Accélération matérielle pour le rendu de scènes multimédia vidéo et 3D

Christophe Cunat

Thèse

présentée pour obtenir le grade de docteur

de l'École Nationale Supérieure des Télécommunications

Spécialité : Électronique et Communications

Christophe Cunat

Accélération matérielle pour le rendu de scènes multimédia vidéo et 3D

Soutenue le 8 octobre 2004
devant le jury composé de

MM.	Jean-Didier Legat	: Président
	Emmanuel Boutillon	: Rapporteur
	Habib Mehrez	: "
	Franck Mamalet	: Examinateur
	Jean Gobert	: Directeur de thèse
	Yves Mathieu	: "

Remerciements

Je tiens tout d'abord à remercier les différentes personnalités qui m'ont fait l'honneur de participer à mon jury de thèse : M. Jean-Didier Legat qui m'a également fait la faveur de présider ce jury, MM. Emmanuel Boutillon et Habib Mehrez qui ont accepté la tâche d'être les rapporteurs de ma thèse ainsi que MM. Jean Gobert, Franck Mamalet et Yves Mathieu.

Je tiens plus particulièrement à exprimer ma gratitude envers M. Yves Mathieu qui a eu la lourde tâche d'être mon directeur de thèse. Ce travail n'aurait jamais abouti sans son indéfectible dynamisme et inébranlable conviction dans l'intérêt de mes travaux. Ses constants encouragements m'ont toujours permis de dépasser les différentes périodes de découragement qui ont pu jalonner ces années de thèse. Merci.

Je tiens également à remercier la société Philips France pour le cadre général de cette thèse CIFRE avec, en particulier, MM. Joseph Adélaïde, Marc Duranton et Jean Gobert qui ont participé à l'encadrement de mon travail. Qu'ils trouvent ici l'expression de ma reconnaissance. Je remercie également M. Thierry Brouste pour m'avoir accueilli au sein de son département tant au LEP, à PRF qu'au PDSL-Paris.

Je n'oublie pas les personnes qui m'ont côtoyé au quotidien dans ces départements et qui m'ont toujours exprimé leur confiance dans la qualité de mon travail : Selim, Ana, Boriana, Robin, Carolina, Thomas, Qin, Laurent, Olivier, Lien, Philippe... Je pense aussi à tous les membres du département ComÉlec et aux nombreuses discussions qui ont émaillé nos repas et pause-cafés : Jean, Jean-Claude, Jean-Luc, Sylvain, Lirida, Renaud, Alexis, les deux Philippe, Cyril, Elizabeth... Je remercie également pour leur disponibilité et leur gentillesse les agents administratifs de ces services : Chantal, Marie-Dominique, Danielle et Marie. Je lui sais gré, ainsi qu'à Joëlle, de m'avoir encouragé tout au long de ma formation initiale et de cette thèse.

Je me permets de continuer cette énumération par les nombreux amis qui doivent désormais pousser un soupir de soulagement : je ne leur rabattrai plus les oreilles avec mes doutes existentiels. Je les remercie d'avoir pu me permettre de prendre suffisamment de recul par rapport à mon travail : Ovarith, Philippe, Baptiste, William, Timothée, Charles, Emmanuelle, Yannick, Fabien, Franck, Thierry, Nicolas, Laurent, Marc, Édouard. Merci à Gérard, Nicolas, Mouhcine et Frédéric pour nos pauses Chez Éric. Sans oublier Besançon : Christophe, Sandrine, Stéphane, Sylvain, Sébastien, Aude...

Merci à Pratchett et ses *Annales du Disque-monde* de me faire toujours rire.

Enfin, je terminerai en exprimant les sentiments que j'éprouve envers mes parents qui m'ont encouragé durant toute ma vie d'élève puis d'étudiant. Je remercie également mon frère pour son soutien permanent. À tous, merci.

La bise et le soleil

La bise et le soleil se disputaient, chacun assurant qu'il était le plus fort, quand ils ont vu un voyageur qui s'avançait, enveloppé dans son manteau.

Ils sont tombés d'accord que celui qui arriverait le premier à faire ôter son manteau au voyageur serait regardé comme le plus fort.

Alors la bise s'est mise à soufflер de toute sa force, mais plus elle soufflait, plus le voyageur serrait son manteau autour de lui, et à la fin, la bise a renoncé à le lui faire ôter.

Alors le soleil a commencé à briller, et au bout d'un moment, le voyageur réchauffé a ôté son manteau.

Ainsi la bise a dû reconnaître que le soleil était le plus fort des deux.

Résumé

Un processus de convergence des techniques algorithmiques de deux domaines autrefois bien séparés (la synthèse d'images 3D et le codage vidéo des images) s'est mis en place au cours de ces dernières années. Cette convergence est facilitée par l'émergence de normes (comme, par exemple, MPEG-4) qui offrent aux utilisateurs des services multimédia. Grâce au concept de codage par objets, une scène peut être reconstituée par la composition de divers objets dans un ordre déterminé.

Cette thèse s'inscrit dans le cadre de la composition d'objets visuels qui peuvent être de natures différentes (séquences vidéo, images fixes, objets synthétiques 3D, etc.). Néanmoins, les puissances de calcul nécessaires afin d'effectuer les opérations de composition demeurent prohibitives sans mise en place d'accélérateurs matériels spécialisés et deviennent critiques dans un contexte de terminal portable.

Une revue tant algorithmique qu'architecturale des différents domaines est effectuée afin de souligner les points de convergence et de mieux appréhender leurs différences. Ensuite, trois axes (interdépendants) de réflexions concernant les problématiques de représentation des données, d'accès aux données et d'organisation des traitements sont principalement discutés.

Les résultats issus de ces réflexions sont alors appliqués au cas concret d'un terminal portable pour la labiophonie : application de téléphonie où le visage de l'interlocuteur est reconstruit à partir d'un maillage de triangles et d'un placage de texture. Une architecture unique d'un opérateur de composition d'image capable de traiter quasiment indifféremment des flux vidéo que des maillages de triangles est ensuite définie. Enfin, une synthèse sur une plateforme de prototypage de cet opérateur autorise une comparaison avec des solutions existantes, apparues pour la plupart au cours de cette thèse.

Abstract

Two fields that used to be well separated are converging: 3D graphics and video coding. The emergence of new standards (such as, for example, MPEG-4) which provide to final users multimedia services makes this convergence easier. Thanks to the concept of object coding, a scene can finally be reconstructed by the composition of various objects in a given order.

This thesis deals with the general framework of visual objects composition. The kinds of objects that can be composed are also various: video sequences, still pictures, synthetic 3D objects, etc. However, the computational power requested in order to perform the composition is still prohibitive without any dedicated hardware and becomes critical in the context of mobile devices.

Algorithmic and architectural reviews are performed in order to determine points that are shared between each field and to emphasize their differences. Next, three (interlinked) major axes of thought are explored: data representation, data accesses and processing organisation.

The concrete case of a mobile device for labiophony allows the direct application of previous results. The labiophony is a smartphone application where the speaker face is reconstructed from a triangle mesh with texture mapping. A single architecture, able to equally process video sequences and triangle meshes, is defined. Finally, the architecture is synthesized and mapped into a prototyping platform. A comparison between this architecture and existing solutions can be performed.

Table des matières

Table des figures

Liste des tableaux

Première partie

Introduction

Portée par l'attente de débits toujours accrus, l'émergence des nouvelles normes de communications autorise la diffusion de nouvelles applications auprès des utilisateurs des nouvelles technologies de l'information et de la communication. Outre les aspects historiques de transmission de voix, ces applications permettent les échanges de données textuelles (comme les SMS), d'images animées ou non : d'une façon plus générale, de données multimédia. La manipulation de ces données multimédia est d'autant plus aisée que ces nouvelles normes (comme MPEG-4) définissent des canevas commun de codage.

Toutefois, ces applications multimédia restent extrêmement consommatrices de ressources : puissances de calcul, mémoire, énergie électrique, etc. Les systèmes exécutant ces applications sont donc particulièrement sollicités afin de respecter des contraintes d'autonomie (dans le cas de systèmes mobiles), d'affichage temps réels (dans le cas d'applications d'images animées) ou, plus prosaïquement, de coût. Si les contraintes de puissances de calcul sont communément relâchées à l'aide de matériels dédiés, la diversification des types de données traitées ne peut s'accompagner d'une diversification similaire de matériel : cette convergence entamée au niveau algorithmique doit se retrouver au niveau matériel.

Notre thèse s'inscrit justement dans ce contexte de convergence engagée par les travaux de normalisation. Parmi les différents supports multimédia, nous nous adressons plus particulièrement à deux domaines qui, jusqu'ici, sont restés cloisonnés. Il s'agit des domaines de la vidéo et de la synthèse graphique 3D. De ces deux domaines, nous envisageons la convergence et l'unification des traitements préalables à l'affichage d'une image sur l'écran d'un système mobile. Ces traitements restant trop coûteux pour un traitement logiciel seul, il est souhaitable de disposer d'un matériel spécialisé effectuant ces traitements quel que soit le type de données (vidéo ou graphisme) à traiter. Nous nous appliquons précisément à la définition de telles architectures matérielles pour le rendu de scènes multimédia constituées à partir de divers types de données.

Cette thèse se découpe en six parties principales dont la première est constituée par cette introduction.

La deuxième partie présente, en s'appuyant sur un scénario concret, le système visé ainsi que les différents domaines que mettront en œuvre les futures applications. Cette partie exposera également les problématiques liées à la convergence de ces domaines.

La troisième partie expose une revue des différents champs, revue abordée sous le double point de vue des algorithmes employés et des architectures existantes ou apparues durant notre travail de thèse.

La quatrième partie apporte un ensemble de réponses personnelles aux problématiques soulevées dans la deuxième partie. Ces réponses s'articulent autour de trois thèmes que sont la représentation des données, l'organisation de traitements matériels dédiés et l'optimisation des échanges mémoires. Nous montrons, à l'aide des points de vue que nous adopterons, comment nous pouvons garantir des fonctionnements optimaux compte tenu des contraintes liées aux aspects mobiles du système visé.

La cinquième partie nous livre une application directe des réponses fournies dans la quatrième partie : elles sont employées à la conception d'un opérateur de compo-

sition d'images compatible avec la norme MPEG-4. Nous rapportons le contexte de cette conception, les développements que nous avons effectués ainsi que les résultats de synthèse de cet opérateur.

Enfin, confortée par l'expérience acquise lors du développement matériel de ce compositeur d'image, la sixième partie propose quelques perspectives de développements ultérieurs afin de compléter cette réalisation. Elle permet également de conclure notre thèse.

Deuxième partie

Convergence entre vidéo et synthèse d'images

Chapitre 1

Contexte

1.1 Un scénario futuriste ?

FIG. 1.1 – L'écran d'un portable dans un avenir proche.

La figure 1.1 nous invite à imaginer ce que seront, dans un avenir plus ou moins lointain, les téléphones portables de demain. Plus qu'un simple combiné téléphonique,

le futur portable grand public devra supporter un large spectre d'applications. Tout d'abord, nous imaginons que l'utilisateur se divertira à l'aide de son portable en se confrontant à d'autres utilisateurs dans des mondes virtuels en trois dimensions. Ces mondes virtuels seront agrémentés de murs sur lesquels pourront s'afficher aussi bien des clips musicaux, publicitaires ou encore des bandes annonces de cinéma. Ce portable continuera encore à assurer des communications téléphoniques classiques mais également en mode visiophonie, sans pour autant qu'un jeu en cours ne soit interrompu.

1.2 Différents domaines supports mis en œuvre

La réalisation du scénario précédent nécessite la mise en œuvre coopérative de différents domaines. L'image finale présentée sur l'écran nous suggère les deux domaines de la synthèse graphique d'image et de la vidéo. Le troisième domaine qui nous intéressera concerne le matériel lui-même.

1.2.1 La synthèse d'images en trois dimensions

La synthèse graphique 3D consiste à créer des images où l'impression de profondeur (la troisième dimension) apparaît. De célèbres artistes, citons notamment Vasarely ou Escher (figure 1.2), se sont fait spécialistes de cette impression de profondeur en jouant plus particulièrement sur la géométrie des objets, les effets de perspectives ou encore les effets d'ombres et de lumières.

Ces différents effets ont été modélisés depuis les années 60 pour les besoins de la synthèse graphique par ordinateur et regroupés dans ce que l'on appelle aujourd'hui le *pipeline graphique*. À partir d'objets conçus par un infographiste, l'enchaînement des fonctions de ce pipeline permet la reconstitution d'une scène prête à l'affichage sur un écran.

Au début des années 90, le contrôle de ce pipeline graphique a été standardisé au travers d'interfaces de programmation d'applications : API ou *Application Programming Interface*. Les interfaces les plus répandues sont OpenGL [1] sur laquelle sont développés les jeux de la famille de Quake et DirectX [2] de Microsoft sur laquelle sont développés (entre autres) les jeux de la console X-box. Ces interfaces, bien implantées dans le monde du jeux vidéo pour PC, ont su évoluer afin de répondre (et parfois devancer) les besoins des joueurs et des programmeurs.

Sans rester cantonnée aux jeux PC et de consoles de salon, la 3D a déjà commencé à envahir les consoles de jeux portables comme la GameBoy Advanced de Nintendo. Avec d'une part, la spécification d'une API particulièrement dédiée aux systèmes embarqués nommée OpenGL-ES [3] (à laquelle ont participé des constructeurs de téléphonie mobile) et d'autre part, le nouveau N-Gage (dernier né de la gamme de Nokia), le portable est la prochaine cible des applications 3D.

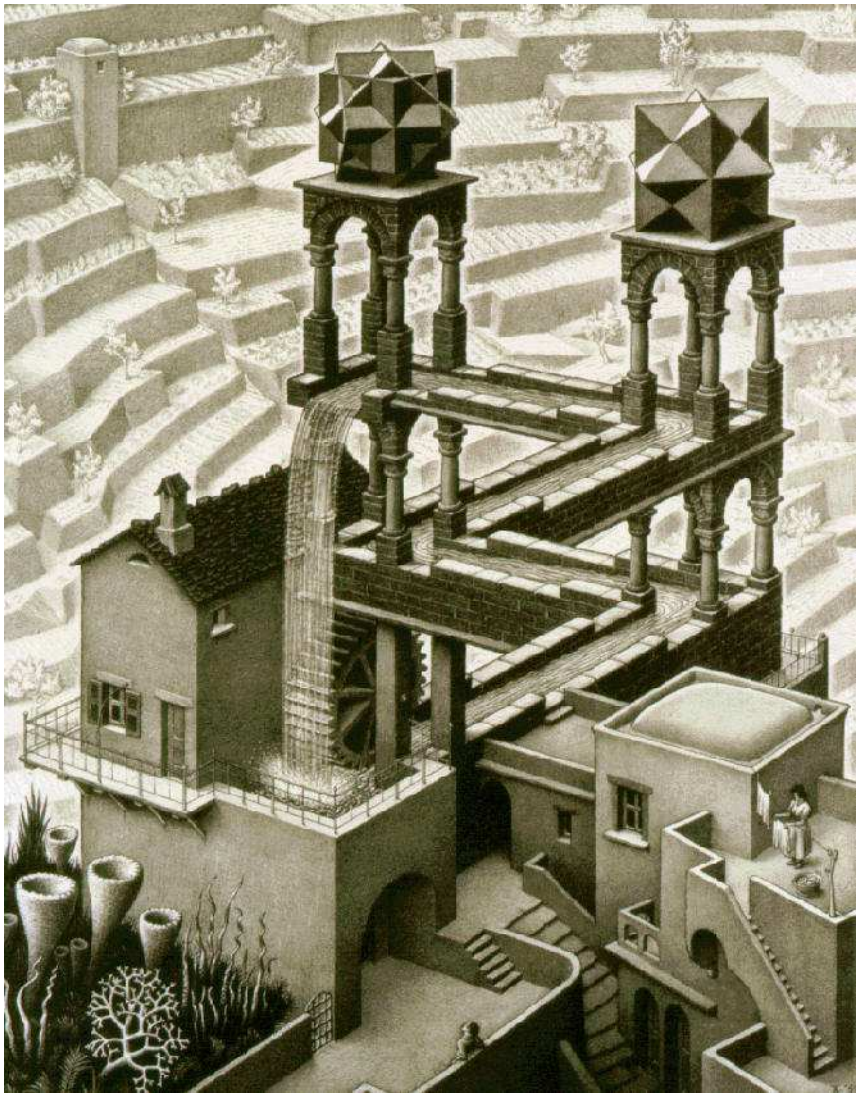


FIG. 1.2 – Reproduction de *Waterfall* 1961,ESCHER. Les effets géométriques et ceux de perspective donnent l'illusion de la profondeur.

1.2.2 La vidéo

Autre domaine mis en jeu, celui de la vidéo est plus ancien que celui de la synthèse graphique.

Depuis 1926 et la première expérience de télévision par John Baird, la vidéo a beaucoup évolué : noir et blanc à ses débuts puis couleur, passage de l'analogique au numérique, et enfin définition de formats de compression normalisés pour le stockage et la diffusion. De même, la chaîne de traitements précédent l'affichage, chaîne qui nous intéressera dans cette thèse, a elle aussi évolué : de traitements passifs (principalement des filtrages), ils sont devenus actifs (composition d'image à partir d'autres images) et même interactifs avec la possibilité pour l'utilisateur d'adapter lui-même le contenu de sa vidéo.

Le développement des applications s'est opéré de pair avec celui des traitements

possibles. Désormais, la spécification de logiciel médiateur comme MHP *(Multimedia Home Platform)* [4] pour la télévision numérique facilitera ce développement en rendant l'application indépendante du boîtier sur lequel elle s'exécute.

Bien installé dans le salon des différents utilisateurs, la vidéo apparaît maintenant sur nos téléphones portables. Munis d'une caméra, ils ne se contentent déjà plus de photographier mais ils filment. Ensuite, ils transmettent cette vidéo par messages multimédia (MMS) : nous pouvons mentionner par exemple le P900 de Sony-Ericsson.

1.2.3 Le support matériel

Le dernier domaine relatif à notre scénario concerne le support matériel sur lequel s'exécutera l'application précédemment décrite.

Le respect des prédictions de la loi de Moore [5] par l'industrie de la microélectronique a contribué à la très forte progression de la puissance de calcul des microprocesseurs. Cette progression a été particulièrement visible pour les processeurs d'ordinateur personnel. Puis, compte tenu de la diminution des tailles des circuits et à la faveur de la naissance des systèmes sur puces (SoC), l'effort d'intégration a augmenté. A l'heure actuelle, cette intégration se poursuit avec les réseaux sur puces (NoC). Ces circuits, en se complexifiant, sont également devenus de plus en plus flexibles de sorte que l'intégration s'est déportée au niveau fonctionnel et applicatif : les dernières consoles de jeux de salons jouent aussi le rôle de lecteur de DVD faisant l'économie d'un autre appareil. Ces appareils marquent le début de la convergence entre les mondes de la synthèse graphique et de la vidéo.

Grâce à l'augmentation de l'autonomie des batteries et aux techniques de conception de circuits basse-consommation, les appareils mobiles ont également profité de l'accroissement des performances des circuits. Comme nous l'avons vu, il existe maintenant des téléphones portables permettant de jouer à des jeux 3D et d'autres permettant de recevoir et d'afficher de la vidéo. Les premiers signes de convergence entre 3D et vidéo sont visibles dans nos salons. Au regard du passé de la microélectronique (et de son avenir), cette convergence ne peut que continuer et se propager sur le marché des appareils portables grand public.

Au cours de cette description du contexte général de notre thèse, nous avons pu remarquer que les bases pour une réalisation de notre scénario de la figure 1.1 existent d'ores et déjà. Les mondes de la vidéo et du graphisme ont commencé à converger. Des appareils grand public sont capables de traiter chacun d'eux séparément. La convergence de ces deux mondes est déjà une réalité en post-production au cinéma avec, par exemple dans l'adaptation cinématographique du *Seigneur des Anneaux*, le personnage de Gollum entièrement de synthèse au milieu des paysages naturels de Nouvelle-Zélande.

Cependant, les dispositifs portables grand public actuels ne permettent pas encore la réalisation de notre scénario. Les prix des appareils portables qui pourront prochainement supporter tout ou partie de notre scénario n'autorisent actuellement pas une démocratisation de telles applications. Enfin, un certain nombre de difficultés qui incluent ces deux précédentes remarques reste à résoudre afin d'assurer la convergence entre les domaines de la vidéo et du graphisme 3D dans un contexte de portable grand public : notre prochain chapitre aborde les différentes problématiques que nous traiterons au long de cette thèse.

Chapitre 2

Problématique

Le développement de nouvelles applications se heurte aux questions de puissance de calcul et d'autonomie énergétique du portable hôte. La puissance de calcul disponible dépend de l'architecture interne du système. La figure 2.1 représente l'architecture typique d'un portable bas-coût composée d'un cœur de microprocesseur embarqué à architecture RISC (typiquement un processeur ARM fonctionnant à 200 MHz), une mémoire unique, un processeur de traitement du signal pour la gestion de l'audio, un (ou plusieurs) coprocesseur(s) dédié(s) accélérant certaines fonctions et un contrôleur d'entrées-sorties. L'interconnexion est assurée par un bus système partagé par la mémoire et les différents blocs.

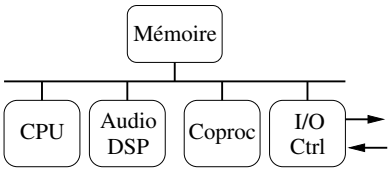


FIG. 2.1 – Architecture typique d'un système pour portable.

L'autonomie énergétique dépend, quant à elle, de la puissance de calcul nécessaire à l'exécution de l'application et de la consommation liée aux accès à la mémoire. Particulièrement gourmandes à la fois en bande-passante mémoire et en puissance de calcul, les applications vidéo et graphiques nécessiteront vraisemblablement le développement de matériels dédiés ajustant au mieux l'efficacité pour ces deux paramètres.

Dans ce contexte de téléphones portables grands publics et bas-coûts, quel est donc le sous-ensemble minimal des algorithmes de synthèse d'images et de traitement vidéo dont l'accélération par du matériel dédié est nécessaire ?

Une réponse à cette première question permet de considérer la question du développement d'un opérateur dédié. Le coût d'un tel opérateur augmentant avec la surface de silicium du circuit, il n'est pas envisageable d'avoir à la fois des circuits dédiés différents pour la vidéo et la synthèse d'images. Ainsi, la convergence attendue des applications implique également une convergence au niveau du matériel. Cependant, les évolutions de ces deux domaines s'étant effectuées indépendamment l'une de l'autre, les algorithmes employés par chacun d'eux sont parfois très éloignés. De plus, les données manipulées sont le plus souvent de nature différente. Dans ces conditions, quelle architecture pouvons-

nous proposer pour effectuer à la fois du rendu vidéo et de la synthèse d'images tout en limitant raisonnablement la puissance de calcul ?

Ces évolutions indépendantes sont fortement liées à une différence fondamentale dont sont empreintes les stratégies de rendu de ces deux domaines. Le rendu vidéo est piloté de manière impérative par la vitesse d'affichage d'une nouvelle image mesurée en Hertz (images par seconde). Qu'elle soit de 25 Hz pour les formats PAL/SECAM ou 30 et 29,97 Hz pour le format NTSC, elle implique un temps de traitement d'une période d'image maximum : le matériel doit nécessairement s'adapter à cette contrainte. En synthèse d'images, les performances se mesurent plutôt en nombre de pixels (points de l'image affichée) et de polygones composant une scène traités par seconde. Ainsi, il importe ici que ce soit l'application qui s'adapte à la contrainte imposée par le matériel : si le nombre de polygones est trop important, la vitesse d'affichage d'une nouvelle image ralentira et la fluidité de la séquence en sera affectée. Comment pourrons-nous rendre compatibles les contraintes orientées « pire cas » *(worst case)* du rendu vidéo avec celles orientées « au mieux » *(best effort)* plus communément appliquées en synthèse d'images ?

Bien que dimensionné pour des conditions d'exécution données, un système peut être amené à faire face à des changements d'échelle d'un ou plusieurs paramètres. Dans le cas d'un système matériel dont l'implémentation est figée, ces changements d'échelle sont rédhibitoires. En particulier, les évolutions de la résolution des écrans entraînent une augmentation du nombre de pixels affichés. Un système extensible assure une aptitude à fonctionner correctement sans perte de ses propriétés essentielles. La maîtrise de la corrélation entre les performances et les besoins d'une application permet à la fois une évolution future d'un système, sa réutilisation dans un cadre non prévu initialement et la mise en place de stratégies de repli. Comment pourrons-nous assurer une extensibilité satisfaisante pour une architecture de rendu d'images ?

Compte tenu de la taille d'une image et du nombre d'images par seconde à afficher, les applications de rendu d'images sont intrinsèquement gourmandes en échanges avec la mémoire. De plus, les traitements effectués sur ces images augmentent encore ces échanges, entraînant une activité importante de la mémoire et un encombrement du bus d'interconnexion. Or ce bus d'interconnexion est devenu le talon d'Achille de tout système à mémoire unifiée : le débit disponible forme un goulot d'étranglement qui peut faire chuter notablement les performances. Il est donc nécessaire de mettre en place des stratégies afin d'optimiser les accès mémoire et de les réduire significativement. De plus, la réduction du nombre d'échanges avec la mémoire est un axe de premier ordre d'optimisation d'un système pour réduire sa consommation énergétique. Quelles stratégies pourrons-nous mettre en place afin de limiter efficacement cette bande-passante mémoire nécessaire à l'exécution d'une application ?

L'usage de tests de performances fait partie des critères objectifs permettant le classement de systèmes de synthèse d'images. Cependant, au delà des performances intrinsèques, le succès commercial d'un système repose aussi sur des critères subjectifs. La qualité des images générées est un de ces critères. Durant la course aux performances engagée par les constructeurs d'accélérateurs de synthèse d'images, la qualité des images a pu parfois apparaître comme secondaire. Au contraire, le monde de la vidéo s'est constamment préoccupé de la question de la qualité des images générées.

Cette préoccupation a conduit au développement de nombreuses innovations pour assurer une qualité toujours meilleure. Il n'est pas envisageable que la convergence vidéo-3D s'accompagne d'une forte régression pour la vidéo. Ainsi, comment la synthèse graphique 3D pourra-t-elle bénéficier des traitements disponibles pour le rendu vidéo ?

Enfin, l'aspect économique du développement d'un circuit est un autre paramètre important. En effet, le prix de fabrication d'un circuit rend prohibitif la moindre erreur de conception. De plus, la complexité d'intégration d'un système complet requiert méthodologie et pragmatisme. Ainsi, les délais de conception se sont allongés alors que les délais de mise sur le marché ont tendance, eux, à se réduire. Ces contraintes économiques ne permettent plus désormais de redévelopper un nouveau circuit spécifique dès qu'une nouvelle application émerge. Une architecture développée aujourd'hui doit être suffisamment souple pour répondre demain à de nouveaux besoins : la réutilisabilité d'une architecture dépend de cette souplesse d'utilisation. Ainsi, de circuits intégrés spécifiques pour une application, nous entrons dans l'ère des processeurs à instructions spécifiques pour une classe d'applications. À ce titre, quel niveau de programmabilité pourrons-nous envisager pour assurer de manière déterminante la pérennité de notre architecture ?

En résumé, voici la liste des questions considérées :

1. Quelle architecture pouvons-nous proposer pour effectuer à la fois du rendu vidéo et de la synthèse d'images tout en limitant raisonnablement la puissance de calcul ?
2. Comment pourrons-nous rendre compatibles les contraintes orientées « pire cas » *(worst case)* du rendu vidéo avec celles orientées « au mieux » *(best effort)* plus communément appliquées en synthèse d'images ?
3. Comment pourrons-nous assurer une extensibilité satisfaisante pour une architecture de rendu d'images ?
4. Quelles stratégies pourrons-nous mettre en place afin de limiter efficacement cette bande-passante mémoire nécessaire à l'exécution d'une application ?
5. Comment la synthèse graphique 3D pourra-t-elle bénéficier des traitements disponibles pour le rendu vidéo ?
6. Quel niveau de programmabilité pourrons-nous envisager pour assurer de manière déterminante la pérennité de notre architecture ?

Troisième partie

Algorithmes et architectures pour le rendu d'images

Chapitre 3

Algorithmes de rendu d'images

Afin de mieux appréhender les enjeux et les conséquences d'une convergence entre les domaines de la vidéo et de la synthèse graphique, il sied d'approfondir notre connaissance de chacun de ces domaines. Cela nous permettra de souligner à la fois les possibilités de convergence et les points de divergence. Nous commençons, dans ce chapitre, par une revue des différents algorithmes de rendu d'images et leurs liens logiques entre eux. Nous nous attarderons plus particulièrement sur les premières similitudes flagrantes qui auront été identifiées. Enfin, le chapitre suivant donnera lieu à une revue des différentes architectures développées au fil des âges pour les domaines qui nous intéressent.

3.1 Le rendu d'une scène graphique 3D

Un vaste ensemble d'algorithmes d'une complexité variable est à la disposition d'un concepteur afin d'automatiser le rendu d'une scène graphique comme celle de la figure 3.1. Ce chapitre vise à présenter les principales étapes du rendu et leurs caractéristiques ainsi que l'enchaînement général des traitements. De nombreuses présentations de ces traitements [6, 7, 8] ou de points plus précis comme le placage de texture [9, 10, 11] sont disponibles. Foley *et al.* [12] effectue une revue plus complète des algorithmes employés en synthèse graphique.

La figure 3.2 reproduit le pipeline graphique complet qui s'est imposé peu à peu à travers la définition des API graphiques. Un large spectre de complexités de scènes peut être rendu par ce pipeline qui n'est cependant pas unique. De même, ce pipeline est suffisamment souple pour échelonner la qualité visuelle du rendu sans pour autant atteindre la qualité des algorithmes de lancer de rayon. Il s'agit, pour ces derniers, d'algorithmes de génération d'images fondés sur le trajet, depuis l'œil de la caméra, du chemin inverse de la lumière à travers chaque pixel pour remonter aux sources lumineuses de la scène. Créés par Appel en 1968 (cf. [12]), ces algorithmes combinent à la fois l'élimination des faces cachées et les calculs d'illumination et d'ombrages ; ils permettent d'obtenir un réalisme très poussé au prix cependant d'une complexité de calcul accrue due au parcours récursif des rayons.

Enfin, notons également des algorithmes fondés sur la manipulation d'images (IBR

FIG. 3.1 – Copie d'écran du jeu Vampire. Image au format VGA (640 × 480).

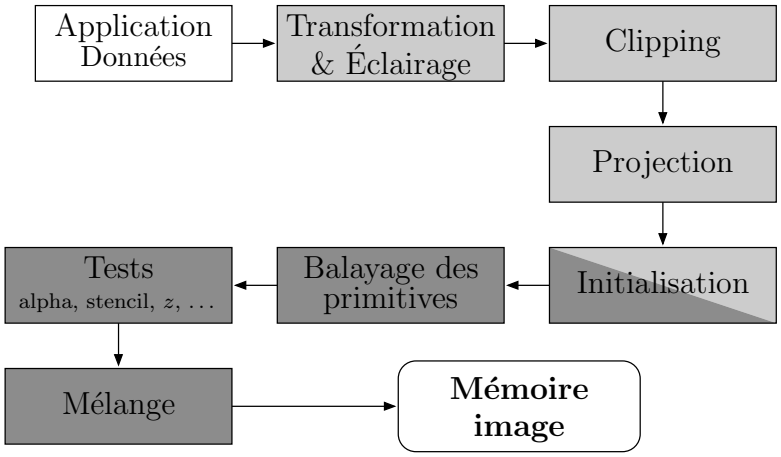


FIG. 3.2 – Vue complète du pipeline graphique classique.

Image-Based Rendering) [13] pour la synthèse d'images de qualité photoréaliste, qualité que ne permet pas encore d'atteindre la synthèse géométrique. Ces algorithmes ne traitent donc plus une représentation géométrique de la scène mais une base de données photométrique. Cette base de donnée contient une image d'une scène ainsi que les informations de profondeur depuis le centre de vision. L'IBR permet de reconstruire cette scène en changeant de point de vue.

3.1.1 Modélisation d'une scène

La modélisation d'une scène ne constitue pas à proprement parlé une phase du pipeline graphique : elle en constitue le préliminaire. Une scène est modélisée hiérarchiquement à l'aide d'un arbre de scène comme un assemblage de divers objets. Ces objets sont eux-mêmes composés d'un assemblage de primitives graphiques élémentaires comme les plans, des quadriques (sphère, ellipsoïde...) et/ou surfaces de Bézier et autres NURBS *(Non-Uniform Rational B-Spline)* [14]. Des méthodes (la CSG, *Constructive Solid Geometry* par exemple) et des langages de modélisation comme VRML *(Virtual Reality Modeling Langage)* [15] permettent la composition également de manière hiérarchique

de ces objets. Les objets graphiques sont ensuite gérés par le moteur graphique d'une application qui se charge d'en nourrir le pipeline graphique avec les primitives de base que peut accepter celui-ci. En raison de la faible complexité des calculs mis en œuvre par la suite et parce qu'ils permettent d'approximer à un niveau de détail variable n'importe quelle surface, le triangle et les maillages de triangles se sont rapidement imposés comme primitives de base. Ces primitives (listes de triangles isolés, de triangles accolés les uns aux autres – *strip* – ou de triangles en éventail – *fan* – autour d'un point commun) permettent la modélisation de la scène complète. La figure 3.3 illustre la modélisation ainsi réalisée d'un des personnages de la scène précédente, figure 3.1, scène composée d'environ 4 000 triangles [16].

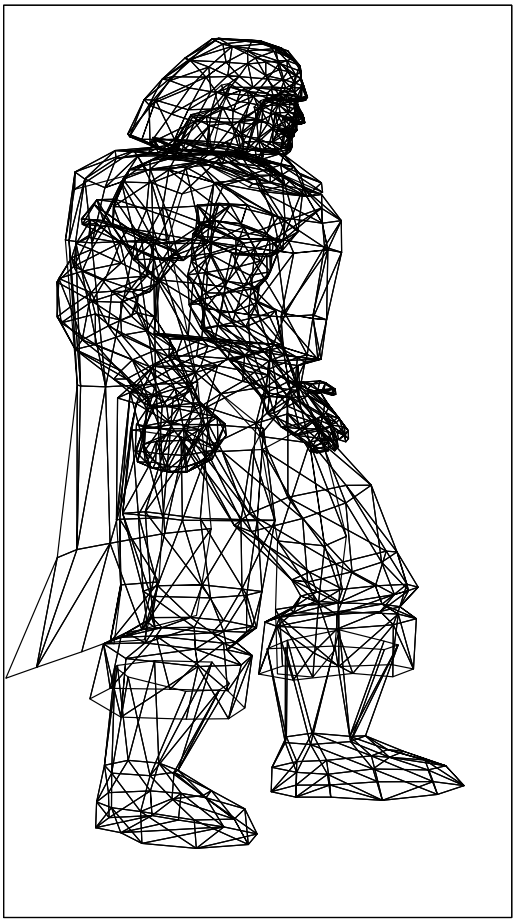


FIG. 3.3 – Modélisation d'un personnage composant la scène de la figure 3.1.

3.1.2 Pipeline graphique

Le pipeline graphique peut se découper en quatre grands ensembles de traitements que sont les transformations géométriques, l'illumination de la scène, le parcours des géométries et l'élimination des surfaces cachées.

3.1.2.1 Transformations géométriques

Lors de sa modélisation, un objet graphique est défini dans son propre repère intrinsèque : l'espace objet. L'instanciation dans une scène s'accompagne d'une série de transformations géométriques que définit l'arbre de scène. La première transformation qui ne dépend que de l'objet consiste à le plonger dans l'espace monde représentant la scène complète. La position, à l'intérieur de ce monde de la caméra définit un nouvel espace, l'espace vue. Une nouvelle transformation commune pour la scène complète permet de passer de l'espace monde à ce nouvel espace. La dernière transformation correspond au passage de la 3D à la 2D, c.-à-d. à la projection de l'espace vue sur l'espace écran où les calculs finaux s'effectueront. La figure 3.4 illustre ces différentes transformations.

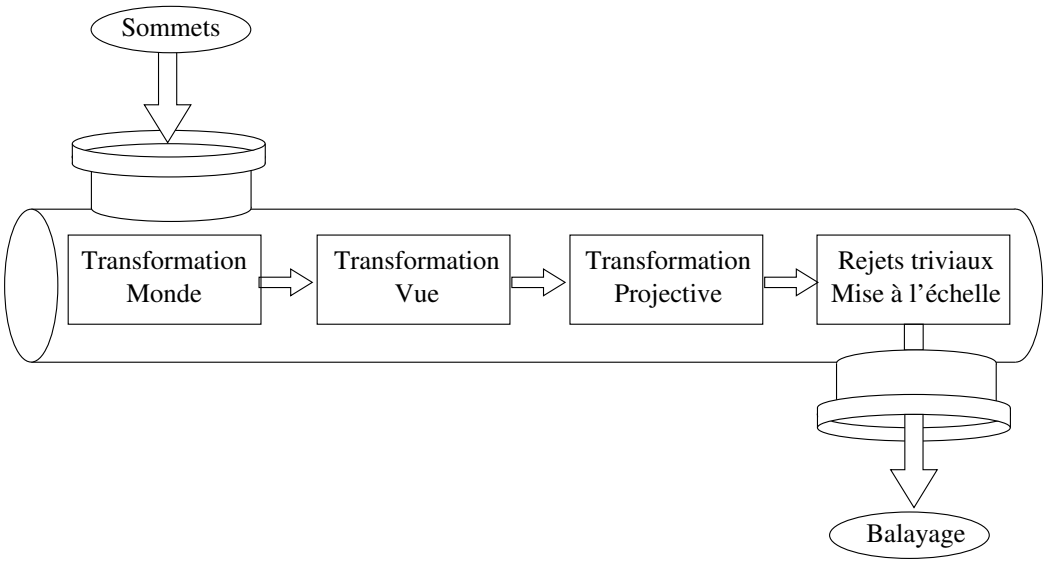


FIG. 3.4 – Enchaînement des transformations géométriques.

Ces transformations s'opèrent grâce à leur formulation sous forme de produits matriciels sur les coordonnées des sommets des différents objets (coordonnées en représentation flottante). L'équation (3.1) met en lumière cette formulation à l'aide de coordonnées homogènes avec M le point d'origine, M' le point transformé et T la matrice de transformation, elle-même produit d'une matrice de projection en perspective et d'une de transformation affine ; le point projeté sur le plan $z = 0$ avec le centre de projection placé à $z = -d$ est le point m' donné en (3.2) requérant deux divisions. L'usage des coordonnées homogènes ainsi que leurs propriétés sont détaillés par Mancini [17] dans le chapitre 4.

$$
\begin{aligned}
M' &= T \cdot M \\
\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 1 \end{pmatrix} \cdot \begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}
\end{aligned}
\tag{3.1}
$$

$$m' = \begin{pmatrix} x'/w' \\ y'/w' \end{pmatrix} \tag{3.2}$$

3.1.2.2 Illumination

Une scène est également composée de sources lumineuses qui éclairent les objets dans l'espace monde précédemment défini. Une phase de calcul d'illumination permet de calculer les informations pertinentes au niveau des objets pour le calcul effectif des couleurs des futurs pixels de l'écran. Encore une fois, cette phase de calcul ne s'effectue qu'au niveau des sommets des objets de la scène.

Le modèle d'illumination couramment employé se décompose en trois composantes de complexité croissante :

- une composante ambiante qui correspond à l'illumination globale de la scène. L'intensité est alors $I = k_a I_a$ où k_a correspond au coefficient de réflexion ambiante spécifique à l'objet et I_a à l'intensité de la lumière ambiante.
- une composante diffuse qui correspond à l'application de la loi de Lambert sur l'illumination d'objets mats. L'intensité d'une source lumineuse réfléchi ne dépend que de son angle θ avec la normale à la courbe et du coefficient k_d de diffusion du matériau. L'intensité pour la source j est alors $I^j = k_d I_d^j \cos\theta$.
- une composante spéculaire qui modélise l'effet de la rugosité d'un objet. Cette composante dépend de l'angle α entre la direction de vision et celle de la source lumineuse. Phong [18] a modélisé cette composante par $I^j = k_s I_s^j \cos^n \alpha$ où n est un coefficient d'atténuation spéculaire.

Ces trois composantes sont illustrées par la figure 3.5.

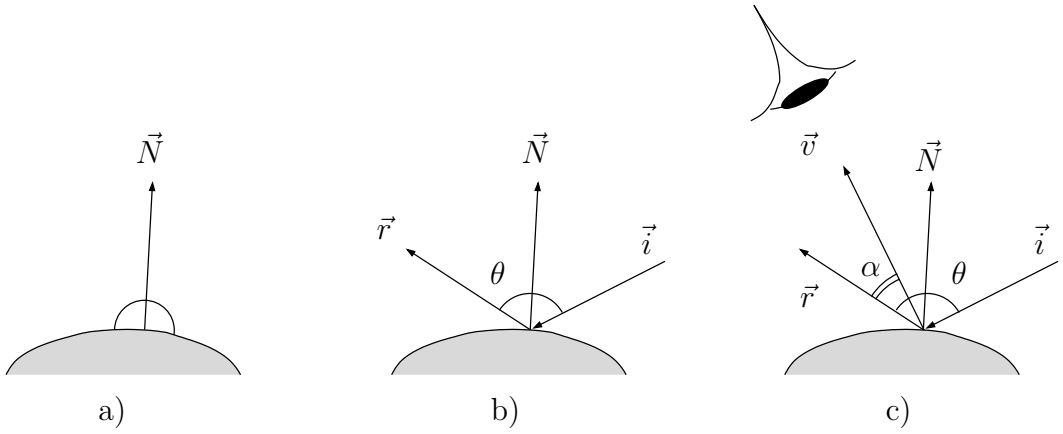


FIG. 3.5 – Modèle d'illumination : a) ambiante, b) diffuse, c) spéculaire.

3.1.2.3 Parcours des primitives

Avec le parcours des primitives, le pipeline graphique marque le passage des traitements touchant les sommets des primitives à ceux touchant à la production massive et répétitive des futurs pixels de l'écran. La couleur des pixels de l'écran est déterminée par celle de la primitive qui se projette sur ce pixel. La technique d'un pipeline classique [1, 19, 2, 20] consiste à parcourir chacune des primitives afin de déterminer les pixels recouverts et leurs couleurs. Ce parcours est décomposé en deux phases. Une première phase d'initialisation permet de déterminer à partir des informations disponibles au niveau des sommets un jeu de coefficients plus adapté aux traitements pixels. Ceux-ci s'effectuent lors de la seconde phase qui interpole pour chaque pixel les différentes données de couleurs qui seront combinées pour produire la couleur finale.

Initialisation

La première phase d'initialisation [21, 22] marque ce passage effectif entre la description de la scène à l'aide de primitives et la production des pixels. Cette phase a pour but de simplifier le traitement des pixels : à partir des valeurs des données aux sommets, elle dérive des coefficients permettant l'interpolation sur toute la primitive de chaque donnée. Cette interpolation s'effectue alors le plus souvent de manière incrémentale. Comme nous le verrons dans le chapitre suivant, cette phase a été aussi la charnière entre les traitements logiciels et les traitements matériels des premières architectures accélératrices de synthèse 3D pour les ordinateurs personnels.

Couleur et placage de texture

Selon la complexité du modèle d'éclairage, une, deux ou trois composantes de couleurs sont disponibles au niveau des sommets. Au niveau des pixels, trois modèles de remplissage, *shading* en anglais, permettent de combiner ces composantes. Ces modèles de complexité croissante (illustrés figure 3.6) sont :

- le remplissage plat ou *flat shading*. Ce modèle simpliste colorie uniformément une primitive. Des composantes aux sommets, l'initialisation se charge de dériver une couleur unique qui sera appliquée à tous les pixels de l'écran recouvert par cette primitive.
- le remplissage de Gouraud [23]. Il permet d'assurer la continuité de la couleur en interpolant linéairement à l'intérieur de la primitive les couleurs des sommets.
- le remplissage de Phong [18]. Il permet d'obtenir des effets plus réalistes dans le cas d'un point lumineux n'éclairant que l'intérieur de la primitive. Dans ce modèle, ce ne sont plus les couleurs qui sont interpolées mais les vecteurs normaux qui permettent pour chaque pixel de recalculer de manière exacte l'intensité lumineuse d'une source.

Seuls, les effets de couleurs requerraient une modélisation extrêmement fine des objets afin de modéliser des surfaces complexes comme des murs ou des mosaïques sur un sol. Le placage de texture développé par Catmull en 1974 [12] s'est rapidement imposé comme solution [24, 25, 9]. Il permet d'augmenter le réalisme d'une scène à l'aide d'une image (la texture) qui recouvre comme du papier peint une primitive. La modélisation

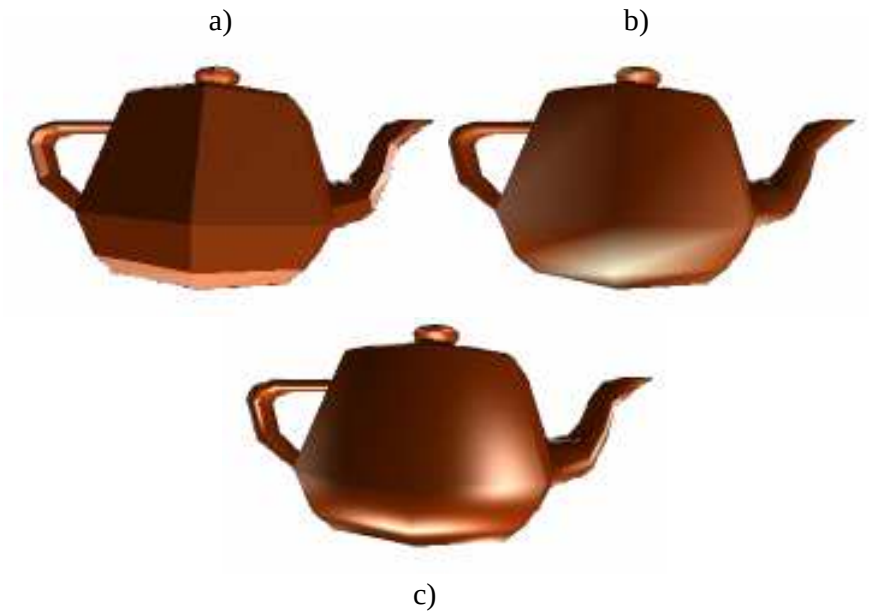


FIG. 3.6 – Modèles de coloriage d'une primitive : a) *flat shading*, b) *Gouraud shading*, c) *Phong shading*.

de la scène précise ainsi la position dans l'espace texture des différents sommets d'une primitive. Les coefficients d'une transformation sont alors dérivés lors de la phase d'initialisation du parcours de la primitive. Ces coefficients permettent par la suite de déduire pour chaque pixel le point correspondant dans la texture (point dénommé *texel*).

3.1.2.4 Élimination des surfaces cachées

Les derniers traitements effectués par le pipeline graphique consistent en l'élimination des surfaces cachées [26, 27]. Cette élimination rejette les surfaces non visibles de la caméra. La complexité en profondeur est définie par le nombre de surfaces se projetant sur un même pixel : elle vaut en moyenne 2,5 pour la figure 3.1. Afin de réduire au maximum cette complexité et la puissance de calcul nécessaire aux traitements des primitives et des pixels, des rejets triviaux sont effectués en divers endroits du pipeline. Ainsi, la caméra définit dans l'espace monde un volume de vision qui permet de rejeter rapidement des objets complets extérieurs à ce volume. L'initialisation du parcours d'une primitive peut également rejeter une primitive en fonction de l'orientation de la surface : la moitié arrière d'une sphère n'a pas besoin d'être traitée.

Pour gérer les cas non triviaux, des algorithmes utilisant des listes de priorité, comme l'algorithme du peintre, traitent les primitives selon l'ordre de priorité et ne conservent dans une mémoire image que la couleur de la dernière primitive qui se projetait sur un pixel donné. L'algorithme du *z-buffer*, présenté toujours par Catmull, permet de gérer correctement les objets qui s'intersectent ou pour lesquels aucun ordre de priorité n'existe, comme figure 3.7. Un plan mémoire est associé à la mémoire image qui contient pour chacun pixel la coordonnée de profondeur z du dernier point visible projeté sur ce pixel. Ainsi, le parcours d'une primitive détermine à la fois une couleur pour

un pixel et la profondeur dans la scène de l'antécédent du point projeté. L'écriture en mémoire image est alors conditionnée par un test entre cette profondeur et la dernière valeur stockée dans le *z-buffer*.

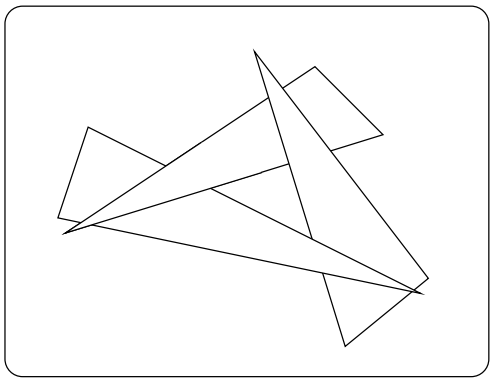


FIG. 3.7 – Cas de primitives ne possédant pas d'ordre de priorité.

3.1.3 Compléments sur le pipeline graphique

La figure 3.8 présente une vue complétée du pipeline graphique classique intégrant les parties désormais programmables de ce pipeline. La version 8 de l'interface de programmation DirectX [2] a introduit la notion de *vertex shader* et de *pixel shader*. Ces *shaders* permettent l'exécution de micro-programmes écrits dans un langage spécifique aussi bien au niveau du traitement géométrique (les sommets des primitives) qu'au niveau des traitements des pixels. Ces mêmes notions ont également été rajoutées dans la bibliothèque de fonctions graphiques OpenGL [19, 28] lors de la définition de la version 2 de cette bibliothèque.

3.2 Traitements vidéo

3.2.1 Modélisation

En vidéo, l'image à afficher a longtemps constitué une entité élémentaire. Cependant, dès ses débuts, cette image tirait profit des propriétés de l'œil : la persistance rétinienne permet de diviser l'image en deux trames ou *fields*. La première trame n'affiche que les lignes paires à la date n tandis que la seconde affichent les lignes impaires à la date $n + 1$. L'image complète est reconstruite au niveau du cerveau humain.

Avec l'émergence des normes de compression numériques, les années 90 ont introduit la notion de blocs et macroblocs dont le pavage compose l'image complète. Les évolutions récentes ont vu l'apparition des notions d'objets visuels et de composition. Une image est désormais segmentée en plusieurs objets indépendants. Une gestion de ces objets est alors mise en place permettant à un compositeur (logiciel ou matériel) de reconstruire l'image avant de la présenter. En particulier, la norme MPEG-4 supporte

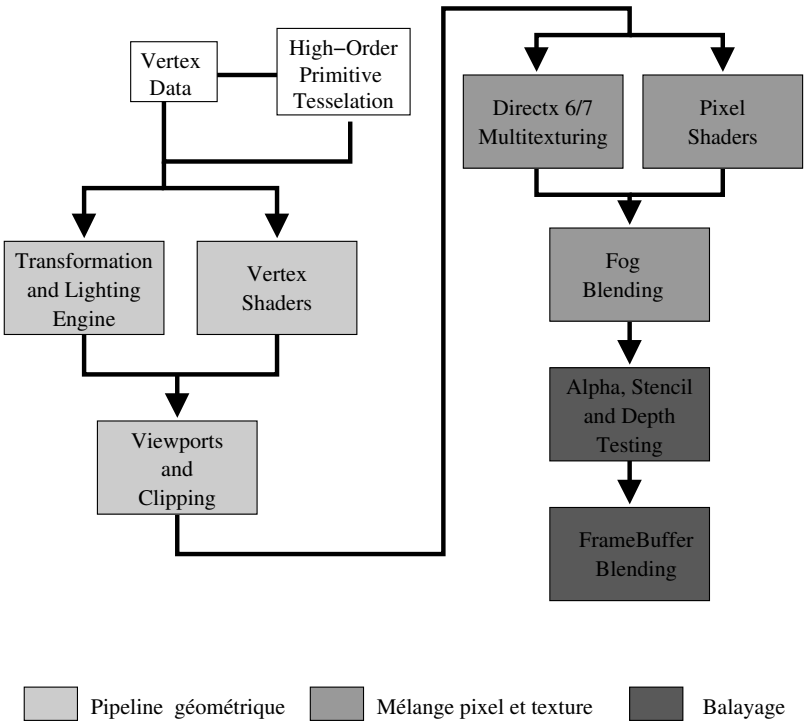


FIG. 3.8 – Pipeline graphique intégrant des parties programmables.

différents types d'objets visuels : objets vidéo, naturellement, mais aussi des objets regroupés sous la dénomination SNHC *(Synthetic/Natural Hybrid Coding)*. Parmi ceux-ci, notons des objets composés d'un réseau maillé déformable, des objets « texte », « texture » et/ou des *sprite* contenant tous les pixels d'une même séquence vidéo [29].

3.2.2 Codage numérique des images

Les groupes de normalisation MPEG *(Moving Picture Expert Group)* de l'ISO (l'organisation internationale de normalisation) ont contribué au développement de nombreuses normes de compression vidéo dont les principes ont été repris par ailleurs. Le codage comprend trois opérations principales :

- Estimation de mouvement. À l'aide d'une image de référence, elle estime des vecteurs de déplacement de portions de l'image afin de souligner la redondance temporelle entre les images.
- Transformation fréquentielle. Afin de réduire la redondance spatiale, une transformation en fréquence est effectuée : typiquement une transformation en cosinus discrète (DCT) ou en ondelettes (DWT).
- Quantification et codage. La quantification tire à nouveau profit de l'œil humain qui est peu sensible aux hautes fréquences en les réduisant de manière plus ou moins forte. Un codage entropique permet enfin de compacter les informations.

L'opération de décodage opère alors dans l'ordre inverse ces étapes : décodage, déquantification, transformation inverse puis compensation de mouvement [30]. Malheureusement, la quantification associée à une transformation fréquentielle comme la DCT peut introduire des effets de bloc. La boucle de décodage complète incorpore également

des traitements spécifiques afin de réduire les artéfacts liés aux techniques de codage employées.

3.2.3 Post-traitements

Une fois décodée, une image peut subir alors des post-traitements spécifiques d'une application visée. Un mobile comme le SH505i de Sharp, par exemple, se livre à des traitements spécifiques afin d'apporter stéréoscopiquement l'illusion de la 3D pour des images 2D. Ces traitements sont également dépendants de la technologie de l'écran d'un tel mobile : dans le cas d'écran LCD nécessitant un rétro-éclairage, une diminution de la puissance de cet éclairage réclame des réglages de luminosité du flux vidéo [31]. Dans le cas télévisuel, les spécificités des écrans plasma exigent un affichage en tout ou rien à très haute fréquence afin de profiter de la persistance du signal lumineux. Les images intermédiaires réclament alors un codage par des méthodes de grains *(dithering)*.

Ces post-traitements font la valeur ajoutée d'un système par rapport à un autre. Une part importante de ces traitements constitue ce que l'on appelle le *smart-imaging*. Il s'agit de traitements hautement non linéaires et non standardisés. À partir d'informations sur l'image globale, des traitements pixels à pixels permettent des modifications d'histogrammes colorimétriques, de rehausser les contours en réduisant l'étendue des zones de transitions et en augmentant le contraste entre zones.

3.2.4 Un pipeline de rendu vidéo

De même que pour le rendu d'une scène graphique, l'ensemble des traitements d'un flux vidéo est organisé à l'intérieur d'un pipeline de rendu vidéo. La figure 3.9 offre une vision de ce pipeline de rendu vidéo tel qu'il peut être inclus aujourd'hui dans un boîtier de réception pour la télévision.

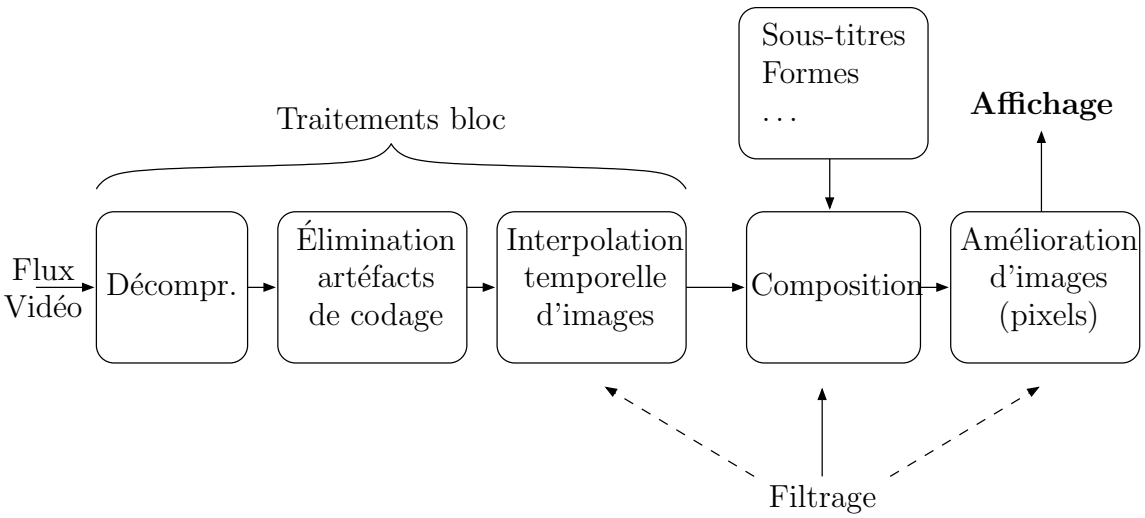


FIG. 3.9 – Pipeline des traitements vidéo.

Compressé pour la transmission, le flux vidéo est décompressé immédiatement après réception. En raison des techniques de compression fondées sur un codage par bloc, la primitive de base utilisée ici est constituée de blocs de 8×8 pixels. Bien que tirant parti des propriétés de la vision humaine pour limiter la dégradation subjective de l'image, la compression introduit parfois des artéfacts de codage. Des algorithmes d'estimations de bruits et de décisions procèdent au niveau du bloc à l'élimination des artéfacts. Ces traitements sont suivis par des algorithmes de conversions de format. Ils impliquent aussi bien des traitements spatiaux que des traitements temporels. Ils intègrent également le désentrelacement de flux vidéo comme pré-requis à ces conversions de format.

Les images générées sont ensuite associées avec des éléments contextuels comme des sous-titres ou des formes pour être composées et former les images finales. Les derniers traitements de cette chaîne concernent des algorithmes d'amélioration d'images qui s'appliquent pixel à pixel.

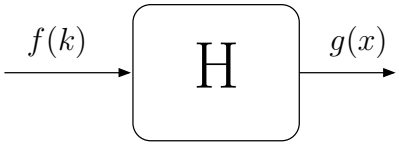
Pour une présentation et des discussions plus détaillées des algorithmes des différents chaînons de ce pipeline, nous pourrons nous reporter aux diverses contributions de G. de Haan [32].

3.3 Premières similitudes

Les présentations que nous venons de voir concernant les pipelines de chaque domaine laissent entrevoir différentes possibilités de convergence. Parmi celles-ci, nous abordons ici plus particulièrement les fonctions de filtrage et l'organisation générale des traitements et discutons des usages qui en sont faits selon le domaine visé.

3.3.1 Le processus de filtrage

Comme indiqué figure 3.9, des fonctions de filtrage sont au cœur des traitements vidéo. Elles le sont également pour le placage de texture, élément désormais central du pipeline graphique. Le traitement de signaux échantillonnés nécessite en effet l'emploi de filtres de reconstructions et/ou passe-bas adéquats afin de respecter, entre autres, les contraintes de la théorie du signal. La figure 3.10 représente d'une manière générale l'opération de convolution effectuée par un filtre.



$$g(x) = \sum_{k \in \mathbf{E}} f(k)h(x,k)$$

Fig. 3.10 – Opération de filtrage par un filtre H.

3.3.1.1 Flux vidéo et filtrage

L'utilisation de filtres pour le traitement de flux vidéo se répartit en deux classes d'applications. La première consiste en les applications de rééchantillonnage d'images. La théorie du signal, émise par Nyquist en 1924 et formalisé en 1949 par Shannon [33], nous explique que la reconstruction à l'identique d'un signal échantillonné n'est possible que sous la contrainte d'une fréquence d'échantillonnage double de la largeur de bande fréquentielle du signal original. Dans le cas contraire, des effets de repliement de spectre *(aliasing)* apparaissent [34, 10, 35, 36]. La figure 3.11 présente l'enchaînement théorique nécessaire dans le cas de rééchantillonnage spatial (remise à l'échelle *(rescaling)*, *picture-in-picture* ou transformations géométriques). Des méthodes issues de la compensation de mouvements sont parfois utilisées afin de compléter la pertinence de cet enchaînement théorique.

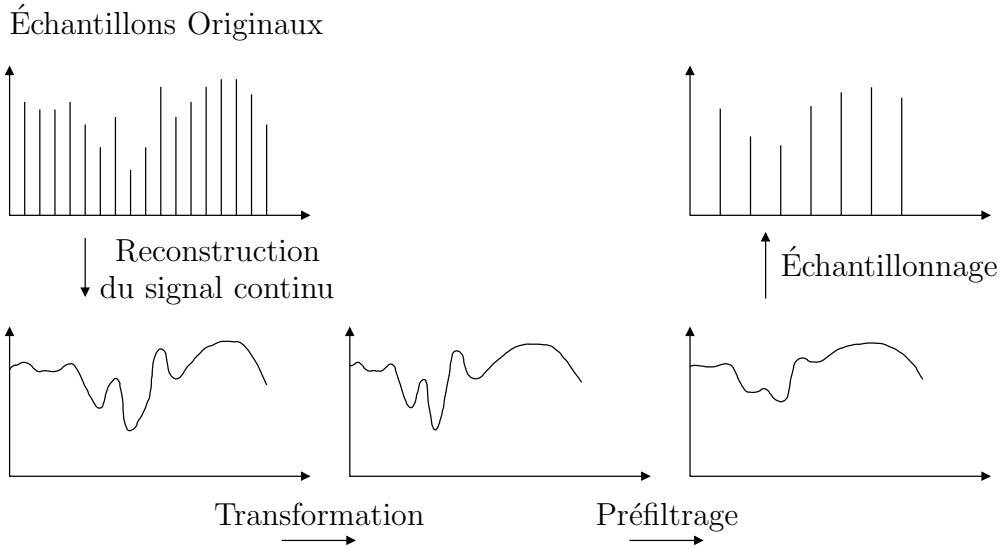


FIG. 3.11 – Enchaînement des opérations pour le rééchantillonnage d'images.

L'amélioration d'image constitue la seconde classe d'applications des filtres pour le traitement de flux vidéo. Les distorsions occasionnées par les techniques de compression vidéo sont atténuées à l'aide de filtres (linéaires ou non) de post-traitements [37]. Le codage MPEG induit plus particulièrement trois distorsions (effet de bloc, contours flou dus aux rebonds du signal et l'effet *mosquito noise*). La figure 3.12(b) illustre le résultat de filtrages spécifiques aux effets de blocs visibles sur la figure 3.12(a). Les algorithmes de réduction de bruits impliquent une classification des zones afin d'adapter dynamiquement à la zone courante la nature du filtre appliqué : une zone comportant naturellement des hautes fréquences exige un filtrage passe-bas moins sélectif qu'une zone uniforme bruitée. Enfin, l'enchaînement de ces différents traitements réclame un soin particulier afin de ne pas provoquer d'incompatibilité entre, par exemple, des algorithmes de rehaussement de contour ajoutant du bruit comme effet secondaire et des algorithmes chargés de réduire celui-ci.

(a) Effet de bloc dû au codage MPEG

(b) Filtrage de ces effets

FIG. 3.12 – Effet de blocs et corrections.

3.3.1.2 Synthèse d'images et filtrage

La projection d'une scène graphique sur la grille de l'écran nécessite de manière analogue le rééchantillonnage des textures par rapport à la grille de l'écran. Cependant, si les traitements vidéo sont le plus souvent limités à des filtres de rééchantillonnage spatialement invariants (le cas des transformations affines), une transformation en perspective tel que symbolisée figure 3.13 requiert l'usage de filtres spatialement variants : intuitivement, nous voyons que l'espacement des texels se resserre en se rapprochant du point de fuite.

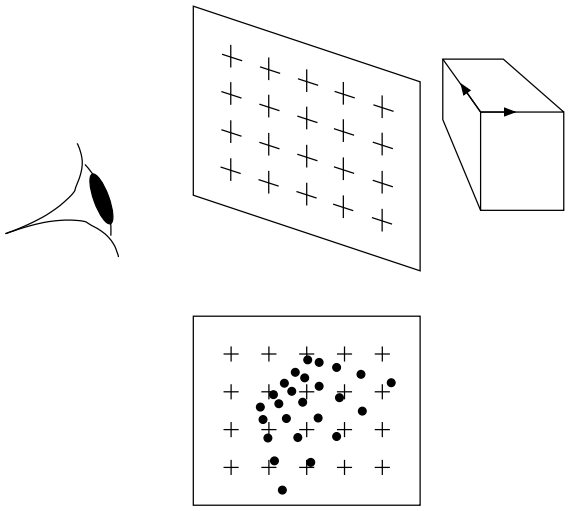


FIG. 3.13 – Projection en perspective.

L'utilisation de tels filtres qui nécessiterait le recalcul de leur empreinte pour chaque pixel apportant une complexité de calcul importante, de nombreuses méthodes approchées ont été mises en place [9], tout d'abord par l'emploi de filtres de reconstruction simples et invariants – *point sampling*, filtre bilinéaire [24] – puis par l'emploi d'images préfiltrées – table précalculée [38], pyramide d'images [39]. Ces pyramides d'images *(MIP-map)* représentent une même image sous différentes résolutions. Divers calculs de la résolution courante [40, 41] permettent de choisir le(s) niveau(x) de *MIP-map* utilisé(s) pour le filtrage des texels. La figure 3.14 illustre le cas d'un filtrage trilinéaire où les résultats d'un filtre bilinéaire appliqué dans deux niveaux consécutifs de *MIP-map* sont linéairement interpolés pour approcher la valeur courante. Enfin, un filtre anisotropique qui applique un suréchantillonnage selon l'axe d'anisotropie de la transformation marque l'évolution actuelle de ces techniques approchées.

Outre les problèmes de repliements de spectre dus au non-respect du critère de Nyquist, le filtrage permet également de prendre en compte la nature physique d'un pixel. En effet, si mathématiquement un pixel n'est qu'un point, sur un écran, un pixel correspond à une surface conduisant en particulier à un crénelage des contours. L'augmentation de la résolution de l'image permet de réduire cet effet au dépend de la puissance de calcul nécessaire à la génération des nouveaux pixels. Cependant, de nombreuses méthodes géométriques de calcul de recouvrement d'aire et de calcul au niveau sous-pixel [42, 43, 44, 45] cherchent à minimiser la puissance de calcul tout en assurant, comme le montre la figure 3.15, un anticrénelage de bord visuellement acceptable.

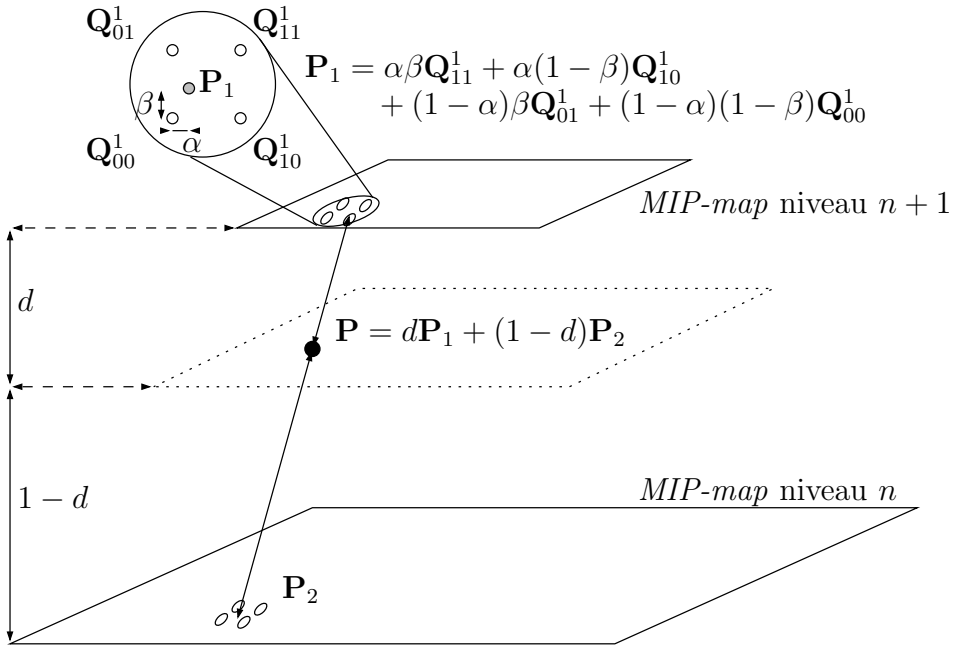


FIG. 3.14 – Filtrage trilinéaire.

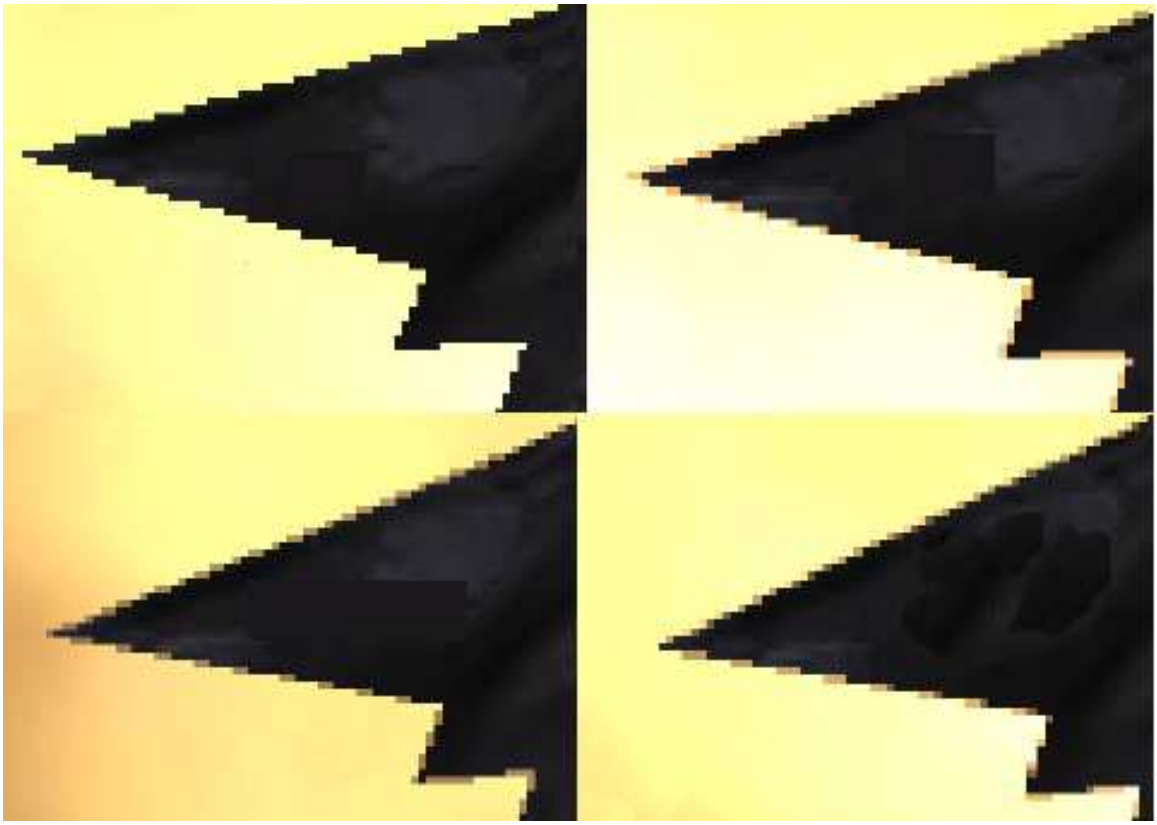


FIG. 3.15 – Anticrénelage de bord [46]. De haut en bas, de gauche à droite : sans anticrénelage, suréchantillonnage $\times 2$, méthode propriétaire *n*VIDIA, suréchantillonnage $\times 4$. La méthode propriétaire est d'une complexité comparable avec la seconde méthode pour une qualité visuelle comparable à la dernière méthode.

3.3.2 Organisation des traitements

Les différents traitements que nous avons vus s'effectuent de manière répétitive sur plusieurs milliers de primitives dans le but de générer les pixels de l'image courante. Le choix de l'organisation et de l'ordonnancement global des traitements implique de fortes conséquences sur les performances d'une architecture et les structures de données échangées avec la mémoire. La figure 3.16 expose, par exemple, trois algorithmes de traitements dans le cas du placage de texture [9]. Le premier est piloté par la production des pixels et nécessite l'inversion de la transformation entre l'image d'entrée et l'image de la sortie ; il correspond à un parcours dit arrière. Le deuxième semble plus naturel puisque sans inversion de la transformation mais requiert une attention particulière lors du parcours pour ne pas oublier des pixels de l'écran ce que limite un algorithme comme proposé par Fant [47] ; il correspond à un parcours dit avant. Enfin, le troisième utilise la séparabilité des transformations pour s'inscrire aisément dans une organisation en flux de données par exemple.

```
Balayage orienté écran

Pour chaque y
  Pour chaque x
    Calculer u(x,y) et v(x,y)
    Copier TEX[u,v] dans ECR[x,y]
```

```
Balayage orienté texture

Pour chaque v
  Pour chaque u
    Calculer x(u,v) et y(u,v)
    Copier TEX[u,v] dans ECR[x,y]
```

```
Balayage en deux passes

Pour chaque v
  Pour chaque u
    Calculer y(u,v)
    Copier TEX[u,v] dans TMP[u,y]
Pour chaque y
  Pour chaque u
    Calculer x(u,y)
    Copier TMP[u,y] dans ECR[x,y]
```

FIG. 3.16 – Ordre des boucles de traitements pour le placage de texture. (x, y) (resp. (u, v)) correspondent aux coordonnées écran (resp. texture).

La figure 3.16 ne fait pas apparaître la boucle sur les primitives de traitements. La prédominance ou non de celle-ci permet également un classement des algorithmes de rendu.

3.3.2.1 Rendu orienté image globale

Le rendu orienté « génération d'une image globale » intervient lorsque la boucle sur les primitives est la plus extérieure comme l'indique le pseudo-code de la figure 3.17.

```
Pour chaque image
  Pour chaque primitive
    Parcourir la primitive
  Afficher l'image contenue dans le tableau ECR
```

FIG. 3.17 – Pseudo-code de rendu orienté image.

Ce mode de rendu est particulièrement utilisé en synthèse d'images 3D. En effet, le pipeline graphique décrit par les interfaces de programmation invite de lui-même à une telle organisation par la description au fur et à mesure de la scène et la présence d'une commande équivalente à `finDeScène()`. De plus, la sémantique des commandes impose

comme présupposé que celles-ci soient logiquement réalisées après avoir été physiquement effectuées par le programme.

Ce mode de rendu possède deux impacts majeurs. Tout d'abord, le rendu pouvant commencer dès la réception de la première primitive, il n'est pas possible de connaître *a priori* le temps de traitement d'une image complète : le parcours est itéré tant que des primitives sont disponibles en entrée dont le nombre ne sera limité que par l'indulgence du programmeur envers l'architecture sous-jacente. Ensuite, l'ordre des objets et leur aire de projection sur l'écran étant quelconques, ce rendu contraint une architecture à disposer d'une mémoire image complète.

Au delà de ces deux contraintes, ce mode de rendu ne nécessite pas de structures de données complexes pour être mis en œuvre. Par suite des évolutions historiques (rendu en fil de fer, rendu de faces puis placage de texture), le balayage orienté image – calculer les coordonnées de texture en fonction des coordonnées courantes de l'écran – est souvent associé à ce type de rendu.

Enfin, le pipeline graphique présente beaucoup de possibilités de parallélisation, tant au niveau des primitives qu'au niveau du parcours de celles-ci, que cette organisation permet d'exploiter aisément. C'est pourquoi cette organisation fut reprise régulièrement par les différentes architectures que nous détaillerons dans notre prochain chapitre.

3.3.2.2 Rendu orienté ligne

Avant les nouveaux types d'écran LCD ou à plasma, un écran était constitué d'un tube dont le faisceau d'électron balaie la matrice active de l'écran pour l'exciter. Le rendu orienté ligne correspond à l'ensemble des algorithmes qui autorise l'affichage de la scène au fil de la production des pixels. Le pseudo-code de la figure 3.18 illustre cette organisation où la boucle sur les primitives se trouve imbriquée à un niveau inférieur que précédemment.

```
Pour chaque image
  Pour chaque ligne
    Pour chaque primitive
      Parcourir la ligne de la primitive
    Afficher la ligne courante
```

FIG. 3.18 – Pseudo-code de rendu orienté ligne.

De même, cette organisation possède deux impacts majeurs. Tout d'abord, elle impose un rendu différé *i. e.* toute les primitives de la scène doivent être connues avant de débuter l'itération sur les lignes de l'image. En second lieu, pour être efficace, cette organisation nécessite de ne pas parcourir inutilement des primitives et donc de maintenir des listes actives remises à jour à chaque ligne. Une phase préliminaire de mémorisation des primitives et de leur tri est donc indispensable.

En contrepartie, seule une mémoire ligne est indispensable dans ce cas-ci limitant la mémoire nécessaire pour une architecture adoptant cette organisation : une mémoire image de 640×480 en couleur (trois octets par pixel) nécessite 900 ko contre environ 275 ko pour mémoriser les primitives d'une scène de 4 000 triangles [16] et 3,75 ko pour

mémoriser une double ligne correspondant à la ligne couramment affichée et celle en voie de production.

Relativement peu employé en synthèse graphique (citons tout de même une architecture de *z-buffer* ligne systolique [48]), cette organisation se retrouve particulièrement en traitement vidéo. Le faible nombre de primitives, voire l'unicité de la source (elle-même transmise dans l'ordre d'affichage comme dans le cas de la télévision hertzienne), joue en faveur d'une telle organisation. Enfin, ce rendu orienté ligne se marie avec des techniques de filtrage monodimensionnel en deux passes, horizontale puis verticale, évoquées figure 3.16 en autorisant un pipeline de traitement. Ces filtrages monodimensionnels se déclinent également en trois passes afin de limiter une perte de résolution lors de la génération de l'image intermédiaire [49].

3.3.2.3 Rendu par tuile

Ce type de rendu, employé à la fois en synthèse graphique [50] et en composition d'image vidéo [51], représente un compromis entre les deux précédentes approches. L'écran est partitionné en zones dénommées tuiles dont le pseudo-code, figure 3.19, indique le traitement effectué.

```
Pour chaque image
  Pour chaque tuile
    Pour chaque primitive dans la tuile
      Parcourir la primitive
  Afficher l'image
```

FIG. 3.19 – Pseudo-code de rendu par tuiles.

Comme le rendu orienté ligne, le rendu par tuile impose un rendu différé d'une scène graphique et un tri des primitives avant de débuter le rendu d'une tuile. Ce tri permet de limiter la puissance de calcul pour le rendu d'une tuile. Cependant, les primitives recouvrant plusieurs tuiles engendrent une augmentation artificielle du nombre de primitives traitées par scène [52, 53, 16].

L'intérêt du rendu par tuile se situe dans la localité des traitements qu'il propose, en opposition avec les rendus orientés ligne ou image. L'aire d'une tuile (typiquement 32×16 pour des circuits tels que le Kyro [50], jusqu'à 128×128 pour des systèmes comme le PixelPlane) permet l'usage efficace de cache de textures limitant les accès mémoires. Ces tailles de tuiles sont aussi adaptées pour les traitements vidéo puisque compatibles avec les tailles des blocs et macroblocs de compression. La taille des tuiles permet également de disposer directement sur le circuit d'une mémoire de profondeur pour une tuile diminuant la mémoire allouée pour l'application et, par suite, le test de profondeur n'a plus besoin d'accéder à la mémoire extérieure. De même, la mise en œuvre d'une mémoire d'accumulation locale *(A-Buffer)* [42] réduit significativement les échanges externes. Une telle mémoire est utilisée lors de rendu en plusieurs passes permettant le rendu de flou de bougé par exemple. La composition entre les différentes passes sollicitant lectures et écritures s'effectue localement. De plus, autorisant un placage de texture différé, c.-à-d. effectué après le test de profondeur par exemple, le rendu par tuile n'autorise l'accès aux

informations de texture pertinentes que pour les pixels visibles de chaque primitives. Il apparaît donc comme un moyen efficace pour limiter la bande-passante mémoire.

Enfin, une architecture de rendu par tuiles présente le dernier avantage de l'extensibilité. En effet, il est possible de paralléliser les traitements au niveau tuile de façon totalement indépendante. Les performances en nombre de tuiles traitées varient donc linéairement avec le nombre de processeurs disponibles. De plus, en travaillant localement sur une tuile, aucune constante ne limite de manière physique la taille des images pouvant être générées reliant en première approximation le temps de traitement d'une image au nombre de tuiles la pavant.

En raison de ces deux aspects, le rendu par tuile est étudié de près par les sociétés marquantes de la synthèse graphique. Bien évidemment par Imagination Technology dont les circuits sont fondés sur ce principe (cf. chapitre suivant), mais également des sociétés au pipeline graphique plus traditionnel comme le montre de récents brevets de chez *n*VIDIA traitant du rendu par tuile et de ces implications en termes de tri de primitives, des commandes et des états du pipeline [54, 55, 56, 57]. Cette organisation sera réétudiée plus en détail chapitre 7 sous des aspects organisation des traitements et gestion des accès mémoire.

3.4 Résumé

De cette présentation algorithmique des deux domaines qui nous intéressent, nous avons pu faire ressortir quelques caractéristiques que résume le tableau 3.1.

	Nb Objets	Taille	Qualité visuelle
Synthèse 3D	Élevé	Petite	Pauvre
Vidéo	Faible	Grande	Très recherchée

TAB. 3.1 – Résumé pragmatique de trois caractéristiques des domaines traités.

Ces caractéristiques et des considérations propres à chaque domaine ont alors conduit à des implémentations matérielles de ces algorithmes de rendu d'images disparates. Nous examinons justement dans notre prochain chapitre les architectures les plus marquantes pour les deux domaines considérés.

Chapitre 4

Architectures matérielles

Le nombre de calculs pour effectuer une opération comme le placage de texture dépend de nombreux paramètres. Pour une scène comme la figure 3.1 au format VGA (640×480), format que l'on peut imaginer retrouver prochainement sur des écrans de mobile, interviennent de façon naïve la complexité en profondeur de la scène (environ 2,5), le nombre de composantes par pixel (3 : Rouge, Vert et Bleu) et le nombre d'opérations nécessaire au rééchantillonnage (12 dans le cas d'un filtre bilinéaire). L'équation (4.1) approxime donc le nombre d'opérations pour un placage de texture simpliste :

$$640 \times 480 \times 2{,}5 \times 3 \times 12 \tag{4.1}$$

Soit 26 millions d'opérations par image. Les puissances de calculs mises en jeu sont donc importantes ; il en est de même pour les accès mémoires. Ces deux contraintes ont donc imposé la conception d'architectures matérielles dédiées afin d'assurer un confort visuel par un rafraîchissement temps réel des images et de limiter le coût de fabrication de systèmes de rendu d'images.

4.1 Architectures pour la synthèse graphique

Les premières architectures de synthèse d'images sont nées dans les années 1960 afin d'accélérer le rendu d'images pour des simulateurs de vol. Seuls les contours géométriques étaient alors dessinés (rendu en fil de fer) compte tenu des puissances de calcul demandées. En effet, trois goulots d'étranglement sont susceptibles de limiter les performances d'un système de synthèse graphique :

- le nombre de sommets pouvant être traités dans la partie transformation géométrique. Cela limite le nombre d'objets dans une scène. Les calculs s'effectuent sur des nombres en représentation flottante.
- le nombre de pixels pouvant être traités lors du parcours des primitives. Cette limite influence à la fois le nombre d'objets d'une scène et la taille des images à l'écran. Les calculs s'effectuent sur des nombres en représentation entière grâce à la mise au point de techniques arithmétiques incrémentales dès 1965 pour le tracé de ligne [58].

– la bande-passante mémoire qui peut limiter l'accès aux données géométriques, aux textures et les traitements au niveau pixel (écriture mémoire, tests de profondeurs...).

De nombreuses architectures ont été bâties afin de résoudre tour à tour ces différentes limitations en vue d'accroître sans cesse les performances. Sans vouloir être exhaustif, nous allons à présent revoir certaines d'entre-elles.

4.1.1 Des architectures pour de hautes performances

En vue de résoudre successivement les deux premières limitations, les architectures développées ont été organisées en machines massivement parallèles. Agencées en systèmes multiprocesseurs SIMD *(Simple Instruction Multiple Data)* ou MIMD *(Multiple Instructions Multiple Data)*, un réseau d'interconnexion se charge alors de distribuer les primitives aux différents processeurs.

4.1.1.1 PixelPlane et PixelFlow

L'université de Caroline du Nord a développé de nombreuses architectures massivement parallèles. La première, le PixelPlane [59, 60, 61, 62] ne visait qu'à résoudre la limitation du nombre de pixels pouvant être traités. Il est composé d'un ensemble de processeurs élémentaires traitant chacun un unique pixel de l'écran. Cet assemblage est illustré figure 4.1.

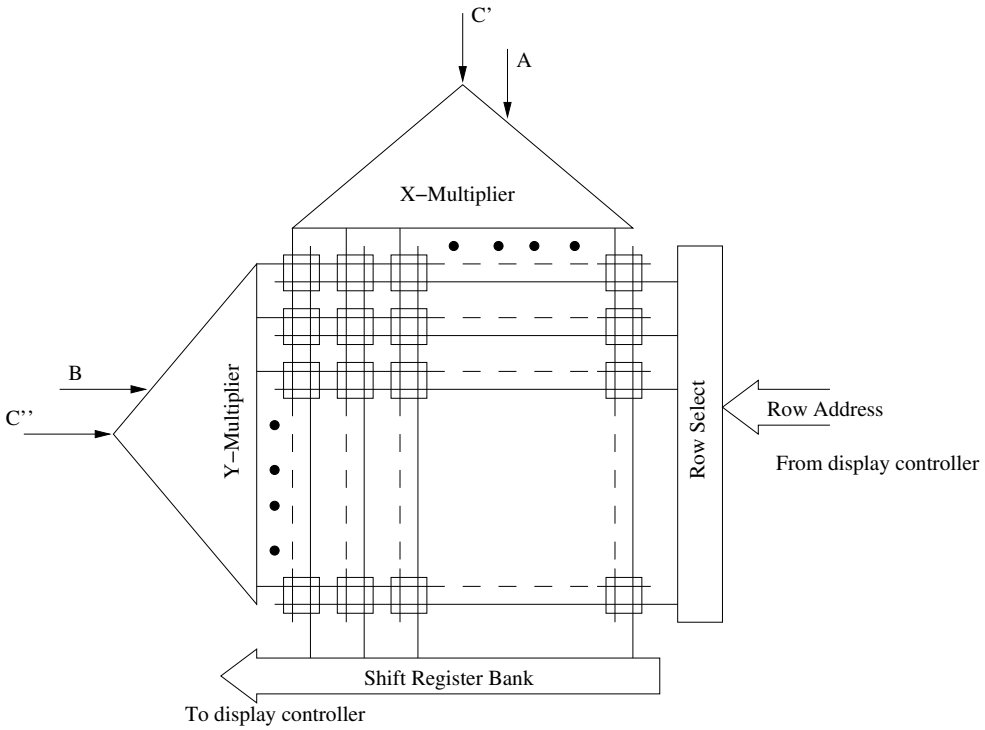


FIG. 4.1 – Unité de traitement du PixelPlane.

À partir d'équations linéaires évaluées pour chaque colonne et chaque ligne de la matrice de cellules élémentaires puis recombinées pour chaque pixel, l'intérieur d'une primitive est calculé en parallèle pour chaque point de la matrice [63]. Grâce à ces évaluateurs d'équations globaux, cette architecture peut supporter pour un surcoût réduit des équations quadratiques pour la gestion de primitives de plus haut niveau que le simple triangle (défini par les trois équations de droite de ces côtés) ainsi que le modèle de Phong pour le remplissage. Une cellule de base maintient en interne des registres de couleurs et de profondeur appliquant l'algorithme du *z-buffer*.

Bien que ne permettant pas le placage de texture, cette architecture présente les avantages suivants :

- extensibilité : l'image est découpée en tuiles de la taille de l'unité de traitement (128×128 pixels) qui sont rendues indépendamment par une ou plusieurs unités interconnectées, le cas échéant, à l'aide d'un réseau de communications avec la mémoire image. Cette découpe en tuile assure une relation entre les performances, le nombre d'unités de traitement et la taille de l'image.
- temps de traitement d'une primitive garanti : l'évaluation en parallèle pour tous les pixels des équations assure un temps de traitement d'une primitive indépendant de la surface de celle-ci sur l'écran. Le temps de traitement d'une image établit une relation strictement linéaire avec le nombre de primitives de l'image, ajoutant à l'extensibilité de cette architecture. Évaluant une équation par cycle d'horloge, cette architecture identifie en 3 cycles les pixels appartenant à un triangle, rejette en 1 cycle par le calcul de la coordonnée z les pixels non visibles et met à jour en 3 cycles les trois composantes RGB de la couleur de celui-ci : 7 cycles par triangle.

Plus qu'un simple circuit de synthèse d'image, la seconde architecture développée par le même laboratoire est un système complet dédié à la synthèse graphique. Le PixelFlow [64, 65, 66, 67, 68] est fondé sur la composition d'images. Une image étant découpée en tuiles, plusieurs chaînes de traitement se répartissent les primitives d'une même tuile. Ces chaînes se recombinent lors de l'écriture finale dans la mémoire image ainsi que l'illustre la figure 4.2.

Outre le découpage en tuiles, cette architecture applique le rendu différé : la détermination de la couleur d'un pixel est séparée du parcours géométrique. Cette séparation tant logique que physique (les traitements s'opèrent sur des processeurs différents) assure une limitation du nombre de pixels dont la couleur est effectivement déterminée, assurance donnée par le rejet anticipé de pixels non visibles. Cette limitation intervient indépendamment de la complexité en profondeur de la scène.

Cependant, cette architecture nécessite un réseau d'interconnexion disposant d'une bande-passante importante (> 10 Go/s), réseau totalement dédié pour la recombinaison de l'image finale. Ce réseau de recombinaison et les mémoires nécessaires pour chaque processeur ne sont pas envisageables dans le cas de systèmes embarqués. Plus généralement, les architectures parallèles sont soumises aux deux contraintes de partage de la charge de travail et de la recombinaison des traitements effectués.

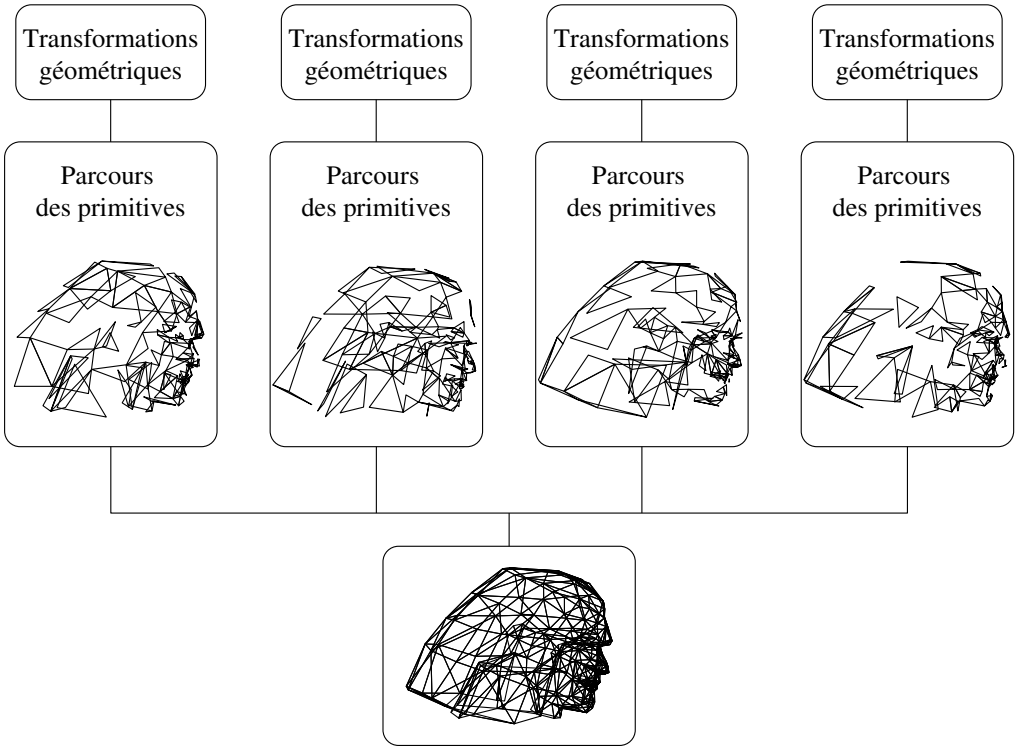


FIG. 4.2 – Composition d'images.

4.1.1.2 RealityEngine et InfiniteReality

Commercialisé par la société SGI, le RealityEngine [69] et son successeur l'InfiniteReality [70] prennent en charge le pipeline graphique en entier découpé en quatre étapes distinctes. Tout d'abord, un processeur interprète les commandes OpenGL et organise le contrôle du pipeline complet. Les informations géométriques sont extraites des commandes et des *geometry engines* (composés de mémoire et d'un cœur de processeur RISC Intel i860 [71] spécialisé dans les calculs flottants) effectuent l'ensemble des opérations de transformations et d'illumination. Les triangles ainsi traités sont ensuite émis en direction de circuits spécifiques de parcours *(fragment generators)* dont le nombre dépend de la configuration de la machine. Chaque circuit est responsable d'une partie de la mémoire image. Cependant, celle-ci est découpée de façon à entrelacer finement les zones de chaque générateur permettant un équilibrage des charges de traitement entre ces circuits lors du parcours de chaque primitive. La portion de la mémoire image gérée par ces circuits est elle-même répartie sur 16 *image engines* qui reçoivent les pixels générés lors du parcours d'un triangle. Ils effectuent alors les derniers tests et opérations avant de stocker le pixel final dans la mémoire. La figure 4.3 illustre une configuration possible de cette architecture.

Cette architecture embarque cependant une large quantité de mémoire. Afin de ne pas limiter la génération des fragments par les accès mémoire, chaque générateur dispose de 16 Mo de mémoire où est dupliquée chaque texture.

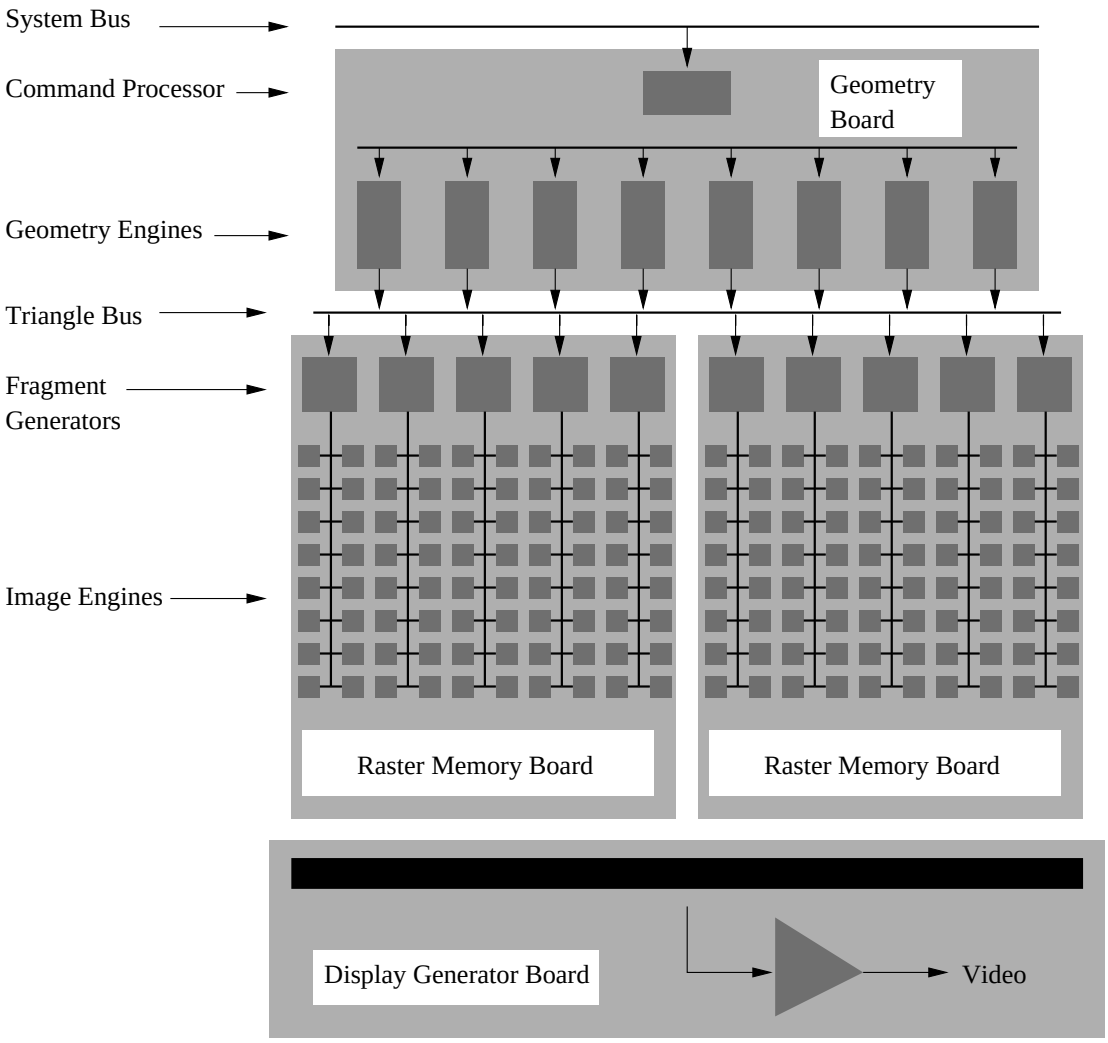


FIG. 4.3 – Architecture du RealityEngine.

4.1.2 Architectures pour processeurs 3D intégrés

4.1.2.1 Talisman

Talisman [72, 73] est une architecture initiée par Microsoft au milieu des années 1990 pour permettre le support multimédia sur ordinateurs personnels et a connu une réalisation partielle en 1999 [74]. Quatre concepts sont mis en œuvre dans cette architecture :

- la composition d'images. La mémoire image est constituée de calques composés à la génération du signal vidéo.
- la compression d'images. Capable de traiter également des flux MPEG-1 et 2, ce système tire partie des techniques de compression en DCT pour réduire les échanges mémoires.
- le rendu par tuiles.
- le rendu multi-passes. L'usage de calques permet à ceux-ci d'être rebouclés en entrée du pipeline graphique comme texture pour de nouvelles primitives.

Cependant, l'écriture des calques puis leur relecture pour la composition à la volée du signal de sortie augmentent la bande-passante mémoire nécessaire pour le rendu d'une image ne tirant ainsi pas totalement partie des avantages liés au rendu par tuiles. Enfin, la composition à la volée requiert des calculs redondants avec ceux effectués par l'opérateur de rendu ce qui n'optimise pas la puissance de calcul au niveau de l'architecture.

4.1.2.2 Accélérateurs graphiques pour PC

La synthèse d'images 3D a subi une révolution en 1996 avec l'introduction pour le marché du PC des premières cartes de la société 3Dfx, rachetée en décembre 2000 par *n*VIDIA. Le tableau 4.1 reprend les évolutions marquantes de ce marché depuis 1996. Un comparatif plus complet est disponible en ligne [75].

Tout d'abord, ce tableau montre chronologiquement l'importance de chacune des limitations rencontrées et les réponses apportées. Afin d'augmenter les performances en terme de nombre de pixels traités, ces architectures ont dupliqué le pipeline de rendu : 1, 2, puis 4 pixels sont rendus indépendamment par primitives.

Plusieurs réponses sont apportées à la limitation de la bande-passante. La première est l'utilisation du rendu par tuile avec des accès à la mémoire centrale restreints grâce à l'utilisation de mémoires locales et aux avantages du rendu différé que nous évaluerons paragraphe 7.1. L'introduction par la société S3 de sa technique de compression S3TC (S3 *Texture Compression*) intégrée dans la version 6.0 de l'API DirectX apporte une deuxième réponse. Cette technique de compression par bloc 4×4 permet un facteur de compression de 6 en représentant le bloc original à l'aide d'une table de 2 couleurs originales et 2 couleurs interpolées comme le montre la figure 4.4. Le gain en bande-passante ainsi obtenu permet d'améliorer la qualité visuelle d'une scène en augmentant le nombre de textures utilisées.

La société Intel a apporté une troisième réponse au problème de la bande-passante

Cartes	**Date**	**Mpix./s**	**Mtri./s**	**Commentaires**
Voodoo1	1996	45	1	3Dfx, 2 Mtransistors.
PCX1	1996			Videologic, premier circuit PowerVR : rendu par tuile.
Riva 128	Q3 1997	100	1,5–5	*n*VIDIA, 3 Mtransistors, interface mémoire 128 bits.
Voodoo2	Q2 1998	90	3	3Dfx, double texture par pixel
Riva TNT	Q3 1998	180	< 8	*n*VIDIA, 2 textures par pixels, 7 Mtransistors.
Savage3D	Q3 1998	125	5	S3, filtre trilinéaire, compression de texture S3TC.
TNT2	Q2 1999	250	9	*n*VIDIA, double pipeline en parallèle, 8 Mtransistors.
	juil. 1999	L'API DirectX7 prend en compte les futures possibilités des cartes graphiques pour supporter l'accélération matérielle des calculs de géométrie.		
GeForce256	Q3 1999	480	15	*n*VIDIA, *Graphic Processor Unit* (GPU) [76] : accélération géométrique. 4 pipelines indépendants de traitement pixel.
Savage 2000	Q4 1999	250	10–25	S3, GPU, jusqu'à 4 textures par pixel, filtrage anisotropique, 12 Mtransistors.
GeForce2	Q2 2000	800	25	*n*VIDIA, GPU de seconde génération, *pixel shading*, 25 Mtransistors.
Radeon	Q3 2000	366	27,5	ATI, GPU, 30 Mtransistors.
	déc. 2000	Rachat de 3Dfx par *n*VIDIA.		
KyroII	Q2 2001	350		ST, cœur PowerVR (tuile) sans accélération géométrique, 15 Mtransitors.
GeForce3	Q2 2001	800	40	*n*VIDIA, support de DirectX8, *Pixel* et *Vertex shader* (v1.3), AntiAliasing de haute résolution, 57 Mtransistors.
Radeon8500	Q3 2001	1 100		ATI, *Pixel* et *Vertex Shading* (v1.4), 60 Mtransistors.
GeForce4	Q2 2002	1 000		*n*VIDIA, 63 Mtransistors.
Radeon9800	Q2 2003	2 600		ATI, interface mémoire 256 bits, 8 pipelines de traitements, 117 Mtransistors.
GeForceFX	Q2 2003	3 600		*n*VIDIA, environ 125 Mtransistors.

TAB. 4.1 – Évolutions marquantes du marché des circuits graphiques pour PC. En comparaison, le Pentium4 ne comprend que 42 Mtransistors.

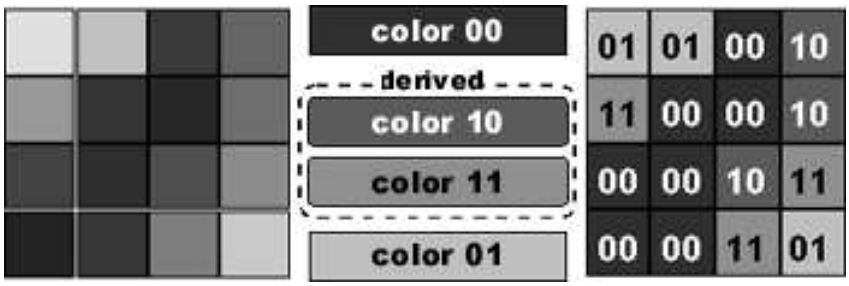


FIG. 4.4 – Technique de compression de texture S3TC.

en dédiant un bus aux communications entre le processeur et le circuit graphique : le bus AGP *(Accelerated Graphics Port)* [77]. La figure 4.5 illustre la nouvelle architecture système des PC depuis l'introduction de ce bus.

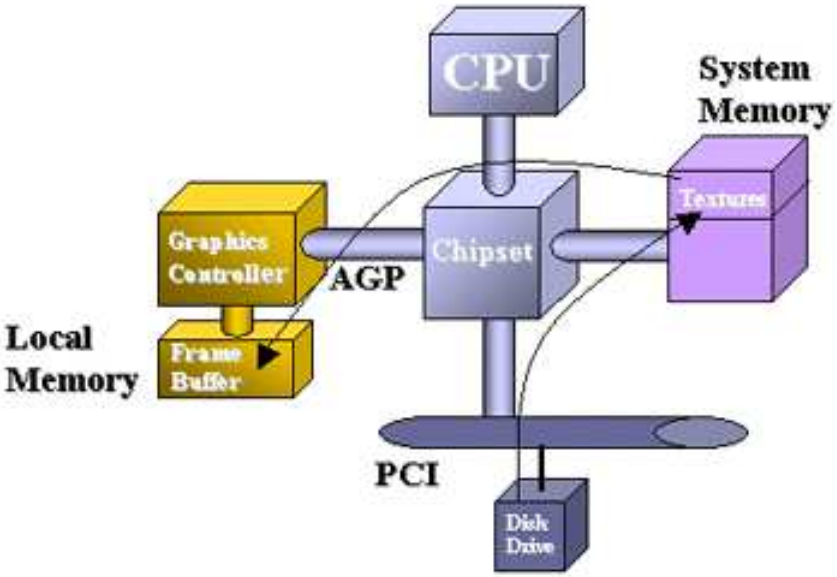


FIG. 4.5 – Architecture système d'un ordinateur comportant le bus AGP. Image reprise d'un tutorial précédemment en ligne sur http://www.intel.com/.

Au fur et à mesure qu'étaient résolus les problèmes précédents, le goulot d'étranglement se déplaçait vers le processeur central. Par exemple en 1998, un processeur PentiumII fonctionnant à 266 MHz ne pouvait fournir que 750 ktriangles à un circuit Riva128 capable d'en traiter le double. Le premier processeur graphique GeForce [76] intègre à la fois le parcours géométrique et la prise en charge des transformations et calculs d'illuminations de ces géométries dans le même circuit.

Enfin, ce tableau montre la rapidité d'évolution de ce secteur particulier des semi-conducteurs dont les performances bénéficient, outre l'évolution des technologies et des fréquences de fonctionnement, de l'emploi de solutions brutales comme le doublement du pipeline de traitement pixel ou le doublement de la largeur du bus d'interface mémoire.

4.1.2.3 Consoles graphiques de salon

Le second débouché des circuits pour le grand public se concentre sur les consoles graphiques de salon où quatre sociétés se sont partagé le marché.

Sega

Datant de 1999, la console Dreamcast se compose d'un processeur RISC Hitachi SH-4 32 bits disposant d'une extension pour le calcul matriciel 4×4 en flottant 64 bits fonctionnant à 200 MHz et d'un accélérateur graphique NEC PowerVR 128 bits à 100 MHz. Issu du Videologic PCX1, ce circuit effectue un rendu par tuiles et est capable de traiter 3 Mtriangles par seconde. Le succès de cette console démontre les possibilités commerciales du rendu par tuiles.

Nintendo

La dernière console graphique de la société Nintendo, dénommée GameCube, comporte un processeur IBM PowerPC fonctionnant à 485 MHz et un processeur graphique ATI/NEC cadencé à 166 MHz. Ce système délivre une puissance de calcul pouvant traiter de 6 à 12 Mtriangles par seconde.

Sony

Commercialisée à partir de 1995, la première console PlayStation de Sony se composait d'un processeur MIPS 32 bits R3000 fonctionnant à 33 MHz. Associés à ce processeur, un premier circuit spécifique de produit matriciel fonctionnant à 66 MHz pour les transformations géométriques et un second pour le rendu de polygones permettaient l'affichage de 180 ktriangles avec placage de texture et illumination de Gouraud.

La seconde version de cette console [78] se compose de deux circuits spécifiques l'*Emotion Engine* [79, 80, 81, 82] et le *Graphics Synthesizer* [83]. La figure 4.6 présente l'architecture système complète de cette console : le processeur d'entrées-sorties contient également un processeur à 34 MHz assurant une compatibilité ascendante entre les versions de cette console.

L'*Emotion Engine* fonctionnant à la fréquence de 300 MHz est constitué autour d'un cœur RISC MIPS R4000 128 bits avec deux unités entières 64 bits et une unité de traitement flottant simple précision. Deux unités vectorielles VU *(Vector Unit)*, organisées en VLIW *(Very Long Instruction Word)*, dont la première est directement connectée au processeur, sont dédiées aux calculs flottants nécessaires à la partie géométrique (capable de traiter des surfaces de Bézier plus complexes que les triangles). La première de ces unités contient 4 multiplieurs-accumulateurs flottants et 1 division, la seconde respectivement 5 et 1. Enfin, l'IPU (unité de traitement d'image) assure la décompression de séquences MPEG-2 : cette console permet aussi bien l'exécution de jeux que la lecture de DVD.

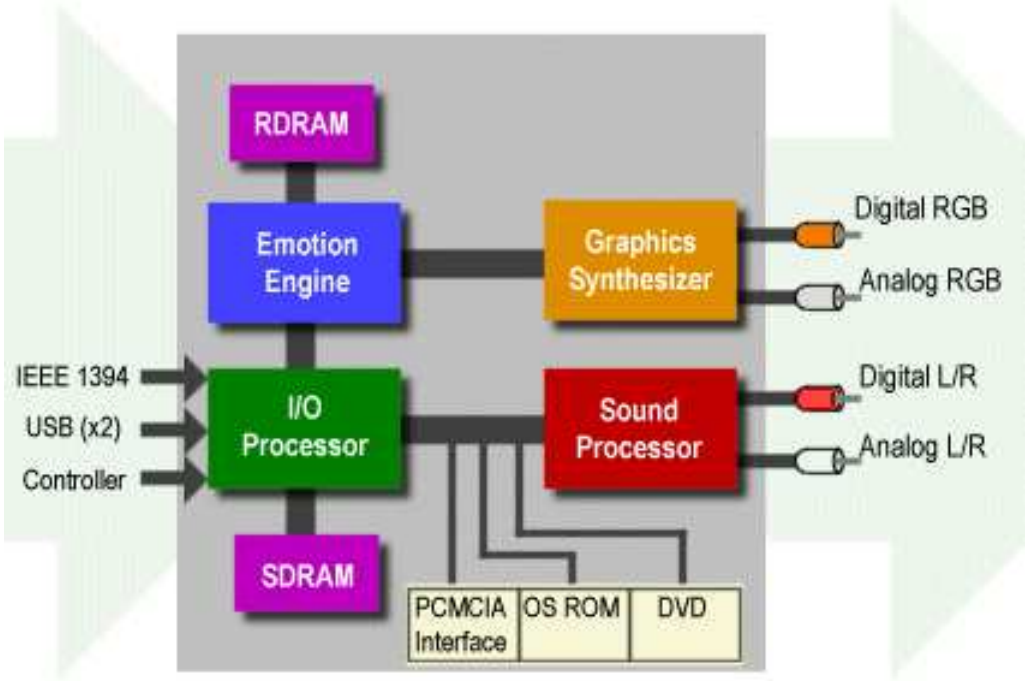


FIG. 4.6 – Architecture système de la console PlayStation2.

Les géométries sont ensuite délivrées au *Graphics Synthesizer* (figure 4.7) fonctionnant à 150 MHz. Un premier bloc matériel effectue à la fois l'initialisation des coefficients de parcours et le calcul concurrent pour 8 ou 16 pixels des différentes valeurs de couleurs, de profondeur et de coordonnées de textures nécessaires pour les traitements pixels suivants. Ceux-ci sont effectués en parallèle par 16 pipelines. Les échanges mémoires sont facilités par l'intégration dans le même circuit des mémoires images et de profondeur ainsi que des données de texture. Cependant, les 16 pipelines requièrent un premier bus de 512 bits pour les accès à la texture et deux bus (lecture et écriture) de 1 024 bits pour les accès aux mémoires image et de profondeur, chaque donnée étant conservée sur 32 bits.

Microsoft

Développée en partenariat avec Intel et *n*VIDIA, la X-box [84] intègre un processeur Intel Celeron à 733 MHz et un processeur graphique dérivé du GeForce3 de *n*VIDIA fonctionnant à 233 MHz. Comparable à un ordinateur personnel dédié aux loisirs, ce système est capable de traiter 116,5 Mtriangles par seconde. Enfin, ce système permet également la lecture de DVD vidéo.

4.2 Architectures pour la vidéo

Contrairement aux architectures de synthèse graphique, les architectures de rendu vidéo ne sont généralement pas destinées à être vendues en tant que telles mais à être intégrées à l'intérieur de systèmes le plus souvent propriétaires (comme des téléviseurs).

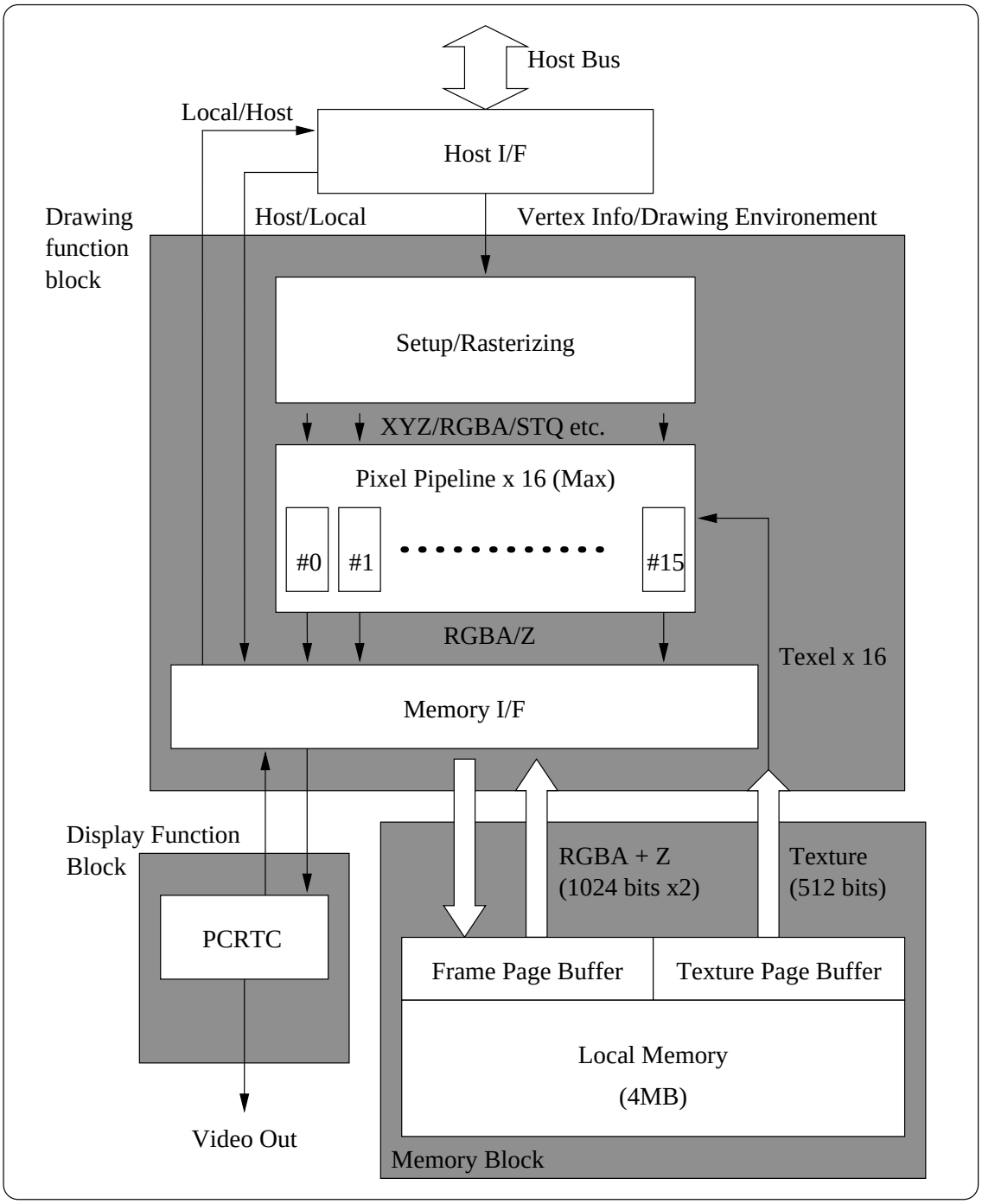


FIG. 4.7 – Architecture interne du *graphic synthesizer*.

Il est alors délicat d'obtenir des informations les concernant. Les travaux n'en sont pas moins nombreux comme l'attestent des revues récentes du domaine [85]. Nous allons effectuer également une revue d'architectures selon l'approche globale de celles-ci en soulignant les domaines d'applications privilégiés de chacune d'entre elles.

4.2.1 Architectures dédiées et reconfigurables

La normalisation du codage et décodage vidéo a spécifié un certain nombre de fonctions. Ces fonctions désormais figées ont donné lieu à l'implémentation d'opérateurs dédiés pour celles-ci comme les fonctions de transformation en bloc (DCT et DCT inverse) [86] ou la compensation de mouvement [87, 88]. Bien que ces fonctions puissent être partagées entre plusieurs normes, la flexibilité de ces blocs dédiés est évidemment réduite.

En ce qui concerne les architectures reconfigurables, elles autorisent une plus grande flexibilité d'utilisation comme leur nom l'indique. Nous entendons par architecture reconfigurable un système matériel capable de s'adapter à son contexte d'exécution sans, toutefois, qu'il s'agisse d'extensibilité. Cependant, l'exemple de filtres linéaires de réechantillonnage [89, 34] dont les coefficients sont paramétrables montre que de telles architectures restent spécifiques au domaine pour lequel elles sont conçues.

4.2.2 Architectures programmables spécialisées

Des architectures de types DSP *(Digital Signal Processing)* et VLIW [90] se prêtent particulièrement bien à la nature des traitements vidéo (une revue de processeurs multimédia est effectuée par M^{lle} Miró dans sa thèse [29]). En effet, l'itération sur un large ensemble de données d'opérations arithmétiques relativement simples (addition, multiplication) avec peu de tests conditionnels permet la mise en place pour un large spectre d'applications de structures en flot de données : Imagine [91, 92] est une telle architecture.

Imagine est un coprocesseur SIMD composé de huit groupements arithmétiques exécutant la même instruction longue (VLIW). L'architecture interne d'un groupement est reproduite figure 4.8. La programmabilité de cette architecture lui permet d'effectuer également de la synthèse d'images [93, 94].

Cependant, ces architectures et les systèmes bâtis autour de ces cœurs de processeur [95] nécessitent des débits mémoires importants qui peuvent se révéler prohibitifs pour des systèmes embarqués.

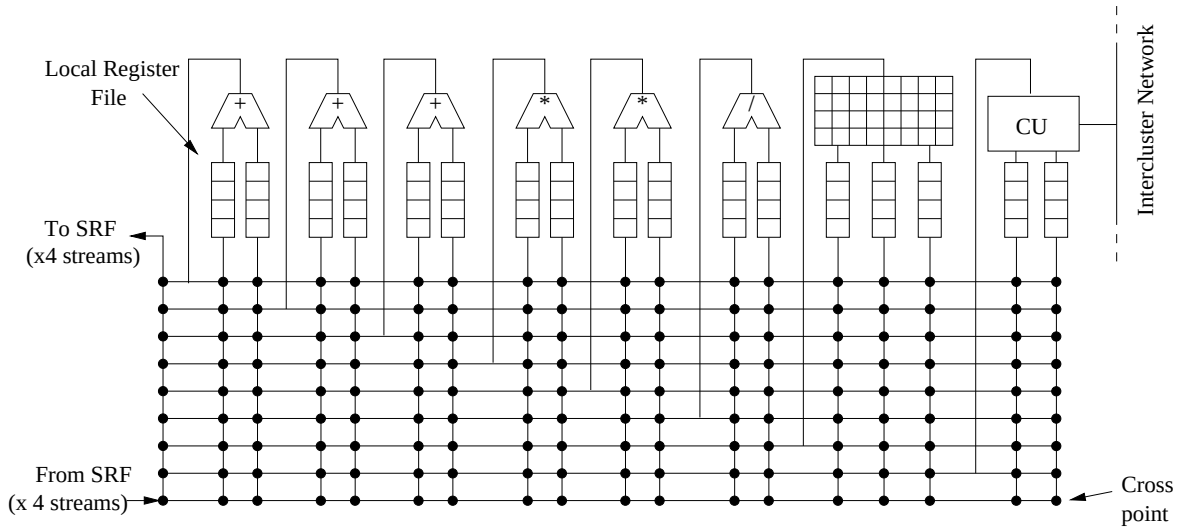


FIG. 4.8 – Organisation d'une unité interne de Imagine.

4.3 Architectures pour les systèmes embarqués

4.3.1 Architectures orientées traitement vidéo

Les architectures dédiées aux traitements vidéo pour les systèmes embarqués se sont focalisées plus particulièrement sur les fonctions de décompression (DCT inverse) de flux MPEG. Ainsi, un processeur spécifique organisé en SIMD permettait le décodage de flux MPEG-2 pour la télévision pour une consommation de 1,2 W [96]. La programmabilité de ce processeur le rendait susceptible de supporter d'autres applications comme le rendu de graphisme.

Plus récemment, les systèmes intègrent les fonctions de compression MPEG-4 pour la vidéophonie au format QCIF pour 1 cm^2 et une consommation de 240 mW en technologie 0,25 μm [97]. Désormais, la compression/décompression de flux vidéo au format CIF combinant également des fonctions de post-traitement ne consomment plus que 160 mW pour une surface de 43 mm^2 en technologie 0,13 μm [98].

Enfin, nous pouvons également noter l'existence de solutions uniquement fondées sur des processeurs généralistes [99, 100] qui, cependant, restent limitées par les puissances de calcul disponibles.

4.3.2 Architectures orientées traitement graphique

Ce domaine est certainement le plus récent, mais connaît aujourd'hui une évolution des plus rapides. Sur le terrain fertile du graphisme pour systèmes embarqués, de nombreuses jeunes pousses ont vu le jour. En particulier, la société finnoise Bitboys produit

une gamme de circuits propriétaires dénommée *acceleon* [101] qui assure aussi bien les traitements des graphismes 2D vectoriels (SVG [102] ou MacromediaFlash [103]) que les traitements 3D répondant aux exigences du nouveau standard OpenGL-ES pour les systèmes embarqués [3].

Associant ses cœurs de processeur et des cœurs de rendu graphique 3D PowerVR quelque peu simplifiés par rapport aux cartes KyroII, la société ARM propose une flexibilité de choix de systèmes complets pour applications portables [104, 105]. Le tableau 4.2 présente les performances de ces cœurs graphiques.

	Cœur complet	**Cœur allégé**
Portes	660 k	$<$ 300 k
Puissance	1,3 mW/MHz	0,5 mW/MHz
Horloge	80-120 MHz	80-120 MHz
Triangles	2-4 Mtriangle/s	0,5-1,5 Mtriangle/s
Pixels	300-500 Mpixel/s	100-250 Mpixel/s
Écran	VGA (640×480)	QVGA (320×240)
Option VGP *(Vertex Graphic Processor)*		
Portes	210 k	150 k
Perf.	300-500 Mflops/s	75-150 Mflops/s

Tab. 4.2 – Performances des cœurs accélérateurs 3D pour ARM en technologie 0,13 μm.

Face à ses solutions, ATI a annoncé début janvier 2004 [106] une architecture dédiée pour la téléphonie mobile. Cette architecture embarque des fonctions de graphisme 2D et 3D répondant également à l'API OpenGL-ES, un codeur-décodeur JPEG pour images fixes et un décodeur MPEG-4 vidéo. Un diagramme en bloc repris figure 4.9 montre cependant que ces fonctions constituent toujours des blocs distincts.

Cependant, dès 2001, un système complet intégrant un processeur, des blocs accélérateurs pour le décodage MPEG-4 et le rendu 3D était disponible [107, 108, 109]. Ne consommant que 160 mW (40 mW pour les fonctions de décodage MPEG-4 et 120 mW pour le rendu 3D), ce système n'assurait que la compensation de mouvement pour le décodage d'une image QCIF (176×144). D'une taille de 84 mm^2, ce système intègre une mémoire image n'autorisant pas de flexibilité au niveau des tailles des images.

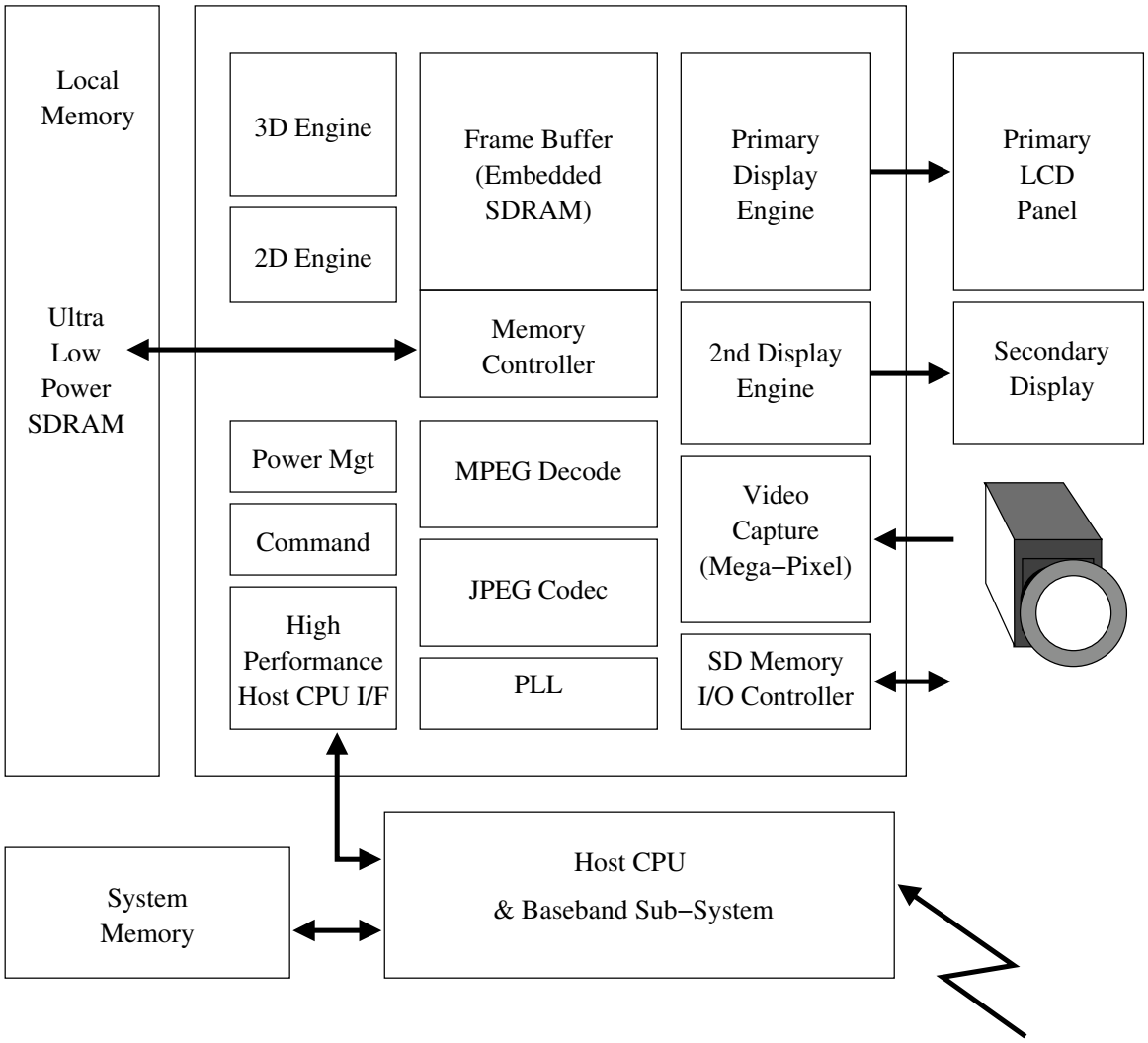


FIG. 4.9 – Diagramme en bloc de l'Imageon 2300.

Chapitre 5

Algorithmes et architectures : conclusion

La présentation que nous venons de réaliser fait apparaître plusieurs remarques. Tout d'abord, nous avons exposé les différents algorithmes employés tant en synthèse d'images graphiques qu'en traitement vidéo. Nous avons souligné les hétérogénéités des contraintes de chaque domaine ; nous avons aussi constaté un certain nombre de similitudes au niveau des opérations exécutées : filtrage, organisation des traitements et opérations au niveau pixel, etc. La figure 5.1 esquisse ce que serait une architecture unifiée de traitement graphique et vidéo avec la combinaison complète des traitements pixels et la mise en commun de blocs lors des traitements amonts comme, par exemple, des fonctionnalités de décompression d'images.

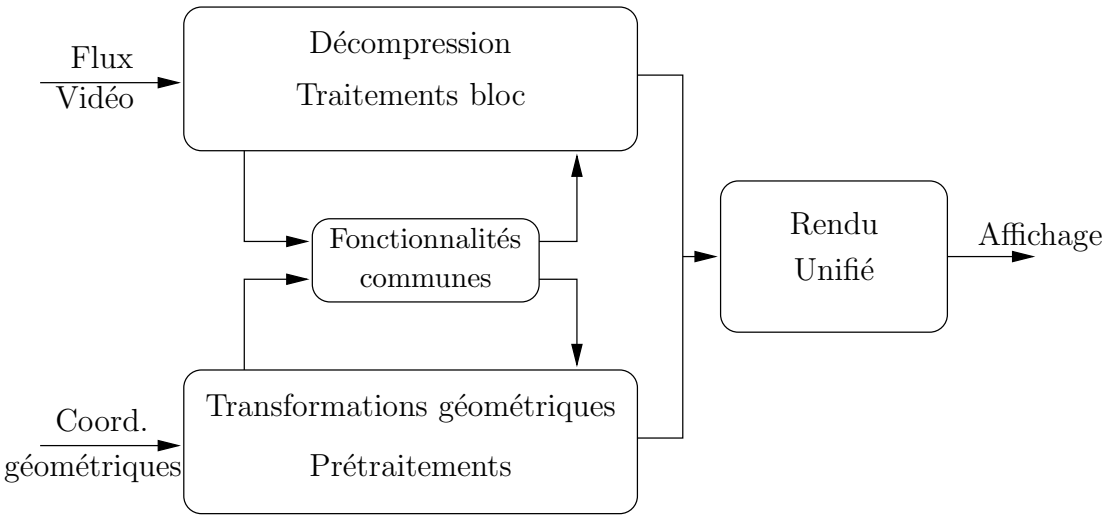


FIG. 5.1 – Pipeline unifié des traitements vidéo et 3D.

Ensuite, nous avons considéré quelques architectures tant pour la synthèse graphique que pour les traitements vidéo. Nous avons pu remarquer alors que, si des architectures optimisées l'étaient pour l'un ou l'autre des domaines, peu de systèmes cherchaient à unifier véritablement les deux pipelines de traitements. Cependant, le tableau 4.1 nous a fait constater la rapidité d'évolution de ce secteur et, à l'heure où nous écrivons ces lignes, les annonces concernant des architectures pour les systèmes embarqués se mul-

tiplient accentuant l'actualité de nos travaux.

Concernant les questions d'extensibilité et de bande-passante mémoire, nous avons pu observer à travers les systèmes de synthèse graphique que l'un allait au détriment de l'autre. L'extensibilité est garantie par le parallélisme intrinsèque des traitements et par la duplication des opérateurs. Les communications entre opérateurs et avec la mémoire impliquent des réseaux d'interconnexion aux bandes-passantes importantes (jusqu'à plusieurs dizaines de gigaoctets par seconde). La bande-passante apparaît plutôt comme un facteur limitatif des performances que comme un objectif à minimiser. De plus, les solutions massivement parallèles posent le problème de la répartition de charge entre les différents processeurs élémentaires.

En conclusion de cette partie, nous pouvons relever l'absence d'architecture matérielle optimisée pour effectuer à la fois du rendu de type vidéo et de la synthèse d'images répondant à des critères compatibles avec les contraintes des systèmes embarqués.

Quatrième partie

Propositions pour un rendu unifié

Chapitre 6

Représentation des données

Nous nous proposons de débuter cette partie sur les conséquences et impacts théoriques de la convergence de la synthèse d'image et de la vidéo par une étude de la représentation des données manipulées par chaque domaine. Nous commencerons par un exposé de leurs différentes représentations. Nous effectuerons par la suite une étude des implications pour un matériel spécifique assurant un traitement sur ces données.

6.1 Présentation

Les données manipulées auxquelles nous nous intéressons ici concernent principalement les informations relatives aux images : images vidéo transmises, décodées, filtrées pour le premier domaine de la vidéo et images naturelles ou non, stockées, plaquées sur des géométries pour le second domaine.

6.1.1 Espaces de couleurs

La première différence flagrante entre les domaines de la vidéo et de la synthèse d'images provient des espaces de couleurs généralement utilisés. La synthèse vidéo utilise un domaine RVB : une couleur est décrite par la valeur de trois composantes fondamentales Rouge, Vert et Bleu. L'emploi d'un domaine colorimétrique à trois dimensions remonte au travaux menés au milieu du XIX^{e} par Maxwell, normalisés par ceux de la Commission Internationale de l'Éclairage (C.I.E.) en 1931. Ces trois couleurs correspondent aux sensibilités maximales des capteurs de couleurs présents sur la rétine humaine : les cônes. Ceux-ci sont de trois types, chacun sensible à une couleur particulière (bleu-violet de longueur d'onde $\lambda = 420$ nm, vert $\lambda = 530$ nm et jaune-rouge $\lambda = 565$ nm).

La rétine est également constituée de bâtonnets sensibles pour leur part à la luminance Y. La luminance pouvant s'exprimer comme combinaison linéaire des trois couleurs RVB, une bande-passante réduite par trois explique l'apparition historique de la télévision noir et blanc avant celle de la télévision couleur. Pour des raisons de compatibilités ascendantes du matériel lors de l'introduction de la couleur, la luminance fut

complétée par des informations de chrominance notées Cr et Cb (notées également U et V). Avant l'affichage, les données de luminance et de chrominance sont transformées en données RVB pour répondre aux stimuli attendus par notre rétine.

Ainsi, la vidéo emploie un espace de couleurs YCrCb tandis que la synthèse graphique manipule ses données directement dans un espace RVB. La recommandation de l'UIT BT.601-5 [110] indique les transformations complètes entre les espaces RVB et YCrCb ; elles sont reprises (de manière simplifiée) équations (6.1).

$$\begin{cases} \mathrm{Y} = 0{,}299\mathrm{R} + 0{,}587\mathrm{V} + 0{,}114\mathrm{B} \\ \mathrm{Cr} = \frac{0{,}5}{0{,}701}(\mathrm{R} - \mathrm{Y}) = 0{,}5\mathrm{R} - 0{,}419\mathrm{V} - 0{,}081\mathrm{B} \\ \mathrm{Cb} = \frac{0{,}5}{0{,}886}(\mathrm{B} - \mathrm{Y}) = -0{,}169\mathrm{R} - 0{,}331\mathrm{V} + 0{,}5\mathrm{B} \end{cases} \tag{6.1}$$

Cependant, la définition de Y varie selon les recommandations : pour la haute-définition, la norme SMPTE-240M (1920×1035) définit la luminance par $\mathrm{Y} = 0{,}212\mathrm{R} + 0{,}701\mathrm{V} + 0{,}087\mathrm{B}$. Les normes SMPTE-274M (1920×1080) et SMPTE-296M (1280×720) la définissent par $\mathrm{Y} = 0{,}212\ 6\mathrm{R} + 0{,}715\ 2\mathrm{V} + 0{,}072\ 2\mathrm{B}$ [111]. Pour un complément d'informations concernant ces différents espaces de couleurs et leurs différentes interactions, le lecteur intéressé pourra se reporter au chapitre 3 du livre de M. Jack [112].

6.1.2 Domaines de variations

Un deuxième aspect de la représentation des données concerne les domaines de variations de celles-ci. Tout d'abord, le codage numérique sous forme de signaux YCrCb peut donner aux signaux des valeurs qui s'étendent sur un intervalle plus grand que pour les signaux RVB. Tout en étant individuellement valable, YCrCb peuvent donner des valeurs hors gamme une fois convertis en signaux RVB. Pour l'éviter, il convient de limiter les signaux sous forme YCrCb plutôt que sous forme RVB [110].

Ensuite, si un codage binaire sur un octet est généralement employé pour quantifier les 256 niveaux d'une composante, certains niveaux sont réservés. Ainsi, en vidéo, les niveaux 0 et 255 sont réservés pour des données de synchronisation, le signal de luminance ne doit occuper que 220 de ces niveaux et le noir occupe le niveau 16. Le domaine de variation est alors restreint à l'intervalle [16..235]. Pour les signaux de chrominance, seuls 225 niveaux sont utilisables.

Enfin, la résolution en nombre de bits de chacune des composantes limite le nombre de couleurs représentables. En synthèse graphique, une API comme OpenGL fonde son espace de couleur RVB sur des valeurs normalisées des couleurs : le triplet (0,0 : 0,0 : 0,0) représente le noir, le triplet (1,0 : 1,0 : 1,0) le blanc. Ces triplets sont alors représentés à l'aide de données dont le résolution peut prendre différentes valeurs. Ainsi, avec un octet par composantes et en ajoutant une composante de transparence, un pixel est codé sur 32 bits. Un codage sur 16 bits peut être uniformément réparti avec 4 bits par composantes de couleur et de transparence, peut correspondre à un unique bit d'opacité et 5 bits par composantes de couleurs ou, en raison de la forte sensibilité à la composante verte, peut représenter 5 bits pour le rouge et le bleu pour 6 bits pour la composante vert. Un codage

sur un unique octet peut enfin être décomposé par 3 bits pour les composantes rouge et verte et 2 bits pour le bleu.

6.1.3 Échantillonnage

Le troisième aspect que nous évoquerons concerne l'échantillonnage des données de texture en synthèse graphique et en vidéo.

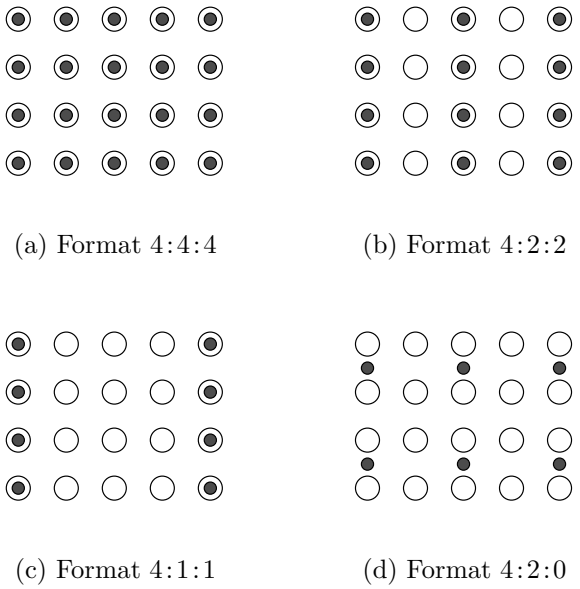
6.1.3.1 Échantillonnage spatial

Une fois encore, les domaines vidéo et graphiques diffèrent par les échantillonnages des données rencontrés. En synthèse graphique, un pixel correspond à l'échantillonnage des trois composantes Rouge, Vert et Bleu. Dans le cas de la vidéo, chaque pixel correspond à un échantillon de luminance. Cependant, l'échantillonnage de la chrominance est variable selon le format. Dans le format $4:4:4$, les échantillonnages de luminance et de chrominance s'effectuent sur un quadrillage orthogonal coïncidant entre eux (figure 6.1(a)). Le format $4:2:2$ échantillonne la luminance à l'aide de la même grille d'échantillonnage que précédemment, mais la chrominance est sous-échantillonnée d'un facteur deux : les échantillons des signaux Cr et Cb coïncident avec les échantillons impairs du signal Y dans chaque ligne (figure 6.1(b)). Les formats $4:1:1$ et $4:2:0$ sous-échantillonnent encore d'un facteur deux la chrominance. Pour le premier, les signaux Cr et Cb ne coïncident plus qu'avec un échantillon de luminance sur quatre toutes les lignes (figure 6.1(c)). Dans le dernier cas, le sous-échantillonnage est à la fois vertical et horizontal. La figure 6.1(d) illustre le cas où les échantillons de chrominance coïncident entre eux en s'entrelaçant toutes les deux lignes avec les échantillons de luminance. Dans le standard DV ($4:2:0$ *co-sited*), les échantillons Cr et Cb sont cette fois-ci entrelacés chaque ligne indépendamment (Y-Cr pour une ligne, Y-Cb pour l'autre ligne).

La télévision haute définition exploite de nombreuses propriétés de la perception visuelle [113]. En particulier, les propriétés intégratrices et de faible pouvoir de résolution vis-à-vis de la couleur justifient le sous-échantillonnage chromatique effectué.

6.1.3.2 Échantillonnage temporel

Une dernière spécificité du domaine vidéo provient de l'entrelacement temporel des trames composants une image comme nous l'avons déjà mentionné paragraphe 3.2.1. Dans le cas d'images progressives, ces deux trames ont été capturées au même instant t. Plus courantes sont les images entrelacées : dans ce cas, la première trame de l'image i est échantillonnée à l'instant t_i tandis que la seconde l'est à l'instant $t_i + \frac{t_{i+1}-t_i}{2}$. Les conséquences d'un échantillonnage à deux instants différents imposent des contraintes particulières pour le rééchantillonnage d'images : le cas d'un arrêt sur image nécessite ainsi soit une duplication de la trame courante et une perte de résolution verticale d'un facteur deux, soit un rééchantillonnage et un filtrage prenant en compte plusieurs trames consécutives.



(a) Format 4:4:4 (b) Format 4:2:2

(c) Format 4:1:1 (d) Format 4:2:0

FIG. 6.1 – Différents formats d'échantillonnage de luminance et de chrominance. Les cercles représentent les échantillons de luminance, les disques, les échantillons de chrominances.

6.1.4 Résumé

En résumé, le tableau 6.1 rappelle les différences propres aux données manipulées par les deux domaines étudiés.

	Synthèse 3D	Vidéo
Couleurs	RVB	YCrCb
Variations	Pleine résolution	Variations limitées
Format	RVB/pixel Progressif	Nombreux formats Progressif/Entrelacé Sous-échantillonnage chromatique

TAB. 6.1 – Différences entre les domaines vidéo et graphiques.

L'unification des traitements et la convergence des domaines de la synthèse graphique et de la vidéo passent par un support matériel qui autorise un niveau d'abstraction suffisant pour rendre transparent les différences notées.

6.2 Support matériel des traitements

Nous présentons deux stratégies afin d'assurer qu'un système matériel pour mobile supporte à la fois de la vidéo et de la synthèse d'images.

6.2.1 Espace de couleur unifié

La première stratégie consiste à choisir un espace de couleur et un format associé unique pour tous les traitements internes au système. Le choix de cet espace unique peut être déterminé simplement en fonction du format accepté par l'écran d'affichage. Le mode naturel de fonctionnement des écrans LCD actuels s'approchant d'un format progressif en RVB, ce format semble approprié. Un convertisseur en entrée du processus complet de traitement effectue alors la conversion des données lorsque celles-ci le nécessitent. La figure 6.2 suggère un tel système présentant un convertisseur de format en entrée de la chaîne de traitement.

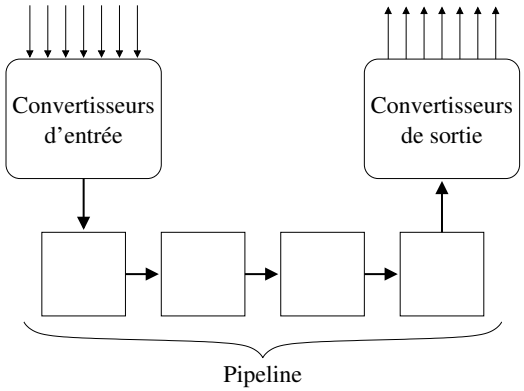


FIG. 6.2 – Système employant un format unique dans sa chaîne de traitements.

L'une des contraintes d'un tel système est d'assurer l'équivalence des traitements après changement d'espace de couleur. La linéarité des opérations de transcodage d'un espace à un autre et des opérations de filtrage effectuées lors des traitement favorise l'équivalence des traitements : dans le cas d'un filtrage bilinéaire, cette équivalence est acquise.

Bien que, dans le cas de la téléphonie mobile, des réceptions de signaux entrelacés soient peu probables, le convertisseur de format peut procurer l'interface nécessaire au passage en format progressif. Cependant, ces conversions sont plus efficaces si elles sont étroitement couplées avec la décompression du flux vidéo. En effet, en tirant profit des vecteurs de mouvements, une interpolation fine exploitant le matériel présent permet la reconstruction d'une image complète de qualité.

La possibilité pour un convertisseur programmable d'être reconfiguré en fonction du format du signal d'entrée garantit la flexibilité d'une telle architecture. De plus, cela permet d'envisager des conversions non triviales comme des modifications de luminosité ou des saturations en couleurs comme des effets de cyanotypes.

6.2.2 Espaces de couleur multiples

Afin de ne pas augmenter la complexité matérielle d'un système par l'ajout d'un convertisseur tout en conservant la garantie de flexibilité, un système doit être alors capable d'adapter les traitements en fonction des formats d'entrée et de sortie. Désormais,

les performances, les formats des données et les puissances de calcul deviennent interdépendants.

Notons t le temps de traitement d'un pixel, p la puissance de calcul disponible par unité de temps et N le nombre total de pixels traités pour une image complète. La puissance de calcul complète est alors $p\,t\,N$; quant aux performances, elles seront exprimées par le temps de traitement d'une image $t\,N$.

6.2.2.1 Traitement pixel constant

En conservant un temps de traitement pixel t constant, la puissance de calcul dépend du format des données à traiter en vue de maintenir le temps de traitement d'une image constant. En effet, il est nécessaire de traiter toutes les composantes d'un pixel durant le même temps t : en format RVB, trois composantes doivent être traitées par pixel ; en noir et blanc, seul Y doit l'être tandis que pour les autres formats vidéo le nombre moyen de composantes par pixel oscille entre 1,5 et 3. Afin d'assurer des performances constantes selon les différents formats d'entrées, la puissance de calcul doit alors être dimensionnée selon le pire cas engendrant ainsi une sous-utilisation des ressources dans un cas moyen.

6.2.2.2 Puissance de calcul constante

En conservant une puissance constante unitaire identique tout au long du traitement d'une image, les performances sont alors liées au format des données. En effet, toutes les composantes d'un pixel devant être traitées à l'aide d'opérateurs dont la puissance de calcul p est fixe, le temps de traitement t dépend alors du nombre de composantes. En conservant toute la puissance de calcul disponible, il est alors possible d'altérer la fréquence de fonctionnement du système en fonction du nombre de composantes par pixel.

6.3 Résumé

En définitive, nous venons de voir un nouvel aspect des différences entre les deux domaines de la vidéo et de la synthèse graphique au travers des données que manipule chacun. La convergence des données manipulées est ainsi un passage obligé : une API comme OpenML (*Open Media Library* [114]) dont le but est de proposer une interface pour le développement d'applications audio, vidéo et graphiques (OpenGL est inclus comme interface pour cette partie) souligne cette nécessité en définissant des extensions graphiques pour le support de format YCrCb. Nous avons ensuite constaté, à l'aide de considérations générales, comment, du point de vue matériel, il est possible d'effectuer à l'intérieur d'une même architecture les rendus vidéo et graphique.

À partir des remarques que nous avons explorées, nous pouvons alors justifier les choix qui conduiront dans la partie V à la mise en place d'un traitement effectuant

les traitements à l'aide d'une puissance de calcul constante. Cette puissance constante conduit à un temps constant par composante de couleur des pixels. Ce choix permet d'assurer *a priori*, dans le cas d'affichage vidéo, un temps de traitement connu et, par conséquent, une part d'extensibilité selon le format de l'image.

Chapitre 7

Le rendu d'images par tuiles : modélisation et optimisation

Ce chapitre traite de l'organisation du rendu d'images par tuiles et de ses conséquences en terme de complexité de traitement puis de bande-passante mémoire. Nous supposerons dans ce chapitre qu'une image est constituée d'objets (primitives vidéo ou primitives graphiques). Dans le cas d'objets graphiques, ceux-ci sont constitués de maillages de triangles sur lesquels nous appliquons une texture par transformation affine. Cette dernière est définie par les positions d'un triangle de référence dans la texture et le triangle à représenter dans l'image. Rappelons que, conformément au modèle d'architecture présenté dans la figure 2.1 chapitre 2, les données de texture sont contenues dans une mémoire unique extérieure à notre opérateur.

7.1 Introduction

Nous avons introduit dans le paragraphe 3.3.2.3 le rendu par tuile d'une image. Nous complétons cette introduction à l'aide de données issues d'une étude que nous avons menée sur la complexité des scènes 3D de jeux commerciaux [16]. Cette étude dont nous reproduisons de larges extraits en annexe A recherchait à déterminer les caractéristiques d'un cœur de rendu 3D unifié avec les traitements vidéo pour des récepteurs de télévision. Outre une familiarisation avec le fonctionnement d'une application graphique fondée sur l'analyse des appels aux API graphiques des applications, nous avons également pu étudier les implications en terme de bande-passante du rendu par tuiles. Nous reprenons ici les conclusions que nous avions faites concernant l'activité du *z-buffer* : cf. paragraphe A.3.1 page 210.

L'élimination des surfaces cachées par la technique du *z-buffer* peut conduire à une consommation importante de la bande-passante. Dans le cas de la scène comme celle présentée au chapitre 3 (figure 3.1), scène que nous nommerons dans la suite de la discussion « Vampire », la complexité en profondeur des pixels peut être modélisée par une loi gaussienne *Lg* de moyenne 2,34 et d'écart-type 1,10. Nous supposerons également une équirépartition de l'ordre de génération des pixels de l'écran (ordre selon la valeur de la coordonnée de profondeur z).

L'algorithme du *z-buffer* associé à une mémoire d'accumulation (*A-buffer*) engendre une lecture de la coordonnée z par pixel et, en fonction du résultat de la comparaison avec la coordonnée courante (c.-à-d. l'ordre de génération des pixels), une lecture du pixel déjà écrit pour effectuer un mélange des couleurs et les réécritures du pixel et de la nouvelle coordonnée z. Le nombre de lectures $\mathrm{N_{l.}}$ ne dépend que de la complexité moyenne de la scène et de la taille de celle-ci ($\mathrm{N_{l.}} \approx 2{,}34 \times 640 \times 480$). Le nombre de comparaisons $\mathrm{N_{c.}}$ impliquant une réécriture après mélange du pixel est donné par l'expression (7.1) où d représente la complexité en profondeur.

$$\mathrm{N_{c.}} = \sum_{d=1}^{\infty} \left(Lg(d) \sum_{i=1}^{d} \frac{1}{i} \right) \tag{7.1}$$

La coordonnée z ainsi que les pixels sont supposés codés sur quatre octets chacun. Les lectures des seules coordonnées z avant comparaison nécessitent une bande-passante, à 30 images pas seconde, de $\mathrm{N_{l.}} \times 4 \times 30$ B/s, soit environ 82 MB/s. En appliquant la formule (7.1) pour le calcul de $\mathrm{N_{c.}}$, les comparaisons impliquant un mélange et une écriture nécessitent une bande-passante d'environ $\mathrm{N_{c.}} \times 4 \times 30 \approx 65$ MB/s pour chacune des opérations de lecture du pixel précédent, d'écriture des nouveaux pixels et d'écriture des coordonnées z. En dehors de toute optimisation des accès, la bande-passante totale nécessaire à la réalisation du rendu est donc de $3 \times 65 + 82 = 277$ MB/s. La localité des traitements n'étant pas assurée, le rendement d'un cache ne permettra pas de réduire significativement cette bande-passante. À l'extrême, une architecture tuile, où le principe de localité est assuré, autorise les comparaisons et les mélanges à partir de mémoires locales. Seule l'écriture finale de la tuile résultante est réalisée et ne nécessite qu'une bande-passante de $640 \times 480 \times 4 \times 30 \approx 35$ MB/s.

La réduction de l'activité de la mémoire et donc de la bande-passante d'un système est un des axes majeurs de réduction de la consommation [115]. La problématique est d'un large intérêt dans le contexte de systèmes embarqués. Le raisonnement précédent est donc un argument de premier plan en faveur d'un rendu par tuile. En outre, nous avons déjà signalé l'extensibilité d'une architecture qui utilise ce mode de rendu.

Cependant, le rendu par tuile introduit des contraintes (déterminer quels triangles traiter, comment les traiter, etc.) dont les effets de bords peuvent être rédhibitoires. C'est pourquoi nous allons tenter de modéliser dans une première partie l'impact de ce type de rendu sur les puissances de calcul mises en jeu. Nous proposerons ensuite un mode de parcours qui a l'avantage de permettre une compatibilité entre un rendu purement vidéo et de la synthèse d'images. Nous réévaluerons alors les puissances de calculs mises en jeu. Dans une seconde partie, nous étudierons plus précisément la manière d'accéder aux données nécessaires au parcours proposé. Finalement nous montrerons comment il est possible de tirer profit des propriétés du rendu par tuiles pour approcher le comportement d'un cache parfait.

7.2 Puissances de calcul lors d'un traitement tuile

Faisant abstraction du processus de *z-buffer*, nous utiliserons les hypothèses simplificatrices suivantes :

- l'image est pavée de tuiles de taille carrée ;
- la scène est composée d'une unique primitive de forme carrée ;
- la primitive est constituée d'un maillage de triangles ;
- l'impact d'un triangle sur la scène est modélisé par son rectangle entourant supposé carré.

Le tableau 7.1 indique les notations que nous utiliserons par la suite.

Paramètre	Description
T_{Cal}	Temps de calcul d'une image
f_{Pix}	Fréquence de traitement des pixels
f_{Tri}	Fréquence de traitement des triangles
$L_{\text{T.}}$	Largeur d'une tuile
L_{P}	Largeur de la primitive
L_{Tri}	Largeur de la boîte entourante d'un triangle
$\text{N}_{\text{T.}}$	Nombre moyen de tuiles de la scène impactées par la primitive
N_{Tri}	Nombre moyen de triangles de la primitive impactant une tuile
$\lceil . \rceil$	Arrondi par excès : $\lceil x \rceil = n$ tel que $n \in \mathbb{N}$ et $n - 1 < x \leq n$

TAB. 7.1 – Notation pour l'étude de l'impact de la taille d'une tuile.

7.2.1 Définition d'un modèle de rendu idéal

Avant d'étudier le rendu par tuile, nous pouvons définir un modèle de rendu idéal disposant des ressources minimales (en terme de puissance de calcul) nécessaires à la réalisation d'un affichage de la scène en temps réel. L'architecture de référence est composée de deux processus indépendants séparés par une mémoire tampon de taille suffisante. La figure 7.1 illustre cette architecture.

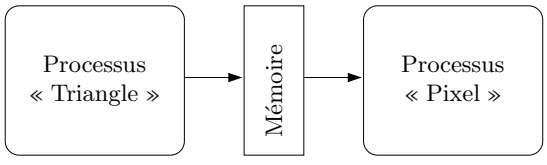


FIG. 7.1 – Processus de rendu d'une primitive. Un premier processus se charge de traiter les triangles composant la primitive, le second parcours tous les pixels de cette primitive.

Le premier processus qui ne traite que des triangles calcule tous les paramètres nécessaires à l'exécution du second processus. Le second processus effectue le parcours des pixels de la primitive et son temps de calcul est proportionnel à la surface de la primitive.

Si l'on suppose que la complexité des scènes ne change pas d'une image à l'autre, la mémoire tampon doit pouvoir contenir les paramètres de deux images consécutives.

Dans ces conditions, étant donné un temps de calcul d'une image T_{Cal_0}, nous pouvons déterminer les fréquences de traitement « Pixel » et « Triangle » idéales des deux processus :

- une fréquence de traitement « Pixel » telle que : $f_{\mathrm{Pix}_0} = \frac{L_{\mathrm{P}}^2}{T_{\mathrm{Cal}_0}}$;
- une fréquence de traitement « Triangle » telle que : $f_{\mathrm{Tri}_0} = \frac{L_{\mathrm{P}}^2}{L_{\mathrm{Tri}}^2 T_{\mathrm{Cal}_0}}$.

Ces fréquences de traitement correspondent à un cas idéal non réaliste. En effet, nous avons dimensionné la mémoire tampon pour contenir l'ensemble des paramètres d'une scène. Si nous prenons l'exemple de l'application étudiée en détail partie V (240 bits d'information pour les paramètres d'un triangle) et si nous considérons une scène du type de la scène « Vampire » modélisée par 4 000 triangles, la taille de la mémoire tampon atteint environ 2 Mbit, c.-à-d. une surface de silicium utilisée pour la mémorisation équivalente [116] à celle des cœurs 3D complets embarqués actuels, cœurs présentés tableau 4.2 page 70.

Nous supposons maintenant que le processus « Pixel » travaille tuile après tuile. Le processus « Triangle » est en charge de fournir pour chaque tuile l'ensemble des paramètres des triangles l'impactant. La taille de la mémoire tampon est alors dépendante de la tuile et non pas de la scène.

7.2.2 Taille d'une tuile et temps de traitement

La taille choisie pour les tuiles a un impact direct sur le recouvrement de multiples tuiles par un même triangle : soit ce triangle doit être retraité pour chaque tuile recouverte, soit des informations doivent être conservées d'une tuile à l'autre par le processus « Pixel ». Afin de limiter la taille de la mémoire locale dans laquelle nous conserverions ces informations et dans le souhait d'uniformiser le traitement de chaque triangle, nous faisons *a priori* le choix de retraiter chaque triangle pour chaque tuile recouverte. Nous voyons d'ores et déjà que le processus « Triangle » doit traiter un nombre efficace de triangles supérieur au nombre réel de triangles dans la scène. Afin de détailler le surcoût induit, nous introduisons de nouvelles hypothèses pour raffiner notre modèle. Nous comparons ensuite les résultats théoriques avec des résultats issus de cas réels.

7.2.2.1 Modélisation

Nous supposons désormais que le processus « Pixel » effectue un balayage systématique de tous les pixels de la tuile. Cette hypothèse homogénéise les traitements de chaque pixel et renforce l'indépendance entre les deux processus à l'aide d'un parcours indépendant des objets impactant la tuile.

La fréquence « Pixel » dépend désormais du temps de traitement total du processus pixel, du nombre de tuiles traitées et de leur surface comme l'indique l'équation (7.2). La fréquence « Triangle » est, quant à elle, conditionnée par le temps de traitement des

triangles, le nombre de tuiles et le nombre de triangles les impactant ; l'équation (7.3) résume cette dépendance.

$$f_{\text{Pix}} = \text{N}_{\text{T.}} \frac{L_{\text{T.}}^2}{T_{\text{Cal}}} \tag{7.2}$$

$$f_{\text{Tri}} = \text{N}_{\text{T.}} \frac{\text{N}_{\text{Tri}}}{T_{\text{Cal}}} \tag{7.3}$$

Les nombres de tuiles impactées par la primitive $\text{N}_{\text{T.}}$ ainsi que de triangles par tuile N_{Tri} peuvent être exprimés à l'aide d'une formulation bien connue du facteur de recouvrement [53, 64, 117]. Cette formulation dérive de l'hypothèse de l'équirépartition des positions possibles d'un carré se superposant à une grille (de tuiles par exemple). La figure 7.2 montre comment obtenir l'expression de ce facteur de recouvrement.

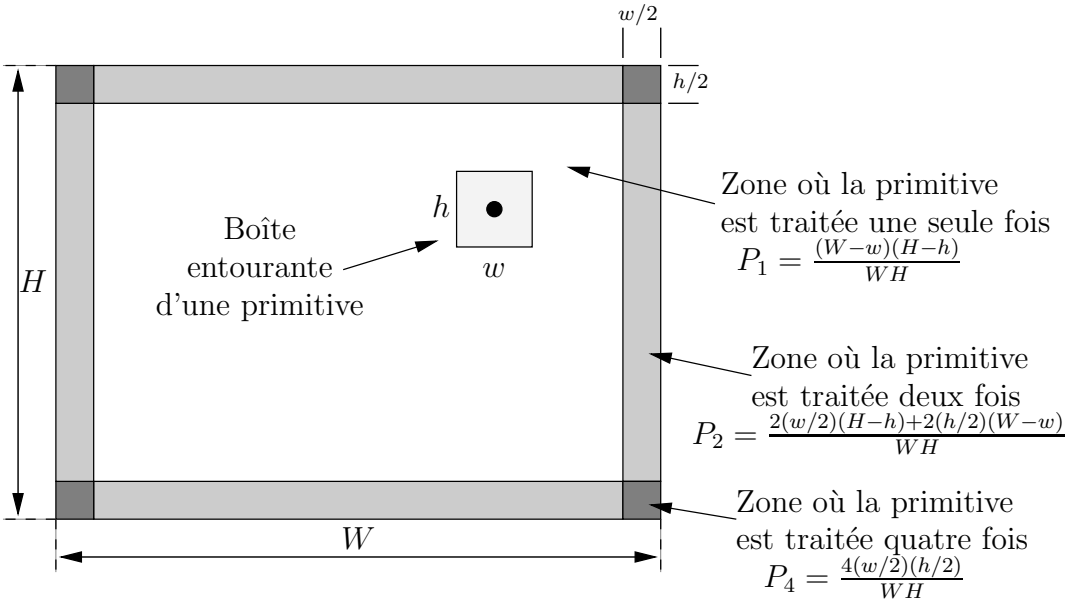


FIG. 7.2 – Calcul du facteur de recouvrement d'un carré sur un treillis.

Dans notre cas, le nombre de tuiles recouvertes par une primitive peut être exprimé par l'équation (7.4).

$$\text{N}_{\text{T.}} = \left(\frac{L_{\text{T.}} + L_{\text{P}}}{L_{\text{T.}}}\right)^2 \tag{7.4}$$

De même, l'équation (7.5) permet d'évaluer le nombre de triangles par tuile.

$$\text{N}_{\text{Tri}} = \left(\frac{L_{\text{T.}} + L_{\text{Tri}}}{L_{\text{Tri}}}\right)^2 \tag{7.5}$$

En réexprimant les différentes fréquences de traitement à l'aide des facteurs de recouvrements fournis par les équations (7.4) et (7.5), les fréquences respectives des

deux processus sont alors données les équations (7.6) et (7.7).

$$\begin{aligned} f_{\text{Pix}} &= \left(\frac{L_{\text{T.}} + L_{\text{P}}}{L_{\text{T.}}}\right)^2 \frac{L_{\text{T.}}^2}{T_{\text{Cal}}} \\ \iff f_{\text{Pix}} &= \left(1 + \frac{L_{\text{T.}}}{L_{\text{P}}}\right)^2 \frac{L_{\text{P}}^2}{T_{\text{Cal}}} \end{aligned} \tag{7.6}$$

et

$$\begin{aligned} f_{\text{Tri}} &= \left(\frac{L_{\text{T.}} + L_{\text{P}}}{L_{\text{T.}}}\right)^2 \left(\frac{L_{\text{T.}} + L_{\text{Tri}}}{L_{\text{Tri}}}\right)^2 \frac{1}{T_{\text{Cal}}} \\ \iff f_{\text{Tri}} &= \left(\frac{1}{L_{\text{P}}} + \frac{1}{L_{\text{T.}}}\right)^2 (L_{\text{T.}} + L_{\text{Tri}})^2 \frac{L_{\text{P}}^2}{L_{\text{Tri}}^2 T_{\text{Cal}}} \end{aligned} \tag{7.7}$$

Afin de maintenir un temps de calcul égal au temps de référence du rendu idéal, c.-à-d. $T_{\text{Cal}} = T_{\text{Cal}_0}$, nous pouvons exprimer l'augmentation des fréquences de traitement relativement aux fréquences de référence en fonction de la taille des tuiles.

Ainsi, de l'équation (7.6) et de la définition de f_{Pix_0} (éq. (7.2)), il vient

$$f_{\text{Pix}}(L_{\text{T.}}) = \left(1 + \frac{L_{\text{T.}}}{L_{\text{P}}}\right)^2 f_{\text{Pix}_0} \tag{7.8}$$

Puis, de l'équation (7.7) et de la définition de f_{Tri_0} (éq. (7.3)), il vient

$$f_{\text{Tri}}(L_{\text{T.}}) = \left(\frac{1}{L_{\text{P}}} + \frac{1}{L_{\text{T.}}}\right)^2 (L_{\text{T.}} + L_{\text{Tri}})^2 f_{\text{Tri}_0} = \left(1 + \frac{L_{\text{Tri}}}{L_{\text{P}}} + \frac{L_{\text{Tri}}}{L_{\text{T.}}} + \frac{L_{\text{T.}}}{L_{\text{P}}}\right)^2 f_{\text{Tri}_0} \tag{7.9}$$

Nous voyons que la fréquence « Pixel » (équation (7.8)), lors d'un rendu par tuile, croît quadratiquement avec la taille de la tuile afin de conserver un temps de calcul constant. Ce surcoût découle du recouvrement partiel de tuiles par la primitive et du choix du parcours intégral de ces tuiles. Concernant la fréquence « Triangle » (équation (7.9)), deux extrêmes se dégagent. Tout d'abord, l'impact du nombre de triangles par tuile est d'autant plus important que la tuile est petite : chaque triangle étant retraité pour chaque tuile impactée, le nombre efficace de triangles implique une fréquence élevée pour ce processus de traitement. Lorsque la taille des tuiles augmente, nous retrouvons le rapport des tailles de tuile et de primitive : le surcoût provient encore une fois du recouvrement partiel des tuiles par la primitive. Néanmoins, le facteur étant strictement plus grand que un, la fréquence « Triangle » sera toujours supérieure à la fréquence de référence. Cette fréquence « Triangle » dispose toutefois d'un minimum pour une taille de tuile $L_{\text{T.}} = \sqrt{L_{\text{Tri}} L_{\text{P}}}$, moyenne géométrique des tailles des triangles et de la primitive. La figure 7.3 illustre les tendances globales des courbes des fréquences « Pixel » et « Triangle » nécessaires pour conserver un temps de calcul d'une primitive constant en fonction de la taille des tuiles. Ces courbes sont renormalisées par rapport aux fréquences de référence f_{Pix_0} et f_{Tri_0}.

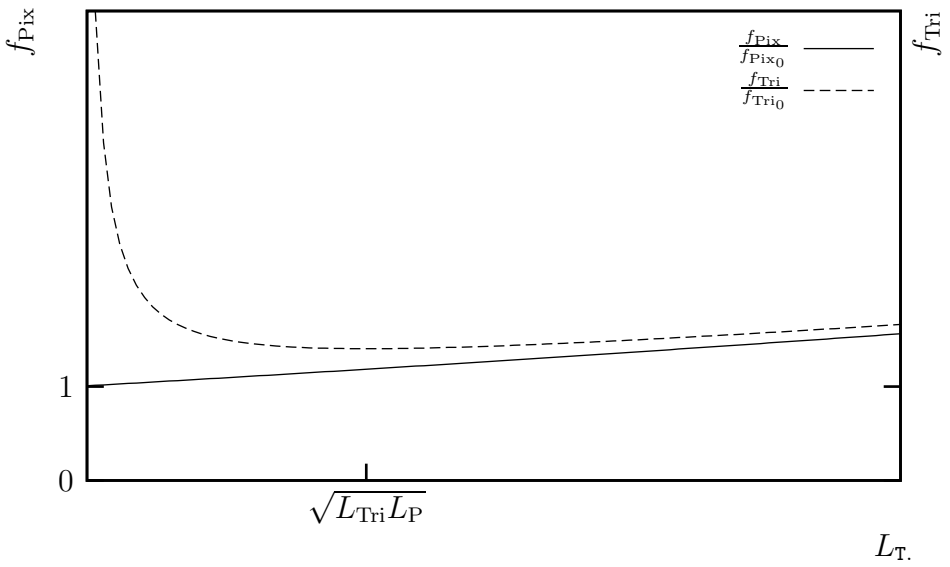


FIG. 7.3 – Variations des fréquences f_{Pix} et f_{Tri} en fonction de la taille de la tuile.

Nous pouvons à présent appliquer les formules (7.8) et (7.9) aux cas de scènes réalistes. Nous étudions pour cela deux cas. Le premier cas correspond à la scène « Vampire » déjà mentionnée. Le personnage principal de cette scène est considéré comme une unique primitive. Celle-ci est composée de 850 triangles visibles pour une aire totale de 24 500 pixels. La surface moyenne de chaque triangle est de 28 pixels. Les paramètres caractéristiques de cette primitive sont $L_{\text{P}} \approx 156$ et $L_{\text{Tri}} \approx 5$. Le tableau 7.2 résume, pour cette primitive, l'impact de la taille des tuiles sur les fréquences des différents processus.

Paramètres : $L_{\text{P}} = 156$ $L_{\text{Tri}} = 5$		
$L_{\text{T.}}$	$f_{\text{Pix}} = \alpha f_{\text{Pix}_0}$ $\alpha = \left(1 + \frac{L_{\text{T.}}}{L_{\text{P}}}\right)^2$	$f_{\text{Tri}} = \beta f_{\text{Tri}_0}$ $\beta = \left(1 + \frac{L_{\text{Tri}}}{L_{\text{P}}} + \frac{L_{\text{Tri}}}{L_{\text{T.}}} + \frac{L_{\text{T.}}}{L_{\text{P}}}\right)^2$
8	$\alpha \approx 1{,}1$	$\beta \approx 2{,}9$
16	$\alpha \approx 1{,}2$	$\beta \approx 2{,}1$
32	$\alpha \approx 1{,}5$	$\beta \approx 1{,}9$
64	$\alpha \approx 2{,}0$	$\beta \approx 2{,}3$

TAB. 7.2 – Impact de la taille des tuiles sur les fréquences « Pixel » et « Triangle ». Cas du personnage principal de la scène « Vampire ».

Le second cas correspond à l'étude d'une application d'animation faciale que nous nommerons scène « Badin ». Cette application est détaillée en partie V. Cette scène est constituée d'un unique maillage d'environ 1 000 triangles correspondant au visage d'un locuteur. Les paramètres caractéristiques de la primitive sont ici $L_{\mathrm{P}} = 256$ et $L_{\mathrm{Tri}} = 8$. Ce cas est considéré comme typique pour une application portable avec des afficheurs de taille réduite (CIF 384×288). Le tableau 7.3 résume l'impact de la taille de la tuile sur les fréquences des différents processus.

Paramètres : $L_{\mathrm{P}} = 256$ $L_{\mathrm{Tri}} = 8$		
$L_{\mathrm{T.}}$	$f_{\mathrm{Pix}} = \alpha f_{\mathrm{Pix}_0}$ $\alpha = \left(1 + \frac{L_{\mathrm{T.}}}{L_{\mathrm{P}}}\right)^2$	$f_{\mathrm{Tri}} = \beta f_{\mathrm{Tri}_0}$ $\beta = \left(1 + \frac{L_{\mathrm{Tri}}}{L_{\mathrm{P}}} + \frac{L_{\mathrm{Tri}}}{L_{\mathrm{T.}}} + \frac{L_{\mathrm{T.}}}{L_{\mathrm{P}}}\right)^2$
8	$\alpha \approx 1{,}1$	$\beta \approx 4{,}3$
16	$\alpha \approx 1{,}1$	$\beta \approx 2{,}5$
32	$\alpha \approx 1{,}3$	$\beta \approx 2{,}0$
64	$\alpha \approx 1{,}6$	$\beta \approx 2{,}0$

TAB. 7.3 – Impact de la taille des tuiles sur les fréquences « Pixel » et « Triangle ». Scène « Badin » cf. partie V.

Cette étude nous montre que, dans les deux cas exposés, il est inutile de dépasser des tailles de tuile supérieures à 32×32 car le minimum de la valeur β est atteint pour des valeurs de $L_{\mathrm{T.}}$ voisines de 32. D'autre part, la fréquence du processus « Triangle » croissant très rapidement avec la diminution de $L_{\mathrm{T.}}$ (jusqu'à un facteur 4 pour des tuiles de tailles 8×8), nous serons conduits à focaliser notre attention sur d'éventuelles simplifications de ce processus dans de futures optimisations (cf. chapitre 8).

Toutefois, la valeur $\mathrm{N}_{\mathrm{Tri}}$ du nombre de triangles effectivement traités par tuile permettant d'équilibrer le processus « Triangle » avec le processus « Pixel » a été jusqu'à présent basée sur un nombre moyen de triangles par tuile. Une étude basée sur des statistiques réelles de répartition des triangles dans chaque tuile d'une scène doit permettre d'affiner les résultats précédents.

7.2.2.2 Raffinement du modèle

Nous supposons dorénavant que les deux processus « Triangle » et « Pixel » ont un temps de traitement identique pour une valeur arbitraire N du nombre de triangles

traités par tuile. L'équation (7.10) est alors vérifiée.

$$f_{\text{Pix}} = \frac{{L_{\text{T.}}}^2}{\text{N}} f_{\text{Tri}} \tag{7.10}$$

Deux cas sont désormais envisageables :
- Le nombre de triangles $\text{N}_{\text{Tri}}(t)$ impactant la tuile t est inférieur (ou égal) au nombre N. La fréquence « Pixel » sera alors limitative et le temps de traitement de la tuile correspondra au temps du point d'équilibre ;
- Le nombre de triangles impactant la tuile est supérieur à N. La fréquence « Triangle » est alors limitative et le temps de traitement de la tuile correspondra à $\frac{\text{N}_{\text{Tri}}(t)}{f_{\text{Tri}}}$: le processus « Pixel » attendra donc la fin du processus « Triangle ».

Le temps de traitement d'une primitive complète est alors donné par l'équation (7.11).

$$T_{\text{Cal}} = \sum_{\substack{t=0 \\ \text{N}_{\text{Tri}}(t) \leq \text{N}}}^{\text{N}_{\text{T.}}} \frac{\text{N}}{f_{\text{Tri}}} + \sum_{\substack{t=0 \\ \text{N}_{\text{Tri}}(t) > \text{N}}}^{\text{N}_{\text{T.}}} \frac{\text{N}_{\text{Tri}}(t)}{f_{\text{Tri}}} \tag{7.11}$$

D'où, en notant $\text{N}_{\text{T.}}^{\text{N}}$ le nombre de tuiles ayant moins de N triangles les impactant, il vient

$$f_{\text{Tri}} = \frac{1}{T_{\text{Cal}}} \left(\text{N} \cdot \text{N}_{\text{T.}}^{\text{N}} + \sum_{\substack{t=0 \\ \text{N}_{\text{Tri}}(t) > \text{N}}}^{\text{N}_{\text{T.}}} \text{N}_{\text{Tri}}(t) \right) \tag{7.12}$$

Or, en exprimant f_{Tri_0} à l'aide du nombre total de triangles de la primitive $\text{N}_{\text{Tri}}^{\text{Tot.}}$: $f_{\text{Tri}_0} = \frac{\text{N}_{\text{Tri}}^{\text{Tot.}}}{T_{\text{Cal}_0}}$, nous obtenons, pour conserver le temps de calcul de référence

$$\frac{f_{\text{Tri}}}{f_{\text{Tri}_0}} = \frac{1}{\text{N}_{\text{Tri}}^{\text{Tot.}}} \left(\text{N} \cdot \text{N}_{\text{T.}}^{\text{N}} + \sum_{\substack{t=0 \\ \text{N}_{\text{Tri}}(t) > \text{N}}}^{\text{N}_{\text{T.}}} \text{N}_{\text{Tri}}(t) \right) \tag{7.13}$$

De cette dernière équation, de l'équation (7.10) et de l'expression de f_{Pix_0} reliant le temps de calcul à $\mathcal{A}_{\text{P}}$, l'aire de la primitive : $f_{\text{Pix}_0} = \frac{\mathcal{A}_{\text{P}}}{T_{\text{Cal}_0}}$, nous en déduisons

$$\frac{f_{\text{Pix}}}{f_{\text{Pix}_0}} = \frac{{L_{\text{T.}}}^2}{\text{N}\mathcal{A}_{\text{P}}} \left(\text{N} \cdot \text{N}_{\text{T.}}^{\text{N}} + \sum_{\substack{t=0 \\ \text{N}_{\text{Tri}}(t) > \text{N}}}^{\text{N}_{\text{T.}}} \text{N}_{\text{Tri}}(t) \right) \tag{7.14}$$

Nous disposons des statistiques réelles de la valeur de N_{Tri} pour la scène « Badin » dans le cas d'une taille de tuiles 16×16. Ces statistiques sont reproduites tableau 7.4.

En appliquant ces statistiques, nous traçons les courbes respectives des équations (7.13) et (7.14). La figure 7.4 rend compte de deux variations opposées. D'une

0	0	0	0	0	4	8	10	11	10	9	8	4	0	0	0	0	0
0	0	0	0	9	17	16	12	10	8	8	14	15	7	0	0	0	0
0	0	3	10	20	15	10	6	8	8	4	10	17	16	7	0	0	0
0	0	9	15	14	8	8	4	8	8	4	8	12	15	17	8	0	0
0	3	13	14	12	8	8	6	8	8	4	8	8	8	14	16	4	0
0	7	15	14	13	8	9	8	6	5	2	6	8	8	12	16	8	0
0	7	15	14	13	10	9	8	8	8	5	8	9	8	13	20	10	0
0	11	16	13	15	15	10	6	12	12	11	7	9	10	13	19	10	0
0	15	21	22	19	22	26	15	17	12	11	10	18	16	15	18	14	0
0	15	14	14	14	20	16	16	15	15	12	14	24	10	21	20	11	0
1	17	14	16	18	20	9	16	12	10	18	27	20	13	17	14	14	0
9	18	14	11	14	18	14	16	12	13	17	15	11	12	14	13	14	0
10	26	12	8	8	12	12	14	8	8	12	14	8	8	8	9	17	0
10	24	14	8	4	8	8	16	8	10	11	13	9	9	9	11	17	0
6	20	15	11	8	9	5	14	9	13	15	7	12	10	7	14	21	0
1	19	11	12	10	6	4	6	7	6	9	5	10	8	7	23	18	0
0	11	17	14	10	6	6	4	4	3	4	3	5	4	4	23	10	0
0	2	15	16	9	7	7	3	1	0	3	6	9	9	11	21	1	0
0	0	13	19	8	8	9	5	6	2	4	7	7	10	11	8	0	0
0	0	7	14	9	8	7	4	9	5	7	12	8	9	9	4	0	0
0	0	4	7	7	8	9	5	10	5	7	9	8	10	7	2	0	0
0	0	2	8	10	12	8	6	8	4	12	13	13	9	3	1	0	0
0	0	0	3	5	9	6	7	10	5	11	8	9	5	1	1	0	0
0	0	0	0	1	5	7	5	8	4	10	7	4	0	0	0	0	0

TAB. 7.4 – Répartition des triangles sur les tuiles de taille 16×16. Cas du locuteur PB au format CIF.

part, lorsque le nombre N de triangles effectivement traités par tuile est faible, la relaxation induite sur la contrainte de fréquence « Triangle » aboutit à un surcoût proche de celui déterminé dans le tableau 7.3. La conservation du temps de référence est alors assurée par une réduction du temps de traitement d'une tuile et donc par l'augmentation significative de la fréquence « Pixel ». Inversement, lorsque N est élevé, l'impact de la distribution des triangles sur les tuiles disparaît débouchant sur une fréquence « Pixel » dont le surcoût est ici proche de celui déterminé dans le tableau 7.3. En contrepartie, la fréquence « Triangle » ne peut qu'augmenter.

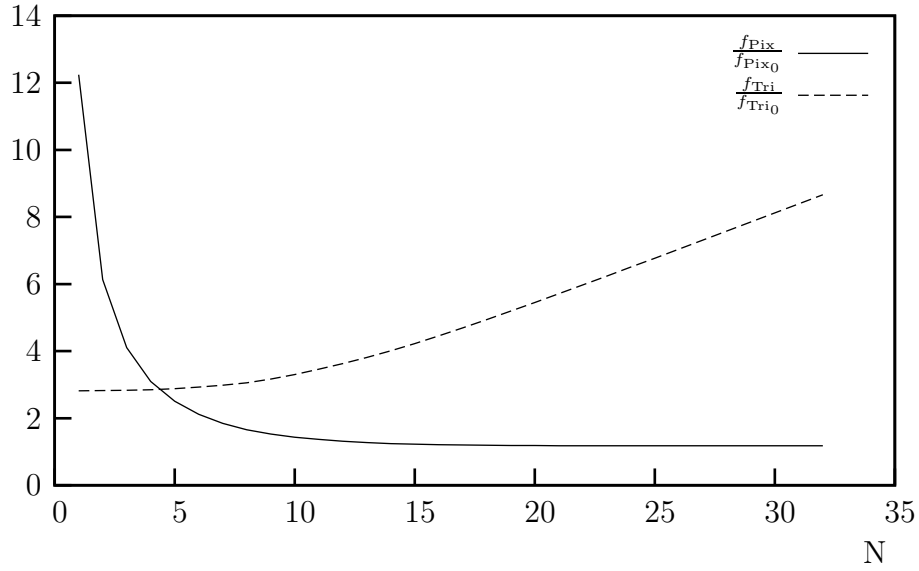


FIG. 7.4 – Évolutions comparées des fréquences « Pixel » et « Triangle ».

7.2.2.3 Implantation réaliste du modèle

Dans le modèle précédent, la taille de la mémoire tampon entre les deux processus reste potentiellement infinie et le processus « Triangle » doit pouvoir gérer un nombre indéfini de triangles. Pour pallier à cet inconvénient, nous imposons arbitrairement une taille de mémoire tampon adaptée au stockage des paramètres d'exactement N triangles. Le traitement d'une tuile est effectué par un nombre suffisant d'itérations du processus « Triangle » par tranche atomique de N triangles. Chaque itération est associée à un parcours complet de la tuile par le processus « Pixel ». Nous définissons alors un atome comme un ensemble de N triangles : pour une tuile t, le nombre d'atomes est donné par $\left\lceil \frac{\mathrm{N}_{\mathrm{Tri}}(t)}{\mathrm{N}} \right\rceil$.

Les équations (7.15) et (7.16) réexpriment, à l'aide du nombre total d'atomes $\mathrm{Nb}_{\mathrm{At}}$, les nouvelles relations des fréquences « Pixel » et « Triangle ».

$$\frac{f_{\mathrm{Pix}}}{f_{\mathrm{Pix}_0}} = \frac{{L_{\mathrm{T.}}}^2 \mathrm{Nb}_{\mathrm{At}}}{\mathcal{A}_{\mathrm{P}}} \tag{7.15}$$

$$\frac{f_{\mathrm{Tri}}}{f_{\mathrm{Tri}_0}} = \frac{\mathrm{N} \cdot \mathrm{Nb}_{\mathrm{At}}}{\mathrm{N}_{\mathrm{Tri}}^{\mathrm{Tot.}}} \tag{7.16}$$

Les statistiques réelles, données tableau 7.4, permettent de calculer le nombre total d'atomes à traiter dans le cas de la scène « Badin ». Nous pouvons observer sur la figure 7.5 les conséquences de l'atomisation du traitement sur les fréquences « Pixel » et « Triangle » nécessaires. Nous constatons que le surcoût fréquentiel reste très raisonnable.

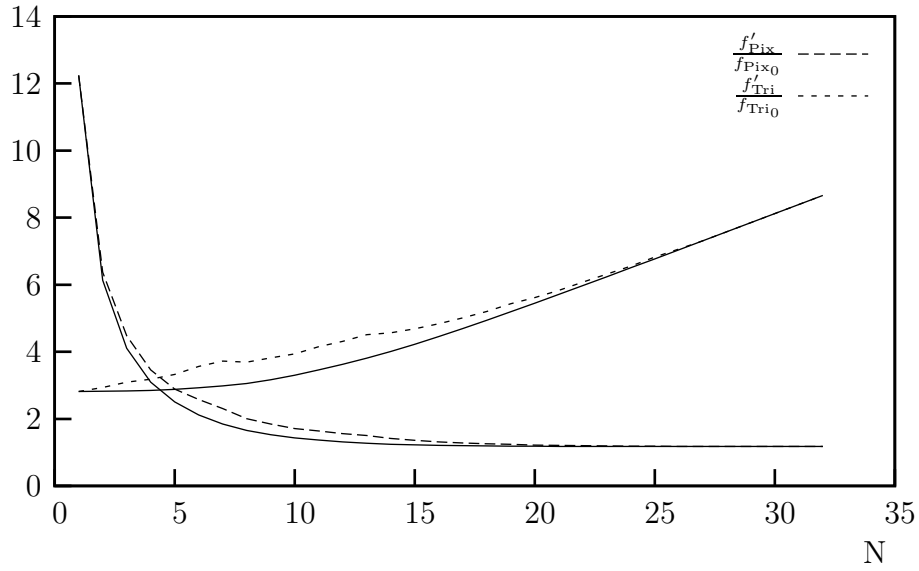


FIG. 7.5 – Évolutions comparées des fréquences « Pixel » et « Triangle » (2).

7.2.2.4 Synthèse

Grâce à notre dernier modèle, nous disposons d'un jeu de paramètres ($L_{\text{T.}}$, N, etc.) avec lesquelles nous pouvons ajuster les différentes fréquences nécessaires des processus de traitement afin d'assurer des conditions d'exécution spécifiques. Cependant, l'organisation proposée est capable de s'accommoder de conditions d'exécution variables notamment par la définition de traitements de nature atomique qui permettent de lisser (avec N le nombre de triangles par atome) les disparités de la scène. Sur des bases d'un rendu pire cas traitant N triangles par tuile, nous autorisons un rendu au mieux par une répétition des traitement assurant ainsi une extensibilité à notre modèle de rendu de triangles. De plus, la surcharge engendrée par l'approche tuile ne constitue pas une difficulté insurmontable : il est possible de choisir des fréquences de fonctionnement proches de notre modèle idéal pour des tailles de tuiles raisonnables afin d'envisager l'implémentation de mémoire interne. Nous étudions dans notre prochain paragraphe une organisation des traitements permettant de raffiner, au niveau matériel, la description haut-niveau offerte par notre modèle.

7.2.3 Parcours d'une tuile

La figure 7.6, à comparer avec la figure 3.19 page 54, raffine l'algorithme du rendu par tuiles par l'insertion du niveau des atomes.

```
Pour chaque image
  Pour chaque tuile
    Pour chaque primitive dans la tuile
      Pour chaque atome
        Effectuer le processus « Triangle »
        Effectuer le processus « Pixel »
    Composer les différentes primitives
  Afficher l'image
```

FIG. 7.6 – Pseudo-code raffiné de rendu par tuiles.

Nous nous intéressons désormais plus particulièrement à la réalisation de la boucle de parcours d'un unique atome. Nous abordons donc chacun des processus en commençant par le processus « Pixel ».

7.2.3.1 Le processus « Pixel »

La notion d'atome telle que nous l'avons introduite est liée aux objets constitués de triangles. Néanmoins, nous pouvons étendre ce concept aux objets de type vidéo pour lesquels seul le processus « Pixel » intervient. Dès lors, deux nouveaux arguments viennent renforcer le choix du balayage intégral de la tuile lors du processus « Pixel ». Il s'agit de l'uniformisation et de l'indifférenciation des traitements, décorrélées du contenu de l'atome. La tuile est balayée de manière traditionnelle comme un balayage d'écran : ligne à ligne de haut en bas et pour chaque ligne, pixel à pixel de gauche à droite. Afin de pouvoir générer les informations de texture pour chaque pixel, il convient de calculer les intersections de chaque ligne de la tuile avec les N triangles de l'atome. La fréquence avec laquelle un nouveau calcul d'intersection doit être effectué est donnée par l'expression (7.17). Cette expression relie le nombre de triangles par atome, la taille de la tuile et la fréquence « Pixel ».

$$f_{\text{Int}} = \frac{\text{N}}{L_{\text{T.}}} f_{\text{Pix}} \tag{7.17}$$

La figure 7.7 représente ce calcul d'intersection faisant intervenir les trois côtés d'un triangle. La comparaison des abscisses des points d'intersection définit l'intérieur du triangle. Ce calcul permet de mettre à jour une table d'index des triangles recouvrant

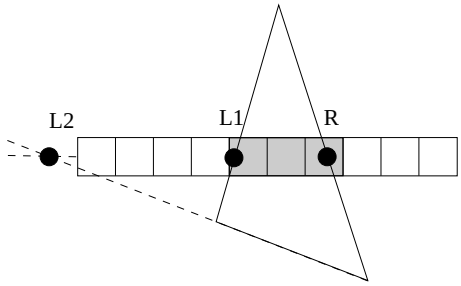


FIG. 7.7 – Calcul d'intersection d'un triangle et d'une ligne de tuile. Un pixel d'abscisse x_i est intérieur au triangle si et seulement si $x_i \geq \max(\text{L1}, \text{L2})$ et $x_i \leq \text{R}$. Les points L1 et L2 (resp. R) correspondent aux intersections des deux côtés gauches (resp. droit) de ce triangle avec la ligne.

chaque pixel. Cette table indique donc, pour chaque pixel de la ligne, le numéro du

triangle (parmi N) le recouvrant. La figure 7.8 indique, dans le cas d'un atome composé de huit triangles impactant une tuile, la valeur de cette table d'index à la fin des différents calculs d'intersections. Les traitements des pixels de la ligne sont ensuite ef-

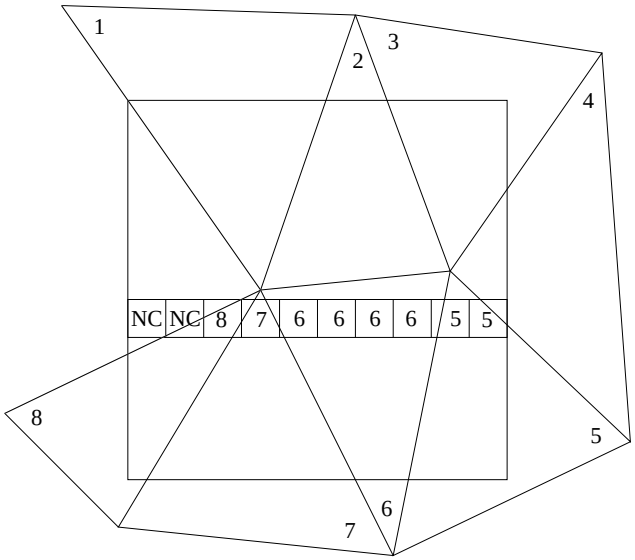


FIG. 7.8 – Table d'index des intersections triangle-ligne de tuile. La valeur particulière NC indique un pixel « Non-Couvert ».

fectués grâce au parcours de cette table d'index. Chaque triangle définissant une transformation géométrique, cette table identifie par conséquent quelle transformation appliquer pour chaque pixel. Cependant, les ruptures de contexte qui peuvent intervenir, c.-à-d. les changements de valeur dans la table d'index pendant le parcours d'une ligne, nécessitent des opérations de sauvegarde et de chargement de données. Suivant le choix des modes de calcul des transformées géométriques, ces opérations peuvent être relativement lourdes. La figure 7.9 présente diverses alternatives pour gérer ces changements de contexte.

```
Pour chaque nouveau contexte
  Sauvegarder le contexte courant
  Charger le nouveau contexte
  Calcul incrémental des pixels
     du segment en cours
```

```
Pour chaque nouveau contexte
  Calcul absolu de ce contexte
  Calcul incrémental des pixels
     du segment en cours
```

```
Calcul absolu pour chaque pixel
    quelque soit le contexte
```

FIG. 7.9 – Calculs possibles lors du processus « Ligne ».

La première alternative conserve les techniques incrémentales traditionnellement employées lors d'un parcours triangle à triangle : elles ne sont cependant plus les mieux adaptées en raison du nombre de données intermédiaires à sauvegarder et à accéder. La deuxième alternative évite la gestion de ces données intermédiaires par un calcul absolu lors d'un changement de contexte. Dès lors que nous mettons en place un calcul direct, la présence de ces opérateurs matériels conduit naturellement à systématiser ces calculs directs pour tous les pixels : il s'agit de la dernière alternative. Le calcul incrémental a pour objectif principal la réduction de la complexité des opérateurs, c.-à-d. le remplacement d'une multiplication par une addition. Dans notre cas, où les traitements sont

liés à un contexte de la taille d'une tuile, la dynamique des données traitées (4 bits pour une tuile 16×16) permet de justifier le choix d'un calcul absolu : les opérateurs sont compacts, il n'y a pas de contexte à sauvegarder.

Nous organisons donc le rendu d'une tuile en découpant le processus « Pixel » défini dans notre modèle en deux processus interfacés par cette table d'index : un premier processus « Ligne » suivi du processus « Balayage » effectuant le balayage des pixels proprement dit. Cette organisation du rendu d'une tuile permet de masquer le temps de traitement spécifique d'un triangle dans le temps complet du traitement d'une tuile : le temps de traitement d'un triangle n'est plus conditionné par la surface qu'il recouvre mais par le nombre de tuiles recouvertes.

Le processus « Ligne »

Le processus « Ligne » détermine l'intersection d'un triangle et d'une ligne courante. Pour cela, le parcours détermine l'intersection de chaque côté du triangle avec la ligne courante et, par comparaison des différentes abscisses de ces intersections, délimite le segment correspondant à l'intersection.

En raison de l'orientation ligne du traitement, nous exprimons logiquement l'abscisse d'un point en fonction de son ordonnée. Connaissant un point de référence $(x_{\text{ref}}, y_{\text{ref}})$, l'équation (7.18) détermine l'abscisse d'un point d'ordonnée y connue.

$$x = x_{\text{ref}} + \frac{\mathrm{d}x}{\mathrm{d}y}(y - y_{\text{ref}}) \tag{7.18}$$

Le segment, intersection entre la ligne et le triangle, est déterminé par les abscisses de ses extrémités. Une des extrémités du segment est déterminée de manière directe par l'intersection du côté de plus grande excursion verticale avec la ligne courante. Il s'agit du côté gauche et donc du point L dans le cas de la figure 7.10 (il s'agissait du côté droit figure 7.7). La seconde extrémité est obtenue en choisissant judicieusement entre les intersections des deux autres côtés : il s'agit de R1 dans le même exemple.

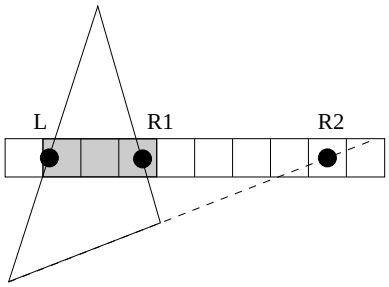


FIG. 7.10 – Calcul d'intersection d'un triangle et d'une ligne de tuile (2). Un pixel d'abscisse x_i est intérieur au triangle si et seulement si $x_i \geq \mathrm{L}$ et $x_i \leq \min(\mathrm{R1}, \mathrm{R2})$. Le point L (resp. R1 et R2) correspond à l'intersection du côté gauche (resp. droits) de ce triangle avec la ligne.

Les extrémités de l'intersection étant déterminées, celles-ci sont ensuite comparées

aux abscisses de chaque pixel de la ligne de tuile de manière à remplir la table d'index. Le tableau 7.5 résume les opérations arithmétiques nécessaires pour le processus « Ligne » lors du traitement d'une seule ligne.

	x	Intersection
Additions	4N	$(1 + 2L_{\text{T.}})$ N
Multiplications	3N	0
Divisions	0	0

TAB. 7.5 – Opérations arithmétiques pour le traitement d'une ligne : processus « Ligne ». Le calcul $(y - y_{\text{ref}})$ est commun pour chaque côté ; les comparaisons sont assimilées à des additions.

Le processus « Balayage »

Après la détermination de la table d'index des triangles recouvrant une ligne de la tuile, le processus « Balayage » se charge de calculer les coordonnées dans l'espace texture du pixel courant. Nous assimilons la transformation géométrique entre les espaces écran et texture à une transformation affine définie par six constantes $\frac{\partial u}{\partial x}$, $\frac{\partial v}{\partial x}$, u_{ref}, $\frac{\partial u}{\partial y}$, $\frac{\partial v}{\partial y}$ et v_{ref}. Les coordonnées $(u_{\text{ref}}, v_{\text{ref}})$ correspondent aux coordonnées de texture d'un point de référence $(x_{\text{ref}}, y_{\text{ref}})$. Connaissant ce point de référence, la détermination des coordonnées de texture d'un point quelconque est rappelée par l'équation (7.19).

$$\begin{cases} u = \frac{\partial u}{\partial x}(x - x_{\text{ref}}) + \frac{\partial v}{\partial x}(y - y_{\text{ref}}) + u_{\text{ref}} \\ v = \frac{\partial u}{\partial y}(x - x_{\text{ref}}) + \frac{\partial v}{\partial y}(y - y_{\text{ref}}) + v_{\text{ref}} \end{cases} \tag{7.19}$$

Le tableau 7.6 donne le nombre des opérations arithmétiques nécessaires pour le processus « Balayage » pour le traitement d'une seule ligne.

	(u, v)
Additions	$6L_{\text{T.}}$
Multiplications	$4L_{\text{T.}}$
Divisions	0

TAB. 7.6 – Opérations arithmétiques pour le traitement d'une ligne : processus « Balayage ».

7.2.3.2 Le processus « Triangle »

Le processus « Triangle » consiste à initialiser l'ensemble des paramètres utiles lors du parcours de la tuile, aussi bien pour le calcul d'intersection triangle-ligne que pour la transformation géométrique appliquée lors du balayage des pixels : il s'agit bien d'une phase d'initialisation. Nous avons souligné l'inadéquation des calculs incrémentaux lors

du parcours d'une tuile et la systématisation de calculs absolus. Néanmoins, le rendu par tuiles nous dispense de calculs absolus dans l'image complète en autorisant des calculs relatifs à la tuile relâchant certaines contraintes, notamment au niveau des précisions des données manipulées. Cette phase d'initialisation devra supporter l'abstraction ainsi faite. Ces calculs relatifs autorisent la définition d'un opérateur générique de parcours tuiles et de triangles sans connaissance *a priori* des caractéristiques de la scène à rendre.

Lors du précédent paragraphe, nous avons identifié deux processus distincts pour le parcours de la tuile. Chacun d'eux réclame l'initialisation de paramètres spécifiques.

Initialisation pour le processus « Ligne »

L'équation (7.18) requiert la connaissance de trois paramètres : les coordonnées du point de référence $(x_{\text{ref}}, y_{\text{ref}})$ et la pente du côté $\frac{\mathrm{d}x}{\mathrm{d}y}$. L'initialisation procède donc au calcul de ces paramètres à partir des coordonnées des extrémités de chaque côté d'un triangle. En notant TMB un triangle, l'équation (7.20) donne les calculs nécessaires lors de l'initialisation.

$$\begin{cases} x_{\text{ref}}^{[\text{TB}]} = x_{\text{T}} + \frac{x_{\text{T}} - x_{\text{B}}}{y_{\text{T}} - y_{\text{B}}}(y_{\text{ref}} - y_{\text{T}}) \\ x_{\text{ref}}^{[\text{TM}]} = x_{\text{T}} + \frac{\Delta x_{\text{TM}}}{\Delta y_{\text{TM}}}(y_{\text{ref}} - y_{\text{T}}) \\ x_{\text{ref}}^{[\text{MB}]} = x_{\text{M}} + \frac{\Delta x_{\text{MB}}}{\Delta y_{\text{MB}}}(y_{\text{ref}} - y_{\text{M}}) \end{cases} \tag{7.20}$$

L'initialisation d'un côté comprend donc le calcul de la pente de ce côté à l'aide d'une division et la détermination de x_{ref} à partir des coordonnées d'une extrémité du côté courant et d'une ordonnée de référence y_{ref} arbitraire. Le tableau 7.7 expose opérations arithmétiques indispensables pour caractériser ces différents paramètres pour les trois côtés des triangles d'un atome.

	x_{ref}	$\frac{\mathrm{d}x}{\mathrm{d}y}$
Additions	6N	6N
Multiplications	3N	0N
Divisions	0N	3N

Tab. 7.7 – Opérations arithmétiques : initialisation du processus « Ligne ».

Initialisation pour le processus « Balayage »

De même, le processus « Balayage » nécessite la connaissance des quatre coefficients affines et les coordonnées d'un point de référence arbitraire $(x_{\text{ref}}, y_{\text{ref}})$ dans la texture : $(u_{\text{ref}}, v_{\text{ref}})$. L'équation (7.21) indique comment déterminer ces dernières coordonnées de texture.

$$\begin{cases} u_{\text{ref}} = \frac{\partial u}{\partial x}(x_{\text{ref}} - x_{\text{T}}) + \frac{\partial v}{\partial x}(y_{\text{ref}} - y_{\text{T}}) + u_{\text{T}} \\ v_{\text{ref}} = \frac{\partial u}{\partial y}(x_{\text{ref}} - x_{\text{T}}) + \frac{\partial v}{\partial y}(y_{\text{ref}} - y_{\text{T}}) + v_{\text{T}} \end{cases} \tag{7.21}$$

Les coefficients affines sont le rapport entre un numérateur dépendant du coefficient à calculer et un dénominateur commun ne dépendant que du triangle. Chaque numérateur et dénominateur demande cinq additions et deux multiplications. En raison des redondances de calculs, au minimum treize additions et dix multiplications sont nécessaires pour les calculs des nombres intervenant dans ces rapports. Le calcul des coordonnées de texture de référence s'opère à l'aide des coordonnées d'un sommet du triangle grâce à six additions et quatre multiplications. Le tableau 7.8 expose les opérations arithmétiques indispensables pour caractériser ces différents paramètres pour un atome.

	$(u_{\text{ref}}, v_{\text{ref}})$	Coef.
Additions	6N	13N
Multiplications	4N	10N
Divisions	0N	4N

TAB. 7.8 – Opérations arithmétiques : initialisation du processus « Balayage ».

7.2.4 Complexité de calcul

Le tableau 7.9 reprend le nombre des opérations arithmétiques exprimé précédemment pour le processus « Triangle » (initialisation) et reporte ceux nécessaires pour les deux processus internes du processus « Pixel ». Ces opérations sont exprimées par atome traité, par tuile complète.

	« Triangle »	« Ligne »	« Balayage »
Additions	31N	$(4 + (1 + 2L_{\text{T.}}))\, L_{\text{T.}}\text{N}$	$6L_{\text{T.}}^2$
Multiplications	17N	$3L_{\text{T.}}\text{N}$	$4L_{\text{T.}}^2$
Divisions	7N	0	0

TAB. 7.9 – Opérations arithmétiques pour le parcours d'un atome. Nous avons exprimé ces complexités de calcul pour une tuile complète.

Afin d'évaluer les complexités relatives des différents processus, nous introduisons une formulation théorique de la complexité des opérateurs élémentaires :

- Un additionneur de taille N_{bit} aura une complexité de N_{bit}.
- Un multiplieur $\text{N}_{\text{bit}} \times \text{N}_{\text{bit}}$ aura une complexité de ${\text{N}_{\text{bit}}}^2$.
- Un diviseur $\text{N}_{\text{bit}}/\text{N}_{\text{bit}}$ aura une complexité de $2{\text{N}_{\text{bit}}}^2$.
- Les comparateurs du processus « Ligne » disposent d'une dynamique limitée par la taille de la tuile. La complexité d'un tel comparateur sera de $\log_2(L_{\text{T.}})$.

– Les deux processus internes « Ligne » et « Balayage » effectuant les opérations de calculs de multiplications sur une faible excursion liée simplement à la taille de la tuile, nous considérerons une complexité pour celles-ci de $\log_2(L_{\text{T.}})\,\mathrm{N_{bit}}$.

Connaissant pour une image complète le nombre d'atomes traités, les complexités de calcul de chacun des trois processus peuvent alors être déduites : la figure 7.11 retrace ces complexités de calcul ainsi que la somme totale dans l'exemple « Badin » en fonction de N pour une taille arbitraire des données $\mathrm{N_{bit}} = 16$ et une taille de tuile $L_{\text{T.}} = 16$. Les allures générales des courbes des complexités de calcul des processus « Pixel » et « Triangle » correspondent à celles des courbes de la figure 7.4. Enfin, l'existence d'un compromis possible est confortée malgré l'introduction d'un nouveau processus.

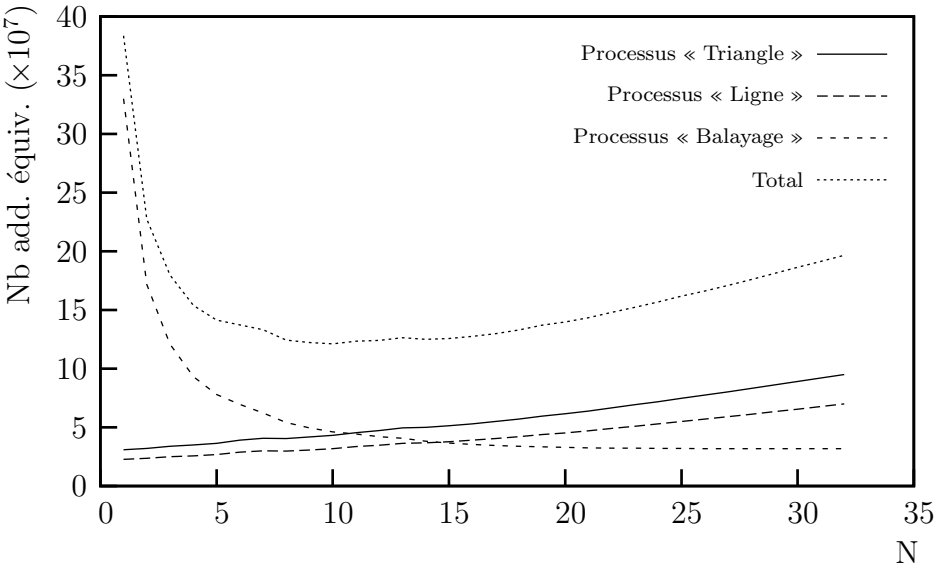


FIG. 7.11 – Complexités de calcul individuelles et cumulées des trois processus « Triangle », « Ligne » et « Balayage ». Complexités exprimées en nombre d'additions 1 bit équivalentes.

7.2.5 Résumé

Ce modèle approximatif de la complexité de calcul ne caractérise pas les complexités matérielles. En effet, ce modèle présuppose que toutes les opérations s'effectuent au même débit que l'additionneur de référence sans que soit prise en compte la fréquence effective des différents processus. Notamment, dans le cas du processus d'initialisation des triangles, cette fréquence peut conduire à l'implémentation de diviseurs hautement pipelinés d'une complexité matérielle largement sous-estimée. D'autre part, une implémentation matérielle peut conduire à un dimensionnement optimal de chaque opérateur de traitement en fonction de l'étape de calcul. Une telle implémentation peut alors modifier considérablement les conditions d'équilibre entre les processus.

En résumé, nous venons de présenter une organisation du rendu par tuiles. En modélisant cette organisation par l'intermédiaire de deux processus, nous avons pu montrer l'impact de la taille des tuiles sur les fréquences de fonctionnement à mettre en

œuvre pour chacun. Nous poursuivons maintenant par la question que nous avons, temporairement et volontairement, laissée de côté jusqu'à présent : l'accès aux données et les échanges avec la mémoire.

7.3 Accès aux données, pipeline et cache déterministe

Le modèle que nous avons proposé pipeline les deux processus « Triangle » et « Pixel ». À l'intérieur de ce dernier, les processus « Ligne » et « Balayage » sont eux-mêmes pipelinés dans le cas d'objets graphiques. Ce dernier pipeline est aisé grâce à la table d'index qui sert d'interface entre ces deux processus. Nous souhaitons cependant étudier plus étroitement l'organisation temporelle des traitements au niveau tuile et plus particulièrement comment s'effectuent, au sein de cette organisation, les différents accès aux données.

7.3.1 Traitements préliminaires : tri des primitives

L'ensemble des processus de rendu débute par le processus « Triangle » qui initialise au plus N triangles. Le traitement des triangles est donc soumis à la réorganisation des primitives (manipulées par l'application s'exécutant sur le processeur hôte) en structures atomiques manipulables par notre opérateur de rendu. La figure 7.12 illustre le cas d'un sous-maillage impactant une tuile ainsi que la découpe (arbitraire) en deux atomes dans le cas où N = 8. Nous présentons une structure de données qui permet de décrire ces

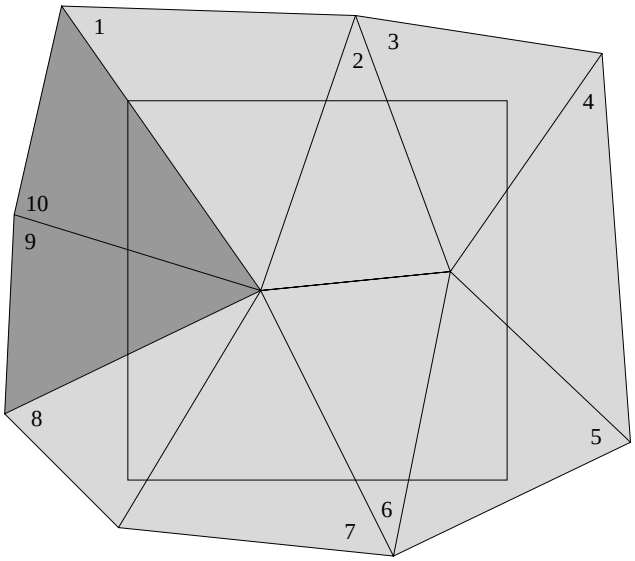


FIG. 7.12 – Sous-maillage impactant une tuile. Le choix de 8 triangles par atome conduit en un découpage en deux atomes de ce sous-maillage : les différences de ton indiquent ces atomes.

atomes. Nous étudions ensuite sa création par le processeur hôte.

7.3.1.1 Structure de données : la liste graphique

La structure de données effectue l'interface entre le processeur et le circuit dédié au traitement unifié de la vidéo et de la synthèse graphique. Elle doit donc être transparente au type de traitement à effectuer. De plus, outre les primitives graphiques, cette structure doit contenir les informations de configuration du pipeline de traitement.

Afin de respecter les problèmes de sémantiques de la composition d'image, nous choisissons une structure de données hiérarchique. Cette structure contient pour chaque tuile la liste ordonnée des objets graphiques (objets vidéo ou 3D) qui doivent être traités lors du parcours de cette tuile : nous la dénommerons ainsi « liste graphique ». Cette liste doit être interprétée par le coprocesseur pour déterminer la nature des données et les traitements à exécuter.

Les atomes que nous avons introduits permettent de faciliter cette interprétation et de maîtriser les ressources nécessaires pour le matériel : nous constituons cette liste à l'aide de sous-structures atomiques de taille fixe dont le temps de traitement sera fixe (ou au minimum borné). Dans le cas d'un objet vidéo, ces atomes permettent de définir la texture, la position ainsi que les informations permettant de calculer la transformation géométrique de cet objet. Dans le cas d'objets graphiques, les atomes doivent permettre de définir la texture appliquée. En raison de la similitude des informations de texture dans le cas vidéo et synthèse graphique, une seule structure est mise en place pour décrire une texture. La description des au plus N triangles constitue également un tel atome dans le cas d'objets graphiques.

7.3.1.2 Création de la liste graphique

La création de cette liste graphique constitue la première étape d'une application graphique ou vidéo. Plus particulièrement, il convient, pour chaque triangle, de déterminer le plus précisément les tuiles qu'il impacte.

Détermination exacte

La détermination exacte des tuiles impactées n'est pas adaptée. En effet, elle revient à effectuer un balayage complet du triangle ou, tout au moins, à parcourir l'ensemble des côtés des triangles : une tuile n'est-elle pas un pixel de grande taille ? Pour cela, les calculs exacts d'intersection des côtés et des lignes de tuiles doivent être menés. Exécutés principalement en représentation flottante, ces calculs requièrent une puissance de calcul incompatible avec la volonté de soulager le cœur de microprocesseur des traitements graphiques.

Déterminations approchées

Une méthode simple de détermination des tuiles impactées par un triangle consiste dans le calcul de la boîte entourante. Cette méthode allie rapidité et faible puissance

de calcul. Cependant, des triangles n'impactant pas une tuile peuvent être identifiés par cette méthode, induisant ainsi un surcoût, que nous avons déjà évoqué, lors du traitement des triangles contenus dans cette liste.

Nous avons comparé cette première méthode avec une méthode dichotomique ne calculant pas explicitement la division nécessaire lors du calcul exact d'intersection. Cette méthode découpe chaque côté d'un triangle en deux créant ainsi quatre sous-triangles. La récursivité de cet algorithme est arrêtée lorsque la boîte entourante atteint un seuil limite ou qu'elle est contenue dans une unique rangée.

La figure 7.13 présente les différentes tuiles identifiées par un triangle pour chacune des méthodes précédentes.

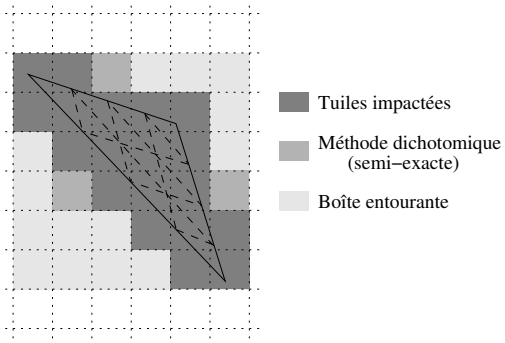


Fig. 7.13 – Tuiles impactées par un triangle.

Nous avons implémenté ces deux dernières méthodes sur un microprocesseur ARM afin d'évaluer leurs performances respectives. Pour cela, nous avons utilisé la scène « Badin ». La taille et la bonne conformation des triangles n'ont pas permis à la méthode dichotomique de faire la preuve de l'efficacité de son identification. Notre implémentation de cet algorithme nécessite 3,7 Minst pour une image. Quant à la méthode de la boîte entourante, elle ne nécessite que 1,6 Minst dans une implémentation directe. De plus, connaissant les identifiants des triangles d'une image à l'autre, une détermination des tuiles impactées exploitant la cohérence entre images est possible : la liste graphique est alors mise à jour en ne consommant que 1,0 Minst. La figure 7.14 relève, pour les 80 premières images d'une séquence de visage animé, le nombre d'instructions nécessaires pour mettre à jour cette liste graphique par la méthode de la boîte entourante.

Cependant, une API comme OpenGL n'identifie par directement les primitives d'une scène à l'autre. En envisageant une extension de marquage explicite des primitives, la mise à jour différentielle de la liste graphique est alors compatible avec des contraintes de puissance de calcul réduite pour assurer le temps réel. De plus, l'accroissement de complexité pour l'augmentation d'efficacité contrastée que nous avons pu constater ne justifie pas, en logiciel, l'emploi d'une méthode dichotomique qui, le cas échéant, pourra profiter d'une éventuelle implémentation matérielle. Ainsi, un triangle marque toutes les tuiles recouvertes partiellement par sa boîte entourante. Enfin, le surcoût qu'engendre la détermination de triangles non-impactant reste limité : au niveau de la liste graphique, cet impact est lissé par la valeur N du nombre de triangles représentables dans un atome ; au niveau du traitement de cet atome, cet impact ne dépend que de la manière dont les traitements sont effectués.

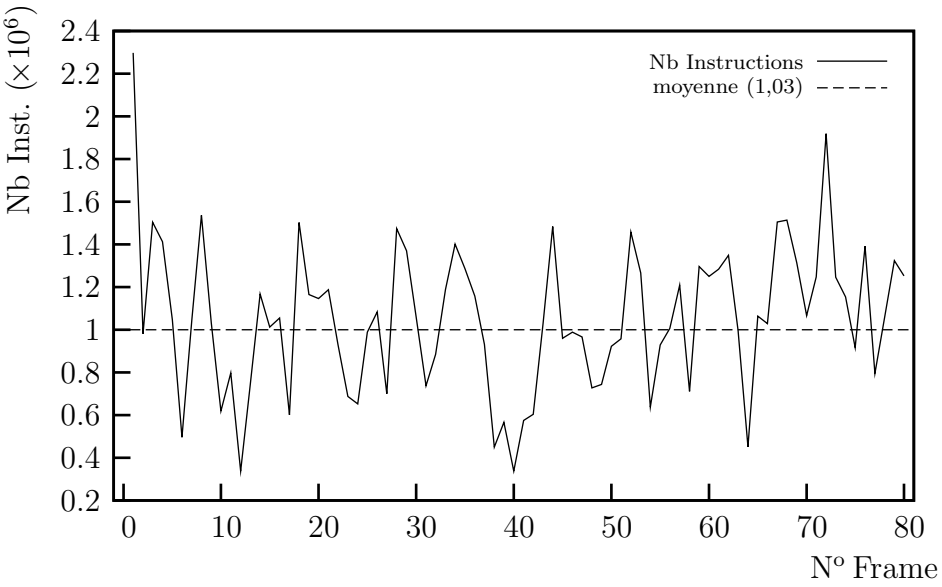


FIG. 7.14 – Nombre d'instructions nécessaires pour mettre à jour la liste graphique.

7.3.2 Organisation pipelinée

Le rendu par tuile permet de définir un macropipeline au niveau tuile. La durée du macrocycle associé est minorée par le temps de parcours des pixels de la tuile : temps que nous avons noté $\frac{L_{\text{T.}}^2}{f_{\text{Pix}}}$. Préalablement aux processus « Ligne » et « Pixel », l'initialisation des N triangles doit être réalisée. Un premier découpage temporel est alors institué, à un niveau tuile cette fois-ci, entre l'initialisation des objets graphiques et le balayage d'une tuile. La taille de la structure de donnée correspondant à un atome étant réduite, nous effectuons l'accès aux données géométriques lors du même macrocycle que l'initialisation. Les coefficients calculés pendant cette phase sont alors mémorisés pour permettre leur utilisations ultérieures. La puissance de calcul nécessaire pour initialiser les triangles peut alors être équirépartie sur (pratiquement) toute la durée d'un macrocycle tuile.

Les traitements dus à la transformation géométrique nécessitent, pour leur part, l'accès en mémoire centrale à une zone de texture. Les échanges de données entre le coprocesseur et la mémoire sont essentiellement dominés par les accès à cette zone de texture. Compte tenu de la nature des données accédées et des traitements subis, ces accès à des éléments d'image s'effectuent principalement de manière bidimensionnelle dans la texture. L'enchaînement des macrocycles au rythme défini par le temps de balayage d'une tuile n'est possible qu'à la condition que les opérations d'entrée-sortie, nécessaires pour le bon déroulement des différents traitements, ne viennent pas le ralentir.

Afin d'assurer le temps réel des traitements dans un profil de fonctionnement donné, il est possible de déterminer un profil pire cas des échanges. La figure 7.15 présente une tuile de côté $L_{\text{T.}}$ définissant, par une transformation géométrique, une première zone (représentée en clair) dans la texture. Grâce aux techniques de pyramides de textures (*MIP-map*), il est possible de conserver un facteur d'échelle proche de un entre l'aire de la tuile et de cette première zone. L'application d'un filtre dont le noyau est de largeur k définit la zone grisée des données effectivement nécessaires pour la génération des pixels

de la tuile ; la zone foncée offre une vision généraliste de cette zone en englobant les zones possibles quelle que soit la transformation affine courante.

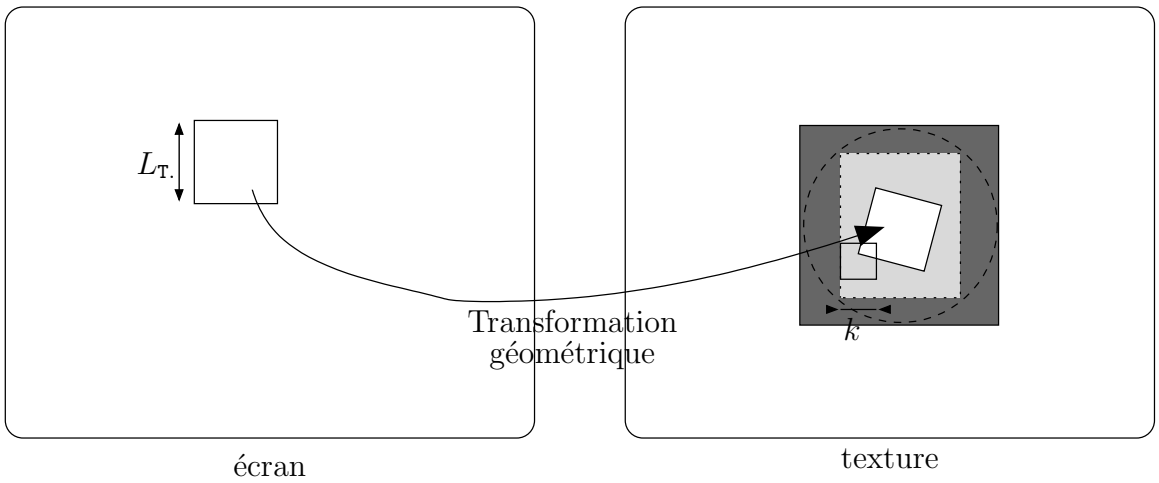


FIG. 7.15 – Zone de texture potentiellement accédée lors du rendu d'une tuile.

L'emploi de caches statistiques standards n'est pas souhaitable dans notre cas. En effet, les latences induites dans les traitements ainsi que dans le pipeline des traitements par les défauts de caches peuvent conduire à la perte de l'assurance du temps réel. De plus, d'une façon générale, les accès aux mémoires sont optimaux lors d'accès par grands blocs de données. Néanmoins, il est possible de masquer le chargement des données à l'intérieur du pipeline général de traitement si nous assurons que le temps de chargement est inférieur au temps de référence du macrocycle. Or, la connaissance du profil pire cas permet de dimensionner les débits dans cette optique-ci.

Ainsi, nous masquons à l'intérieur du pipeline de traitement général au niveau tuile les accès à la mémoire des zones de texture : nous effectuons donc un chargement *a priori* avant le début des traitements nécessitant ces données de texture. Cependant, afin de garantir une utilisation minimaliste de la bande-passante, nous ne souhaitons pas recopier toute la zone que définit le pire cas, mais bien la zone de texture couramment indispensable. Cette utilisation optimale de la bande-passante se justifie également par un soucis de consommation et de charge du bus : la libération de bande-passante permet, en outre, au processeur hôte d'obtenir l'accès pour lui-même à la mémoire pour effectuer d'autres opérations.

Pour cela, nous mettons en place un calcul *a priori* des adresses des données afin d'accéder *a posteriori* à celles-ci de façon déterministe rendant, de ce fait, inutile un cache statistique. Cette détermination des adresses est effectuée lors du balayage de la tuile. Ces adresses sont analysées pour déterminer la zone de texture à accéder puis conservées pour leurs utilisations ultérieures. Nous dénommons cette opération *DST* pour détermination du secteur de texture qui constitue donc un macrocycle complet du pipeline. Ce macrocycle est suivi par l'accès effectif au secteur déterminé lors du macrocycle suivant qui le recopie localement. À l'aide des adresses conservées et des données recopiées, les opérations de rendu (filtrage, etc.) peuvent alors avoir lieu. Ce rendu convie alors à un reparcours complet de la tuile à partir des coordonnées de texture pour accomplir les derniers traitements au niveau pixel. Nous rejetons alors dans

un macrocycle supplémentaire les opérations de sauvegarde c.-à-d. la recopie en mémoire centrale de la tuile résultante. La figure 7.16 illustre les différentes opérations effectuées durant chaque macrocycle ; nous avons utilisé les notations suivantes :

- Lg indique le chargement des informations de géométrie ;
- Init indique le processus d'initialisation ;
- DST indique le processus de détermination du secteur de texture ;
- Lt indique le chargement des informations de texture ;
- Render indique le processus de rendu final ;
- Save indique la recopie de la tuile résultante. Cette sauvegarde n'intervient qu'une fois pour une tuile et dépend donc du nombre d'atomes à traiter pour cette tuile particulière.

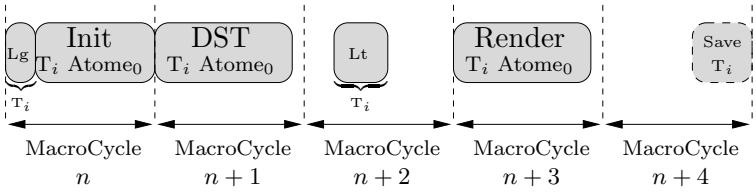


FIG. 7.16 – Découpage temporel en macrocycle des opérations de rendu d'une tuile.

Afin d'illustrer l'organisation macropipelinée de l'opérateur de parcours de tuile, la figure 7.17 complète le précédent découpage temporel. Elle indique les différents états de chaque étage de pipeline et retrace le rendu d'une tuile T_i composée de deux atomes Atome$_0$ et Atome$_1$. L'opération de sauvegarde optionnelle n'intervient alors que lors du macrocycle suivant celui consacré au rendu du dernier atome.

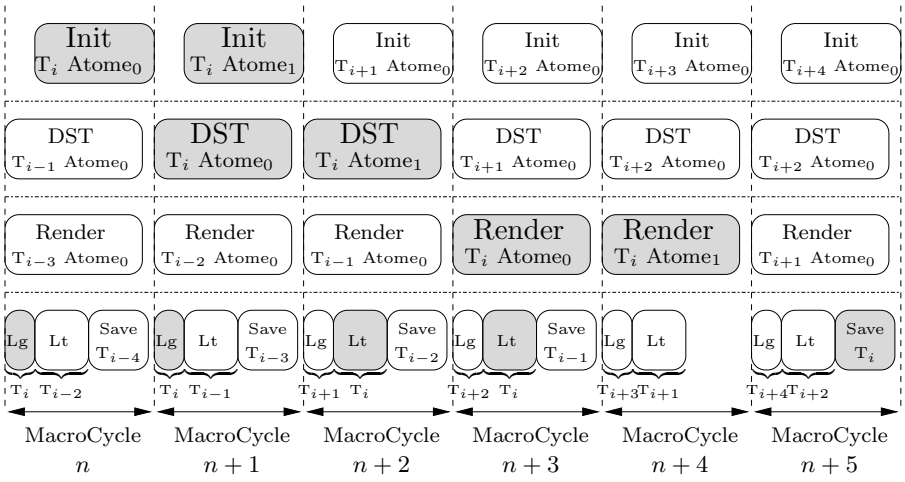


FIG. 7.17 – Macropipeline tuile complet.

Nous conclurons cette discussion en remarquant que le mécanisme de détection de zone de texture associé avec une organisation temporelle décorrélant l'accès à la mémoire et l'utilisation des données ainsi accédées n'est pas sans rappeler celui dont disposent les microprocesseurs actuels. Ceux-ci incorporent en effet du matériel spécialisé qui déroule les instructions en avance. Ce matériel identifie alors les `jump` à effectuer et déduit les instructions qu'il faut alors précharger afin d'éviter un défaut de cache d'instructions terriblement pénalisant.

7.3.3 Cache déterministe

La connaissance *a priori* des coordonnées de texture permet un accès aux seules données d'une texture d'un objet visuel pertinentes pour les traitements des pixels d'une tuile. Cependant, en fonction de la complexité des accès mémoires que nous nous autorisons, la détection de zone peut être de complexité échelonnée : un simple rectangle entourant sera le minimum et assurera un accès à un unique bloc bidimensionnel de données. De plus, en raison des contraintes d'alignement mémoire dont doit tenir compte la détection de zone, la zone effectivement chargée sera assurément plus grande que le secteur idéalement nécessaire. Nous mettons alors en œuvre un mécanisme qui nous permet de limiter au mieux cet aspect sous-optimal des accès mémoires.

La découpe en atome de taille fixe impose le reparcours de la tuile pour chacun tout en conservant la même référence de texture. Afin de maintenir l'homogénéité des traitements, un accès mémoire est effectué après chaque parcours de tuile comme l'indique la figure 7.17 : l'entrelacement des secteurs de texture nécessaires ne fait alors pas de doute. La figure 7.18 reprend l'exemple des atomes de la figure 7.12 pour illustrer l'entrelacement des zones de textures que définit chaque détection de zone.

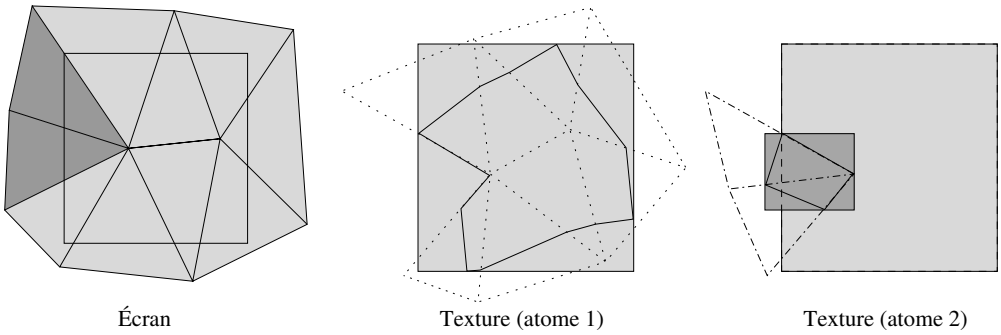


FIG. 7.18 – Entrelacement des zones de texture déterminées. Les transformations géométriques appliquées déforment la tuile définissant ainsi les éléments de texture nécessaires lors du filtrage. Le premier atome définit une première zone ; le second atome définit indépendamment une nouvelle zone. Nous avons indiqué la zone précédemment définie.

Les zones effectivement accédées et contenues en mémoire locale contiennent donc un certain nombre de redondances qui peuvent être détectées. Ces cas de redondances des accès aux données peuvent se produire lors du traitement d'une même tuile dans le cas où plusieurs parcours traitent le même objet visuel mais aussi entre les traitements de tuiles successives dans le cas particulier d'un unique objet impactant ces tuiles.

Grâce au rendu différé, il est possible d'identifier ces entrelacements de zones à accéder en mémoire et ainsi d'assimiler le fonctionnement de notre mémoire locale à celui d'un cache. Pour cela, nous encapsulons la mémoire locale dans une unité de gestion de la mémoire. Initialement chargée de la détection de zone, elle est désormais également en charge d'un calcul différentiel des zones détectées entre plusieurs parcours de tuile pour un même objet visuel. La figure 7.19 illustre le cas de deux tuiles consécutives où, encore une fois, des redondances d'accès sont effectives et indique les recouvrements que

notre unité de gestion mémoire se charge d'identifier.

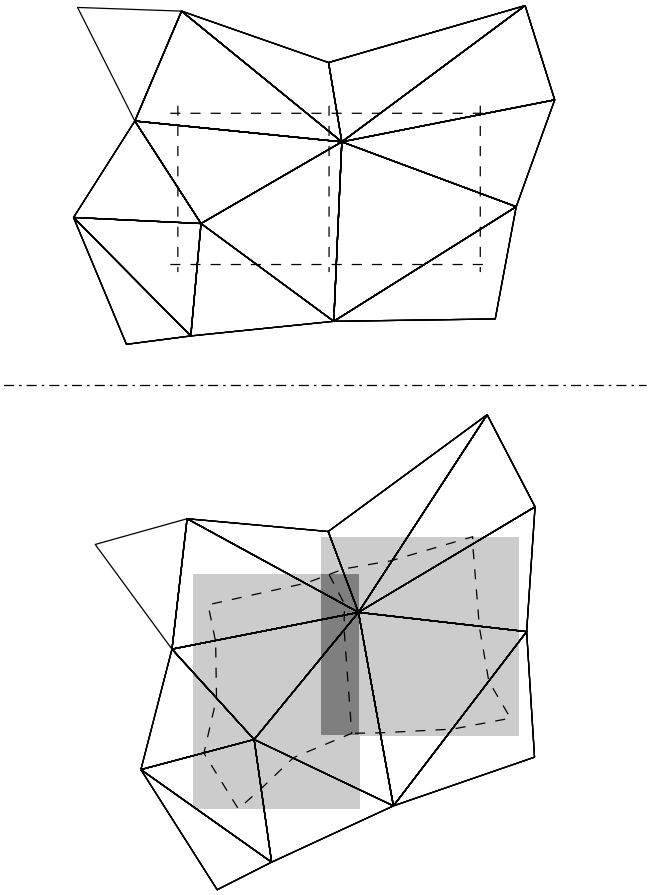


FIG. 7.19 – Zones de texture consécutivement déterminées pour deux atomes. Indications des rectangles entourants en gris clair : la partie foncée indique le recouvrement des secteurs de texture.

En simplifiant notre description aux cas de rectangles entourants, cette unité de gestion de la mémoire peut déterminer le déplacement global de la zone. Compte tenu des dimensions de la précédente zone accédée et de la zone courante détectée, elle détermine également la zone réelle à accéder en mémoire centrale. Toutefois, bien que de taille réduite, la mémoire locale doit pouvoir stocker les différentes zones sans conflit.

Nous définissons une taille minimale de mémoire locale à partir des spécifications des transformations maximales que peut subir l'ensemble d'une tuile et des contraintes associées d'alignement mémoire : $L_\mathrm{m} \times L_\mathrm{m}$. Remplie partiellement par la zone déterminée par le parcours d'un premier atome, elle est complétée par la nouvelle zone. Cependant, compte tenu de l'évolution du déplacement global de la zone, la gestion de la mémoire locale doit assurer le lien de continuité entre les coordonnées de texture et les adresses physiques : nous considérerons donc cette mémoire comme torique, ce qui consiste à effectuer un calcul modulo L_m sur les adresses. L'unité de gestion de la mémoire dispose ainsi de l'assurance d'une continuité de parcours à partir d'un point de départ (x_0, y_0) quelconque pris dans la mémoire : la figure 7.20 illustre ce point-ci.

Cependant, comme nous l'avons signalé, deux tuiles consécutives peuvent partager des données communes. Désormais, nous devons considérer que chaque zone approche la

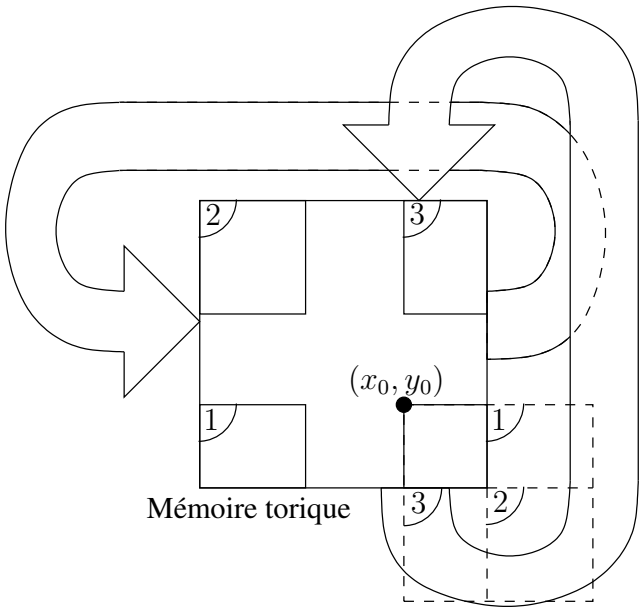


FIG. 7.20 – Continuité de la texture à l'intérieur de la mémoire locale.

taille $L_\mathrm{m} \times L_\mathrm{m}$ et, tandis que la première zone est lue par le processus de rendu (filtrage, etc.), le complément de la seconde zone est recopié depuis la mémoire centrale. Cette recopie ne doit pas écraser les données précédemment contenues. Une multiplication par quatre de la taille de la mémoire locale assure aisément le non-chevauchement (torique-ment parlant) des différentes zones comme l'illustre la figure 7.21. Malheureusement, cette multiplication augmente la quantité de mémoire locale ($4L_\mathrm{m}^2$) de notre opérateur.

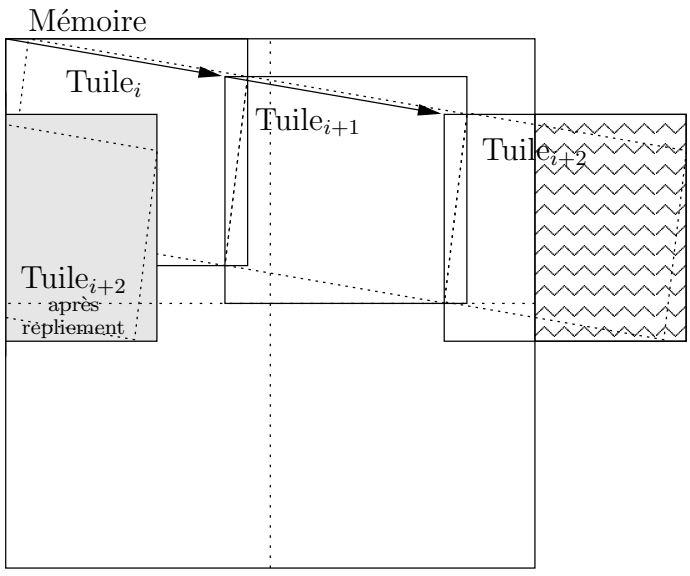


FIG. 7.21 – Mémoire locale agrandie garantissant le non-chevauchement des données. Seule la troisième zone peut présenter un conflit avec la première zone ; cependant, les accès en lecture et écriture sont ici temporellement disjoints.

Nonobstant les risques de conflit mémoire, il est possible de limiter la taille de la mémoire à $2L_\mathrm{m}^2$ en réorganisant judicieusement les données dans la mémoire et en conservant à chaque nouvelle étape de chargement une information sur la position des

données précédemment stockées. Pour ce faire, nous ajoutons deux plans de bits ($P_{\text{écr.}}$ et $P_{\text{lec.}}$) de taille L_{m}^2. Le plan $P_{\text{lec.}}$ de lecture, constant durant toute la phase de lecture des données de la tuile t indique dans quelle zone de la mémoire locale sont contenues les données valides : la zone de la mémoire locale contenant la donnée d'adresse (i, j) vaut $P_{\text{lec.}_t}(i, j)$.

Le plan $P_{\text{écr.}}$ d'écriture évolue, quant à lui, au long de la recopie des données en provenance de la mémoire locale, données nécessaires pour le traitement de la tuile $t+1$. Identique au début de la phase de recopie au plan $P_{\text{lec.}}$, il indique à l'inverse dans quelle zone mémoire il convient d'écrire pour éviter tout conflit. Il se met ainsi à jour continuellement afin de constituer au début du prochain macrocycle un nouveau plan qui sera recopié dans le plan $P_{\text{lec.}}$: ce plan représente une photographie du plan $P_{\text{écr.}}$ à la fin d'une phase de chargement. Une donnée écrite en adresse (i, j) disposera alors d'une nouvelle adresse de lecture $P_{\text{lec.}_{t+1}}(i, j) = \overline{P_{\text{lec.}_t}(i, j)}$; les valeurs des plans correspondantes à des données en adresse (i', j') non réécrites ne variant pas : $P_{\text{lec.}_{t+1}}(i', j') = P_{\text{lec.}_t}(i', j')$. La figure 7.22 illustre à sa manière cette évolution au cours du temps : les zones mémoires sont dénommées zone « blanche » et zone « noire ». Le plan $P_{\text{écr.}}$ étant initialisé entièrement à « noir », l'écriture d'un premier bloc de données met à jour une partie de ce plan en « blanc » : cette mise à jour est symbolisée par un grisé clair. Le chargement d'un deuxième bloc de données lors de l'étape 1 complète la zone « blanche » et sauvegarde les données conflictuelles en zone « noire » en mettant à jour cette fois-ci une partie du plan originellement « blanc » à « noir » : cette mise à jour est symbolisée par un grisé foncé.

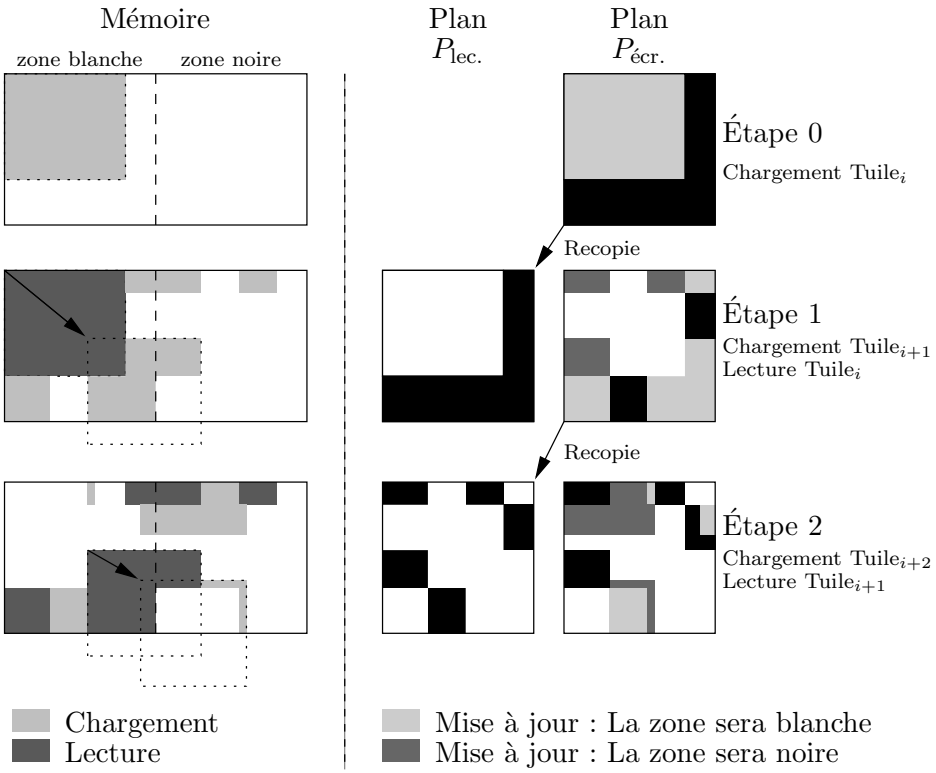


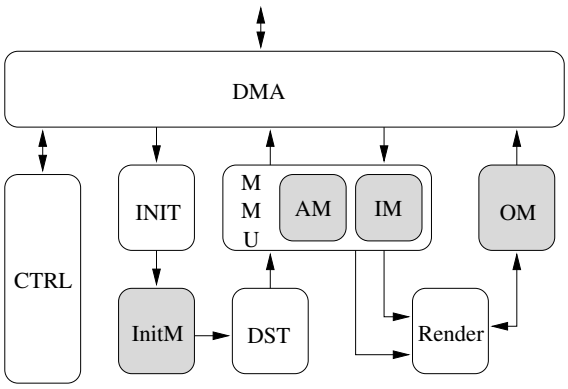
FIG. 7.22 – Évolution des différents plans mémoires.

La gestion que nous proposons et qui a fait l'objet d'un dépôt de brevet [118]

assure une bande-passante proche de l'idéal grâce à la conservation du souvenir du passé le plus récent. Les zones de recouvrement entre deux zones consécutives ne sont plus recopiées comme l'obligerait une mémoire double qui permettrait une recopie de la zone de texture pour la tuile $t + 1$ pendant la lecture des données de la zone pour la tuile t. Néanmoins, l'ajout de nos deux plans de bits permet de limiter l'augmentation de la taille de la mémoire : au lieu d'un facteur quatre correspondant à la figure 7.21, la taille de notre système approche la taille minimale d'une mémoire double pour une complexité légèrement supérieure.

7.3.4 Architecture d'un opérateur

La figure 7.23 présente spacialement une découpe en bloc d'un opérateur effectuant un rendu d'une tuile en s'appuyant sur le macropipeline précédemment présenté : un contrôleur de DMA *(Direct Memory Access)* effectue l'interface entre cet opérateur et le monde extérieur. Le contrôleur CTRL gère le séquencement des tâches internes : le bloc INIT effectue l'initialisation et fournit les coefficients nécessaires au parcours géométrique à la mémoire d'initialisation InitM afin que le bloc DST puisse y accéder lors du prochain macrocycle du pipeline. Les adresses que détermine le bloc DST sont délivrées à l'unité de gestion de la mémoire MMU où la zone de texture est explicitement calculée. Les adresses sont entreposées dans la mémoire d'adresses AM tandis que la recopie des données au travers du DMA s'effectue dans la mémoire d'entrée IM *(Input Memory)*. Les adresses ainsi que les données sont ensuite transmises au bloc de rendu Render qui effectue les opérations de mélange, le cas échéant, et de restockage des pixels dans la mémoire de sortie OM *(Output Memory)*. Une fois le traitement d'une tuile complété, l'écriture en mémoire centrale des données de la mémoire de sortie est accomplie au travers du DMA.



DMA : Direct Memory Access
CTRL : Contrôleur
INIT : Initialisation
AM : Mémoire d'adresses
InitM : Mémoire d'initialisation
IM : Mémoire d'entrée
OM : Mémoire de sortie
Render : Rendu
DST : Détermination du Secteur de Texture
MMU : Unité de gestion de la mémoire

FIG. 7.23 – Diagramme en bloc d'un opérateur de rendu par tuiles. Les parties grisées indiquent les éléments de mémorisation.

La figure 7.24 se focalise plus particulièrement sur le bloc de gestion mémoire : les adresses (u, v) fournies par l'opérateur de détection de secteur de texture DST concernant

la tuile $t+2$ sont analysées au vol par le contrôleur de gestion des mémoires avant d'être emmagasinées dans la mémoire AM qui, physiquement, consiste en une mémoire FIFO *(First-In First-Out)*. Tandis que cette analyse détermine la future zone, le contrôleur effectue l'accès à la zone $t+1$ précédemment déterminée au travers du DMA. Les données ainsi recopiées localement dans la mémoire IM. Les adresses de recopie sont issues du contrôleur qui assure la correspondance entre les adresses globales et les adresses physique locales, complétées par les informations délivrées par le plan de bits de la mémoire d'écriture W. Enfin, parallèlement, le rendu de la tuile t est effectué par le bloc Render à l'aide des données $D(t)$ issues de la lecture du bloc IM adressé par les adresses (u, v) contenues dans AM complétées par le plan de bits de lecture R. Ces adresses, utiles pour les opérations de rééchantillonnage, sont également propagées au bloc de rendu.

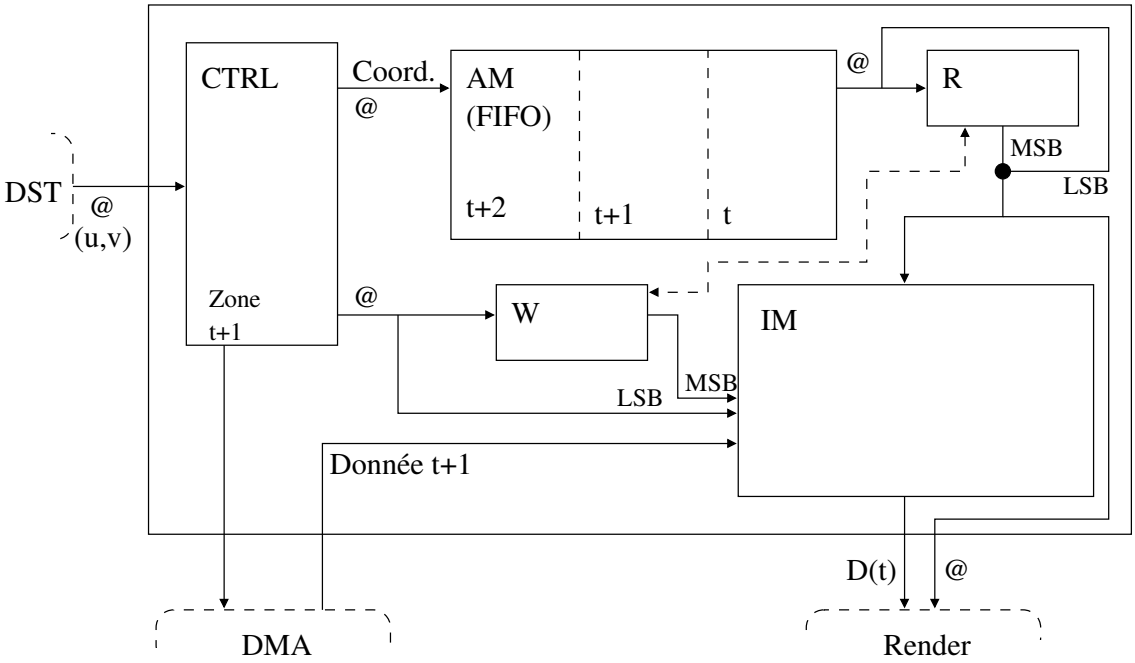


FIG. 7.24 – Bloc de gestion mémoire.

7.4 Bilan

Nous avons présenté dans ce chapitre une organisation de rendu par tuile. Nous avons montré comment les différences d'objets manipulés par la vidéo ou la synthèse graphique pouvaient être masquées, au niveau pixel, dans un pipeline de traitements aussi générique que possible. L'ajout de mémoire locale dispense de l'échange d'un grand nombre de données entre la mémoire centrale et un coprocesseur de traitement spécifique. La découpe du traitement complet de l'image en un macropipeline tuile conforte la minimisation de la bande-passante par la mise en place d'un mécanisme de prédiction dynamique des zones exactes de données nécessaires aux traitements. Enfin, nous noterons qu'une partie des résultats présentés lors de ce chapitre a donné lieu à une communication lors d'un colloque international [119].

Chapitre 8

Transformations affines et filtrages

Le terme de placage de texture cache deux interventions en une. Tout d'abord, il fait intervenir de façon plus ou moins implicite une transformation géométrique pour produire tout ou partie de l'image de sortie à partir d'une image d'entrée. Ensuite, un rééchantillonnage, comportant une opération de filtrage, est nécessaire.

Ces deux opérations sont à la fois communes aux domaines de synthèse graphique et de la vidéo et gourmandes en puissance de calcul. C'est pourquoi nous aborderons notre étude relative au problème spécifique du placage de texture sous la contrainte de limiter la puissance de calcul et donc les coûts subséquents d'un opérateur matériel unique. Enfin, les techniques de rééchantillonnage étant indissociables de la qualité visuelle subjective de l'image finale, nous discuterons une méthode de rendu de qualité vidéo adaptée au cas du rendu graphique.

8.1 Introduction

Nous avons présenté, partie III paragraphes 3.3.1 et 3.3.2, les aspects de filtrage et d'organisation des traitements. Nous les approfondissons un peu plus en détail dans le cas des transformations affines.

8.1.1 Parcours arrière

Le placage de texture par parcours arrière ou *backward texture mapping* correspond au parcours orienté image. Piloté par les pixels de sortie, c'est un parcours de ce type que nous avons exposé dans le chapitre précédent. Les équations (8.1) et (8.2) rappellent les expressions des coordonnées de texture et des coefficients affines, coefficients qui nécessitent 13 soustractions, 10 multiplications et 4 divisions (ou 1 inversion et 4 multiplications supplémentaires) pour être calculés lors de la phase d'initialisation (cf. paragraphe 3.1.2.3 et chapitre précédent).

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} \frac{\partial u}{\partial x} & \frac{\partial v}{\partial x} \\ \frac{\partial u}{\partial y} & \frac{\partial v}{\partial y} \end{pmatrix} \begin{pmatrix} x - x_{\text{ref}} \\ y - y_{\text{ref}} \end{pmatrix} + \begin{pmatrix} u_{\text{ref}} \\ v_{\text{ref}} \end{pmatrix} \tag{8.1}$$

$$\frac{\partial u}{\partial x} = \frac{\Delta u_{01}\Delta y_{02} - \Delta u_{02}\Delta y_{01}}{\Delta x_{01}\Delta y_{02} - \Delta x_{02}\Delta y_{01}} \tag{8.2}$$

Ce parcours présente l'avantage de relier directement la puissance de calcul nécessaire lors du parcours des géométries au nombre de pixels de l'image à générer. L'inconvénient provient de la relecture des données pour l'opération de filtrage. Malgré l'utilisation de mémoire cache, cette opération peut avoir besoin de relire plusieurs fois la même donnée augmentant ainsi la bande-passante mémoire.

8.1.2 Parcours avant

Le parcours avant ou *forward texture mapping* est particulièrement adapté au cas de la vidéo. Ce parcours présente l'avantage de disposer de filtres (re)connus pour la qualité visuelle résultante, ce pour une consommation de bande-passante mémoire minimale puisque ce parcours ne lit qu'une unique fois chaque donnée d'entrée. La figure 8.1 présente deux profils de filtres polyphases de quatre coefficients pour 8 et 16 phases respectivement.

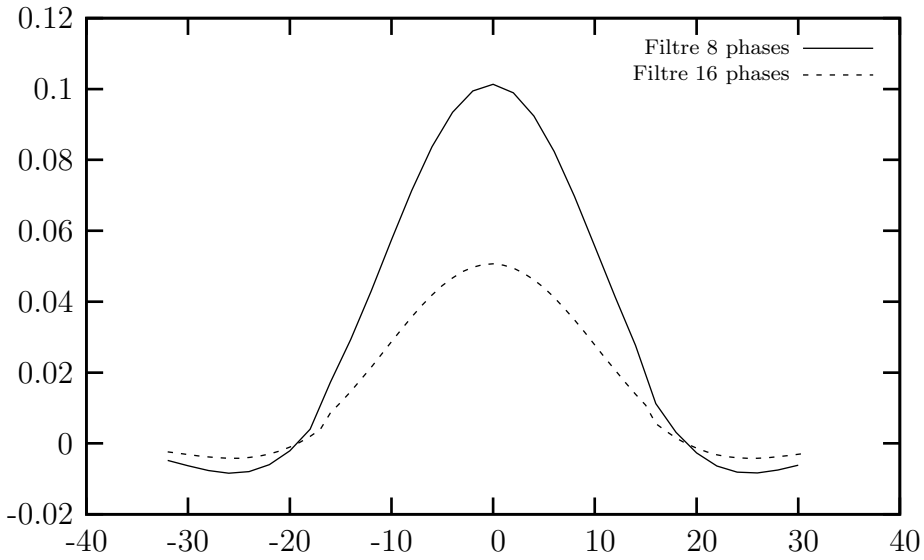


Fig. 8.1 – Profils de filtres polyphases.

Ce parcours est toutefois peu adapté à la synthèse graphique. Néanmoins, une équipe de chercheurs néerlandais au sein de la société Philips a intégré dans le pipeline graphique un parcours piloté par la lecture des données de texture alimentant un filtre

de rééchantillonnage couramment employé en vidéo [120, 121]. Ce parcours autorise naturellement, concomitamment avec une mémoire de contributions similaire à un *A-buffer*, un anticrénelage de bord de polygone : l'étendue de l'empreinte du filtre permet de calculer des contributions partielles pour des pixels proches des frontières de la primitive traitée [122].

Dans le cas d'une réduction géométrique, la densité des texels assure un calcul adéquat des contributions pour chaque pixel de l'écran comme l'indique la figure 8.2. L'étendue de l'empreinte du filtre limite le facteur d'agrandissement possible pour maintenir une densité minimale des texels sous ce filtre. Une discussion concernant ce parcours dans le cas d'une réduction est effectuée par Popescu [123] : il étudie exhaustivement les contraintes nécessaires et suffisantes afin d'assurer un recouvrement complet d'un triangle de l'écran lors d'un parcours.

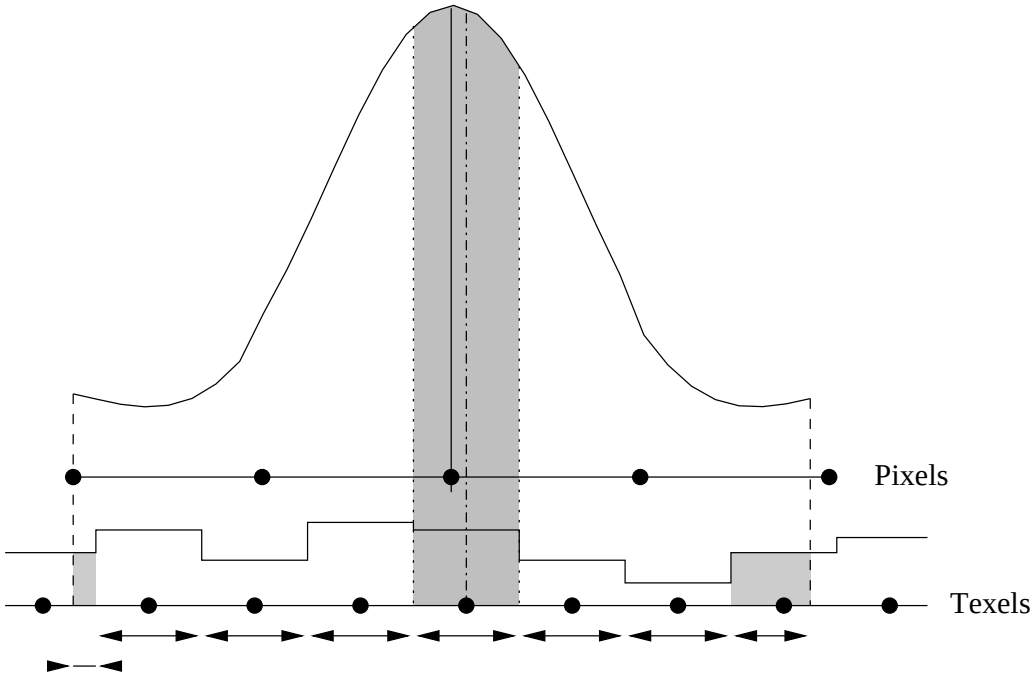


FIG. 8.2 – Calcul de contribution d'un texel au pixel courant.

En ce qui concerne la puissance de calcul, le calcul des coefficients de parcours réclame la même puissance que dans le cas d'un parcours arrière. Cependant, dorénavant, la puissance de calcul du parcours n'est plus directement reliée au nombre de pixels générés mais à celui des texels lus. Dans le cas d'une réduction, le nombre de texels traités et filtrés est donc supérieur au nombre de données en sortie. L'usage de textures préfiltrées apporte alors une aide à la limitation de la puissance de calcul.

Malgré un processus d'initialisation aussi coûteux que dans le cas d'un parcours arrière, ce parcours présente l'avantage de la mise en place naturelle en flux de données d'un traitement de qualité vidéo compatible avec le pipeline graphique. Néanmoins, la puissance de calcul de la phase d'initialisation reste importante comparativement à la puissance de calcul nécessaire pour le balayage des pixels : nous l'avons étudiée dans le chapitre précédent. Nous présentons dans le prochain paragraphe une réorganisation du

mode de parcours des triangles qui permet de limiter les puissances de calcul nécessaires lors de l'initialisation.

8.2 Réorganisation du parcours des triangles

L'objectif de ce paragraphe est de démontrer qu'en réorganisant le mode de parcours des triangles, il est possible de rééquilibrer les puissances de calcul entre les différents processus mis en jeu lors du rendu. Ce rééquilibrage en faveur du processus d'initialisation permet alors de diminuer globalement la puissance de calcul de l'opérateur complet de transformation géométrique.

8.2.1 De la simplification de la transformation affine...

8.2.1.1 Présentation

Les maillages de triangles trouvent de nouvelles applications comme dans le codage hybride (SNHC cf. paragraphe 3.2) de la norme MPEG-4. Le suivi des nœuds du maillage d'une image à l'autre définit les vecteurs de mouvements de cette image. La compensation de mouvements entre les images s'effectue alors à l'aide de la déformation du maillage par le calcul d'une transformation affine. Le maillage de la première image étant libre, un choix judicieux de celui-ci autorise une simplification des calculs des coefficients affines [124, 125, 126].

La figure 8.3 reproduit le cas particulier ainsi employé par Badawy : un triangle isocèle rectangle aligné sur les axes de coordonnées ; quant à l'équation (8.3), elle reprend la précédente équation (8.2) appliquée au cas présent. Le calcul des coefficients affines requiert alors quatre soustractions et quatre divisions. Cependant, la maîtrise du diviseur permet de le fixer à une puissance de deux, valeur favorable à une implantation matérielle très simple : un simple décalage de bits.

$$\begin{cases} \frac{\partial u}{\partial x} & = & \frac{u_B - u_M}{c} \\ \frac{\partial v}{\partial x} & = & \frac{v_T - v_M}{c} \end{cases} \tag{8.3}$$

8.2.1.2 Problématique

La simplicité des équations des coefficients affines est cependant fondée sur la contrainte de la conformation particulière des triangles. Les équations nous offrent deux possibilités pour tirer parti de la simplification du calcul d'un coefficient affine :

- Un parcours arrière. L'écran est alors pavé de triangles rectangles isocèles. Malheureusement, il est délicat de contraindre ainsi une application dès la conception. Ainsi, il conviendra d'approcher un triangle quelconque par un maillage de triangles particuliers dont le nombre sera sans doute important (sans toutefois

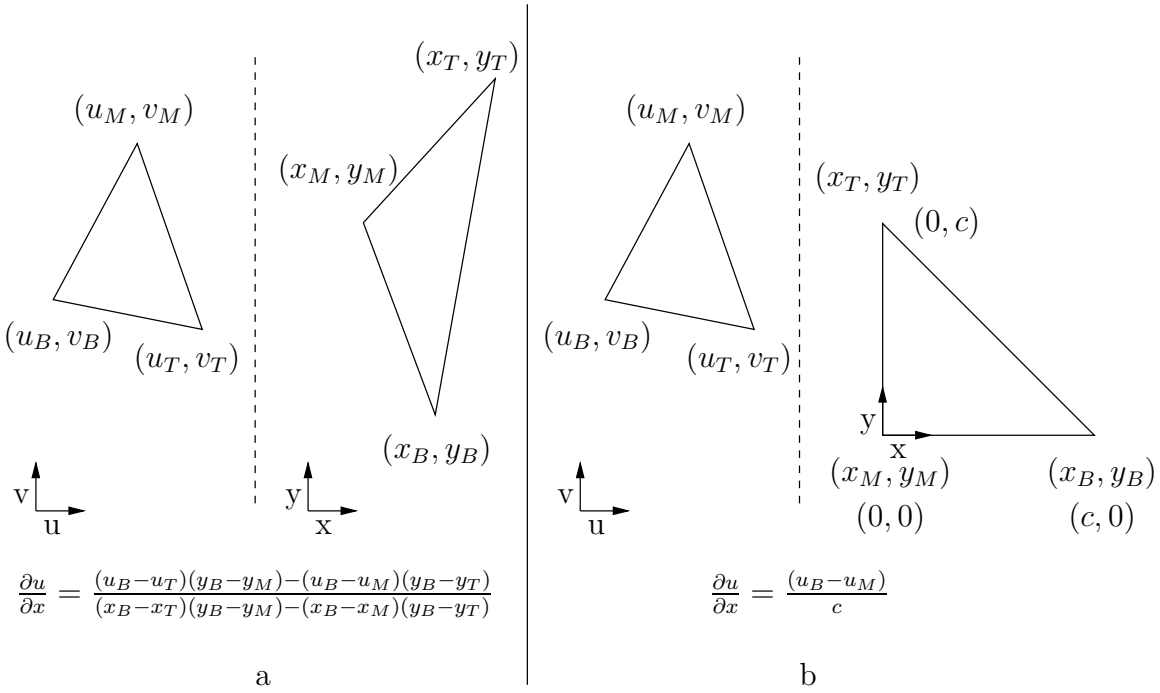


FIG. 8.3 – Transformation géométrique pour un triangle rectangle isocèle.

que nous l'ayons quantifié) impliquant de nombreux changements de contexte lors du rendu.
- Un parcours avant. Les triangles rectangles sont alors dans l'espace texture et la simplicité du parcours de ceux-ci accentuent les propriétés de la transformation affine. Cependant, la contrainte reste élevée pour une application dont nous ne maîtrisons pas le développement.

La simplification attendue des coefficients affines n'est donc pas possible avec les deux parcours précédents. Afin de conserver une large généralité, nous devons donc considérer des triangles quelconques dans les espaces texture et écran.

Nous introduisons alors un espace intermédiaire de parcours des triangles. Le choix du triangle sera donc laissé libre dans cet espace et servira d'espace origine pour le balayage de l'écran et la détermination des texels à filtrer dans l'espace texture. Il est donc nécessaire de calculer les transformations affines de cet espace intermédiaire vers les deux espaces écran et texture, puis calculer pour chaque pixel intermédiaire ses images correspondantes dans les deux autres espaces.

Nonobstant cette augmentation intrinsèque du nombre de calculs demandés, nous allons montrer que la simplification des calculs ainsi engendrés, tant au niveau de l'initialisation des coefficients de transformation affine que du parcours du triangle intermédiaire, permet de diminuer globalement la puissance de calcul d'un opérateur de rendu.

8.2.2 . . . À un nouvel espace de parcours des triangles

Un triangle est donc parcouru dans un espace intermédiaire dont nous noterons les coordonnées (s, t). Le choix du triangle dans cet espace, puis la manière de parcourir ce triangle déterminent les puissances de calcul qu'implique ce nouvel algorithme.

8.2.2.1 Choix du triangle intermédiaire

La figure 8.4 reprend le nouvel espace de parcours et les relations entre celui-ci et les deux espaces texture et écran. Des transformations τ et σ définies sur l'espace intermédiaire, nous déduisons la transformation affine arrière θ précédemment employée comme : $(u, v) = \theta(x, y) = \tau \circ \sigma^{-1}(x, y)$. Le triangle intermédiaire est déterminé par le choix de la coordonnée c. Nous allons voir tout d'abord comment ajuster au mieux ce choix puis une première adaptation du choix du triangle complet.

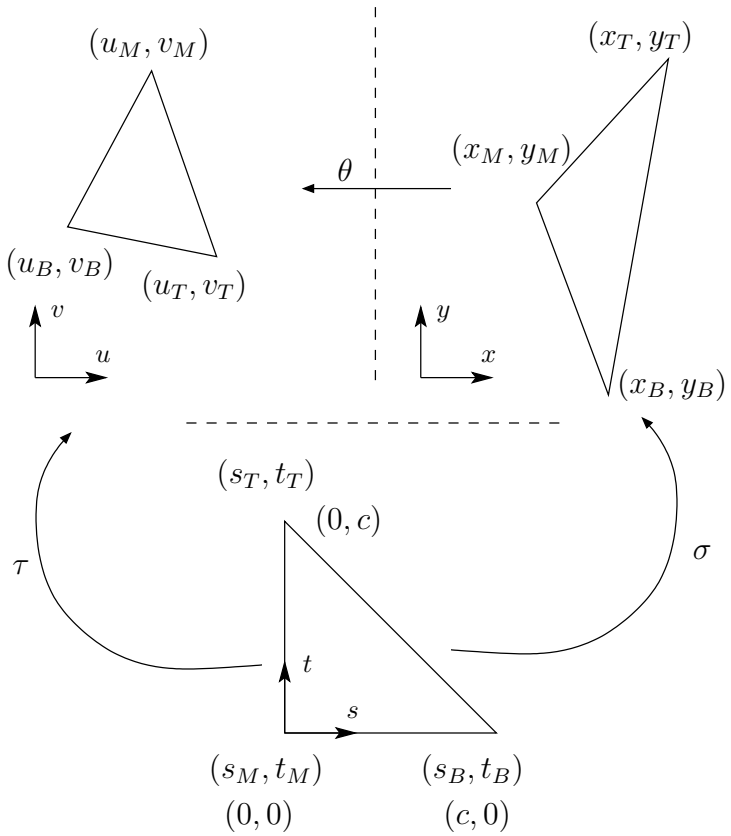


FIG. 8.4 – Les différents espaces de coordonnées.

Longueurs des côtés

Nous souhaitons équilibrer les puissances de calculs dans le nouvel espace avec celles précédemment nécessaires pour un parcours traditionnel dans l'écran. Pour cela, il convient d'ajuster au mieux le nombre de pixels intermédiaires parcourus (et donc la taille du triangle intermédiaire) sur celui de l'espace écran. Cependant, la transformation affine

simplifiée impose une transformation affine en avant : ce type de parcours convenant mieux dans le cas d'une réduction, nous choisissons le triangle intermédiaire plus grand que le triangle destination. Si le choix du côté correspondant à l'hypoténuse est trivial (nous supposerons qu'il s'agit du segment [TB]), le choix d'une longueur unique pour les côtés adjacents de l'angle droit peut conduire à une faible corrélation entre les surfaces des deux triangles comme sur le cas illustré sur la figure 8.5.

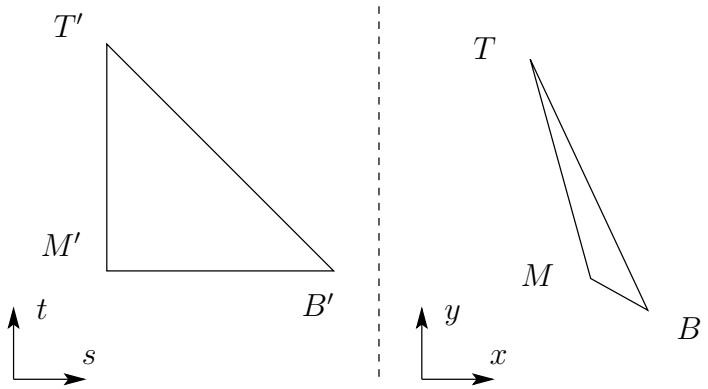


FIG. 8.5 – Large disproportion entre les deux triangles.

Afin de limiter cette disproportion, nous commençons par relâcher la contrainte d'un triangle isocèle : seul nous importe que les longueurs des côtés adjacents à l'angle droit soient des puissances de deux. Nous disposons désormais d'un second degré de liberté dans le choix du triangle intermédiaire qui est désormais défini par deux longueurs. Sans perte de généralité, nous nous intéresserons à la détermination de la longueur a du côté correspondant au segment [TM]. Cette longueur vérifie la définition 8.2.1 assurant ainsi un facteur de zoom compris entre un et deux : nous discutons une méthode exacte puis une méthode approchée afin de calculer les longueurs des différents côtés pour déterminer l'encadrement adéquat.

Définition 8.2.1 $a = 2^n$ *avec* $2^{n-1} < TM \leq 2^n$

Distance euclidienne Déterminer a revient à déterminer n. La longueur exacte TM est, bien entendu, définie par la distance euclidienne entre les points T et M donnée par l'équation (8.4).

$$\mathrm{TM} = \sqrt{(x_\mathrm{M} - x_\mathrm{T})^2 + (y_\mathrm{M} - y_\mathrm{T})^2} \tag{8.4}$$

Bien que généralement l'extraction d'une racine carrée ne soit pas immédiate [127], nous pouvons nous contenter de calculer et d'encadrer TM^2. De ce calcul, trois additions et deux multiplications, nous obtenons l'encadrement (équation (8.5)) :

$$2^{m-1} < \mathrm{TM}^2 \leq 2^m \tag{8.5}$$

Ainsi, n se déduit de m par une division par deux et un arrondi supérieur. Le calcul de m s'effectue par une détection de premier « 1 » pouvant s'effectuer en même temps que l'addition finale de TM^2. Au total, cette méthode requiert quatre additions et deux multiplications.

Approximation de la distance euclidienne Le calcul de n permettant une approximation de la valeur de TM^2, le calcul exact de cette dernière est superflu. Plus généralement, les normes des espaces vectoriels (sur $\mathbb{R}$) de dimension finie sont toutes équivalentes. Outre la distance euclidienne issue de la norme $\|.\|_2$, nous pourrions utiliser des distances issues d'autres normes, en particulier les normes $\|.\|_1$ ou $\|.\|_\infty$ dont nous rappelons les définitions équation (8.6).

$$\text{Soit } \overrightarrow{\mathrm{OM}} = \begin{vmatrix} x \\ y \end{vmatrix} \text{ alors } \begin{cases} \left\|\overrightarrow{\mathrm{OM}}\right\|_1 &= |x| + |y| \\ \left\|\overrightarrow{\mathrm{OM}}\right\|_2 &= \sqrt{x^2 + y^2} \\ \left\|\overrightarrow{\mathrm{OM}}\right\|_\infty &= \max\left(|x|, |y|\right) \end{cases} \tag{8.6}$$

La figure 8.6 rappelle les sphères unités de ces trois normes et l'équation (8.7) leur relation d'équivalence. L'usage de la norme $\|.\|_\infty$ sous-estime la longueur sauf au prix d'une multiplication par $\sqrt{2}$ qui la surestime alors jusqu'à plus de 40 %. Cette erreur est la même que celle induite par la norme $\|.\|_1$. Une sous-estimation de la longueur du côté peut conduire à la détermination d'une puissance de deux immédiatement supérieure mais tout de même inférieure à la longueur réelle de ce côté. Les contraintes de parcours que nous nous imposons ne permettent pas ce cas de figure.

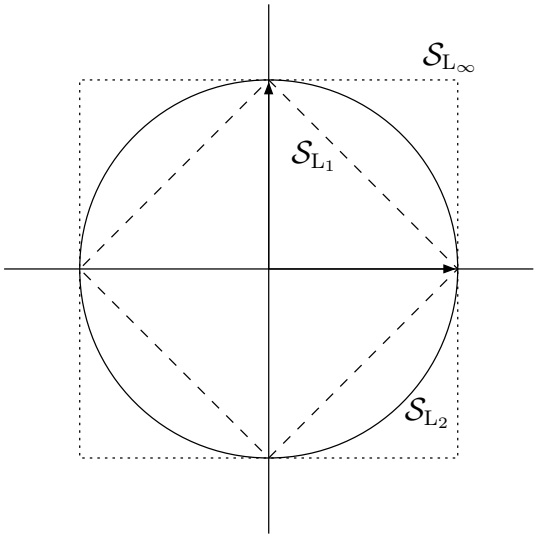


FIG. 8.6 – Sphères unités des normes $\|.\|_1$, $\|.\|_2$ et $\|.\|_\infty$.

$$\begin{cases} \frac{\sqrt{2}}{2}\|x\|_1 \leq \|x\|_2 \leq \|x\|_1 \\ \|x\|_\infty \leq \|x\|_2 \leq \sqrt{2}\|x\|_\infty \end{cases} \tag{8.7}$$

Or, si la demi-somme des distance $\|.\|_\infty$ et $\|.\|_1$ que reproduit la figure 8.7 surestime également la distance euclidienne, une analyse de cette distance produit l'équivalence de l'équation (8.8).

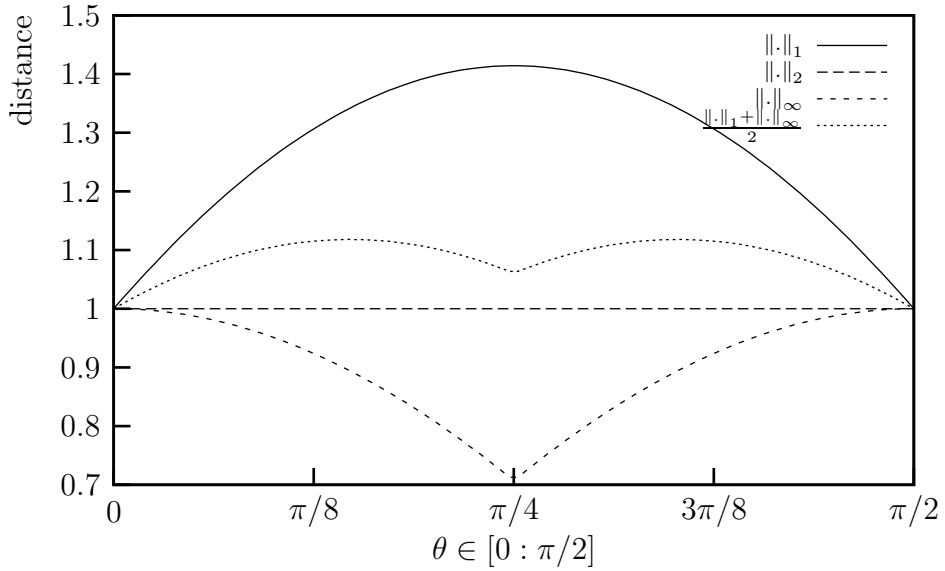


FIG. 8.7 – Comparaison des distances induites pour un point appartenant à la sphère unité $\mathcal{S}_{\|.\|_2}$.

$$\|x\|_2 \leq \frac{\|x\|_1 + \|x\|_\infty}{2} \leq \frac{\sqrt{5}}{2}\|x\|_2 \tag{8.8}$$

La constante $\sqrt{5}/2$ n'induit qu'une surestimation maximale de 11,8 %. De plus, si chaque distance nécessite cinq additions (dont deux pour chaque valeur absolue) pour être calculée, la redondance des termes n'ajoute qu'une simple addition (où s'opère également la détection du premier « 1 ») pour le calcul de cette distance approximative. En définitive, le tableau 8.1 compare les complexités de ce calcul approché et du calcul direct du carré de la distance euclidienne.

	$\|.\|_2^2$	$\frac{\|.\|_1+\|.\|_\infty}{2}$
Additions	4	6
Multiplications	2	0

TAB. 8.1 – Comparaison des calculs pour la détermination de valeur de n.

Abaissement d'une hauteur

La liberté de choix des longueurs de deux côtés n'est pas suffisante pour assurer la plus totale corrélation entre le triangle intermédiaire parcouru et le triangle final. La figure 8.8 illustre ce propos dans le cas de triangles fortement obtus et propose une

nouvelle adaptation par l'abaissement d'une hauteur du triangle. Nous exposons maintenant l'enchaînement algorithmique nécessaire pour déterminer un nouveau triangle intermédiaire.

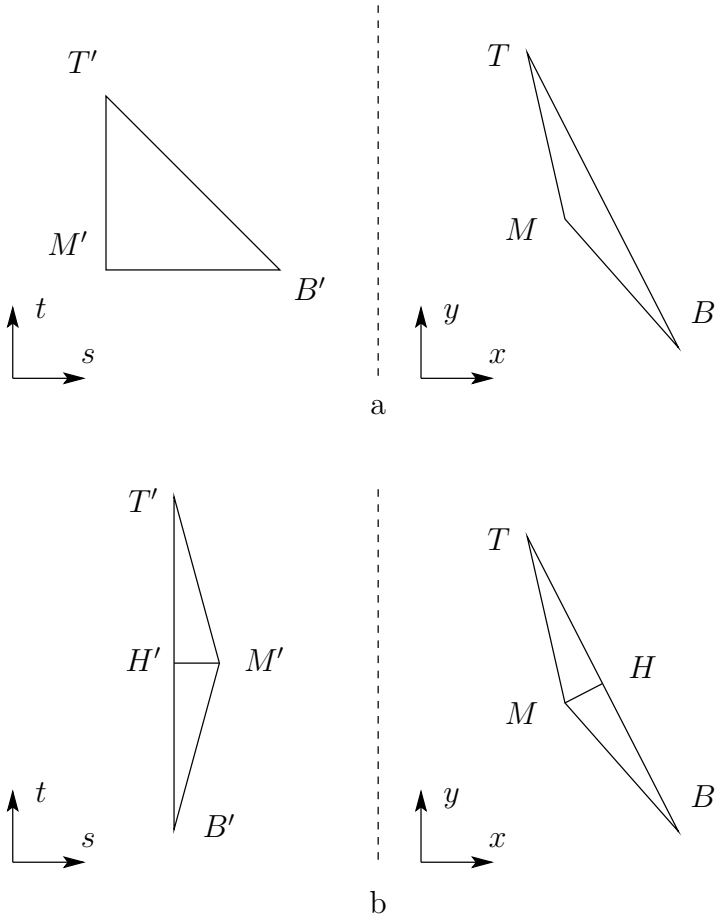


FIG. 8.8 – Découpe d'un triangle par abaissement d'une hauteur pour une meilleure adéquation entre les triangles intermédiaires et de l'écran.

Tri des sommets L'abaissement d'une hauteur interne permet la découpe d'un triangle en deux sous-triangles rectangles THM et MHB disposant désormais de trois degrés de liberté dans le choix des longueurs $a = \mathrm{T'H'}$, $b = \mathrm{H'B'}$ et $c = \mathrm{H'M'}$. Le choix systématique du plus grand côté afin d'abaisser la hauteur permet d'uniformiser les traitements des triangles obtus et aigus. Afin de garantir que les sommets T et B définissent systématiquement le plus grand côté, nous mettons en place un tri des sommets capable de réordonnancer ceux-ci. Comme l'éclaire la figure 8.9, un tri fondé uniquement sur l'ordonnée des sommets grâce à l'enchaînement de trois comparaisons-échanges n'assure pas cette garantie. Afin de déterminer si un tel cas pathologique intervient, nous effectuons un produit scalaire entre les vecteurs $\overrightarrow{\mathrm{TB}}$ et $\overrightarrow{\mathrm{TM}}$: le signe de ce produit scalaire détermine si le triangle est obtu en T. Dans ce cas, nous échangeons le rôle des points T et M. Le tri associé au calcul du produit scalaire conduit à une complexité de calcul de sept additions et deux multiplications.

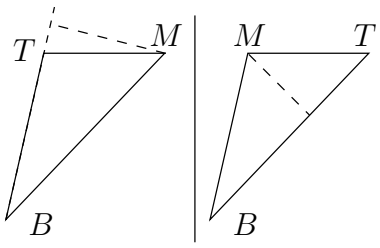


FIG. 8.9 – Abaissement d'une hauteur et triangles obtus. Un tri des simples ordonnées peut conduire à abaisser une hauteur extérieure au triangle dans des cas exceptionnels. Le calcul du produit scalaire définit alors de manière pertinente les sommets T, M et B.

Coordonnées de la base de la hauteur Le côté [TB] défini, la détermination des coordonnées de la base de la hauteur H est la prochaine étape. Nous ne pouvons résoudre le système linéaire définissant ce point comme intersection de la perpendiculaire au segment [TB] passant par M : des divisions interviendraient. De plus, nous devons également déterminer les coordonnées dans l'espace texture du point correspondant.

Nous pouvons cependant mettre en place une méthode itérative fondée sur l'annulation du produit scalaire entre les vecteurs $\overrightarrow{\mathrm{MH}}$ et $\overrightarrow{\mathrm{TB}}$. Or, le choix des sommets T et B assure que H vérifie l'équation (8.9).

$$\overrightarrow{\mathrm{TH}} = \alpha \overrightarrow{\mathrm{TB}} \text{ avec } \alpha \in [0,1] \tag{8.9}$$

Nous pouvons alors en déduire l'équation (8.10).

$$\begin{aligned} & \overrightarrow{\mathrm{MH}}.\overrightarrow{\mathrm{TB}} &&= 0 \\ \Leftrightarrow \quad & \left(\overrightarrow{\mathrm{MT}} + \overrightarrow{\mathrm{TH}}\right).\overrightarrow{\mathrm{TB}} &&= 0 \\ \Leftrightarrow \quad & \left(\overrightarrow{\mathrm{MT}} + \alpha\overrightarrow{\mathrm{TB}}\right).\overrightarrow{\mathrm{TB}} &&= 0 \\ \Leftrightarrow \quad & \overrightarrow{\mathrm{MT}}.\overrightarrow{\mathrm{TB}} + \alpha\overrightarrow{\mathrm{TB}}.\overrightarrow{\mathrm{TB}} &&= 0 \end{aligned} \tag{8.10}$$

Comme $\overrightarrow{\mathrm{TB}}.\overrightarrow{\mathrm{TB}}$ est positif, $\overrightarrow{\mathrm{MT}}.\overrightarrow{\mathrm{TB}}$ est donc négatif et l'introduction de la suite (α_n) obtenue par dichotomie permet d'approcher la valeur de α. Au fur et à mesure de la convergence de cette suite, il est possible de maintenir une suite (H_n) convergente vers la base de la hauteur, les suites (u_{H_n}) et (v_{H_n}) des coordonnées de texture du point correspondant ainsi que des distances (TH_n), (BH_n) et (MH_n) si nécessaire.

Ce calcul itératif nécessite les calculs préliminaires de $\overrightarrow{\mathrm{TB}}.\overrightarrow{\mathrm{TB}}$ et de $\overrightarrow{\mathrm{MT}}.\overrightarrow{\mathrm{TB}}$. Compte tenu des opérations déjà menées pour vérifier si le triangle est obtus ou non, ces calculs se limitent dans le pire à quatre multiplications et quatre additions. S'ajoutent également deux additions pour le futur maintien des suites (u_{H_n}) et (v_{H_n}). Quant au processus itératif, une itération effectue une addition pour déterminer le produit scalaire courant et son signe. En fonction de celui-ci, une addition pour chaque coordonnée (ou aucune) est effectuée. Le tableau 8.2 résume les calculs requis.

	Préliminaire	Itération
Additions	6	4
Multiplications	4	0

Tab. 8.2 – Calculs requis pour le calcul itératif du point H. Seuls les suites des coordonnées de H et du point correspondant dans la texture sont prises en compte.

Le point H déterminé, il est alors possible de définir les deux triangles intermédiaires T′H′M′ et H′M′B′. Bien que la transformation affine d'un espace vers un autre soit définie par chaque sous-triangle, la base commune limite à six (et non huit) le nombre de coefficients nécessaires.

Résumé : algorithme d'initialisation du nouveau parcours

L'introduction d'un nouvel espace de parcours du triangle dispose d'une première phase d'initialisation dont nous rappelons, figure 8.10, le pseudo-code. Le tableau 8.3 reporte les calculs nécessaires.

```
Pour chaque triangle
  Trier les sommets S0, S1 et S2 en ST, SM et SB
  Calculer le point H
    Initialiser le parcours itératif
    Tant que nécessaire
        Itérer
  Calculer les longueurs a = T′H′, b = H′B′ et c = H′M′
  Pour les deux sous-triangles T′H′M′ et H′M′B′
    Calculer les coefficients affines des deux transformations
    Calculer la pente de l'hypoténuse
```

Fig. 8.10 – Pseudo-code de l'initialisation du nouveau parcours.

	Add.	Mult.
Tri	7	2
H	$6 + 4$/it.	4
Distances	3×6	0
Coefficients	2×6	0
Total	$43 + 4$/it.	6

Tab. 8.3 – Calculs pour l'initialisation du nouveau parcours.

Compte tenu du contexte d'écran de mobile, la taille des triangles est assurée d'être limitée. Ainsi, la convergence des suites itératives est résolument rapide. En nous basant sur une implémentation logicielle de cet algorithme, un nombre d'itérations de cinq permet de définir la base de la hauteur avec une précision relative convenable. Nous pouvons maintenant comparer ce résultat à une technique d'initialisation classique telle que celle proposée dans le chapitre précédent (cf. tableau 7.9) ; le tableau 8.4 effectue cette comparaison.

	Classique	Proposée
Add.	31	64
Mul.	17	4
Div.	7	0

TAB. 8.4 – Comparaison des initialisations d'un parcours classique et du parcours proposé.

8.2.2.2 Parcours du triangle intermédiaire

Suite à la phase d'initialisation du nouveau parcours, le balayage des triangles peut débuter. Nous présentons deux parcours possibles selon l'ordre d'application des transformations affines présentées figure 8.4.

Parcours arrière-avant

Ce parcours correspond aux applications successives des transformations τ et σ (cf. figure 8.4). Naïvement, la transformation τ définit une position dans la texture permettant le calcul de la couleur du point intermédiaire correspondant à l'aide d'un premier filtrage. Par l'application de la transformation σ, les contributions de ce point aux pixels de l'écran sont calculées par un parcours en mode avant.

Les figures 8.11(a) à 8.11(d) présentent les résultats obtenus par une implémentation de cet algorithme. Les deux premiers résultats, obtenus avec des textures naturelles, paraissent relativement acceptables du point de vue visuel. Cependant, avec des textures synthétiques comportant de hautes fréquences, ces résultats présentent des artéfacts visuels. Deux causes principales interviennent ici :

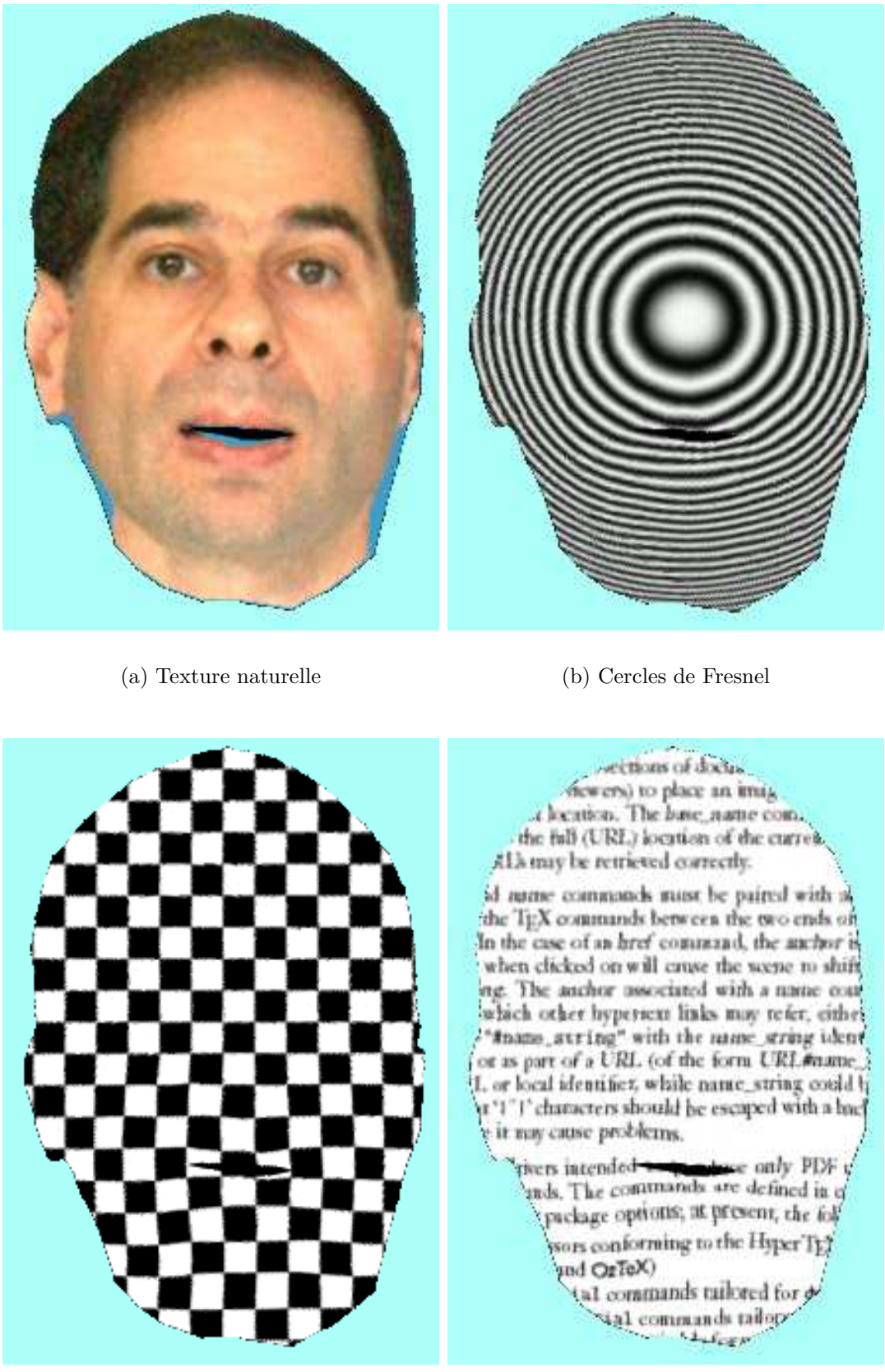
- La mise en œuvre du parcours en avant [128] ;
- La succession de deux filtrages.

La figure 8.12 qui met en œuvre deux rotations d'angles opposé sur un damier illustre ces effets.

Parcours avant-arrière

Le triangle intermédiaire n'étant qu'un simple espace de parcours, nous souhaitons conserver cette approche en évitant de calculer des couleurs intermédiaires. Ainsi, un seul filtrage sera nécessaire limitant les artéfacts visuels ainsi que le nombre d'opérations indispensable. Nous appliquons tout d'abord la transformation σ au point intermédiaire courant. En représentant géométriquement un point intermédiaire par un carré de longueur unité dans l'espace intermédiaire, nous définissons un nouveau maillage de parallélogrammes dans l'espace écran. Le choix des triangles intermédiaires et les remarques de Popescu [123] assurent qu'un tel parallélogramme contient zéro, un et exceptionnellement deux pixels écran (points mathématiques). La figure 8.13 présente sur une grille de pixels de l'écran un triangle pavé par de tels parallélogrammes.

Nous identifions alors, s'il y a lieu, le(s) pixel(s) contenu(s) dans le parallélogramme.



(a) Texture naturelle

(b) Cercles de Fresnel

(c) Damier

(d) Texte

FIG. 8.11 – Textures plaquées sur le maillage de la scène « Badin ».

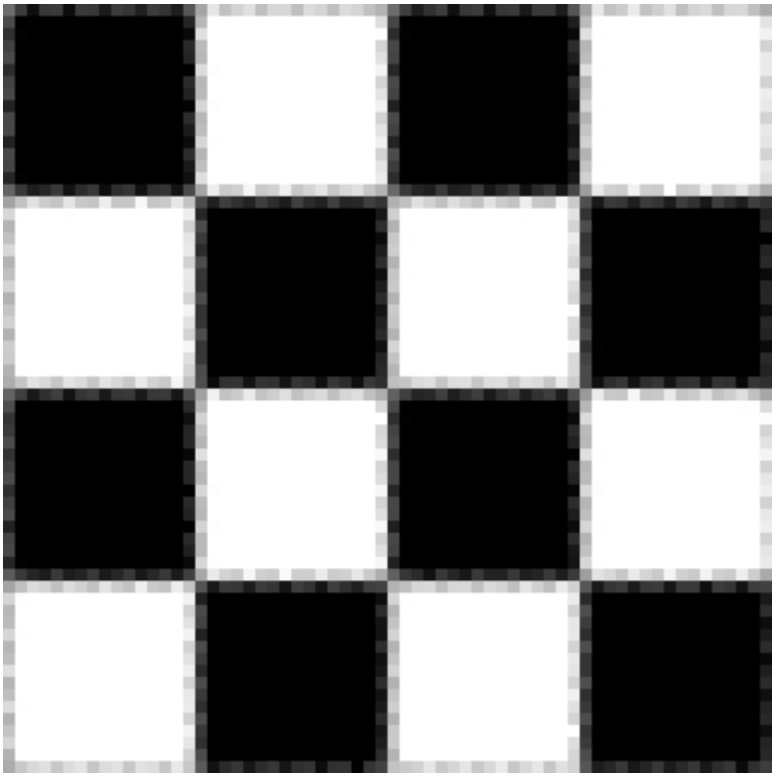


FIG. 8.12 – Damier après deux rotations d'angles opposés (détail). Le résultat est obtenu en utilisant l'API OpenGL.

Il est possible de déterminer la position exacte $(\alpha, \beta) \in [0,1]^2$ d'un tel pixel dans le repère constitué par les côtés du parallélogramme. Ces coordonnées coïncident avec les parties fractionnaires de l'antécédent du pixel dans l'espace intermédiaire. Le calcul de la position dans la texture de ce pixel se fait alors en deux étapes : la première consiste à appliquer la transformation τ sur le point intermédiaire en coordonnées entières. Nous effectuons alors une correction à l'aide des valeurs α et β et de la transformation τ. La figure 8.14 reprend les notations utilisées pour ce parcours.

La position du pixel à l'intérieur du parallélogramme définit un vecteur $\overrightarrow{l} + \overrightarrow{h}$ donné par l'équation (8.11).

$$\overrightarrow{l} + \overrightarrow{h} = \alpha \mathrm{d}\overrightarrow{s} + \beta \mathrm{d}\overrightarrow{t} = \begin{vmatrix} x \\ y \end{vmatrix} \tag{8.11}$$

L'équation (8.12) donne l'expression de α et β en fonction de ces coordonnées et de la transformation affine.

$$\begin{cases} \alpha = \frac{x\frac{\partial y}{\partial t} - y\frac{\partial x}{\partial t}}{PV} \\ \beta = \frac{y\frac{\partial x}{\partial s} - x\frac{\partial y}{\partial s}}{PV} \end{cases} \tag{8.12}$$

avec

$$PV = \frac{\partial x}{\partial s}\frac{\partial y}{\partial t} - \frac{\partial y}{\partial s}\frac{\partial x}{\partial t} \tag{8.13}$$

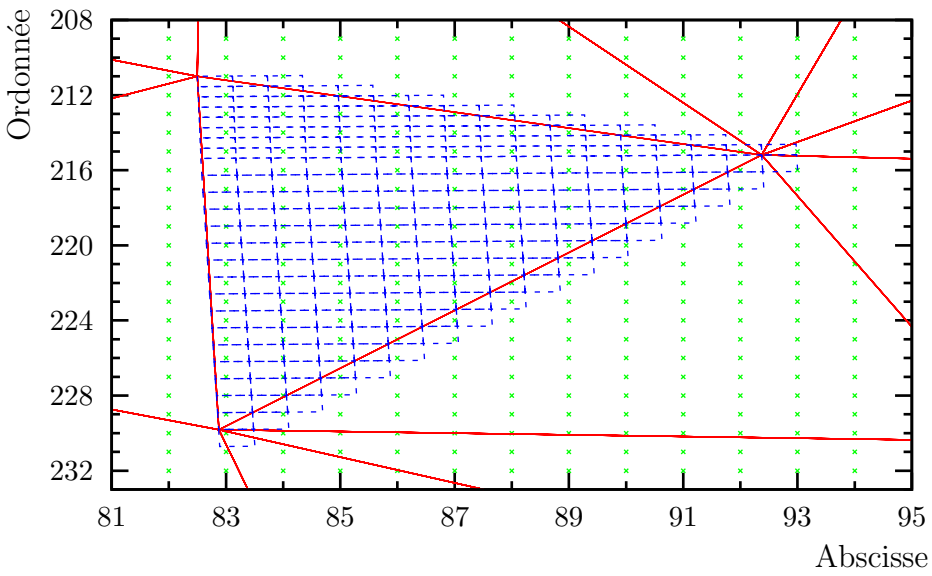


FIG. 8.13 – Triangles sur la grille de l'écran et parcours associé.

Les vecteurs $\overrightarrow{l'}$ et $\overrightarrow{h'}$ correspondent respectivement à $\alpha\overrightarrow{ds'}$ et $\beta\overrightarrow{ds'}$. Le calcul de α et β ne nécessite le calcul explicite ni de $\overrightarrow{l}$ ni de $\overrightarrow{h}$. Les coefficients multiplicateurs ainsi que la division, ne dépendants que de la transformation affine, peuvent être précalculés pendant la phase d'initialisation deux additions, deux inversions et 12 multiplications pour les deux sous-triangles. Lors du parcours, α et β requièrent quatre multiplications et quatre additions comprenant le calcul des coordonnées écran du vecteur $\overrightarrow{l} + \overrightarrow{h}$. La position exacte dans la texture du pixel est déterminée à partir de celle issue du point intermédiaire, rajoutant quatre additions et autant de multiplications. Il est ensuite possible d'appliquer un filtrage traditionnel (bilinéaire, etc.) comme dans n'importe quel parcours arrière.

En conclusion de ce parcours que nous retiendrons, nous noterons la mise en œuvre d'un parcours à double niveau de raffinement :

1° Basé sur le parcours en coordonnées entières des pixels intermédiaires, le premier raffinement permet un déplacement incrémental absolu aussi bien dans l'écran que dans la texture.

2° Basé sur un déplacement relatif par rapport à ce point d'appui, le second raffinement autorise une précision dans les calculs que nous pouvons ajuster au besoin.

Nous discutons, paragraphe 8.3, des questions d'implémentations matérielles et des résultats obtenus par ce parcours. En particulier, nous traiterons des calculs que nous rajoutons à l'initialisation de ce parcours.

Résumé : parcours du triangle

En résumé, nous présentons figure 8.15 le pseudo-code de parcours d'un triangle intermédiaire constitué de deux sous-triangles intermédiaires. Le tableau 8.5 retrace, quant à lui, les calculs nécessaires lors de la phase de parcours. La phase d'identification

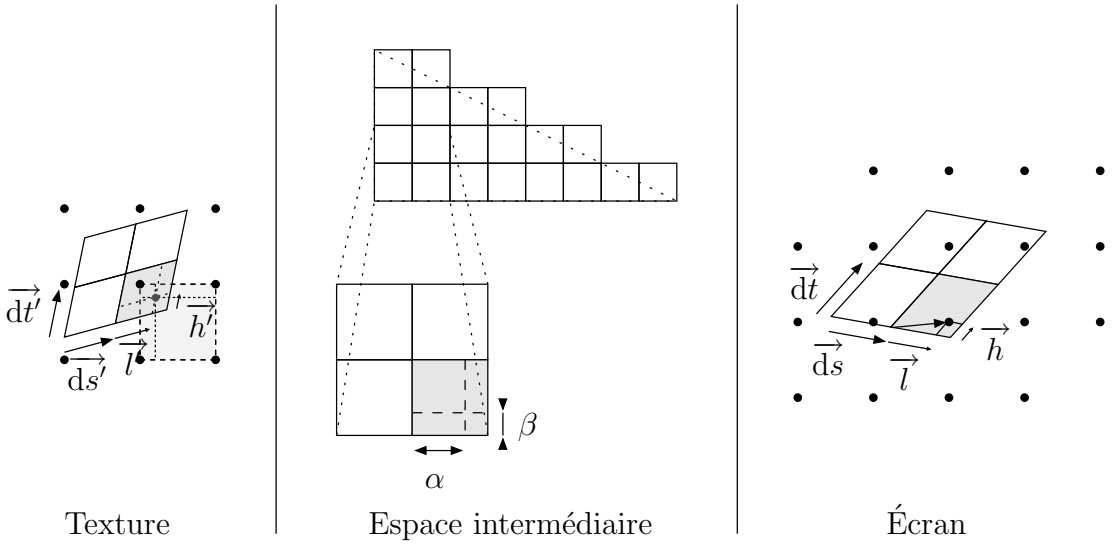


FIG. 8.14 – Notation pour le parcours avant-arrière.

n'ayant pas été optimisée, notre implémentation logicielle l'effectue par test exhaustif des pixels contenus dans une boîte entourante du parallélogramme. Nous ne reportons donc pas de valeurs concernant cette phase particulière qui reste aujourd'hui encore à l'étude.

```
// 1er sous-triangle
Pour t de a à 0
  Pour s de 0 à s_fin de ligne
    Calculer (x,y) = σ1(s,t) par incrément
    Identifier les pixels recouverts
    Si aucun pixel n'est recouvert
      Continuer
    Sinon
      Calculer (u,v) = τ1(s,t) par incrément
      Pour chaque pixel (x',y') identifié // 2 maximum
        Calculer les valeurs α et β
        Calculer le vecteur déplacement dans la texture
        Filtrer à la position (u',v') déterminée
        Mettre à jour la mémoire écran en (x',y')
  Mettre à jour s_fin de ligne
  Mettre à jour les coordonnées (x,y) et (u,v) de début de ligne
// 2e sous-triangle
Mettre à jour les transformations et la pente utilisée
Recommencer pour t de 0 à −b
```

FIG. 8.15 – Pseudo-code du parcours de triangle intermédiaire.

8.2.2.3 Généralité de la méthode proposée

L'utilisation d'une grille de parcours dans un espace intermédiaire s'avère être une solution réapplicable dans d'autres cas. Ainsi, une telle grille est mise en place dans un parcours avant dans le cas de l'application de plusieurs textures sur même triangle

	Add.	Mult.	Div.
Calcul de σ	2	0	0
Identification	-	-	0
Calcul de τ	2	0	0
Calcul de (α, β)	4	4	0
Déplacement	4	4	0
Total Parcours	12	8	0

TAB. 8.5 – Calculs pour le parcours du triangle intermédiaire. Nous n'avons indiqué que les traitements de la boucle interne sur un point intermédiaire.

[128] et, toujours pour un parcours avant, avec l'utilisation de fonctions programmables comme les *pixel shaders* [129].

Enfin, nous pouvons noter que la technique présentée est extensible aux transformations en perspective. Il est possible depuis l'espace intermédiaire (s, t) de parcourir les espaces (x, y) et (U, V) tout en interpolant la coordonnée homogène W qui permettra par division le calcul des coordonnées texture (u, v). L'ajout de complexité est ici identique lors du parcours des pixels (calcul de W et divisions) comparativement à un parcours traditionnel avec transformation en perspective. L'initialisation est simplifiée compte tenu des propriétés du triangle intermédiaire. Néanmoins, nous n'avons pas testé ce cas précis.

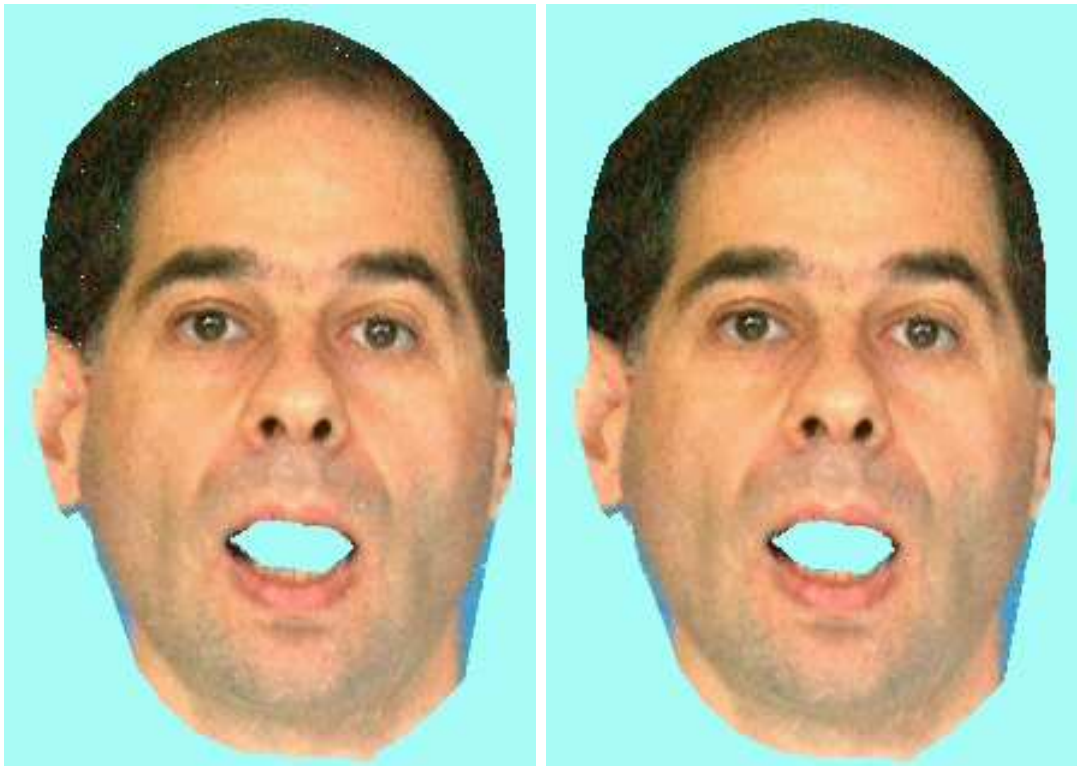
8.3 Questions autour d'une implémentation matérielle

La question de la puissance de calcul pour une possible implémentation matérielle émerge pour deux raisons principales. La première provient du nombre de pixels intermédiaires parcourus. En effet, le triangle intermédiaire parcouru est plus grand que le triangle écran. Toutefois, il est possible, grâce au calcul de la base de la hauteur mis en place, de majorer le rapport entre l'aire du triangle intermédiaire et l'aire du triangle original. Le choix de la puissance de deux supérieure et l'erreur commise par l'approximation des distances limitent le majorant à 5. Cependant, dans un tel cas, tous les pixels intermédiaires ne recouvriront pas un pixel de l'écran et la phase de détection se chargera d'accélérer leur traitement.

La seconde question provient de la réintroduction de divisions dans la phase d'initialisation, en dépit de leurs proscriptions souhaitées. Néanmoins, les résultats de ces divisions interviennent dans le raffinement local et relatif du parcours du triangle intermédiaire. Les déplacements étant effectués avec une précision sous-pixel réduite, il est possible de calculer les valeurs de (α, β) avec une précision relativement réduite. Par conséquent, il est possible en remontant la chaîne de production de limiter la précision de chaque terme du calcul. Il a été procédé à une étude des conséquences de la précision

en terme des erreurs de parcours (différence de pixels écrans traités entre ce parcours et un parcours traditionnel, pixels non détectés) et en terme de différences des coordonnées textures générées. Cette étude permet d'envisager la tabulation des valeurs possibles de la division en raison des faibles dynamiques de données générées et nécessaires à leur génération. La complexité des multiplications subséquentes s'en retrouve également réduite.

La figure 8.16 présente une comparaison entre les résultats d'un parcours traditionnel et la nouvelle organisation du parcours de triangle. Les pixels non recouverts sont dus principalement aux difficultés de mise en œuvre d'un parcours avant, difficultés que notre implémentation rapide n'a pas cherché à résoudre.



(a) Parcours proposé (b) Parcours arrière traditionnel

FIG. 8.16 – Comparaison entre les résultats du nouveau parcours présenté et d'un parcours traditionnel.

Enfin, nous proposons, figure 8.17, un diagramme en bloc d'un opérateur effectuant l'initialisation décrite. Cet opérateur englobe un contrôleur séquençant le processus d'initialisation et divers sous-opérateurs de tri, de calcul itératif, d'estimation de distance et une table pour le calcul des valeurs α et β. La production des coefficients s'effectue au travers d'un registre à barillet *(barrel shifter)*. La figure 8.18 reproduit un diagramme en bloc d'un opérateur de parcours à l'aide des coefficients préalablement initialisés. Un contrôleur régule le déroulement des opérations et s'assure de la bonne mise à jour des registres de parcours. Un sous-opérateur de calcul incrémental fournit les coordonnées (x, y) à un sous-opérateur déterminant les pixels en coordonnées entières (x', y') recouverts lors du parcours. Ces deux paires de coordonnées sont alors transmises afin de

déterminer les valeurs α et β permettant le second niveau de raffinement par le calcul du déplacement relatif à partir des coordonnées (u, v) déterminées par le parcours incrémental. Les paires de coordonnées (x', y') et (u', v') sont alors délivrées pour la suite des opérations de filtrage notamment.

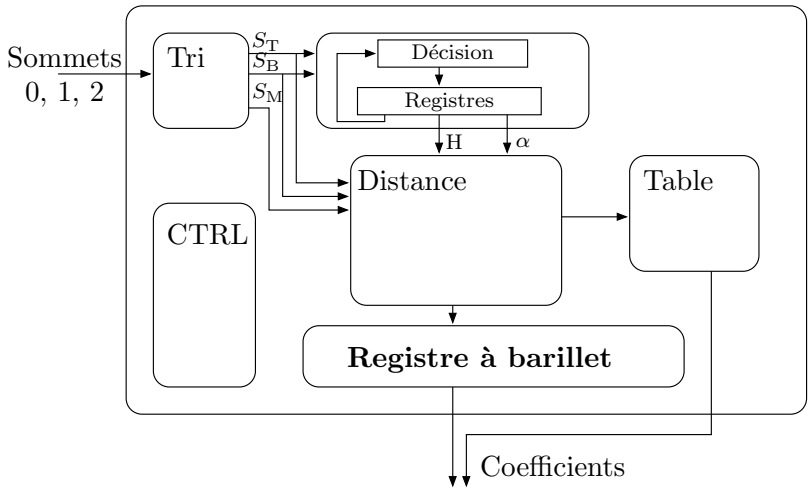


FIG. 8.17 – Diagramme en bloc d'un opérateur d'initialisation.

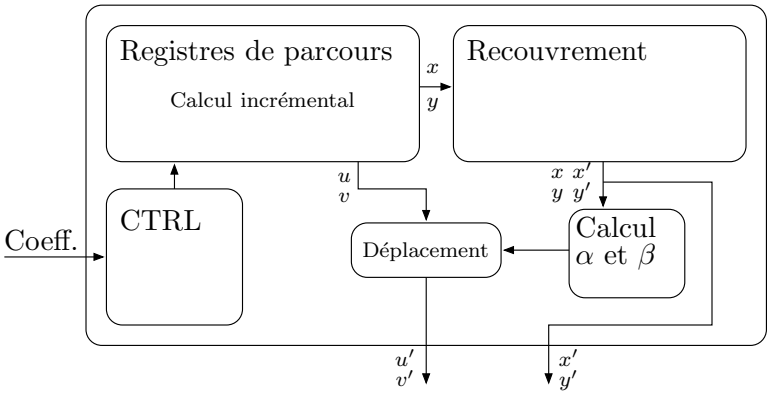


FIG. 8.18 – Diagramme en bloc d'un opérateur de parcours.

8.4 Bilan

Nous venons de voir un nouvel algorithme de parcours de triangle fondé sur un parcours dans un espace choisi, éliminant ainsi les calculs arithmétiques complexes lors de la phase d'initialisation. Les rares divisions que le parcours nécessite encore peuvent être tabulées n'ajoutant pas de calculs supplémentaires. Quant au parcours, si deux transformations affines doivent être calculées, les calculs restent incrémentaux limitant l'augmentation de la puissance de calcul requise. La mise en place d'un parcours à deux niveaux de raffinement permet d'envisager différents modes de fonctionnement en fonction du contexte de performances désirées. Enfin, la généralisation du parcours d'une grille intermédiaire aux cas de *forward texture mapping* associés à des filtres de qualité vidéo autorise une qualité visuelle des images de synthèse compatible avec nos contraintes.

Bien que l'étude de complexité de ce parcours n'ait pas été menée à terme en raison de contraintes de temps, l'introduction de cette technique nous a paru suffisamment prometteuse pour justifier le dépôt d'un brevet [130].

Cinquième partie

Réalisation d'un opérateur de rendu MPEG-4

Chapitre 9

Contexte : le projet TEMPOVALSE

Cette partie V présente la réalisation d'un opérateur de rendu MPEG-4. Elle constitue l'application des principes précédemment exposés. Cependant, certaines simplifications ont été menées que nous justifierons au fur et à mesure que nous y serons confrontés.

Tout d'abord, nous débutons cette nouvelle partie par la présentation du projet dénommé TEMPOVALSE [131] (Terminal Expérimental MPEG4 POrtable de Visiophonie et Animation Labiale ScalablE). Ce projet du Réseau National de Recherche en Télécommunication (RNRT) a en effet servi de support à cette réalisation.

Le but de ce projet a été d'étudier – en fonction des récentes avancées dans les domaines de l'image, de la parole et du multimédia en général – la faisabilité technique et économique d'un terminal multimédia portable grand public basé sur MPEG-4.

9.1 Le rendu de visage parlant

L'application prétexte du projet s'articule autour d'une communication interactive multimodale comprenant la voix, l'animation à distance d'avatars (visages parlants) et la vidéo. Cet ensemble est appelé visio-labiophonie.

9.1.1 Justification

Au delà du simple stimulus auditif, la perception de la parole s'inscrit plus généralement dans un contexte bimodale : l'interaction audio-visuel y est tout à fait importante [132]. L'effet McGurk démontre cette interaction par une illusion auditive lorsqu'un conflit apparaît entre les stimuli auditif et visuel : l'audition du son [ba] synchronisé avec l'image d'un locuteur prononçant la syllabe [ga] produit l'illusion du son [da]. Le tableau 9.1 présente différents exemples de ces illusions auditives.

Ainsi, une fausse information visuelle peut perturber l'interprétation des sons. Inversement, une information visuelle pertinente permet de compléter un stimulus auditif défaillant. Nombre de malentendants ont besoin de regarder en face leur interlocuteur afin

Audio + Visuel → Perçu		
ba	ga	da
pa	ga	ta
ma	ga	na

TAB. 9.1 – Exemples d'effet McGurk.

de faciliter leur compréhension. La lecture sur les lèvres ne permet-elle pas à John Steed de découvrir où est retenue prisonnière Emma Peel dans l'un des épisodes de *Chapeau melon et bottes de cuir* ? Cependant, tous les auditeurs profitent d'une augmentation du seuil d'intelligibilité de la parole grâce à la vision [133, 134, 135]. Par suite, disposer de l'image de son interlocuteur sur son mobile est justifié tant par l'usage en environnement bruité (hall de gare, rue) qui en est fait que par le besoin maintenir un niveau sonore de conversation acceptable.

Les réseaux de futures générations prédisant des débits de plus en plus important, des communications de type visiophonie pourront voir le jour. Cependant, les conditions de transmissions et de débits ne pouvant toujours être optimales, il convient de prévoir un mode dégradé de transmission à base d'avatars (appelé labiophonie) requérant moins de bande-passante qu'une vidéo. Le paragraphe suivant présente une modélisation fine des mouvements des lèvres donnant lieu à une animation vidéo-réaliste d'avatars.

9.1.2 Modèle articulatoire physique

Les travaux menés à l'Institut de la Communication Parlée (ICP) par une équipe partenaire du projet portent sur la modélisation physique de la parole, plus particulièrement des mouvements des lèvres et de la face [135, 136]. Ils démontrent qu'il est possible de modéliser finement les mouvements articulatoires dans un espace vectoriel à six dimensions (ouverture, avancement de la mâchoire, étirement des lèvres, élévation des lèvres supérieures, inférieures et de la gorge). Combinés avec un modèle spécifique 3D du locuteur comme présenté figure 9.1, ces paramètres articulatoires permettent de caractériser une déformation du visage.

Grâce à cette modélisation physique des paramètres articulatoires, le modèle du locuteur permet une reconstruction réaliste de la déformation courante d'un visage du locuteur à partir de positions extrêmes de ce visage. L'extraction des paramètres articulatoires correspondant à une position à un instant t peut se faire par la mise en place d'algorithmes de type analyse-synthèse : analyse visuelle de la zone labiale, extrapolation de nouveaux paramètres, synthèse d'image et détermination d'une mesure d'erreur. La figure 9.2 présente une illustration de cette boucle d'analyse-synthèse.

La reconstruction du visage proprement dite est effectuée par la déformation à partir d'une position dite neutre du modèle du locuteur en fonction des paramètres courants. Le visage est en fait définit comme un maillage de triangles déformable sur lequel une texture vient se plaquer. La figure 9.3 représente ce maillage en position

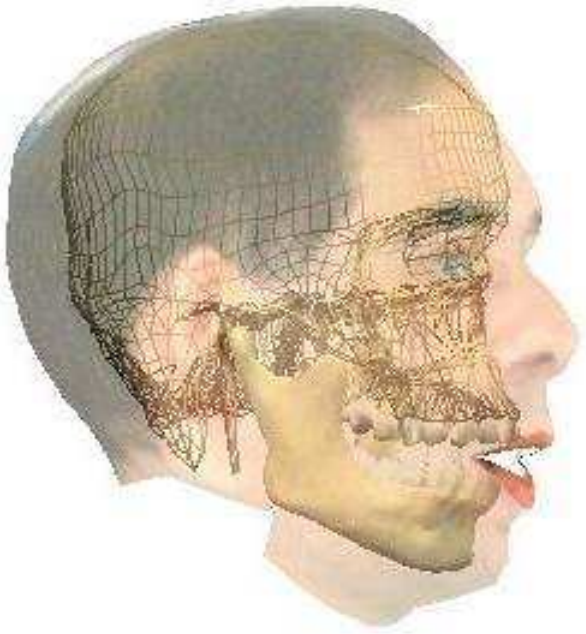


FIG. 9.1 – Modèle facial d'un locuteur © ICP. Modèle en 3D incluant mâchoires, face, langue et texture.

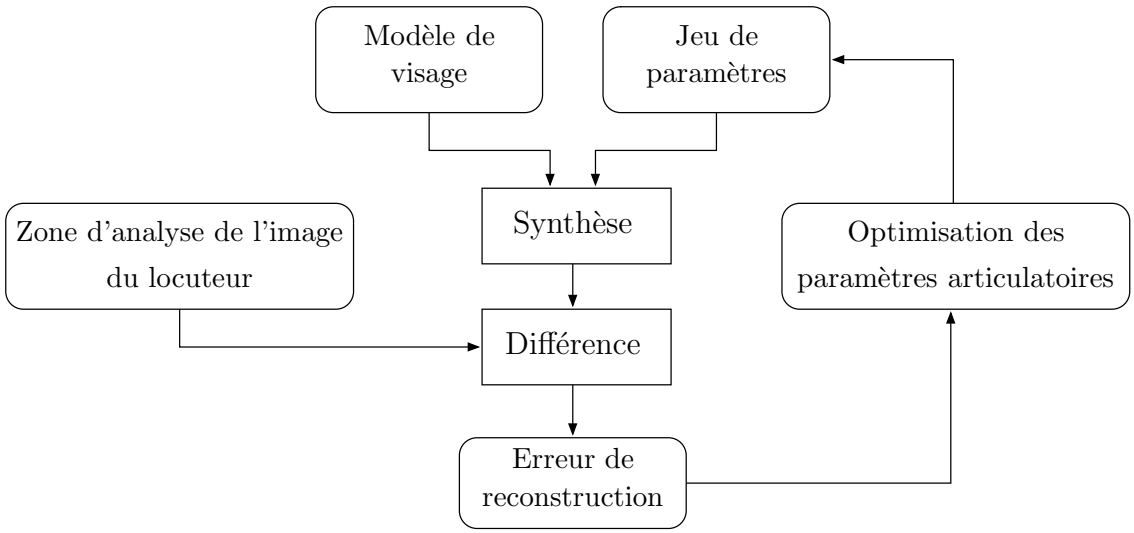


FIG. 9.2 – Extraction des paramètres articulatoires par analyse-synthèse.

neutre. La texture plaquée est elle-aussi une interpolation linéaire des positions extrêmes du visage dans l'espace à 6 dimensions.

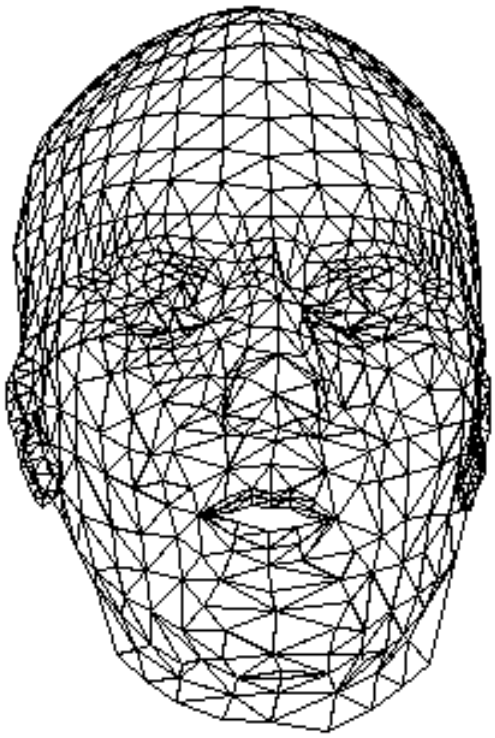


FIG. 9.3 – Maillage déformable de triangles fourni par un modèle de locuteur. Cas de la scène « Badin » évoquée au chapitre 7.

9.1.3 Paramètres d'animation faciale : FAP

Les recherches menées autour de la modélisation et de l'animation d'avatars ont conduit le comité MPEG à inclure parmi les outils SNHC (cf. paragraphe 3.2.1) des outils d'animation de visages et de corps (FBA *Face and Body Animation*) [137, 138]. En particulier, MPEG-4 définit un cadre de travail adapté pour l'animation de visage. Cette norme spécifie sur un modèle abstrait de visage (complété par un modèle de langues, dents, yeux, etc.) un ensemble de points de référence ainsi que les règles de déformation du visage en fonction de paramètres d'animation faciale (FAP *Facial Animation Parameters*). De manière métonymique, le terme FAP désigne aussi régulièrement le cadre complet de l'animation faciale MPEG-4. La figure 9.4 illustre les points de référence concernant le modèle de visage. L'usage des FAP dans le cadre d'une communication mobile est des plus pertinents : en effet, les règles de déformations étant connues *a priori*, seuls ces paramètres nécessitent une transmission dont la taille, une fois comprimés, n'excède pas une valeur typique de 80 bits par unité de données (soit moins de 2 kbit/s pour une animation temps réel).

Afin de rendre le modèle articulatoire physique – modèle dont les mouvements répondent à une sémantique particulière – conforme à la norme MPEG-4, une transformation inversible est appliquée aux paramètres articulatoires pour obtenir des FAP. En

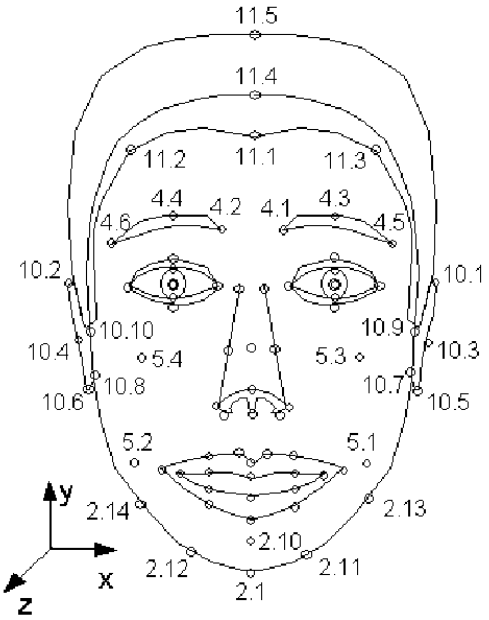


FIG. 9.4 – Exemple de points de référence MPEG-4 pour l'animation faciale.

définitive, ceux-ci sont alors employés comme couche de transport des informations du modèle physique : après acquisition des paramètres articulatoires de la position courante du visage et encodage à l'aide des FAP, ces derniers sont alors transmis de manière normative dans un flux MPEG-4, décodés puis interprétés pour retrouver la sémantique du modèle physique développé et permettre la reconstruction du visage. La figure 9.5 met en lumière le processus complet d'une communication en mode labiophonique.

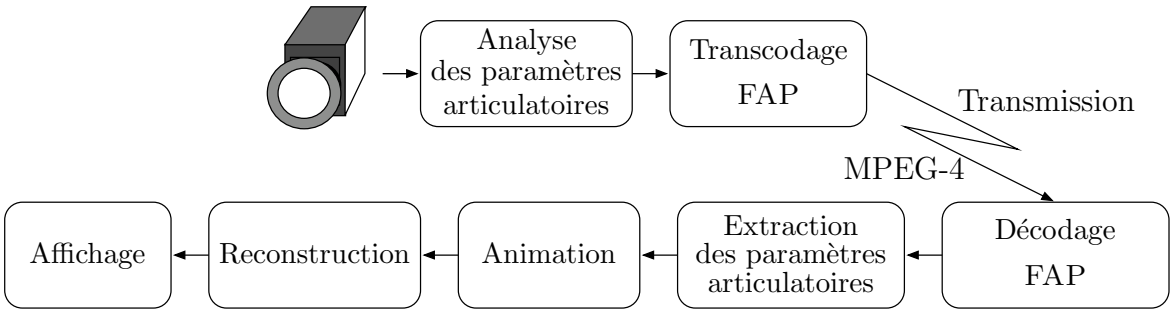


FIG. 9.5 – Vue complète d'une communication en mode labiophonie.

9.2 Composition d'images

Une fois décodé, un flux MPEG-4 subit une opération de composition comme nous l'avons mentionné dans le chapitre 3.2. Un compositeur d'images, comme celui développé

par la société Philips [51], effectue cette opération. Nous présentons ici les caractéristiques de ce compositeur qui fut le point de départ de notre développement que nous exposerons dans le chapitre suivant.

9.2.1 Caractéristiques fonctionnelles

Ce compositeur 2,5D se présente sous la forme d'un coprocesseur capable de générer une image à partir d'une description de scène et d'objets décodés. Avant d'être composés, les objets peuvent être soumis à des transformations affines (rotations et zooms par exemple). Deux principales fonctionnalités sont mises en œuvre par ce coprocesseur.

Composition La composition d'objet vidéo s'accomplit par l'algorithme du peintre. En affectant une valeur de transparence α à chaque objet, la composition s'effectue à partir de l'arrière plan, objet après objet regénérant un nouvel arrière plan pour la composition suivante. Coprocesseur orienté composition vidéo, les objets supportés sont des VOP de forme arbitraire. Ce compositeur supporte nativement différents formats AYUV : A correspond à une composante α de transparence, U et V sont, rappelons-le, des synonymes de Cr et Cb (cf. chapitre 6).

Transformation affine La transformation affine associée à un objet vidéo est appliquée par un parcours arrière : nous parcourons l'espace destination en appliquant la transformation inverse dont les coefficients sont fournis par le pilote du coprocesseur. Un filtre d'interpolation bilinéaire est en charge de déterminer la couleur du fragment d'objet avant composition et réécriture du pixel en mémoire image. Pour cela, les coordonnées sont calculées avec une précision du 8^{e} de pixel soit, pour une représentation entière en virgule fixe des nombres, 3 bits de partie fractionnaire.

9.2.2 Caractéristiques temporelles

Ce compositeur d'images effectue un rendu par tuile. Le macropipeline au niveau tuile que propose ce coprocesseur ne présente en fait qu'une unité de traitement pipelinant elle-même les traitements des pixels. En effet, ne traitant que de grands objets vidéo, les coefficients des transformations affines sont calculés par le processeur hôte ainsi que le secteur de texture nécessaire pour le rendu. La figure 9.6 expose ce macropipeline pour deux objets vidéo (TS_0 et TS_1) recouvrant une même tuile i.

Les traitements pixels sont micropipelinés afin de traiter l'une des quatre composantes AYUV par cycle d'horloge. La figure 9.7 reproduit le diagramme en blocs des différentes pièces matérielles qui composent ce coprocesseur. Les traitements sont répartis entre les sous-blocs de rendu dénommés GT, FIF, IP et BL. Le bloc dénommé GT *(Geometric Transform)* applique la transformation géométrique pour le pixel courant. Les coordonnées de texture adressent alors soit la mémoire de forme BSM *(Binary Shape Memory)*, pour traiter la composante A, soit la mémoire objet OM *(Object Memory)* pour les composantes de luminance et de chrominances. Afin de permettre à la fois les transferts mémoires en prélude des traitements concernant la prochaine surface et les lectures occasionnées par les traitements de la surface courante, ces mémoires sont organisées en

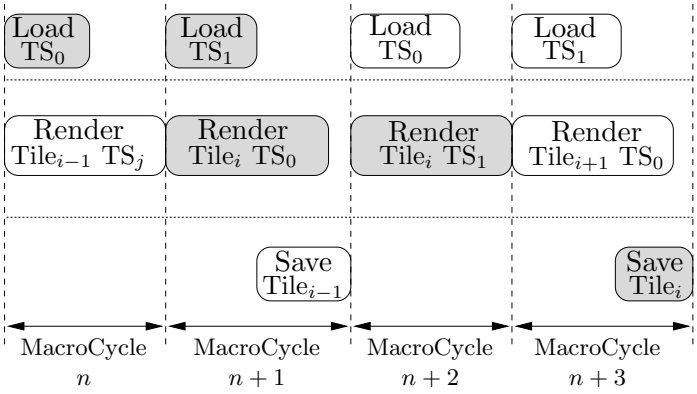


FIG. 9.6 – Macropipeline original du compositeur d'images.

mémoire double dont l'usage s'inverse à chaque macrocycle. Un réordonnancement des données est effectué par le formateur des entrées du filtre FIF *(Filter Input Format)* : les données provenant de la mémoire globale sont en effet formatées par BSF *(Binary Shape Formatter)* (respectivement OF *(Object Formatter)*) pour permettre l'accès aux quatre pixels nécessaires pour le filtre bilinéaire. Le filtrage est effectué dans le bloc IP *(Interpolation unit)* après lequel BL *(Blending unit)* prend en charge le fragment pour effectuer la composition avec le pixel présent localement en mémoire image BLM *(Blending Memory)*. Outre le contrôleur CTRL qui effectue l'analyse de la liste graphique, un DMA *(Direct Memory Access)* se consacre aux accès en lecture et en écriture à la mémoire globale.

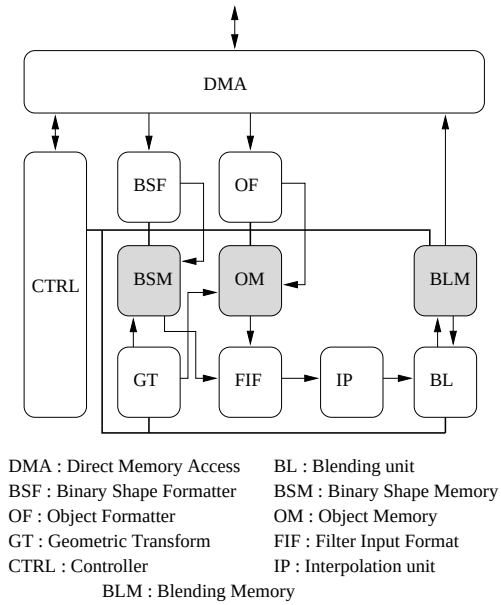


FIG. 9.7 – Diagramme en blocs du compositeur d'objet vidéo. Les mémoires internes apparaissent en grisée.

La taille de la tuile est fixée à un carré de 16×16 pixels parcouru entièrement pour chaque objet. En traitant une composante par cycle d'horloge, ce coprocesseur est capable de traiter 78 125 tuiles par seconde en format AYUV 4:4:2:0 (une composante A par pixel, YUV en 4:2:0) à 50 MHz. Le tableau 9.2 présente un résumé des performances possibles de cette architecture.

format	Nb tuiles/image	Performances
AYUV 4:4:2:0	3 125	**QVGA** (320×240) 10 objets **CIF** (352×288) 7 objets
AYUV 4:4:2:2	2 604	**QVGA** 8 objets **CIF** 6 objets
AYUV 4:4:4:4	1 953	**QVGA** 6 objets **CIF** 4 objets

Tab. 9.2 – Extensibilité des performances de l'architecture du compositeur. Le nombre de tuiles par image est donné pour 25 images par seconde.

9.3 Problématiques

En mode visiophonique, le compositeur d'image précédent est à même de générer une image résultante. Cependant, dans un mode labiophonique, cette architecture ne possède pas les compétences pour rendre le modèle de visage fondé sur un maillage de triangles, modèle que nous avons présenté : il s'agit donc d'ajouter cette fonctionnalité au compositeur existant.

De même, compte tenu des contraintes du projet, il n'est pas envisageable de redévelopper complètement cette architecture et son ordonnancement complet. Tout d'abord, le mode de parcours de la tuile et les filtrages réalisés en mode de parcours arrière nous conduisent à conserver ceux-ci : le traitement des triangles se fera selon les principes évoqués au chapitre 7. Cependant, afin de limiter les modifications des blocs de contrôle, nous circonscrivons au maximum les modifications du pipeline de traitement. Ainsi, nous souhaitons que le parcours des triangles s'inscrive dans le pipeline existant.

Enfin, le modèle physique fondé sur six paramètres articulatoires permet une animation fine du maillage de triangle. La texture dont nous allons habiller ce maillage géométrie se chargera de rendre le modèle ressemblant à un niveau de détail suffisant. En pratique, plusieurs textures seront utilisées pour couvrir l'apparition de détails de parole qui ne pourraient pas être capturées assez finement par une géométrie de taille raisonnable : le pli qui se matérialise à la frontière intérieure de la joue en est un bon exemple (sillon naso-génien). Pour des raisons d'efficacité, il n'est pas possible d'utiliser un trop grand nombre de textures : celles-ci exigeraient la composition de nombreux objets approchant rapidement les limites théoriques du compositeurs telles que nous les avons indiquées. En pratique, trois à cinq textures semblent être un maximum raisonnable : nous devons veiller à effectuer une parfaite adéquation algorithme-architecture afin de limiter les puissances de calcul mises en œuvre pour l'exécution de l'application dans un souci lié aux contraintes énergétiques d'un terminal mobile.

Chapitre 10

Partitionnement logiciel-matériel

Nous présentons maintenant la réalisation d'un prototype d'un opérateur matériel pour la composition d'images vidéo et synthétique dans le cadre de l'application de visio-labiophonie présentée. Nous commençons par une discussion sur le déroulement complet de l'application. Ensuite, nous explicitons l'ordonnancement mis en place afin de rester compatible avec le macropipeline existant. Enfin, nous donnons les résultats d'implantation obtenus.

10.1 Partitionnement et interface

L'exécution de l'application complète est répartie entre le processeur hôte et le coprocesseur. Comme nous l'avons signalé, il convient tout d'abord de réaliser la meilleure adéquation algorithme/architecture afin que les puissances de calcul soient conformes aux contraintes d'un terminal portable. Ensuite, nous précisons les interfaces de communications choisies.

10.1.1 Adéquation algorithme/architecture

Les résultats présentés dans ce paragraphe correspondent aux premières études effectuées dans le cadre du projet TEMPOVALSE. Nous n'avons cependant pas participé à ces premières études.

Le modèle physique des mouvements articulatoires définit un espace à six dimensions dans lequel la déformation courante n'est qu'un simple vecteur où deux phénomènes se superposent. En premier lieu, les paramètres articulatoires déterminent la géométrie complète du maillage de triangles dans la position courante. En second lieu, la texture appliquée sur le maillage déformé résulte de la composition de plusieurs images du visage. Ces images du visage disponibles en position extrême sont alors déformées par transformation affine entre la position extrême et la position courante. Les coefficients maîtrisant cette composition sont déduits des paramètres articulatoires.

Une étude complémentaire a été menée afin de déterminer le nombre optimal de

texture à recombiner et montre, à partir d'un ensemble de 34 postures d'apprentissage, que la recombinaison de seulement trois textures correspondant aux visèmes [a], [afa] et [upu] offre des résultats acceptables [135].

La génération de l'image finale s'effectue par placage de texture sur la position courante du maillage de triangle. Cette position est elle-même déduite de la position neutre du maillage qui correspond au vecteur nul de l'espace des paramètres articulatoires. Cependant, il est souhaitable de minimiser le nombre de traitements de triangles. Ainsi, accomplir la composition de la position courante à partir de chaque texture en position extrême, déformée par transformation affine, n'est pas des plus subtils. L'équation (10.1) exprime cette composition avec α, β et γ les coefficients de pondération de chaque texture, σ_1, σ_2 et σ_3 les transformation affines entre les positions extrêmes et la position courante et T_1, T_2 et T_3 les textures en position extrêmes.

$$\mathrm{Image}_{\mathrm{finale}} = \alpha\sigma_1(\mathrm{T}_1) + \beta\sigma_2(\mathrm{T}_2) + \gamma\sigma_3(\mathrm{T}_3) \tag{10.1}$$

Or, ces mêmes positions extrêmes correspondent à un jeu particulier de paramètres articulatoires permettant de reconstituer leur maillage à partir de la position neutre. Il est alors possible de considérer une texture extrême T_i comme elle-même résultat d'une déformation θ_i d'une texture en position neutre T_i^n comme exprimé équation (10.2).

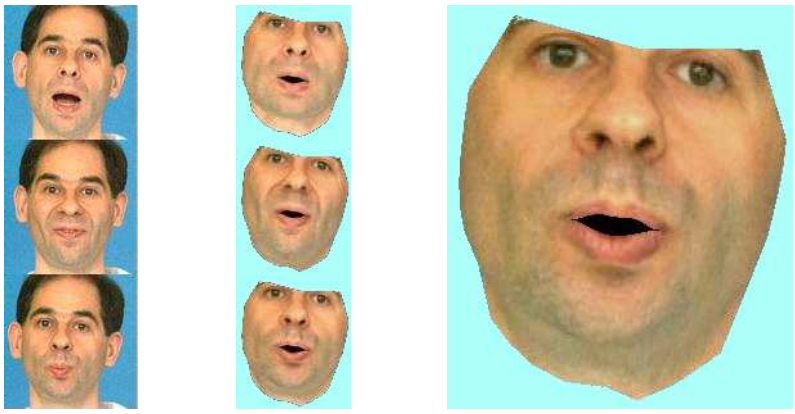
$$\mathrm{T}_i = \theta_i(\mathrm{T}_i^n) \tag{10.2}$$

Nous reportons cette dernière équation dans l'équation précédente (10.1). En remarquant l'égalité des transformations composées $\sigma_i \circ \theta_i$, équivalentes à une unique transformation ϑ entre la position neutre et la position courante, nous nous apercevons que la composition peut s'effectuer au niveau des textures en positions neutres (équation (10.3)).

$$\begin{aligned}
\mathrm{Image}_{\mathrm{finale}} &= \alpha\sigma_1(\mathrm{T}_1) + \beta\sigma_2(\mathrm{T}_2) + \gamma\sigma_3(\mathrm{T}_3) \\
&= \alpha\sigma_1\left(\theta_1(\mathrm{T}_1^n)\right) + \beta\sigma_2\left(\theta_2(\mathrm{T}_2^n)\right) + \gamma\sigma_3\left(\theta_3(\mathrm{T}_3^n)\right) \\
&= \alpha\vartheta(\mathrm{T}_1^n) + \beta\vartheta(\mathrm{T}_2^n) + \gamma\vartheta(\mathrm{T}_3^n) \\
&= \vartheta(\alpha\mathrm{T}_1^n + \beta\mathrm{T}_2^n + \gamma\mathrm{T}_3^n)
\end{aligned} \tag{10.3}$$

Or, une telle composition des textures en position neutre ne nécessite qu'une simple composition en mode vidéo avec des calculs peu complexes puisque les textures ne subissent qu'une transformation identité à laquelle s'ajoute une transparence. Le modèle physique fournit alors l'ensemble des textures à composer non plus en position extrême mais leur correspondante en position neutre. Ces textures seront alors composées dynamiquement pour chaque image. La figure 10.1 illustre le processus de recombinaison ainsi défini.

Le pseudo-code figure 10.2 décrit alors le déroulement des différentes étapes fondamentales de la partie réception lors d'une communication labiophonique. Les trois premières étapes sont traitées par le processeur hôte. Il se chargera également de l'animation du visage, c.-à-d. de la déformation du maillage de triangles. En raison des possibilités du coprocesseur, les opérations de recombinaisons des textures en position neutre et le placage sur le maillage courant sont dévolues au compositeur d'image.



(a) (b) (c)

FIG. 10.1 – Processus de recombinaison des textures extrêmes. Figure 10.1(a) : textures en positions extrêmes, figure 10.1(b) : textures reproduites en position neutre, figure 10.1(c) : textures composées en positon neutre. ©ICP.

```
// Traitement par l'hôte
Décodage du flux MPEG-4 FAP
Extraction des paramètres articulatoires
Induction des coefficients de mélange des textures
Déformation du maillage de triangles en position courante
// Traitement par le compositeur
Recombinaison des textures
Placage de texture par transformation affine
Affichage de l'image résultante
```

FIG. 10.2 – Opérations fondamentales de la partie réception en mode labiophonique.

10.1.2 Interfaces de communication

Les différentes étapes ayant été assignées soit au processeur hôte, soit au compositeur matériel, nous devons définir les interfaces de communication. Le coprocesseur dispose de son propre format de liste graphique pour le rendu de scène vidéo dont la figure 10.3 reproduit la structure hiérarchique. Cette liste graphique définit une scène constituée de tuiles *(Tile)* contenant différents objets dénommés ici *Tile Surface* correspondant à l'intersection d'un objet visuel et d'une tuile et disposant ainsi d'informations locales à la tuile.

Tout d'abord, l'introduction d'une structure définissant un ensemble de triangles, structure telle que nous l'avons définie paragraphe 7.3.1, s'inscrit naturellement comme un ensemble particulier de *Tile Surface*. La texture associée au maillage de triangles est alors caractérisée par cette structure habituelle à laquelle nous rajoutons un niveau hiérarchique supplémentaire correspondant aux ensembles de triangles que nous avons vus. La figure 10.4 présente la nouvelle structure hiérarchique de la liste graphique après ajout des ensembles de triangles *(Triangle Set)*.

Connaissant la taille désormais fixée des tuiles, et à l'aide de l'étude que nous avons

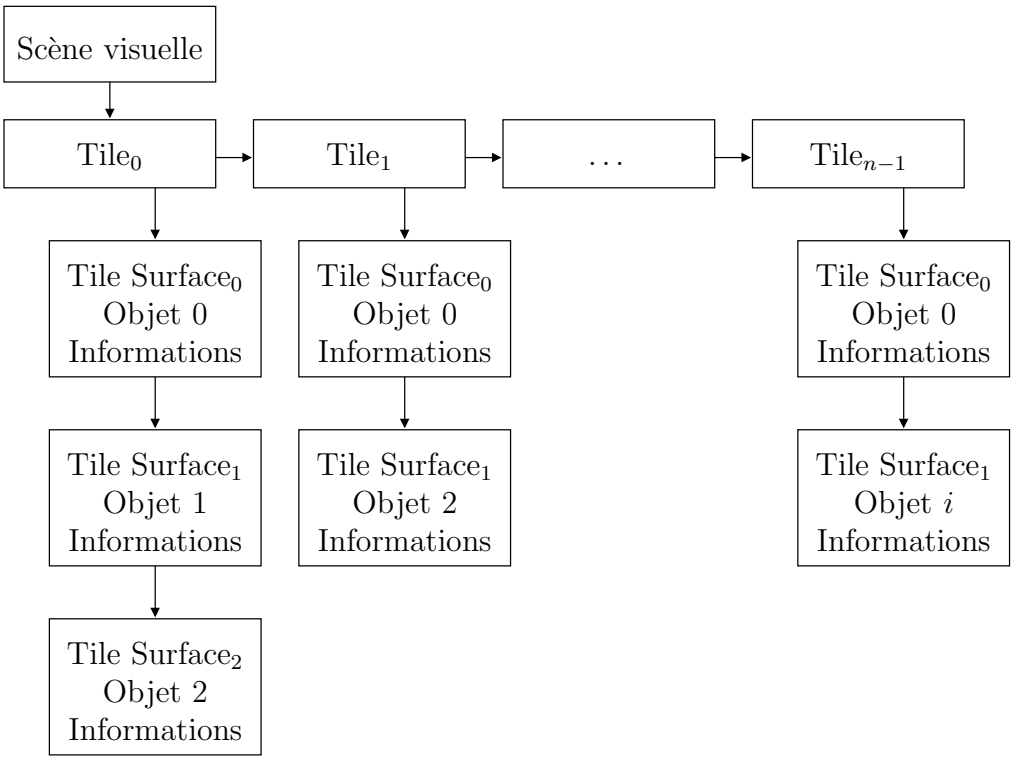


FIG. 10.3 – Vision synthétique de la structure hiérarchique de la liste graphique originale.

menée au chapitre 7, le nombre N maximal de triangles par ensemble peut maintenant être choisi. La valeur retenue vaut 16. Elle présente les propriétés suivantes : elle n'est pas éloignée de la valeur optimale proposée par notre modèle théorique, elle est une puissance de deux et est égale à la taille de la tuile. Corollairement à ce nombre maximal de triangles, nous avons souhaité minimiser la taille d'un ensemble de triangles au mieux : la structure de maillage mutualise les sommets entre les triangles. Ainsi, 3N sommets pour définir N triangles d'un maillage représentent une borne supérieure non optimale. La figure 10.5 exhibe les conséquences de la limitation sur le nombre d'atomes du nombre de sommets Pt effectivement décrits dans cet atome, le nombre de triangles N étant alors fixé à 16. Cette variation est inversement proportionnelle au nombre Pt.

Un triangle est alors défini par trois index de sommets : une telle information peut donc être codée sur $3\lceil\log_2(\mathrm{Pt})\rceil$ bits. Alors que, comme nous l'avons signalé paragraphe 9.2.1, la précision interne des calculs est de 3 bits de partie fractionnaire, les coordonnées du maillage de triangles sont fournies en représentation entière avec une partie fractionnaire sur 4 bits. Les dimensions QVGA (240×320) de la scène finale nécessite une partie entière codée sur 9 bits : une coordonnée est codée sur 13 bits. Par conséquent, le stockage des coordonnées (x, y) et (u, v) des Pt sommets requièrent $4\mathrm{Pt} \times 12$ bits. La bande-passante engendrée par les coordonnées des sommets croît plus rapidement que celle due à la définition des triangles. Un nouveau compromis entre bande-passante et performance (nombre d'atomes surnuméraires) est effectué par le choix $\mathrm{Pt} = 16$.

Des contraintes d'alignement mémoire et de facilité de manipulation de données par le processeur hôte entraînent cependant une taille de structure supérieure au minimum nécessaire. Le tableau 10.1 représente la structure complète d'un ensemble de triangles.

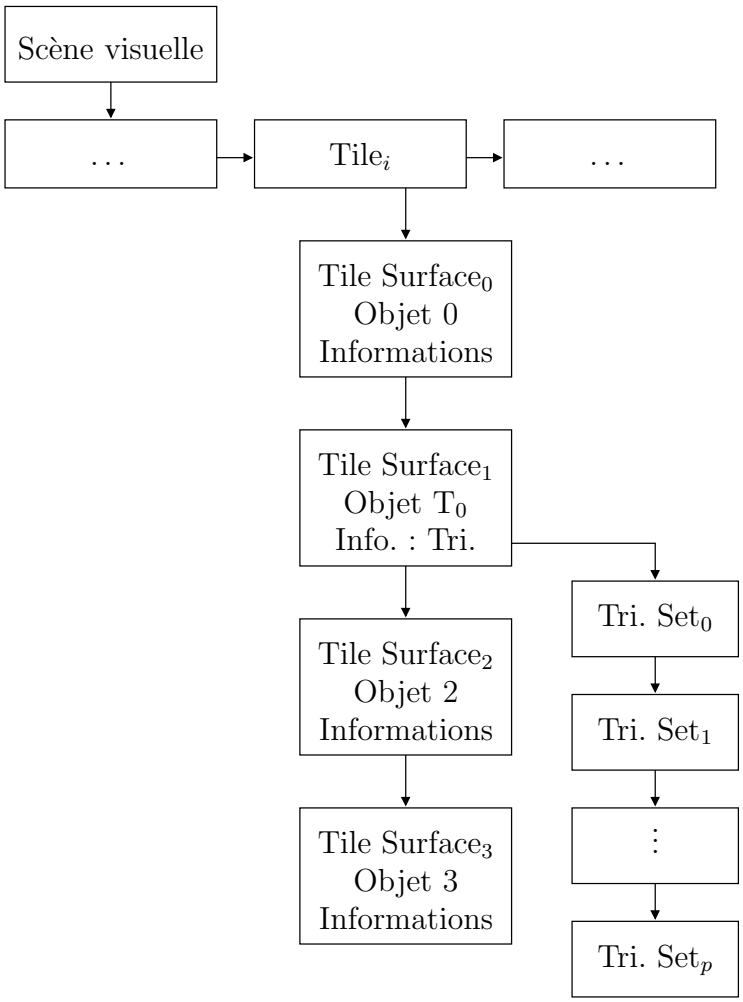


FIG. 10.4 – Vision synthétique de la structure hiérarchique de la liste graphique après ajout des ensembles de triangles.

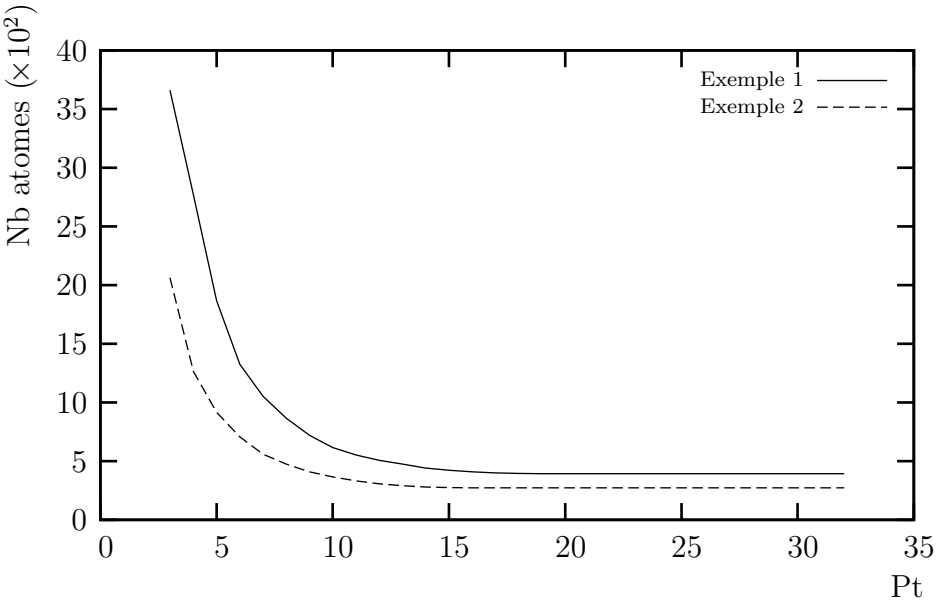
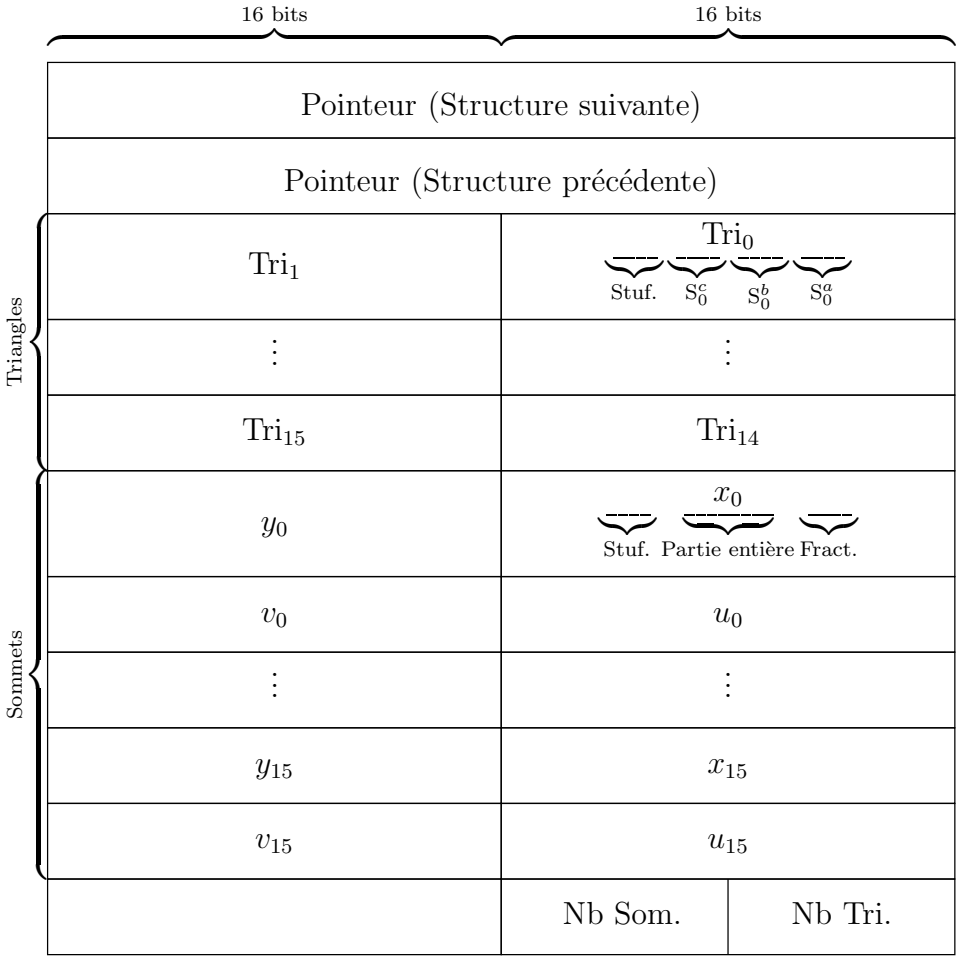


FIG. 10.5 – Évolution du nombre d'atomes en fonction du nombre Pt de points pour définir N = 16 triangles.



Tab. 10.1 – Structure complète d'un ensemble de triangles : 172 octets.

Les structures définies, le processeur hôte est en charge de leur gestion. Cependant, en raison des choix de conception du compositeur, il n'est pas possible de réutiliser directement une surface créée comme source d'une transformation affine : physiquement, il y a séparation des mémoires OM, source des données où s'applique la transformation affine, et BLM, où est créée et conservée la surface résultante. Néanmoins, cette surface peut être interprétée comme scène à part entière. Cette scène peut alors être référencée par une seconde scène. C'est ainsi que la recombinaison des textures constitue une scène comportant trois objets visuels traités en mode vidéo. Réécrite en mémoire, elle est accédée lors du traitement du maillage de triangles constituant une seconde scène distincte.

Cette solution en deux temps n'induit pas de puissance de calcul supplémentaire. En revanche, elle ajoute à la bande-passante mémoire existante celle liée à l'écriture et la relecture de la scène intermédiaire. Pour éviter un tel accroissement, il serait nécessaire de modifier et la structure de la liste graphique et l'architecture du coprocesseur par l'ajout d'une mémoire interne BLM′. Cette nouvelle mémoire, distincte de BLM afin de ne pas interférer avec un arrière-plan déjà existant, devrait permettre de créer et de préserver temporairement la composition requise comme source pour le placage de texture de la tuile courante. La solution en deux temps ainsi proposée et adoptée constitue un

compromis cette fois-ci entre la bande-passante et la taille des mémoires internes du coprocesseur.

10.2 Ordonnancement

Après ces différentes considérations globales, nous nous intéressons désormais au séquencement interne des opérations se déroulant au sein du coprocesseur. En particulier, nous insisterons sur la manière dont nous avons pu conserver au mieux le pipeline original du coprocesseur et comment nous avons pu borner nos modifications majeures aux seuls blocs CTRL et GT désormais dénommé GTM *(GT Mesh)*.

10.2.1 Ordonnancement global de l'opérateur de rendu MPEG-4

L'opérateur de rendu MPEG-4 vidéo enchaîne les calculs au niveau *Tile Surface* à l'aide d'un processus en trois temps *Load-Scan-Write*. Le processus de chargement *Load* correspond aux lectures d'informations de contrôle de la liste graphique et, après analyse, du secteur de texture nécessaire pour rendre la surface correspondante. La détermination de ce secteur est en effet effectuée par le processeur hôte et inscrite dans la liste graphique. Le processus de rendu *Scan* inclut les opérations exécutées dans les blocs GT, FIF, IP et BL. L'écriture *Write* n'intervient après un processus de chargement que si la précédente surface a conclu le traitement de la tuile courante. Un macrocycle peut débuter lorsque sont finis les processus de rendu et les échanges mémoires prévus : lecture et possiblement écriture.

Le processus de chargement constitue donc deux phases distinctes. Il devient alors possible de fondre le processus que nous avions décrit en cinq temps « Init-DST-Charge-Mélange-Écriture » dans celui-ci. En effet, l'initialisation comprend la lecture des données de la liste graphique : en trois temps, cela donne « Init/Charge-Parcours-Écriture ». La figure 10.6 retrace les modifications apportées entre le processus original figure 10.6(a) et le processus modifié figure 10.6(b). Un macrocycle est terminé lorsque la conjonction des fins des processus d'initialisation, de parcours et d'échanges mémoire est réalisée.

Le dimensionnement des puissances de calcul de la phase d'initialisation supposant que celle-ci dispose d'un macrocycle complet, il convient de débuter le plus tôt possible le transfert des données d'un ensemble de triangles. Ce transfert intervient à la fin des chargements préliminaires effectués par le contrôleur CTRL en raison de la position hiérarchique des ensembles. Le transfert s'effectue alors directement par délégation sous la gestion du bloc modifié GTM qui recopie localement les données avant d'initialiser les coefficients de parcours comme nous le verrons dans le paragraphe suivant. Le chargement de la zone de texture intervient donc après ce nouvel accès.

Le compositeur original dispose de trois instructions différentes pour le processus de rendu (outre une instruction générale de non-évolution `NOOP`). La première correspond

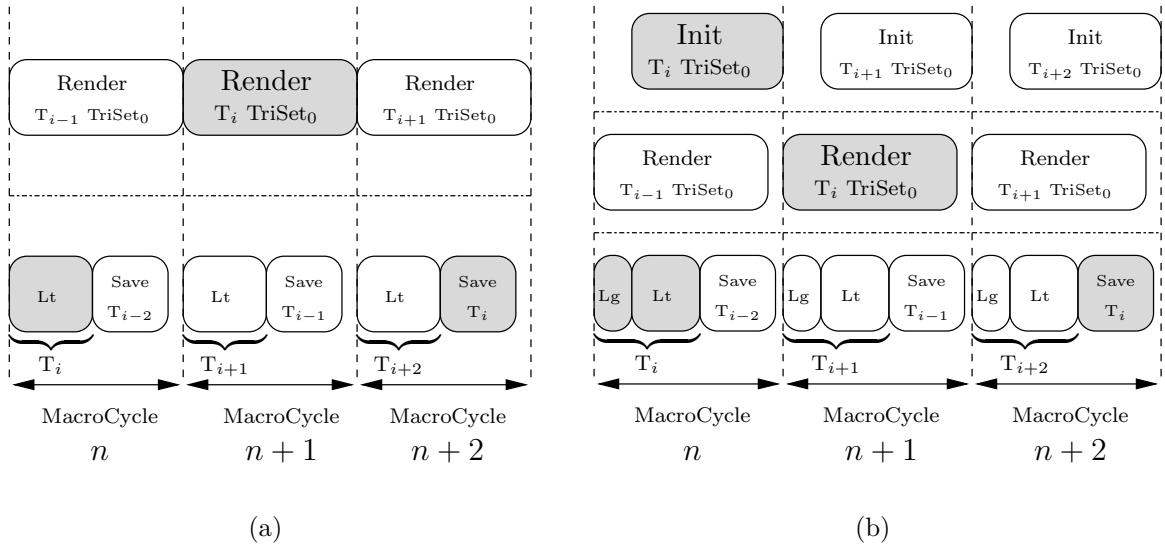


FIG. 10.6 – Comparatif du déroulement des processus de traitements (a) original (cf. figure 9.6) et (b) modifié.

à une tuile non recouverte par une quelconque surface. La deuxième indique qu'une première surface impacte une tuile et la dernière indique qu'une autre surface impacte la tuile. Le premier mode permet d'inhiber les blocs GT, FIF et IP pour n'exploiter que BL afin de remplir avec une couleur de fond la mémoire BLM. Le deuxième indique que BL doit effectuer le mélange non pas avec le contenu (inapproprié) de BLM mais avec la couleur de fond. Le troisième mode indique que BL effectue le mélange avec le contenu de la mémoire BLM. Ces instructions indiquent également aux mémoires OM et BSM fonctionnant en double mémoire d'échanger les zones de lecture et d'écriture.

L'ajout de la fonctionnalité du parcours de maillage de triangles conserve l'ancien mode de parcours en mode vidéo. Nous ajoutons donc de nouvelles instructions afin d'activer, en fonction du type de rendu souhaité, un mode ou l'autre. Cependant dans le processus à trois étapes et compte tenu de la hiérarchie de la liste graphique, l'indication de la zone de texture à charger est commune aux ensembles de triangles d'un même maillage d'une même tuile. Ainsi, le parcours d'un ensemble de triangles ne s'accompagne pas toujours d'un échange des mémoires doubles OM et BSM : trois modes de rendu de maillage interviennent. Le premier indique un parcours de triangles d'un premier ensemble sur une tuile vierge ; il implique, comme la deuxième instruction préexistante, un échange des mémoires OM et BSM et un mélange avec la couleur du fond. Le deuxième mode indique un parcours d'un premier ensemble de triangles sur une tuile non vierge : outre un échange des mémoires, il implique pour BL de mélanger les fragments générés avec BLM correspondant à l'action de la troisième instruction. Enfin, le troisième mode indique un parcours d'un autre ensemble. Nécessairement, un premier ensemble ayant été parcouru, la mémoire BLM contient des informations cohérentes et ce mode implique un mélange avec relecture de BLM par BL. Cependant, ce mode doit correspondre à l'instruction `NOOP` pour les différentes mémoires OM et BLM qui ne doivent alors pas échanger leurs zones mémoires.

Ainsi, nous pouvons masquer par un codage judicieux de l'information de parcours

de triangles les modifications du processus de contrôle aux blocs mémoires. Ces derniers blocs peuvent donc rester inchangés.

10.2.2 Ordonnancement local du parcours géométrique modifié

Le bloc de parcours géométrique est modifié de telle manière à contenir à la fois les éléments matériels réalisant le processus d'initialisation et le processus de parcours géométrique. Le parcours géométrique est constitué de deux parties exclusives : parcours en mode vidéo du bloc GT et le parcours des triangles. La figure 10.7 présente une vue de cette opérateur dont le sous-bloc original GT_OP prend en charge le parcours en mode vidéo.

Les architectures de chaque bloc internes sont précisées en annexe C qui enrichit à l'aide des schémas fonctionnels les questions d'ordonnancement que nous abordons maintenant.

10.2.2.1 Préliminaires

Conjointement à la définition de l'architecture, de l'enchaînement des calculs et de l'ordonnancement du parcours géométrique, nous avons conduit une étude analytique des précisions et des dynamiques des données pour chaque étape de calcul. À partir du cahier des charges définissant la précision finale des calculs, nous avons pu remonter les différentes dépendances de données pour déterminer, à l'aide d'hypothèses vraisemblables dans un contexte de terminal mobile (taille d'écran notamment), les dynamiques et précisions nécessaires et suffisantes des résultats intermédiaires. Nous avons alors pu dimensionner exactement chaque opérateur afin d'assurer une taille minimale pour le coprocesseur.

Nous rapportons les différents calculs que nous avons menés au cours de cette étude analytique en annexe B. Une confirmation, sous la forme de simulations purement logicielles, est venue compléter cette étude.

10.2.2.2 La phase d'initialisation

La phase d'initialisation débute par le chargement des 172 octets (43 mots de 32 bits) de données d'un ensemble de triangles. Ce chargement débute après analyse, par le contrôleur général, du contenu de la liste graphique et plus particulièrement de la structure de *Tile Surface*. La mémoire centrale est accédée par l'intermédiaire du DMA du coprocesseur ; les données sont recopiées localement. Le contrôleur général est averti à l'aide d'un drapeau de la fin de ce chargement dont la latence, pouvant être variable, repousse le début de l'initialisation proprement dite.

Ainsi, l'initialisation des (au plus) 16 triangles ne dispose pas d'un macrocycle complet de pipeline pour s'effectuer. Nous avons estimé à environ une centaine de cycles le délai entre le début d'un macrocycle et la fin du chargement de l'ensemble de triangles

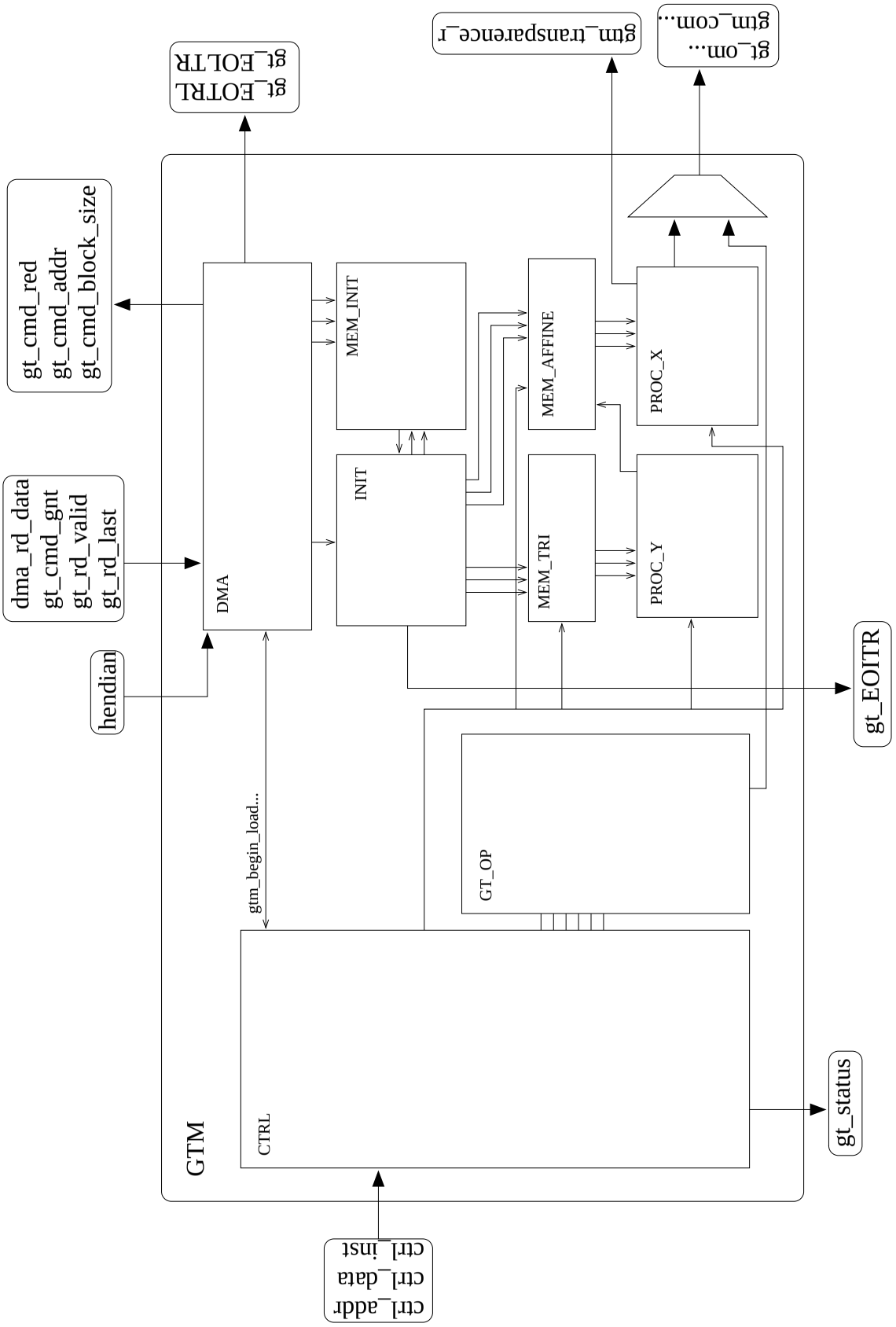


FIG. 10.7 – Vue complète de l'opérateur GT modifié pour le rendu de maillages.

(incluant les accès par le contrôleur aux premières structures hiérarchiques de la liste graphique pour la tuile courante). Dans le cas de scènes au format AYUV $4:4:2:0$, le parcours d'une tuile requiert 640 cycles d'horloge définissant ainsi la durée minimale d'un macrocycle. Nous allouons finalement un budget de 512 cycles pour effectuer la globalité des calculs que nous avons détaillés dans le chapitre 7.2.3.2. Les calculs étant identiques pour chaque triangle, nous sérialisons ceux-ci triangle après triangle : nous les débutons ainsi au rythme d'un nouveau triangle tous les 32 cycles.

L'initialisation d'un triangle débute par l'accès en mémoire locale du triangle courant puis des sommets durant l'accès desquels un tri est effectué, déterminant les sommets Top, Bot et Mid. La similitude des calculs, déduite des équations des coefficients nécessaires pour les processus « Ligne » et « Balayage », nous conduit à la mise en place d'un opérateur générique combinant multiplication et division. Cet opérateur symbolisé figure 10.8 réalise conjointement la division nécessaire pour le calcul des pentes et des coefficients affines et la multiplication-accumulation déterminant les coordonnées du point de référence pour chaque parcours. Cette opération conjointe est légitimée en exploitant une structure itérative de type série/parallèle.

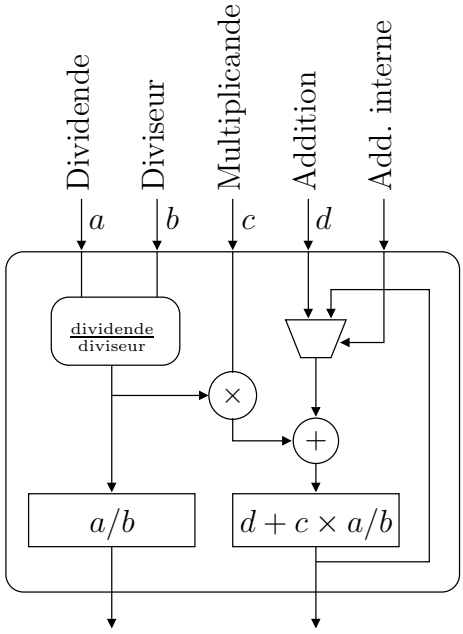


FIG. 10.8 – Opérateur combiné de calcul des coefficients du parcours géométrique. Le signal d'addition interne permet de réutiliser un résultat précédent comme nouvelle valeur d.

La généricité de cet opérateur lui permet de supporter les différences de précisions requises par les deux processus liés au parcours de la tuile. Néanmoins, la complexité de cet opérateur nous astreint à le pipeliner lui-même et impose une latence entre deux requêtes de calculs distincts. Cette latence dépendant de la précision des données nous permet d'instancier un unique opérateur pour les calculs correspondants au processus « Ligne » mais nous conduit à instancier deux opérateurs identiques pour les calculs correspondants au processus « Balayage ». Un rebouclage des calculs spécialise chaque opérateur dans le calcul des coefficients de transformations d'une seule coordonnée (u ou v). Le premier opérateur de division doit donc effectuer trois divisions en 32 cycles ;

quant à chaque opérateur de transformation affine, il doit débuter un nouveau calcul tous les 16 cycles au maximum.

Enfin, en vue de limiter au maximum le nombre de registres nécessaires pour le processus d'initialisation complet, nous avons décidé de recalculer chaque différence dont nous avons besoin lors des divers calculs. Les déclenchements des nouvelles divisions imposent le rythme des calculs et les dépendances de données nous contraignent à prévoir deux soustracteurs ainsi qu'un multiplicateur-accumulateur dédié. Organisés à la manière d'un VLIW, ces opérateurs arithmétiques sont pilotés par un contrôle implicite basé sur un ordonnancement prédéfini des tâches.

Une initialisation d'un triangle est donc initiée tous les 32 cycles et requiert une latence de 77 cycles pour compléter les écritures mémoires des différents coefficients. En définitive, l'initialisation de 16 triangles s'effectue en 557 cycles.

10.2.2.3 La phase de parcours de la tuile

Logiquement découpée par les deux processus « Ligne » et « Balayage », la phase de parcours de la tuile l'est également matériellement : la mémorisation des données entre les deux macrocycles d'initialisation et de parcours est assurée par deux mémoires indépendantes, elles-mêmes connectées à deux blocs matériels. L'un est en charge du processus « Ligne », tandis que le second se charge du processus « Balayage ».

Notre impératif est ici de maintenir la fréquence de génération des coordonnées des pixels de la tuile. Ces coordonnées restent inchangées pendant le traitement complet de chaque composante d'un pixel. Ainsi, la production d'une nouvelle coordonnée est variable en fonction du format de l'image. Nonobstant cette variabilité, le rythme maximal est d'un nouveau pixel tous les deux cycles : nous avons donc dimensionné les différents opérateurs afin d'obtenir les coordonnées en accord avec cette dernière fréquence. Dans le cas où plus de deux cycles sont disponibles, notre contrôle retarde la production des coordonnées aux deux derniers cycles : soit un bloc fonctionne au maximum de ces capacités, soit son fonctionnement est suspendu jusqu'au pénultième cycle précédent la présentation des nouvelles coordonnées. Le processus pixel appelé PROC_X (cf. annexe C, figure C.4) calcule alors séquentiellement les coordonnées u puis v du pixel courant à l'aide des coefficients issus de l'initialisation et fournis par le bloc mémoire interne MEM_AFFINE. Ce bloc mémoire est adressé par la table d'index des triangles, table dont nous avons évoqué le rôle précédemment paragraphe 7.2.3.

Néanmoins, contrairement au mode vidéo où chaque pixel dispose d'un antécédent valide dans la texture et où la transparence est assurée par l'information de forme arbitraire, le mode maillage nécessite une attention particulière. En effet, le sous-maillage impactant une tuile pouvant être découpé en plusieurs ensembles de triangles, certains pixels de la tuiles peuvent n'être recouverts d'aucun triangle. Nous devons identifier de tels cas et donc délivrer une information de transparence aux blocs effectuant le rendu proprement dit.

Le rythme de production des pixels implique que la table d'index soit mise à jour en 16×2 cycles. En nous rappelant que 16 triangles constituent au maximum un en-

semble de triangles, les triangles seront traités individuellement et successivement en deux cycles d'horloge par le bloc de traitement dénommé PROC_Y. Ce bloc effectue donc les calculs d'intersection des trois côtés du triangle avec la ligne courante et la mise à jour simultanée des 16 registres d'index ainsi que de l'information de transparence. Ces registres sont réinitialisés au fur et à mesure qu'ils adressent la mémoire pour le processus pixel. Cet adressage est contrôlé par l'information de transparence. En synchronisant ce traitement avec les traitements pixels, nous pouvons suspendre d'une manière similaire les traitements lorsque le nombre de composantes du pixel courant excède deux. Seul le traitement de la première ligne est décorrélé du nombre de composantes et se réalise au plus rapide afin de limiter la latence complète du parcours.

10.3 Résultats d'implémentation

Une fois l'architecture définie précisément, nous avons mené le flot de conception jusqu'à la synthèse de l'architecture et son placement-routage en vue de permettre son prototypage sur une maquette de démonstration. Nous présentons tout d'abord cette maquette de prototypage et de démonstration avant d'exposer les résultats de synthèse du parcours géométrique et de l'opérateur complet.

10.3.1 Maquette de prototypage

La photographie représentée figure 10.9 exhibe la plateforme de démonstration finale. Elle est constituée d'un PC communicant par le bus PCI avec une carte FPGA de prototypage. Celle-ci contient principalement un premier FPGA Xilinx virtex xcv300 et une SDRAM de 32 Mo. La figure 10.10 [139] présente l'architecture de cette première carte. Ce premier FPGA effectue l'interface entre le bus PCI, la mémoire SDRAM et un bus spécifique DIME d'une largeur de 64 bits sur lequel des cartes filles peuvent se connecter. La figure 10.11 [140] est un telle carte fille contenant deux FPGA Xilinx virtex xcv1000e.

Ne disposant pas de processeur embarqué sur ces cartes, le maquettage complet de l'application de labiophonie a alors été partitionné entre le processeur central du PC qui sert de processeur hôte et les cartes FPGA : le coprocesseur est alors programmé dans le virtex n° 0 de la carte fille. Quant au second circuit programmable, il est utilisé comme générateur programmable du signal d'horloge.

En raison des latences importantes du bus PCI, l'usage d'une mémoire unique est inadapté. Pour minimiser les accès PCI, nous décidons d'effectuer la recopie intégrale de l'espace mémoire nécessaire à l'exécution des calculs dévolus au coprocesseur. Cette espace contient donc la liste graphique complète. Une fois la liste graphique créée par le processeur du PC dans la mémoire centrale, la recopie est opérée en direction des 32 Mo de SDRAM la carte PCI. Du point de vue du coprocesseur, cette SDRAM est considérée comme la mémoire unique du système et les accès s'accomplissent donc par le bus DIME interne aux cartes. La création par le processeur hôte de la liste graphique doit tenir

FIG. 10.9 – Maquette de prototypage.

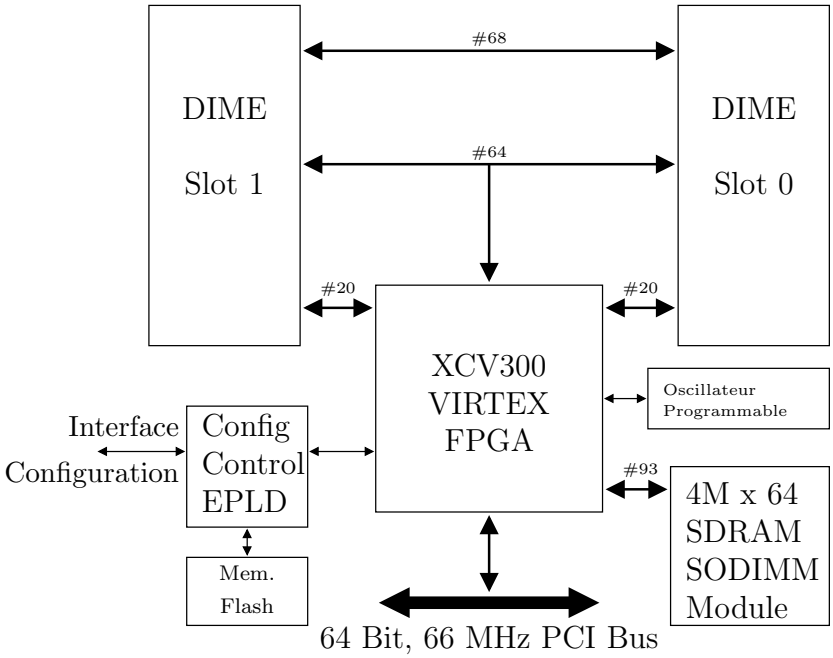


FIG. 10.10 – Diagramme en bloc de la carte PCI.

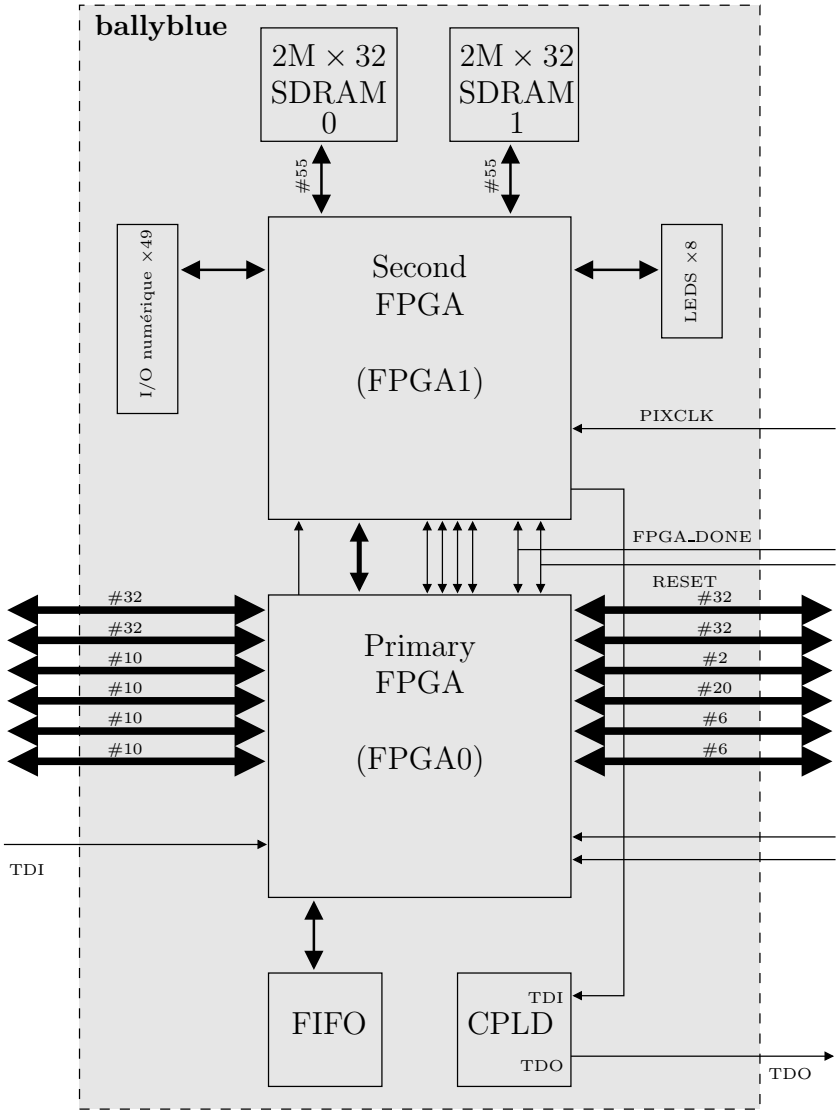


FIG. 10.11 – Diagramme fonctionnel principal d'une carte fille.

compte des contraintes d'organisation en mode paginée des accès à cette mémoire. Cet impératif conduit à une augmentation artificielle de la taille de la structure de données créée. Or, si la recopie complète de la liste graphique permet au processeur de mettre à jour celle-ci pendant que le compositeur effectue le rendu d'une scène, la taille de la liste induit un délai préjudiciable pour les performances du système. Certains informations contenues dans la liste graphique étant statiques, nous introduisons une distinction dans cette liste graphique entre données statiques (textures utilisées) et données dynamiques (liste des ensembles de triangles) : seules les données dynamiques seront alors recopiées à chaque nouvelle scène, les données statiques étant recopiées une unique fois lors du rendu de la première scène par le coprocesseur.

La recombinaison des textures est exécutée en interne par réécriture dans cette mémoire. Quant aux pixels de l'image finale, ils peuvent être réémis vers un contrôleur LCD pour un affichage extérieur ou réécrits en mémoire SDRAM. Il est alors possible de relire cette image par le bus PCI pour la sauvegarder (comme c'est le cas pour les images de la figure 10.13).

L'application de test est constituée de l'affichage d'une séquence d'animation labiale au format QVGA (240×320). Outre le maillage de triangles et les trois textures constituant la base du visage, le modèle est complété par trois images fixes représentant la langue et les mâchoires inférieure et supérieure du locuteur. La scène est habillée avec une image de fond, les logos du projet disposant de forme arbitraire et un habillage du visage émulant ainsi une application réaliste de communication. La figure 10.12 illustre ces différentes composantes ainsi que la scène résultante de la composition de ces éléments.

10.3.2 Synthèse du parcours géométrique

Le bloc matériel de parcours géométrique regroupe tous les calculs de l'étape d'initialisation du macropipeline ainsi que les premières opérations de l'étape de rendu. Ces opérations correspondaient à celles que nous eussions effectuées dans l'étape de DST si nous avions mis en place le macropipeline à cinq étages proposé dans le chapitre 7. L'ensemble des blocs a été décrit en langage VHDL en utilisant au maximum les techniques de généricité. Ainsi la définition d'un contexte différent (taille de tuiles, précisions des positions sous-pixels, etc.) s'appuyant sur l'étude de l'annexe B ne nécessite que la modification d'une dizaine de paramètres primaires dont dépendent tous les chemins de données internes.

10.3.2.1 Le bloc d'initialisation

La complexité de ce bloc provient des opérateurs arithmétiques mis en place, des tailles des données intermédiaires et des registres de stockage des valeurs intermédiaires. Sur un total proche de 1 600 registres, près de 1 400 proviennent des besoins de pipeline des parties opératives. De plus, la logique combinatoire complète nécessite près de 1 800 blocs logiques Xilinx.

Enfin, une mémoire locale, où nous recopions intégralement l'ensemble de triangles

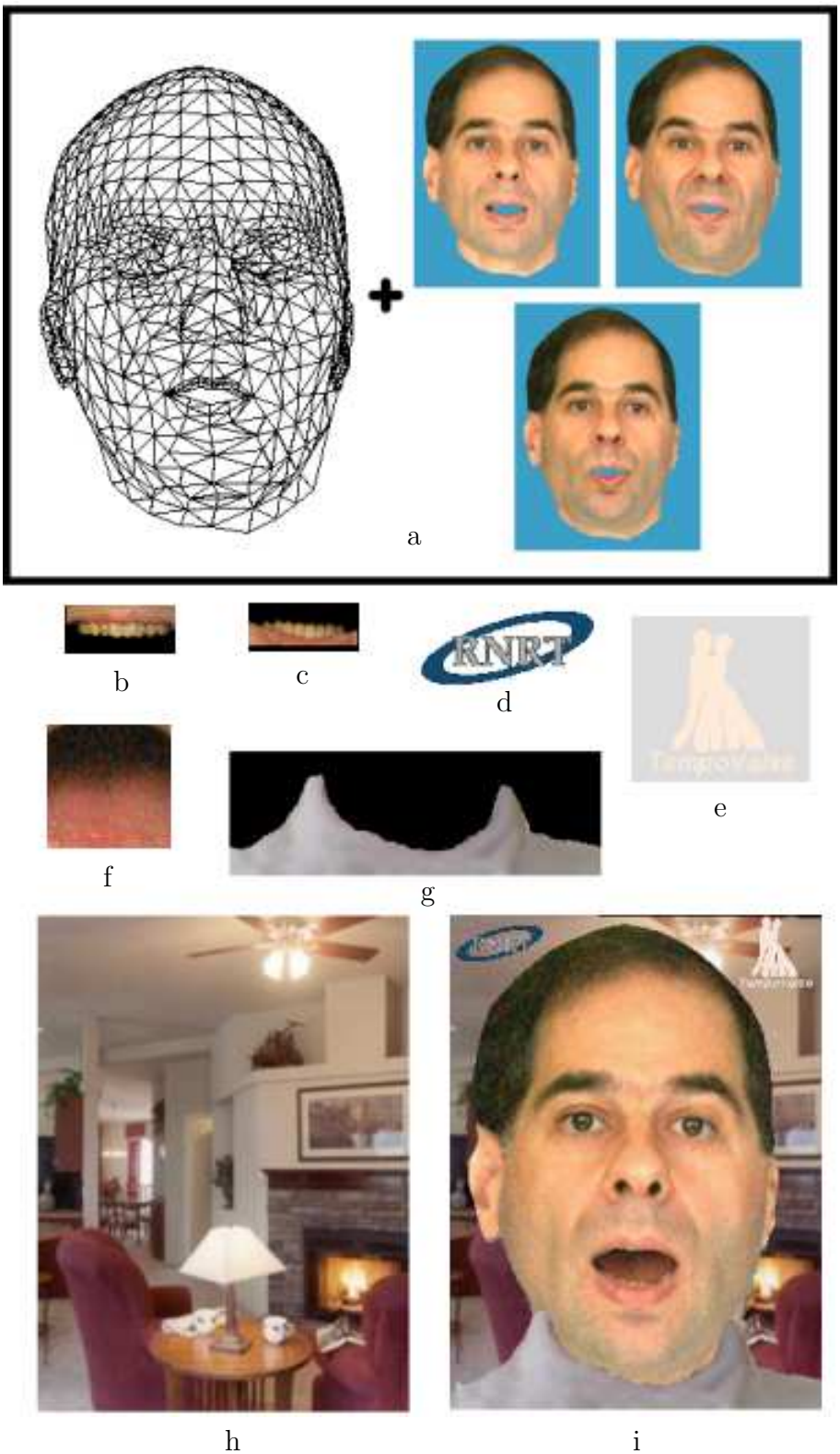


FIG. 10.12 – Détail de l'application test : a) modèle du visage, b) mâchoire supérieure, c) mâchoire inférieure, d) premier logo, e) second logo, f) langue, g) habillage, h) fond d'écran et i) image résultante.

courant grâce à un bloc d'interface avec le DMA du coprocesseur, est accolée au bloc d'initialisation. La taille de celle-ci est de 1,3 Kbit.

10.3.2.2 Les blocs de parcours de la tuile

Regroupant les processus « Ligne » et « Balayage », les ajouts en terme de complexité sont ici réduits. Seule une cinquantaine de registres est nécessaire. En revanche les mémoires tampon entre les étapes du macropipeline requièrent un total de 7,5 Kbit de mémoires embarquées supplémentaires.

Au niveau de la synthèse, nous pouvons remarquer que le processus « Ligne » est entièrement combinatoire avant la mise à jour du registre ligne qui s'effectue une fois tous les deux cycles au maximum. Afin de relâcher les contraintes de synthèse, nous avons pu identifier ici une possibilité de synthèse multicycle de ce bloc particulier.

10.3.2.3 Complexité des blocs

Le tableau 10.2 illustre pour chaque sous-partie du bloc GTM (cf. figure 10.7) le nombre de registres, de blocs logiques et la taille des mémoires.

Les études de complexités « algorithmique » précédemment menées au chapitre 7 concluaient à une possibilité de compromis entre les processus d'initialisation « Triangle » et les processus de parcours « Ligne » et « Balayage ». Les résultats pratiques obtenus montrent clairement que la complexité du processus d'initialisation est très largement sous-estimée. Une possible raison à cela est le caractère complexe du contrôle de ce processus : il nécessite en effet un acheminement conditionnel des données à travers un *cross-bar* de multiplexeurs vers les opérateurs proprement dits. Une deuxième raison est due au pipeline intensif dans les multiplieurs et diviseurs de grandes tailles. Enfin, une troisième raison peut être liée au dimensionnement optimal des différents opérateurs arithmétiques : la dynamique des données traitées diminue en effet rapidement d'un processus à l'autre.

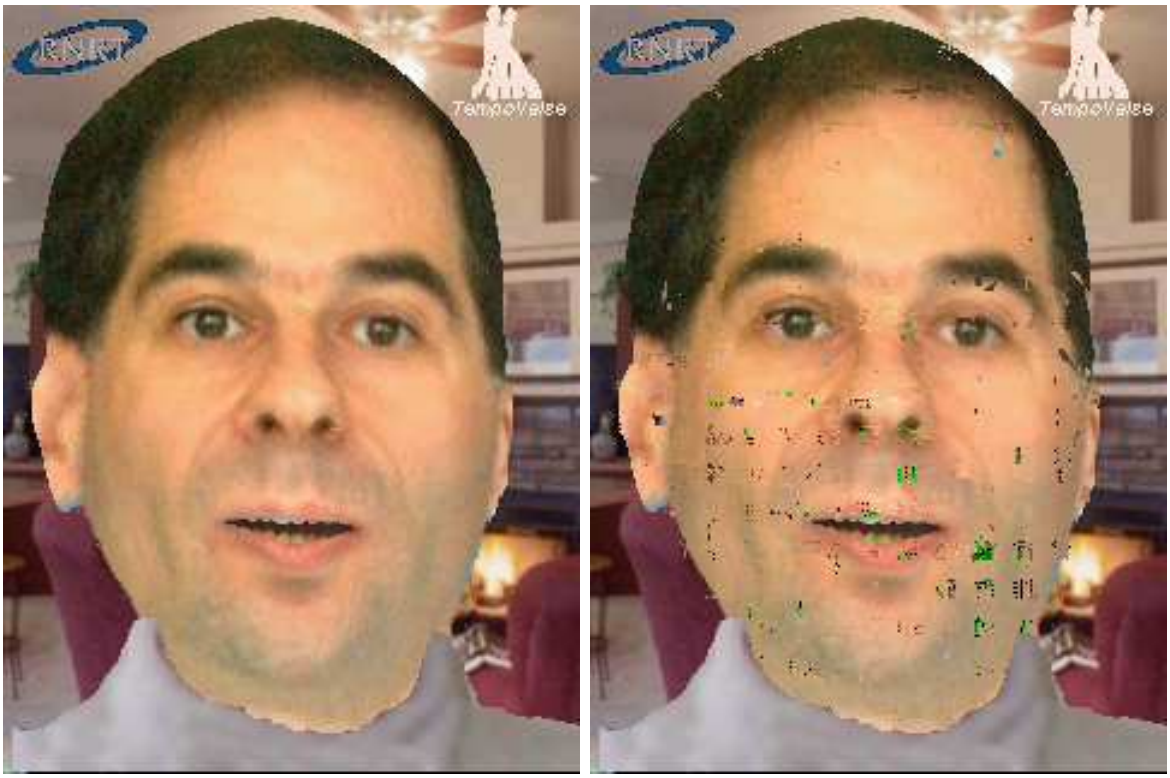
10.3.3 Synthèse du coprocesseur complet

Le coprocesseur a été synthétisé et testé avec succès pour une fréquence de fonctionnement de 50 MHz. Il assure donc les performances minimales décrites dans le tableau 9.2. Nous avons testé la maquette jusqu'à une fréquence de 62 MHz sans erreur visible sur les images générées comme le montre la figure 10.13(a). Au delà, les zones correspondantes au maillage de triangles font apparaître des erreurs (figure 10.13(b)). L'origine de ces erreurs ont été identifiée : il s'agit de chemins critiques dans l'opérateur PROC_Y qui enchaîne une série d'opérateurs arithmétiques sans étage de pipeline. Les objets vidéo, quant à eux, sont traités sans erreurs jusqu'à une fréquence de 75 MHz.

En ce qui concerne une extrapolation en technologie ASIC, il n'est pas de doute que des fréquences de fonctionnement de l'ordre de 200 MHz puissent être atteintes

Sous-partie	**Bloc GT initial**	**Bloc GTM modifié**
CTRL	– 158 blocs logiques – 315 registres	– 238 blocs logiques – 476 registres
GT_OP	– 122 blocs logiques – 173 registres	Non modifié
DMA	Inexistant	– 93 blocs logiques – 121 registres
INIT	Inexistant	– 1 749 blocs logiques – 1 596 registres
PROC_Y	Inexistant	– 281 blocs logiques – 20 registres
PROC_X	Inexistant	– 102 blocs logiques – 28 registres
MEM_INIT	Inexistant	1,3 Kbit
MEM_TRI	Inexistant	4,6 Kbit
MEM_AFF	Inexistant	2,9 Kbit

Tab. 10.2 – Évolution du bloc de transformation géométrique.



(a) Fonctionnement à 62 MHz (b) Fonctionnement à 67 MHz

FIG. 10.13 – Conséquences de l'augmentation de la fréquence interne du coprocesseur.

sans modification notable de l'architecture. La complexité de cet opérateur, exprimée en équivalent portes, est de 90 Kporte auxquelles s'ajoutent 14 KB de mémoire interne. Le compositeur original dispose d'une complexité de 60 Kporte pour 13 KB de mémoire interne. Le support des maillages de triangles apportent donc une complexité supérieure proche de 33 % : en technologie 0,18 μm, nous pouvons donc estimer ce compositeur à une surface proche de 4,5 mm^2 (le compositeur original occupant une surface de 3,5 mm^2). Bien que n'effectuant que de la composition en 2,5D, la complexité de notre compositeur reste très largement inférieur à celles des cœurs de processeurs 3D actuellement disponible (300 Kporte pour le cœur allégé d'ARM : cf. page 70).

Chapitre 11

Conclusion

Nous venons de présenter une réalisation pratique d'un opérateur de composition d'images traitant à la fois des objets MPEG-4 de type vidéo et maillages de triangles. Nous avons pu ainsi démontrer la validité des algorithmes que nous avons développés dans la partie IV. En particulier, le principe de rendu par tuile d'un maillage de triangle est désormais validé tout comme le concept d'ensemble de triangles qui effectue la liaison entre des contraintes « pire cas » et « au mieux ». De plus, les résultats obtenus pour notre plateforme de prototypage font apparaître une échelle de performances intéressante. Ces résultats soulignent l'opportunité du développement de nouvelles applications pour mobiles fondées sur des communications audio-visuelles.

La réalisation de cette architecture ainsi que son implémentation au sein de la plateforme de prototypage nous ont permis toutefois de nous rendre compte de deux aspects sensibles de notre système complet. Tout d'abord, le support des maillages de triangles s'est traduit par une augmentation sensible de la taille de notre coprocesseur. Le coprocesseur fonctionnant au maximum de ces capacités, l'augmentation de sa taille est directement reliée à la puissance de calcul requise. Cette augmentation, majoritairement localisée au sein du bloc réalisant l'initialisation d'un ensemble de triangles, délimite la zone où des efforts d'optimisation doivent être menés. En particulier, la simplification de la phase d'initialisation par l'emploi d'un nouvel espace intermédiaire de parcours des triangles s'avère d'une pertinence accrue à la vue de ces résultats. Ensuite, la conservation du pipeline initial en trois étapes grâce à l'information de zone de texture inscrite dans la liste graphique est apparue délicate dans sa mise en œuvre. En effet, un calcul dynamique étant malaisé pour le processeur hôte, la taille des zones ainsi définies approchait régulièrement les limites matérielles des mémoires internes du coprocesseur. Une fois encore, la mise en place du macropipeline en cinq étapes incluant la Détection de Secteur de Texture s'avère totalement pertinente.

Outre un relâchement des contraintes sur les tailles des mémoires internes et une bande-passante limitée aux données strictement nécessaires pour la tuile courante, cette détection permettra également de simplifier la liste graphique en économisant cette information. Toutefois, sans que nous l'ayons formellement vérifié, les performances de notre plateforme de prototypage sont également limitées en raison des communications *via* le bus PCI et de la recopie d'une partie de l'espace mémoire du processeur hôte sur la SDRAM de la carte contenant notre coprocesseur.

De plus, pour conclure sur les aspects de la labiophonie, nous pouvons ajouter que, au-delà d'un compositeur d'images employé pour effectuer un mélange de textures et un habillage d'un maillage de triangles, notre coprocesseur peut être utilisé dans la boucle d'analyse-synthèse afin d'extraire les paramètres articulatoires d'une image. En effet, en limitant en mode réception au seul mélange des trois textures au format AYUV 4:4:2:0 et le placage du résultat sur le maillage de triangles, 1 200 tuiles sont consommées sur le budget de 3 125 tuiles évoqué tableau 9.2. L'extraction des paramètres articulatoires d'une image requérant environ une cinquantaine d'itérations de l'algorithme d'analyse-synthèse, le parcours d'une quarantaine de tuiles est alors possible pour une itération.

La figure 11.1 illustre la phase d'analyse-synthèse de l'algorithme d'extraction des paramètres articulatoires d'un visage. Effectuée sur une partie restreinte du visage aux abords des lèvres, l'analyse-synthèse réalise une reconstruction partielle du visage qui, comparée à l'image réelle du visage, produit une valeur de corrélation. Cette valeur permet alors d'adapter le jeu de paramètres articulatoires.

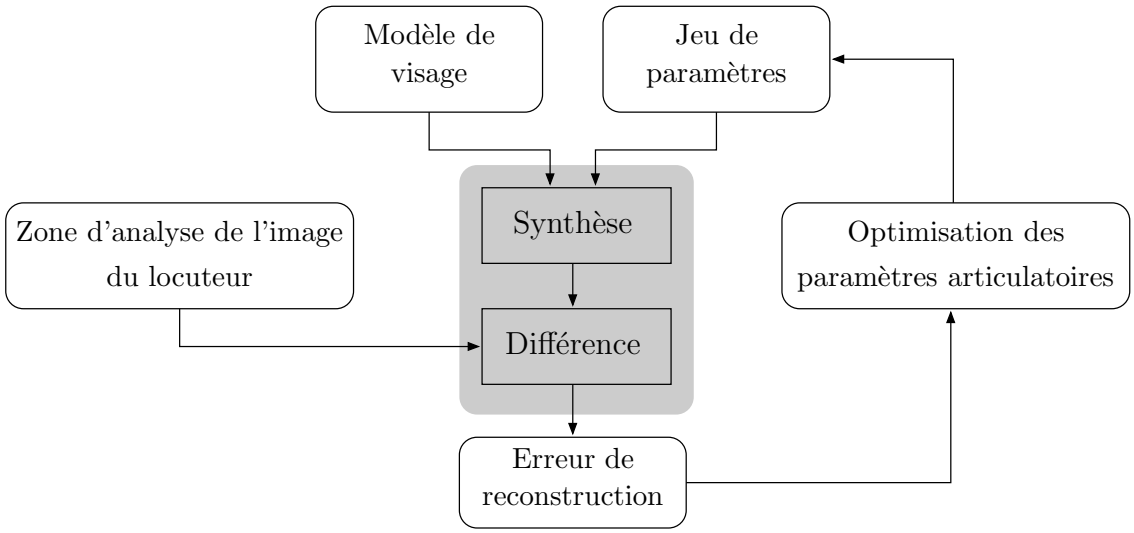


FIG. 11.1 – Accélération de l'extraction des paramètres articulatoires par analyse-synthèse. La zone grisée indique les tâches candidates à une accélération matérielle.

La reconstruction des abords des lèvres peut ainsi être déléguée au coprocesseur. La comparaison avec l'image réelle du visage nécessite cependant une modification du bloc effectuant le mélange de plusieurs objets. Ce bloc doit désormais non plus effectuer un mélange mais une différence tout en maintenant la valeur de corrélation produite. Ces possibilités démontrent ainsi la pluri-potentialité de ce coprocesseur.

Sixième partie

Conclusions

Chapitre 12

Bilan général

Dans un contexte de convergence entre divers domaines suscitée tant par des facteurs économiques (besoin de réduction des coûts) que conjoncturels (développement de nouvelles normes comme MPEG-4), nos études ont porté plus spécifiquement sur les implications algorithmiques et matérielles de la convergence de deux domaines distincts (vidéo et synthèse graphique) sur les fonctions de rendu.

Nous avons exploré tout d'abord une convergence au travers de la représentation des données permettant d'effectuer à la fois du rendu vidéo et de la synthèse d'image explorant ainsi le premier thème notre problématique. Nous avons étudié cette convergence dans l'optique de toujours limiter au mieux les puissances de calcul déployées : cela nous a conduit à définir un nouveau parcours des triangles qui les rééquilibre entre les processus pour réduire globalement la puissance de calcul.

Nous avons montré comment harmoniser les contraintes des deux types de rendu grâce, notamment, au rendu par tuiles. Ce rendu nous permet aussi bien d'accorder les contraintes orientés « pire cas » du rendu vidéo avec celles orientées « au mieux » de la synthèse d'images traditionnelle que d'assurer une extensibilité des performances de l'architecture proposée : le modèle que nous avons dégagé dispose d'un ensemble de paramètres que nous pouvons ajuster en fonction des performances souhaitées. Toujours dans le cadre du rendu par tuile, nous avons considéré une stratégie optimale d'utilisation de la bande-passante entre un coprocesseur et une mémoire unifiée d'un système complet. L'enchaînement temporel des accès mémoires associé à un système de mise à jour de cache déterministe permet en effet de minimiser les échanges avec la mémoire. Cette approche nous a permis de répondre simultanément aux questions 2, 3 et 4 de notre problématique.

Nous avons également rendu compte de méthodes alternatives de rendu de triangles (*forward texture mapping*) faisant bénéficier la synthèse d'images des traitements plus spécifiquement vidéo assurant une qualité visuelle supérieure. Nous pouvons toutefois regretter de ne pas avoir approfondi ces techniques, en relation avec des aspects de traitement du signal. De même, les questions de programmabilités permettant des évolutions ou des adaptions simples de notre architecture n'ont pas pu être abordées véritablement au cours de cette thèse. Les questions 5 et 6 de notre problématique restent encore largement ouvertes.

Enfin, nous avons pu mettre à profit ces développements théoriques qui tentaient

de répondre aux différentes questions de notre problématique au sein d'un projet dont le but était la réalisation d'un compositeur matériel d'images. Afin de garantir au plus juste les puissances de calcul, cette réalisation fut précédée d'une étude analytique des précisions nécessaires à chaque étape de calcul afin de garantir les contraintes de notre cahier des charges. Nous avons mis en place les architectures internes des blocs assurant le rendu des maillages de triangles, nous avons collaboré au développement VHDL de ceux-ci, à leur optimisation ainsi qu'à la synthèse complète du compositeur d'image. Ce coprocesseur peut être estimé à une surface de l'ordre de 4,5 mm^2 en technologie 0,18 μm. Ce compositeur d'image a ensuite été implémenté sur une plateforme de prototypage ayant permis de démontrer les performances de cette architecture.

Chapitre 13

Perspectives

Après avoir exposé et discuté certains aspects théoriques de la convergence entre la vidéo et la synthèse graphique, nous avons pu les confronter aux résultats d'une première réalisation matérielle d'un opérateur unique effectuant le rendu d'objets hybrides. Ainsi, cette expérience nous permet-elle d'envisager un certain nombre de perspectives.

13.1 Rendu par tuiles

Nous avons longuement traité du rendu par tuiles au cours des parties précédentes. Cependant, nous relèverons deux pistes que nous n'avons pas eu l'occasion d'aborder.

13.1.1 Tuiles et modes de rendu alternatifs

Nous avons montré chapitre 7 comment le rendu par tuiles permettait de répondre aux problématiques d'un rendu unifié vidéo-synthèse d'image et de bande-passante de façon aussi optimale que possible. Le chapitre 8 présente, quant à lui, une réponse à la question de la puissance de calcul nécessaire au rendu de triangles dans le cas des transformations affines. Néanmoins, nous n'avons pas eu l'opportunité d'envisager la synthèse de ces différentes réponses.

Afin de limiter au mieux les puissances de calculs, l'association d'un parcours tuile et d'un espace intermédiaire réclame le parcours de points intermédiaires dont l'image à l'écran appartient effectivement à la tuile comme le montre la figure 13.1. La détermination d'un premier pixel intermédiaire dont l'image appartient à la tuile semble primordiale : nous pourrons alors mettre en place, dans l'espace intermédiaire, des algorithmes « gloutons » de remplissage de géométrie 2D en fonction des images des points du triangle intermédiaire dans l'espace écran.

Concernant la question d'un traitement de qualité vidéo, nous avons suggéré un parcours des triangles en mode avant développé par une équipe néerlandaise. Nous avons noté avec intérêt les caractéristiques de ce type de parcours ainsi que sa similitude avec notre parcours par la mise en place d'une grille intermédiaire. Il serait intéressant

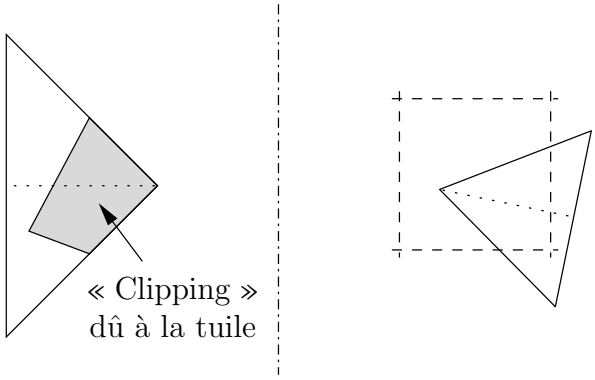


FIG. 13.1 – Association du rendu par tuiles et par triangles intermédiaires. La zone grisée correspond à la seule partie du triangle intermédiaire pertinente pour la tuile considérée.

de poursuivre notre étude afin d'identifier, le cas échéant, de nouvelles synergies et de permettre une réalisation matérielle de ces algorithmes.

13.1.2 Compression de texture

Lors de notre état de l'art, nous avons évoqué des techniques de compression de texture employées par des architectures commerciales afin de réduire encore la bande-passante ou, tout au moins, autoriser toujours plus de texture afin d'assurer un réalisme toujours plus poussé. Nous avons également évoqué les affinités entre le rendu par tuiles et les techniques de compression par blocs communément exploitées en traitement vidéo.

Toujours dans le souci de réduire la bande-passante mémoire requise, il conviendrait de poursuivre des réflexions sur les techniques de compression que nous pourrions exploiter comme la décomposition en ondelettes. Tout d'abord, la décomposition fréquentielle que réalise cette transformation s'apparente à la décomposition en *MIP-map* adoptée en synthèse graphique. Ensuite, la compression des images intermédiaires et une décompression ultérieure seraient amplement justifiées dans le cas d'une application de type labiophonie telle que nous l'avons détaillée.

Enfin, l'augmentation du réalisme d'une image de synthèse conduit à l'emploi de textures naturelles. Celles-ci disposent de techniques de compression adaptées : leur emploi semble d'une grande pertinence tout en ajoutant un nouveau couplage des fonctionnalités entre vidéo et synthèse d'image. Néanmoins, ces techniques recourent de plus en plus à un vaste contexte de décodage. Les hypothèses sous-jacentes concernant l'ordre de décodage n'étant pas vérifiées en synthèse d'image, ces techniques nécessitent toutefois des adaptations.

13.2 Évolutivité du compositeur d'image

Nous avons traité des questions d'extensibilité de notre architecture. Cependant, nous n'avons pas eu l'occasion d'aborder la problématique des aspects programmables, de la souplesse d'évolution et de la pérennité de l'architecture vis-à-vis des développements futurs dont elle pourrait faire l'objet. Nous abordons ici quelques pistes de réflexion au sujet de cette pérennité ainsi que quelques idées de développements possibles.

13.2.1 Contrôle microprogrammé

Le compositeur d'image utilise un contrôle câblé et, par conséquent, rigide. L'ajout d'une nouvelle structure hiérarchique, comme les ensembles de triangles, nous a conduit à modifier profondément ce contrôle. Gérant un nouveau niveau hiérarchique, le compositeur d'image n'en demeure pas moins un circuit intégré spécifique : un ASIC *(Application Specific Integrated Circuit)*.

Nous pourrions envisager de modifier le contrôle de cet opérateur pour le microprogrammer à l'aide d'un jeu d'instruction spécifique du type d'application que nous visons : le compositeur deviendrait alors un ASIP *(Application Specific Instruction set Processor)* [141]. De plus, nous souhaiterions revoir l'agencement de la liste graphique. En effet, la structure hiérarchique de celle-ci induit une communication importante avec la mémoire. En outre, la répétition des données accroît la taille de cette liste alors que des informations concernant la globalité d'une scène seraient susceptibles d'être extraites. Les instructions dont nous pourrions doter notre coprocesseur permettraient alors une meilleure prise en compte de ces données globales.

Enfin, en conservant la structure actuelle de la liste graphique, une simple technique de cache des structures atomiques de type globale permettrait de réduire les échanges mémoires.

13.2.2 Graphisme 3D

Nous nous sommes focalisés au cours de nos travaux vers la synthèse d'images de type 2,5D. Il conviendrait d'étendre nos résultats au graphisme 3D par l'ajout de la coordonnée de profondeur z : de manière similaire aux autres coordonnées, de nouvelles puissances de calcul s'ajoutent lors de l'initialisation afin de faciliter par la suite le calcul effectif de cette coordonnée. Ce calcul s'effectuerait durant la phase DST afin de tirer profit du rendu différé et du rejet anticipé des points non visibles.

Enfin, il serait également souhaitable d'étendre le jeu des transformations que traite le compositeur avec des transformations en perspective. Dans le cas du parcours à l'aide d'un espace intermédiaire, nous avons déjà évoqué la possibilité de l'extension du parcours par l'interpolation des coordonnées (x, y), (U, V) et la coordonnée homogène W, interpolation suivie d'une division pour obtenir les coordonnées de texture (u, v).

13.2.3 Graphisme 2D vectoriel

Enfin, alors que le graphisme 3D peut sembler haut de gamme, il convient de se pencher sur ce qui semble disposer d'un avenir prometteur sur mobile : le graphisme 2D vectoriel. De jeunes pousses émergent de nos jours afin d'offrir aux opérateurs la possibilité pour leurs usagers d'agrémenter leurs écrans de portable à l'aide de graphisme dans le style de SVG [102] ou Flash [103].

Pour cela, il convient d'ajouter des fonctionnalités de tracé de contours (droites d'épaisseurs variables, continues ou non...), de remplissage de secteur (sans placage de texture) et surtout le tracé de courbes de Bézier. Nous pouvons remarquer ici que le format Flash limite les courbes de Bézier aux seules courbes quadratiques, définies par seulement trois points de contrôle. Ainsi, il s'avère qu'une courbe de Bézier peut s'obtenir à partir d'une autre courbe de Bézier à l'aide la transformation affine définie entre les triangles de contrôle. Disposant d'une texture générique d'une courbe de Bézier, il est actuellement possible de tracer n'importe quelle autre. Néanmoins les questions de filtrages que soulèvement le rééchantillonnage d'une simple courbe diffèrent de celles liées aux surfaces.

Bibliographie

Tous les brevets américains sont disponibles en ligne : http://www.uspto.gov.

[1] OpenGL Architecture Review Board, *OpenGL Reference Manual : The Official Reference Document for OpenGL, Release 1.* Reading, MA, USA : Addison-Wesley, 1993. 26, 42

[2] Microsoft, « DirectX Web Site. » [En ligne]. Disponible : http://www.microsoft.com/directx/ 26, 42, 44

[3] D. Blythe, Ed., *OpenGL ES Common/Common Lite Profile Specification, Version 1.0.* Promoter of the Khronos Group, 2003. [En ligne]. Disponible : http://www.khronos.org/opengles/spec.html 26, 70

[4] « Digital Video Broadcasting (DVB) ; Multimedia Home Platform Specification 1.0.3, » European Telecommunications Standards Institute, ETSI Specification, Draft ES 201 812 V1.1.1 (2003-07), 2003. [En ligne]. Disponible : http://www.etsi.org 28

[5] G. E. Moore, « Cramming more components into integrated circuits, » *Electronics*, vol. 38, n° 8, avril 1965. 28

[6] P. N. Glaskowsky, « A concise Review of 3D Technology, » *MicroProcessor Report*, juin 1999. 37

[7] T. Mitra et T.-C. Chiueh, « Three-Dimensional Computer Graphics Architecture, » *Current Science : Special Section on Computational Science*, vol. 78, n° 7, pp. 101–109, avril 2000. [En ligne]. Disponible : http://www.comp.nus.edu.sg/~tulika/publications.htm 37

[8] K. Akeley et P. Hanrahan, « CS448A : Real-time graphics architectures, » Support de cours, Université de Stanford. [En ligne]. Disponible : http://www.graphics.stanford.edu/courses/cs448a-01-fall/ 37

[9] P. S. Heckbert, « Survey of Texture Mapping, » *IEEE Comput. Graph. Appl.*, vol. 6, n° 11, pp. 56–67, novembre 1986. 37, 42, 50, 52

[10] P. S. Heckbert, « Fundamentals of Texture and Image Warping, » Mémoire de Master, Dept. of Electronical Engineering and Computer Science, University of California, Berkeley, California, juin 1989. 37, 48

[11] P. Haeberli et M. Segal, « Texture mapping as A fundamental drawing primitive, » in *Fourth Eurographics Workshop on Rendering*, M. F. Cohen, C. Puech, et F. Sillion, Eds., 1993, pp. 259–266. [En ligne]. Disponible : citeseer.nj.nec.com/haeberli93texture.html 37

[12] J. D. Foley, A. van Dam, S. K. Feiner, et J. F. Hughes, *Computer Graphics : Principles and Practice – second edition in C.* Addison-Wesley Publishing Company, 1992. 37, 42

[13] L. McMillan et G. Bishop, « Plenoptic Modeling : An Image-Based Rendering System, » *Computer Graphics (SIGGRAPH' 95 Proceedings)*, pp. 39–46, août 1995. 38

[14] G. Farin, *Curves ans Surfaces for Computer Aided Geometric Design*, 2e ed. Academic Press, 1990. 38

[15] ISO/IEC 14772-1 :1997. *The Virtual Reality Modeling Language* (VRML 97), 1997. [En ligne]. Disponible : http://www.web3d.org/x3d/specifications/vrml/index.html 38

[16] C. Cunat, « 3D games scenes characterization for the specification of a unified 3D/MPEG4 Rendering Engine (MEGANE), » PRF, Rap. Tech. C 2001-787, octobre 2001. 39, 53, 54, 85, 193

[17] S. Mancini, « Architectures matérielles pour la synthèse d'image par lancer de rayon, » Mémoire de Thèse, École Nationale Supérieure des Télécommunications, janvier 2000, Thèse no ENST 00 E 023. 40

[18] B.-T. Phong, « Illumination for Computer Generated Pictures, » *Communications of the ACM*, vol. 18, no 6, pp. 311–317, juin 1975. 41, 42

[19] M. Segal et K. Akeley, *The OpenGL Graphics System : A Specification (Version 1.5)*, 2003. [En ligne]. Disponible : http://www.opengl.org/documentation/specs/version1.5/glspec15.pdf 42, 44

[20] « The Mesa 3D Graphics Library, » Mesa Home Page. [En ligne]. Disponible : http://www.mesa3d.org 42

[21] A. Kugler, « The Setup for Triangle Rasterization, » in *Proceedings of the 11th Eurographics Workshop on Graphics Hardware.* Poitiers, France : Eurographics Association, août 1996, pp. 49–58. [En ligne]. Disponible : http://citeseer.nj.nec.com/kugler96setup.html 42

[22] C. Priem et D. Kirk, « Method and apparatus for accelerating the rendering of graphical images, » Office américain des brevets, Brevet US6226012 B1, mai 2001. 42

[23] H. Gouraud, « Continuous Shading of Curved Surfaces, » *IEEE Trans. Comput.*, vol. 20, no 6, pp. 623–629, juin 1971. 42

[24] J. F. Blinn et M. E. Newell, « Texture and Reflection in Computer Generated Images, » *Communications of the ACM*, vol. 19, no 10, pp. 542–547, octobre 1976. 42, 50

[25] E. Catmull et A. Smith, « 3D Transformation of Images in Scanline Order, » *Computer Graphics*, vol. 14, no 3, pp. 279–285, 1980. 42

[26] I. E. Sutherland, R. F. Sproull, et R. A. Schumacher, « A Characterization of Ten Hidden-Surface Algorithms, » *ACM Computing Survey*, vol. 6, no 1, pp. 1–55, mars 1974. 43

[27] D. Cohen-Or, Y. Chrysanthou, C. Silva, et F. Durand, « A survey of visibility for walkthrough applications, » *IEEE Trans. Visual. Comput. Graphics*, vol. 9, no 3, pp. 412–431, juillet-septembre 2003. [En ligne]. Disponible : citeseer.nj.nec.com/cohen-or00survey.html 43

[28] J. Kessenich, D. Baldwin, et R. Rost, *The OpenGL Shading Language 1*, Février 2003. [En ligne]. Disponible : http://developper.3dlabs.com/ 44

[29] C. Miró Sorolla, « Architecture d'un accélérateur matériel pour la composition d'objets vidéo MPEG-4, » Mémoire de Thèse, École Nationale Supérieure des Télécommunications, février 2000, Thèse no ENST 2000 E 015. 45, 68

[30] G. Balestrat et A. Biaggi, « Le Signal Vidéo, » Support de cours, Département Traitement du Signal et de l'Image, ENST, 2000. 45

[31] S. Pasricha, S. Mohapatra, M. Luthra, N. Dutt, et N. Venkatasubramanian, « Reducing Backlight Power Consumption for Streaming Video Applications on Mobile Handheld Devices, » in *ESTImedia, First Workshop on Embedded Systems for Real-Time Multimedia*, Newport Beach, California, USA, octobre 2003, pp. 11–17. 46

[32] G. de Haan, *Video Processing for Multimedia Systems*, 2000. 47

[33] C. E. Shannon, « Communication in the Presence of Noise, » *Proceedings of the IRE*, vol. 37, n° 1, pp. 10–21, janvier 1949, reparu dans Proc. IEEE, vol. 86, no. 2, février 1998. 48

[34] L. Alves de Barros, « Architecture intégrée pour le ré-échantillonnage d'images animées, » Mémoire de Thèse, École Nationale Supérieure des Télécommunications, 1994, Thèse n° ENST 94 E 024. 48, 68

[35] G. Wolberg, *Digital Image Warping*. IEEE Computer Society Press, 1990. 48

[36] R. C. Lansdale, « Texture Mapping and Resampling For Computer Graphics, » Mémoire de Master, University of Toronto, janvier 1991. 48

[37] L. Atzori, F. G. B. D. Natale, et F. Granelli, « Adaptive Anisotropic Filtering (AAF) for Real-Time Visual Enhancement of MPEG-Coded Video Sequences, » *IEEE Trans. Circuits Syst. Video Technol.*, vol. 12, n° 5, pp. 285–298, mai 2002. 48

[38] F. C. Crow, « Summed-Area Tables for Texture Mapping, » in *Computer Graphics (SIGGRAPH '84 Proceedings)*, vol. 18. ACM Press, janvier 1984, pp. 207–212. 50

[39] L. Williams, « Pyramidal Parametrics, » in *Computer Graphics (SIGGRAPH '83 Proceedings)*, vol. 17. ACM Press, juillet 1983, pp. 1–11. 50

[40] J. P. Ewins, M. D. Waller, M. White, et P. F. Lister, « MIP-Map Level Selection for Texture Mapping, » *IEEE Trans. Visual. Comput. Graphics*, vol. 4, n° 4, pp. 317–329, octobre-décembre 1998. 50

[41] Y. Kamen et L. Shirman, « Graphics mechanism and apparatus for mipmap level estimation for anisotropic texture mapping, » Office américain des brevets, Brevet US6219064, avril 2001. 50

[42] L. Carpenter, « The A-buffer, an Antialiased Hidden Surface Method, » in *Computer Graphics (SIGGRAPH '84 Proceedings)*, vol. 18. ACM Press, janvier 1984, pp. 103–108. 50, 54

[43] A. Schilling, « A New Simple and Efficient Antialiasing with Subpixel Masks, » in *Computer Graphics (SIGGRAPH '91 Proceedings)*, vol. 25. ACM Press, juillet 1991, pp. 133–141. 50

[44] M. D. Waller, J. P. Ewins, M. White, et P. F. Lister, « Efficient Coverage Mask Generation for Antialiasing, » *IEEE Comput. Graph. Appl.*, vol. 20, n° 6, pp. 86–93, novembre/décembre 2000. 50

[45] D. W.-h. Wong et M. M. Aleksic, « Method and apparatus for a three-dimensionnal graphics processing system including anti-aliasing, » Office américain des brevets, Brevet US6172680, janvier 2001. 50

[46] nVIDIA, « HRAA : High-Resolution Antialiasing through Multisampling, » nVIDIA, Rap. Tech., 2001. 51

[47] K. M. Fant, « A Nonaliasing, Real-Time Spatial Transform Technique, » *IEEE Comput. Graph. Appl.*, vol. 6, n° 1, pp. 71–80, janvier 1986. 52

[48] P. Winser, T. Bonnet, D. Dumont, et Y. Mathieu, « Architectures for Mass Market 3D Displays, » in *Eurographics Conference Proceedings*, 1988, pp. 273–283. 54

[49] C. B. Owen et F. Makedon, « Bottleneck-Free Separable Affine Image Warping, » in *Image Processing, 1997. Proceedings., International Conference on*, vol. 1, 1997, pp. 683–686. 54

[50] PowerVR, « 3D Graphical Processing, » PowerVR, Rap. Tech., novembre 2000. 54

[51] Q.-L. Nguyen-Phuc et C. Miró Sorolla, « VLSI architecture of a MPEG-4 visual renderer, » in *Signal Processing Systems, 2001 IEEE Workshop on*, Antwerp, Belgique, 2001, pp. 309–320. 54, 146

[52] M. Cox et N. Bhandari, « Architectural Implications of Hardware-Accelerated Bucket Rendering on the PC, » in *1997 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, S. Molnar et B.-O. Schneider, Eds. New York City, NY : ACM Press, 1997, pp. 25–34. 54

[53] M. Chen, G. Stoll, H. Igehy, K. Proudfoot, et P. Hanrahan, « Models of the impact of overlap in bucket rendering, » in *1998 SIGGRAPH / Eurographics Workshop on Graphics Hardware*. New York City, NY : ACM Press, 1998. 54, 89

[54] O. Baltaretu, D. L. Digman, et S. O. Gupta, « Method for determining tiles in a computer display that are covered by a graphics primitive, » Office américain des brevets, Brevet US6437780 B1, août 2002. 55

[55] S. C. Heeschen et M. B. Zhu, « Circuit and method for processing render commands in a tile-based graphics system, » Office américain des brevets, Brevet US6380935 B1, avril 2002. 55

[56] M. B. Zhu et S. C. Heeschen, « Optimized system and method for binning of graphics data, » Office américain des brevets, Brevet US6344852 B1, février 2002. 55

[57] M. B. Zhu et S. C. Heeschen, « Circuit and method for deferring the binding of render states to primitives in a graphics system, » Office américain des brevets, Brevet US6323860 B1, novembre 2001. 55

[58] J. E. Bresenham, « Algorithm for computer control of a digital plotter, » *IBM Systems Journal*, vol. 4, n° 1, pp. 25–30, 1965. 57

[59] H. Fuchs, « Graphics Display System Using Logic-Enhanced Pixel Memory Cells, » Office américain des brevets, Brevet US4590465, mai 1986. 58

[60] H. Fuchs et J. Poulton, « VLSI Graphics Display Image Buffer Using Logic Enhanced Pixel Memory Cells, » Office américain des brevets, Brevet US4783649, novembre 1988. 58

[61] H. Fuchs, « Image Buffer Having Logic-Enhanced Pixel Memory Cells and Method For Settings Values Therein, » Office américain des brevets, Brevet US4827445, mai 1989. 58

[62] H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, et L. Israel, « Pixel-Planes 5 : A Heterogenous Multiprocessor Graphics System Using Process Enhanced Memories, » *Computer Graphics*, vol. 23, n° 3, pp. 79–88, juillet 1989. 58

[63] J. Pineda, « A Parallel Algorithm for Polygon Rasterization, » *Computer Graphics (SIGGRAPH '88 Proceedings)*, vol. 22, n° 4, pp. 17–20, 1988. 59

[64] S. Molnar, « Image Composition Architectures for Real-Time Image Generation, » Mémoire de Thèse, University of North Carolina at Chapel Hill, octobre 1991. 59, 89

[65] S. Molnar, J. Eyles, et J. Poulton, « PixelFlow : High-speed rendering using image composition, » *Computer Graphics (SIGGRAPH '92 Proceedings)*, vol. 26, n° 2, pp. 231–240, 1992. 59

[66] J. Eyles, S. Molnar, J. Poulton, T. Greer, A. Lastra, N. England, et L. Westover, « PixelFlow : The realization, » In 1997 SIGGRAPH/Eurographics Workshop on Graphics Hardware, pages 57–68, 1997. 59

[67] J. Poulton, S. Molnar, et J. Eyles, « Architecture and Apparatus for Image Generation, » Office américain des brevets, Brevet US5388206, février 1995. 59

[68] J. Poulton, S. Molnar, et J. Eyles, « Architecture and Apparatus for Image Generation utilizing Enhanced Memory Devices, » Office américain des brevets, Brevet US5481669, janvier 1996. 59

[69] K. Akeley, « RealityEngine Graphics, » *Computer Graphics (SIGGRAPH '93 Proceedings)*, pp. 109–116, août 1993. 60

[70] J. S. Montrym, D. R. Baum, D. L. Dignam, et C. J. Migdal, « InfiniteReality : a real-time graphics system, » *Computer Graphics (SIGGRAPH '97 Proceedings)*, vol. 31, n° Annual Conference Series, pp. 293–302, 1997. 60

[71] L. Kohn et N. Margulis, « Introducing the Intel i860 64-Bit Microprocessor, » *IEEE Micro*, vol. 9, n° 4, pp. 15–30, août 1989. 60

[72] J. Torborg et J. Kajiya, « Talisman : Commodity Real-time 3D Graphics for the PC, » in *Computer Graphics (SIGGRAPH '96 Proceedings)*. ACM Press, 1996, pp. 353–364. 62

[73] M. Randall, « Talisman : Multimedia for the PC, » *IEEE Micro*, vol. 17, n° 2, pp. 11–19, mars-avril 1997. 62

[74] S. Dutta, V. Mehra, W. Zhu, D. Singh, M. Janssens, R. Vengalasetti, B. Ben-Nun, P. Pothana, V. Adusumilli, N. King, L. L. John Yen-Han Huang, C. Nelson, J. Bannur, et S. Wu, « Architecture and Design of a Talisman-Compatible Multimedia Processor, » *IEEE Trans. Circuits Syst. Video Technol.*, vol. 9, n° 4, pp. 565–579, juin 1999. 62

[75] « HHP : Table of Graphics Chips. » [En ligne]. Disponible : http://www.hwhunpage.hu/gvt_e.htm 62

[76] J. E. Lindholm, S. Moy, K. Dawallu, M. Yang, J. Montrym, D. B. Kirk, P. E. Sabrella, M. N. Papakipos, D. A. Voorhies, et N. J. Foskett, « Transform, lighting and rasterization system embodied on a single semiconductor platform, » Office américain des brevets, Brevet US6198488, mars 2001. 63, 64

[77] « Accelerated Graphics Port, » AGP V3.0 Interface Specification, Revision 1.0, Intel, septembre 2002. 64

[78] D. Carey, H. Curtis, T. Hunter, et B. Weigler, « A Wolf in Sheep's Clothing, » *Microprossor Report*, octobre 2000. 65

[79] K. Diefendorff, « Sony's Emotionnally Charged Chip, » *Microprossor Report*, avril 1999. 65

[80] M. Oka et M. Suzuoki, « Designing and Programming the Emotion Engine, » *IEEE Micro*, pp. 20–28, novembre-décembre 1999. 65

[81] A. Kunimatsu, N. Ide, T. Sato, Y. Endo, H. Murakami, T. Kami, M. Hirano, F. Idhihara, H. Tago, M. Oka, A. Ohba, T. Yutaka, T. Okada, et M. Suzuoki, « Vector Unit Architecture for Emotion Engine, » *IEEE Micro*, vol. 20, pp. 40–47, mars-avril 2000. 65

[82] *EE User's Manual*, Sony Computer Entertainment Inc., avril 2001. 65

[83] *GS User's Manual*, Sony Computer Entertainment Inc., avril 2001. 65

[84] N. Olivero et J.-F. Fourmond, « Travail d'étude Xbox : architecture, sécurité et Linux, » Licence Informatique, Université Nice Sophia Antipolis, 2002-2003. 66

[85] A. Dasu et S. Panchanathan, « A Survey of Media Processing Approaches, » *IEEE Trans. Circuits Syst. Video Technol.*, vol. 12, n° 8, août 2002. 68

[86] N. Demassieux, « Architecture VLSI pour le traitement d'images : Une contribution à l'étude du traitement matériel de l'information, » Mémoire de Thèse, École Nationale Supérieure des Télécommunications, mars 1991, Thèse n° ENST 91 E 012. 68

[87] T. Masaki, Y. Morimoto, T. Onoye, et I. Shirakawa, « VLSI Implementation of Inverse Discrete Cosine Transformer and Motion Compensation for MPEG2 HDTV Video Decoding, » *IEEE Trans. Circuits Syst. Video Technol.*, vol. 5, n° 5, pp. 387–395, octobre 1995. 68

[88] C. Havet, « Architecture et implantation intégrée de systèmes de codage d'images : application à un nouvel algorithme d'estimation de mouvement, » Mémoire de Thèse, École Nationale Supérieure des Télécommunications, 1997, Thèse n° ENST 97 E 046. 68

[89] M. Hatamian et S. K. Rao, « A 100 MHz 40-TAP Programmable FIR Filter Chip, » in *IEEE International Symposium on Circuits ans Systems*, mai 1990, pp. 3053–3056. 68

[90] S. Rathnam et G. Slavenburg, « An Architectural Overview of the Programmable Multimedia Processor, TM-1, » in *Proceedings of COMPCON*. IEEE, février 1996, pp. 319–326. 68

[91] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. López-Lagumas, P. R. Mattson, et J. D. Owens, « A Bandwith-Efficient Architecture for Media Processing, » in *International Symposium on Microarchitecture*, 1998. [En ligne]. Disponible : http://citeseer.nj.nec.com/rixner98bandwidthefficient.html 68

[92] B. Khailany, W. J. Dally, U. J. Kapasi, P. R. Mattson, J. Namkoong, J. D. Owens, B. Towles, A. Chang, et S. Rixner, « Imagine : Media Processing with Streams, » *IEEE Micro*, vol. 21, mars-avril 2001. 68

[93] J. D. Owens, W. J. Dally, U. J. Kapasi, S. Rixner, P. Mattson, et B. Mowery, « Polygon Rendering on a Stream Architecture, » *2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pp. 23–32, août 2000. [En ligne]. Disponible : http://citeseer.nj.nec.com/owens00polygon.html 68

[94] J. D. Owens, « Computer Graphics on a Stream Architecture, » Mémoire de Thèse, Stanford University, novembre 2002. [En ligne]. Disponible : http://graphics.stanford.edu/papers/jowens_thesis/ 68

[95] M. Berekovic, H.-J. Stolberg, et P. Pirsch, « Multicore System-On-Chip Architecture for MPEG-4 Streaming Video, » *IEEE Trans. Circuits Syst. Video Technol.*, vol. 12, n° 8, pp. 688–699, août 2002. 68

[96] H. Kubosawa, H. Takahashi, S. A. Y. Asada, A. Asato, A. Suga, M. Kimura, N. Higaki, H. Miyake, T. Sato, H. Anbutsu, T. Tsuda, T. Yoshimura, I. Amano, M. Kai, et S. Mitarai, « A 1.2-W, 2.16-GOPS/720-MFLOPS Embedded Superscalar Microprocessor for Multimedia Applications, » *IEEE J. Solid-State Circuits*, vol. 33, n° 11, pp. 1640–1648, novembre 1998. 69

[97] M. Takahashi, T. Nishikawa, M. Hamada, T. Takayanagi, H. Arakida, N. Machida, H. Yamamoto, T. Fujiyoshi, Y. Ohashi, O. Yamagishi, T. Samata, A. Asano, T. Terazawa, K. Ohmori, Y. Watanabe, H. Nakamura, S. Minami, T. Kuroda, et T. Furuyama, « A 60-MHz 240-mW MPEG-4 Videophone LSI with 16-Mb Embedded DRAM, » *IEEE J. Solid-State Circuits*, vol. 35, n° 11, pp. 1713–1721, novembre 2000. 69

[98] H. Arakida, M. Takahashi, Y. Tsuboi, T. Nishikawa, H. Yamamoto, T. Fujiyoshi, Y. Kitasho, Y. Ueda, M. Watanabe, T. Fujita, T. Terazawa, K. Ohmori, M. Koana, H. Nakamura, E. Watanabe, H. Ando, T. Aikawa, et T. Furuyama, « A 160mW, 80nA Standby, MPEG-4 Audiovisual LSI with 16Mb Embedded DRAM and a 5GOPS Adaptive Post Filter, » in *ISSCC*, 2003. 69

[99] J. F. Bartlett, L. S. brakmo, K. I. Farkas, W. R. Hamburgen, T. Mann, M. A. Viredaz, C. A. Waldspurger, et D. A. Wallach, « The Itsy Pocket Computer, » Compaq Western Research Laboratory, WRL Research Report 2000/6, octobre 2000. 69

[100] W. R. Hamburgen, D. A. Wallach, M. A. Viredaz, L. S. Brakmo, C. A. Waldspurger, J. F. Bartlett, T. Mann, et K. I. Farkas, « Itsy : Stretching the bounds of mobile computing, » *Computer*, vol. 34, n° 4, pp. 28–37, 2001. [En ligne]. Disponible : citeseer.nj.nec.com/hamburgen01itsy.html 69

[101] « acceleon. » [En ligne]. Disponible : http://www.acceleon.com 70

[102] SVG Working Group, « Scalable Vector Graphics (SVG) 1.1 Specification, » W3C Recommendation 14 january 2003. [En ligne]. Disponible : http://www.w3.org/TR/2003/REC-SVG11-20030114/ 70, 180

[103] « Macromedia Flash (SWF) File Format Specification, version 6, » 2002. [En ligne]. Disponible : http://www.macromedia.com/software/flash/open/licensing/fileformat/ 70, 180

[104] J. Adélaïde, « Minutes of ARM 3D Seminar, » Philips Semiconductors France, Rap. Tech., octobre 2002. 70

[105] A. Stevens, « ARM 3D Graphics, » Présentation multimédia, janvier 2002. 70

[106] « Multimedia Solution for Mobile Phones, » Brochure Commerciale, ATI, 2004. [En ligne]. Disponible : http://www.ati.com/products/imageon2300/index.html 70

[107] C.-W. Yoon, R. W. J. Kook, S.-J. Lee, K. Lee, et H.-J. Yoo, « An 80/20-MHz 160-mw Multimedia Processor Integrated With Embedded DRAM, MPEG-4 Accelerator, and 3-D Rendering Engine for Mobile Applications, » *IEEE J. Solid-State Circuits*, vol. 36, n° 11, pp. 1758–1767, novembre 2001. 70

[108] R. Woo, C.-W. Yoon, J. Kook, S.-J. Lee, et H.-J. Yoo, « A 120-mW 3-D Rendering Engine With 6-Mb Embedded DRAM and 3.2-GB/s Runtime Reconfigurable Bus for PDA, » *IEEE J. Solid-State Circuits*, vol. 37, n° 10, pp. 1352–1355, octobre 2002. 70

[109] J.-H. Sohn, R. Woo, et H.-J. Yoo, « Optimization of portable system architecture for real-time 3D graphics, » in *ISCAS 2002*, 2002, pp. 769–772. 70

[110] « Paramètres de codage en studio de la télévision numérique pour des formats standards d'image 4:3 (normalisé) et 16:9 (écran panoramique), » Recommendation UIT-BT.601-5. 78

[111] « SMPTE Standard for Television — 1920 × 1080 Image Sample Structure, Digital Representation and Digital Timing Reference Sequences for Multiple Picture Rates, » Proposed SMPTE Standard, SMPTE 274M, The Society of Motion Picture and Television Engineers. 78

[112] K. Jack, *Video Demystified*, 3e ed. LLH Publications, 2001, ch. 3 : Color Spaces. [En ligne]. Disponible : http://www.video-demystified.com 78

[113] A. Boukelif, *La télévision haute définition*, ser. Collection Technique et Scientifique des Télécommunication. Masson, 1997. 79

[114] S. Howell, Ed., *OpenML V1.0 Specification 19 July 2001*, juillet 2001. [En ligne]. Disponible : http://www.khronos.org/openml/spec.html 82

[115] F. Catthoor, F. Franssen, S. Wuytack, L. Nachtergaele, et H. D. Man, « Global Communication and Memory Optimizing Transformations for Low-Power Signal Processing Systems, » in *VLSI Signal Processing Workshop*, octobre 1994, pp. 178–187. [En ligne]. Disponible : http://citeseer.ist.psu.edu/catthoor94global.html 86

[116] C. Berthet, « Going Mobile : The Next Horizon for Multi-million Gate Designs in the Semi-Conductor Industry, » in *39th Design Automation Conference*. ACM, juin 2002. 88

[117] S. Molnar, M. Cox, D. Ellsworth, et H. Fuchs, « Sorting Classification of Parallel Rendering, » *IEEE Comput. Graph. Appl.*, vol. 14, n° 4, pp. 23–32, juillet 1994. 89

[118] C. Cunat, J. Gobert, et Y. Mathieu, « Method of temporarily storing data values in a memory, » Demande de brevet européen — 1er Dépôt, N° Enregistrement 04 300218.7, avril 2004. 113

[119] C. Cunat, J. Gobert, et Y. Mathieu, « A coprocessor for real-time MPEG4 facial animation on mobiles, » in *ESTImedia, First Workshop on Embedded Systems for Real-Time Multimedia*, Newport Beach, California, USA, octobre 2003, pp. 102–108. 115

[120] K. Meinds, « Perspective transformation of images with a FIR-filter, » Nat.Lab., Rap. Tech. TN 2000/141, 2000. 119, 207

[121] K. Meinds, « Variable sample rate conversion without DC-ripple, » Nat.Lab., Rap. Tech. TN 2000/422, 2000. 119, 207

[122] K. Meinds et B. Barenbrug, « Resample hardware for 3d graphics, » in *Proceedings of the conference on Graphics hardware 2002*. Eurographics Association, 2002, pp. 17–26. 119

[123] V. S. Popescu, « Forward Rasterization : A Reconstruction Algorithm for Image-Based Rendering, » Mémoire de Thèse, University of North Carolina, mai 2001. 119, 129

[124] W. Badawy et M. Bayoumi, « A multiplication-free parallel architecture for affine transformation, » in *Application-Specific Systems, Architectures, and Processors, 2000. Proceedings. IEEE International Conference on*, juillet 2000, pp. 25–34. 120

[125] W. Badawy, « An SIMD architecture for texture mapping, » in *Electronics, Circuits and Systems, 2001. ICECS 2001. The 8th IEEE International Conference on*, vol. 1, septembre 2001, pp. 225–228. 120

[126] W. Badawy et M. A. Bayoumi, « A Low Power VLSI Architecture for Mesh-Based Video Motion Tracking, » *IEEE Trans. Circuits Syst. II*, vol. 49, pp. 488–504, juillet 2002. 120

[127] W. Attia, « Calculateur d'intersection rayon-quadrique, » Mémoire de fin d'études, École Nationale Supérieure des Télécommunications, juillet 1999. 123

[128] B. Barenbrug, « Rasterization and filtering in a forward texture mapping architecture, » Nat.Lab., Rap. Tech. TN 2003/315, 2003. 129, 134

[129] B. Barenbrug et K. Meinds, « Programmable pixel shading and forward texture mapping architecture, » Nat.Lab., Rap. Tech. TN 2003/314, 2003. 134

[130] C. Cunat et Y. Mathieu, « Method of rendering graphical objects, » Demande de brevet européen — 1er Dépôt, N° Enregistrement 03 300010.8, janvier 2004. 137

[131] « RNRT 99 - projet tempovalse - numero 17, » Terminal Expérimental MPEG4 POrtable de Visiophonie et Animation Labiale ScalablE, 1999. 141

[132] T. Chen et R. Rao, « Audio-Visual Integration in Multimodal Communication, » *Proc. IEEE*, vol. 86, n° 5, pp. 837–852, mai 1998. 141

[133] F. Lavagetto, « Converting speech into lip movements : a multimedia telephone for hard of hearing people, » *IEEE Trans. Rehab. Eng.*, vol. 3, pp. 90–102, mars 1995. 142

[134] I. S. Pandzic, J. Ostermann, et D. Millen, « User evaluation : Synthetic faces for interactive services, » *The Visual Computer*, vol. 15, n° 7-8, pp. 330–340, novembre 1999. 142

[135] M. Odisio et F. Elisei, « Clônage 3d et animation articulatoire du visage d'une personne réelle pour la communication parlée audiovisuelle, » in *Journées de l'AFIG*, Grenoble - France, 2000, pp. 225–232. [En ligne]. Disponible : http://www.icp.inpg.fr/~elisei/AFIG2000/ 142, 150

[136] F. Elisei, M. Odisio, G. Bailly, et P. Badin, « Creating and controlling video-realistic talking heads, » in *Auditory-visual Speech Processing Workshop*, Aalborg - Danemark, 2001. 142

[137] M. Preda et F. Prêteux, « Critic review on mpeg-4 face and body animation, » in *Proceedings IEEE International Conference on Image Processing (ICIP'2002), Rochester, NY*, vol. 3, 22-25 septembre 2002, pp. 505–508. [En ligne]. Disponible : http://www-artemis.int-evry.fr/Artemis/Publications/ 144

[138] F. M. Burgos, M. Kitahara, C. Joslin, et F. Vexo, « SNHC FAQ (Frequently Asked Questions) 10.0, » document ISO/IEC JTC 1/SC 29/WG 11 N5246, rencontre MPEG, Shanghai, Octobre 2002. [En ligne]. Disponible : http://www.chiariglione.org/mpeg/faq/mp4-snh/mp4-snh.htm 144

[139] *Ballyinx PCI64 prototyping board user's guide*, Nallatech, août 2000, nT107-0048 – Issue 3. 161

[140] *Ballyblue User Guide*, Nallatech, octobre 2001, nT107-0061 – Issue 3. 161

[141] K. Keutzer, S. Malik, et A. R. Newton, « From ASIC to ASIP : The Next Design Discontinuity, » in *International Conference on Computer Design ICCD*, 2002. [En ligne]. Disponible : http://www.gigascale.org/pubs/204.html 179

Septième partie

Annexes

Annexe A

Mesures expérimentales de jeux vidéo 3D

Nous incorporons ici de larges extraits d'une étude expérimentale que nous avons effectuée concernant des jeux vidéo commerciaux [16]. À l'aide d'un outil interceptant les appels aux différentes fonctions définies par les API classiques de programmation d'applications graphiques (OpenGL et DirectX), nous avons pu déduire certaines caractéristiques (nombre de triangles par scènes, informations concernant les textures, etc.) des applications graphiques que visait un accélérateur graphique pour téléviseur. Ces caractéristiques ont alors pu alimenter nos réflexions sur les thèmes des tailles de mémoires nécessaires pour stocker les données, les complexités en terme de puissance de calculs, d'accès à la mémoire et l'activité d'un *z-buffer*.

A.1 Context

[...]

A.1.1 Tools and methodology

PC games were tested to determine their requirements in term of complexity (such as number of triangles per frame, depth complexity, size of triangle and so on). As the source code is not available, it is not possible to directly reach these information. However, these games delegate the rasterization process to some API (application programming interface). Three APIs were used in those games: DirectX from Microsoft, GLIDE for 3Dfx graphic cards (this API will not be used anymore due to the recent acquisition of 3Dfx by *n*VIDIA) and OpenGL. API calls may be intercepted and analysed. The Graphic Performance Tool (GPT) is an application developed by Intel and was available on Intel's web site. It replaces Windows' DLLs by its own DLLs which capture API calls and resend them to the real API. GPT can handle OpenGL and DirectX7 calls.

In practice, we had some difficulties using GPT. First of all, at the time we used

GPT for 3D games evaluation, Intel announced that "[they were] no longer offering GPT for sale and the support for this product will end on August 31, 2001." Fortunately, a time limited demo version was still available within Philips. Nevertheless, with Windows98, this demo could not support DirectX applications: the installation was "OpenGL only". So tests were performed under Windows2000 where GPT was able to manipulate both OpenGL and DirectX applications.

This tool provides several working modes. These modes are more or less time consuming for the CPU. In this way, the program will not considerably modify games' behaviour if not necessary: for computing the frame rate, the tool must not interact with the 3D application for example. However, intercepting every API call is more time consuming and may interact with the application (typically, the frame rate will scale down). In order to alleviate this problem, we used a high-end PC with more processing power than requested by the games. Our configuration was:

- Pentium II @ 448 MHz
- Memory: 128 MB
- Graphic card: *n*VIDIA TNT2.

Moreover, in order to be able to test PlayStation1 games, an emulator was used. This emulator is associated with a graphic plug-in which renders the scene by calling the DirectX API. Then, GPT considers this emulator as any other game.

GPT log files are then analysed. A program was written in order to simplify this analysis. API calls are reinterpreted and a simple rasterizer completes log information. This program also allows to display the scene in a wireframe mode.

Figure A.1, A.2 and A.3 show GPT environment.

A.1.2 Characteristic of selected games

In this part, we will give some characteristics of games that our system will support.

A.1.2.1 Generality

Firstly, we identified three classes of games determined according to the scene complexity:

- Simple games: these games are generally basic 2D games. For example, chess games and Tetris are what we call simple games.
- Advanced games: these games are typically Playstation-1 or DreamCast games. Every quake-like may be ordered in this class.

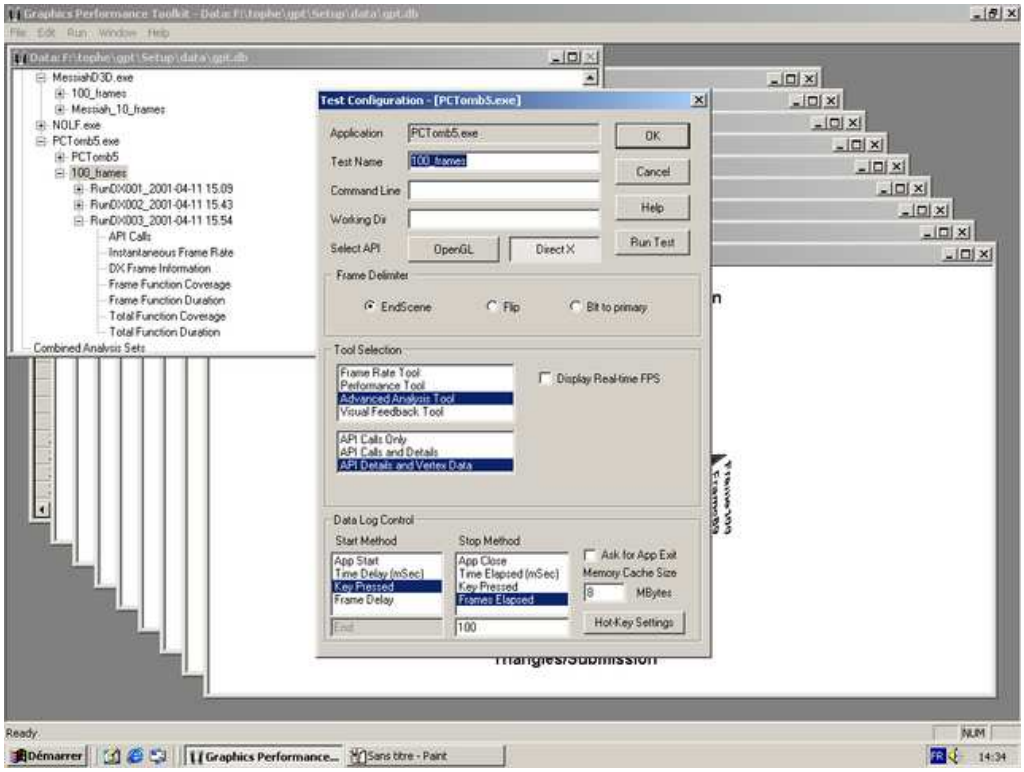


Figure A.1 : GPT main window and configuration window.

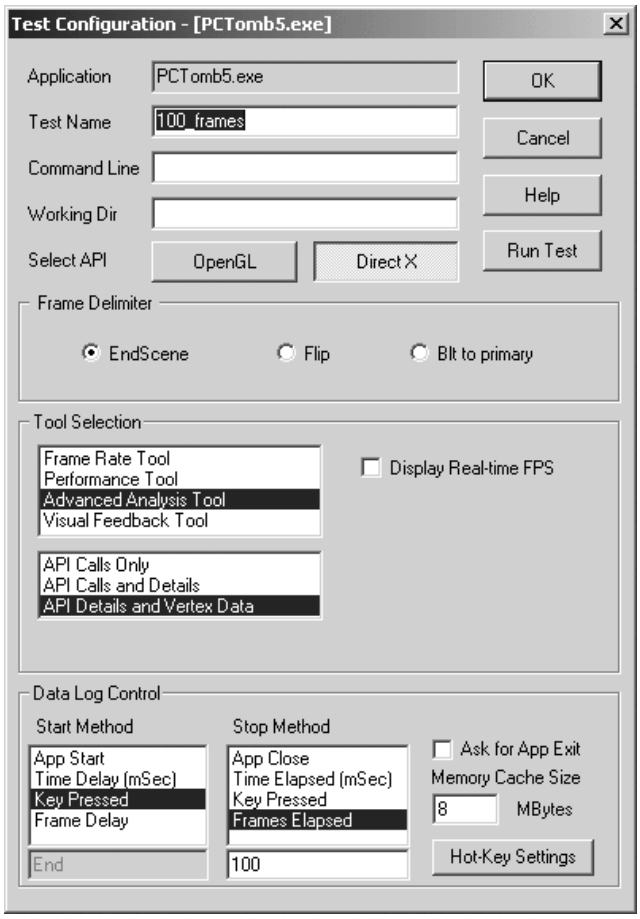


Figure A.2 : GPT configuration window (zoom).

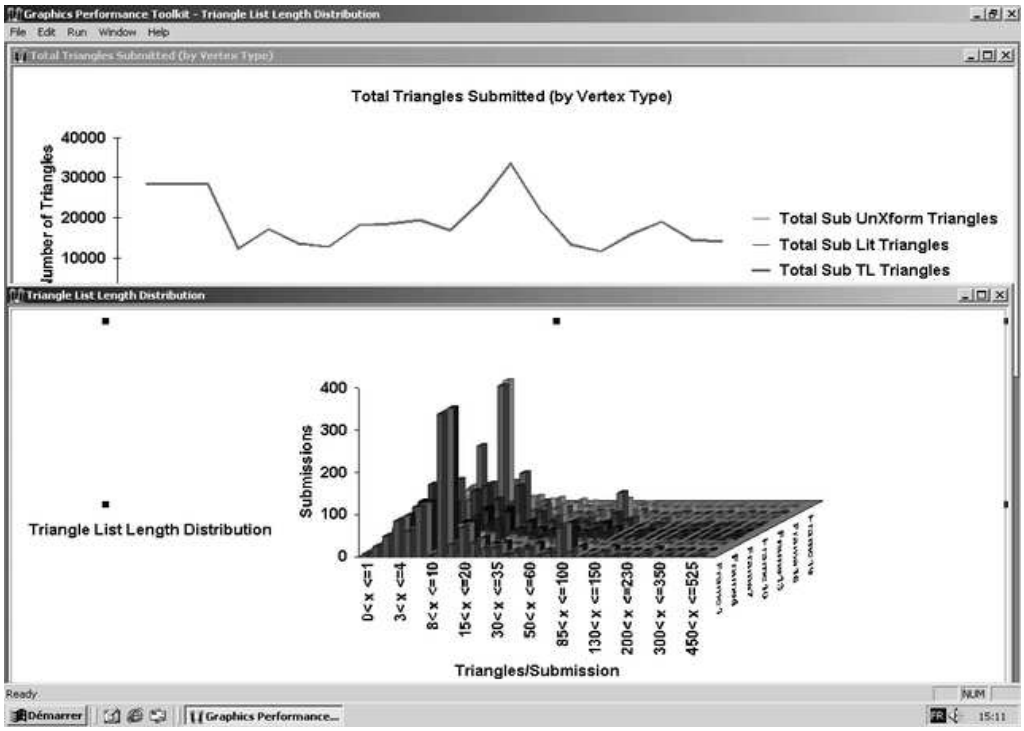


Figure A.3 : Some possible results using GPT.

- Cutting-edge games: new generation games will be ordered in this category. Games for the Playstation-2 or the future Microsoft's XBox are typically cutting-edge games.

Obviously, a cutting-edge game in 2001 can become an advanced-game in 2005. This classification is state-of-the-art dependent and, for example, a new DreamCast game can be a cutting-edge game. Image quality and scene complexity are the criteria for sorting games into this classification.

Among these three classes of games, our architecture will have to support games that, at this time, are defined as advanced games. "Quake-like" are targeted, but are not the only games we will have to support: games may be sorted according to five principal types.

- First-Person: Quake III, Tomb Raider...
- Race: Moto Racer2, Test Drive5...
- Flight: EF-2000, Lucas Art's Xwing, Decent2...
- Fight: Virtua Fighter II...
- Sport: NHL, NBA, FIFA...

Our architecture will have to support all these types of games. However, they do not need the same requirements in terms of data rate, number of triangles or frame rate. Due to difficulties with GPT (see next section), only first-person games were tested.

A.1.2.2 Tested games

The selected games have been tested using GPT operating on Windows2000 OS. Unfortunately, some applications did not support this OS and their installation has failed. Some others (the "Colin McRae 2" race game or "NHL2001" for instance) refused to launch themselves with GPT looking at their API calls.

Nevertheless, the games listed hereafter were tested. We now will briefly introduce them.

- First-person: Messiah, Vampire, Tomb Raider 5.
- Playstation One: Final Fantasy 8.

Messiah : This action game uses some API calls which let us analyse more deeply the scene. The next section will deal with it.

Vampire : API calls used are also usable for an in-depth analysis. Moreover, the scenes complexity, the textured details and lots of pre-computed 3D sequences make this action/role-playing game very interesting. However, it is almost considered as a cutting-edge game.

Tomb Raider 5 : This last episode uses interesting API calls using DirectX6 data structures and will be used in the next section.

FF8 : This emulated game stands for a typical PlayStation1 games.

Each game was tested in two modes : Frame rate tool and advanced analysis tool modes. Several scenes which looked as critical were logged and tested on a 100-frames basis.

[...]

A.2 Results using GPT

In figure A.1, we can see GPT's four modes into the configuration window: frame rate tool, performance tool, advanced analysis tool and visual feedback tool. The first one gives the instantaneous frame rate. The second gives API calls duration and the CPU workload. The third, the most important one in our case, can give very useful details such as primitive types, triangles per frame, texture information... In the fourth mode, the screen was divided into four regions: the actual scene, the scene untextured, a wireframe representation and a representation of the depth complexity. However, no data are logged so this mode is not exploitable but may provide a fast analysis of the

scene that can be used by the operator in order to identify interesting sequences. Next section deals with results from the first mode and the third mode is used for the other sections in this paragraph.

A.2.1 Frame rate

Visually acceptable frame rate will be discussed here. No objective criterion can be found to characterize this parameter, even if a set top box environment will probably target a frame rate of 25 or 30 fps (even 60 fps in Digital TV). So, the instantaneous frame rate was measured and players' visual analysis was used to decide the acceptable frame rate. The figure A.4 show two typical frame rates observed.

The first one called "Scene 1" was measured using GPT in a simple mode: no visible impact was noticed. For "Scene 2", in order to simulate more complex scene to render, the CPU load was artificial increased due to GPT in complete mode: instantaneous frame rate suddenly decreases down to 1 or 2 frames per second. Frame rates were tested with people from Philips Research France as witnesses. The first was said acceptable whereas the second was clearly rejected. The most important point was the appearance of frame fluidity. The first rate is quite constant (15 fps ±2). The second is clearly not: the most bothering feature is the repetitiveness of frame rate falls. The minimum number of frame per second is also disturbing: only one fall to 1 fps is sufficient to disturb players. However, some moderate falls were acceptable. In that case, the frame rate and the number of frames involved are critical.

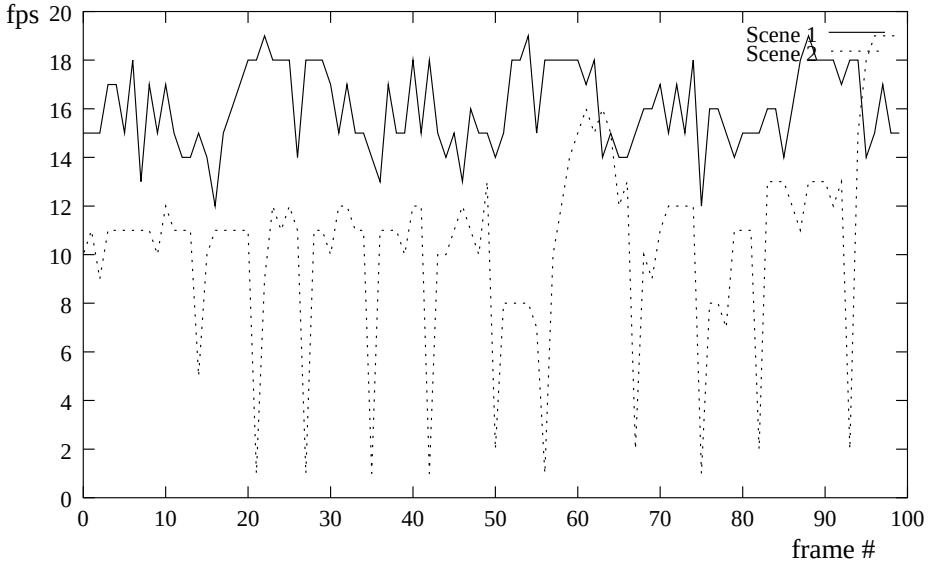


Figure A.4 : Instantaneous frame rate.

Conclusion

Only visual impressions were taken into account in this study. In that case, a frame rate about 14-17 fps is acceptable. For more convenience and in order to be less exposed to variations, an average frame rate of 20 fps is recommended. Of course, players would enjoy 30 or 50 fps which are compatible with the TV frame rate required by settop boxes. However, no noticeable difference was found for frame rates above 30. A frame rate of 20 fps seems to be a good visual compromise. Regarding instantaneous frame rate, variations may appear but must be limited in time and range. The minimum instantaneous frame rate shall never be less than 6 fps. And this rate must not exceed one frame. In order to assure some quality of services, a minimum of 10 fps during at most 1 second would be recommended (see table A.1). However, in a settop box context, the required frame rate would be 30 fps and the visual quality is ensured. Table A.1 proposes minimal frame rate. In the next sections, bandwidth will be estimated with 30 fps. Moreover, in order to estimate bandwidth, the following assumptions are made: a single memory holds the whole data, no cache is available and every data are (re-)fetched when necessary.

Av. rate	20 fps
Minimum	1s @ 10 fps

Table A.1: Minimal frame rate proposal on visual impression.

A.2.2 Primitives type

In this section, 3D primitives are analysed in order to estimate the number of triangles per frame. Number of vertices and bandwidth are also estimated.

A.2.2.1 PC games

We will begin with those kinds of games. To illustrate this part, figure A.5 shows a triangle view of a scene in Vampire (all wireframe view in this document are clipped at the resolution of 640x480). We have chosen a resolution of 640x480 because it is similar to the standard video definition of 720x576. Among tested games, Vampire was the one with the highest number of triangles per frame. Moreover, this game is nearly considered as a cutting-edge game: so this game will be considered as our upper limit in terms of number of triangles. Figure A.6 represents the evolution of the number of triangles over 100 frames for each sequence. Most common scenes share about 1,000-4,000 triangles. High complex scenes such as fight in Vampire need nearly 20,000 triangles depending on the view point. In that case, important variations within the number of triangles are possible. Although this game is close to the limits of our application area requirements, we can notice that an architecture must support high variations around the average number of triangles. We are interested in evaluating this average number. If a peak number of triangles appends, the frame rate would decrease.

Nevertheless, such a figure is interesting and may decrease the number of triangles supported by our chips. Initially, 50,000 triangles were targeted. **15,000 triangles per frame** in average would be recommended: 10,000 would be sufficient, plus 5,000 as a safety margin. Moreover, increasing the number of triangles in order to get a better resolution seems to be losing momentum: improvement of light modeling and techniques such as vertex or pixel shaders seem to be better candidates for next years to increase realism.

Thus, in average, 15,000 triangles have to be rendered at the frame rate proposed in table A.1. With less triangles, the frame rate can increase, with more triangles the frame rate can decrease.

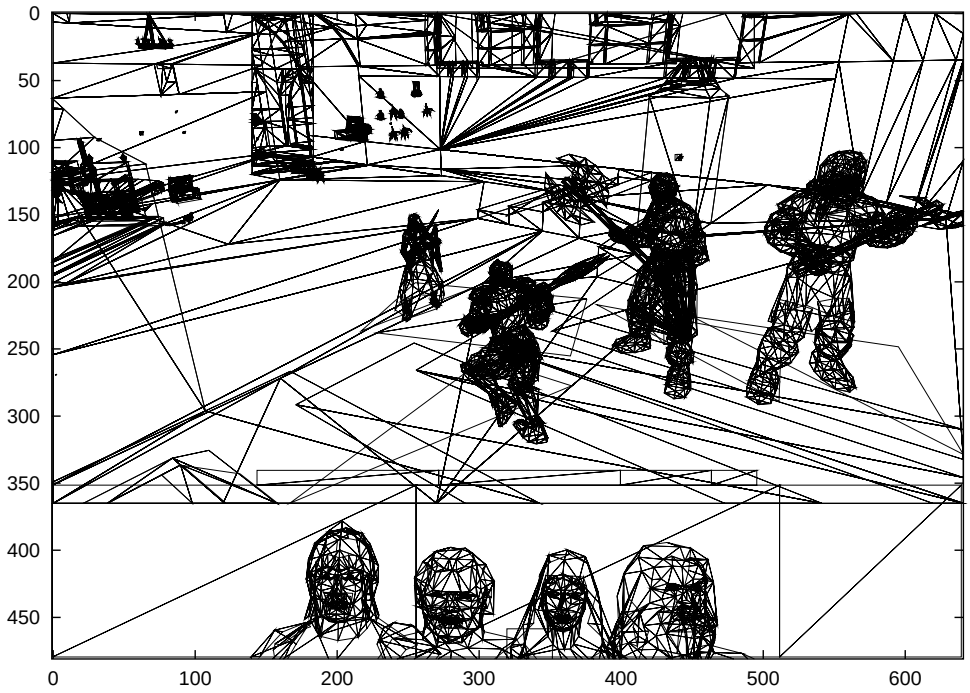


Figure A.5 : Triangle view of Vampire (17,172 triangles).

The number of vertices needed to define those triangles depends on the primitive type: 3 kinds of primitives are possible (triangle lists, fans or strips) and those primitives can be indexed. In a traditional way, a triangle list is made of $3n$ vertices which define n triangles. In the indexed way, a triangle list is made of m vertices and $3n$ indices which define n triangles. This can help to reduce the number of primitives and use efficient memory storage (1 index need less memory than 1 vertex). However, the primitive choice is done by the 3D engine. It builds the scene but does not take care of displaying it: that is the role of the rasterization done by OpenGL or DirectX. For instance, the 3D engine in TombRaider was made in order to use primitives with the DirectX6 rasterizer and this engine does not use indexed primitives. On the opposite side, more recent games such as Vampire or Messiah (in our tested games) define primitives with index.

In our case, this distinction is important in order to estimate the memory needed to store our scene: in the case of tile based rendering, all triangles (vertices and indices) have to be memorized during the tile sorting processing. The tables A.2, A.3 and A.4 group the results and their analysis comes hereafter.

Prim. Type	Messiah		Vampire		TR5	
	Nb Prim.	Nb Vert.	Nb Prim.	Nb Vert.	Nb Prim.	Nb Vert.
List	341,822	1,025,466	0	0	11,053	959,931
Fan	0	0	2,467	9,868	0	0
Strip	18,534	74,136	0	0	0	0
Total	360,356	1,099,602	2,467	9,868	11,053	959,931

Table A.2: Simple primitives for the 900 tested frames.

Prim.Type	Messiah			Vampire		
	Prim.	Vert.	Ind.	Prim.	Vert.	Ind.
Indexed List	18,025	278,427	231,150	47,449	2,447,663	6,627,411
Indexed Fan	1,554	7,872	7,872	13,224	67,291	58,513
Indexed Strip	18,905	246,228	143,367	0	0	0
Total	38,484	532,527	372,389	60,673	2,514,954	6,685,924

Table A.3: Indexed primitives for the 900 tested frames.

Primitive type	Simple Primitives			Indexed Primitives			
	Prim.	Vert.	Triangles	Prim.	Vert.	Ind.	Triangles
List	94.4%	95.9%	661,799	66%	89.4%	97%	2,286,187
Fan	0.6%	0.5%	4,934	14.9%	2.5%	1%	36,829
Strip	5%	3.6%	37,068	19.1%	8.1%	2%	105,557
Simple vs Indexed	79%	40.4%	703,801	21%	59.6%	-	2,428,573

Table A.4: Summary for the 900 tested frames.

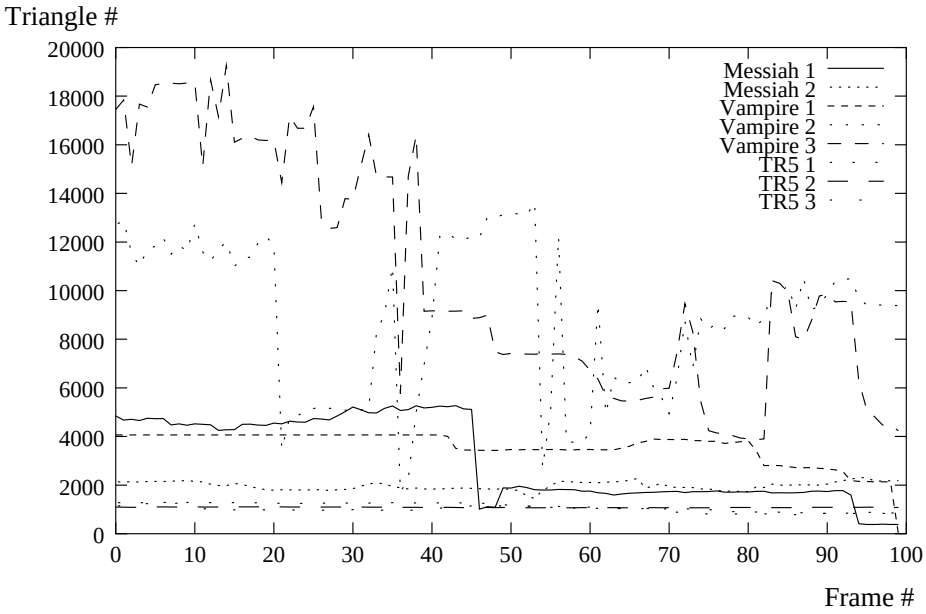


Figure A.6 : Number of Triangle for first person tested games.

First, we can see that results depend on the application: Tomb Raider never uses indexed primitives, Messiah uses lists and indexed lists but the latter are under-optimized (0.83 indexes/vertex) and Vampire uses mostly indexed primitives and particularly, indexed lists are well optimized (2.71 indexes/vertex). However, in order to estimate information for the targeted number of triangles, the whole data from these 3 games will be merged leaving aside games-dependent characteristics.

Then, 3,132,574 triangles (simple and indexed triangles) for 900 frames are generated: about 3,500 triangles per frame in average. Once again, 15.000 will allow us to deal with high variation rates per frame. However, "simple primitives" which represent 79% of primitives calls take only 22.5% of triangles. Shortly, table A.5 gives an interpolation of the repartition of vertices, indices and primitives in the case of 15,000 triangles. A typical application will reference 2.300 primitives, 25,000 vertices and 34,000 indices: in practice, 20,000 vertices are saved due to indices and fan or strip primitives (theoretically, a maximum of 45,000 vertices are needed in order to define our triangles). The ratio vertices per triangle falls to 1.63 for this *average* application.

In order to estimate the memory space needed to store the scene, we will use the following assumptions:

- Indices: **2 bytes.**
- Vertices: Vertices are transformed before our rendering engine.
  - Coordinates: $x,\ y,\ z,\ w^{-1}$ 16 bytes;

Primitive type	Simple Primitives			Indexed Primitives			
	Prim.	Vert.	Triangles	Prim.	Vert.	Ind.	Triangles
List	1,690	9,510	3,170	314	13,054	32,841	10,947
Fan	12	48	24	71	361	319	177
Strip	89	354	177	91	1,180	687	506
Total	1,791	9,912	3,371	476	14,595	33,847	11,630

Table A.5: Repartition of primitives for targeted number of triangles.

  - Diffuse and specular Color: 8 bytes;
  - Texture coordinates: 2 sets of textures (cf. section A.2.3.1): 16 bytes;
  - Total: **40 bytes.**

- Primitives:
  - Indexed Primitives: 1 byte for the primitive type, 2 bytes for the description of the vertices, 4 bytes as a pointer to the list of vertices to be used in the primitive sequence, 2 bytes to define the number of vertices in the list, 4 bytes as a pointer to the list of indices, 2 bytes to define the number of indices. **15 bytes.**
  - Simple Primitives: 1 byte for the primitive type, 2 for the description of the vertices, 4 bytes as a pointer to the list of vertices and 2 to define the number of vertices. **9 bytes.**

Thus, less than 1.05MB (1.02MB more precisely) are needed to store indices and vertices for the whole scene. This memory space is required for a retained rendering (needed for example in a tile architecture). Like this, the minimal bandwidth to read the definition of the scene is about 31.5MB/s.

Cuijpers[1] observed scenes with less than 5,000 vertices in average (with respect to table A.5, it represents about 3,100 triangles) and planed 8,000-22,500 vertices for today's (year 2000) games (about 5,000-13,800 triangles). We only see 3,500 triangles in average: +13% within 3 years (4.1% per year). If games follow this evolution, the number of triangles in 2005 will be about 4,700. It may appear difficult to estimate next years evolution, but 15,000 triangles for 2005 would represent +35% per year: 15,000 triangles appear to be realistic for our range of applications.

However, new games under development such as DoomIII or QuakeIV based on a new 3D engine may make those recommendations obsolete. These recommendations have been given being aware of this uncertainty.

In conclusion, this generalization is certainly subject to objection : this part highly depends on the application to rasterize. It is really hard to ensure that the repartition

[1] M. Cuijpers, "Statistics of 3D graphics applications for windows terminal," Nat.Lab. Tech. Note 211/98, 1998.

of the primitives in a particular application will exactly follow our distribution (we bet it will not). However, our number of triangles seems to be a good compromise. Thus, table A.5 sums up our reference repartition for the 15,000 targeted triangles.

A.2.2.2 PlayStation Games

Figures A.7 and A.8 show the same scene taken from FF8. The number of triangle is equivalent to TombRaider5. Besides, due to the emulation and the translation of PlayStation API to DirectX, the rendering part seems not to be really optimized: triangles are not clipped in a previous pipeline stage; the rasterizer must do it, increasing its complexity.

However, the average number of triangles per frame is less than 2,000. Strip primitives are used to connect 2 triangles in order to reduce the vertices by triangle ratio to 2. Table A.6 recapitulates relevant information.

Primitive type	Primitives		Vertices		Triangles	
	Nb	%	Nb	%	Nb	%
List	97,917	72.5	293,751	66.42	97,917	56.87
Strip	37,135	27.5	148,540	33.58	74,270	43.13
Total	135,052	-	442,291	-	172,187	-

Table A.6: Summary for PlayStation game (88 frames).

Bringing back those results to 15,000triangles gives the table A.7. With the same assumptions as previously (40 bytes per vertices and 9 bytes per primitive), the memory space needed to store our scene would be 1.57 MB and the associated bandwidth 47.1 MB/s with the frame rate from table A.1.

Primitive type	Prim.	Vert.	Triangles
List	8,531	25,593	8,531
Strip	3,235	12,940	6,470
Total	11,766	38,533	15,001

Table A.7: Repartition of primitives for targeted number of triangles.

PlayStationI was released in 1995. Then, even most recent games developed for this platform are limited by five years old capabilities. First games for PlayStationII would have been probably more interesting to test. However, no emulator exist nowadays.

Conclusion

To conclude this discussion about primitives types, the way the application manages its scene influences the number of triangles to compute. We can optimistically think the application will globally keep the repartition as smart as possible (as discussed in

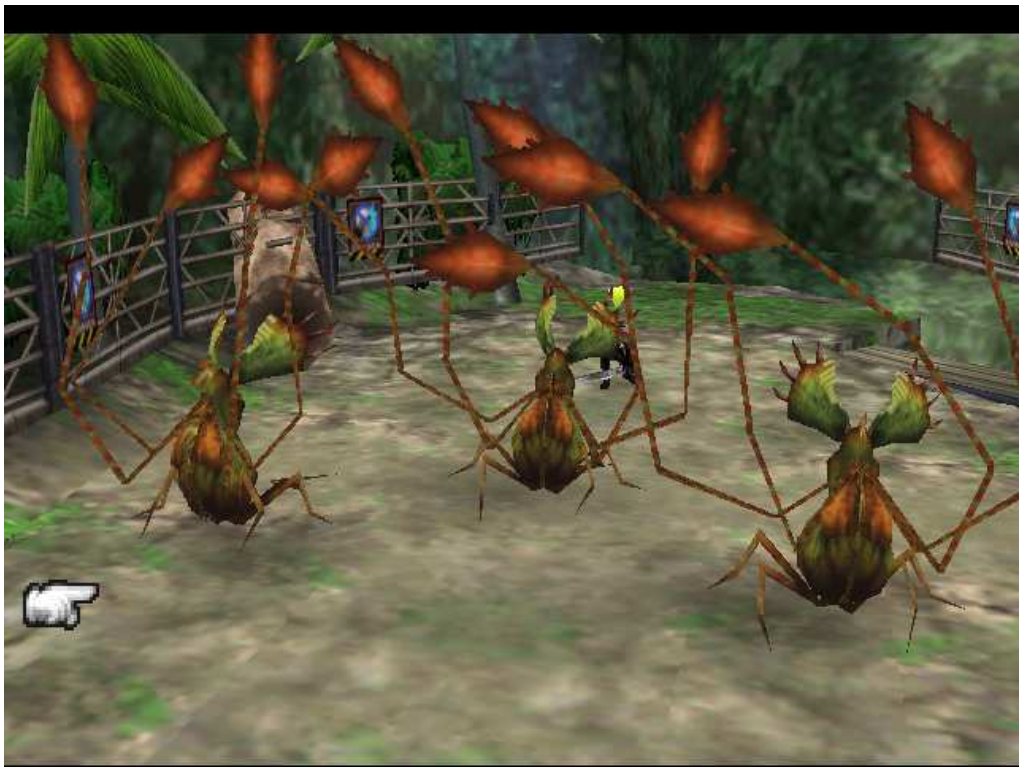


Figure A.7 : Screen Shot of Final Fantasy 8.

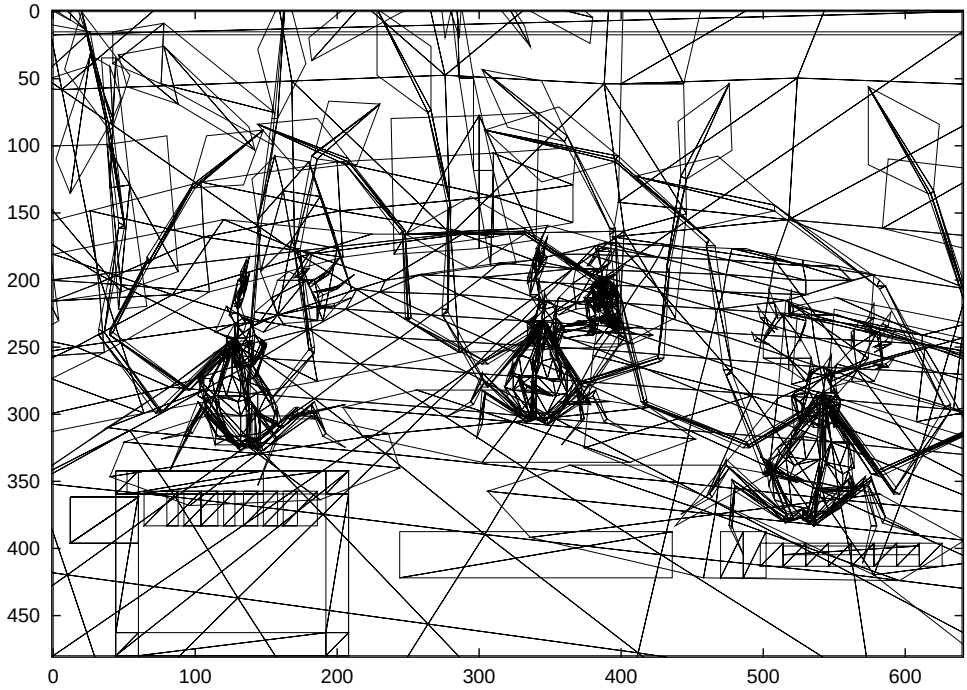


Figure A.8 : Triangle view of Final Fantasy 8 (2,013 Triangles).

section A.2.2.1) with the least possible data and bandwidth. Pessimistically, 1 triangle per indexed primitive is possible: then 15,000 indexed primitives with 45,000 vertices. Table A.8 sums up those cases.

	Smart case	PS1 games case	Worst case
Memory (1 frame)	1 MB	1.5 MB	2 MB
Bandwidth	30 MB/s	45 MB/s	60 MB/s

Table A.8: Scene management and binded memory.

A.2.3 API calls

We have already started to discuss about API calls with the distribution of primitives. However, API calls are not limited to function to render a primitive: control calls are a significant part of calls. Particularly, texture management and state management will be explored.

A.2.3.1 Texture calls

Textures are a key point in 3D games. They bring the biggest part of the realism in a scene. Samples from the input image (the texture) are mapped to the output samples on the screen. Texels (texture elements) have to be filtered in order to minimize resampling artefacts. Texture use can also break down performances due to the high bandwidth required in order to perform texture mapping.

The number of textures per primitive is, for instance, of great interest. DirectX API allows to support up to 8 textures for the same primitive. However, in tested games, the number of textures for a primitive is never higher than 2: 70.2% of primitives are double textured and 29.8% are simple textured. We now face two choices. On the one hand, we may imagine a double texture mapping unit capable of working in parallel. In this case, half of the silicon would be unused a third of time. On the other hand, a single unit will need to loop and reduce performances 70% of time.

However, looking at the details, surprises can be found: some of the double textured primitives are rendered with only their first texture defined. No explication was found to answer this phenomenon. Thus, the number of double textured primitives is artificially increased. Unfortunately, the real number of simple textured primitive was not found. The point is that previous results have to be considered as a worst case.

With the number of textures needed and their size, we will be able to estimate the texture memory. That texture memory is also a key point and its management too. Two other games (Giants and No One Lives Forever: cutting-edge games) were also used in this study. Table A.9 shows results.

Several information can be extracted from table A.9. First, the number of textures created increases for new games. This corroborates that game developers would empha-

Games	Frames	Texture Created	Total memory	Tex./frame	Mem./frame (MB)	Texture size
Messiah	100	287	71.8 MB	10–15	3.8	256x256
	100	435	108.8 MB	10–15	3.8	
	100	408	102.0 MB	20–25	6.3	
Vampire	200	405	28.8 MB	50–60	4.3	16x16 up to 256x256
NOLF	32	907	48.2 MB	60–70	3.7	8x8 up to 512x512
	100	1,086	52.3 MB	60–70	3.4	
Giants	1	-	-	41	13.1	16x32 up to 1024x1024

Table A.9: Texture results, texture 32 bits.

size realism with textures or light effect rather than increasing the number of triangles. Moreover, the size of those textures increases too up to 1024x1024 with a standard texture resolution of 4 bytes/texel. Although this size is 16 times less than the maximum size supported by typical PC hardware (4096x4096), this already represents 4 MB of information and requires a pretty large bandwidth. Besides, we do not take into account of the overhead in term of memory space due to mipmap levels (a texture mipmapped is one third larger than a texture not mipmapped).

If the number of textures created for a whole sequence is quite impressive, the number of textures used in a single frame is more reduced. Only Vampire exceeds 50 textures per frame, but uses smaller textures than other games. The number of textures is obviously linked to the memory size needed to store the texel. Further studies must be led to refined these following figures but at this point, we suggest: **6.3 MB** of memory space are required for a single frame. Due to the high probability of texture reuse for next frame, we expect 50% of texture reuse in that memory. The loading of the new 50% must be as fast as possible: less than 16ms (half the time allowed for the generation of a frame at 30 fps). Thus the required bandwidth would be 189 MB/s. This value will be our average case. The worst case will be indeed 378 MB/s when all textures have to be reloaded. Then, that memory will be heavily accessed in order to perform the texture mapping. A rough approximation would give 4 texels/pixel (bilinear filtering), 4 bytes/texel, screen size of 640x480, depth complexity of 2.34 (see section A.3.1), 16 ms to render the scene (first 16 ms are used to load textures): 658.28 MB/s. Obviously, this estimation does not take into account of new texture mapping techniques developed in [120, 121].

A.2.3.2 State changes

The OpenGL or DirectX pipelines are in fact state machines. Their states are set and remain unchanged until a function call orders to change. Typically, the function SetRenderState may change any state of the machine according to its parameters. Those state changes are critical for several reasons: first, the pipeline may have to be emptied before the new state can be applied, depending on how this pipeline is implemented. Secondly, due to our tile based approach and the retained associated mode, state changes have to be recorded and carefully handled during the tile sorting. Indeed, OpenGL specifications emphasize that API calls have to be handled in the order they appear (the immediate mode): primitives are rendered through a FIFO (First-In, First-Out) pipeline. However, the last primitive (for example) has to be rendered while the rasterizer proceeds the tile in which it appears and this tile will probably not be the last performed. Tile based rendering typically needs a deferred mode and will have to emulate the immediate rendering mode. In order to illustrate this point, imagine a simple scene that requires two states. The first state may be set when rendering begins and the second set in the middle of the rendering process. Now, tile based rendering can imply state swapping at each tile. The number of states may increase dramatically.

In order to estimate the amount of memory needed to store those calls, the number of state change calls are counted (see figure A.9).

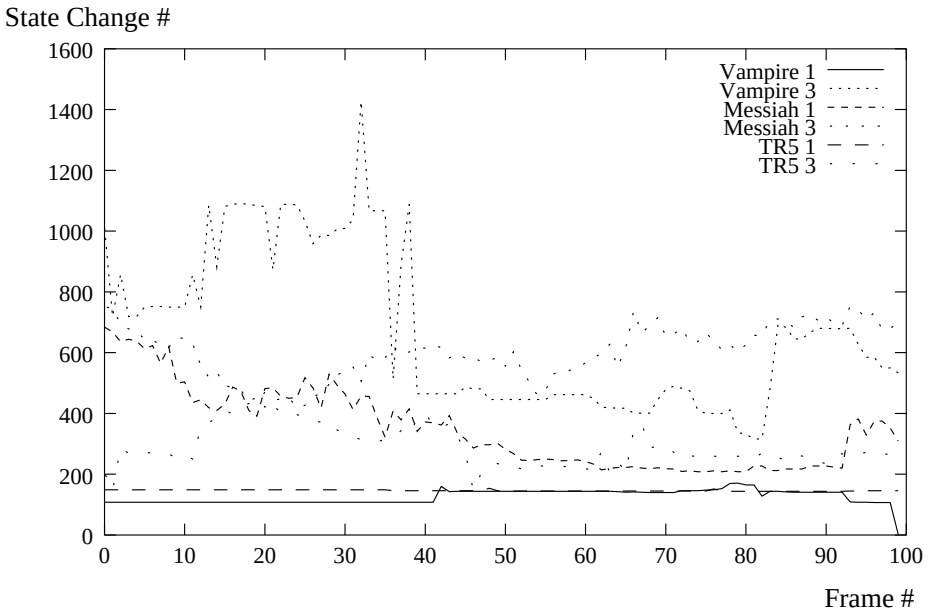


Figure A.9 : State Changes.

Three state changes were found: *setRenderState*, *setTexture* and *setTextureStageState*. Table A.10 shows the repartition of those changes. Once again, it depends on the application. Moreover, the number of state changes depends also on the number

Games	Frames	setTexture	setRenderState	Tex. stage	Nb Prim.
Messiah	100	17,033	12,360	5,246	
	100	8,358	5,314	2,844	
	100	23,120	22,077	14,775	
		43.65%	35.77%	20.58%	398,840
Vampire	100	4,380	3,957	3,637	
	100	17,743	69,206	43,497	
	100	12,469	33,129	20,285	
		16.61%	51.03%	32.37%	63,140
TR5	100	1,722	4,866	0	
	100	2,954	8,216	0	
	100	2,693	10,031	244	
		23.98%	75.22%	0.79%	11,053
Total		25.84%	48.31%	25.85%	473,033
Targeted application	1	433.59	810.68	433.85	2,267 (cf. table A.5)

Table A.10: State change type.

of primitives. From table A.5, we know that a typical scene will handle about 2,300 primitives. From the number of primitives and state changes observed, we may estimate our chip will typically manage 1,700 state changes. Then, from table A.10, those calls will split up into 850 *setRenderState*, 425 *setTexture* and 425 *setTextureStageState*.

The two first ones need two parameters while the last one needs three. Each parameter needs 4 bytes and we need 1 byte to identify the API call. Then, in a simple approach, the entire calls needs 16.6 kB to be stored. However, in the worst case (the last primitive is rendered within all tiles), we may read all API calls for each tile within the frame (typically 300 32x32 tiles): the required bandwidth is then 145.90 MB/s.

State changes may appear as a disadvantage for a tile based architecture: indeed, they need a particular attention but we analyse pros and cons of both approaches in a next section.

To conclude this section, GPT was a useful tool. It helps to understand how the 3D application works. We also have been able to specify some important features of our future architecture (frame rate, number of triangles, primitives repartition...). However, it is sometimes limited and some questions (mostly about tiles) can not be answered with a simple analysis of GPT log files.

A.3 "Advanced Results"

GPT does not provide any written information related to the Visual Feedback Tool. A program was then written to get data such as depth complexity for instance using informations logged into the advanced analysis mode. Results are presented hereafter.

A.3.1 Depth complexity

A frame is defined with primitives made of a plurality of triangles. Those triangles may overlap each other. Due to this overlapping, a pixel may be rendered more than once. The final pixel is the one with the least z value. We call "depth complexity" the number of triangles overlapping a single pixel. Using alpha blending, those triangles can contribute or not to the pixel final color. Figures A.10, A.11 and A.12 show the mean depth complexity while figure A.13 shows the maximum depth complexity.

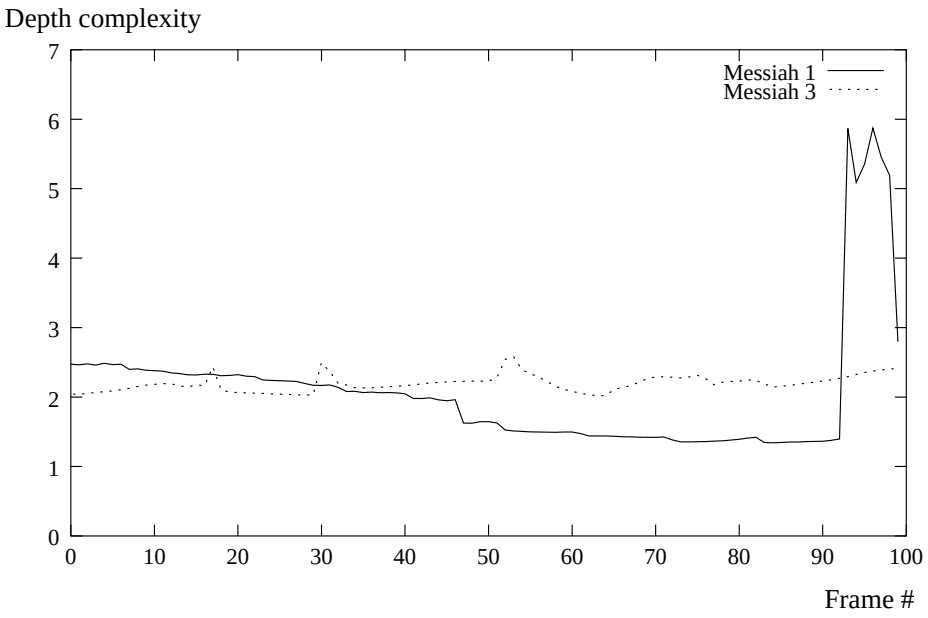


Figure A.10 : Mean depth complexity for Messiah.

This depth complexity allows to measure the z-buffer activity and frame buffer memory bandwidth required to render the scene. According to figure A.13, Vampire seems to be the most complex with a maximum depth complexity of more than 150: this depth complexity is due to the high number of particles modelised in the central bowl. However, mean value is still about 2.5 and median about 3. As this parameter (number of particles modelised) may be defined in the setup screen, we do not have to worry too much about maximum depth complexity in Vampire. Moreover, whatever the game, the distribution is typically shown by figure A.14: the distribution may be

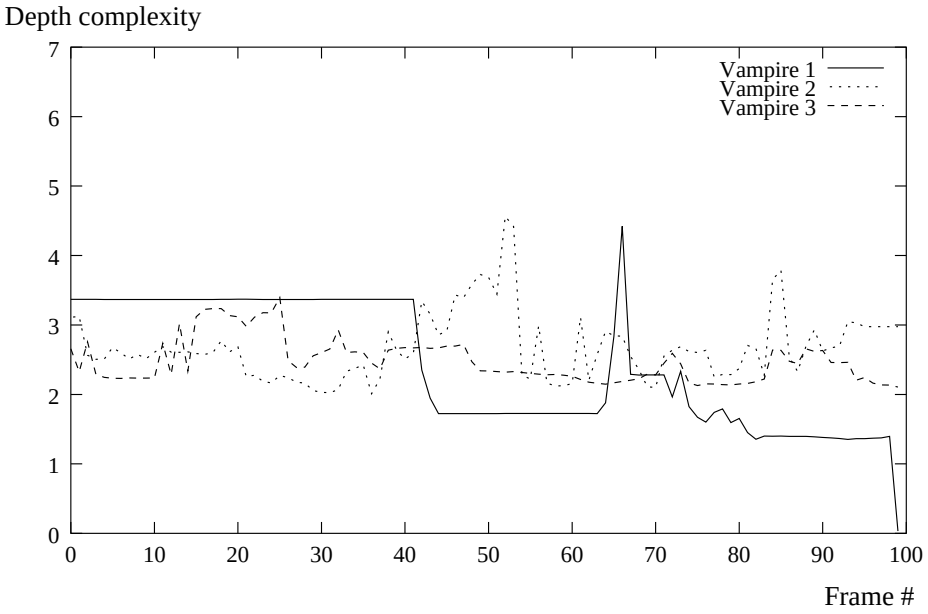


Figure A.11 : Mean depth complexity for Vampire.

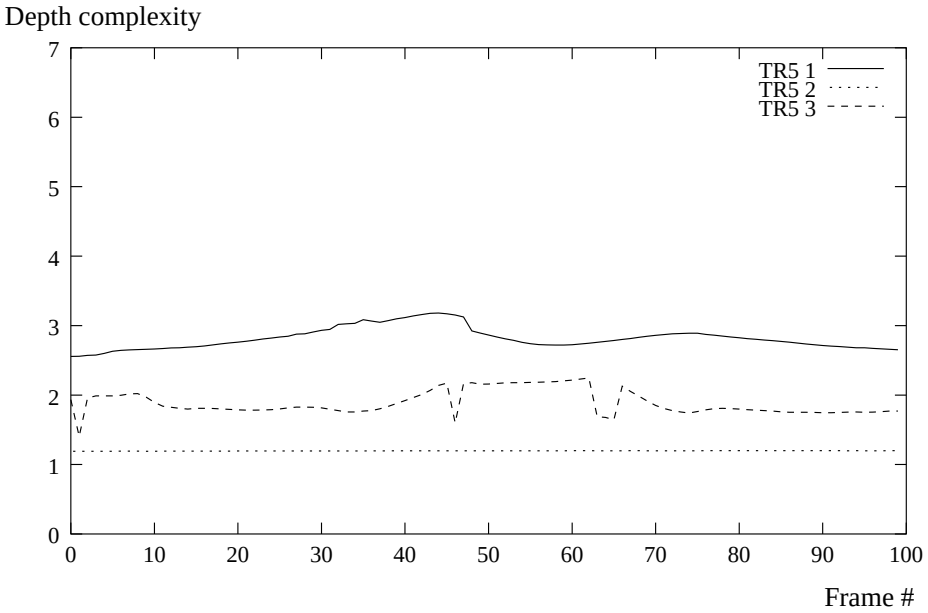


Figure A.12 : Mean depth complexity for tomb raider 5.

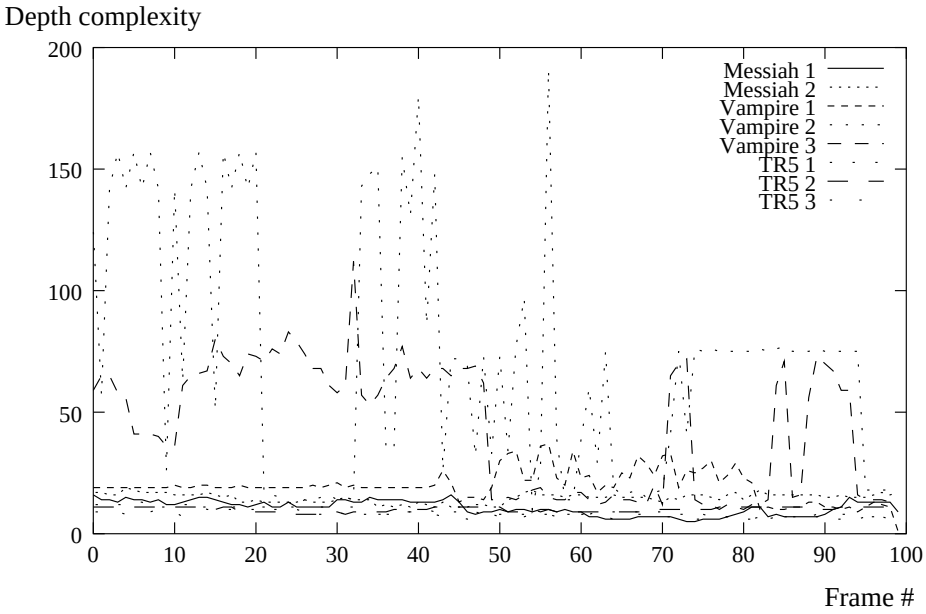


Figure A.13 : Maximum depth complexity for the previous games.

approximated in a rough estimation by a Gaussian distribution. We will use the mean depth. This mean value is **2.34**, taking into account of all sequences.

The activity of the Z-buffer is obviously linked to this depth complexity. Let's calculate this activity. A more precise demonstration can be found in [section] A.4:

- Number of pixels with a depth complexity of n: we use the gaussian approximation of figure A.14 with mean and variance depth complexity values from this particular frame. This distribution has to be refined and generalized (a Poisson distribution may appear more suitable): $f(n) = \frac{140000}{640*480} e^{-\frac{(n-2.172943)^2}{2*1.216667}}$
- Readings: for a depth complexity of n, we need n readings. For the whole frame: $\sum_{k=1}^{\infty} k \cdot f(k)$. Of course, this result is also the mean depth complexity multiplied by the number of pixels.
- Writings: We need to estimate the number of writings for a pixel at a given depth complexity n. This number may be expressed as:

$$\sum_{k=1}^{n} P(z_i \text{ must be written in the Z-buffer})$$

where P is the probability and z_1 is the first submitted pixel z value, z_2 the second and so on until z_n. A fast demonstration will show that the probability $P(z_i)$ that the i-th submitted pixel has to be written is equal to

$$\frac{1}{i}$$

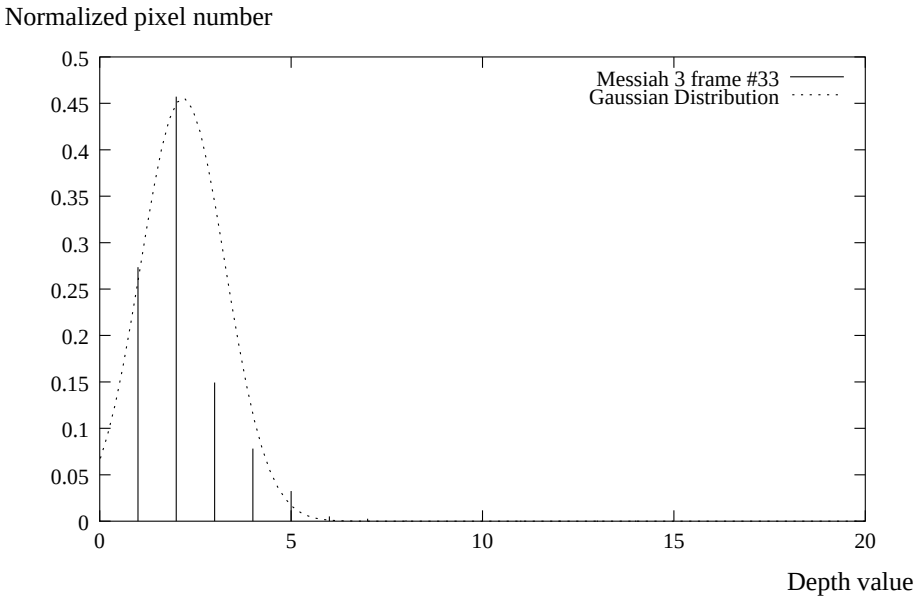


Figure A.14 : *Typical* depth distribution: Number of pixels = f(depth complexity).

Thus, the number of writings for a single pixel with depth complexity of n is

$$\sum_{k=1}^{n} \frac{1}{k}$$

and for the whole scene:

$$\sum_{k=1}^{\infty} f(k) \cdot \sum_{i=1}^{k} \frac{1}{i}$$

Table A.11 recapitulates and displays some elements in order to estimate the Z-buffer activity.

Conclusion

In conclusion, the following results are available:

- The mean depth complexity is **2.34**.
- At first approximation, a gaussian distribution of depth can be used.
- A typical frame will need **843,557** read accesses to the Z-buffer and **567,553** write accesses. A z value needs 4 bytes, so the required bandwidth is **96.54 MB/s** for reading and **64.95 MB/s** for writing.

[. . .]

Depth	Nb pixels $f(depth)$	Nb writings/pixel $\sum_{k=1}^{depth} \frac{1}{k}$	Readings $depth \cdot f(depth)$	Writings $f(depth) \cdot \sum_{k=1}^{depth} \frac{1}{k}$
1	79,539	1	79,539	79,539
2	138,290	3/2	276,580	207,435
3	105,693	11/6	317,079	193,771
4	35,509	25/12	142,036	73,977
5	5,244	137/60	26,220	11,974
6	340	49/20	2,040	833
7	9	363/140	63	24
Total	364,624	-	843,557	567,553

Table A.11: Z-buffer activity.

A.4 Mathematical add-in: Z-Buffer activity

In section A.3.1, the z-buffer activity is derived from the distribution of the depth complexity.

Equation (A.1) expressed the number of writings for a single pixel at a given depth complexity n: n pixels are then submitted before the output can be deduced.

$$\sum_{k=1}^{n} P(z_{\sigma(k)} \text{ must be written in the Z-buffer}) \tag{A.1}$$

where P is the probability and $z_{\sigma(i)}$ is the i-th submitted pixel and:

$$\begin{gathered} z_{\sigma(k)} \text{ must be written in the Z-buffer} \\ \Longleftrightarrow \\ z_{\sigma(k)} < \min(z_{\sigma(j)},\ 1 \leq j < i) \end{gathered} \tag{A.2}$$

Now, let us consider our n submitted pixels are ordered like in (A.3)

$$z_1 \leq z_2 \leq \cdots \leq z_n \tag{A.3}$$

The submission may be interpreted as a permutation. The $(i+1)$-th submitted pixel will be written in the following favorable cases:

- z_1 is submitted after i submissions from $\{z_k, k \in [2..n]\}$, $\binom{n-1}{i}$
- z_2 is submitted after i submissions from $\{z_k, k \in [3..n]\}$, $\binom{n-2}{i}$
- $\vdots$
- z_{n-i} is submitted after i submissions from $\{z_k, k \in [n-i+1..n]\}$, $\binom{i}{i}$

$$\begin{aligned} F &= \binom{i}{i} + \binom{i+1}{i} + \cdots + \binom{n-2}{i} + \binom{n-1}{i} \\ &= \sum_{j=i}^{n-1} \binom{j}{i} \end{aligned} \tag{A.4}$$

But

$$\begin{aligned} \binom{i}{i} &= 1 \\ &= \binom{i+1}{i+1} \end{aligned} \tag{A.5}$$

And

$$\binom{n}{p} + \binom{n}{p+1} = \binom{n+1}{p+1} \tag{A.6}$$

Replacing (A.5) and (A.6) into equation (A.4), the number of favorable cases is then:

$$\begin{aligned} F &= \binom{i}{i} + \binom{i+1}{i} + \cdots + \binom{n-2}{i} + \binom{n-1}{i} \\ &= \sum_{j=i}^{n-1} \binom{j}{i} \\ &= \underbrace{\binom{i+1}{i+1} + \binom{i+1}{i}}_{\binom{i+2}{i+1}} + \binom{i+2}{i} + \cdots + \binom{n-1}{i} \\ &= \quad \vdots \\ &= \binom{n}{i+1} \\ &= \frac{n!}{(i+1)!(n-i-1)!} \end{aligned} \tag{A.7}$$

The number of total cases is simply the number of possible pixels to submit after i first submissions:

$$T = (n-i)\binom{n}{i} \tag{A.8}$$

Then, from equations (A.7) and (A.8), the probability is:

$$\begin{aligned} P &= \frac{F}{T} \\ &= \frac{n!}{(i+1)!(n-i-1)!} \cdot \frac{i!(n-i)!}{(n-i)n!} \\ &= \frac{1}{i+1} \end{aligned} \tag{A.9}$$

Thus, the probability the $(i+1)$-th submitted pixel has to be written in the z-buffer is, from equation (A.9):

$$\frac{1}{i+1}$$

Then,

$$\sum_{k=1}^{n} P(z_{\sigma(k)} \text{ must be written in the Z-buffer}) = \sum_{k=1}^{n} \frac{1}{k} = \ln n + \gamma + o\left(\frac{1}{n}\right) \tag{A.10}$$

A.4.1 Triangle

Regarding the triangle size distribution (figure A.15), some data are remarkable although we limited the distribution for triangles covering less than 150 pixels. First, this distribution was taken arbitrarily from a frame in Vampire but generalization is possible. Here, 4,000 triangles were defined: average size was 255 pixels, 50% pixels covered less than 20 pixels and the largest triangle covered 39,200 pixels (13% of the screen). In general, the variance is very large: small triangles coexist with very large ones.

A.4.2 Tile

[...]

A.4.2.1 Overhead

Due to tiles, a frame is partitioned into independent regions. This partition may increase the number of triangles handled by the scene because a triangle does not often fit into a single tile. As a tile must be finish before rendering the next tile, a triangle overlapping 2 tiles will be partially rendered in the first tile and will have to wait for an undetermined time before its completion. Thus, parameters needed for interpolation and rasterization have to be stored or computed again for the second tile. In the last case,

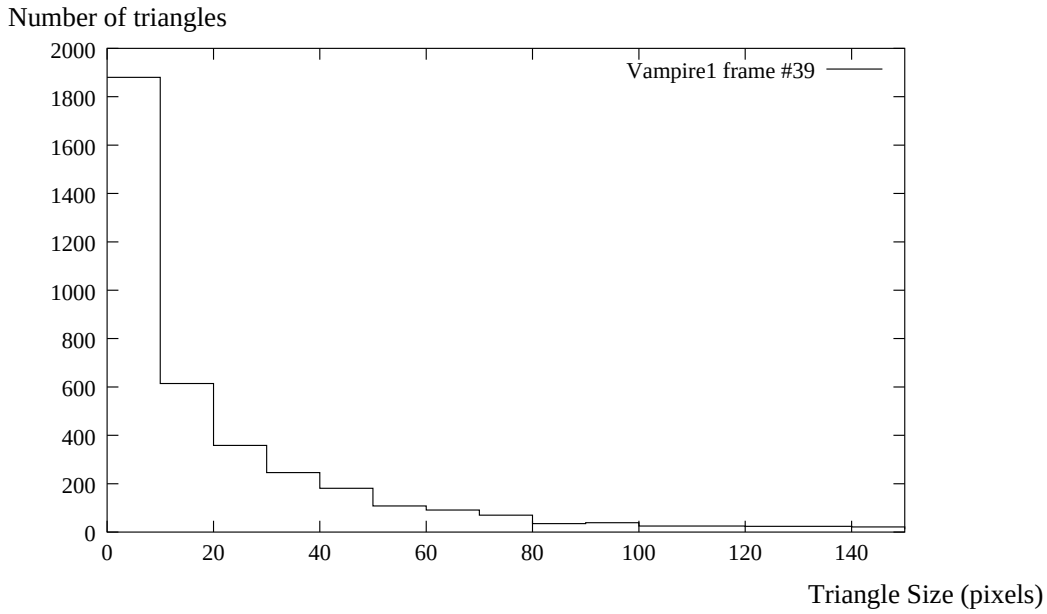


Figure A.15 : Triangle size distribution.

the setup process has to be reused several times for the same triangle, increasing the number of triangle in the scene. We call *Overhead Ratio* the ratio between the number of triangles taking into account the number of impacted tiles and the real number of triangles in the scene. For example, two triangles overlap 2 tiles for the first and 4 for the second, the overhead ratio is 3.

Figure A.16 shows the overhead ratio according to the number of triangles in the frame. These ratios will be reused in the later discussion about tile based rendering reliability. However, if a scene with 3,000 triangles for instance has an overhead ratio of 2, the actual number of triangles (6,000) is still less than the number of triangles in a complex scene (initially 10,000 triangles) despite a smaller ratio (1.7 in this case). Figure A.17 focuses more precisely on tile 32x32 (the largest tile tested). The table A.12 resumes ratios for several numbers of triangles. The overhead decreases when the tile area increases. It also decreases when the number of triangles increases: generally, triangles are then smaller and often fit in a single tile. At this point, overhead can be understood as the mean number of tile a triangle impacts.

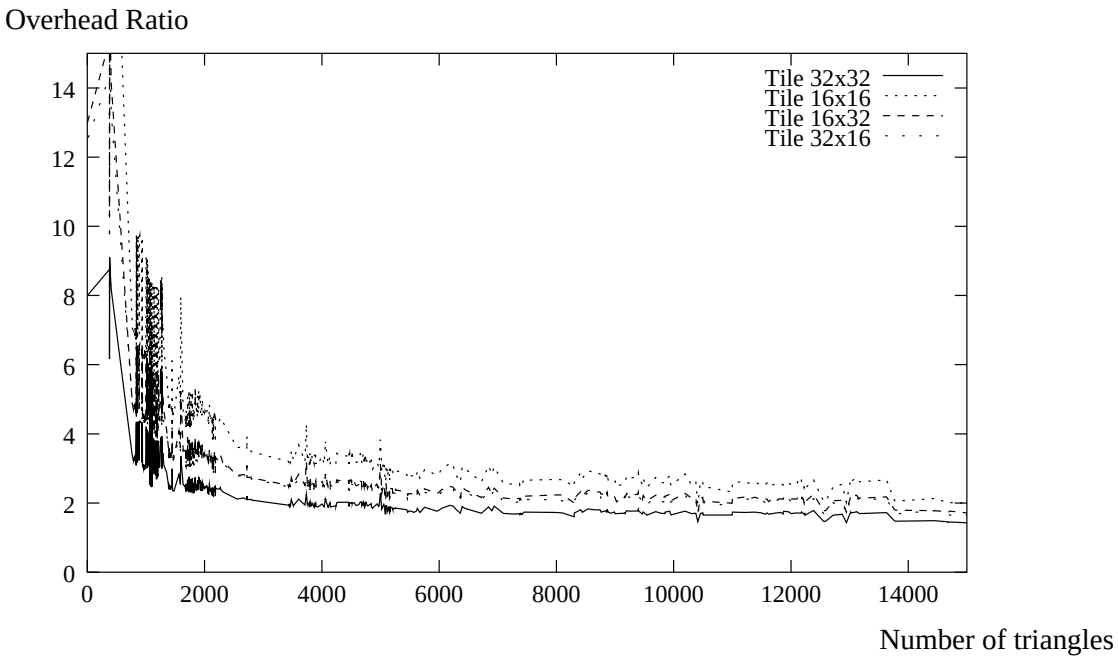


Figure A.16 : triangle overhead due to Tile Based Rendering.

[...]

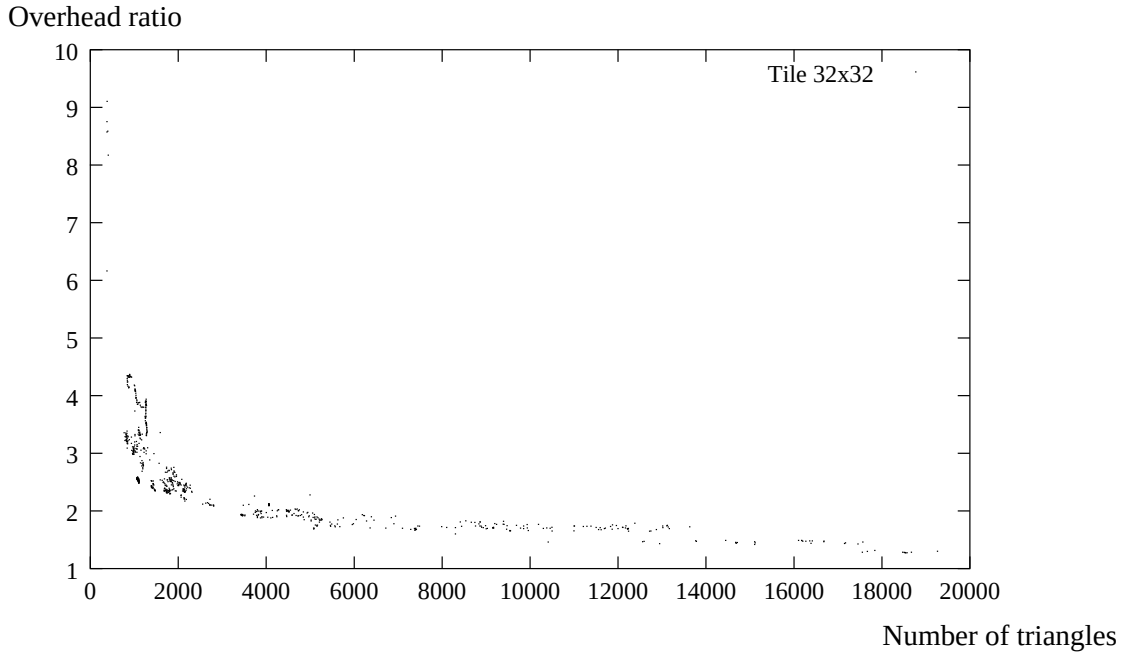


Figure A.17 : triangle overhead with tile 32x32.

Tiles	Number of triangles			
	3,000	5,000	10,000	15,000
Tile 32x32	2	1.89	1.7	1.42
Tile 32x16	2.6	2.37	2.12	1.66
Tile 16x32	2.6	2.43	2.17	1.69
Tile 16x16	3.34	3.04	2.67	1.99

Table A.12: Overhead in term of number of triangles.

Annexe B

Précision et dynamiques d'opérateurs

L'opérateur de transformation géométrique dont les étapes de traitements ont été présentées dans la partie 7.2.3 détermine pour chaque triangle traité les pixels de l'écran qu'il recouvre ainsi que la position de ces pixels dans la texture d'origine. Les spécifications de l'opérateur précise avec quelle définition ces calculs doivent être effectués.

Afin de dimensionner chaque bloc de traitement matériel de l'opérateur de transformation géométrique, nous avons mené une étude analytique des précisions et des dynamiques nécessaires lors des différentes étapes des traitements. Nous présentons maintenant cette étude tout d'abord dans un cadre général avant que nous n'effectuions l'application numérique dans le cas des spécifications du projet Tempovalse.

B.1 Modélisation

B.1.1 Hypothèses et notations

B.1.1.1 Hypothèses

Les calculs de précision seront menés dans le cas général en respectant les algorithmes de parcours de triangle développés. De plus, nous poserons les hypothèses suivantes :

- L'espace écran est carré et la longueur du côtée est une puissance de 2.
- Le rendu se faisant par tuile, les tuiles sont carrées et la longueur du côté est une puissance de 2.
- Nous pourrons négliger la taille des tuiles devant celle de l'image.
- L'espace texture est également carré et la longueur du côté est une puissance de 2.
- Un triangle est défini par la position de ces sommets dans l'écran et dans la texture. Ces sommets sont inclus dans ces espaces.

– Les sommets *Top, Mid* et *Bot* du triangles sont ordonnés dans l'espace écran. Ainsi, nous avons $0 \leq y_{\mathrm{T}} \leq y_{\mathrm{M}} \leq y_{\mathrm{B}}$ et $y_{\mathrm{T}} < y_{\mathrm{B}}$.
– La boîte entourante du triangle intersecte la tuile.
– Le point C représente le centre de la tuile.

B.1.1.2 Notations

Le tableau B.1 résume les différentes notations que nous utiliserons au cours de notre étude.

Paramètre	Description
e	Paramètre de la taille de l'écran : 2^e
t	Paramètre de la taille de la tuile : 2^t
m	Paramètre de la taille de la texture : 2^m
P_e	Paramètre de la précision dans l'écran : 2^{-P_e}
P_m	Paramètre de la précision dans la texture : 2^{-P_m}
$(x_{\mathrm{T}}, y_{\mathrm{T}})$	Coordonnées dans l'espace écran du sommet *Top* (indice $_M$ pour *Mid*, $_B$ pour *Bot*)
$(u_{\mathrm{T}}, v_{\mathrm{T}})$	Coordonnées dans l'espace texture du sommet *Top*
(x_1, y_1)	Coordonnées dans l'espace écran du sommet supérieur S_1 (ordonnée minimale) d'un côté de triangle
$x_{\mathrm{c}}^{\mathrm{i}}$	Abcisse de l'intersection d'un côté avec la ligne centrale de la tuile d'ordonnée y_{c}
P	Pente du côté du triangle
M	Mantisse d'un calcul en virgule flottante : $M \in [1; 2[$

Tab. B.1 – Les différentes notations utilisées.

La figure B.1 illustre nos hypothèses et ces paramètres.

Enfin, nous noterons les différentes représentations de nombres de la manière suivante. Dans le cas de nombre en représentation entières à virgules fixes, nous indiquerons le nombre de bits des parties décimales (précédé de la lettre S dans le cas où un bit de signe est employé) suivi du nombre de bits de la partie fractionnaire. Pour plus de clarté, un point séparera ces indications. Dans le cas de représention flottantes des nombres, nous indiquerons de la même manière le nombre de bits employés pour les mantisses et exposants.

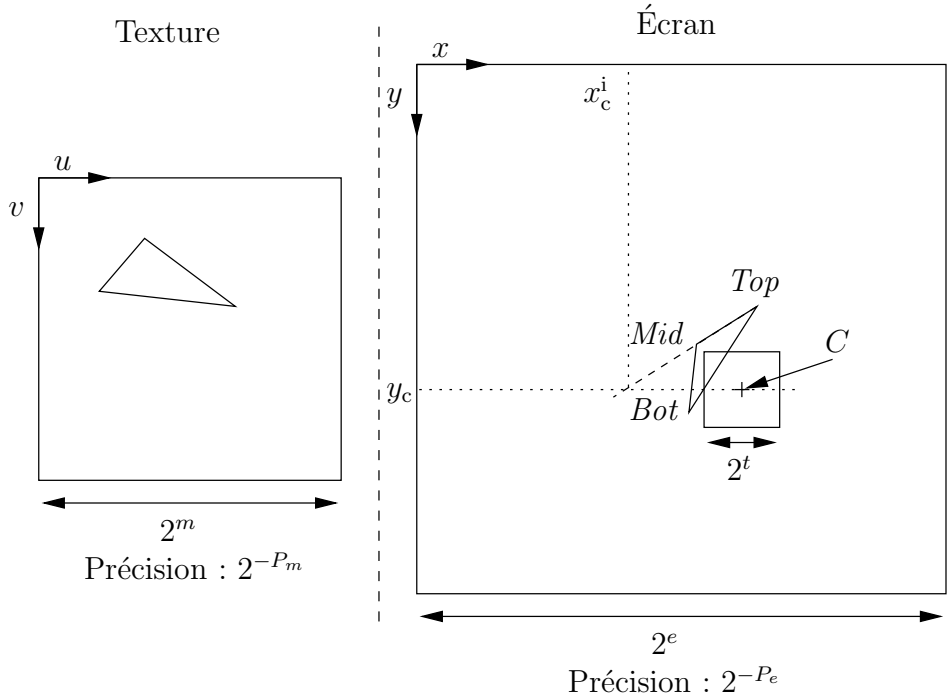


FIG. B.1 – Illustration des hypothèses et des différents paramètres.

B.1.2 Études analytiques

B.1.2.1 Parcours des triangles

Le parcours des triangles nécessite de calculer l'abscisse x de l'intersection entre chaque côté du triangle et la ligne courante de la tuile d'ordonnée y. Ce calcul s'effectue par l'intermédiaire de l'intersection avec la ligne centrale de la tuile d'ordonnée y_c et de la pente P du côté considéré (éq. (B.1)). Les inconnues ici concernent x_c^i et la pente P. Afin d'obtenir une précision $\epsilon_x \leq 2^{-P_e}$ sur le résultat final, nous partagerons l'erreur entre les termes x_c^i et $(y - y_c) \cdot P$. Ainsi, chacun de ces deux termes sera calculé avec une erreur $\epsilon \leq 2^{-P_e-1}$. L'équation (B.1) rappelle le calcul effectué ainsi que ce partage d'erreur.

$$\underbrace{x}_{\epsilon_x \leq 2^{-P_e}} = \underbrace{x_c^i}_{\epsilon \leq 2^{-P_e-1}} + \underbrace{(y - y_c) \cdot P}_{\epsilon \leq 2^{-P_e-1}} \tag{B.1}$$

Dynamique de la pente

Celle-ci est déterminée par la pente maximale qui se calcule par l'équation (B.2).

$$\begin{aligned} P_{\max} &= \frac{\Delta x_{\max}}{\Delta y_{\min}} \\ &= \frac{2^e}{2^{-P_e+1}} \\ &= 2^{e+P_e-1} \end{aligned} \tag{B.2}$$

Le terme $(y - y_\mathbf{c}) \cdot P$

Nous avons $\epsilon\,|(y - y_\mathrm{c}) \cdot P| = |y - y_\mathrm{c}| \cdot \epsilon_p$, ainsi nous déduisons qu'il suffit de calculer (avec $y \neq y_\mathrm{c}$) :

$$\begin{aligned} |y - y_\mathrm{c}| \cdot \epsilon_p &\leq 2^{-P_e-1} \\ \iff \epsilon_p &\leq \frac{2^{-P_e-1}}{|y - y_\mathrm{c}|} \end{aligned} \tag{B.3}$$

Comme y_c représente l'ordonnée de la ligne centrale de la tuile courante et y une ordonnée d'une ligne de la tuile, il vient $|y - y_\mathrm{c}| \leq 2^{t-1}$. Ainsi, il suffit alors de connaitre la pente avec une précision donnée par l'équation (B.4).

$$\epsilon_p \leq 2^{-P_e-t} \tag{B.4}$$

Le terme $x^\mathbf{i}_\mathbf{c}$

Précision L'équation (B.5) nous rappelle l'expression de x^i_c.

$$x^\mathrm{i}_\mathrm{c} = x_1 + (y_\mathrm{c} - y_1) \cdot P \tag{B.5}$$

L'erreur sur x_1 étant nulle, l'erreur sur x^i_c revient à une imprécision sur le calcul $(y_\mathrm{c} - y_1) \cdot P$. Il suffit de calculer

$$\epsilon\,|(y_\mathrm{c} - y_1) \cdot P| \leq 2^{-P_e-1} \tag{B.6}$$

De manière analogue à l'équation (B.3), il vient

$$\epsilon_p \leq \frac{2^{-P_e-1}}{|y_\mathrm{c} - y_1|} \tag{B.7}$$

Or, nous pouvons écrire $|y_\mathrm{c} - y_1|$ inférieur à $2^e - 2^{t-1}$. Par hypothèses, nous pouvons négliger la taille de la tuile devant la taille de l'image. Ainsi, nous obtenons, pour un calcul de x^i_c avec une precision de $\epsilon_{x^\mathrm{i}_\mathrm{c}}$ inférieur à 2^{-P_e-1}, une précision nécessaire sur la pente donnée par l'équation (B.8).

$$\epsilon_p \leq 2^{-P_e-e-1} \tag{B.8}$$

Dynamique x_c^i intervient comme point intermédiaire dans les traitements. Pour calculer la dynamique de ce point, nous pouvons ajouter comme hypothèse que nous ne traitons pas les côtés totalement au dessus ou au dessous de la tuile. Nous avons donc pour un côté $[S_1S_2]$:

$$y_1 < y_2 \tag{B.9}$$
$$y_c - 2^{t-1} \leq y_2 \tag{B.10}$$
$$y_1 < y_c + 2^{t-1} - 1 \tag{B.11}$$

Ainsi, nous avons $2^{t-1} - 1 < y_c - y_1 \leq 2^{t-1} + (y_2 - y_1)$ et nous en déduisons, sachant que $x_c^i = x_1 + (y_c - y_1) \cdot P$:

$$\begin{aligned} |x_c^i| &\leq |x_1| + |(y_c - y_1) \cdot P| \\ &\leq |x_1| + \left|\left(2^{t-1} + (y_2 - y_1)\right) \cdot P\right| \\ &\leq |x_1| + 2^{t-1} \cdot |P| + |x_2 - x_1| \\ &\leq x_{\max} + 2^{t-1} \cdot P_{\max} + \Delta_x^{\max} \\ &\leq 2^e + 2^{(t-1)+(e+P_e-1)} + 2^e \\ &\leq 2^{e+1} + 2^{e+t+P_e-2} \end{aligned} \tag{B.12}$$

Calcul en flottant de la pente

Nous avons vu que le terme P intervient à la fois pour le calcul de x_c^i et dans le calcul de la position courante x avec une précision différente en virgule fixe. Nous nous intéressons ici à la précision nécessaire pour un calcul en virgule flottante de ce facteur. Pour cela, nous revenons au calcul direct de $x = x_1 + (y - y_1) \cdot P$ d'où nous tirons

$$y - y_1 = \frac{(x - x_1)}{P} \tag{B.13}$$

Or, en flottant, nous avons l'égalité

$$\frac{\epsilon_p}{P} = \frac{\epsilon_M}{M} \tag{B.14}$$

Ainsi, il vient

$$\begin{aligned} (x - x_1) \cdot \frac{\epsilon_p}{P} &= (x - x_1) \frac{\epsilon_M}{M} \\ (y - y_1) \cdot \epsilon_p &= (x - x_1) \frac{\epsilon_M}{M} \end{aligned} \tag{B.15}$$

Comme $\epsilon_x = (y - y_1) \cdot \epsilon_p$, nous obtenons pour $\epsilon_x \leq 2^{-P_e-1}$

$$\begin{aligned} |x - x_1| \frac{\epsilon_M}{|M|} &\leq 2^{-P_e-1} \\ \iff \epsilon_M &\leq \frac{2^{-P_e-1} |M|}{|x - x_1|} \end{aligned} \tag{B.16}$$

Avec $M \in [1;2[$ (par hypothèses) et $(x, x_1) \in \left[0; 2^e - 2^{-P_e}\right]^2$, nous obtenons la précision sur la mantisse donnée par léquation (B.17).

$$\epsilon_M \leq 2^{-e-P_e-1} \tag{B.17}$$

Nous pouvons faire les deux remarques suivantes. La précision sur la mantisse est ainsi identique à la précision de la pente calculée par l'équation (B.8). Ainsi, la précision de la pente en représentation flottante sera plus importante (ou égale) dès lors que cette pente P sera inférieur à 2. La seconde remarque concerne l'utilisation de cette pente en représentation flottante. Cette pente garantit une précision de calcul de 2^{-P_e-1} pour tout point d'abscisse compris dans l'espace écran. La précision de la mantisse étant suffisante, il est possible d'utiliser cette pente en représentation en virgule fixe dans l'équation (B.1) si l'abscisse x_c^i appartient à l'espace écran. Dans le cas contraire, le passage en virgule fixe qui fera perdre de la précision sur la mantisse est sujet à plus ample discussion.

Ainsi, pour les pentes de 2 à $2^{(e+P_e+1)-(P_e+t)}$ (en valeur absolue), le calcul en virgule fixe par un point intermédiaire (lui-même calculé par la pente en flottant) avec les précisions précédemment données n'est plus justifié.

Or pour une telle pente, en perdant b bits de précision ($b \in [1, e-t+1]$) lors du passage de représentation flottante en représentation à virgule fixe, nous avons $|P|$ appartenant à l'ensemble $[2^b; 2^{b+1}[$.

Donc, nous pouvons en déduire les deux équations (B.18)

$$\begin{aligned} & x = x_1 + (y - y_1)P \\ \iff\ & x = x_1 + (y_c - y_1)P + (y - y_c)P \end{aligned} \tag{B.18}$$

et (B.19).

$$(y - y_1) = \frac{x - x_1}{P} \tag{B.19}$$

Or, $(x, x_1) \in [0; 2^e]^2$ et $|P| \in [2^b; 2^{b+1}[$ implique que $|y - y_1| \in [0; 2^{e-b}]$. Nous pouvons remarquer ici qu'il suffit simplement que $|x - x_c^i|$ soit inférieur à 2^e sans pour autant que x_c^i appartienne à l'espace écran pour que les calculs précédents soient valables et assurent un résultat correct. Malheureusement, $|y_c - y_1| \in [0; 2^{e-b} + 2^{t-1}]$ et lorsque $|y_c - y_1| \in [2^{e-b}; 2^{e-b} + 2^{t-1}]$, $|x - x_c^i|$ est supérieur à 2^e.

Dans ce dernier cas, nous pouvons cependant recalculer la précision $\epsilon_{x_c^i}$ du résultat intermédiaire (équation (B.20)). Nous apercevons alors qu'il ne manque qu'un seul bit de précision pour un retour à la précision de l'équation (B.1).

$$\begin{aligned} \epsilon_{x_c^i} &= |y_c - y_1|\epsilon \\ &\leq (2^{e-b} + 2^{t-1})2^{b-e-P_e-1} \\ &\leq 2^{-P_e-1} + 2^{b+(t-e-1)-P_e-1} \\ &\leq 2^{-P_e} \end{aligned} \tag{B.20}$$

B.1.2.2 Transformation affine

Les coordonnées de l'espace texture sont déduites des coordonnées de l'espace écran par une transformation affine que résume l'équation (B.21) où apparait également la répartition des imprécisions selon les termes de l'équations. Les inconnues sont ici les coordonnées (u_c, v_c) et les quatres coefficients affines.

$$\begin{cases} u = u_c + \frac{\partial u}{\partial x}(x - x_c) + \frac{\partial v}{\partial x}(y - y_c) \\ \underbrace{v}_{\epsilon \leq 2^{-P_m}} = \underbrace{v_c}_{\epsilon \leq 2^{-P_m - 1}} + \underbrace{\frac{\partial u}{\partial y}(x - x_c) + \frac{\partial v}{\partial y}(y - y_c)}_{\epsilon \leq 2^{-P_m - 1}} \end{cases} \tag{B.21}$$

Dynamique des coefficients affines

Les expressions étant symétriques en x, y, u et v, nous nous intéresserons, sans perte de généralité, au calcul de $\frac{\partial u}{\partial x}$ dans ce paragraphe. Nous avons ainsi

$$\begin{aligned} \frac{\partial u}{\partial x} &= \frac{\mathrm{Pvux}}{\mathrm{Pv}} \\ &= \frac{\Delta u_{\mathrm{TB}} \Delta y_{\mathrm{MB}} - \Delta u_{\mathrm{MB}} \Delta y_{\mathrm{TB}}}{\Delta x_{\mathrm{TB}} \Delta y_{\mathrm{MB}} - \Delta x_{\mathrm{MB}} \Delta y_{\mathrm{TB}}} \end{aligned} \tag{B.22}$$

Pour |Pv| non nul, nous pouvons minorer le dénominateur par $2^{-2(P_e-1)}$. Concernant le numérateur, nous avons

$$\begin{aligned} \mathrm{Pvux} &= \Delta u_{\mathrm{TB}} \cdot \Delta y_{\mathrm{MB}} - \Delta u_{\mathrm{MB}} \cdot \Delta y_{\mathrm{TB}} \\ &= (u_{\mathrm{B}} - u_{\mathrm{T}}) \cdot \Delta y_{\mathrm{MB}} - (u_{\mathrm{B}} - u_{\mathrm{M}}) \cdot \Delta y_{\mathrm{TB}} \\ &= (u_{\mathrm{B}} - u_{\mathrm{M}} + u_{\mathrm{M}} - u_{\mathrm{T}}) \cdot \Delta y_{\mathrm{MB}} - (u_{\mathrm{B}} - u_{\mathrm{M}}) \cdot \Delta y_{\mathrm{TB}} \\ &= (u_{\mathrm{B}} - u_{\mathrm{M}}) \cdot (\Delta y_{\mathrm{MB}} - \Delta y_{\mathrm{TB}}) + (u_{\mathrm{M}} - u_{\mathrm{T}}) \cdot \Delta y_{\mathrm{MB}} \end{aligned} \tag{B.23}$$

Or, nous avons $0 \leq \Delta y_{\mathrm{MB}} \leq \Delta y_{\mathrm{TB}}$. Ainsi, $|\Delta y_{\mathrm{MB}} - \Delta y_{\mathrm{TB}}| = \Delta y_{\mathrm{TB}} - \Delta y_{\mathrm{MB}}$ et nous pouvons en déduire que

$$\begin{aligned} |\mathrm{Pvux}| &\leq \Delta_{u_{\max}} \cdot (\Delta y_{\mathrm{MB}} - \Delta y_{\mathrm{TB}} + \Delta y_{\mathrm{MB}}) \\ &\leq 2^{m+e} \end{aligned} \tag{B.24}$$

En conclusion, nous obtenons l'inégalité (B.25) où l'égalité peut être réalisée.

$$\left| \frac{\partial u}{\partial x} \right|_{\max} \leq 2^{m+e+2P_e} \tag{B.25}$$

Précision des données

Le terme $\frac{\partial u}{\partial x}(x - x_{\mathbf{c}}) + \frac{\partial v}{\partial x}(y - y_{\mathbf{c}})$ De manière similaire à celle utilisée pour le parcours des triangles, la précision de 2^{-P_m} pour les calculs de l'équation (B.21) est obtenue en calculant séparément u_{c} et $\frac{\partial u}{\partial x}(x - x_{\mathrm{c}}) + \frac{\partial v}{\partial x}(y - y_{\mathrm{c}})$ avec une précision de 2^{-P_m-1} (cf. équation (B.21)).

Ainsi,

$$\begin{aligned} \epsilon \left| \frac{\partial u}{\partial x}(x - x_{\mathrm{c}}) + \frac{\partial v}{\partial x}(y - y_{\mathrm{c}}) \right| &\leq 2^{-P_m-1} \\ \epsilon_{\partial} &\leq \frac{2^{-P_m-1}}{|x - x_{\mathrm{c}}| + |y - y_{\mathrm{c}}|} \end{aligned} \tag{B.26}$$

Or, comme x et y appartiennent à la tuile dont le point central est le point de coordonnées $(x_{\mathrm{c}}, y_{\mathrm{c}})$, nous obtenons

$$\begin{aligned} \epsilon_{\partial} &\leq \frac{2^{-P_m-1}}{2^{t-1} + 2^{t-1}} \\ \epsilon_{\partial} &\leq 2^{-P_m-t-1} \end{aligned} \tag{B.27}$$

Le terme $u_{\mathbf{c}}$ Comme $u_{\mathrm{c}} = u_{\mathrm{T}} + \frac{\partial u}{\partial x}(x_{\mathrm{c}} - x_{\mathrm{T}}) + \frac{\partial v}{\partial x}(y_{\mathrm{c}} - y_{\mathrm{T}})$, la précision du calcul de u_{c} est déterminée par la précision de $\frac{\partial u}{\partial x}(x_{\mathrm{c}} - x_{\mathrm{T}}) + \frac{\partial v}{\partial x}(y_{\mathrm{c}} - y_{\mathrm{T}})$ Ainsi,

$$\begin{aligned} \epsilon \left| \frac{\partial u}{\partial x}(x_{\mathrm{c}} - x_{\mathrm{T}}) + \frac{\partial v}{\partial x}(y_{\mathrm{c}} - y_{\mathrm{T}}) \right| &\leq 2^{-P_m-1} \\ \epsilon_{\partial} &\leq \frac{2^{-P_m-1}}{2^e + 2^e} \\ \epsilon_{\partial} &\leq 2^{-P_m-e-2} \end{aligned} \tag{B.28}$$

B.1.2.3 Optimisation des dynamiques des coefficients affines

Compte tenu de la dynamique précédemment calculée pour les coefficients affines (éq. (B.25)), la dynamique de u_{c} et v_{c} est également importante. Cependant, lorsqu'un calcul de coordonnées de l'espace texture sera effectué, nous pouvons garantir que le résultat sera dans cet espace. Il devient alors possible d'effectuer les calculs intermédiaires modulo la taille de cet espace. En particulier, nous avons

$$\begin{aligned} u &= u\,[2^m] \\ &= \left(u_{\mathrm{c}} + \frac{\partial u}{\partial x}(x - x_{\mathrm{c}}) + \frac{\partial v}{\partial x}(y - y_{\mathrm{c}}) \right) [2^m] \\ &= (u_{\mathrm{c}}\,[2^m]) + \left(\frac{\partial u}{\partial x}\,[2^m] \right) \cdot (x - x_{\mathrm{c}}) + \left(\frac{\partial v}{\partial x}\,[2^m] \right) \cdot (y - y_{\mathrm{c}}) \end{aligned} \tag{B.29}$$

Cependant, le nouvel espace modulo présente la particularité que $0 = 2^m - \epsilon\,[2^m]$ pour n'importe quelle précision ϵ. Il convient alors d'effectuer ces calculs avec une marge de sécurité rendant impossible cette égalité : les calculs seront effectués modulo 2^{m+1}.

B.2 Application au contexte du projet Tempovalse

B.2.1 Détermination des tailles des données

Le tableau B.2 exprime les valeurs des différents paramètres utilisés lors du projet Tempovalse. Le format des images étant du QVGA, l'espace écran est représenté par un carré de largeur $2^9 = 512$. L'espace texture correspond aux mémoires locales de l'opérateur où est chargée une partie de la texture globale. Cette espace est également élargit à un carré de largeur $2^6 = 64$.

Paramètre	valeur
e	9
t	4
m	6
P_e	5
P_m	4

TAB. B.2 – Les valeurs des paramètres dans le contexte Tempovalse.

Le tableau B.3 résume les résultats et leur application numérique : afin de simplifier une implantation matérielle, les arrondis sont systématiquement calculés par « troncature » ce qui revient à utiliser un nombre de bits pour la partie fractionnaire égal à l'opposé de l'exposant de la précision.

B.2.2 Organisation des traitements

B.2.2.1 Parcours des triangles

L'équation (B.30) reprend les différentes étapes de calculs effectuées pour le parcours des triangles.

$$x = \underbrace{[x_\mathrm{T} + (y_\mathrm{c} - y_\mathrm{T}) \cdot P]}_{x_\mathrm{c}^\mathrm{i}} + (y - y_\mathrm{c}) \cdot P \tag{B.30}$$

En raison de la simplicité d'implantation, la pente qui requiert une division sera donc calculée en flottant $\mathbf{S_M 1.15} \cdot \mathbf{2^{S_e 4}}$ où S_M et S_e représentent des bits de signe.

Cette pente sert au calcul de x_c^i qui sera fourni en virgule fixe **S17.6**. La discussion que nous avons menée paragraphe B.1.2.1 montre toutefois que dans certains cas la précision spécifiée ne sera pas tenue en effectuant les calculs suivants avec une pente en virgule fixe au format $S14.9$. La précision de x_c^i peut cependant être améliorer à $\epsilon_{x_c^i} \leq \frac{3}{2}2^{-P_e-1}$ (tiré de l'équation (B.20)) en concervant un bit supplémentaire sur la pente. Cette précision sera alors jugée acceptable. Ainsi, la valeur de la pente sera restituée en virgule fixe dans le format **S14.10**.

B.2.2.2 Transformation affine

L'équation (B.31) reprend les différentes étapes de calcul effectuées pour les coefficients de transformation affine.

$$u = \underbrace{\left(\left[x_T + \frac{\partial u}{\partial x}(x_c - x_T)\right] + \frac{\partial v}{\partial x}(y_c - y_T)\right)}_{u_c} + \frac{\partial u}{\partial x}(x - x_c) + \frac{\partial v}{\partial x}(y - y_c) \quad \text{(B.31)}$$

L'organisation des traitements est similaire à celle présentée plus haut. Bien que nos investigations ne nous aient pas permis de démontrer de résultats satisfaisant pour des calculs en flottant, des simulations tendent à indiquer qu'une mantisse de $\mathbf{S_M1.28}$ bits serait suffisante pour la précision souhaitée au final. Les coefficients sont donc calculés par division flottante en $\mathbf{S_M1.28 \cdot 2^{S_e5}}$.

Ces coefficients servent au calcul de u_c et v_c qui seront fournis en virgule fixe en **7.5** après calcul modulo. Quant aux coefficients, ils sont restitués en virgule fixe dans le format **7.10**, un bit supplémentaire étant ajouté pour les mêmes raisons évoquées au paragraphe précédent.

B.3 Conclusion

Nous avons effectué un calcul analytique de la taille nécessaire pour nos chemins de données afin d'obtenir la précision souhaitée. Nous avons simulé les traitements décrits en incluant les opérations précises au bit près et en testant à la fois des cas pathologiques et quelques millions de configurations de triangles et transformées affines tirées aléatoirement dans un espace 512×512 pour l'image et 64×64 pour la texture. Les résultats obtenus confirment les calculs présentés précédemment.

Description	**Valeur analytique**		**Application Numérique**
Dynamique de la pente maximale	$P_{\max} = 2^{e+P_e-1}$	éq. (B.2)	14 bits
Précision de la pente pour le calcul de x à partir de $x_{\mathrm{c}}^{\mathrm{i}}$	$\epsilon_p \leq 2^{-P_e-t}$	éq. (B.4)	9 bits
Précision de la pente pour le calcul de x_{c}	$\epsilon_p \leq 2^{-P_e-e-1}$	éq. (B.8)	15 bits
Précision de $x_{\mathrm{c}}^{\mathrm{i}}$	$\epsilon \leq 2^{-P_e-1}$	éq. (B.1)	6 bits
Dynamique de $x_{\mathrm{c}}^{\mathrm{i}}$	$\left\lvert x_{\mathrm{c}}^{\mathrm{i}} \right\rvert \leq 2^{e+1} + 2^{e+t+P_e-2}$	éq. (B.12)	17 bits
Précision de la mantisse de la pente pour un calcul en flottant de x	$\epsilon_M \leq 2^{-e-P_e-1}$	éq. (B.17)	15 bits
Dynamique de l'exposant de la pente pour un calcul en flottant de x	$\log_2(e + P_e - 1)$	cf. éq. (B.2)	4 bits
Dynamique des coefficients affines	$\left\lvert \frac{\partial u}{\partial x} \right\rvert_{\max} = 2^{m+e+2\cdots(P_e-1)}$	éq. (B.25)	24 bits
Précision des coefficients affines pour le calcul de u à partir de u_{c}	$\epsilon_\partial \leq 2^{-P_m-t-1}$	éq. (B.27)	9 bits
Précision des coefficients affines pour le calcul de u_{c}	$\epsilon_\partial \leq 2^{-P_m-e-2}$	éq. (B.28)	15 bits
Précision de u_{c}	$\epsilon \leq 2^{-P_m-1}$	cf. éq. (B.21)	5 bits
Valeur du modulo	2^{m+1}		7 bits

TAB. B.3 – Résumé des précisions nécessaires et application numérique.

Annexe C

Opérateur de transformation géométrique

Cette annexe détaille l'opérateur dédié de transformation géométrique pour le rendu de maillages de triangles GTM.

C.1 Vue d'ensemble

La figure C.1 illustre la vue en bloc globale de l'opérateur GTM. Développé comme extension d'un précédent opérateur de transformation géométrique, GTM conserve l'ancienne partie opérative nécessaire pour le rendu d'objet autre que les maillages. Nous avons dû modifier le contrôleur d'origine en rajoutant les états nécessaires aux traitements d'un maillage. Nous avons finalement développé les blocs dénommés PROC_X, PROC_Y, DMA, Init ainsi que trois blocs mémoire. Un multiplexeur en sortie du bloc sélectionne entre les sorties de la partie opérative originale et les sorties du nouveau bloc PROC_Y.

C.2 Détail des blocs développés

C.2.1 Le bloc Init

Le bloc d'initialisation est composé de trois parties. Un contrôleur qui indique à chaque élément l'étape courante d'initialisation d'un triangle. La seconde partie correspond à l'interface avec la mémoire d'initialisation qui se charge d'accéder aux sommets en fonction de la définition du triangle et de trier ces sommets selon les ordonnées croissantes. La dernière partie regroupe les opérateurs arithmétiques nécessaires aux calculs d'initialisation. Un banc de registre contient les différents résultats intermédiaires qui sont connectés aux deux soustracteurs, au multiplieur-accumulateur et au trois opérateurs spécifiques de division-multiplication-accumulation : deux pour les coefficients affines, un pour les coefficients parcours des triangles. Ce bloc s'interconnecte

avec la mémoire affine et celle de parcours où sont mémorisés les résultats de ces trois opérateurs spécifiques.

La figure C.2 illustre l'organisation interne de ce bloc.

C.2.2 Le bloc PROC_Y

Le bloc PROC_Y est en charge du traitement « Balayage » : il identifie l'intersection d'un triangle avec une ligne de la tuile. Ce bloc reçoit des informations venant du contrôleur et s'interface avec les mémoires de coefficients de parcours des triangles (en entrée) et de coefficients de transformation affine (en sortie) dont il pilote la lecture pour le bloc PROC_X à l'aide de sa table d'index (double registre).

La figure C.3 illustre l'organisation interne de ce bloc.

C.2.3 Le bloc PROC_X

Le bloc PROC_X est en charge du traitement du processus « Balayage ». Outre des signaux de contrôle indiquant le numéro de ligne courante ainsi que le pixel courant, ce bloc s'interface avec la mémoire de coefficients affines pour calculer la coordonnée de texture correspondante au pixel courant.

La figure C.4 illustre l'organisation interne de ce bloc.

C.2.4 Les mémoires

Notre opérateur embarque trois mémoires différentes. La mémoire d'initialisation est la première d'entre elles. Après une requête du DMA auprès de la mémoire centrale du système, les transferts s'effectuent en direction de cette première mémoire qui contient donc une copie locale de l'ensemble de triangles couramment traité.

Les deux autres mémoires sont mises à jour par le processus d'initialisation à l'aide des coefficients de transformation affine et de parcours de triangle. Ces mémoires sont organisées en double tampon à l'aide de mémoires DPRAM.

Les figures C.5, C.6 et C.7 illustrent les organisation internes des blocs respectifs de mémoire d'initialisation, de mémoire de coefficients affines et de coefficients pour le parcours de triangles. La figure C.5 illustre l'organisation interne de ce bloc.

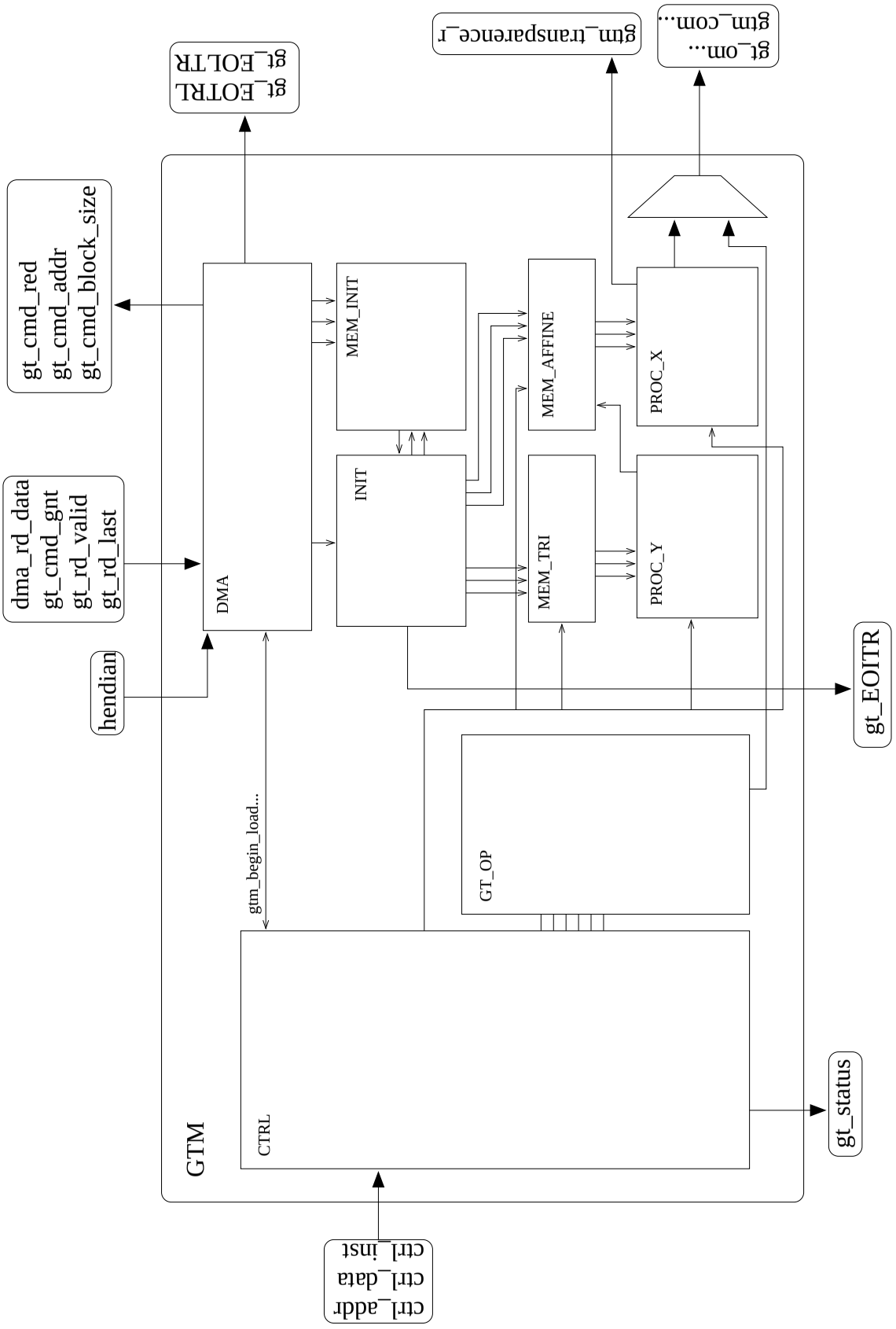


FIG. C.1 – Vue d'ensemble de l'opérateur dédié de transformation géométrique pour maillages de triangles.

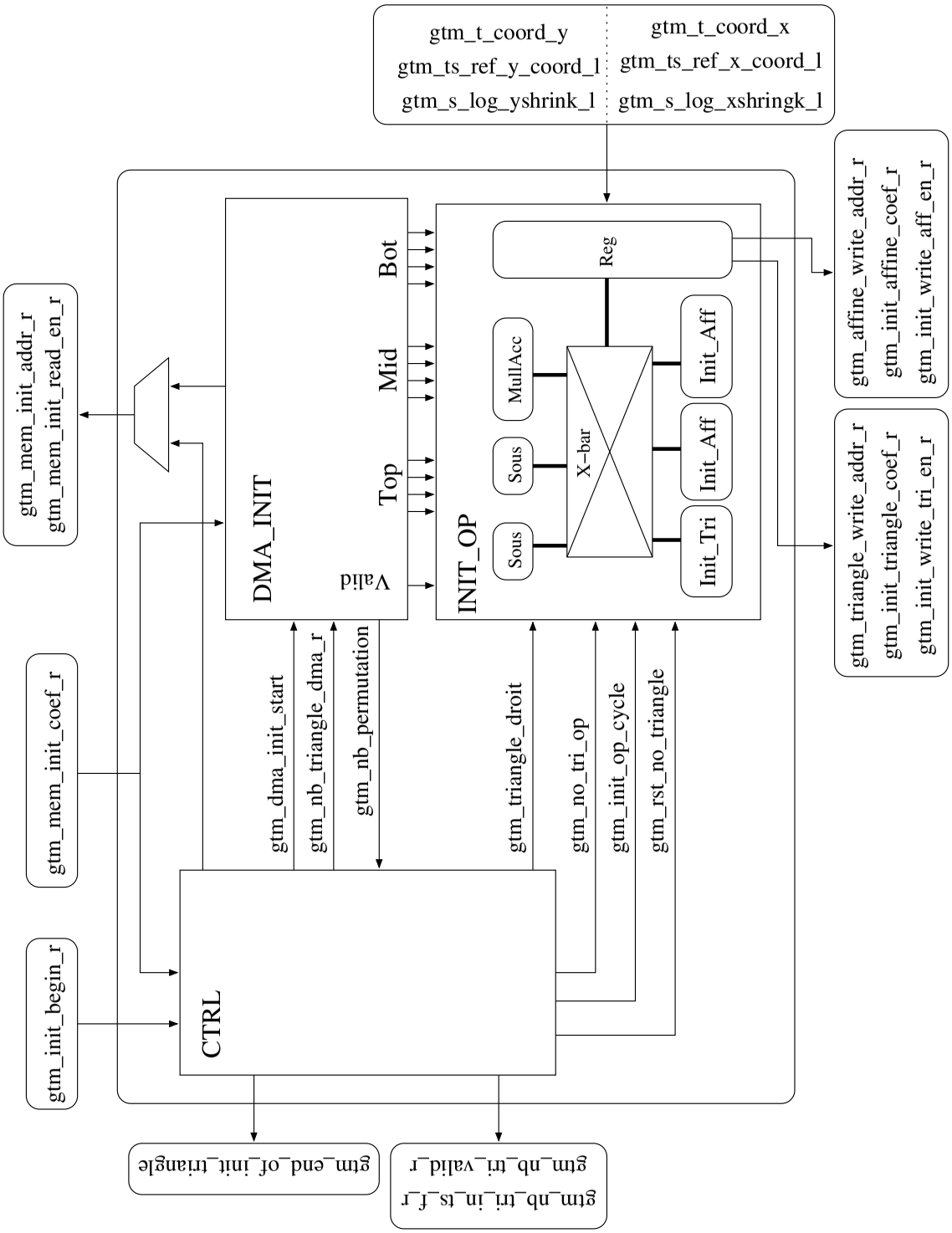


FIG. C.2 – Diagramme interne du bloc d'initialisation.

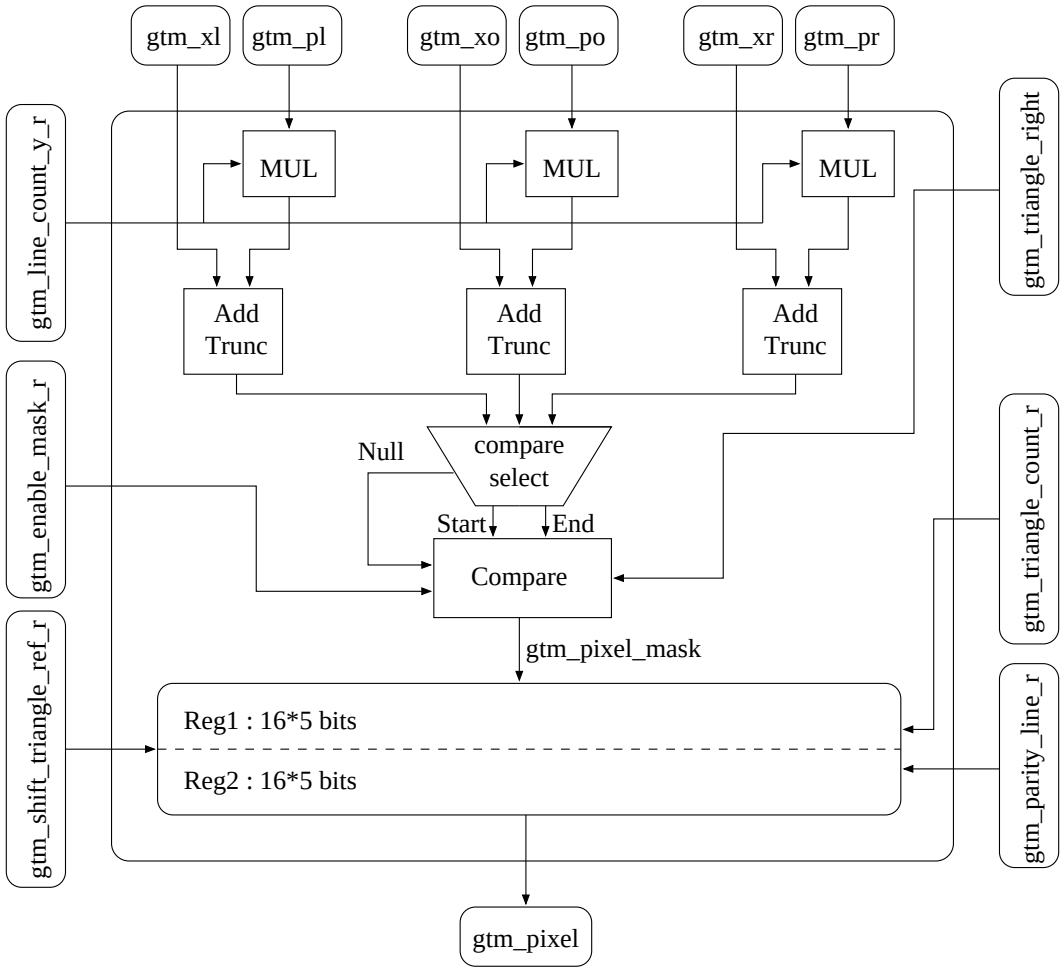


Fig. C.3 – Diagramme interne du bloc PROC_Y.

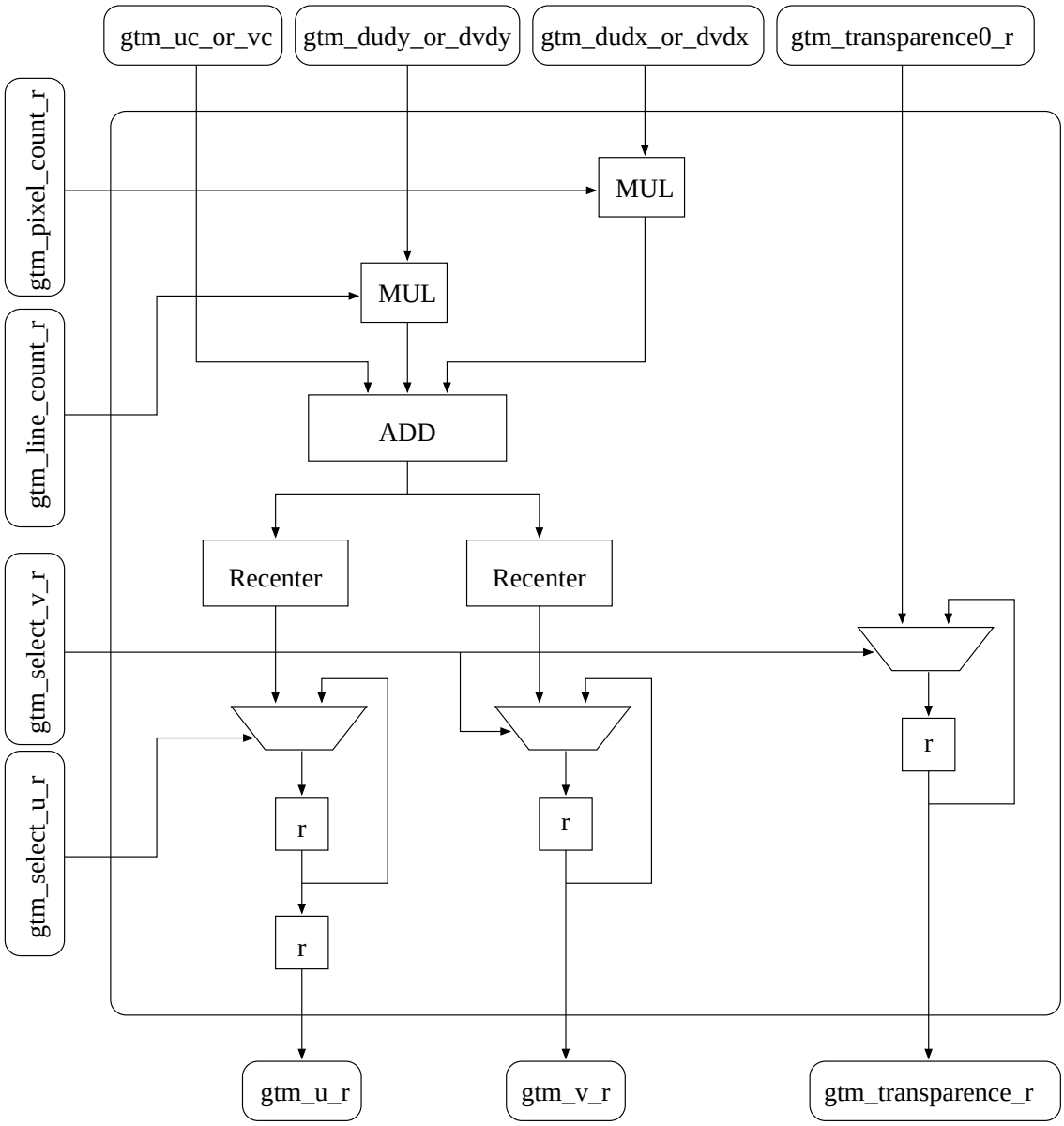


FIG. C.4 – Diagramme interne du bloc PROC_X.

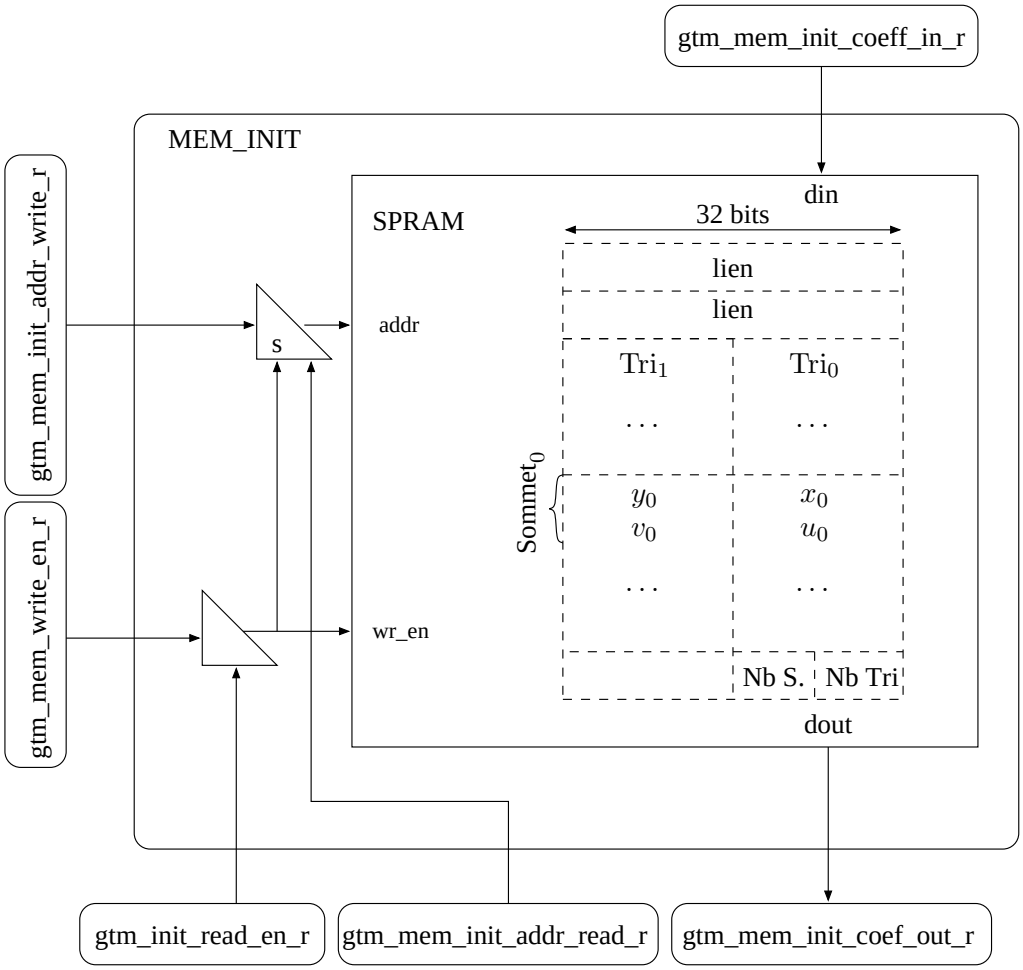


FIG. C.5 – Mémoire d'initialisation et logique de contrôle.

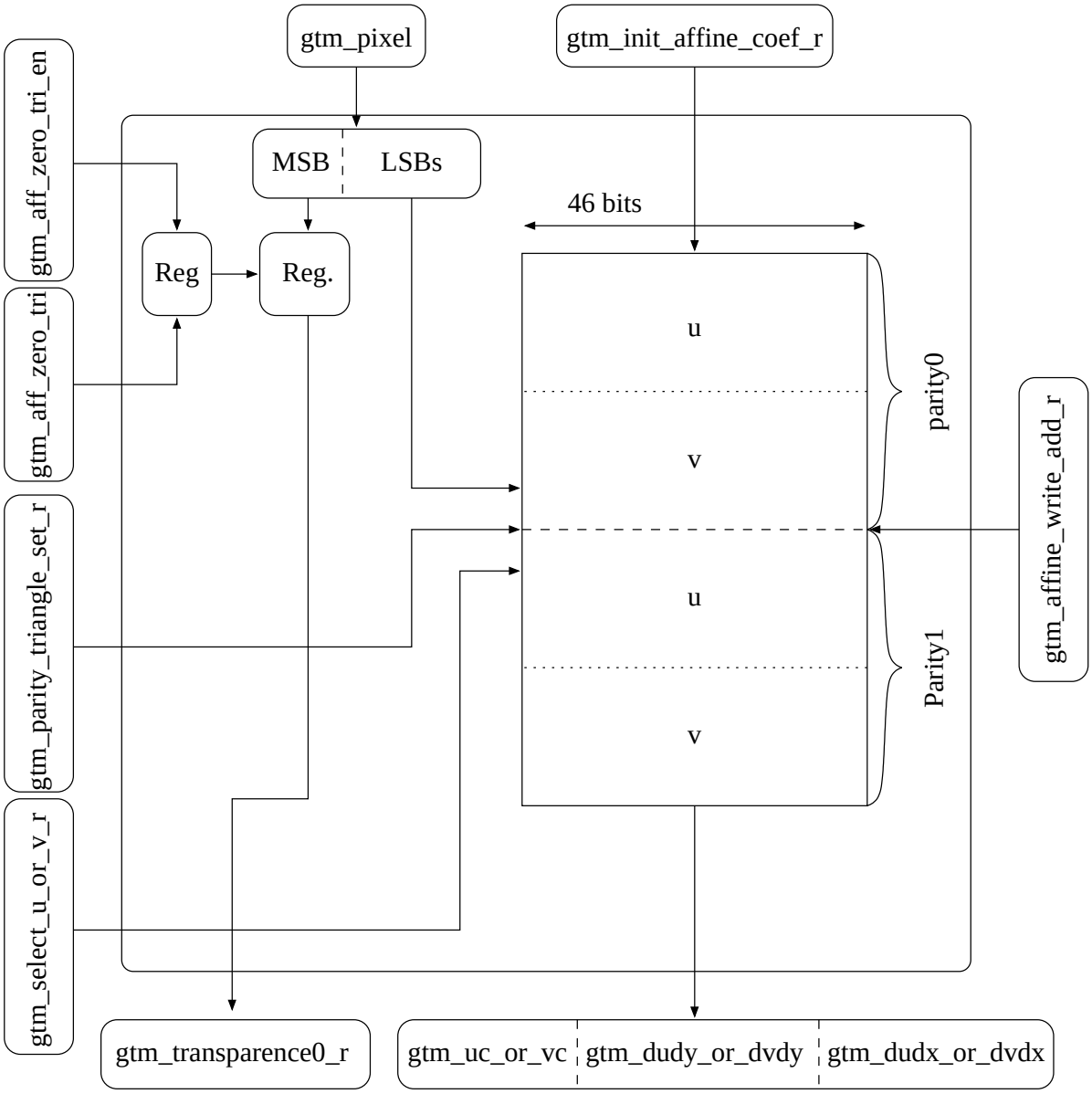


FIG. C.6 – Mémoire affine et logique de contrôle.

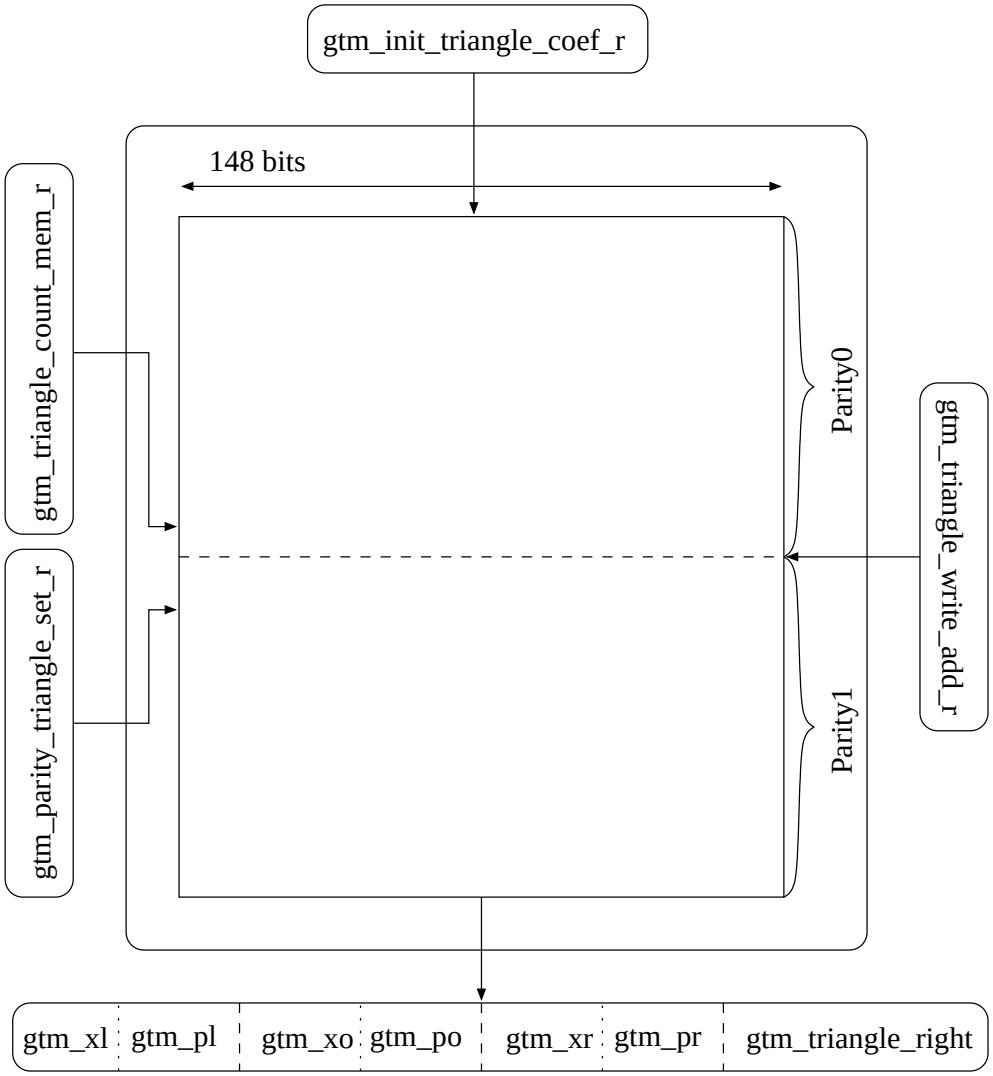


FIG. C.7 – Mémoire de parcours de triangles.