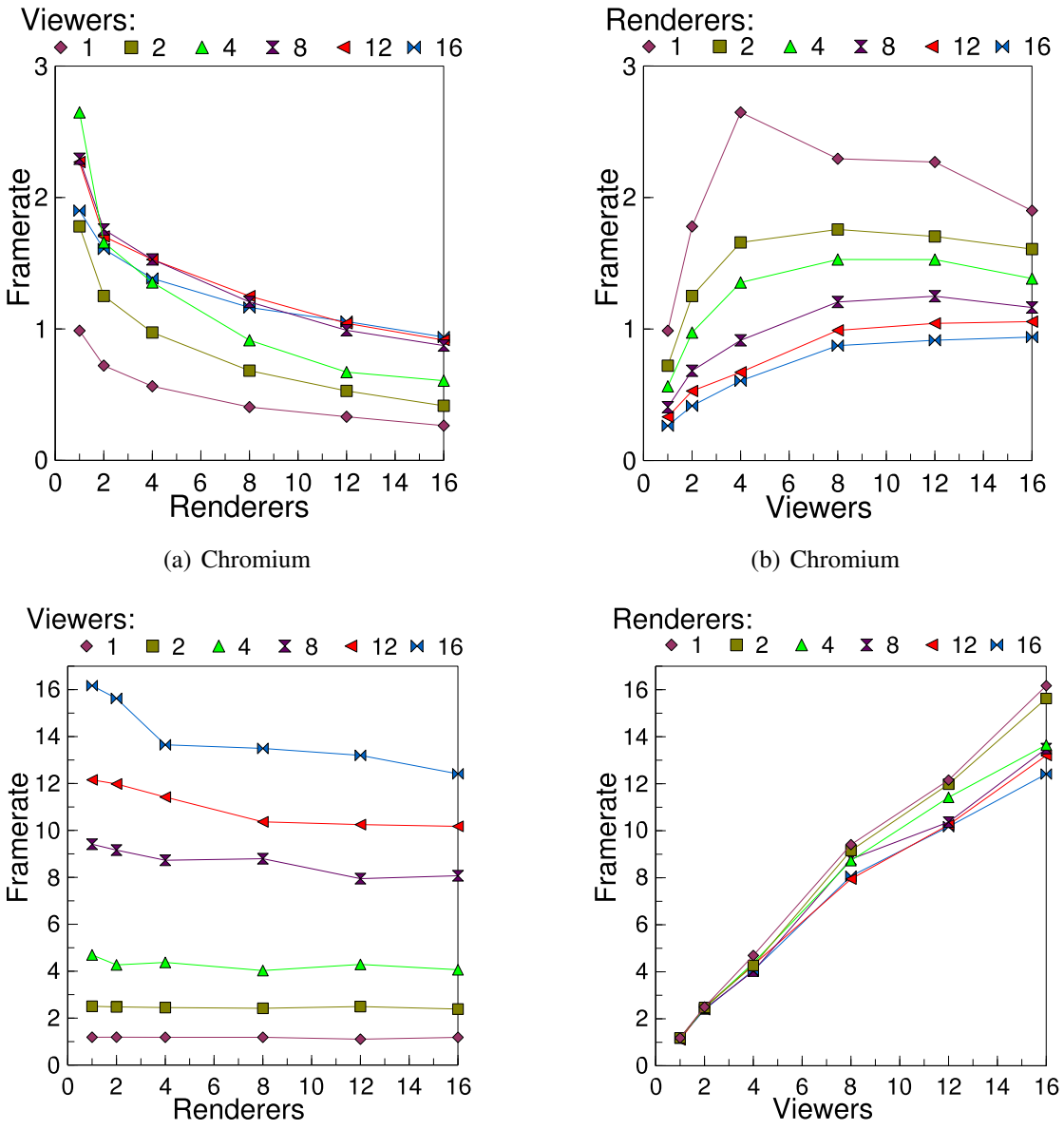
Figure 13.2 *Iso-surface extraite d'un pas de temps d'une simulation de fluide 3D.*

de la surface pour obtenir un rendu beaucoup plus lisse. Une vidéo présentant le résultat est disponible : `http://www-id.imag.fr/~allardj/these/vtk-fluid3d.avi`.

13.1.3 Rendu volumique

Le deuxième test utilise de manière plus avancée les shaders afin de démontrer leurs avantages dans le cadre du rendu parallèle. Nous étudions donc le cas d'une application de rendu volumique en rendu parallèle *sort-first*. Notez que pour le rendu volumique les méthodes *sort-last* se sont en général montrées plus efficaces [181]. Nous voulons ainsi démontrer que l'utilisation de shaders peut augmenter de manière significative les performances des algorithmes *sort-first* pour les raisons suivantes :

- Grâce à la nature massivement parallèle des GPU actuels, les shaders ont accès à beaucoup plus de ressources que les programmes classiques, à la fois en terme de bande passante mémoire et en puissance de calcul [100].
- Un shader peut appliquer les fonctions de transfert permettant de convertir les données volumiques brutes en couleur et opacité finale. Dans le cas ou seul ces fonctions sont modifiées, ceci permet de n'envoyer qu'une seul fois les données volumiques et ensuite uniquement mettre à jour la fonction de transfert lorsque nécessaire. Même dans le cas ou les données sont dynamiques au cours du temps cette approche est intéressante car ces données peuvent être jusqu'à quatre fois plus petites que les couleurs et opacités finales (une valeur par voxel contre quatre pour les couleurs).
- En utilisant des fonctions de transfert *pré-intégrées* [53] et un pas d'échantillonnage adaptatif [151], un shader peu créer une image de très haute qualité tout en nécessitant moins d'accès aux données volumiques, permettant ainsi d'utiliser de plus gros volumes de données.



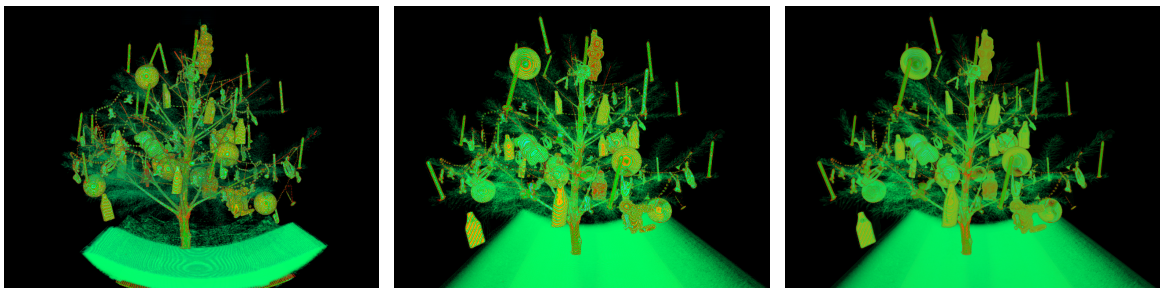
(a) Chromium

(b) Chromium

(c) FlowVR Render

(d) FlowVR Render

Figure 13.3 *Extraction parallèle d'iso-surface avec un rendu sort-first, utilisant Chromium (a)-(b) ou FlowVR Render (c)-(d). Les courbes de gauche présentent l'évolution en augmentant le nombre de renderers, alors qu'à droite sont présentées les courbes en fonction du nombre de viewers.*



(a) Plans texturés 2D VTK (b) Shader de raycasting (c) Raycasting de raycasting avec fonction de transfert pré-intégrée

Figure 13.4 *Rendu volumique d'un sapin de noël utilisant différentes méthodes de calcul (*512 *pas d'échantillonnage par pixel).*

Méthode	Pas d'échantillonage	1 écran 1024×768	Mur d'image 4×4 4096×3072	Mur d'image 4×4 2048×1536	Mur d'image 4×4 1024×768
Plans texturés 2D VTK	512	0.18			
Shader raycasting	512	1.16	2.25	5.40	8.21
Shader raycasting pré-intégré	512	1.10	2.04	4.97	7.70
Shader raycasting pré-intégré	200	2.79	5.14	12.44	19.11

TAB. 13.1 *Performances en nombre d'images par seconde du rendu volumique parallèle avec des données statiques en* $512 \times 512 \times 512$.

Par défaut VTK utilise une approche basée sur un ensemble de plans 2D texturés pour le rendu volumique matériel. Nous avons implanté une nouvelle classe utilisant des shaders. Un pixel-shader est utilisé pour parcourir un rayon au travers du volume (*raycasting*) en accumulant la couleur et l'opacité via un calcul de transparence par atténuation, par additivité ou par valeur maximale. Comme cette accumulation est effectuée dans des registres temporaires du shader et non via le buffer d'image, il garde une précision en 32-bits et économise la bande passante normalement nécessaire pour écrire et relire les valeurs du buffer d'image utilisée par les approches traditionnelles multi-passes.

Pour implanter une fonction de transfert pré-intégrée nous utilisons une texture 2D qui, étant donnée la valeur de densité précédente et courante sur le rayon, stocke la couleur et l'opacité obtenue en intégrant toutes les densités intermédiaires dans la fonction de transfert originale. Ce calcul augmente sensiblement la qualité visuelle du rendu, surtout pour le cas des fonctions de transfert utilisant des hautes fréquences, et permet aussi d'utiliser de plus grands pas d'échantillonnage pour les données volumiques qui ne comportent pas de variation abrupte.

Comme cette application n'est limitée que par le *fill-rate* de la carte graphique (nombre de pixels que la carte graphique peut traiter en une seconde), nous avons utilisé pour l'envoi des données un simple schéma de diffusion où tout est envoyé à tous les renderers.

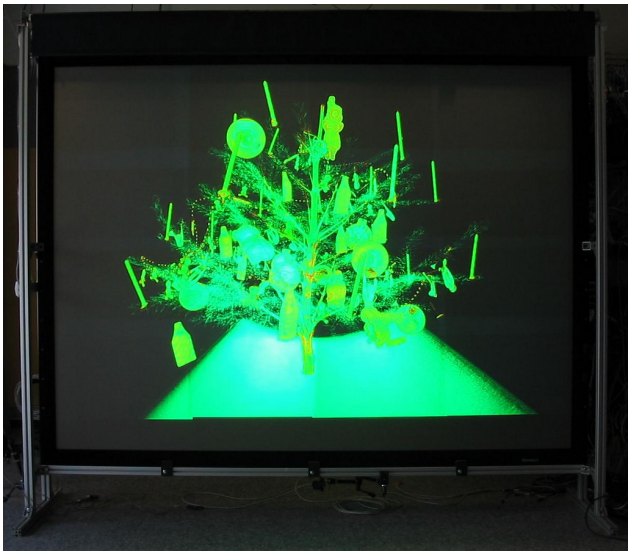


Figure 13.5 *Rendu volumique sur le mur d'image.*

La figure 13.4 présente les résultats utilisant la méthode originale de VTK ainsi que la nouvelle approche utilisant FlowVR Render et notre shader de tracé de rayon, optionnellement utilisant une fonction de transfert pré-intégrée. Les données sont un volume de $512 \times 512 \times 512$ contenant un sapin de noël [94]. La table 13.1 rassemble les mesures de performance de cette application. Le rendu parallèle sur le mur d'image n'introduit pas de surcoût significatif par rapport au rendu sur une seule machine comme les données transférées sont relativement petites (la position de la caméra ainsi que la fonction de transfert). Nous obtenons même de meilleures performances sur le mur d'image du fait d'une plus grande cohérence entre les pixels voisins qui favorise les caches texture à l'intérieur de la carte graphique.

On peut aussi noter qu'il est possible d'améliorer les performances pendant les interactions (pendant que la caméra se déplace par exemple), en réduisant la résolution de rendu en plus de la fréquence d'échantillonnage. Ceci permet d'obtenir des mouvements tout en gardant une qualité raisonnable.

13.2 Autres applications

L'application GrImage a été la raison initiale du développement de FlowVR Render. Les applications utilisant VTK décrites ci-dessus ont permis de le valider en comparant avec l'outil existant prédominant pour le rendu distribué. Depuis, des applications développées par d'autres personnes ont utilisé l'architecture de FlowVR Render. Cette section en présente quelques unes.

Figure 13.6 *Calcul de la texture du modèle reconstruit en utilisant un shader pour recombiner les images des caméras en calculant les coefficients appropriés (représentés par des couleurs sur le rendu en haut à gauche).*

13.2.1 Texturage du modèle reconstruit

Cette application est une collaboration avec Florian Geffray, Marc Lapierre et Clément Ménier.

La reconstruction 3D multi-caméras décrite dans la section 8.3 permet d'obtenir en temps réel un modèle géométrique de l'utilisateur. Toutefois il manque les informations de couleur et de texture pour obtenir une représentation réaliste. Il est possible de reprojeter les images des caméras sur le modèle 3D pour obtenir cette information. Plusieurs problèmes se posent :

- Les flux vidéos doivent être transmis aux machines de rendu. Comme la bande passante réseau et d'envoi vers la carte graphique n'est pas suffisante il faut utiliser des algorithmes de sélection d'une partie des caméras, et de compression des données.
- Certains objets cachent des objets plus loin sur certaines caméras (comme la main par rapport au corps), ce qui introduit des erreurs si on projette l'image de la main sur le corps.
- Une fois les flux à utiliser déterminés, il faut les recombiner en calculant des coefficients pour utiliser la meilleure image tout en cachant les transitions.

Nous avons implanté une première approche utilisant FlowVR Render. Le viewer existant **Video** a été adapté pour recevoir les données sous forme compressée. La compression utilisée est pour l'instant très simple et consiste à n'envoyer que les couleurs des pixels de l'objet et non du fond de l'image, ce qui supprime environ 75 % des données. Les flux d'images compressées sont ensuite diffusés à toutes les machines de rendu. Sur chacune de ces machines un viewer décompresse les images et les envoie sous forme de texture. Ensuite le viewer **Model** a été adapté pour utiliser un nouveau shader utilisant ces flux de textures ainsi que les matrices de calibration des caméras pour calculer à chaque pixel

la projection du point sur chaque image et les recombiner en fonction de l'angle entre le vecteur normal à la surface et la direction de la caméra. Ce coefficient permet de n'utiliser que les caméras qui font face à la surface. Le résultat est visible sur la figure 13.6.

Avec 6 caméras en 780×580 et le mur de 16 vidéo-projecteurs ceci permet d'envoyer quelques images par seconde. Pour augmenter ces performances et éviter de saturer le réseau pour le reste de l'application nous avons utilisé un second réseau gigabit pour toutes les communications des images compressées. Cela permet d'envoyer environ 10 images par seconde sans affecter les autres modules.

Cette première implantation a servi de prototype, plusieurs améliorations sont depuis en développement. Pour augmenter le nombre de caméras ainsi que la fréquence de rafraîchissement un module de sélection des flux les plus utiles en fonction de la position de la caméra va être ajouté. Bien que le résultat visuel est satisfaisant, les occlusions ne sont pour l'instant pas gérées. Pour le faire il faudrait utiliser un shader plus avancé, utilisant des textures de distances (*depth maps*) ou des volumes d'ombres calculés avec le *stencil buffer*.

13.2.2 Vidéo haute-définition sur mur d'image

Cette application est une collaboration avec Clément Ménier.

Plutôt qu'envoyer un ensemble de flux vidéos, un autre problème concerne l'affichage de vidéos haute-résolution. La haute résolution du mur d'image permet d'afficher des images très précises, ce qui demande beaucoup de puissance de calcul et de bande passante dans le cas d'une vidéo. Les étapes nécessaires sont :

- lecture du fichier ;
- décodage de la vidéo ;
- conversion des images du format YUV (luminance et chrominance) vers RGB ;
- affichage des images.

Clément a converti le logiciel open-source de lecture de vidéo MPlayer[1] en viewer FlowVR Render. Utiliser FlowVR Render pour l'affichage permet de bénéficier des corrections liées aux vidéo-projecteurs (matrices de calibration, masques de blending) et d'utiliser plusieurs schémas de distributions. Nous avons testé deux approches : un seul viewer décode la vidéo et le résultat est diffusé aux différents renderers (rendu *sort-first*), ou alors une *réplication* du viewer décodant la vidéo sur chacune des machines de rendu. La première stratégie est bien adaptée à la lecture de vidéos de taille similaire au format DVD (720×576), en revanche la bande passante du réseau n'est pas suffisante pour des vidéos plus haute résolution. La réplication des viewers permet de supprimer ce goulot d'étranglement. Le facteur limitant devient alors la puissance de calcul nécessaire. Pour décharger le processeur des machines nous avons utilisé un shader qui se charge du décodage des couleurs YUV vers RGB. Du fait de l'encodage de la plupart des vidéos en

[1] http://www.mplayerhq.hu/

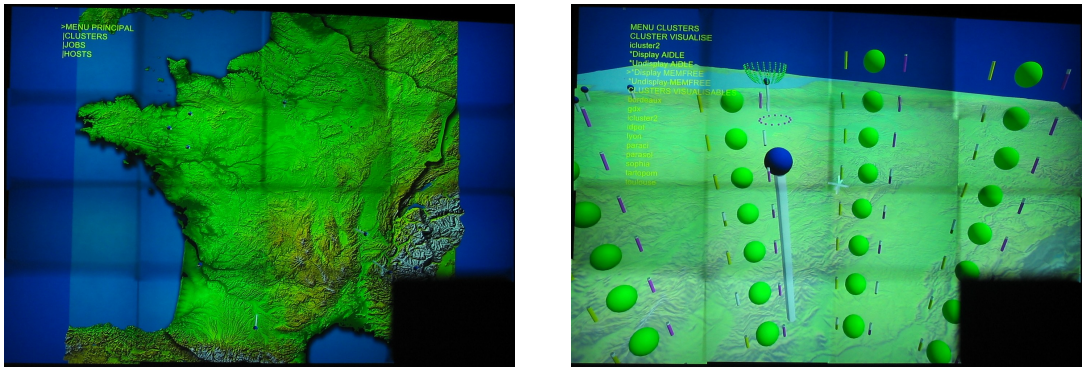


Figure 13.7 *Visualisation de l'état de la grille Grid 5000 sur le mur d'image.*

plans séparés avec un sous-échantillonnage des canaux de couleur, cela permet aussi de diminuer la taille des données à transmettre à la carte graphique (12 bits par pixels au lieu de 24). Cette architecture permet de visualiser sur le mur d'image des vidéos au format HDTV (1920×1080 à 60 Hz).

Cette application offre une solution simple à un problème concret mais reste assez limitée. En effet, toutes les données sont décodées sur chaque machine et envoyées sur chaque carte graphique. Pour atteindre la résolution du mur d'image (4000×3000) il faudrait paralléliser le décodage de la vidéo.

13.2.3 Visualisation interactive de l'état d'une grille

Cette application est développée dans le cadre du stage d'Oscar Ayoun et Julien Bouvier en collaboration avec Olivier Richard.

Le but est de visualiser en 3D sur un ordinateur portable ou un environnement immersif l'état de la grille Grid 5000 (statistiques d'utilisation des ressources, tâches lancées) pour démontrer le déploiement des applications en temps réel. Pour cela un ensemble de modules FlowVR est déployé sur la grille et recueille les informations nécessaires au niveau des machines, des switchs réseaux ou des logiciels de monitoring et d'allocation des tâches (*Ganglia*[2] et *OAR*[3]). Ensuite des modules viewers placés sur la grappe de visualisation les convertissent en représentations visuelles. D'autres viewers ajoutent des informations de contextes (carte de France). L'utilisateur peut alors sélectionner les données visibles dans la scène 3D en contrôlant par une GUI ou des boutons d'un Pad des petits filtres qui vont modifier ou bloquer les messages des viewers concernés.

[2] http://ganglia.sf.net/
[3] http://oar.imag.fr/

Bilan 14

Nous avons présenté un outil de rendu parallèle appelé FlowVR Render proposant un protocole de description basé sur des shaders et utilisant un ensemble de groupes de polygones comme primitives. Par rapport aux approches standards basées sur des commandes OpenGL, ce protocole offre les avantages suivant :

- Des shaders sont utilisés pour spécifier l'apparence visuelle des objets graphiques. Ils requièrent seulement quelques paramètres et non la complexité de toute la machine à état du pipeline classique OpenGL. Cela permet de définir un protocole plus simple qui n'a pas à gérer les changements d'états.
- Les shaders permettent d'utiliser toute les fonctionnalités des dernières générations de cartes graphiques.
- Les primitives ne sont pas ordonnées et sont indépendantes (du fait de l'absence d'état commun), ce qui rend les opérations de filtrages plus simples et efficaces et permet d'optimiser la boucle de rendu.
- Des informations de haut niveau comme des volumes englobants ou les changements entre chaque image permettent de réduire le coût des traitements et des communications.

Le fait d'utiliser une description de la scène particulière oblige à modifier les applications existantes. Toutefois cet inconvénient peut être facilement surmonté, par exemple dans le cas des applications de visualisation qui n'utilise que quelques primitives graphiques différentes. L'adaptation du code de rendu d'un outil comme VTK, l'un des outils de visualisation les plus utilisés, permet de supporter de manière transparente toutes les applications existantes l'utilisant.

Cet outil n'est pas limité aux applications de visualisation scientifique mais trouve des utilisations pour d'autres applications. La description modulaire de la scène permet

de construire de façon incrémentale les éléments d'un monde virtuel et de les distribuer de manière appropriée sur la grappe de visualisation. L'utilisation de shaders permet de répartir la charge de travail entre les CPU et les GPU. Enfin, l'adjonction de petits filtres entre les viewers et les renderers permet d'implanter facilement des contrôles permettant à l'utilisateur de sélectionner les données à afficher.

Parmi les améliorations intéressantes à développer par la suite il y a la conception d'un mécanisme standardisé d'interaction avec les différents éléments de la scène 3D. Un peu comme le *Window Manager* gère les fenêtres 2D des autres clients d'un serveur *X Window* (section 12.6), il manque un viewer particulier et un jeu de filtre permettant de répartir intuitivement l'espace virtuel et les actions de l'utilisateur entre les différents viewers.

Pour supporter de plus grands environnements virtuels, une autre amélioration importante consiste en l'utilisation de plusieurs niveaux de détails au niveau des primitives, permettant lors du rendu de répartir les ressources en fonction de la visibilité des objets. Ici il est possible d'étendre le protocole utilisé par FlowVR Render, peut être en intégrant des outils existants comme Magellan [111].