Université de Marne-La-Vallée

École Doctorale ICMS

ESIEE, Lab. A^{2}SI

THÈSE

pour obtenir le grade de

Docteur de l'Université de Marne-La-Vallée

Spécialité: Informatique

présentée et soutenue publiquement par

Benoit Kaufmann

Spécification et Conception d'un système auto-stéréoscopique multi-vues pour l'affichage tri-dimensionnel

Directeur de thèse : Mohamed Akil Professeur, ESIEE, lab. A^{2}SI

Date de soutenance : 21 Décembre 2006

Composition du Jury :

Président du jury :	Gilles Roussel	Professeur, Université de Marne-la-Vallée, IGM
Rapporteur :	Bernard Goossens	Professeur, Université de Perpignan, Équipe DALI
Rapporteur :	Daniel Thalmann	Professeur, EPFL, laboratoire Vrlab
Examinateur :	Gilles Bertrand	Professeur, ESIEE, lab. A^{2}SI
Examinateur :	Philippe Fuchs	Professeur, ENSMP, Centre de robotique
Examinateur :	Renaud Keriven	Professeur, ENPC, laboratoire CERTIS
Examinateur :	Patrick Sangouard	Professeur, ESIEE, lab. ELMI

Remerciements

Ce travail de thèse a été effectué au laboratoire “Algorithmique et Architecture des Systèmes Informatiques” (A^2SI) du groupe ESIEE et au sein de l'équipe “Géométrie discrète et imagerie” (CNRS-UMLV-ESIEE). Je remercie Monsieur le professeur Gilles Bertrand, responsable du laboratoire et de l'équipe “Géométrie Discrète et imagerie” de m'avoir accueilli dans son laboratoire et d'avoir accepté de participer au jury.

Je tiens à remercier Mohamed Akil, professeur à l'ESIEE, qui a dirigé cette thèse dans la continuité de mon stage de DEA. Partant du principe de base sur le dispositif d'affichage 3D que j'ai proposé, il a d'une part cru à ce projet et a d'autre part su tout le long de ces travaux m'apporter son aide et de nombreux conseils avisés aussi bien sur l'orientation de ces travaux que sur la rédaction de ce manuscrit de thèse.

Je tiens à remercier monsieur Gilles Roussel, professeur et directeur du Laboratoire d'Informatique de l'Institut Gaspard-Monge, Université de Marne-la-Vallée de m'avoir fait l'honneur de présider ce jury.

Je tiens à exprimer mes remerciements à messieurs Bernard Goossens, professeur à l'université de Perpignan et Daniel Thalmann, professeur à l'Ecole Polytechnique Fédéral de Lausanne, pour l'intérêt qu'ils ont porté à ces travaux en acceptant de les rapporter.

Je remercie également messieurs Phillippe Fuchs, professeur à l'Ecole Nationale des Mines de Paris et Renaud Keriven, professeur à l'Ecole des ponts et Chaussées, pour avoir accepté de participer au jury.

Mes remerciements à Patrick Sangouard, professeur à l'ESIEE, au laboratoire « Electronique et Microélectronique » pour avoir accepté de participer au jury, pour ses conseils et son aide sur l'étude de la faisabilité de ce nouveau dispositif d'affichage 3D et sur le processus de dépôt de brevet sur ce dispositif.

Je tiens à remercier tous les membres du laboratoire A^2SI, en particulier Thierry Grandpierre, Laurent Najman, Laurent Perroton, Denis Bureau, Michel Couprie, Eva Dokladalova, les assistantes et ingérieurs du laboratoire, Elisabeth Bastien, Fabiana Della zuana, Cristophe Dietrich et Eric Llorens.

Enfin, je remercie les thésards et stagiaires de recherche que j'ai côtoyés pendant toute la durée de ma thèse et notamment Ilhem Benachour, Rossani Roche, Charbel Fares, Cédric Sibade et Fadi Yacoub, pour leur amitié et leur soutien.

Résumé

Les images 3D sont actuellement en plein essor dans de nombreux domaines aussi variés que la médecine, les simulations physiques, la réalité virtuelle ou les jeux vidéos, mais également dans des domaines où les images de synthèse n'ont, à priori, pas une grande place, comme l'architecture, l'aménagement urbain ou la biologie moléculaire. Dans ces domaines, le fait de pouvoir afficher des images en relief permet de mieux appréhender le résultat affiché et d'en avoir une vision plus complète.

Il existe actuellement de nombreux systèmes permettant de percevoir des images en relief. Chacun a ses avantages et ses inconvénients. En particulier, beaucoup nécessitent de porter des dispositifs optiques (des lunettes, par exemple) pour percevoir l'image en relief. D'autres limitent la position de l'observateur à une certaine distance, ou nécessitent à l'observateur d'être détecté par le dispositif, ce qui limite le nombre d'observateurs possibles.

Nous avons apporté, au cours de cette thèse, des améliorations à un type particulier de ces systèmes : les systèmes d'affichage auto-stéréoscopique, permettant de cumuler les avantages pour l'observateur de tous les systèmes existants, sans en avoir les inconvénients.

Cette thèse présente donc le principe de ce système et en discute les avantages et inconvénients par rapport aux différents dispositifs d'affichage 3D existants.

Elle propose également une solution aux différents problèmes posés par ce principe, liés au nombre important de vues différentes à calculer : le temps nécessaire à calculer toutes ces vues, la taille mémoire nécessaire à les mémoriser avant affichage et le débit nécessaire pour les envoyer au dispositif d'affichage en moins d'un trentième de seconde, vitesse nécessaire à un affichage temps-réel. Cette solution consiste en un algorithme permettant de générer toutes les vues nécessaires à l'affichage en projetant les objets à afficher une seule fois et pas une fois pour chacune des vues. Les données étant générées sous forme de données 3D, les redondances présentes dans des vues différentes sont évitées, ce qui diminue la taille des données à mémoriser et, par conséquent le débit nécessaire pour les envoyer au dispositif d'affichage. Ces données peuvent ensuite être décodées très rapidement, afin de permettre un affichage de toutes les différentes vues

suffisamment rapidement pour être affichées. De plus, cet algorithme est un algorithme de compression des images 3D sans perte, permettant de générer les vues avec la même qualité que si elles avaient été générées individuellement.

Enfin, cette thèse étudie la faisabilité technique d'un tel dispositif, en particulier en décrivant et en discutant l'implémentation de la partie codage de cet algorithme sur une architecture spécialisée à base de GPU. Cette architecture spécialisée se présente en deux parties : la première partie, formée par le GPU, traite les données 3D des objets à afficher (coordonnées, texture, éclairage...) et génère les voxels à afficher, puis les rassemble en blocs de voxels contigus, placés horizontalement et à la même distance de l'écran. La deuxième partie récupère ces blocs, les trie et termine la mise en forme des données avant de les envoyer au dispositif d'affichage.

Mots-clés : affichage 3D, auto-stéréoscopie, multi-vues, observateurs multiples, sans convergence, compression et visualisation d'images 3D, algorithmes 2D/3D, architecture temps réel

Table des matières

Table des figures

Liste des algorithmes

Liste des tableaux

Introduction

Cadre et motivations

Les images 3D sont actuellement en plein essor dans de nombreux domaines aussi variés que la médecine, les simulations physiques, la réalité virtuelle ou les jeux vidéos, mais également dans des domaines où les images de synthèse n'ont, à priori, pas une grande place, comme l'architecture, l'aménagement urbain ou la biologie moléculaire. Dans ces domaines, le fait de pouvoir afficher des images en relief permet de mieux appréhender le résultat affiché et d'en avoir une vision plus complète.

En biologie moléculaire, par exemple, la forme ternaire d'une enzyme, c'est-à-dire la position des atomes les uns par rapport aux autres dans l'espace, permet de déterminer certaines de ses caractéristiques. Dans cette optique, un affichage permettant aux observateurs de voir cette structure d'un simple coup d'oeil au lieu d'avoir à tourner la représentation de cette molécule à l'aide d'une souris ou d'un autre dispositif est appréciable.

Il existe actuellement de nombreux systèmes permettant de percevoir des images en relief. Chacun a ses avantages et ses inconvénients. En particulier, beaucoup nécessitent de porter des dispositifs optiques (des lunettes, par exemple) pour percevoir l'image en relief. Il s'agit des systèmes stéréoscopiques, qui produisent une image calculée pour chaque oeil de l'observateur.

D'autres nécessitent une détection de la position de l'observateur pour afficher les images correspondantes, voire pour permettre à chaque oeil de l'observateur de percevoir

uniquement l'image qui lui est destinée. Ceci implique que l'observateur porte un dispositif permettant au système de déterminer avec exactitude la position de ses yeux et que le logiciel de rendu soit spécialement écrit pour modifier les images affichées en fonction de cette position. De plus, les caractéristiques physiques de l'appareil et des logiciels utilisés limitent le nombre d'observateurs pouvant utiliser le dispositif simultanément. En effet, la plupart de ces systèmes ne peuvent détecter qu'un seul observateur à la fois. Certains, utilisant des lunettes à obturation, peuvent en détecter deux, mais la vitesse d'affichage de l'écran ne peut pas afficher les images pour un nombre supérieur.

Certains appareils limitent la zone d'observation, comme les écrans auto-stéréoscopiques, utilisant une barrière parallaxe. Ce type d'écran permet de percevoir une image en relief sans port d'appareil particulier, mais oblige l'observateur à se trouver dans une zone réduite, le plus souvent juste en face et à une certaine distance de l'appareil et l'empêche de se déplacer librement autour des objets virtuels, sans être obligé de modifier le point de vue virtuel à l'aide du clavier ou d'un dispositif de pointage, comme une souris.

Un quatrième type d'appareil, que certains appellent appareil à projection volumique, permet de faire apparaître des points lumineux dans le volume délimité par la zone d'affichage. Ce type d'affichage permet de se déplacer librement autour de l'objet sans interaction logicielle, ni limitation du nombre d'observateurs, mais, de par leur conception même (projection de lumière sur une surface translucide) les objets affichés ne peuvent qu'être transparents et il est impossible d'afficher des objets opaques. Ceci limite leurs champs d'applications. Par exemple, il est impossible de rendre de manière réaliste un bâtiment car ses murs seraient transparents et on verrait les autres façades et bâtiments au travers.

Certains de ces différents systèmes, essentiellement des systèmes stéréoscopiques et des écrans LCD à barrière parallaxe, sont d'ores et déjà commercialisés par diverses sociétés, en particulier NVidia, Philips et Sharp, qui développent par ailleurs d'autres systèmes, essentiellement auto-stéréoscopiques basés sur des réseaux lenticulaires.

Une autre approche utilise des hologrammes générés par ordinateur qui sont ensuite restitués grâce à un LASER. Pour cela, on considère que chaque point représentant une partie d'un objet 3D est éclairé par une lumière cohérente et qu'il ré-émet une partie de cette lumière. Tous les points étant éclairés par la même lumière, on calcule les interférences entre les rayons ré-émis entre eux et avec la lumière incidente initiale sur un plan virtuel. Ceci est exactement la simulation de ce qui se passe réellement lorsqu'on crée un hologramme. On obtient donc un champ d'interférences sous forme d'une image 2D. En affichant cette image avec des points suffisamment petits pour que les interférences de la lumière puissent avoir lieu (de l'ordre de 0,1 à 1 micromètre) et en l'éclairant avec un LASER ayant la même longueur d'onde que celle utilisée pour calculer les interférences, on obtient une image 3D sous la forme d'un hologramme.

Cette technique est actuellement à l'étude dans plusieurs laboratoires de part le monde, notamment pour accélérer le calcul des interférences afin de pouvoir générer un hologramme complet en moins d'un trentième de seconde.

Nous avons apporté, au cours de cette thèse, des améliorations à un type particulier de ces systèmes qui sont les systèmes auto-stéréoscopiques à vues multiples.

En particulier, nous avons mis au point un algorithme de compression des données 3D et une chaîne de traitements de données permettant de générer des images 3D compressées. Ceci permet de pallier le fait que les données brutes à afficher représentaient une taille de plusieurs giga-octets, ce qui est difficilement utilisable par le dispositif, tant au niveau de la mémorisation, qu'au niveau des débits nécessaires pour envoyer ces données au dispositif. En effet, ces données doivent être envoyées en un temps suffisamment court pour assurer un affichage de 30 images par seconde nécessaire pour une application temps-réel, comme des jeux vidéos, des simulations physiques ou des affichages d'images tridimensionnelles, par exemple.

Nous avons également étudié l'adaptation de cette chaîne de traitements sur architecture spécialisée intégrant un GPU, afin d'atteindre cette cadence.

Enfin, nous avons étudié un nouveau système d'affichage 3D permettant de s'affranchir de certaines contraintes d'utilisation des autres systèmes auto-stéréoscopiques à vues multiples. Ces systèmes permettent d'afficher des images en relief sans besoin de porter un appareil optique particulier, ni d'avoir une quelconque interaction entre l'observateur et l'appareil, tout en permettant d'afficher des objets opaques ou transparents dans un volume perçu supérieur à celui utilisé par l'appareil lui-même. La quantité de calculs nécessaire pour produire des images affichables par de tels systèmes est telle qu'il est également possible d'envisager des applications temps réel, c'est-à-dire afficher au moins 25 images 3D par seconde à des résolutions d'au moins 1 024 × 768.

Résumé des contributions

Le principal problème qui se pose dans ce genre de systèmes d'affichage est la gestion des différentes images. En effet, pour que l'observateur ait réellement l'impression de se trouver devant les objets affichés, le passage d'une vue à l'autre doit se faire avec un minimum de différences entre elles. Pour cela, deux images successives doivent être très similaires. Par conséquent, il faut un grand nombre d'images différentes, au moins 20, chacune correspondant à une vue différente des objets affichés.

Il se pose donc un double problème : tout d'abord le temps nécessaire pour calculer toutes ces images et ensuite la taille mémoire nécessaire à leur stockage. Par exemple, pour une image 1 024 × 768 en couleurs codées sur 24 bits, une carte graphique accélératrice 3D parmi les plus puissantes du commerce comme la *NVidia GeForce 7900 GTX SLI* permet d'obtenir des cadences maximales de moins de 400 images par seconde. Si le nombre d'images à calculer est égal à 200, cette carte est incapable de produire les 200 vues plus de 2 fois par seconde, ce qui est un taux de rafraîchissement trop faible pour une application temps réel. Quant à la taille nécessaire à ces images, elle est de 1 024 × 768 × 3 × 200 = 471 859 200 octets, soit 450 méga-octets pour les 200 vues.

Les recherches actuelles se sont principalement concentrées sur la compression des images. L'approche actuellement utilisée par la plupart des chercheurs dans ce domaine est de calculer un certain nombre d'images, puis de coder l'ensemble avec des techniques

de détection de zones communes entre les différentes images. Ces techniques donnent des taux de compression importants (de l'ordre de 70%) mais nécessitent de calculer plusieurs images indépendantes à partir des mêmes données, ce qui est une perte de temps puisqu'une même information est calculée plusieurs fois. De plus, la compression nécessite des calculs complexes, comme des calculs de DCT (Discrete Cosine Transform) ou des codages d'entropie, utilisés par les compressions JPEG et MPEG qui peuvent être adaptées pour compresser les données 3D. La restitution de ces images nécessite donc une décompression similaire à la décompression MPEG que nous savons être gourmande en ressources. Le nombre important d'images à décompresser, correspondant aux différents points de vue possibles nécessiterait alors une architecture matérielle dédiée parallèle.

Nous avons donc proposé et validé un algorithme de compression de données 3D sans perte, permettant de restituer toutes les images en évitant les calculs redondants, qui permet une décompression des données générées avec une complexité quasi-linéaire et qui ne nécessite un calcul qu'en une seule passe.

Cet algorithme est fondé sur un codage direct des pixels des différents objets sous forme de voxels. Chaque voxel est caractérisé par ses coordonnées x, y et z dans le référentiel du système d'affichage et par sa couleur. Ces coordonnées sont entières et indiquées en pixel. Ainsi, dans le cas de l'utilisation de l'algorithme pour afficher une image 2D, chaque voxel aura comme coordonnées x et y correspondant à sa position à l'écran et z correspondant à 0 ou à la profondeur à laquelle l'image 2D devra être affichée. La couleur est définie par quatre composantes : rouge, vert, bleu et alpha (transparence) ce qui permet d'afficher des pixels transparents ou opaques.

Pour limiter la taille mémoire, nous ne codons que les voxels visibles (transparents ou non) et pas la grande majorité de voxels complètement transparents situés entre ou dans les objets.

Cet algorithme présente trois avantages : dans un premier temps, il diminue l'espace mémoire nécessaire aux images à une dizaine de méga-octets pour une image de 1 024 $\times$ 768 $\times$ 24 bits, quel que soit le nombre de vues différentes nécessaires. Ensuite, il permet d'obtenir jusqu'à plusieurs centaines d'images différentes, sans augmenter le temps de

calcul. Il permet de restituer ces images en un temps suffisamment court (inférieur à 500 millisecondes pour des images de 1 024 × 768 sur un AMD Athlon cadencé à 1.7 GHz, avec 256 méga-octets de RAM) pour décoder et restituer toutes les vues d'une image 3D en moins d'un trentième de seconde, en utilisant le parallélisme et une architecture spécialisée. Enfin, il permet de coder certains effets particuliers des objets comme les réflexions et la réfraction, ainsi que les zones des images pouvant être cachées par d'autres objets et de les restituer de manière réaliste pour tous les points de vue possibles, permettant ainsi d'obtenir une image sans défaut, quel que soit le nombre de vues voulues et les caractéristiques de la scène affichée. La mise au point et la validation de cet algorithme ont conduit à l'écriture de plusieurs programmes écrits en C, C++ et JAVA, utilisant différentes implémentations, et permettant de calculer une scène virtuelle puis de générer les différentes vues.

Cet algorithme peut également être utilisé dans d'autres cadres que les systèmes d'affichage auto-stéréoscopique : il peut, par exemple, être utilisé pour gérer des sprites 3D dans un jeu vidéo en permettant d'en re-générer la vue correspondant à la position du joueur, sans que toutes les vues possibles aient besoin d'être conservées en mémoire, ce qui permet un gain de place.

Cet algorithme entre dans une chaîne de traitement des données 3D permettant de générer les vues d'une scène 3D à partir des données issues d'un ordinateur. Cette chaîne de traitement est similaire à celles que constituent les cartes accélératrices 3D du commerce. Elle possède deux pipe-lines parallèles :

– Le pipe-line 3D calcule les projections des données 3D et les informations de rendu (plaquage de textures, éclairement, ombres, etc.) Ce pipe-line fournit les coordonnées 3D et la couleur de chaque voxel à afficher et le code en utilisant l'algorithme de compression 3D dont nous avons parlé plus haut.

  Les algorithmes utilisés dans ce pipe-line sont, pour la plupart, identiques à ceux

utilisés dans les cartes accélératrices 3D du commerce, en particulier, les algorithmes de projection des points 3D des objets et de rasterisation et de calcul des éclairements des objets et n'ont pas besoin d'être adaptés. Certains ont juste besoin d'être adaptés, comme l'algorithme de "clipping" dont le calcul des coordonnées de clipping doit être modifié pour élargir le champ de vision qui peut passer de 60 à 120° à cause du nombre de points de vue différents, et celui de l'anti-crénelage ("anti-aliasing" en anglais) qui doit générer des voxels sans prendre en compte le voxel placé juste derrière (comme c'est le cas pour les pixels des images 2D) mais avec sa couleur et un canal alpha calculé en fonction du lissage. Les autres algorithmes, comme le codage des voxels, le calcul des éclairages spéculaires, les réflexions et réfractions sont complètement nouveaux et ont dû être entièrement développés par nous.

– Le pipe-line 2D est utilisé pour les affichages n'utilisant pas de données 3D (comme les traitements de texte, par exemple), mémorise les pixels de l'affichage 2D, comme le ferait une interface VGA classique et les transforme en données 3D afin de pouvoir les afficher sur le dispositif. Cette transformation se fait simplement en rajoutant une profondeur définie comme coordonnée z de ces pixels et en codant les voxels obtenus selon notre algorithme.

Les voxels fournis par ces deux pipe-lines sont ensuite fusionnés (en prenant en compte les informations de positions des fenêtres 3D et des affichages 2D à l'intérieur des affichages 3D, le cas échéant) pour former une seule image 3D incluant à la fois les données correspondant aux objets 3D et à l'affichage 2D. Enfin, cette image est envoyée au dispositif d'affichage pour être affichée.

La contrainte temps réel nous a amenés à étudier l'implémentation de cet algorithme sur une architecture associant logiciel (processeur GPU) et matériel (circuit dédié). Nous avons donc étudié la possibilité de l'inclure dans une chaîne de traitement de données se comportant comme une carte accélératrice 3D. Cette chaîne de traitement permet de coder directement les données provenant d'un programme d'affichage 3D et de restituer

toutes les vues au dispositif d'affichage.

Nous avons donc proposé et étudié différentes implantations possibles de cette chaîne que nous avons simulée grâce à SystemC et un logiciel de modélisation et simulation développé par la société Summit nommé Visual Elite et adapté une partie des traitements sur GPU afin d'obtenir une vitesse de calcul des voxels d'une scène 3D complète et leur codage en moins d'un trentième de seconde, vitesse compatible avec la contrainte temps réel, et de valider la possibilité d'utiliser cette technologie dans la chaîne de traitement. Ce codage a été adapté de la technique du "Depth peeling", utilisée pour le "BluePrint Rendering" (dessin dans le style plan d'architecte). Il consiste à faire dessiner un objet par le GPU, de récupérer l'ensemble des pixels visibles et leur profondeur (pour en faire des voxels) puis de relancer le rendu à partir de la profondeur récupérée pour chaque pixel de l'image. On obtient ainsi certaines des faces cachées de l'objet. En répétant cette technique jusqu'à ce qu'il n'y ait plus de pixel visible (ou que leur nombre soit inférieur à un seuil prédéfini) on récupère tous les voxels constituant la scène 3D à rendre.

Le GPU peut également indexer l'emplacement du début de chaque motif. Ainsi les seules tâches du circuit dédié consistent à classer les motifs et à les restituer sous forme de données compressées, qui sont des tâches très lentes à implémenter sur un GPU. Ceci est dû à l'architecture même des shaders du GPU. En effet, ces shaders fonctionnent en parallèle et leurs données de sorties doivent avoir une taille et une position définies dans la mémoire de résultat avant le calcul. La taille du codage de chaque motif dépendant de la taille de celui-ci et de la taille des précédents, il est très difficile de prédire ces informations. De plus, l'algorithme impose de compter le nombre de motifs dans chaque plan et de plans dans chaque ligne, opérations que le GPU ne peut faire qu'au prix d'un nombre d'itérations égal au nombre d'éléments à compter. Ce comptage devant se faire au fur et à mesure, le nombre de boucles devient important pour une scène complète et l'utilisation des shaders pour chacune de ces itérations étant limité, le parallélisme du GPU n'est plus du tout exploité. Il nous a donc paru plus efficace de laisser ces tâches au circuit dédié et de laisser le GPU exploiter ce temps pour faire des calculs de projection et de rendu, pour lesquels il obtient de meilleures performances qu'un CPU (de

l'ordre de 5 fois plus rapide - performances obtenues pour un encodage vidéo réalisé par CHIP Online avec le programme "Avivo XCode" développé par ATI, avec un AMD Athlon 64 FX 57, 2,8 GHz, 1 giga-octet de RAM DDR-550 et un GPU Nvidia nForce 4 SLI).

La théorie d'un nouveau procédé d'écran auto-stéréoscopique a également été étudiée. Cet écran permettra non seulement de percevoir une scène virtuelle en relief, sans utilisation de lunettes ni de détection de la position de l'observateur, mais permettra également à celui-ci de se déplacer à droite et à gauche (voire vers le haut et le bas) afin de changer son angle de vue comme si les objets affichés étaient physiquement devant lui.

Le nombre d'observateurs est virtuellement infini et chacun d'entre eux perçoit l'affichage selon son propre point de vue, en fonction de sa position par rapport au dispositif d'affichage.

Ils peuvent également (et c'est l'avantage principal de ce système sur les autres systèmes auto-stéréoscopiques) se déplacer d'avant en arrière sans être obligés de se trouver à une distance précise de l'écran.

Ce principe optique a également été validé grâce à un logiciel que nous avons écrit et qui nous a permis de vérifier la perception des observateurs en fonction de leur position par rapport à cet écran.

La faisabilité de ce principe optique a également été validé en 2D par un logiciel de simulation optique 2D nommé raytrace et en 3D par un logiciel de simulation optique plus puissant nommé ray-eval.

Le principe optique, ainsi que les moyens de sa réalisation sont actuellement en passe d'être brevetés.

Plan de la thèse

Cette thèse présentera tout d'abord le principe de la théorie du nouveau procédé d'écran auto-stéréoscopique, le comparera aux autres procédés existants et discutera de leurs avantages et inconvénients dans le cadre de la problématique qui nous intéresse.

Ensuite, nous expliquerons l'algorithme de compression des images 3D que nous avons proposé, ainsi que l'algorithme de décompression associé, et l'ensemble de la chaîne de traitement des données 3D permettant de générer, compresser, puis restituer les vues à afficher. Nous montrerons les résultats obtenus par simulation et leurs caractéristiques.

Nous verrons ensuite comment nous pouvons accélérer et paralléliser l'algorithme de compression pour l'adapter à une implémentation sur GPU.

Enfin, nous étudierons la possibilité de mise en oeuvre du système d'affichage complet. Dans un premier temps, nous montrerons comment nous pouvons paralléliser cet algorithme pour l'adapter à une implémentation sur GPU. Puis nous étudierons la modélisation fonctionnelle de la chaîne de traitement contenant ce GPU et discuterons de différentes implémentations possibles. Puis nous étudierons la faisabilité du système physique, en analysant les capacités des systèmes optiques, électroniques et mécaniques nécessaires à la mise en oeuvre du système et en les comparant aux caractéristiques de la perception humaine et les contraintes du temps réel.

Chapitre 1

L'autostéréoscopie multi-vues

Le procédé que nous avons proposé est fondé sur une adaptation du principe de l'autostéréoscopie, procédé permettant de reconstituer la vision binoculaire, c'est-à-dire le principe optique permettant aux êtres humains de percevoir le relief, sans nécessiter l'utilisation de lunettes, ni d'interaction entre l'observateur et le dispositif d'affichage, comme le "head-tracking".

Ce chapitre décrit tout d'abord le principe de l'autostéréoscopie, puis fait un résumé des principaux dispositifs autostéréoscopiques et autres, permettant un affichage en relief. Nous discutons ensuite des principes et avantages de la nouvelle méthode pour afficher les objets en trois dimensions (3D) que nous avons proposée, ainsi que les tests qui nous ont permis de la valider, ainsi que leurs résultats.

1.1 Principe de l'autostéréoscopie

1.1.1 Principe de la vision binoculaire

Un peu d'histoire...

La recherche sur la compréhension de la vision en relief intéresse les scientifiques depuis très longtemps. En effet, les premières traces de recherches dans ce domaine remontent à l'antiquité. Il s'agit du recueil "l'optique" d'Euclide (325 av JC - 265 av JC, dates approximatives) qui énonce que voir le relief c'est recevoir au moyen de chaque œil l'impression simultanée de deux images dissemblables du même sujet. Il associe cette idée avec les prémices de la persistance rétinienne en écrivant que si nous croyons voir avec nos

deux yeux un objet unique, cela tient à la rapidité extrême avec laquelle nos deux yeux en parcourent toutes les parties, et à la simultanéité d'impression faite ainsi dans nos deux yeux, par ces deux images, pourtant distinctes. Cette idée fut reprise plus tard par le médecin grec Galien, qui vécut à Rome, sous l'empereur Marc-Aurèle, c'est-à-dire vers l'an 170 après Jésus-Christ. Ensuite, aucun autre écrit ne parle du principe de la vision stéréoscopique avant le traité sur la peinture de léonard de Vinci, édité en 1651, mais il avait réalisé les premiers dessins et réflexions sur la vision en relief dans un manuscrit écrit à Milan en 1584. D'autres scientifiques ont repris ces travaux, mais aucun n'a réalisé de prototype permettant de vérifier cette hypothèse, avant Charles Wheatstone (physicien anglais né en 1802 et mort en 1875) en 1838. Il décrivit également avec précision le phénomène et comment reconstituer le relief avec deux images différentes (dessins, photographies ...) et modifier l'impression de profondeur (perspective forcée) en changeant le sens ou les prises de vue des photographies. Cette recherche lui valut la *Royal Medal of the Royal Society* en 1840. Depuis, beaucoup de systèmes d'affichages 3D ont vu le jour.

La vision humaine

L'œil humain (dont le schéma anatomique est représenté figure 1.1) est l'organe de la vision. Il permet donc à un être vivant de "voir", c'est-à-dire d'analyser la lumière visible pour interagir avec son environnement. La lumière visible correspond à certains rayonnements électromagnétiques dont les longueurs d'ondes sont généralement situées entre 390 nm (les ultraviolets) et 700 nm (les infrarouges).

Il se comporte comme un objectif de caméra : le cristallin sert à l'accommodation (ou mise au point) afin que l'image de l'objet observé projetée sur la rétine soit nette, l'iris sert à faire varier la quantité de lumière utilisée (afin d'obtenir une sensibilité variant entre des valeurs d'intensité lumineuse ayant un rapport de l'ordre de 10 000) et la rétine sert à transformer les informations lumineuses en signaux neuronaux qui sont envoyés directement au cerveau. Le schéma 1.2 montre ce principe.

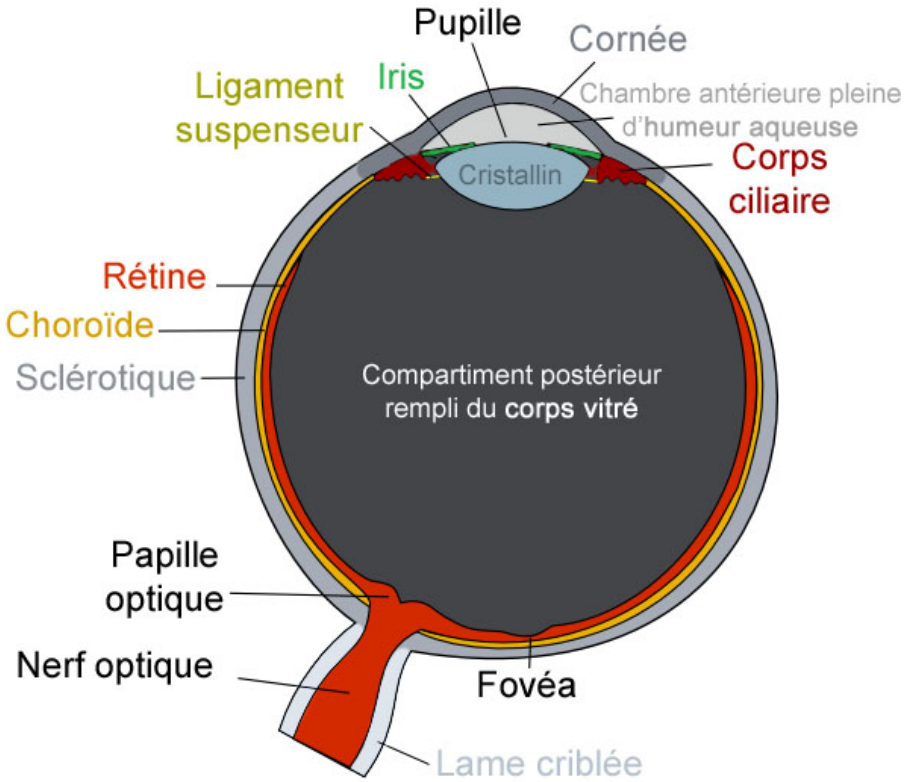


FIG. 1.1 – Schéma anatomique de l'oeil humain

La rétine est formée de plus de 100 millions de cellules visuelles. Ces cellules sont de deux types :

- les bâtonnets (au nombre de 120 millions) particulièrement sensibles aux faibles lumières,
- les cônes (6 millions) qui permettent la discrimination des couleurs.

Elle comporte une forte concentration de ces cellules au centre et de moins en moins en allant vers les bords, où l'on ne trouve plus que de rares bâtonnets. Sa zone centrale est occupée par la macula ou tache jaune, d'environ 2 mm^2 et richement dotée en cônes et en bâtonnets. Le milieu de la macula est lui-même occupé par une petite dépression de 0,4 mm de diamètre, la fovéa, où ne se trouvent que des cônes extrêmement serrés. C'est là que l'acuité visuelle est maximale, lorsque la luminosité est suffisante, par exemple en plein soleil. La nuit, faute de bâtonnets, la fovéa est aveugle et pour percevoir la lumière

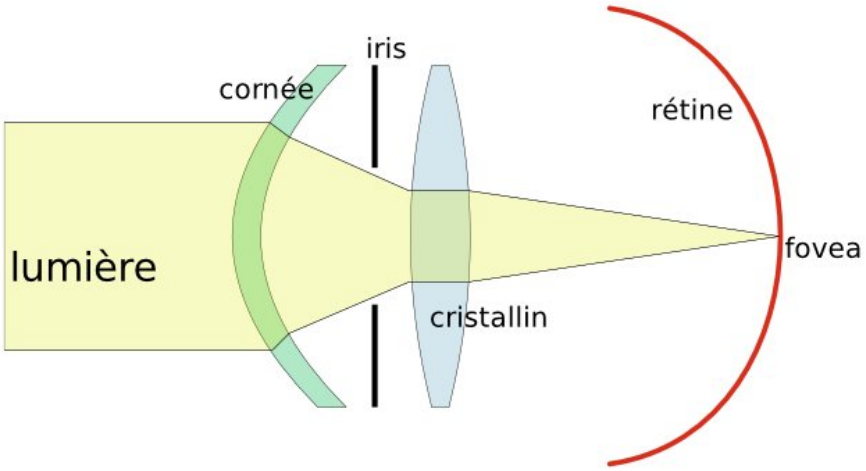


FIG. 1.2 – Principe optique de l'oeil humain

d'étoiles visibles les plus faibles, il faut regarder un peu à côté, en utilisant les cellules de la macula. Le schéma 1.3 montre la position de la fovéa dans l'œil. La fovéa correspond à un angle de vision d'environ 1,5° et c'est sur elle que nous amenons instinctivement l'image du point que nous sommes en train de fixer.

Loin de la fovéa, l'acuité visuelle diminue rapidement jusqu'à devenir très médiocre, mais l'œil compense cette insuffisance par une très grande mobilité grace à laquelle il peut explorer les diverses parties du sujet qui se trouve devant lui. Cette mobilité est permise grace aux muscles oculomoteurs qui permettent au globe oculaire entier de se tourner dans la direction des objets à observer. Ces muscles sont au nombre de 6 :

- 4 muscles droits : droit supérieur, droit inférieur, droit interne et droit externe
- 2 muscles obliques : oblique interne et oblique externe.

Les mouvements du globe oculaire sont les mouvements les plus rapides que peut produire le corps humain. La vitesse de rotation d'un globe oculaire peut atteindre 400° par seconde. Les recherches ont montré qu'un œil normal pouvait séparer des points

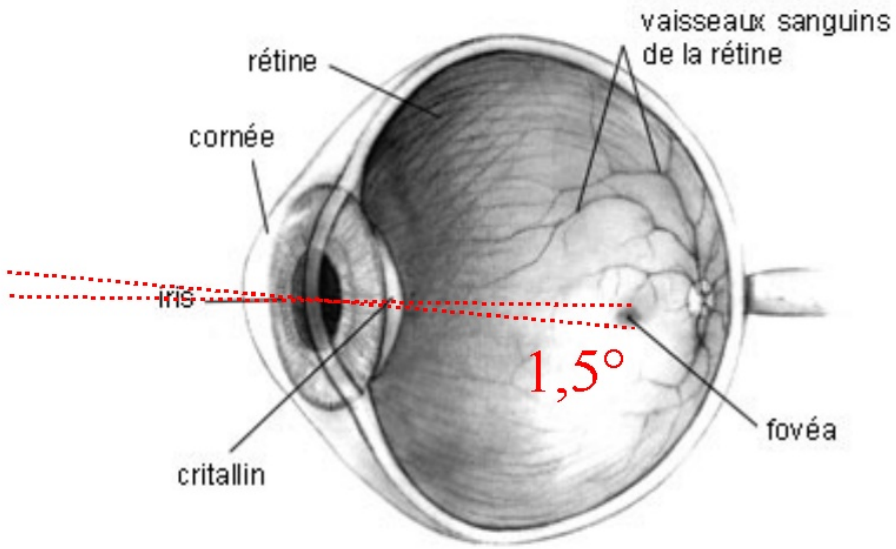


FIG. 1.3 – Position de la fovéa dans l'oeil humain

lumineux distants angulairement d'environ 1' (une minute) d'angle, ce qui équivaut à $\frac{1}{60}$ de degré ou environ $\frac{1}{3000}$ de radian. Les caractéristiques optiques moyennes de l'œil sont indiquées dans le tableau 1.1.

La convergence stéréoscopique

Un seul oeil ne permet pas la vision binoculaire, c'est-à-dire la perception simultanée d'un même objet selon deux points de vue légrement différents, permettant de percevoir le relief directement. Pour cela, il faut impérativement deux points de vue légèrement distincts. Nos deux yeux étant séparés par une distance d'environ 6 cm en moyenne, lorsque nous regardons un objet, nous voyons en réalité deux images différentes (voir figure 1.4) de cet objet. C'est la *parallaxe stéréoscopique*.

Cette perception du relief est due aux mouvements que font nos yeux pour parcourir les objets. En effet, ils ne parcourent pas la totalité de l'objet, mais convergent sur un certain nombre de points sur lesquels les deux yeux peuvent se focaliser, comme un coin, une tache, un croisement de lignes, etc. pour que ceux-ci soient projetés sur la fovéa et donc vus avec le maximum de précision. Cette convergence du regard s'appelle la

Structures	Rayon de courbure antérieur	Rayon de courbure postérieur	Indice de réfraction
cornée	7,8 mm	6,8 mm	1,377
humeur aqueuse	-	-	1,337
cristallin	10 mm	6 mm	1,413
humeur vitrée	-	-	1,337

- distance focale image : +22 mm
- distance focale objet : -17 mm
- distance foyer objet → face antérieure de la cornée : +15mm
- distance face antérieure de la cornée → plans principaux : +2 mm
- distance face antérieure de la cornée → rétine : +24 mm
- rayon de courbure : +6 mm
- minimum séparable : $\alpha = 1'$ (une minute d'angle)
  Cette valeur correspond à l'angle que forment les rayons lumineux issus des deux points les plus proches séparables qui arrivent dans l'œil. Si cet angle est inférieur à cette valeur (si la distance entre les points diminue ou si l'observateur s'éloigne) les deux points ne seront plus perçus comme deux points distincts mais comme un seul.

TAB. 1.1 – caractéristiques optiques moyennes de l'œil

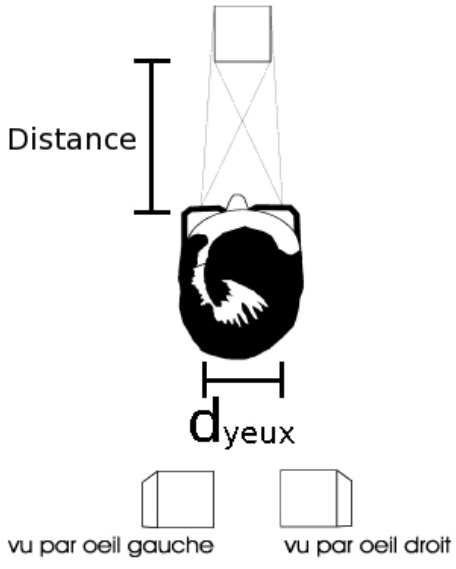


FIG. 1.4 – Perception de deux images différentes avec chaque oeil

convergence stéréoscopique. En convergeant sur un point, les axes des yeux vont former un angle, noté φ (voir figure 1.5) et appelé *angle de site*.

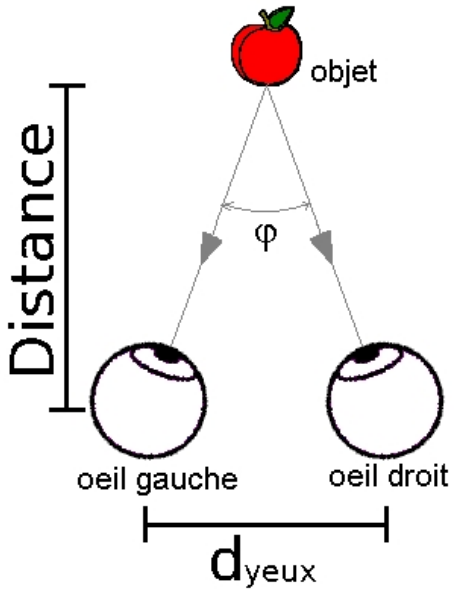


FIG. 1.5 – Angle de site

Cet angle de site est égal à $\varphi = 2 \times \arctan \frac{\frac{d_{yeux}}{2}}{Distance}$

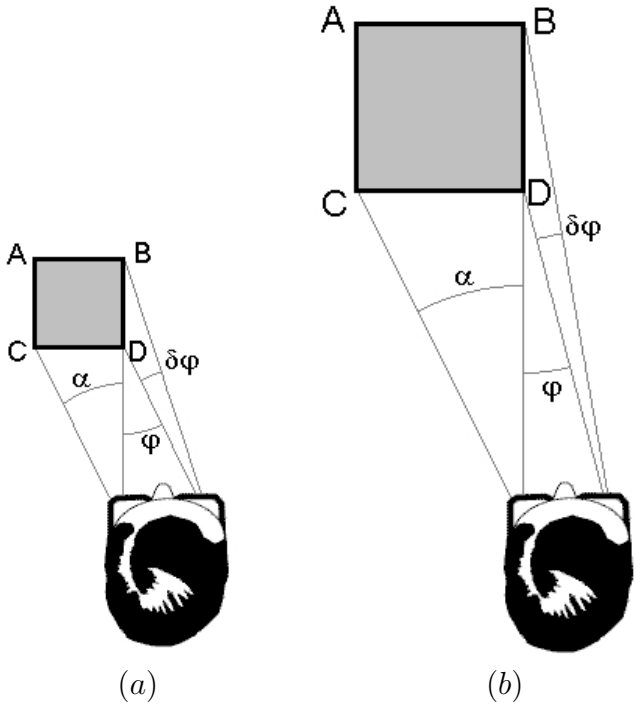
où

- d_{yeux} est la distance entre les deux yeux de l'observateur,
- *Distance* est la distance séparant les yeux de l'observateur de l'objet observé.

Ceci est vrai tant que le volume de l'objet observé est suffisamment grand par rapport à sa distance. Si deux points (ou deux objets) se trouvent à une distance relativement proche, notre oeil n'est pas capable de voir la différence entre les deux images perçues et tous les objets nous paraissent être à la même distance. C'est pour cela que, lorsqu'on regarde un ciel étoilé, toutes les étoiles nous paraissent être sur une sphère au centre de laquelle nous nous trouvons. C'est également pour cela qu'on apprécie mieux un trompe-l'oeil lorsque l'on se place à quelques mètres : parce que la perception du relief par la vision binoculaire disparaît et qu'elle est remplacée par des déductions faites à partir des ombres sur les objets et portées par eux.

Pour comprendre le phénomène de la diminution de la parallaxe stéréoscopique, imaginons un cube d'une certaine taille vu depuis une certaine distance. Sur le dessin 1.6-a, l'observateur regarde le coin D du cube. L'angle φ lui permet d'évaluer la distance qui l'en sépare. La largeur apparente du cube, c'est-à-dire l'angle sous lequel il est vu, est représenté par α sur le schéma. L'impression de relief du cube est donnée par la distance BD, que le cerveau évalue uniquement en estimant l'angle $\delta\varphi$, car l'œil gauche de l'observateur ne peut pas voir le point B.

Si on remplace ce cube par un autre cube deux fois plus grand mais situé deux fois plus loin de l'observateur (figure 1.6-b), sa largeur apparente α reste la même. Quant à l'angle $\delta\varphi$, il est divisé par 2. L'impression de relief diminue donc lorsque la distance de l'objet à l'observateur augmente : ainsi une même longueur est vue sous un angle apparent différent suivant qu'elle est vue axialement ou radialement.



(*a*) (*b*)

FIG. 1.6 – Perte de précision de la parallaxe stéréoscopique

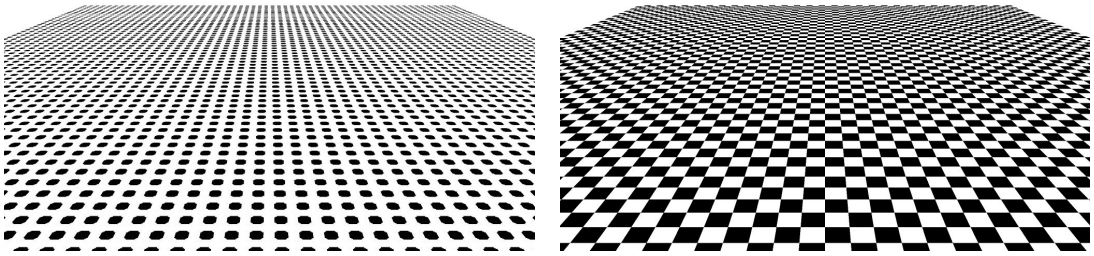
La perception de la profondeur de manière cognitive

Simultanément à cette action de convergence, les yeux accommodent, c'est-à-dire "mettent au point" l'image rétinienne. Il existe une synergie acquise entre ces deux actions : convergence et accommodation. Cependant, l'accommodation n'est vraiment remarquable que pour des objets proches et ne permet pas de percevoir un réel relief de manière consciente.

D'autres paramètres non stéréoscopiques permettent aussi d'évaluer le relief. Ces paramètres sont utilisés en parallèle de la vision binoculaire et permettent de percevoir quand-même le relief lorsque celle-ci n'est pas possible ou est inefficace. Par exemple, les personnes borgnes, atteintes de cécité à un œil ou souffrant de strabisme sévère ne peuvent prétendre à la vision stéréoscopique. Plus simplement, lorsqu'on observe une photo, l'image est plane. Il n'y a donc pas de différence entre les images perçues par l'œil gauche et par le droit et donc pas de relief. De même, lorsque l'objet observé est très loin, la différence entre les deux images perçues n'est plus détectable par nos yeux. Pourtant, dans la plupart des cas, on arrive à discerner la forme des objets et leur profondeur relative. Ceci est dû à la manière dont le cerveau interprète ce qu'il voit.

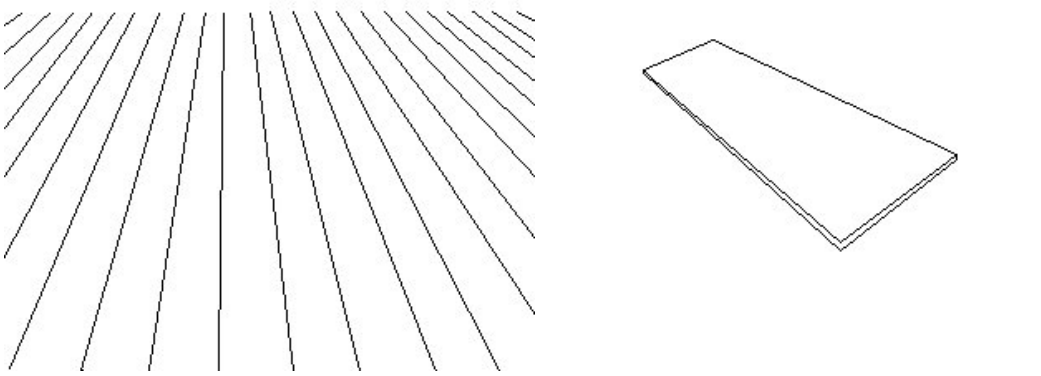
Par exemple, les gradients de texture ou de densité optique permettent d'avoir une idée de la profondeur d'un objet ou d'une partie d'objet. En effet, un plan au niveau du sol comportant des objets régulièrement espacés (voir figure 1.7-a) produit une image visuelle en gradient : plus les objets sont éloignés, plus leur fréquence spatiale est élevée. De même, ce même plan texturé de manière régulière (voir figure 1.7-b) aura un gradient de texture croissant avec la profondeur, c'est-à-dire que la texture sera plus petite loin de l'observateur que proche.

De la même manière, les objets sont également déformés par cette perspective : des lignes parallèles (voir figure 1.8-a) paraissent se rapprocher les unes des autres en s'éloignant de l'observateur, en convergeant vers un point de fuite, l'image apparente d'un rectangle (voir figure 1.8-b) vu devant soi est un quadrilatère, etc.



a - Gradient de densité optique b - Gradient de texture

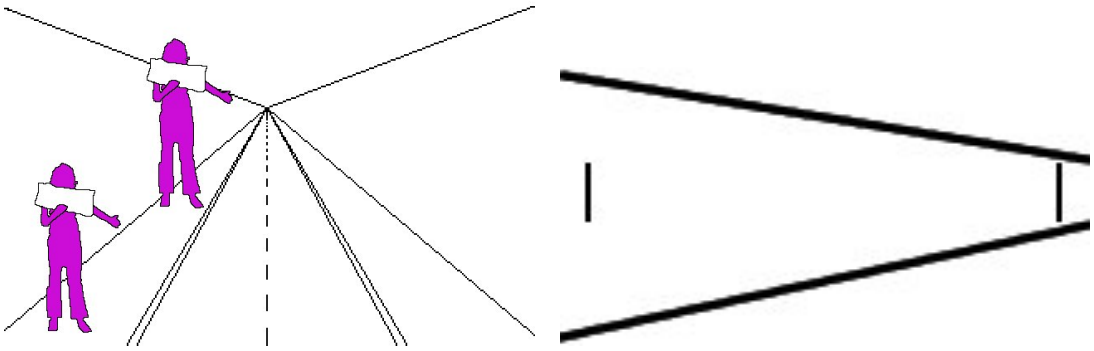
FIG. 1.7 – Impression de relief obtenus par gradients



a - Lignes parallèles b - Déformation d'un rectangle

FIG. 1.8 – Déformations d'objets par la perspective

De plus, la perspective permet d'avoir une idée des taille et distance relatives des objets en se basant sur les lignes et points de fuite. La figure 1.9-a montre un exemple de dessin où les lignes convergeant en un point sont interprétées par le cerveau comme une perspective de fuite, c'est-à-dire des objets parallèles dont la perception est déformée par la distance. Il va donc prendre ces lignes comme référence pour estimer la distance et la taille relative des objets se trouvant à proximité de ces lignes. Ainsi les deux silhouettes au bord de la "route" ont strictement la même taille sur le dessin, mais celle située près du point de fuite (le point de convergence des lignes de fuite) semble plus lointaine et donc (comme elles ont la même taille apparente) plus grande. De la même manière,



a - avec un point de fuite b - sans point de fuite

FIG. 1.9 – Exemples de perception erronée à cause de la notion psychophysiologique de la perspective

même en l'absence de point de fuite, deux lignes obliques, comme celles de la figure 1.9-b, peuvent être interprétés comme référence factice et donner l'apparence d'une différence de taille des deux lignes verticales, alors qu'elles ont également la même taille. C'est cette faculté du cerveau qui permet d'estimer les tailles et distances relatives d'objets sur un dessin ou une photographie de rue, alors qu'aucun relief réel n'est perceptible. C'est grace à elle qu'il est plus facile de reconnaître le relief dans une photo plane qu'en regardant avec ses deux yeux des avions en vol ou des étoiles. Ceci est dû à plusieurs choses :

1. Les objets observés sont trop loin pour que l'on puisse utiliser notre vision stéréoscopique.
2. Leur taille relative est trop proche pour que l'œil puisse l'apprécier. Elle serait de toute façon trompeuse car l'étoile la plus brillante et l'avion le plus gros ne sont pas forcément plus proches.
3. L'observation de ces objets se fait dans un environnement uni, donc sans point de repère ou de référence permettant de déterminer leur distance.

Il existe cependant d'autres indices permettant de connaître la profondeur de certains objets.

La désaturation des couleurs (voir figure 1.10-a) et l'atténuation des contrastes (voir

figure 1.10-b) avec la distance permettent également de donner une impression de la profondeur des objets. Dans la nature, ces deux phénomènes sont dûs au gouttelettes

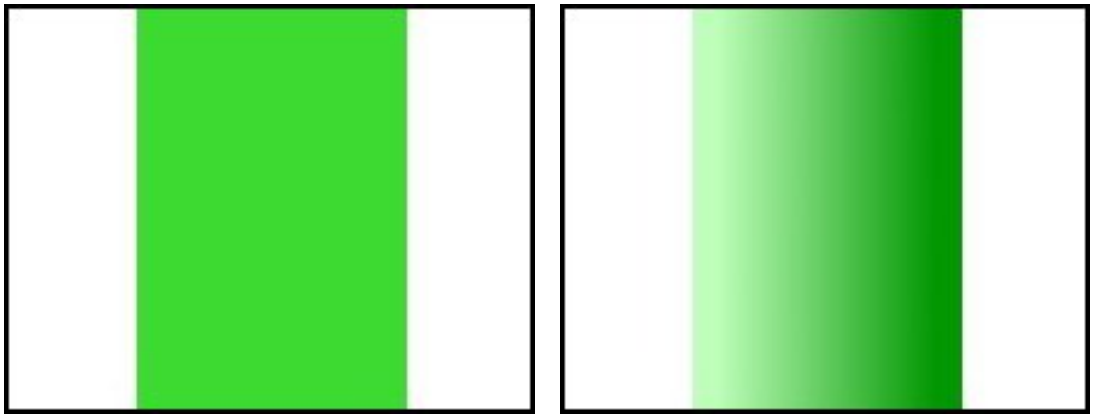
a - Désaturation des couleurs b - Atténuation des contrastes

FIG. 1.10 – Exemples de dénaturation de l'image avec la distance

d'eau en suspension dans l'air qui diffracte les rayons lumineux. Le cas extrême de ces phénomène peut facilement être observé lors du brouillard.

Les ombres participent également à la perception de profondeur. En particulier, elles donnent une information sur la forme de l'objet (différence d'éclairement en fonction de l'angle que fait la surface avec la direction des sources lumineuses) et sur sa position par rapport aux autres objets (ombres portées par l'objet sur d'autres). Par exemple, la figure 1.11 montre le même rectangle texturé de deux manières différentes : celui de gauche a une texture unie et semble être plat, alors que celui de droite est dégradé et semble être un cylindre. Cette illusion vient du fait que le dégradé est interprété comme un éclairage différent selon les points du rectangle à cause d'une orientation différente.

Il existe également des procédés plus ou moins conscients basés sur notre mémoire. En particulier, la connaissance à priori de la forme des objets et les lignes brisées nous permet de détecter les objets placés devant d'autres, comme ceux affichés sur la figure 1.12-a. Sur cette figure, il paraît évident que le disque est le plus lointain des trois objets, le triangle, le plus proche et le carré entre les deux. Mais ce n'est qu'un à priori car rien



a - Éclairage uniforme (pas d'ombre)

b - Texture donnant l'illusion de l'éclairage d'un cylindre

FIG. 1.11 – Différence de perception du volume d'un objet en fonction de son éclairage

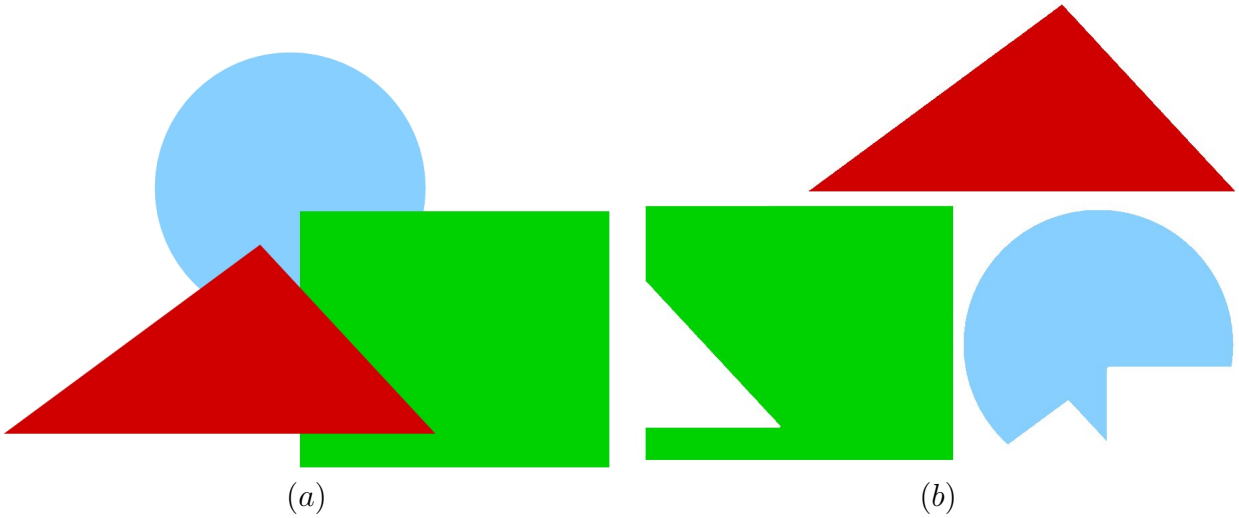


FIG. 1.12 – Interposition de plusieurs objets

ne nous dit que ces trois objets n'ont pas les formes indiqués figure 1.12-b. Ceci est dû au fait que nous sommes habitués à voir ces formes et que notre cerveau essaie instinctivement de reconstituer des formes simples et connues. Si ces formes sont tronquées et que la troncature correspond exactement avec le contour d'un autre objet, le cerveau en déduit que le second objet cache en partie le premier. Ce dernier est donc forcément plus loin que l'autre.

Tous ces phénomènes (à part l'accommodation, la désaturation des couleurs et l'atténuation des contrastes ; il reste donc la disparité rétinienne, la perspective, le gradient de densité optique et les ombres) doivent être respectées lorsqu'une image en relief est affichée. Dans le cas contraire, l'image sera perçue (inconsciemment) comme choquante et non réelle. La désaturation des couleurs avec la distance augmente l'impression de relief, mais n'est pas nécessaire pour que l'image soit correctement perçue.

Il existe également un moyen d'utiliser la vision stéréoscopique de manière cognitive lorsque celle-ci n'est pas utilisable : la *motion parallaxe* ou parallaxe de mouvement. Elle consiste à comparer deux images perçues à deux moments et deux positions différentes. Pour en comprendre le principe, il suffit de regarder par la fenêtre latérale d'un véhicule (voiture, bus, train ...) lorsqu'il se déplace devant un paysage dégagé ou lorsqu'il passe devant une rue transversale. On remarque facilement que tous les objets que l'on voit semblent se déplacer dans le même sens (contraire au mouvement du véhicule) et plus les objets sont proches de nous, plus ils se déplacent rapidement. Ce mouvement relatif des objets nous renseigne sur leur distance.

Cette technique est utilisée par certains algorithmes de reconstruction 3D utilisant plusieurs photos d'un même objet prises en faisant tourner celui-ci devant l'appareil. En calculant le déplacement des points de l'objet d'une image à l'autre, ces algorithmes en déduisent leur distance par rapport à l'appareil photographique. La même technique est utilisée en astronomie (voir figure 1.13) pour déterminer la distance d'un astre lointain par rapport à un autre (l'astre de référence) dont on connaît la distance. Il suffit de calculer à deux moments donnés, par exemple un peu après le coucher et un peu avant le lever du soleil, correspondant à deux positions différentes, notées "Position d'observation 1" et "Position d'observation 2" sur la figure 1.13, l'angle que forme l'image de l'astre observé avec l'astre de référence. Sur la figure 1.13, ces angles correspondent à α pour la position d'observation 1 et β pour l'autre. Connaissant la position de l'astre de référence, donc l'angle δ sur la figure, on peut déduire l'angle γ et donc la distance de l'astre observé. On obtient une vision stéréoscopique avec une distance entre les points d'observation plus

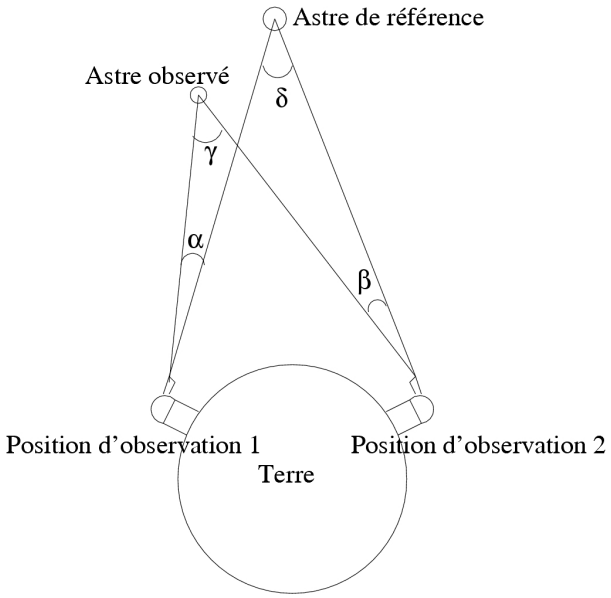


FIG. 1.13 – Utilisation de la motion parallaxe en astronomie

grande, donc une précision plus grande. Lorsque les objets à observer sont plus loin, on peut attendre 6 mois et utiliser cette fois deux positions opposées de l'orbite terrestre.

1.1.2 Les systèmes autostéréoscopiques

Le principe de l'affichage stéréoscopique

La plupart des systèmes d'affichage 3D sont basés sur une séparation de ce que voit chacun des deux yeux de l'observateur. Cette séparation est opérée soit par l'appareil lui-même, par l'intermédiaire de masques ou de caches, soit par des lunettes.

Dans le premier cas, le dispositif affiche les images destinées à chacun des deux yeux de l'observateur. Chacune de ces deux images n'est vue que par l'oeil auquel elle est destinée, grace à un système optique composé de miroirs, prismes, lentilles ou obturateurs mécaniques ou électroniques. Les images perçues doivent être alignées et être perçues de telle sorte que la convergence rétinienne puisse se faire. C'est-à-dire qu'elles doivent pouvoir être perçues superposées et ne doivent pas être trop éloignées, par exemple.

Dans le second cas, le système affiche ou projette les deux images au même endroit et les lunettes permettent à chaque œil de ne voir que l'image qui lui est destinée. L'image

en relief apparaît donc sur l'écran avec les objets situés devant ou derrière cet écran.

Cette technique, très simple, permet d'obtenir une vision de très bonne qualité, indépendamment de la position de l'observateur, sauf dans le cas d'une séparation opérée par l'appareil si celui-ci n'est pas transportable. Dans ce cas, on obtient une vision de très bonne qualité à l'unique position possible de l'observateur.

L'utilisation de lunettes pour séparer les images pose cependant le problème de pseudo-mouvement. Ce problème vient du fait que, quelle que soit la position de l'observateur, les deux images affichées sont les mêmes. Les objets placés en deçà ou au delà de l'écran sembleront se déplacer avec l'observateur.

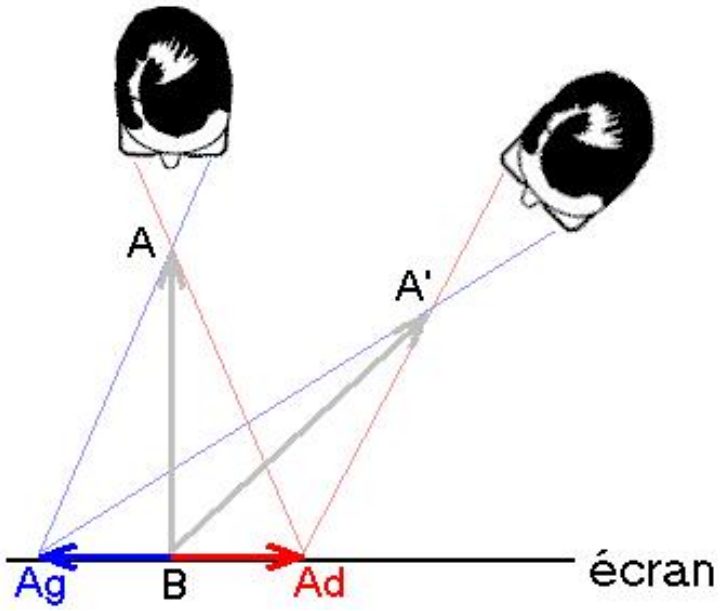


FIG. 1.14 – Problème de pseudo-mouvement

Dans l'exemple représenté sur la figure 1.14, l'image vue par l'œil gauche de l'observateur comporte une flèche allant du point B au point Ad, alors que celle du droit en comporte une similaire allant du point B au point Ag. Lorsqu'il se trouve en face du point B, l'observateur voit donc une flèche BA perpendiculaire à l'écran et pointée vers lui. Mais s'il se déplace, le point A de la flèche va se déplacer avec lui jusqu'au

point A'. Il verra alors une flèche dirigée vers lui, mais cette fois-ci, elle ne sera plus perpendiculaire à l'écran. Il aura alors l'impression que les objets affichés se déforment alors qu'il se déplace.

De la même manière, le rapport entre la longueur de la flèche et la distance entre l'observateur et le point B est constant. Par conséquent, en faisant varier la distance le séparant de l'écran, l'observateur déforme l'image en modifiant la profondeur des objets observés.

Le principe de l'autostéréoscopie

Les progrès techniques récents ont permis la mise au point de plusieurs techniques permettant de percevoir une image en relief, sans être obligé de porter de lunettes, ni de regarder directement dans les objectifs du dispositif.

Ces dispositifs se présentent sous la forme d'écrans permettant d'afficher directement des images 3D.

Le principe est sensiblement le même que pour les dispositifs à lunettes : les images destinées à chacun des deux yeux sont affichées sur le même écran, mais de telle sorte que chacune d'elle ne soit visible que dans une direction donnée. L'observateur doit donc se placer (voir figure 1.15) dans une position telle que chacun de ses deux yeux ne voit qu'une seule des deux images. L'effet est le même qu'avec des lunettes, mais l'observateur ne peut pas se déplacer par rapport à l'écran.

Beaucoup des systèmes d'affichage autostéréoscopique ne projettent pas qu'une seule fois les deux images. Cela permet de multiplier les positions d'où l'image en relief peut être perçue convenablement. Cependant, pour que chaque image soit entièrement perçue, il faut que les rayons lumineux qu'elle émet convergent dans les yeux de l'observateur. Ceci empêche l'observateur de trop se rapprocher ou s'éloigner de l'écran car, dans ce cas, il ne percevrait pas une image complète pour chaque œil.

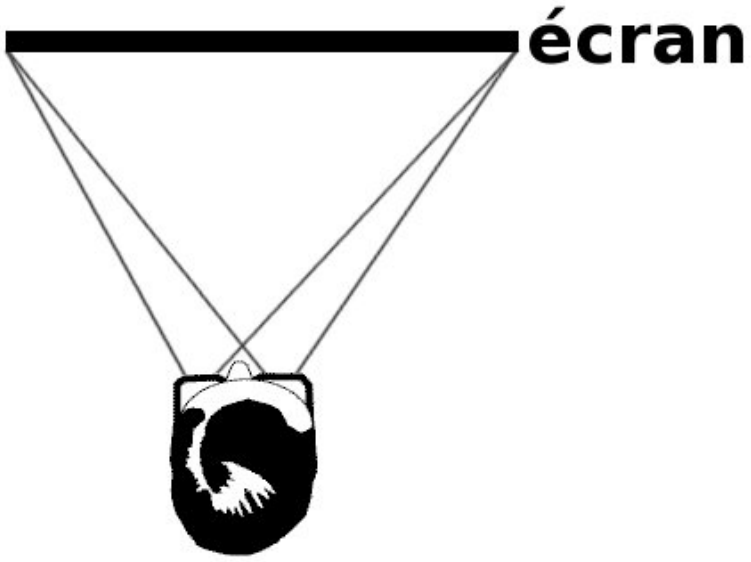


FIG. 1.15 – Principe de l'autostéréoscopie

Les images projetées étant toujours au nombre de deux, le problème de pseudo-mouvement reste présent pour ces dispositifs. Sauf pour certains appareils munis d'un dispositif de détection de position permettant à la fois de projeter les deux images exactement vers les yeux de l'observateur, mais aussi de recalculer ces images en fonction de sa position réelle par rapport à l'écran. Les principaux inconvénients de ce dernier procédé sont que l'observateur doit porter un dispositif permettant au système de déterminer sa position et que le nombre d'observateurs est limité.

1.1.3 L'autostéréoscopie multi-vues

Pour résoudre le problème de pseudo-mouvement et permettre aux observateurs de se déplacer par rapport aux objets affichés sans nécessiter de détecter leur position, certains procédés projettent plus de deux images. Ces images correspondent à différentes vues des objets à afficher selon plusieurs directions. Chacune de ces images est projetée dans la direction correspondante à partir de l'écran. La vue perçue par chaque oeil dépend donc de sa position par rapport au dispositif. En se déplaçant, l'observateur verra donc les objets affichés selon des angles différents, dépendant de sa position, comme cela se passe dans la réalité.

La figure 1.16 illustre un tel dispositif. Dans cet exemple, l'écran projette 5 images différentes qui sont visibles aux positions numérotées par les chiffres 1 à 5. Chaque paire d'images successives correspond donc à une position possible de la tête de l'observateur. Dans cette position, chacun de ses yeux verra une image différente donc une image en trois dimensions pourra être reconstituée. On obtient ainsi 4 positions possibles, notées A, B, C et D sur les dessins.

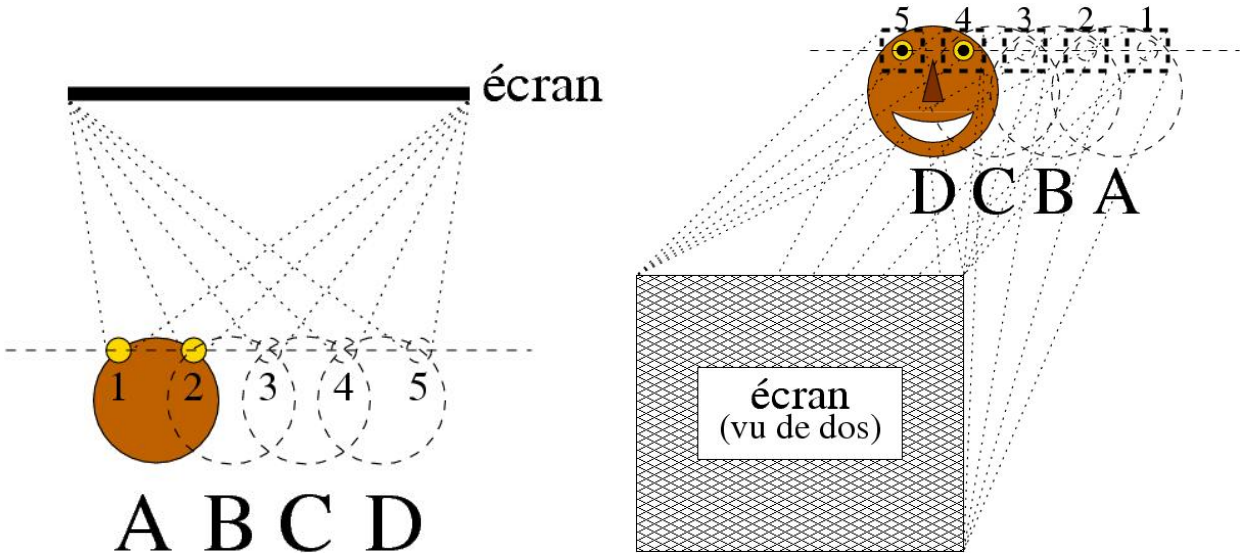


FIG. 1.16 – Principe de l'autostéréoscopie multi-vues

1.2 Principaux systèmes d'affichage 3D existants

1.2.1 Les systèmes stéréoscopiques

Les systèmes stéréoscopiques "directs"

Ce sont les plus anciens et les plus simples des systèmes stéréoscopiques. Le tout premier (représenté sur la figure 1.17) a été mis au point en 1838 par Charles Wheatstone [100] [101]. Il était simplement constitué de deux miroirs verticaux formant un angle de 90 degrés et de deux images. L'invention de la photographie étant postérieure à 1838, la première version de cet appareil comprenait des dessins géométriques qui ont ensuite été remplacés par des photographies. En plaçant son visage contre ces miroirs et en

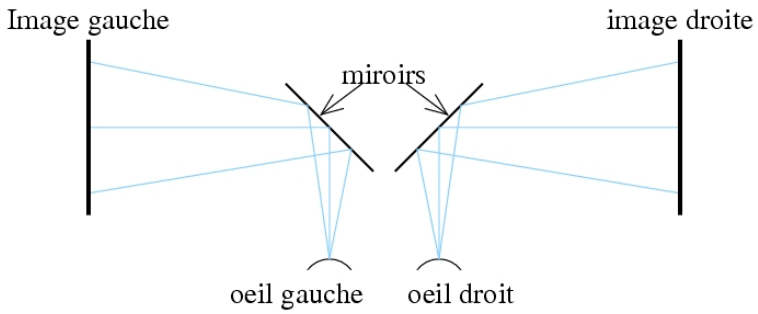
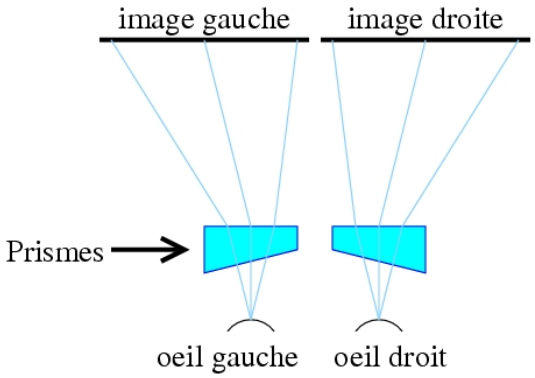


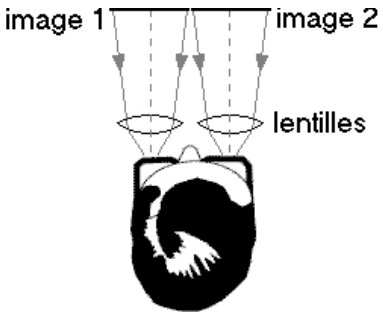
FIG. 1.17 – Principe du stéréogramme de C. Wheatstone (1838)

regardant dedans, un observateur peut voir les deux images, chacune vue seulement par l'œil qui regarde dans le miroir placé du même coté.

Plus tard, les miroirs ont été remplacés par des prismes [26] (voir figure 1.18-a) en 1856, puis par des lentilles (figure 1.18-b), afin de réduire l'encombrement des appareils. Cette dernière technique est actuellement encore largement utilisée, comme le montre la figure 1.19.



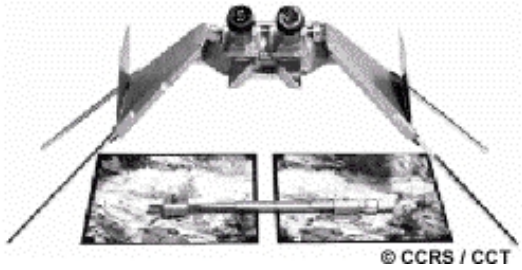
a - avec des prismes (Brewster - 1841)



b - avec des lentilles

FIG. 1.18 – Adaptations du stéréogramme de C. Wheatstone

Un des principaux défauts de ces systèmes est qu'ils ne peuvent être utilisés que par une seule personne à la fois. Des recherches ont donc été menées pour trouver un moyen d'utiliser le principe de la stéréoscopie pour plusieurs utilisateurs simultanément.



a - Visionneuse stéréoscopique

b - Casque de réalité virtuelle
(Head-Mounted Display ou HMD)

FIG. 1.19 – Exemples d'appareils actuels utilisant le principe du stéréogramme de C. Wheatstone

Les systèmes stéréoscopiques à filtres

Un tel moyen a été décrit en 1853 par Wilhelm Rollman : ce moyen consistait en une image composée de lignes bleues et rouges dessinées sur un fond noir, projetée sur un mur. Les observateurs, munis de lunettes aux verres colorés des mêmes couleurs (un verre rouge et l'autre bleu) ne percevaient, avec chaque œil , que les traits de même couleur que le verre derrière lequel il se trouve. Ainsi, l'œil derrière le verre rouge percevait les traits rouges, mais les traits bleus lui apparaissaient noirs et se fondaient avec le fond. De même pour l'autre œil , seules les lignes bleues étaient visibles.

En 1856, Joseph Charles d'Almeida améliora ce procédé en utilisant deux photographies qu'il superposait en les projetant au moyen de deux lanternes magiques (l'ancêtre du projecteur) munis de deux filtres bleu et rouge. Les observateurs munis des lunettes à verres colorés pouvaient ainsi voir la photographie en relief.

En 1856, Louis Ducos De Huron améliore cette technique en imprimant les deux images l'une sur l'autre sur un papier. Il appelle les images obtenues par ce procédé les anaglyphes. La figure 1.20 montre deux exemples d'anaglyphe.

(*a*) (*b*)

FIG. 1.20 – Deux exemples d'anaglyphes

Depuis, le procédé a été adapté au cinéma et amélioré, en particulier grace aux travaux de Louis Lumière qui améliora la colorimétrie des filtres en 1935. Il existe même des techniques pour créer des photographies en couleur qui peuvent être regardées comme des photographies "classiques" en couleur sans lunettes ou en relief avec les lunettes colorées. Ces photographies sont des photographies auxquelles on a rajouté des traits rouges ou bleus au bord des objets pour créer un léger effet de relief lorsqu'on les observe avec les lunettes, mais l'effet obtenu n'est pas réellement une mise en relief de l'image, elle ne fait que donner un léger volume aux objets de premier plan.

En 1950, Edwin-Herbert Land met au point un filtre polarisant, constitué de molécules organiques intégrées dans un film polyvinyle étiré pour les aligner. Ce filtre permet de ne laisser passer que les photons ayant un sens de vibration déterminé par l'orientation du filtre. Cette invention permit d'améliorer le principe des anaglyphes en séparant les images selon la polarisation de la lumière au lieu de la couleur. Il suffit de projeter les images destinées à chaque œil avec deux projecteurs séparés et synchronisés ou un projecteur spécial équipé de deux objectifs, sur le même écran, comme indiqué sur la figure

1.21. En plaçant un filtre polarisant devant chacun des deux objectifs, la lumière utilisée pour projeter chacune des images sera polarisée dans le même sens que le filtre. Si les deux filtres sont orientés dans deux directions perpendiculaires (représentées par le sens des hachures sur la figure 1.21) et que les observateurs portent devant les yeux des filtres alignés dans les mêmes directions, chaque œil ne verra que l'image projetée au travers du filtre polarisé dans la même direction. On obtient ainsi une vision stéréoscopique en couleur.

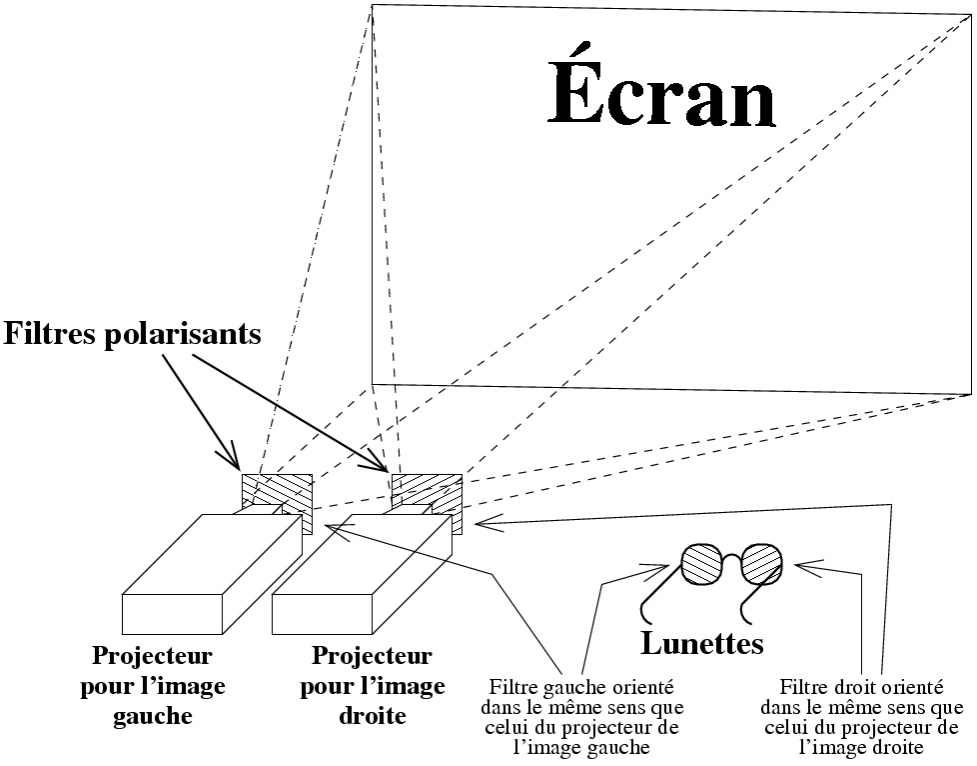


FIG. 1.21 – Principe d'utilisation des filtres polarisants pour l'affichage stéréoscopique

Ce procédé a été adapté aux écrans à cristaux liquides. En effet ces cristaux ont la propriété de faire tourner la lumière lorsqu'ils sont soumis à un champs électrique. Dans un écran classique, deux filtres orientés perpendiculairement l'un par rapport à l'autre sont placés de part et d'autre de la dalle LCD. Lorsque le rétro-éclairage est allumé, mais qu'aucune tension n'est appliquée, le premier filtre oriente la lumière émise dans un sens. Cette lumière n'étant pas modifiée, elle est stoppée par le second filtre et

on obtient un pixel éteint, c'est-à-dire noir. Lorsqu'une tension électrique est appliquée aux cristaux, ceux-ci font tourner la lumière polarisée qui traverse le premier filtre. Lorsqu'elle arrive au deuxième filtre, elle est donc polarisée dans le même sens et passe jusqu'à l'observateur. On a donc un pixel allumé.

Pour l'affichage 3D, on fabrique des écrans LCD qui utilisent cette propriété de polarisation de la lumière des écrans LCD pour séparer les images.

Le principe est que les filtres polarisants ne recouvrent pas la totalité de l'écran, mais seulement une ligne horizontale ou verticale. La ligne suivante sera polarisée dans l'autre sens. La troisième sera polarisée dans le même sens que la première et ainsi de suite. Lorsqu'un observateur regarde cet écran au travers de filtres polarisants, chacun de ses yeux ne verra donc qu'une ligne sur deux. Ce procédé peut aussi être adapté à d'autres types d'écrans (cathodiques, plasma...) et même à des projecteurs. Tous ces appareils sont commercialisés, entre autres, par la société VREX [19] en utilisant leur procédé qu'ils ont appelé μPol technology [9].

L'avantage de ce procédé est qu'il n'est plus besoin de se placer à certains endroit pour pouvoir observer l'image en relief. Le nombre d'observateurs n'est pas non plus limité. par contre, chaque observateur doit porter des lunettes avec filtres polarisants devant les yeux pour pouvoir voir chaque image avec l'œil pour lequel elle est destinée.

Les systèmes stéréoscopiques à lunettes à obturation

Une autre méthode simple à mettre en œuvre pour séparer les deux images et les faire percevoir par un seul œil consiste à faire un multiplexage temporel de ces deux images et de faire un démultiplexage au niveau des yeux de l'observateur. Cela revient à les afficher successivement sur le même écran et à placer devant les yeux de l'observateur un dispositif permettant de laisser la lumière atteindre l'œil pour lequel l'image est destinée et de l'empêcher d'atteindre l'autre. Puis de faire la même chose pour l'image suivante mais avec l'autre œil.

Ce procédé, mis au point par Laurens Hammond et William F. Cassidy, a été présenté au public en 1922. Il était constitué de deux projecteurs synchronisés afin de projeter les

images destinées aux deux yeux suffisamment rapidement et de visionneuses au travers desquelles les spectateurs regardaient l'écran. Ces visionneuses (représentées figure 1.22) étaient constituées d'une fenêtre d'observation et d'un obturateur rotatif qui obturait successivement la partie gauche (correspondant à l'œil gauche du spectateur) de cette fenêtre puis la droite (correspondant à l'œil droit) de manière synchronisée avec la projection des images destinées à chaque œil. Ainsi, chaque œil des spectateurs ne voyait que les images qui lui étaient destinées.

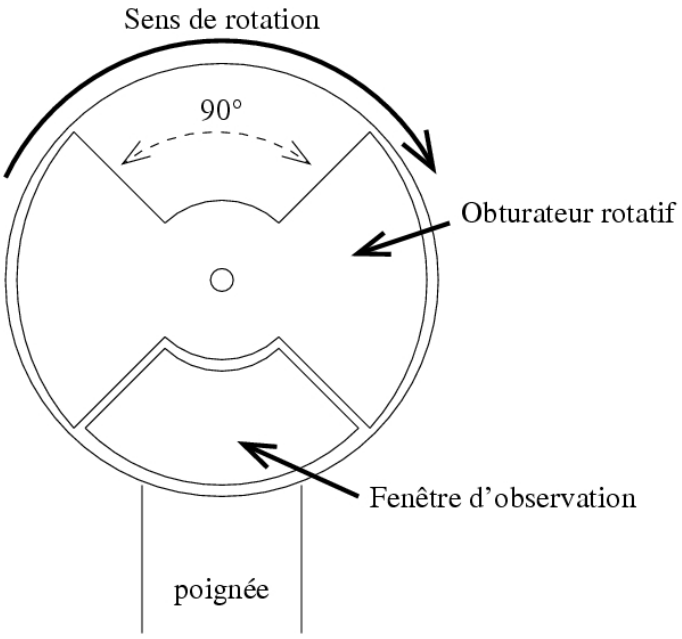


FIG. 1.22 – Principe des visionneuses "Televiewˮ

Ce procédé, entièrement mécanique, présentait l'inconvénient de se désynchroniser assez rapidement au cours de la projection. Avec l'apparition des obturateurs à cristaux liquides, ce problème a été corrigé et les visionneuses ont été remplacées par des lunettes à obturateurs électroniques plus légères et plus maniables, synchronisées par un signal électrique, radio ou infrarouge. Il a également été adapté à l'informatique avec la commercialisation de cartes accélératrices 3D capables d'envoyer un signal vidéo avec une fréquence de 60 images par secondes associées à des écrans vidéos capables d'afficher ces images et des lunettes à obturation synchronisées avec le signal vidéo au moyen d'un signal électrique ou infrarouge.

Le procédé chromadepth^{TM} ou chromostéréoscopie

Une autre méthode pour séparer les images en fonction de leur couleur a été inventée par Richard Steenblik en 1986 et commercialisée par l'entreprise Chromatek Inc. [32] en 1991. Il s'agit du procédé chromadepth^{TM}.

Ce procédé est fondé sur la chromostéréoscopie, ou stéréoscopie par les couleurs, effet décrit pour la première fois par Willem Einthoven (1860 - 1927) au cours de sa thèse de doctorat en médecine intitulée "Stereoscopie door kleurverschil" (Stéréoscopie par variation de couleur), soutenue en 1885. Cet effet est dû au fait que l'axe visuel, c'est-à-dire la direction vers laquelle on regarde un objet, et l'axe optique, c'est-à-dire l'axe de la lentille formée par la cornée et le cristallin, sont différents. Les rayons lumineux qui créent une image sur la fovéa (voir figure 1.23) traversent donc la surface de la cornée selon un certain angle. Celle-ci et le cristallin agissent comme des prismes. Les rayons

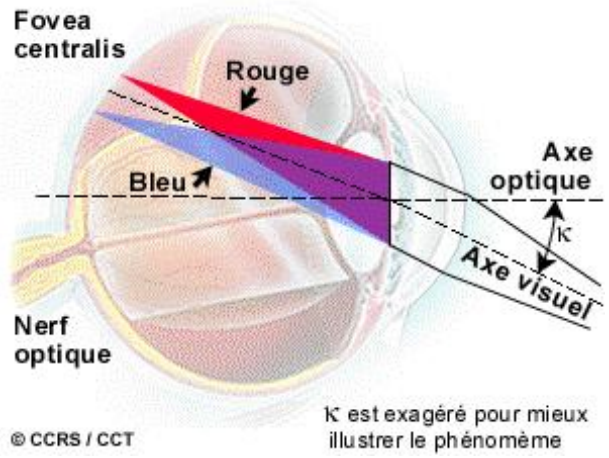


FIG. 1.23 – Diffraction des rayons lumineux à l'intérieur de l'œil (*Source : centre canadien de télédétection*)

lumineux sont donc réfractés différemment selon leur longueur d'onde, c'est-à-dire leur couleur. Les rayons de longueurs d'onde plus courtes (proches du bleu ou du violet) sont donc réfractés davantage que ceux de longueurs d'onde plus grandes (proche du rouge). Par conséquent, sur la rétine, la lumière bleue converge plus vers le nez, tandis que la lumière rouge converge plus vers les tempes. Un objet rouge semblera donc plus proche

qu'un objet bleu.

Le procédé chromadepth^{TM}(dont le principe est schématisé sur la figure 1.24) accentue cet effet en utilisant des lunettes agissant comme des prismes et combinant les effets de la réfraction et de la diffraction.

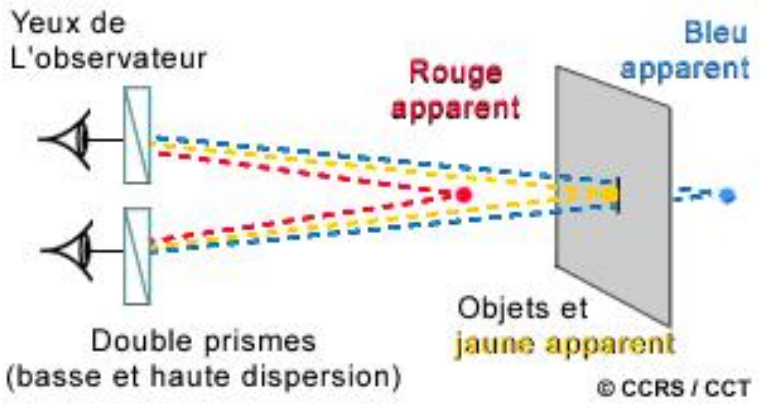
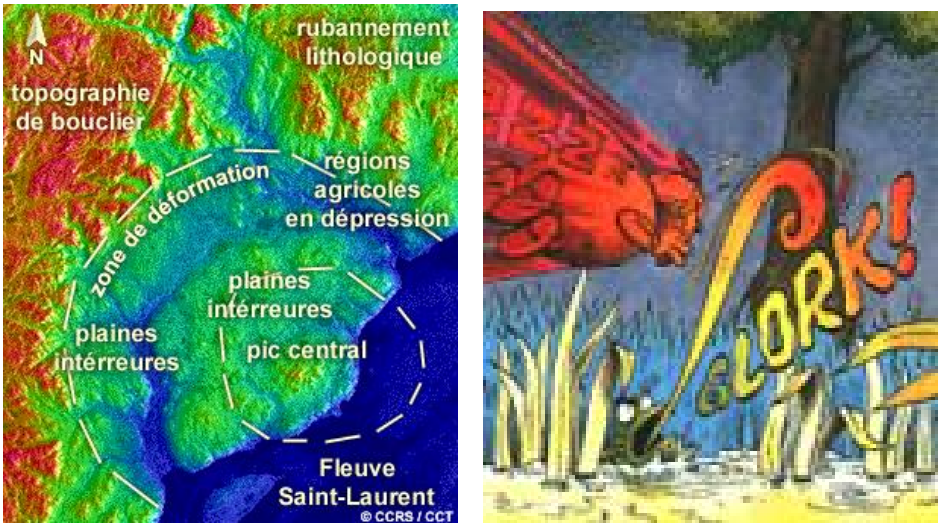


FIG. 1.24 – Principe du procédé chromadepth^{TM}(*Source : centre canadien de télédétection*)

Pour construire une image en relief avec ce procédé, il suffit de dessiner des objets en leur donnant une couleur en fonction de leur profondeur : les objets les plus lointains doivent avoir des couleurs proches du violet, alors que les objets proches des couleurs tirant plus vers le rouge. Les autres couleurs apparaissent entre les deux en fonction de leur position dans le spectre visible, c'est-à-dire leur position dans un arc-en-ciel : rouge, orange, jaune, vert, bleu, indigo et violet. En observant ces images au travers de deux prismes tournés dans le sens opposés, les objets proches sont plus décalés vers la gauche pour l'œil droit et vers la droite pour l'œil gauche que les objets lointains. Ils sont donc perçus plus proches et l'image apparait en relief. La figure 1.25 montre quelques exemples d'images utilisant ce procédé.

Ce procédé a l'avantage de produire une image visible à la fois avec et sans les lunettes. En effet, ces images se présentent sous la forme d'images parfaitement normales à l'unique différence que les couleurs indiquent la profondeur des objets. Ce genre d'images peut être utilisé pour toutes sortes de dessins ou des photos satellites [90] qui doivent être traitées en fausses couleurs.



(*a*) (*b*)

FIG. 1.25 – Exemples d'images utilisant le procédé chromadepth^{TM}

Avantages et inconvénients

Tous ces procédés permettent d'afficher des images en relief pour un ou plusieurs observateurs. Mais ils nécessitent tous un appareillage à mettre devant ses yeux, ce qui est contraignant pour l'utilisateur. De nombreuses recherches ont été menées pour pallier cet inconvénient. Les résultats de ces recherches ont abouti à différents systèmes d'affichage, en particulier, les systèmes à projection volumique, les systèmes holographiques et les systèmes d'affichage 3D autostéréoscopiques.

1.2.2 Les systèmes à projection volumique

Tous les systèmes que nous avons présentés jusqu'à présent sont basés sur le principe de la stéréoscopie, c'est-à-dire l'affichage simultané d'au moins deux images différentes du même objet, une pour chaque œil de l'observateur. Il existe cependant d'autres procédés permettant d'afficher des objets en relief, sans pour autant calculer les images correspondant à un certain nombre de points de vue différents et les afficher. L'un de ces procédés est la projection volumique.

Son principe est que chaque objet à afficher peut être assimilé à un certain nombre de voxels, c'est-à-dire de points dans l'espace. Chacun de ces points émet de la lumière

dans toutes les directions et c'est cette lumière qui nous permet de les voir. Ces appareils tentent donc de reconstituer cette lumière en faisant apparaître des points lumineux sur une surface en mouvement, soit en projetant des rayons lumineux sur une surface transparente dépolie, soit en utilisant une plaque recouverte de diodes électroluminescentes, ou encore en projetant des électrons sur une surface qui va réagir en émettant de la lumière, comme dans un tube à rayons cathodiques.

Cette surface se comporte alors comme un écran de télévision ou un écran sur lequel on projette une photographie : chaque point lumineux va émettre de la lumière dans toutes les directions. Si on déplace cette surface, on peut créer d'autres points situés à un autre endroit dans l'espace. En additionnant toutes les positions successives de cette surface, on obtient une succession de "coupes" qui constituent un volume à l'intérieur duquel on peut créer des points lumineux. Comme ces points lumineux peuvent se trouver n'importe où à l'intérieur de ce volume, on peut y reconstituer n'importe quel objet.

Les avantages de ces procédés sont qu'il n'est pas nécessaire de porter de filtres devant les yeux, que l'observateur peut se déplacer librement à 360° tout autour de l'appareil pour observer le ou les objets représentés et ce à n'importe quelle distance. Cependant, parceque les objets ne sont constituées que par des points lumineux, ils doivent être observés dans la pénombre pour obtenir une meilleur visibilité. Pour la même raison, ils sont forcément transparents. Il est donc impossible de cacher un objet derrière un autre. De plus, il est impossible de faire apparaître de surfaces réfléchissantes ou de représenter des effets de réfraction. Et les objets affichés ne peuvent dépasser du volume d'affichage du dispositif.

1.2.3 Les systèmes holographiques

Une autre méthode pour faire apparaître des objets en relief avec une bonne qualité est d'utiliser l'holographie. Cette technique a été inventée par Dennis Gabor (1900 - 1979) qui, en 1947, remarqua pour la première fois que si le spectre de diffraction d'un objet et l'information sur les phases pouvaient être enregistrés, l'image d'un objet pouvait être reconstruite par une illumination cohérente du spectre de diffraction enregistré

[46]. En 1958, le Russe Youri Denisyuk(1927 - 2006) échafauda le principe d'un hologramme visible en lumière blanche (holographie de réflexion) en utilisant les travaux de photographie en couleur naturelle non chimique (photographie interférentielle) de 1890 de Gabriel Lippmann [65] [66], mais il fallut attendre 1960 et l'arrivée du LASER mis au point par l'américain Théodore Maiman (né en 1827) pour pouvoir produire les premiers hologrammes. Ses travaux ont été publiés en 1962 [36] en russe et en 1963 [37] en anglais. Le premier hologramme a été créé en 1962 par Emmeth Leith et Juris Upatnieks [63] de l'université du Michigan. Cet hologramme était un hologramme par transmission, c'est-à-dire que les information étaient imprimées sur une surface transparente et que l'image en relief était obtenue en éclairant cette surface par l'arrière (par rapport à l'observateur) par un rayon LASER de même fréquence. En 1972, Lloyd Cross développa l'hologramme intégral en combinant l'hologramme de transmission visible en lumière blanche avec la cinématographie conventionnelle pour produire des images en trois dimensions avec mouvement.

Le principe de l'holographie utilise les interférences entre un rayon LASER réfléchi par l'objet à représenter et un autre rayon, synchronisé avec le premier, utilisé comme référence. Le champ d'interférences est imprimé sur une surface sensible et développé comme une photographie sur une plaque transparente. Lorsqu'on éclaire cette "photographie" avec un LASER (pour l'holographie par transmission) de même fréquence que celui utilisé pour la prise de vue, chaque point clair suffisamment petit diffracte la lumière qui va se comporter comme une source de lumière ponctuelle. Les lumières issues de toutes ces sources vont alors créer des interférences les unes avec les autres et reconstituer, dans l'espace situé devant la plaque photographique, le champ d'interférences qui a été créé lors de la prise de vue. Un observateur verra alors l'objet apparaître devant ses yeux et pourra se déplacer dans toutes les directions autour de lui.

Depuis, ce procédé a été adapté à l'infographie : grace à la puissance des ordinateurs, il est maintenant possible de calculer le champ d'interférence de plusieurs points dans

un environnement 3D virtuel. En imprimant une coupe du champ d'interférences (ce qu'on aurait obtenu si on avait placé une plaque photographique dans ce champ, comme lors de la prise de vue des hologrammes) sur un support transparent avec des points suffisamment petits pour obtenir la diffraction de la lumière, on obtient un hologramme d'un objet virtuel.

L'université du Texas travaille actuellement à un projet permettant de générer en temps réel des hologrammes [7] en utilisant la technologie DLP^{TM}[98] développée par la société Texas Instrument [97]. La technologie DLP, utilisée dans des vidéo-projecteurs, est constituée d'un module contenant des micro-miroirs de 1 μm^2, c'est-à-dire suffisamment petits pour créer des interférences les uns avec les autres et générer un hologramme. Ces micro-miroirs sont éclairés par un LASER dont la longueur d'onde correspond à celle

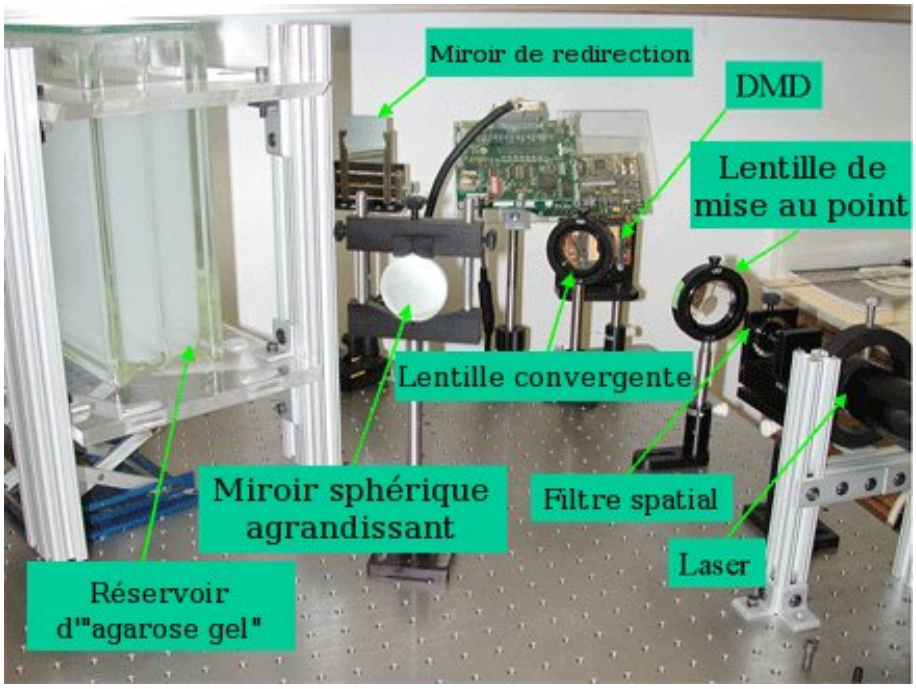


FIG. 1.26 – Prototype du dispositif d'affichage holographique développé par l'université du Texas (*Source : University of Texas Southwestern Medical Center at Dallas [7]*)

utilisée pour le calcul du champ d'interférences et la lumière réfléchie (qui forme un hologramme de la taille de la matrice de micro miroirs, c'est-à-dire quelques centimètres

carrés) est projetée dans une cuve transparente contenant un gel qu'ils appellent "Agarose gel" et dans laquelle les objets en relief vont apparaître. La figure 1.26 montre le prototype de ce dispositif d'affihage.

Les avantages de ce procédé sont que les images obtenues sont visibles depuis une multitude de directions différentes et que les effets tels que la réflexion et la réfraction sont plus facilement reproductibles.

L'inconvénient majeur reste la taille du réservoir et l'utilisation d'un LASER, qui n'est pas très bon pour les yeux et oblige à certaines précautions d'emploi.

1.2.4 Les systèmes autostéréoscopiques sans détection de position

Les systèmes d'affichage autostéréoscopiques permettent d'afficher des objets réels ou virtuels en relief, sans obliger l'observateur à porter un filtre devant les yeux. S'il existe des systèmes permettant de projeter les images directement dans les yeux d'un observateur après avoir détecté sa position par rapport au dispositif, la plupart projettent les images de telle sorte que celui-ci puisse voir en relief sans que sa position ne soit connue.

Systèmes autostéréoscopiques à réseau lenticulaire

Le principe de l'affichage 3D autostéréoscopique à réseau lenticulaire a été énoncé pour la première fois en 1908 par Gabriel Lippmann (1845 - 1921) sous le nom de photographie intégrale [67]. Il s'agissait d'une plaque photographique placée derrière une série de petites lentilles sphériques (le réseau lenticulaire, schématisé figure 1.27) placées côte-à-côte. Chaque lentille se comporte comme un petit appareil photographique qui photographie l'objet à partir d'une position donnée. Une fois la plaque photographique développée, on place la photographie derrière la même série de lentilles. Ainsi, chaque point de la photographie, correspondant à ce qui est vu à partir de la lentille dans une direction donnée projette l'image photographiée à partir du même point et dans la même direction. Un observateur regardant cette photographie munie des lentilles

FIG. 1.27 – Schéma des lentilles imaginées par G. Lippmann

percevra donc, avec chaque œil , l'objet photographié selon un angle correspondant à la position de cet œil . Il percevra donc l'objet en relief et aura même la possibilité de changer de position pour l'observer selon des angles de vue différents en tournant la photographie verticalement ou horizontalement.

Ce système présente plusieurs inconvénients majeurs :

- Les objets photographiés doivent se trouver à quelques centimètres du dispositif lors de la prise de vue et ne doivent pas dépasser la taille du film photographique.
- La résolution de l'image perçue dépend de la taille des lentilles, ce qui donne une image de résolution plus faible que le support photographique. En effet, comme chaque lentille se comporte comme un minuscule appareil photo, elle ne restitue, dans chaque direction, qu'un point de couleur. Elle sera donc perçue par l'observateur sous la forme d'un point de couleur uniforme, c'est-à-dire d'un pixel. Pour augmenter la résolution de cette image, il faudrait diminuer la taille des lentilles, ce qui pose des problème de fabrication (coût et qualité), de placement des lentilles sur la photographie, et de perte de résolution angulaire qui dépend du grain de l'émulsion photosensible et de la taille de la lentille. En effet, la surface sous chaque lentille doit enregistrer les informations de couleurs dans toutes les directions. Or cette surface a une résolution limitée (le grain) et ne peut donc pas enregistrer une quantité d'informations infinie. Le fait de réduire la taille des lentilles réduit

donc le nombre d'informations mémorisables pour chacune d'elle. Cette réduction se traduit par une diminution de la précision de l'image en terme d'angles d'observation.

– Le coût de fabrication d'un tel procédé reste assez élevé. C'est d'ailleurs vraisemblablement à cause de son prix que G. Lippmann n'en a jamais réalisé de prototype.

En 1931, Maurice Bonnet (1907 - 1994) améliora et simplifia ce système en remplaçant les lentilles sphériques par des lentilles cylindriques et en montrant comment photographier des objets plus grands en utilisant plusieurs photographies "classiques" projetées sur l'ensemble lentilles-plaque photographique, qu'il appela "film gauffré". La figure 1.28 représente ce 'film gauffré".

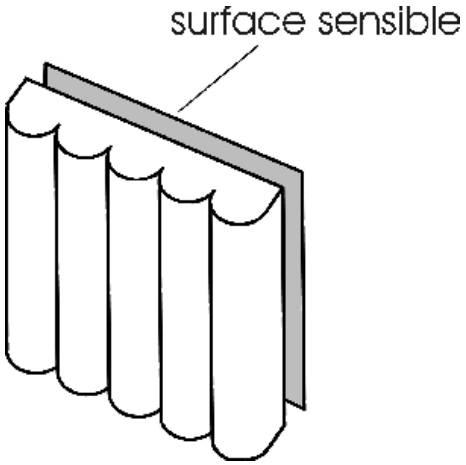


FIG. 1.28 – Schéma du "film gauffré" imaginé par M. Bonnet

La prise de vues se faisait en deux phases : la première (représentée figure 1.29) consistait à photographier l'objet ou la scène à partir de plusieurs points de vue, parfois espacés de plusieurs mètres, à l'aide d'appareils photo 2D "classiques", alignés sur la même ligne horizontale. Ensuite, on développait les négatifs des photos et on les projetait sur le film final (figure 1.30) équipé du réseau lenticulaire cylindrique, selon la même direction que lors de la prise de vue. Les lentilles permettaient de concentrer les rayons provenant d'une direction donnée, en une ligne verticale dont la position dépendait de l'angle par lequel les rayons arrivaient. Voir figure 1.31. Une fois le film développé et

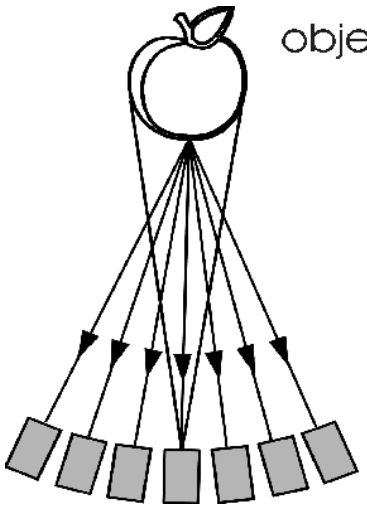


FIG. 1.29 – Prise de vues par le procédé de M. Bonnet

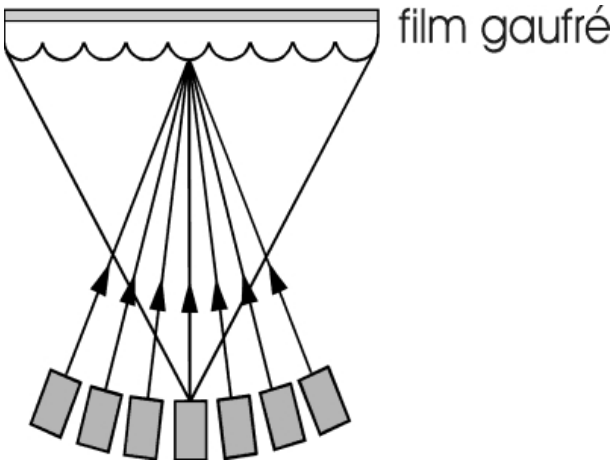


FIG. 1.30 – Impression du film gauffré (procédé Bonnet)

équipé du même réseau lenticulaire, un observateur pouvait observer l'objet photographié (figure 1.32) en relief, car les lentilles renvoient les images de chaque ligne verticale dans la direction correspondant à celle de leur provenance lors de l'impression du film sensible. Voir figure 1.33

Ce procédé n'est plus réellement une "photographie intégrale" dans le sens où les différents points de vues possibles de l'objet photographié ne sont répartis que dans la direction horizontale. L'observateur ne peut donc tourner la photographie pour voir des angles différents qu'horizontalement et plus verticalement. Ceci ne change rien à la

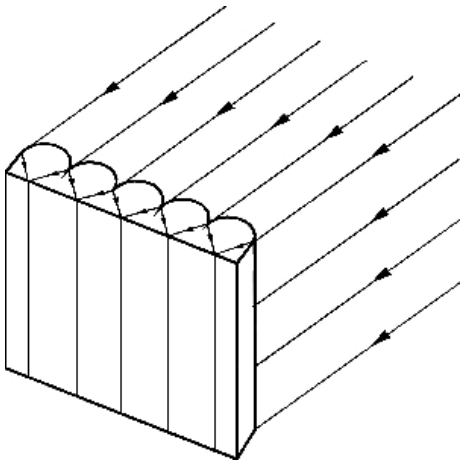


FIG. 1.31 – Détail du procédé d'impression du film gauffré (procédé Bonnet)

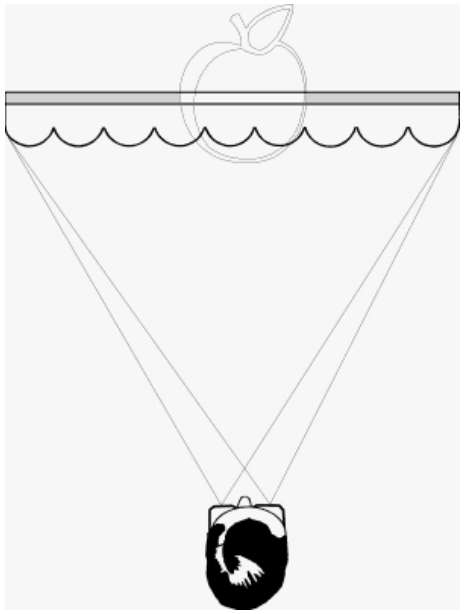


FIG. 1.32 – Visualisation de l'image en relief dans le procédé Bonnet

perception en relief de l'objet car celle-ci est due à la disparité entre ce qui est vu par chaque œil. Or, comme les yeux sont placés horizontalement, cette méthode suffit pour afficher n'importe quel objet en 3D.

Le nombre d'images (de photographies) différentes nécessaires pour mettre en œuvre ce procédé peut varier de deux jusqu'au maximum permis par la taille des lentilles et la résolution du support.

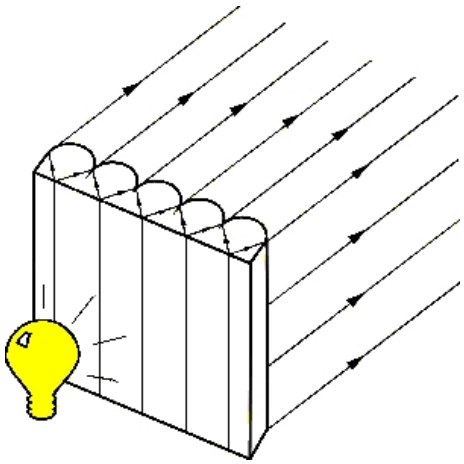


FIG. 1.33 – Restitution des rayons enregistrés (procédé Bonnet)

Cette méthode présente plusieurs avantages par rapport au procédé Lippman :

- Les objets n'ont plus besoin d'être placés près de la photographie et peuvent dépasser la taille de celle-ci car le procédé de photographie permet de les réduire. Ils n'apparaissent donc plus en taille réelle.
- La perte de résolution ne s'opère plus que dans une seule direction, perpendiculairement à l'axe des cylindres. L'image perçue aura donc un meilleur grain.
- Les lentilles cylindriques sont plus faciles à fabriquer. Elles sont donc moins chères à fabriquer et plus petites.
- Il est plus facile de prédire où vont se concentrer les rayons lumineux sur le film sensible. Il est donc aisé de fabriquer des images en 3D à partir de données 3D sans être obligé de photographier un objet réel.

Ces deux procédés, et en particulier le second, ont été largement étudiés, adaptés et améliorés depuis. Le procédé Bonnet a été adapté à la télévision et aux écrans d'ordinateur et de télévision en 1987 par Pierre Alliot, sous le nom d'alioscopie [1]. Ce procédé consiste à filmer la scène au travers d'une lentille (voir figure 1.34) permettant de filmer sous quatre angles légèrement différents (voir figure 1.35). Les images sont ensuite mélangées (figure 1.36) en intercalant une ligne de chacune des quatre images toutes les quatre colonnes de l'image finale. Puis on réarrange les sous-pixels (rouge, vert et

FIG. 1.34 – Lentilles utilisées pour l'alioscopie (*Source : Sciences et Avenir*)

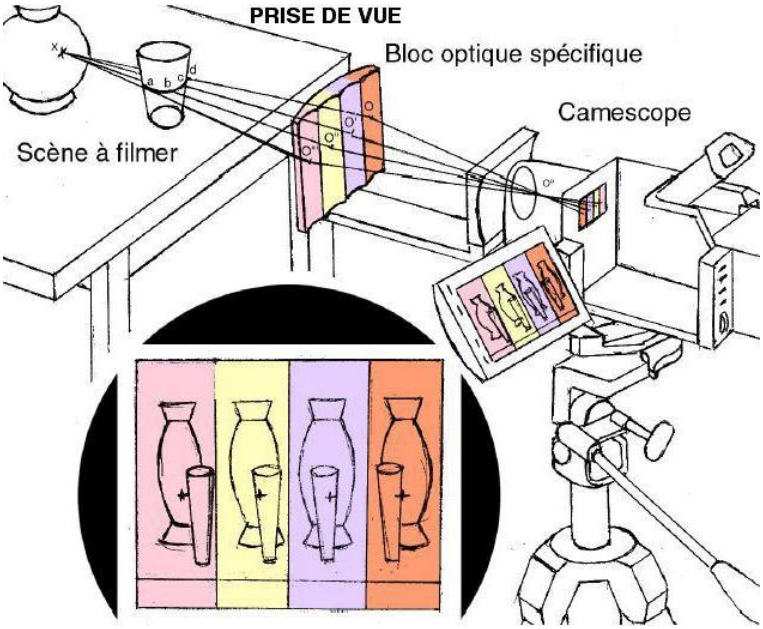


FIG. 1.35 – Prise de vues pour l'alioscopie (*Source : Sciences et Avenir*)

bleu) de chaque pixel (voir figure 1.37) de l'image finale, afin que les rayons issus soient projetés dans la bonne direction. Ainsi, l'écran final permet de projeter les quatre images

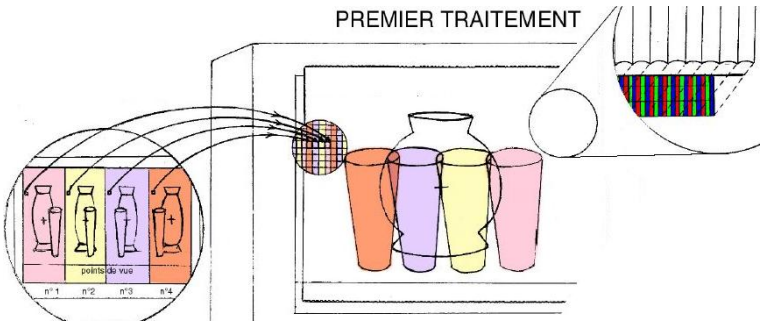


FIG. 1.36 – Mélange des images (*Source : Sciences et Avenir*)

selon une multitude d'angles différents (voir figure 1.38) permettant aux observateurs de percevoir la scène filmée en 3D selon plusieurs positions, et même de changer légèrement de point de vue en "dodelinant" de la tête. Ce changement de point de vue reste cependant très réduit et la distance et les positions des observateurs sont également limitées. Par exemple, si un observateur se place de telle sorte qu'il perçoit l'image n°4 avec son œil gauche et l'image n°1 avec son œil droit, il percevra la scène avec un relief inversé.

L'utilisation de ces procédés pour des images de synthèse est très simple : il suffit de calculer les différentes images correspondant aux différents points de vue et de les imprimer (sur un support papier ou électronique dans le cas de l'alioscopie) en les découpant et en les intercalant soit en points pour le procédé Lippmann, soit en colonnes pour les procédés Bonnet et Allio et de coller le réseau lenticulaire adéquat dessus.

Systèmes autostéréoscopiques à barrière parallaxe

Un autre procédé est commercialisé depuis quelques années par plusieurs fabricants, en particulier Sharp qui a été parmi les premiers. Il s'agit des affichages de type LCD à barrière parallaxe [41]. Ce sont des écrans à cristaux liquides classiques (ou LCD : Liquid Cristal Display) auxquels on a intercalé un cache entre la dalle LCD et le rétro-éclairage. Ce cache, dont le principe est représenté figure 1.39, est placé de telle sorte qu'un observateur placé à une certaine distance de l'écran ne percevra avec chaque œil qu'une colonne de l'affichage sur deux. Il suffit alors de découper les images destinées à

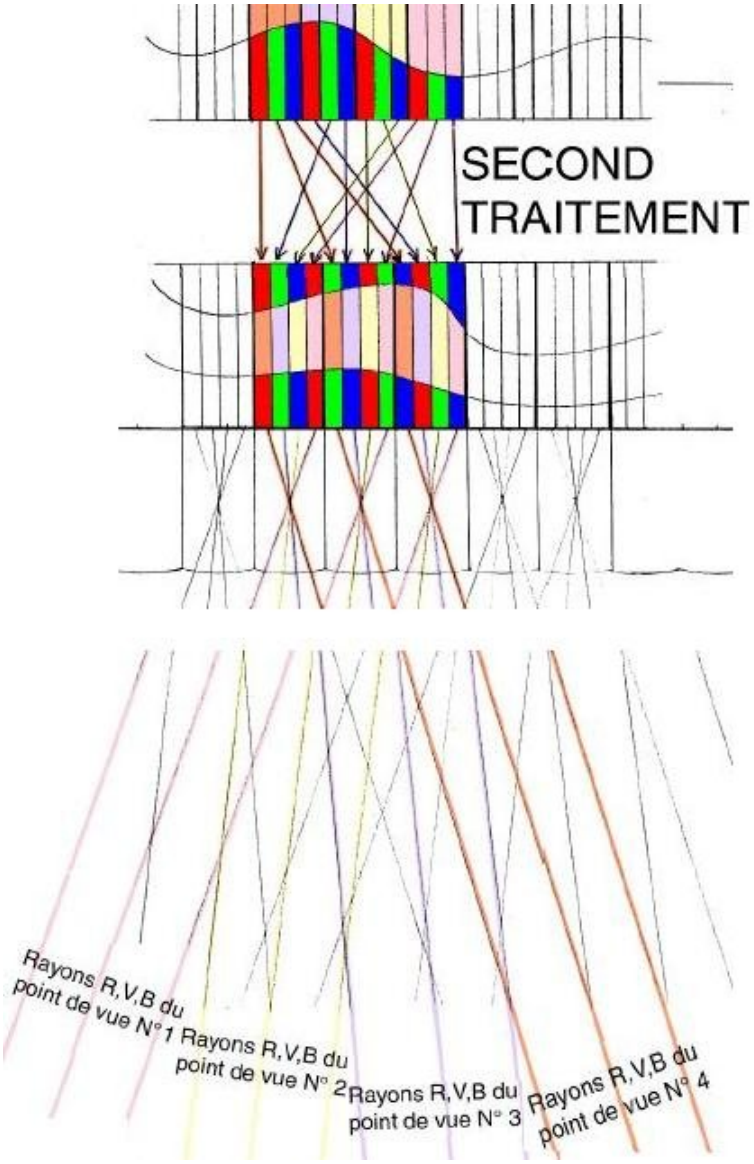


FIG. 1.37 – Réarrangement des sous-pixels (*Source : Sciences et Avenir*)

chaque œil en colonnes d'un pixel de large et de les afficher en intercalant les colonnes de chacune des images. Chaque œil ne percevant qu'une colonne sur deux verra donc une seule des deux images.

Systèmes autostéréoscopiques multi-vues

Un autre système d'affichage 3D est l'affichage autostéréoscopique multi-vues. Ce procédé consiste à projeter les différentes images, correspondant aux différentes vues

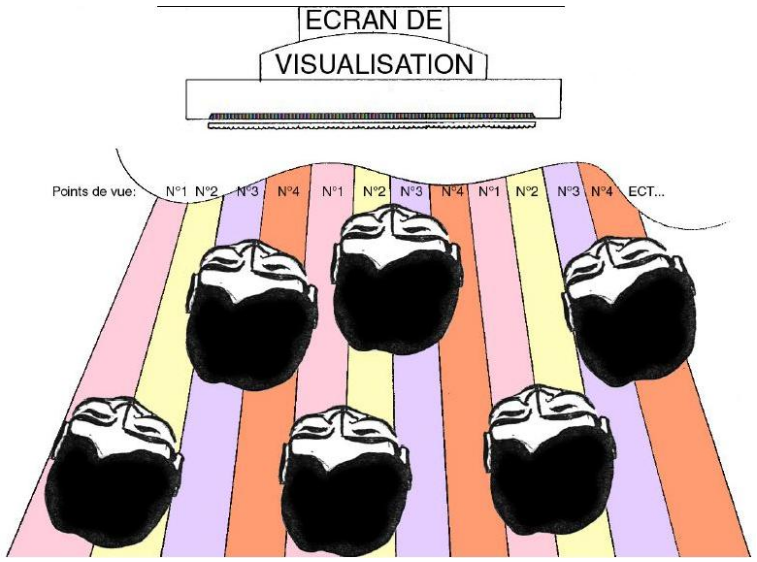


FIG. 1.38 – Restitution de la scène 3D par le procédé de l'alioscopie (*Source : Sciences et Avenir*)

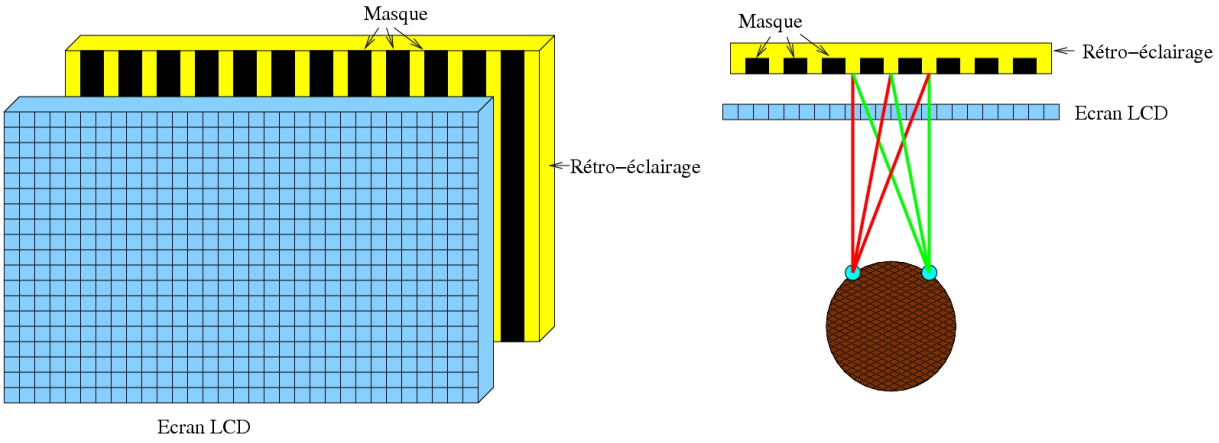


FIG. 1.39 – Principe des écrans LCD à barrière parallaxe

possibles, dans plusieurs directions différentes les unes après les autres.

Cette projection peut se faire soit en parallèle (figure 1.40), soit avec un multiplexage temporel [21], c'est-à-dire une image après l'autre. Ce multiplexage peut être obtenu soit avec un afficheur à transmission rapide (comme un écran LCD ferroélectrique, plus rapide qu'un écran LCD "classique", par exemple) en utilisant des sources de lumière (voir figure 1.41), soit un afficheur rapide (un tube cathodique capable d'afficher au

moins 100 images par seconde, par exemple) et un système de caches (voir figure 1.42), permettant de ne voir l'image affichée que dans une direction donnée. Un observateur

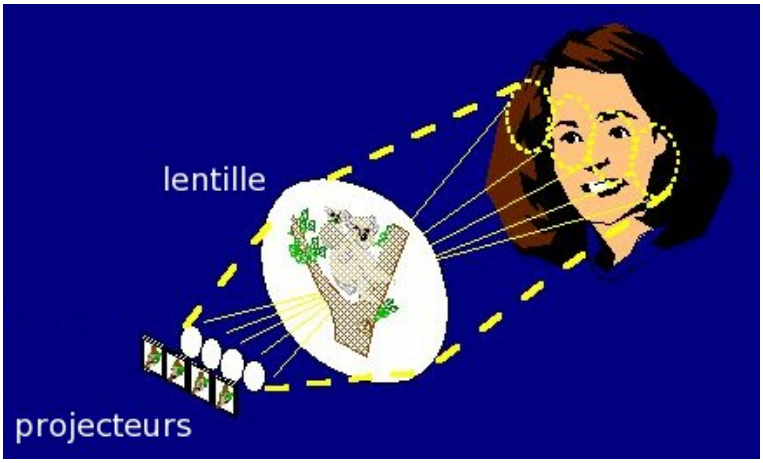


FIG. 1.40 – Utilisation d'afficheurs multiples pour l'affichage autostéréoscopique multi-vues (*Source : Cambridge University Department of Engineering Three dimensional television research [6]*)

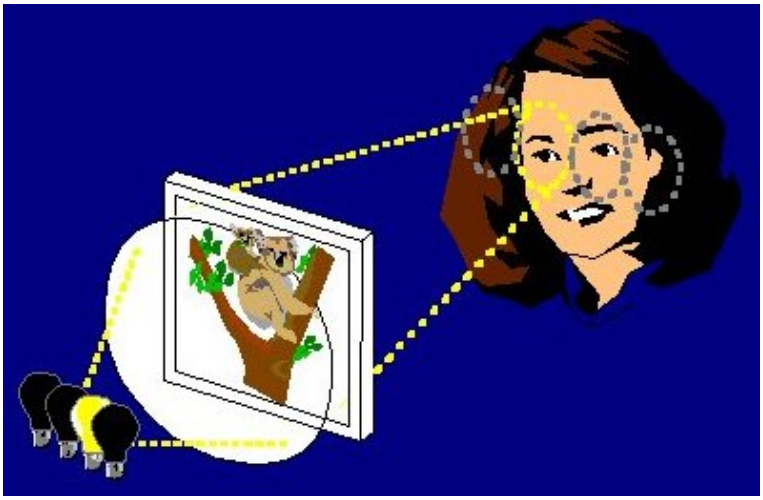


FIG. 1.41 – Principe de l'afficheur autostéréoscopique multi-vues avec un écran et plusieurs sources lumineuses (*Source : Cambridge University Department of Engineering Three dimensional telvision research [6]*)

qui regarde l'écran verra donc, avec chacun de ses deux yeux, l'image correspondant à la position de ceux-ci. Il aura donc la possibilité de se déplacer par rapport à l'objet

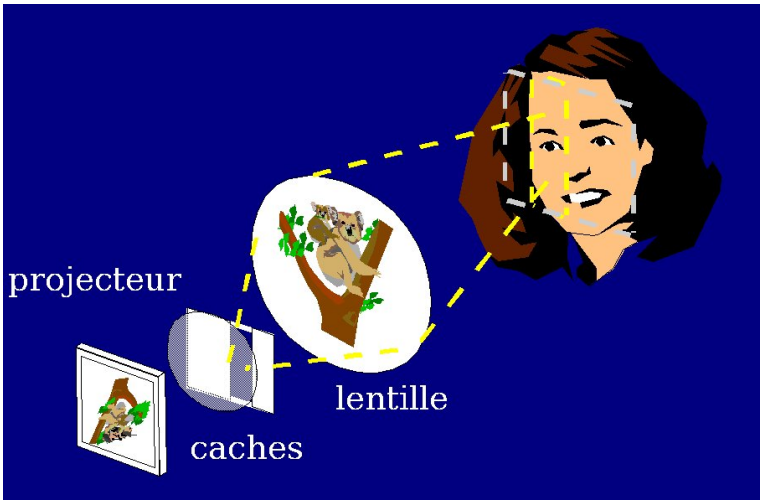


FIG. 1.42 – Utilisation d'un écran émissif et de caches pour l'affichage autostéréoscopique multi-vues (*Source : Cambridge University Department of Engineering Three dimensional television research [6]*)

en se déplaçant simplement par rapport à l'écran. Et ceci, sans avoir besoin de porter de lunettes, ni de fournir sa position au système, ce qui permet d'avoir un nombre d'observateurs presque illimité.

Ce type de procédé a été développé par l'université de Cambridge [73] en utilisant des caches à cristaux liquides et des tubes cathodiques rapides.

Avantages et inconvénients

Les procédés d'affichage autostéréoscopique sans détection de position sont parmi les plus simples à mettre en œuvre en informatique puisqu'il suffit de générer les deux (ou plus) images à afficher, puis d'intercaler leurs colonnes ou de les afficher sur plusieurs projecteurs ou successivement sur le même pour que l'affichage en relief puisse avoir lieu. Ils présentent également l'avantage d'être très pratiques pour l'observateur qui les utilise de manière très naturelle, sans avoir à porter de lunettes ou de dispositif de détection de position. Ils présentent néanmoins l'inconvénient d'être limités en résolution, sauf pour ceux utilisant un multiplexage temporel, car en augmentant le nombre de directions d'observation possibles, on réduit d'autant la résolution (au moins) horizontale des images perçues. Ceux utilisant un multiplexage temporel ont également l'inconvénient

de devoir traiter une grande quantité de données. De plus, la position de l'observateur, même si celui-ci peut se déplacer horizontalement est limitée à une distance donnée pour recevoir les images correctement.

1.2.5 Les systèmes à suivi de position

Un des moyens les plus simples pour augmenter le nombre de vues possibles sans pour autant réduire la résolution des images perçues ni augmenter la quantité des données à traiter, consiste à détecter la position de l'observateur par rapport à l'écran et d'afficher les deux images correspondant à la position de ses yeux.

Ce suivi peut être utilisé avec un dispositif d'affichage stéréoscopique pour afficher en relief. Dans ce cas, il ne sert qu'à générer des images différentes selon la position d'observation. C'est le principe des systèmes stéréoscopiques à suivi de position. Mais il peut également servir, dans le même temps, à projeter les images dans la bonne direction et ainsi éviter à l'observateur de porter des filtres et avoir ainsi les avantages d'un système d'affichage autostéréoscopique multi-vues. Ce principe est celui des systèmes autostéréoscopiques à suivi de position.

Les systèmes stéréoscopiques à suivi de position

Ces systèmes ne sont pas, à proprement parler, des systèmes d'affichage 3D, mais des améliorations de systèmes d'affichage 3D existant.

Ils consistent simplement à associer un système d'affichage 3D avec un dispositif permettant au système de déterminer la position de l'observateur dans l'espace et, par conséquent, de ses yeux. Les deux images affichées sont alors calculées en fonction de cette position pour permettre à celui-ci de se déplacer par rapport aux objets affichés en même temps qu'il se déplace devant le dispositif.

Les systèmes autostéréoscopiques à suivi de position

Le système de suivi de position peut également servir à projeter les images directement vers les yeux de l'observateur. Un tel système a été mis au point par un des laboratoires de recherche de la société Sharp[34]. Ce système est composé de deux écrans

affichant chacun une image et d'un dispositif, ressemblant à un miroir semi-réfléchissant monté sur un moteur, permettant de ne percevoir qu'une seule des deux images et de faire varier la position où cette séparation se fait.

Un dispositif permet de déterminer la position de la tête de l'observateur et modifie la position du miroir afin que la séparation entre les deux images passe au niveau de son nez. Ainsi, chaque œil ne perçoit qu'une seule des deux images et peut ainsi voir en relief.

De plus, la position de l'observateur étant connue, le système peut adapter les images affichées en fonction de celle-ci.

Inconvénients

Ces deux types de systèmes présentent cependant l'inconvénient de nécessiter une adaptation du logiciel de rendu, afin de lui permettre de rendre les images adéquates en fonction des informations fournies par le système détectant la position de l'observateur.

Le suivi de l'observateur nécessite également que celui-ci porte quelque chose permettant au dispositif de le localiser. Les systèmes actuels utilisent souvent une sorte de mire : un point doré ou de couleur collé sur le front, ou encore un petit émetteur d'ondes radios ou d'un signal infrarouge. Ce qui oblige l'observateur à s'en affubler. Cependant, la recherche actuelle sur la détection de visages montre qu'une simple caméra posée au dessus de l'écran peut suffire.

L'autre problème majeur vient du fait qu'il est difficile de détecter simultanément plusieurs observateurs et d'afficher les images correspondantes pour chacun d'eux. Ceci est principalement dû au système d'affichage 3D qui n'est pas conçu pour afficher plus de deux images simultanément. La plupart de ces systèmes ne permettent qu'un seul observateur à la fois. Certains peuvent aller jusqu'à deux, mais pas au delà.

1.3 Principes et avantages de notre système autostéréoscopique multi-vues

Parmi les systèmes d'affichage 3D que nous avons présentés, nous nous sommes particulièrement intéressé, dans cette thèse, aux systèmes autostéréoscopiques sans suivi de position. Ce sont ceux qui nous paraissent les plus avantageux dans le cadre d'une utilisation informatique et présentant les meilleurs résultats visuels pour un minimum de contraintes pour l'observateur.

Leur principe est d'afficher un certain nombre d'images (au maximum 20, actuellement [40]) et de les projeter dans différentes directions. Un observateur perçoit donc une image différente en fonction de sa position par rapport au dispositif. Il a, par conséquent, la possibilité de se déplacer par rapport aux objets affichés pour les observer selon des directions différentes en se déplaçant simplement devant l'écran, sans aucune interaction avec le logiciel, ni le dispositif d'affichage.

Cependant, les systèmes existants [73] restent axés sur la reconstitution du phénomène de stéréoscopie, c'est-à-dire qu'ils calculent un certain nombre d'images différentes pour les faire percevoir entièrement par l'utilisateur. Ceci oblige le dispositif à faire converger tous les rayons lumineux constituant une image vers un point où doit se situer un des yeux de l'observateur. Celui-ci doit donc se trouver dans une des positions permettant la visualisation de ces images, ce qui limite sa liberté de mouvement. En particulier, la distance à respecter entre l'observateur et le dispositif doit, pour les dispositifs que nous avons présentés, rester constante. En cas de modification de cette distance, les images perçues ne seront pas complètes et les objets affichés ne seront pas perçus correctement.

Le système que nous proposons permet de corriger ce défaut en permettant à l'observateur de modifier sa distance à l'écran. De plus, ce qui sera perçu de la scène affichée changera en fonction de cette distance : la largeur du champ se réduira proportionnellement à la distance d'observation, comme si la scène affichée était réellement présente derrière l'écran. Par exemple, sur la figure 1.43, lorsque l'observateur se trouve à une distance $D1$ de l'écran, son champ de vision dans la scène virtuelle forme un angle α. Cet angle α est déterminé par la formule : $\alpha = \arctan(\frac{largeur\ écran}{D1})$. S'il recule jusqu'à une

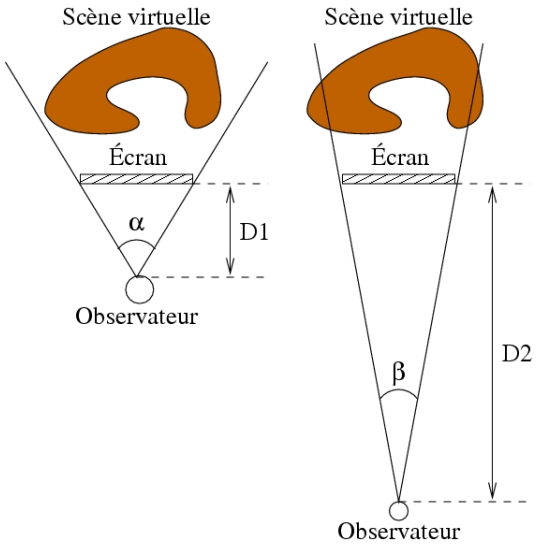


FIG. 1.43 – Variation de la largeur du champ de vision en fonction de la distance de l'observateur à l'écran

distance $D2$ ($D2 > D1$), l'angle formé par son champ de vision se réduit à β, déterminé par la formule : $\beta = \arctan(\frac{largeur\ écran}{D2})$. Par conséquent, si $D2 > D1$, alors $\beta < \alpha$.

Un principe similaire est déjà utilisé par le système HoloVizio^{TM}, commercialisé par la société hongroise Holografika[50]. L'inconvénient de ce dispositif est que les images sont générées et projetées par 64 vidéo-projecteurs dirigés vers un écran lenticulaire spécial, ce qui pose des problèmes de consommation (et de quantité de chaleur dissipée) et d'encombrement.

Notre dispositif permet d'obtenir les même résultats, sans ces deux problèmes. De plus, il permet à l'utilisateur de se déplacer horizontalement et verticalement, alors que le système HoloVizio ne permet de se déplacer qu'horizontalement, ou alors il faudrait rajouter une ou plusieurs rangées de 64 projecteurs, ce qui accentuerait les problèmes que nous venons de citer.

Notre dispositif et le principe d'affichage associé sont en passe d'être brevetés. Afin de ne pas compromettre la validité de ce brevet et afin de respecter la confidentialité devant précéder sa validation, nous ne pouvons pas ici donner plus de détails.

1.3.1 Terminologie

Pour simplifier les descriptions, nous utiliserons, à partir de maintenant, les termes et définitions suivantes :

- **dispositif d'affichage 3D**, aussi appelé **dispositif d'affichage** ou simplement **dispositif** est l'appareil que nous décrivons ici.
- **écran** est la surface du **dispositif d'affichage** émettant les rayons lumineux dans les différentes directions.
- **pixel** est un point sur l'**écran**, d'où sortent les faisceaux lumineux dans les différentes directions.
- **voxel** est un élément atomique (un point de couleur) en 3D.
- **scène**, **scène 3D**, **objet** ou **objet 3D** (virtuel, à afficher) représente ce qui est affiché par le **dispositif**. Ces objets sont définis pas les coordonnées de points dans un environnement 3D virtuel qui sont ensuite projectés sur l'écran pour former le dessin qui les représente. Il peut s'agir d'une simple forme géométrique, d'un objet (au sens commun : un verre, une table ...) 3D plus ou moins complexe ou d'une scène 3D complexe composée de plusieurs millions de **voxels**.
- **vue** est une image plane (2D) représentant l'**objet à afficher** selon un angle de vue particulier.
- **image 3D** est l'ensemble des **vues** du même objet selon tous les angles de vues permis par le **système d'affichage**.
- **chaîne de traitement** est le dispositif électronique permettant de générer les différentes **vues** d'un **objet**.
- **zone de projection** est la zone dans laquelle la scène 3D affichée par le dispositif est visible en relief par un observateur. En dehors de cette zone, un observateur ne percevra plus la scène affichée ou la percevra de manière déformée (sans relief ou en partie seulement).
- **rasteriser** ou **rasterisation** sont la francisation des mots anglais *"rasterize"* et *"rasterization"* qui représentent la transformation d'un objet virtuel en voxels ou en pixels en vue de son affichage.

- **vertex**, au pluriel **vertices**, est un point géométrique dans un environnement 3D. Ce sont les coordonnées d'un point d'un objet avant que celui-ci ne soit **rasterisé**, c'est-à-dire transformé en voxels.
- **DirectX** et **OpenGL** sont deux APIs (Advance Programmation Interface ou inteface de programmation avancée) permettant d'accéder à l'accélération matérielle des cartes accéléatrices 3D. DirectX a été développée par Microsoft pour les plateformes Windows. OpenGL a été développé par Silicon Graphics (SGI) et fonctionne pour de multiples plateformes (y compris Windows et Linux). En réalité, DirectX est une série d'APIs permettant d'accéder aux graphiques 2D, à la gestion des sons, etc. Seule une partie de DirectX, nommée Direct3D permet d'accéder à l'accélération 3D.
- **point d'observation virtuel** est le point utilisé comme point d'origine pour la projection des objets sur l'écran, correspondant au point d'observation pour la vue centrale, c'est-à-dire celle projetée perpendiculairement à l'écran.
- **référentiel de la caméra** ou **référentiel de l'observateur** est le référentiel où on place les objets et dans lequel on calcule leurs coordonnées afin de les projeter sur l'écran. Ce référentiel a comme origine le point d'observation virtuel, les axes X et Y parallèles aux bords de l'écran et l'axe Z dans la direction dans laquelle l'observateur regarde.

1.3.2 Avantages

Les avantages de ce procédé sont multiples :

- Les images perçues par les observateurs peuvent être très proches de ce qu'ils verraient dans la réalité et peuvent même être des photographies.
- Les observateurs peuvent se déplacer naturellement autour de l'objet pour l'observer de plusieurs points de vue différents et même de distances différentes, et ceci sans utilisation de lunettes ou d'interaction avec le système comme le head-tracking. Ce déplacement modifie l'aspect des objets affichés en fonction de leur position, mais également de leur distance : un objet prendra une proportion plus

importante de la surface de l'écran à mesure que l'observateur s'éloigne (comme indiqué figure 1.43), comme c'est le cas si on regarde au travers d'un cadre fixe.
- Les objets affichés peuvent être opaques ou transparents, quel que soit l'angle de vue.
- Il est facile de calculer les informations (couleurs des rayons) à afficher car elles sont obtenues à partir d'algorithmes de calcul d'images 3D connus et accélérables matériellement, comme le ray-tracing ou les algorithmes de projection utilisés dans les cartes accélératrices 3D.
- Les phénomènes compliqués comme la réflexion, l'éclairage spéculaire, la réfraction ou le brouillard peuvent être reconstitués.
- Les rayons lumineux utilisés n'ont pas besoin d'être issus de rayons LASER.
- Il est possible de modifier logiciellement la largeur angulaire de la zone d'observation possible, voire de la diviser en plusieurs zones indépendantes dans lesquelles les observateurs pourront voir des choses complètement différentes.

1.3.3 Inconvénients

Ce procédé comporte cependant un inconvénient majeur : la grande quantité de rayons à calculer oblige à avoir une puissance de calculs et une quantité d'informations à mémoriser importantes. Par exemple, si on considère un écran ayant une résolution de 1 024 × 768 pixels, et que l'on veut pouvoir afficher 200 vues différentes des objets, le nombre de rayons à calculer pour chaque image 3D à afficher sera de l'ordre de 1 024 × 768 × 200 = 157 286 400 pixels. Quand à la taille nécessaire pour mémoriser ces pixels, si on considère qu'un pixel est codé en couleurs "vraies", c'est-à-dire 24 bits, elle est de l'ordre de 1 024 × 768 × 3 × 200 = 471 859 200 octets, soit 450 méga-octets.

La quantité de pixels à calculer a été réduite grace à un algorithme (décrit dans le chapitre 2 : **Algorithme de génération et compression temps-réel des données 3D**) permettant de générer tous les pixels avec une seule projection des objets et de mémoriser toutes les données nécessaires à l'affichage de ces pixels en une taille réduite (de l'ordre de quelques méga-octets) et de les restituer ensuite.

1.4 Validation du principe

1.4.1 Simulations optiques

Cette méthode d'affichage, fondée sur une amélioration du principe de l'autostéréoscopie et qui reconstitue les rayons lumineux issus des objets affichés et passant par l'écran, a été validée par une série de tests optiques réalisée grace à un logiciel de ray-tracing modifié que nous avons écrit. Ce logiciel est un logiciel de ray-tracing auquel nous avons ajouté un objet qui simule optiquement notre dispositif en simulant des rayons lumineux projetés dans des directions choisies. Nous faisons ensuite se déplacer la caméra devant cet objet pour simuler ce que l'observateur verrait à ces différentes positions.

1.4.2 Vitesses de traitements nécessaires

La quantité d'informations à traiter dépend, bien entendu, de la résolution spatiale de l'écran, du nombre de rayons à projeter pour chaque pixel et de la vitesse de rafraîchissement de ces rayons. Si on considère un écran ayant une résolution de 1 024 $\times$ 768, avec 200 vues et un taux de rafraîchissement de 30 images par seconde, on obtient une vitesse de génération des rayons de $1024 \times 768 \times 200 \times 30 = 4\ 718\ 592\ 000$ pixels à générer par seconde. Si on considère que chacun de ces rayons est codé sur 24 bits, cela correspond à une vitesse de transfert des données vers le dispositif d'affichage de $4\ 718\ 592\ 000 \times 3 = 14\ 155\ 776\ 000$ octets par seconde. Ce qui correspond à un débit d'environ 14 Giga-octets par seconde. Ce débit peut être atteint par l'utilisation du parallélisme, mais reste dans le domaine de ce qui est faisable avec la technologie actuelle. Par exemple, le taux de transfert des données entre un GPU (Graphics Processing Unit) d'une carte accélératrice 3D et la mémoire de cette carte peut atteindre 35 Giga-octets par seconde, ce qui est largement suffisant dans ce cas.

1.5 Conclusion du chapitre

Nous avons présenté, dans ce chapitre, une nouvelle méthode pour afficher les objets en trois dimensions, sans avoir recours à des lunettes, ni à une interaction avec le système

et permettant à plusieurs observateurs simultanés de percevoir ces objets selon leur propre point de vue et de changer ce point de vue en se déplaçant simplement devant le dispositif, y compris à des distances différentes. Les tests et simulations que nous avons réalisés montrent que ce dispositif fonctionne et que l'effet obtenu correspond à ce qui était attendu. Ce dispositif étant en passe d'être breveté, il ne nous est pas possible de décrire avec plus de précision la théorie, ni les techniques prévues pour réaliser ce dispositif.

Les inconvénients de ce type de dispositif d'affichage existent. Ils ont été étudiés et des solutions ont été proposées et validées. Ces solutions sont présentées dans les chapitres suivants.

Chapitre 2

Algorithme de génération et compression temps-réel des données 3D

La mise au point d'un dispositif utilisant le principe que nous venons de décrire, l'autostéréoscopie multi-vues sans convergence, pose un certain nombre de problèmes. Outre les problèmes optiques et le balayage des différentes directions, la quantité de calculs et de données générées pose également problème. Ce problème se pose également pour tous les systèmes autostéréoscopiques "classiques" (avec convergence) multi-vues. En effet, lorsque le nombre de vues augmente, le nombre de calculs et la quantité de données générées augmentent en conséquence.

Pour pouvoir générer toutes ces vues et les fournir au dispositif d'affichage, il est nécessaire de mettre au point une chaîne de traitements (représentée figure 2.1) dont le rôle est de projeter les objets à afficher, fournis par l'ordinateur hôte et de générer les différentes vues au dispositif d'affichage. Cette chaîne se comporte, du point de vue de

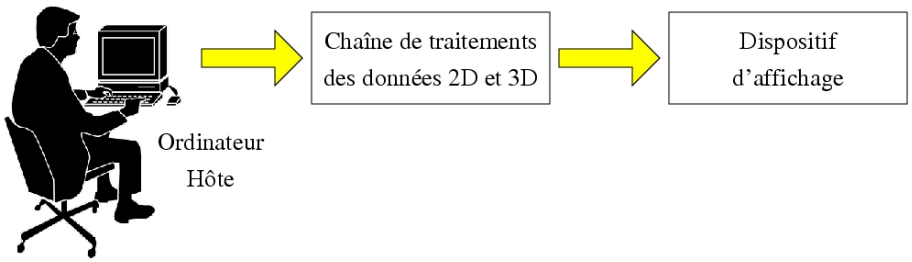


FIG. 2.1 – Schéma global de la chaîne de traitements

l'ordinateur hôte, comme une carte graphique "classique" : elle récupère les informations 2D et 3D à afficher, les traite et les envoie au dispositif d'affichage.

Ce chapitre décrit tout d'abord les problèmes inhérents à la génération de ces différentes vues et à leur mémorisation. Ensuite, nous y décrivons les différents algorithmes de compression des données 3D existants, puis les solutions logicielles que nous avons proposées et validées, et leurs résultats. Enfin, nous décrivons une chaîne de traitements implémentant l'ensemble de ces algorithmes permettant de générer les différentes vues tout en respectant les contraintes de place (moins de 1 giga-octet de données pour l'ensemble des vues) et de temps (génération de toutes les vues en moins de $\frac{1}{30}$ de seconde) nécessaire à l'utilisation temps-réel du dispositif d'affichage.

2.1 Problématiques

Comme nous l'avons vu précédemment, le procédé d'affichage multi-vues sans convergence ne peut fonctionner qu'avec un nombre de vues différentes important (au moins une centaine de vues différentes). Il est donc nécessaire de générer et de mémoriser ces différentes vues à une vitesse suffisamment importante pour permettre la visualisation. Si nous posons comme vitesse de rafraîchissement 30 images par seconde, comme c'est le cas pour la plupart des systèmes d'affichage actuels, il est nécessaire de pouvoir générer 30 rayons lumineux par seconde pour chacune des directions possibles pour le système d'affichage.

Il est donc nécessaire d'afficher au moins 30 vues dans chacune des directions possibles. Si on considère un système d'affichage permettant d'afficher des vues dans 200 directions différentes, on arrive à une vitesse d'affichage de $30 \times 200 = 6\ 000$ vues par seconde.

Ceci pose plusieurs problèmes : tout d'abord, le temps nécessaire à la génération de ces 6 000 vues, ensuite la taille de la mémoire nécessaire pour mémoriser les données qui les composent, et enfin la vitesse de débit nécessaire pour transférer ces données au dispositif d'affichage.

Génération des vues

Le calcul de l'ensemble des vues pose un problème de temps d'exécution, pour rester dans le cadre de 30 images par seconde. En effet, chaque vue doit être calculée comme une image de synthèse "normale", grâce à la projection des objets virtuels sur un plan, en utilisant des algorithmes de projection, rasterisation, plaquage de textures, etc. La seule différence réside dans les paramètres de projection, en particulier la largeur de la zone de visualisation et la direction d'observation.

Si on veut générer toutes les vues en les projetant indépendamment, il faut donc pouvoir calculer au moins 6 000 vues par seconde. Les cartes accélératrices 3D les plus rapides du commerce n'atteignent pas cette vitesse. À titre de comparaison, la carte accélératrice 3D *NVidia GeForce 7900 GTX SLI* qui est une des cartes accélératrices 3D les plus puissantes du commerce, actuellement, atteint des taux de rafraîchissement de l'ordre de 120 images par seconde, avec le jeu vidéo *Quake4*, utilisant OpenGl et 160 avec le jeu *Battlefield2*, utilisant Direct3D, pour des images ayant une résolution de 1600 par 1200. Pour d'autres jeux plus gourmands en ressources, ce taux est plus bas. Par exemple, il descend à moins de 50 pour certaines scènes du jeu *Oblivion HDR*. Ces vitesses correspondent à des tests réalisés par le site internet FiringSquad (http ://www.firingsquad.com/). À de telles cadences, même une diminution de la résolution des images générées à 1 024 $\times$ 768 ne nous permet pas d'atteindre une vitesse de 6 000 images par seconde. Il faut donc trouver un moyen pour obtenir les différentes vues sans être obligé de toutes les calculer indépendamment.

Taille des données

En plus du temps nécessaire à calculer toutes ces données, celles-ci doivent être mémorisées avant d'être affichées. Si on considère que le dispositif d'affichage possède une résolution de 1 024 colonnes et 768 lignes, ce qui correspond à la résolution minimale pour un ordinateur domestique actuel, le nombre de pixels de cet écran est de 1 024 $\times$ 768 = 786 432, pour chaque vue. Pour pouvoir mémoriser les 200 vues différentes à afficher, le nombre de pixels passe à 786 432 $\times$ 200 = 157 286 400. En considérant que chacun

de ces pixels est codé sur 24 bits, c'est-à-dire 3 octets, ce qui est la norme actuellement, la taille nécessaire passe à 157 286 400 × 3 = 471 859 200 octets, c'est-à-dire plus de 450 méga-octets. Cette taille est trop importante pour qu'une telle quantité de données soit utilisée telle quelle dans une application temps réel, surtout si on considère qu'une image 2D ayant la même résolution ne prend que 2 359 296 octets, soit un peu plus de 2 méga-octets.

Vitesse de transfert des données

L'envoi d'une telle quantité de données pose également un problème de débit pour permettre de les envoyer de la chaîne de traitements (voir figure 2.1) au système d'affichage. En effet, ces 450 méga-octets de données représentent seulement les images affichées en $\frac{1}{30}$ de seconde. Les données nécessaires à une seconde d'affichage représentent donc une taille de 471 859 200 × 30 = 14 155 776 000 octets, soit plus de 13 giga-octets. Ils doivent donc être envoyés au dispositif d'affichage en une seconde au maximum, les images de la seconde suivante devant être envoyées immédiatement après. Le débit doit donc être supérieur à 13 giga-octets par seconde, c'est-à-dire plus de 100 giga-bits par seconde. La connectique de plus en plus utilisée depuis quelques années pour connecter un moniteur numérique, le DisplayPort version 1.0 [16] (nouvelle norme de connexion vidéo développée par le groupe VESA - Video Electronics Standards Association [17], encore plus rapide que le DVI Dual Link et l'HDMI) possède un débit maximal de 10,8 giga-bits par seconde de débit typique, ce qui n'est pas suffisant. Même des connexions ultra haut débit comme les fibres optiques utilisées pour des transmissions transcontinentales ne dépassent pas 40 giga-bits par seconde de débit typique. De plus, une telle technologie est difficilement utilisable dans notre cadre d'utilisation.

2.2 Solution proposée

Pour résoudre ces trois problèmes, nous avons proposé un algorithme offrant une solution à chacun d'eux : il permet à la fois de réduire la quantité des données et par conséquent, le débit nécessaire pour envoyer ces données au dispositif d'affichage, mais

aussi permet de générer ces données en ne calculant la projection, rasterisation et autres, des objets qu'une seule fois pour générer toutes les vues.

2.3 État de l'art ciblé pour la compression des données 3D

Les algorithmes de compression de données 3D se répartissent en deux catégories. La première regroupe les algorithmes qui compressent le maillage représentant la surface des objets. Il existe deux manières de représenter un objet : avec une équation et avec un maillage. Représenter un objet avec une équation ne prend que quelques octets en mémoire car un objet simple (un cube, une sphère, un cylindre, un cône...) aura une équation simple. Par exemple, l'équation cartésienne d'une sphère de centre (a, b, c) et de rayon r sera $(x-a)^2 + (y-b)^2 + (z-c)^2 = r^2$. Tout point de coordonnées (x, y, z) vérifiant cette équation est sur la surface de la sphère. Pour créer des objets un peu plus complexes, on a recours à la Constructive Solid Geometry, ou CSG [55], permettant de faire des opérations arithmétiques sur les objets, comme des unions, des soustractions et des intersections. Par exemple, pour représenter une demi-sphère (voir figure 2.2), on crée une sphère et on lui soustrait un cube qui englobe la demi-sphère qu'on veut retirer. Ceci permet, par la suite, de calculer le rendu de l'objet.

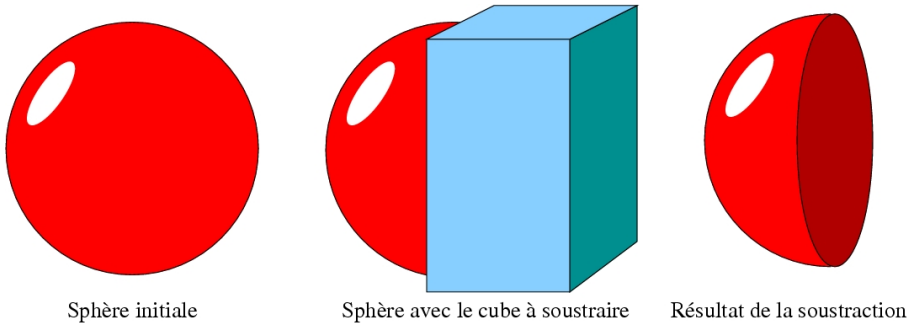


FIG. 2.2 – Exemple d'utilisation de la CSG pour créer une demi-sphère

L'inconvénient de cette méthode est que, pour coder des objets plus complexes, comme des données médicales ou des relevés de terrains, il devient très difficile de

trouver une équation permettant de s'approcher de la forme de l'objet et le nombre d'opérations de CSG pour le représenter à partir d'un objet simple devient très important. Par exemple, si on veut représenter un terrain accidenté, il faudra soustraire à un cube un autre cube pour chaque altitude relevée. Ceci prendra un temps considérable à calculer et l'aspect de la surface de l'objet entre deux relevés risque de ne pas correspondre à ce qui est attendu.

Une autre méthode de représentation d'un objet complexe consiste à le définir sous la forme d'un maillage. Chaque objet représenté est défini par une série de facettes, le plus souvent triangulaires, définies par les coordonnées de leurs sommets dans le référentiel 3D de la scène. Cet ensemble de facettes est appelé *maillage*. Il existe plusieurs façons de représenter ce maillage : soit on indique, pour chaque facette, les coordonnées de tous ses sommets, soit on indique les coordonnées de tous les points utilisés par tous les objets de la scène et chaque facette est définie par une table de connectivité, c'est-à-dire la liste des indices (dans la liste des sommets) des sommets qui la composent. Cette dernière méthode est celle utilisée par les fichiers de type DXF (Drawing eXchange Format) [3] introduit par le logiciel Autocad [4] d'AutoDesk [5] et couramment utilisés par les logiciels de CAO et de synthèse d'image du commerce. Avec une telle représentation, la taille nécessaire à la représentation des facettes est nettement plus importante que celle des coordonnées des points.

C'est donc sur cette partie que la plupart des algorithmes de compression portent leurs efforts. Cette compression se fonde sur les travaux réalisés sur la compression des maillages plans ([94], [95], [54], [93], [74], [69], [58], [70], [80], [102]) qui représentent les contacts entre les différentes facettes sous la forme d'un graphe de contact, afin d'éviter les répétitions. Par exemple, si on veut représenter une sorte de ruban, composé d'une série de triangles, comme celui représenté sur la figure 2.3, il n'est pas nécessaire de renseigner, pour chaque facette, ses trois sommets, car elle en a au moins deux en commun avec la précédente. Sur la figure 2.3, la facette A est définie par les sommets 1, 2 et 3. Une fois cette facette définie, on va coder la facette B, définie par les sommets 2, 3 et 4. Or, si on sait que la facette A touche la facette B, en ayant les sommets 2 et 3 en

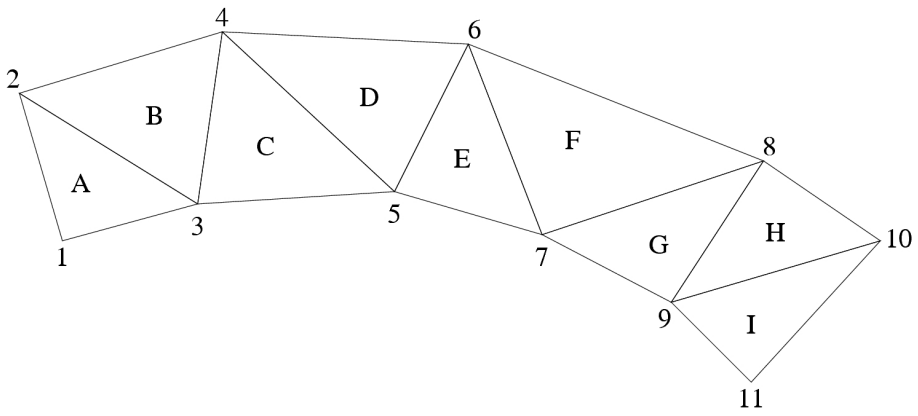


FIG. 2.3 – Exemple de compression d'une surface plane : pour chaque facette, on évite de répéter les sommets de la facette précédente

commun (les deux derniers définis), il suffit de donner le troisième sommet, le sommet 4. Pour la facette C, on utilise le même principe : les deux derniers sommets définis sont 3 et 4, il suffit d'ajouter le sommet 5 pour définir la facette C. Il suffit donc de coder les trois sommets de la première facette, puis uniquement le sommet supplémentaire des autres. Le codage des facettes A, B, C, D, E, F, G, H et I, formant la totalité du ruban peut donc se limiter au codage des sommets 1, 2, 3, 4, 5, 6, 7, 8, 9 10 et 11, soit 11 sommets au lieu de 27 si on avait codé les trois sommets de chaque facette.

Ce principe peut ensuite être adapté à des surfaces plus complexes, en codant les contacts entre les différentes facettes sous la forme d'arbres ou de graphes, pour pouvoir représenter des objets 3D complexes ([35], [31], [38], [51], [47], [64], [45], [91], [82], [83], [60]).

Ces surfaces peuvent également être compressées en utilisant les techniques de compression par ondelettes [85] [71] [72], comme pour les images. En effet, tout comme une forme plane, un objet 3D peut également être décomposé en une somme de fonctions d'ondes. La forme globale sera codée par les fonctions ayant des fréquences faibles, alors que les détails correspondront aux fonctions ayant des fréquences élevées.

D'autres méthodes permettent de décomposer la forme de l'objet de manière progressive et hiérarchique : la forme globale de l'objet est donnée, puis on affine cette surface en envoyant des données supplémentaires pour les zones contenant plus de détails,

par exemple, la méthode dite “Compressed Progressive Meshes” [29] [76]. Khaled Mamou [59] a proposé en 2005 une méthode de compression progressive de maillage 3D par approximation B-spline. Cette méthode consiste à segmenter le maillage en patchs. Chaque patch est ensuite paramétré et approximé par une surface B-spline. Les points de contrôle de cette surface sont ensuite compressés de manière progressive à l'aide d'un codeur JPEG2000 [89].

Toutes ces méthodes ne compressent que les surfaces des objets. Une fois la décompression effectuée, l'objet est manipulable et observable depuis n'importe quel point de vue. Par contre le rendu n'est pas du tout pris en compte dans cette projection et doit être fait en projetant chaque objet après décompression. Ce genre d'algorithme permet donc de compresser des objets 3D, mais ne permet aucun gain de place mémoire pour les données affichées (les différentes vues), ni de temps pour le calcul de ces données, ce qui était le but recherché. Par conséquent, même si ces algorithmes peuvent être utiles pour le transfert de données 3D, ils ne sont pas adaptés à nos besoins car ils n'ont aucun impact sur la taille nécessaire à mémoriser les différentes vues, ni le temps nécessaire à générer ces vues.

La deuxième catégorie d'algorithmes de compression de données 3D regroupe les algorithmes qui compressent une série d'images, correspondant aux objets vus depuis des points d'observation placés dans des directions différentes.

Les premiers algorithmes [68] [77] [20] ne codaient qu'un couple stéréoscopique, c'est-à-dire les images vues par chacun des deux yeux de l'observateur. Le principe était de coder une des deux images, puis, au lieu de coder la seconde indépendamment, de ne coder que les décalages horizontaux de blocs de pixels similaires entre la première image (l'image de référence) et la seconde (l'image cible) afin de réduire la taille des données. Ensuite, d'autres algorithmes ont généralisé cette méthode en permettant de coder un nombre quelconque (supérieur ou égale à 2) d'images, en considérant les différentes images successives comme les images d'une vidéo et en utilisant dessus les même principes que les compressions vidéo, comme la norme MPEG [62] [81]. Les différences entre

ces algorithmes portent essentiellement sur la détection des blocs de pixels communs aux deux images qui correspondent aux objets placés à des distances différentes. Ces algorithmes peuvent être basés sur un calcul de projection des pixels [24] [23] [22], ou sur une extension de la transformée de Fourier discrète adaptée à la 3D [86] [53], ou en découpant les images en surfaces triangulaires [44] ou quelconques [88] ou en décomposant l'images en plans parallèles à l'écran [96] [103], correspondant aux différentes profondeurs des voxels.

Pour réduire encore la taille des données, certains algorithmes [103] ne codent pas toutes les images, mais seulement l'image centrale en tant qu'image de référence et la profondeur de chaque pixel, représentée sous la forme d'une image (appelée Depth map) en niveau de gris, chaque niveau représentant une profondeur, comme image cible. Les autres images sont alors calculées en déformant l'image de référence en fonction de la profondeur de chaque pixel. D'autres [30] utilisent les deux images extrêmes (l'image correspondant au point de vue situé le plus à gauche et celle correspondant au point de vue situé le plus à droite) comme images de référence et codent les depth maps des images intermédiaires.

Ces algorithmes de compression permettent de réduire la taille des données mémorisées et de restituer ensuite les différentes vues. Ils présentent cependant l'inconvénient de nécessiter de calculer les différentes images à compresser. Lorsque le nombre d'images différentes devient grand, cela nécessite un temps de calcul non négligeable. Par contre, dans le cas où le nombre d'images à calculer est limité, mais que l'algorithme de décompression permet de générer plus d'images qu'en entrée, il est possible que certaines parties de ces images générées ne puissent être reconstituées convenablement car les informations s'y rapportant n'auront pas été présentes dans les images compressées par l'algorithme. Prenons, par exemple, une scène composée de deux cubes placés côte à côte, avec un trou entre eux. Si on les regarde de face, on voit une partie limitée de ce qu'il y a derrière. Lorsqu'on va modifier la position de l'observateur, la partie visible de ce qu'il y a derrière va changer. Mais si l'algorithme de codage n'a pris en compte que

les deux images extrêmes ou l'image centrale et les images extrêmes, il ne pourra coder que ce qui est vu par l'observateur sur ces deux ou trois images, mais pas ce qui est visible pour les autres points de vue.

En conclusion, si les algorithmes de compression des données 3D existant permettent de réduire la taille des données à envoyer au dispositif d'affichage, ils ne permettent de réduire la quantité de calculs nécessaires au calculs des vues qu'au prix d'une perte de certaines données d'affichage.

2.4 Algorithme de compression des données 3D

Nous avons donc mis au point un algorithme pouvant être utilisé pour tout type de dispositif autostéréoscopique, mais également pour des reconstitutions d'objets 3D (comme des sprites 3D, par exemple) selon un certain nombre de directions et codant directement les données 3D, présentant l'avantage de ne projeter les objets 3D qu'une seule fois et de ne pas être dépendant d'un choix de données en entrée, car toutes les donnés 3D seront codées directement, sans passer par un calcul d'image.

Le choix de cet algorithme a été motivé par le fait que la plus grosse contrainte que nous avions était la vitesse importante nécessaire pour la restitution des vues. Nous avons donc opté pour un algorithme de restitution le plus simple possible et facilement implémentable sur une architecture spécialisée. Nous en avons donc proposé un qui se limite, pour la partie décodage, à quatre boucles imbriquées et à une recopie de données.

2.4.1 Principe

Projection des objets

Cet algorithme utilise directement les objets 3D à afficher. Ces objets sont projetés exactement comme pour obtenir une image 2D. Les deux seules différences sont que la profondeur de chaque pixel obtenu est conservée (on va donc coder des voxels et plus des pixels) et que tous les voxels de la scène sont codés, pas seulement ceux qui sont visibles. Le point d'observation utilisé pour cette projection est celui correspondant à

un observateur situé au centre de la zone de projection, c'est-à-dire en face de l'écran.

La projection des voxels (schématisée sur la figure 2.4) se fait en projetant les différents points des objets à afficher sur un plan représentant l'écran d'affichage dans le référentiel de la scène 3D. Cet "écran virtuel" a généralement une largeur de 1 dans

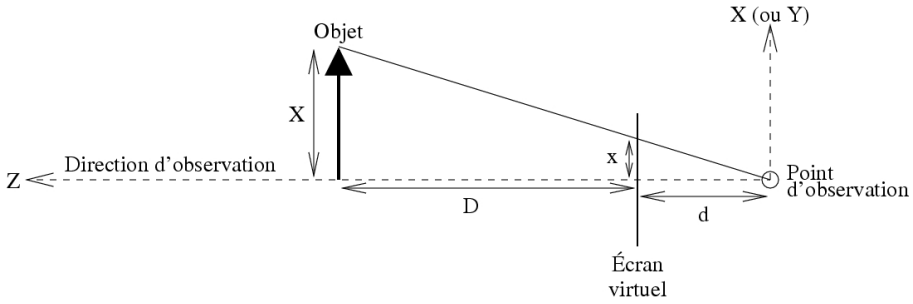


FIG. 2.4 – Principe de projection des objets virtuels

et un rapport largeur/hauteur égal à celui de l'écran réel du dispositif d'affichage. La position à laquelle le point est projeté (x sur le schéma de la figure 2.4) détermine des coordonnées comprises entre 0 et 1 qui sont ensuite multipliées par le nombre de pixels des lignes de l'écran, ce qui donne les coordonnées du pixel où ce point doit être dessiné. Cette coordonnée x est égale à $\frac{X \times d}{D+d}$ dans le référentiel de l'observateur, centré sur le point d'observation et dont l'axe Z est aligné avec la direction d'observation, comme indiqué sur la figure 2.4.

Le fait de projeter les voxels et de mémoriser les coordonnées du pixel correspondant où ils doivent être projetés évite de devoir les reprojeter au moment du décodage. Celui-ci se limite donc à une simple recopie des voxels avec un décalage dépendant de la profondeur des voxels. On va ainsi coder les voxels de toutes les facettes des objets, y compris celles cachées, et projeter également les objets et parties d'objets placés derrières d'autres objets qui pourraient les cacher. On obtient ainsi les coordonnées x et y, correspondant aux coordonnées du pixel sur l'écran où chaque pixel devrait être projeté si on générait une image 2D.

La profondeur des voxels est ensuite mise à l'échelle pour que le rapport entre la profondeur et les coordonnées x et y soit respecté. Cette mise à l'échelle est faite en

multipliant la coordonnée z des voxels par la résolution horizontale de l'écran.

La profondeur est ensuite discrétisée, c'est-à-dire qu'elle ne pourra prendre qu'un nombre limité de valeurs. Ce nombre et ces valeurs dépendent des caractéristiques du dispositif d'affichage, en particulier du nombre de vues différentes et de l'angle qui les sépare. Plus le nombre de vues et la résolution de l'écran sont élevés, plus la précision de la profondeur doit être importante. La profondeur réelle et la profondeur discrétisée sont mémorisées toutes les deux et définissent le voxel.

Chaque voxel est donc défini par ses coordonnées x et y, correspondant aux coordonnées de la projection du voxel sur l'écran, ainsi que sa coordonnée r_z, correspondant à sa profondeur mise à l'échelle et discrétisée, z, correspondant à sa profondeur mise à l'échelle, mais non discrétisée et sa couleur, définie par ses trois canaux de couleur rouge, vert et bleu, ainsi que son canal alpha, c'est-à-dire sa transparence.

Les voxels ainsi définis sont ensuite triés selon leurs coordonnées. Tout d'abord, selon leur coordonnée y. Les voxels sont ainsi codés en lignes horizontales. Chaque ligne est codée indépendemment des autres. Ces lignes sont classées et codées de la plus haute à la plus basse sur l'écran.

Pour chaque ligne, les voxels sont réunis en plans. Chaque plan réunit l'ensemble des voxels de la ligne située à la même profondeur discrétisée. Seuls les plans contenant au moins un voxel sont codés afin de réduire la taille mémoire nécessaire. Un plan est donc défini par sa profondeur et la ligne dans laquelle il se trouve.

À l'intérieur d'un plan, les voxels sont regroupés en blocs de voxels contigus, appelés des motifs. Un motif est donc un ensemble de voxels ayant la même coordonnée y, la même coordonnée z et ayant des coordonnées x entières qui se suivent. En d'autres termes, si la coordonnée x du voxel le plus à gauche d'un motif composé de n voxels est X, les coordonnées x des voxels du motif sont respectivement X, $X+1$, $X+2$ ... $X+n-1$.

En résumé, les voxels sont codés de la manière indiquée dans le tableau 2.1.

```
+ Couleur du fond de la scène
+ Pour chaque ligne (de haut en bas) de l'écran, faire
|  + Nombre de plans utilisés dans la ligne
|  + Pour chaque plan (du plus lointain au plus proche) utilisé dans la ligne, faire
|  |  + Profondeur du plan
|  |  + Nombre de motifs de ce plan
|  |  + Pour chaque motif (de gauche à droite) du plan, faire
|  |  |  + Coordonnée x du voxel le plus à gauche du motif
|  |  |  + Nombre de voxels du motif
|  |  |  + Pour chaque voxel (de gauche à droite) du motif, faire
|  |  |  |  + Couleur (ARGB) du voxel
|  |  |  + Fin du codage des voxels
|  |  + Fin du codage du motif
|  + Fin du codage du plan
+ Fin du codage de la ligne
```

TAB. 2.1 – Principe du codage des informations 3D

Codage des voxels

L'algorithme de codage de voxels consiste donc à réunir les voxels ayant la même coordonnée y, la même coordonnée z et des coordonnées x successives en motifs et à classer ces motifs d'abord selon leur coordonnée y croissante, puis selon leur coordonnée z décroissante et enfin selon leur coordonnée x croissante. Il se déroule en deux phases : la première consiste à mémoriser les voxels en les regroupant en motifs et en les classant. Cette phase doit se faire avec une structure de données permettant d'insérer de nouveaux motifs et de les fusionner entre eux, pour éviter de nombreux et inutiles recopiages de données. Celle que nous considérons comme la plus adaptée pour classer les plans est celle du tas car elle permet une insertion et un effacement ayant tout les deux une complexité en o(log(n)), où n est la taille des données contenues dans le tas. Pour le tri des motifs, nous avons utilisé des listes chaînées. Cette structure présente l'inconvénient de n'être pas très efficace pour classer les différents éléments (motifs) mais permet d'accéder très facilement et rapidement au motif suivant pour vérifier si une fusion de motifs doit avoir lieu et opérer facilement cette fusion. La seconde phase est juste une mise en forme des données mémorisées sous la forme de données brutes.

Les données en entrée sont soit des voxels indépendants, soit des motifs déjà créés. Dans le premier cas, on recherche si le voxel doit être inséré dans un motif existant. Si oui, on l'insère, sinon on le crée. Dans ce cas, les tests de fusion des motifs sont réduits car il ne peut plus y avoir fusion entre plus de deux motifs. L'algorithme complet de codage des voxels est celui décrit dans l'algorithme n°1 : **Insertion d'un voxel**. Dans le second cas, on recherche la position où insérer le motif et on vérifie si un (ou plusieurs) motif(s) existant(s) peu(ven)t fusionner avec le nouveau motif. Une fusion arrive lorsqu'un ou plusieurs voxels d'un des motifs doivent être insérés à la même position qu'un ou plusieurs voxels d'un autre motif, c'est-à-dire qu'ils ont les mêmes coordonnées x, y et z, ou que le motif inséré se trouve juste à coté d'un autre motif. Dans ce dernier cas, les deux motifs sont contigus et le voxel le plus à droite de l'un des deux se trouve juste à gauche du voxel le plus à gauche de l'autre. En d'autres termes, la coordonnée x du voxel le plus à droite d'un des motifs est égale à a, la coordonnée x du voxel le plus à gauche de l'autre motif est égale à b et $a = b - 1$. Ces deux cas peuvent se produire simultanément pour le même motif inséré. Si la taille du motif inséré est importante, il peut fusionner avec plusieurs autres motifs simultanément. Il faut donc prévoir tous les cas de figure dans l'algorithme. La complexité de cet algorithme est $o(n \times r \times p \times i)$, où n est le nombre d'éléments (voxels ou motifs) à insérer, r est le temps mis pour la recherche, dépendant de l'algorithme utilisé, p est la probabilité de fusion d'un élément avec les éléments déjà insérés et i est le temps moyen nécessaire pour cette fusion.

La seconde phase (la mise en forme des données) consiste ensuite à lire toutes les informations mémorisées et à les mettre bout à bout, selon le principe décrit dans le tableau 2.1, afin de les envoyer au dispositif d'affichage. L'algorithme utilisé est indiqué sous la référence algorithme n°2 : **Mise en forme des données**. Cet algorithme consiste simplement à parcourir les données mémorisées grace à l'algorithme n°1 et à les écrire dans l'ordre. Sa complexité est $o(l \times p \times m \times t)$, où l est le nombre de lignes, p est le nombre moyen de plans par ligne, m est le nombre moyen de motifs par plan et t la taille moyenne des motifs. Cette complexité peut également être définie en fonction de

Algorithme n° 1 Insertion d'un voxel

Données : Un espace mémoire où écrire les données mémorisées
Données : Un voxel à insérer, défini par :

- **x**, **y** et **z** : ses trois coordonnées correspondant aux coordonnées x et y du pixel où le voxel doit être projeté sur l'écran et z sa profondeur mise à l'échelle (voir le texte) des autres coordonnées et discrétisée.
- $\mathbf{r_z}$: sa profondeur "réelle" (non discrétisée) mise à l'échelle.
- **col** : Les composantes ARGB (couleur et transparence) du voxel

Résultat : La zone mémoire résultat contient le nouveau voxel trié

```
1: Chercher l'adresse des données de la ligne correspondant à y
2: Si la ligne ne contient pas de plans utilisés, alors
3:    Créer le plan de profondeur z dans la ligne y
4:    Créer un motif à la coordonnée x et de taille 1 dans ce plan
5:    Copier r_z et col dans le voxel de ce motif
6: Sinon
7:    Rechercher, dans la ligne le plan correspondant à la profondeur z du voxel ou la position à laquelle
      ce plan devrait se trouver, s'il n'existe pas
8:    Si le plan correspondant n'existe pas, alors
9:       Créer un nouveau plan de profondeur r_z du voxel
10:      Créer un motif de coordonnée x et de taille 1 dans le plan
11:      Copier r_z et col dans le voxel de ce motif
12:      Insèrer le nouveau plan tel que les plans soient triés selon leur profondeur dans la ligne
13:   Sinon
14:      Rechercher, dans le plan, le motif dans lequel le voxel devrait être inséré ou la position où il
         devrait se trouver, s'il n'existe pas
15:      Si le motif correspondant n'existe pas, alors
16:         Créer un motif de coordonnée x et de taille 1 dans le plan
17:         Copier r_z et col dans le voxel de ce motif
18:         Insèrer le nouveau motif tel que les motifs soient triés selon leur coordonnée x dans le plan
19:      Sinon
20:         Calculer la position où le voxel devrait se trouver à l'intérieur du motif
21:         Si un voxel existe déjà dans le motif à cette position, alors
22:            Si le nouveau voxel est plus près que l'ancien (si sa valeur r_z est inférieure à celle de
               l'autre voxel), alors
23:               Copier r_z et col dans le voxel, en effaçant l'ancien
24:            Fin Si
25:         Sinon
26:            Insèrer le voxel dans le motif {Le voxel est à une des deux extrémités du motif}
27:            Si le voxel est le dernier (resp. le premier) du motif et touche le premier (resp. le dernier)
               voxel du motif suivant (resp. précédent), alors
28:               Fusionner les deux motifs
29:            Fin Si
30:         Fin Si
31:      Fin Si
32:   Fin Si
33: Fin Si
```

la taille globale des données, n. En effet, la taille totale des données n est proportionnelle à l'expression $l \times p \times m \times t$. La compléxité de cet algorithme devient alors o(n), où n est la taille totale des données à décoder.

Algorithme n° 2 Mise en forme des données

Données : Zone mémoire contenant les voxels codés avec l'algorithme n°1
Données : Nombre de lignes du dispositif d'affichage 3D auquel les données sont destinées
Données : Couleur de fond de la scène à coder
Données : Zone mémoire ou tube de communication où écrire les données

```
1: Écrire la couleur de fond de la scène
2: Pour chaque ligne du dispositif de haut en bas, faire
3:    Écrire le nombre de plans utilisés par cette ligne
4:    Pour chaque plan de la ligne du plus lointain au plus proche, faire
5:       Écrire la profondeur du plan
6:       Écrire le nombre de motifs de ce plan
7:       Pour chaque motif du plan de gauche à droite, faire
8:          Écrire la coordonnée x du voxel le plus à gauche du motif
9:          Écrire le nombre de voxels de ce motif
10:         Pour chaque voxel du motif de gauche à droite, faire
11:            Écrire la couleur (ARGB) du voxel
12:         Fin Pour
13:      Fin Pour
14:   Fin Pour
15: Fin Pour
```

Décodage des voxels

L'algorithme de décodage est très simple car il se limite à quatre boucles imbriquées et des copies de données. La première boucle sert à compter les lignes, la seconde les plans utilisés de chaque ligne, la troisième les motifs de chaque plan et la quatrième à parcourir les voxels à l'intérieur de chaque motif. Pour chaque plan, on détermine une valeur de décalage des voxels de ce plan. Cette valeur de décalage est égale à une valeur entière relative, correspondant à l'angle de la vue à générer, multipliée par la profondeur z du plan. Cette valeur est calculée à partir du rapport entre la position horiontale du point d'observation par rapport au centre de l'écran et sa distance à l'écran. Le décalage de chaque voxel est donc égal à cette valeur de décalage additionnée à la position horizontale du voxel par rapport au milieu de l'écran. Chaque voxel est ensuite simplement recopié dans la mémoire résultat en utilisant le canal alpha pour mélanger la couleur des voxels avec la couleur qui se trouvait à cette position dans la mémoire résultat pour gérer la transparence des voxels. L'algorithme complet est indiqué dans l'algorithme n°3 : **Reconstitution des vues**.

On remarque aisément que cet algorithme se limite à quatre boucles imbriquées : une pour les lignes, une pour les plans, une pour les motifs et une pour les voxels. Les

Algorithme n° 3 Reconstitution des vues

Données : Zone mémoire (ou fichier) contenant les voxels codés avec l'algorithme n°2
Données : Résolution du dispositif d'affichage 3D (ou de la vue à reconstituer) auquel les données sont destinées
Données : Angle correspondant à la vue à reconstituer, défini par une valeur entière, permettant de déterminer le décalage des motifs en fonction de leur profondeur
Données : Zone mémoire de taille suffisante pour recevoir l'image 2D résultat ayant la résolution définie ci-dessus, c'est-à-dire largeur de l'écran × hauteur de l'écran × taille utilisée par un pixel (par exemple 24 bits)
Résultat : La vue correspondant à l'angle passé en paramètre des données codées selon l'algorithme n°1 est écrite dans la zone mémoire résultat
Données : **plane_offset**, **pattern_offset**, **alpha** et **old_pixel** sont des variables locales de travail

```
1: Remplit la mémoire résultat avec la couleur de fond
2: Pour chaque ligne de l'écran , faire
3:    Pour chaque plan de cette ligne , faire
4:       plane_offset ← angle × profondeur du plan
5:       Pour chaque motif de ce plan , faire
6:          pattern_offset ← plane_offset + position horizontale du motif
7:          Pour chaque voxel de ce motif , faire
8:             alpha ← canal alpha du voxel étendu à un nombre réel, ramené dans l'intervalle [0, 1]
9:             old_pixel ← pixel de coordonnées (pattern_offset + numéro du voxel, y) dans la zone
               mémoire résultat
10:            Pour chaque composante R, G et B , faire
11:               composante du pixel de coordonnées (pattern_offset + voxel number, y) dans la zone
                  mémoire résultat ← (alpha × composante du voxel) + ( (1 - alpha) × composante de
                  old_pixel)
12:            Fin Pour
13:         Fin Pour
14:      Fin Pour
15:   Fin Pour
16: Fin Pour
```

opérations exécutées à l'intérieur de ces boucles (lectures et écritures de données) étant en temps constant, la compléxité de cet algorithme est égale à o($l \times p \times m \times t$), où l est le nombre de lignes, p est le nombre moyen de plans par ligne, m est le nombre moyen de motifs par plan et t la taille moyenne des motifs. Cette complexité peut également être définie en fonction de la taille globale des données, n. En effet, la taille totale des données n est proportionnelle à l'expression $l \times p \times m \times t$. La compléxité de cet algorithme devient alors o(n), où n est la taille totale des données à décoder.

On remarque également que le décalage de chaque vue est indépendant du décodage des autres vues. En effet, l'algorithme n°3 sert à générer la vue correspondant à une direction donnée et ne modifie pas les données en entrée. Il peut donc être exécuté plusieurs fois simultanément en parallèle sur les même données. Il en va de même pour

les lignes : chaque ligne étant codée indépendamment des autres, le décodage de plusieurs lignes différentes de la même vue peut se faire en parallèle.

2.4.2 Problème de la discontinuité des facettes

Si on utilise l'algorithme n°1 tel quel, on se rend compte que certains trous apparaissent sur la surface de certains objets sur certaines vues. Ces trous prennent la forme d'arcs de cercle qui se déplacent d'une vue à l'autre et disparaissent sur les vues proches de celles correspondant à la direction utilisée pour la projection initiale. Un exemple de vue présentant de tels trous est présentée figure 2.5.

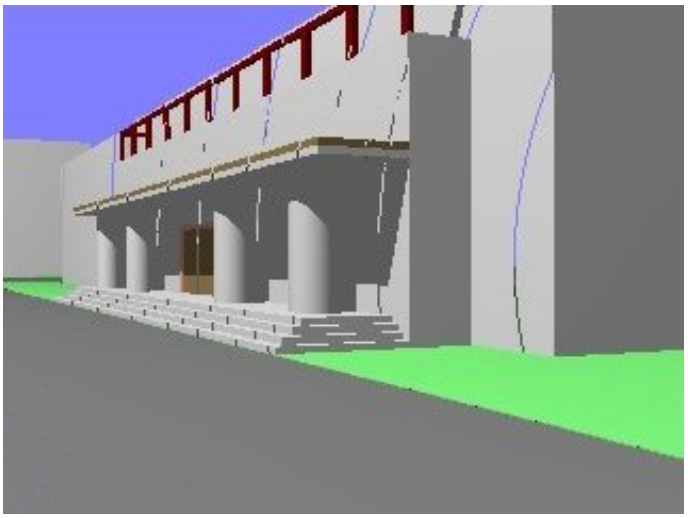


FIG. 2.5 – Exemple de discontinuité des facettes

Ces trous proviennent de la différence de décalage des différents plans. Si une facette n'est pas parallèle à l'axe horizontal de l'écran, elle est va être constituée de plusieurs motifs situés dans des plans (profondeurs) différents. Comme les voxels sont générés par projection des objets selon une direction donnée, il est possible que deux voxels placés l'un à coté de l'autre sur l'écran (un de coordonnée horizontale X et l'autre $X + 1$) soient placés sur des plans éloignés. Dans ce cas, le décalage du plan le plus lointain sera plus important que celui du plan le plus proche. La figure 2.6 illustre ce phénomène : les voxels représentés par des carrés correspondent à ceux créés par la projection des objets. Cette projection se fait dans la direction perpendiculaire à l'écran, c'est-à-dire

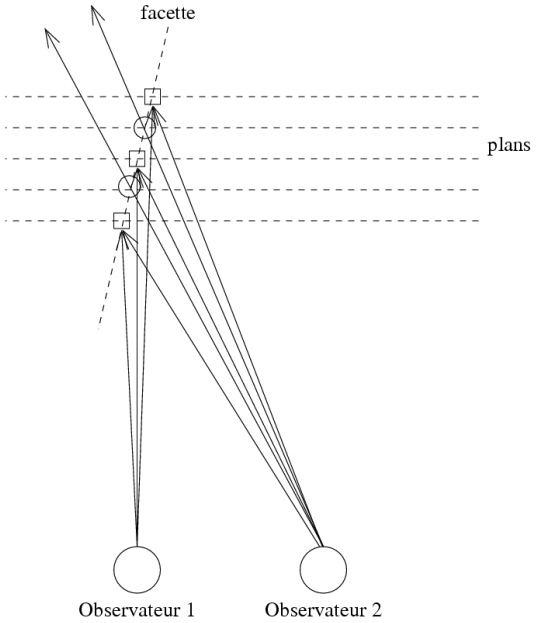


FIG. 2.6 – Illustration de la raison de la discontinuité des facettes.

en direction de l'observateur 1. Celui-ci voit donc les voxels tels qu'il les verrait si la projection avait servi à générer une image plane. Par contre, pour l'observateur 2, les décalages des plans successifs sont différents, ce qui équivaut à regarder l'objet selon un angle différent, en faisant apparaître des trous entre ces voxels, au niveau des ronds. Nous avons donc proposé un algorithme permettant de combler ces trous en créant de nouveaux voxels.

Comme nous l'avons vu précédemment, au moment de la génération des vues, le décalage de chaque plan est égal à la valeur d'angle multipliée par la profondeur de plan, où la valeur d'angle est un entier signé correspondant à l'angle de la vue à générer.

Si on considère deux voxels, notés A et B, de coordonnées x_A, y_A, z_A, r_{zA} pour A et x_B, x_B, z_B, r_{zB} pour B, telles que $x_A = x_B - 1$ et A et B font partie de la même facette, il ne devrait pas y avoir de trou entre eux. Mais si $z_A \neq z_B$, A and B sont dans deux plans différents. Considérons max_{angle} et min_{angle} le maximum (resp. minimum) de la valeur d'angle, telle que A et B soient tous les deux visibles sur la vue résultat. Le décalage de A sera de $max_{angle} \times z_A$ vers la droite pour une des deux vues et $min_{angle} \times z_A$

vers la gauche pour l'autre. Pour B, ces valeurs seront respectivement $max_{angle} \times z_B$ et $min_{angle} \times z_B$. Si ($(z_A < z_B)$ et décalage de A - décalage de B) > 1) ou ($(z_A > z_B)$ et (décalage de A - décalage de B) < -1) alors un espace vide apparaîtra entre A et B.

Pour faire disparaître ce trou, de nouveaux voxels doivent être insérés derrière le voxel le plus proche, c'est-à-dire avec la même coordonnée x et la même coordonnée y que lui, mais avec une coordonnée z comprise entre celle du voxel initial le plus proche et celle du plus lointain. Pour éviter que d'autres trous n'apparaissent, ces voxels doivent vérifier la condition suivante :

$Si(z_C < z_D),\ alors$

$max_{angle} \times z_C - max_{angle} \times z_D = 1$

sinon

$max_{angle} \times z_C - max_{angle} \times z_D = -1$

où z_C et z_D sont les coordonnées z de deux voxels successifs notés C et D, respectivement, entre A et B tels que :

$Si(z_A < z_B),\ alors$

$x_C = x_D = x_a$

sinon

$x_C = x_D = x_b$

De même, soit C_1 le premier voxel inséré juste derrière A et D_1 le dernier voxel inséré juste avant B, les points A, B, C_1 et D_1 vérifient les deux équations :

$Si(z_A < z_{C_1}),\ alors$

$max_{angle} \times z_A - max_{angle} \times z_{C_1} = 1$

sinon

$max_{angle} \times z_A - max_{angle} \times z_{C_1} = -1$

et

$Si(z_{D_1} < z_B),\ alors$

$max_{angle} \times z_{D_1} - max_{angle} \times z_B = 1$

sinon

$max_{angle} \times z_{D_1} - max_{angle} \times z_B = -1$

La couleur associée à chacun de ces voxels peut être obtenue en reprojetant les objets ou de nouveaux rayons sur les objets, ou par interpolation à partir des informations de texturage des objets ou directement à partir des couleurs des voxels A et B.

La compléxité de l'insertion de chacun des voxels insérés peut être de o(p) où p est le nombre moyen de motifs par plan, car il suffit d'insérer le voxel dans un plan qui est déjà connu : le plan situé juste derrière le voxel A.

Cet algorithme permet de faire disparaître les trous dans les facettes en ajoutant des voxels dans la scène 3D à afficher. D'après les tests que nous avons réalisés, ces trous ne représentent qu'une petite partie (moins de 10%) de la scène 3D. Les voxels ajoutés pour les boucher ne feront donc pas augmenter la taille des données et, par conséquent, la vitesse du décodage, au delà de cette valeur.

2.4.3 Codage des effets lumineux particuliers

Les deux algorithmes que nous venons de décrire ne permettent de représenter que des objets parfaitement diffus. Pour pouvoir représenter des objets présentant un éclairage spéculaire, une réflexion ou une réfraction, il nous fallait un système de codage permettant de restituer ces phénomènes de manière réaliste. En effet, ces trois phénomènes ont la particularité de n'être pas figés par rapport à l'objet qui les génère. En d'autres termes, lorsque l'observateur modifie son point de vue, l'image perçue par chacun de ces trois effets est modifiée également. Par exemple, si on regarde dans un miroir, l'image réfléchie dans le miroir change lorsqu'on se déplace par rapport au miroir : on ne voit pas la même image réfléchie selon sa position car la position où les objets sont perçus par rapport au miroir est différente selon la position de l'observateur. Le schéma de la figure 2.7 montre le déplacement d'un objet, noté *objet réfléchi*, lorsque l'observateur bouge. Les rayons lumineux issus de l'objet réfléchi, notés *rayons incidents*, sont réfléchis symétriquement par rapport à la normale à la surface réfléchissante au point d'incidence. Lorsqu'il est à la position notée *Observateur 1* sur la figure, l'observateur voit l'objet

réfléchi au *point de réflexion 1*. Par contre, lorsqu'il se trouve à la position *Observateur 2*, l'image réfléchie de l'objet se trouve au *point de réflexion 2*.

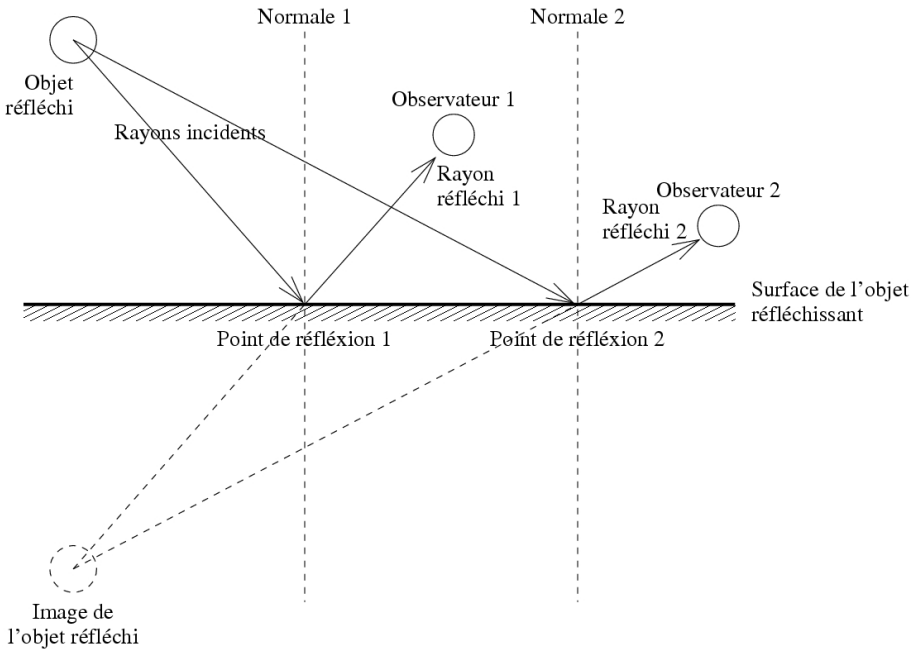


FIG. 2.7 – Réflexion d'un objet sur une surface

Dans le cas de certains algorithmes de rendu 3D rapide de surfaces réfléchissantes [25] [79], on calcule d'abord l'image réfléchie, puis on plaque celle-ci sur l'objet réfléchissant sous la forme d'une texture. Dans le cas d'images fixes, cela donne l'illusion d'une surface réfléchissante. Mais dans notre cas, ce subterfuge se verrait car l'image réfléchie ne changerait pas en fonction de la position d'observation. Il a donc été nécessaire de trouver un algorithme permettant de rendre l'image réfléchie de manière réaliste quelle que soit cette position.

Nous avons donc proposé une adaptation des algorithmes n°1, n°2 et n°3 pour qu'ils puissent afficher d'autres scènes 3D au travers des motifs. Cette adaptation consiste à coder une scène 3D secondaire indépendamment de la scène 3D principale et d'associer cette scène à un objet de la scène principale. Le codage demeure presque inchangé : la scène secondaire est codée indépendemment de la scène principale et la fusion des

motifs et l'insertion des voxels dans un motif doivent prendre en compte la présence ou l'absence d'une scène secondaire pour éviter de fusionner deux motifs incompatibles, c'est-à-dire l'un codant une surface réfléchissante et l'autre non. L'algorithme de codage modifié est indiqué dans l'algorithme n°4 : **Insertion d'un voxel avec détection de la compatibilité des motifs**. La mise en forme des données est réalisé par une adaptation de l'algorithme n°2 : **Mise en forme des données** que nous avons décrit précédemment. Cette adaptation consiste à mettre le bit de poids fort de la taille des motifs à un lorsqu'ils contiennent une sous-scène et à ajouter la place mémoire nécessaire à mémoriser l'indice de début de cette sous-scène. L'algorithme modifié est noté algorithme n°5 : **Mise en forme des données avec gestion des sous-scènes**. La scène secondaire est codée de la même façon que la scène principale, en utilisant l'algorithme n°6 : **Mise en forme des données des sous-scènes**. Le codage complet d'une scène contenant des objets réfléchissants, réfractant ou présentant un éclairage spéculaire se fait en utilisant l'algorithme 7 : **Codage d'une scène contenant des sous-scènes**.

Les adaptations de ces algorithmes ne modifient pas leur compléxité qui reste la même que celle définie précédemment.

Ceci permet d'afficher des objets permettant de montrer d'autres objets complètement indépendants. L'effet obtenu est similaire à celui observé en regardant un hologramme : l'objet permettant de voir les autres objets se comporte comme la plaque sensible de l'hologramme et les objets vus au travers sont perçus comme les objets représentés par l'hologramme. Les objets représentés peuvent être situés devant, derrière ou à la même profondeur que l'objet au travers duquel ils sont visibles. Cet effet peut être facilement imaginé en visualisant un environnement 3D dans lequel flottent des fenêtres qui affichent des images 3D. Ces fenêtres sont composées de motifs affichant une autre scène et cette autre scène est celle affichée à l'intérieur de la fenêtre. Dans une scène quelconque, cette scène secondaire peut soit représenter quelque chose de complètement indépendant de la scène principale, comme pour les fenêtres, soit la même scène vue par réflexion ou réfraction. Dans ce dernier cas, le fait de changer de position permettra de percevoir la scène réfléchie ou réfractée selon des angles différents, ce qui est conforme à la réalité.

Algorithme n° 4 Insertion d'un voxel avec détection de la compatibilité des motifs

Données : Un espace mémoire où écrire les données mémorisées
Données : Un voxel à insérer, défini par **x**, **y**, **z**, $\mathbf{r_z}$ et **col**
Résultat : La zone mémoire résultat contient le nouveau voxel inséré à la bonne place

```
Cherche l'adresse des données de la ligne correspondant à y
Si la ligne ne contient pas de plans utilisés, alors
    Créer le plan de profondeur z dans la ligne y
    Créer un motif à la coordonnée x et de taille 1 dans ce plan
    Copier r_z et col dans le voxel de ce motif
Sinon
    Rechercher, dans la ligne le plan correspondant à la profondeur z du voxel ou la position à laquelle ce plan devrait se trouver, s'il n'existe pas
    Si le plan correspondant n'existe pas, alors
        Créer un nouveau plan à la profondeur r_z du voxel
        Créer un motif de coordonnée x et de taille 1 dans le plan
        Copier r_z et col dans le voxel de ce motif
        Insèrer le nouveau plan tel que les plans soient triés selon leur profondeur dans la ligne
    Sinon
        Rechercher, dans le plan, le motif dans lequel le voxel devrait être inséré ou la position où il devrait se trouver, s'il n'existe pas
        Si le motif correspondant n'existe pas, alors
            Créer un motif de coordonnée x et de taille 1 dans le plan
            Copier r_z et col dans le voxel de ce motif
            Insèrer le nouveau plan tel que les plans soient triés selon leur profondeur dans la ligne
        Sinon
            Calculer la position où le voxel devrait se trouver à l'intérieur du motif
            Si un voxel existe déjà dans le motif à cette position, alors
                Si le nouveau voxel est plus près que l'ancien (si sa valeur r_z est inférieure), alors
                    Si le nouveau voxel est compatible avec le motif en terme de scène secondaire, alors
                        Copier r_z et col dans le voxel, en effaçant l'ancien
                    Sinon
                        Diviser le motif en deux : un motif contenant les voxels à gauche du voxel inséré et un motif contenant ceux à droite
                        Créer un nouveau motif contenant uniquement le nouveau voxel et l'insérer entre les deux motifs ci-dessus
                    Fin Si
                Fin Si
            Sinon
                Insère le voxel dans le motif
                Si le voxel est le dernier (resp. le premier) du motif et touche le premier (resp. le dernier) voxel du motif suivant (resp. précédent) et est compatible en terme de scène secondaire, alors
                    Fusionner les deux motifs
                Fin Si
            Fin Si
        Fin Si
    Fin Si
Fin Si
```

D'autres informations, comme un décalage d'angle de rendu, peuvent être rajoutées pour permettre de modifier la manière dont cette scène secondaire sera affichée pour

Algorithme n° 5 Mise en forme des données avec gestion des sous-scènes

Données : Zone mémoire contenant les voxels codés avec l'algorithme n°4
Données : Nombre de lignes du dispositif d'affichage 3D auquel les données sont destinées
Données : Couleur de fond de la scène à coder
Données : Zone mémoire où écrire les données

```
1: Écrire la couleur de fond de la scène
2: Pour chaque ligne du dispositif de haut en bas, faire
3:     Écrire le nombre de plans utilisés par cette ligne
4:     Pour chaque plan de la ligne du plus lointain au plus proche, faire
5:         Écrire la profondeur du plan
6:         Écrire le nombre de motifs de ce plan
7:         Pour chaque motif du plan de gauche à droite, faire
8:             Écrire la coordonnée x du voxel le plus à gauche du motif
9:             Si le motif contient une sous-scène, alors
10:                Écrire le nombre de voxels de ce motif avec le bit de poids fort à 1
11:                Laisser un espace pour l'offset des données de la sous-scène
12:                Laisser un espace (le cas échéant) pour le modificateur de projection (voir texte)
13:            Sinon
14:                Écrire le nombre de voxels de ce motif
15:            Fin Si
16:            Pour chaque voxel du motif de gauche à droite, faire
17:                Écrire la couleur (ARGB) du voxel
18:            Fin Pour
19:        Fin Pour
20:    Fin Pour
21: Fin Pour
```

Algorithme n° 6 Mise en forme des données des sous-scènes

Données : Zone mémoire contenant les voxels codés avec l'algorithme n°4
Données : Couleur de fond de la sous-scène à coder
Données : Zone mémoire où écrire les données

```
1: Écrire la couleur de fond de la sous-scène
2: Écrire le nombre de plans utilisés par cette sous-scène {Rappel : une sous-scène ne contient qu'une
   seule ligne}
3: Pour chaque plan de la sous-scène du plus lointain au plus proche, faire
4:     Écrire la profondeur du plan
5:     Écrire le nombre de motifs de ce plan
6:     Pour chaque motif du plan de gauche à droite, faire
7:         Écrire la coordonnée x du voxel le plus à gauche du motif
8:         Si le motif contient une sous-scène, alors
9:             Écrire le nombre de voxels de ce motif avec le bit de poids fort à 1
10:            Laisser un espace pour l'offset des données de la sous-scène
11:            Laisser un espace (le cas échéant) pour le modificateur de projection (voir texte)
12:        Sinon
13:            Écrire le nombre de voxels de ce motif
14:        Fin Si
15:        Pour chaque voxel du motif de gauche à droite, faire
16:            Écrire la couleur (ARGB) du voxel
17:        Fin Pour
18:    Fin Pour
19: Fin Pour
```

Algorithme n° 7 Codage d'une scène contenant des sous-scènes

Données : Un espace mémoire où écrire les données mémorisées
Données : Une scène 3D à coder
Résultat : La zone mémoire contient les voxels de la scène 3D à coder avec les scènes secondaires éventuelles

```
1: Coder la scène principale, en utilisant les algorithmes n°4 et n°5
2: Si la scène contient des objets présentant des effets devant être codés avec une scène secondaire (réflexion, réfraction ou autre), alors
3:    Pour chaque facette devant contenir une scène secondaire, faire
4:       Coder la ligne correspondant à la scène secondaire, en utilisant les algorithmes n°4 et n°6, et placer les données après la fin du codage de la scène 3D principale
5:       Si cette sous-scène contient des sous-scènes, alors
6:          Recommencer avec les facettes qui contiennent des sous-scènes {Note : cette récursion ne doit pas être infinie ; elle doit être arrêtée si un seuil est dépassé}
7:       Fin Si
8:       Noter le(s) motif(s) contenant cette scène secondaire et indiquer l'adresse du début de son codage
9:    Fin Pour
10: Fin Si
```

permettre à plusieurs motifs d'afficher cette scène secondaire de manière légèrement différente. Ceci est facilement compris en visualisant un cylindre vertical réfléchissant. Chaque motif représentant une partie de ce cylindre réfléchira la même chose que les autres motifs de la même ligne de l'écran, mais avec un angle différent. Pour éviter de devoir créer une scène secondaire indépendante pour chaque motif, nous pouvons en créer une seule et préciser, pour chaque motif, un décalage des motifs de la scène secondaire permettant de modifier cet angle.

Au niveau du codage, chaque scène secondaire est codée sous la forme d'une ligne unique à la fin des données envoyées au dispositif d'affichage, précédée de la couleur de fond. Au moment du décodage, la couleur de fond de la scène est ajoutée à la couleur des voxels du motif, ce qui permet de rendre des effets comme l'éclairage spéculaire. Le décodage doit juste permettre une réentrance du décodage pour décoder la scène secondaire dans l'espace réduit du motif qui la contient. Cet algorithme est indiqué sous la dénomination algorithme n°8 : **Reconstitution des vues avec gestion des scènes secondaires**.

Cet algorithme permet de coder toutes sortes d'effets visuels, comme la réflexion ou la réfraction, mais aussi des effets de diffraction. Ces différents effets peuvent être cumulés en ajoutant, à la même profondeur que le motif initial, dans la scène secondaire,

Algorithme n° 8 Reconstitution des vues avec gestion des scènes secondaires

Données : Zone mémoire (ou fichier) contenant les voxels codés avec l'algorithme n°7

Données : Résolution du dispositif d'affichage 3D (ou de la vue à reconstituer) pour lequel le fichier a été codé

Données : Angle correspondant à la vue à reconstituer, défini par une valeur entière, permettant de déterminer le décalage des motifs en fonction en fonction de leur profondeur

Données : Zone mémoire de taille suffisante pour recevoir l'image 2D résultat ayant la résolution définie ci-dessus, c'est-à-dire largeur de l'écran × hauteur de l'écran × taille utilisée par un pixel (par exemple 24 bits)

Résultat : La vue correspondant à l'angle passé en paramètre des données codées selon l'algorithme n°1 est écrite dans la zone mémoire résultat

1: Remplit la mémoire résultat avec la couleur de fond
2: **Pour chaque** ligne de l'écran **, faire**
3: Décode la ligne de l'écran en utilisant l'algorithme 9 : **Reconstitution d'une ligne avec gestion des scènes secondaires** avec comme paramètres :
   - la zone mémoire contenant les voxels
   - l'indice où le codage de la ligne à décoder commence
   - la résolution du dispositif
   - angle correspondant à la vue à reconstituer
   - la zone mémoire résultat où écrire le résultat du décodage de la ligne (une partie de la zone mémoire résultat pour la vue)
4: **Fin Pour**

un motif contenant également une scène secondaire.

L'inconvénient de cette modification est que, comme les informations des scènes secondaires peuvent être utilisées par plusieurs motifs différents, le décodage n'est plus linéaire en fonction de la taille des données. Si chaque scène secondaire n'est utilisée que par un seul motif, la complexité de l'algorithme de décodage reste en o(n), où n est la taille des données codées, scène(s) secondaire(s) comprise(s). Dans le cas contraire, la complexité de l'algorithme de décodage est en o(n+m×t), où

- n est la taille de l'ensemble des données, scènes secondaires comprises,
- m est la taille moyenne des scènes secondaires et
- t est le nombre moyen de motifs utilisant la même scène secondaire.

Cet algorithme permet de coder de manière réaliste des effets particuliers comme la réfraction, la réflexion et l'éclairage spéculaire. De plus, il permet également le relocage, c'est-à-dire le placement de scènes 3D dans des fenêtres placées dans une scène 3D ou 2D. Il est possible que cet algorithme soit utilisable pour d'autres applications. Par exemple, pour simuler une diffraction, il est possible d'ajouter à un même objet plusieurs scènes

Algorithme n° 9 Reconstitution d'une ligne avec gestion des scènes secondaires

Données : Zone mémoire contenant toute la scène, codée avec l'algorithme n°7
Données : indice, dans la zone mémoire, où le codage de la ligne à décoder commence
Données : Résolution du dispositif d'affichage 3D (ou de la vue à reconstituer)
Données : Angle correspondant à la vue à reconstituer
Données : Zone mémoire de taille suffisante pour recevoir l'image résultat ayant la résolution définie ci-dessus (largeur × hauteur × taille d'un pixel, par exemple 24 bits)
Résultat : La ligne de la vue calculée est écrite dans la zone mémoire résultat
Données : **plane_offset**, **pattern_offset**, **alpha**, **old_pixel** et **adresse** sont des variables locales de travail

```
1:  Pour chaque plan de cette ligne , faire
2:    plane_offset ← angle × profondeur du plan
3:    Pour chaque motif de ce plan , faire
4:      pattern_offset ← plane_offset + position horizontale du motif
5:      Si ce motif contient une scène secondaire , alors
6:        Créer une zone mémoire temporaire ayant une taille égale à la largeur du motif
7:        adresse ← adresse de la scène secondaire
8:        Lire la couleur de fond de la scène secondaire
9:        Pour chaque voxel de ce motif , faire
10:         Remplir la zone mémoire temporaire avec (couleur du voxel + la couleur du fond de la scène secondaire)
11:       Fin Pour
12:       Décoder la scène secondaire, en utilisant l'algorithme 9 avec les paramètres mémoire contenant les données, adresse, largeur du motif, angle, zone mémoire temporaire
13:       Pour chaque voxel de ce motif , faire
14:         alpha ← canal alpha du voxel (nombre réel) ramené dans l'intervalle [0, 1]
15:         old_pixel ← mémoire résultat[pattern_offset + numéro du voxel, y]
16:         Pour chaque composante R, G et B , faire
17:           composante du pixel de coordonnées (pattern_offset + voxel number, y) dans la zone mémoire résultat ← (alpha × composante du pixel à la position voxel number dans la zone mémoire temporelle) + ( (1 - alpha) × composante de old_pixel)
18:         Fin Pour
19:       Fin Pour
20:     Sinon
21:       Pour chaque voxel de ce motif , faire
22:         alpha ← canal alpha du voxel étendu à un nombre réel, ramené dans l'intervalle [0, 1]
23:         old_pixel ← pixel de coordonnées (pattern_offset + numéro du voxel, y) dans la zone mémoire résultat
24:         Pour chaque composante R, G et B , faire
25:           composante du pixel de coordonnées (pattern_offset + voxel number, y) dans la zone mémoire résultat ← (alpha × composante du voxel) + ( (1 - alpha) × composante de old_pixel)
26:         Fin Pour
27:       Fin Pour
28:     Fin Si
29:   Fin Pour
30: Fin Pour
```

secondaires représentant la même chose, mais avec des couleurs différentes.

2.4.4 Effacement des voxels inutiles

Comme nous l'avons dit précédemment, tous les voxels des objets doivent être codés pour permettre de voir ces objets sous des angles différents. Cependant, tous ces voxels ne sont pas utiles. En effet, un cube, par exemple, ne pourra pas montrer plus de trois de ses faces à l'observateur, et ce, quelle que soit la position de celui-ci. De plus, certains objets ou parties d'objets peuvent être complètement invisibles. Par exemple s'ils se trouvent à l'intérieur d'un autre objet opaque. Tous les voxels invisibles prennent donc de la place inutilement dans les données envoyées au dispositif d'affichage et ralentissent inutilement la phase de décodage. Il nous semble donc utile de pouvoir effacer ces voxels inutiles. Cependant, les algorithmes habituellement utilisés, comme l'algorithme du Z-buffer [28], qui consiste à mémoriser la profondeur de chaque pixel de l'image finale et, au cas où deux pixels doivent être dessinés à la même position sur l'écran, à ne prendre que le plus proche, ne peuvent être utilisés. En effet, à cause du nombre de points de vue différents possibles, de tels algorithmes feraient disparaître des voxels qui sont visibles pour certaines positions d'observation. La figure 2.8 illustre ce problème : en utilisant l'algorithme du Z-buffer, les motifs b et c seraient effacés car cachés par le motif a. Or, si ces deux motifs sont effectivement cachés pour l'observateur 2, le motif c est visible pour l'observateur 3. Seul le motif b est effectivement caché par le motif a quelle que soit la position d'observation et doit être effacé.

Pour cela, nous avons mis au point un algorithme permettant de détecter et d'effacer ces voxels, sans effacer les autres : l'algorithme n°10 : **Effacement des voxels cachés**. Cet algorithme (voir figure 2.9) devra prendre place entre la phase de mémorisation et la phase de mise en forme des données.

Cet algorithme part du principe que chaque motif ne peut cacher que des voxels placés derrière lui et sur la même ligne. Chaque motif crée donc une sorte de "cône d'ombre" dans lequel les voxels ne sont pas visibles. La figure 2.10 illustre, de manière simplifiée les différentes raisons pour lesquelles des voxels ne sont pas visibles. Sur ce dessin, le grand triangle en pointillés représente la zone de visualisation, c'est-à-dire ce qui est représenté par le dispositif d'affichage, l'observateur se trouvant en bas du dessin.

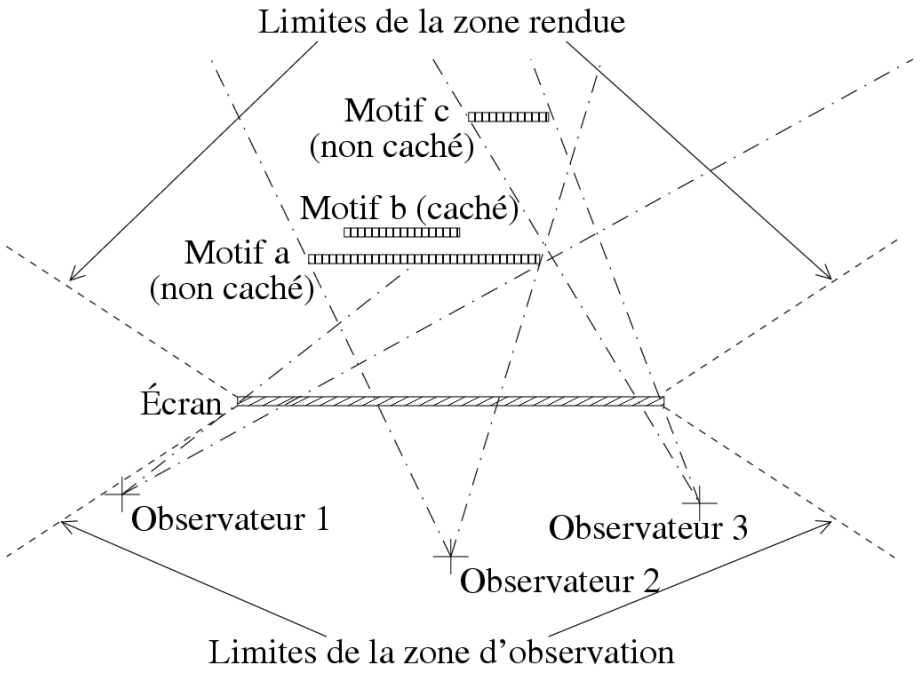


FIG. 2.8 – Différence entre les motifs cachés selon certains points d'observation et ceux réellement cachés

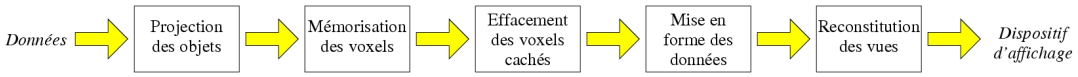


FIG. 2.9 – Phases de l'algorithme de compression

Les traits pleins horizontaux représentent des motifs et les traits horizontaux pointillés représentent les plans utilisés par ces motifs. Le trait horizontal hachuré en haut du schéma représente le fond de l'affichage, c'est-à-dire la couleur de fond ou le plan ayant la profondeur la plus grande possible. Les petits triangles pointillés représentent les cônes d'ombre des différents motifs représentés. Les traits horizontaux rouges représentent les parties des motifs qui se trouvent dans les cônes d'ombre et donc non visibles. Ces motifs ou parties de motifs sont donc inutiles et doivent être retirés. Comme on le voit sur la figure 2.10, dans certains cas, c'est tout le motif qui doit être effacé. Dans d'autres, il s'agit juste de réduire et, parfois déplacer un motif, si cette réduction s'opère à gauche. Enfin, dans les autres cas, le motif est coupé en deux. Il est alors nécessaire de créer un nouveau motif.

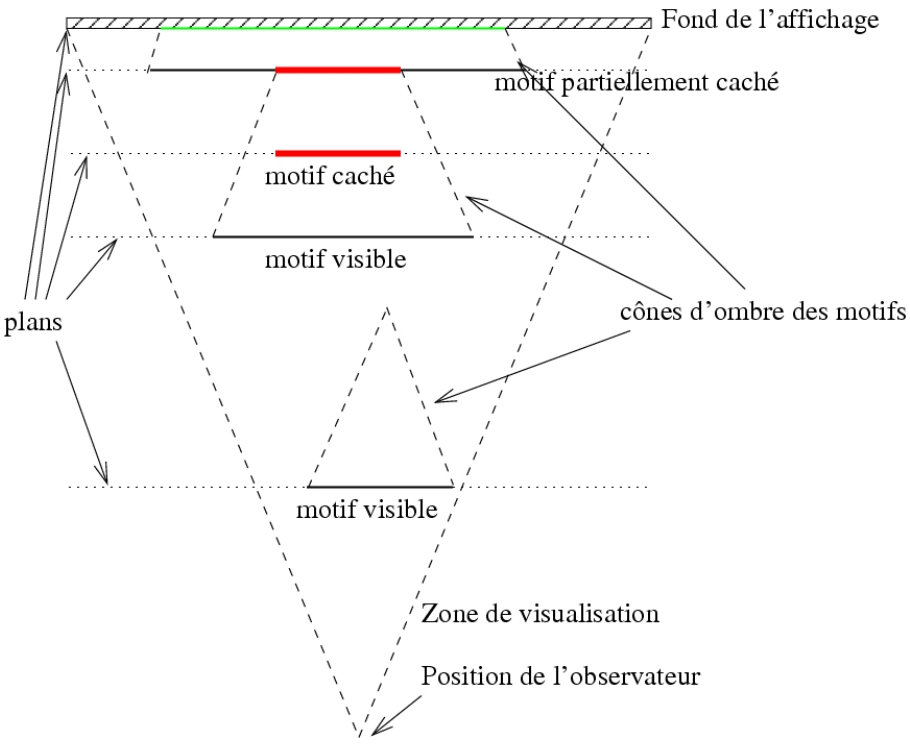


FIG. 2.10 – "Cône d'ombre" générés par les motifs

Nous appellerons ces zones où les voxels sont complètement cachés des "masques". La forme et la taille des masques sont déterminées par la position de l'écran de projection et des angles possibles extrêmes de visualisation. Le masque est alors un triangle ou un parallélépipède (ces formes, ainsi que la manière dont elles sont déterminées sont montrées figure 2.11) dont la limite inférieure est le motif qui l'a généré et les cotés des demi-droites partant des extrémités de ce motif et s'éloignant de l'écran et dont la direction est celle partant des extrémités de l'écran pour arriver aux extrémités du motif. Dans le cas où un masque a une forme de parallélépipède, sa limite supérieure est le plan ayant la profondeur la plus grande de la ligne. Au-delà, il devient inutile.

Chaque masque est défini par la profondeur z du motif qui l'a généré, par les coordonnées $x1$ et $x'1$ de ce motif et des angles α et β calculés à partir de $x0$, $x1$ et z pour α et $x'0$, $x'1$ et z pour β, comme définis sur la figure 2.12.

Pour simplifier les calculs, ces masques peuvent également être définis par la coordonnée x de l'intersection entre les deux demi-droites définissant les limites du masque

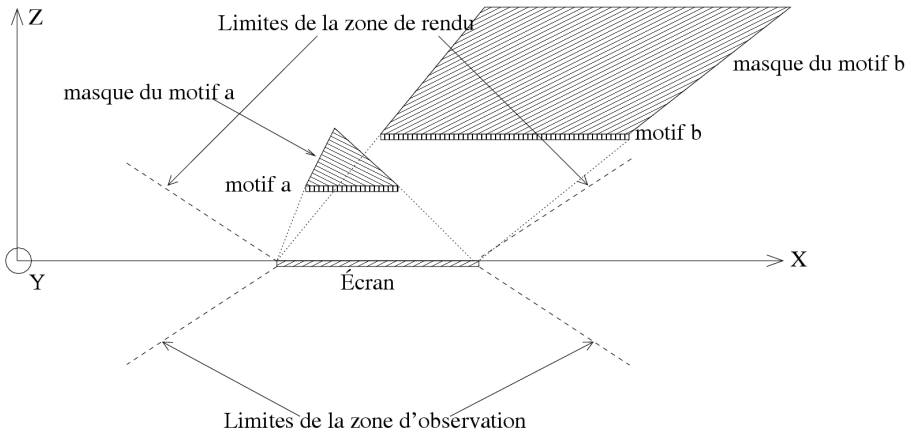


FIG. 2.11 – Formes de masques générés par les motifs

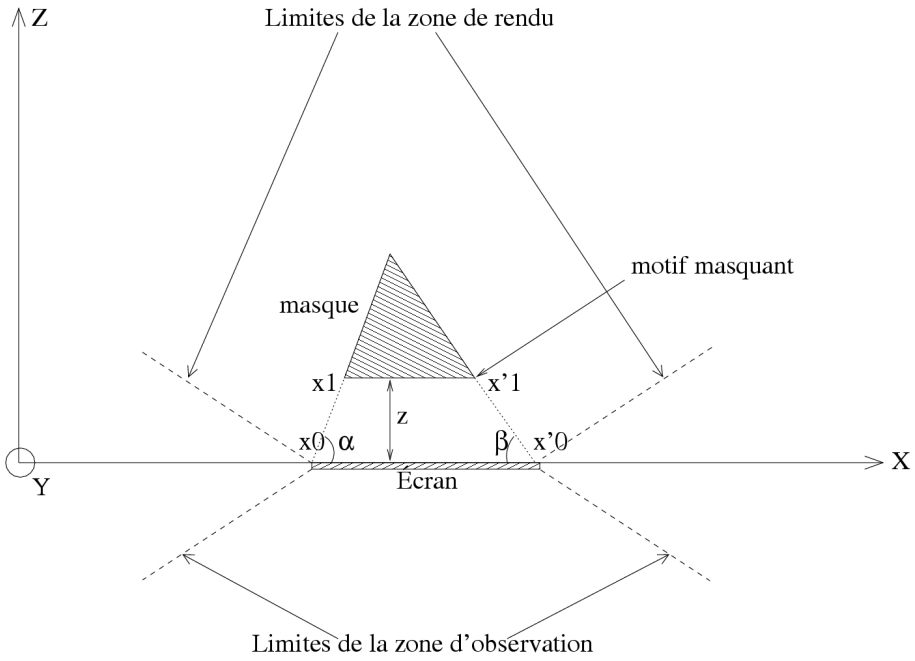


FIG. 2.12 – Définition des masques générés par les motifs

avec le plan de profondeur maximale possible, notés $x''1$ et $x''2$ et les coordonnées $x1$, $x'1$ et z du motif l'ayant généré. La figure 2.13 illustre cette notion.

L'algorithme complet d'**effacement des voxels cachés** est indiqué sous la référence algorithme n°10. Il consiste donc à prendre, les uns après les autres, tous les motifs en commençant par les motifs situés sur les plans les plus proches de l'observateur. Pour chaque motif, on calcule le masque associé. Si le motif contient un ou plusieurs

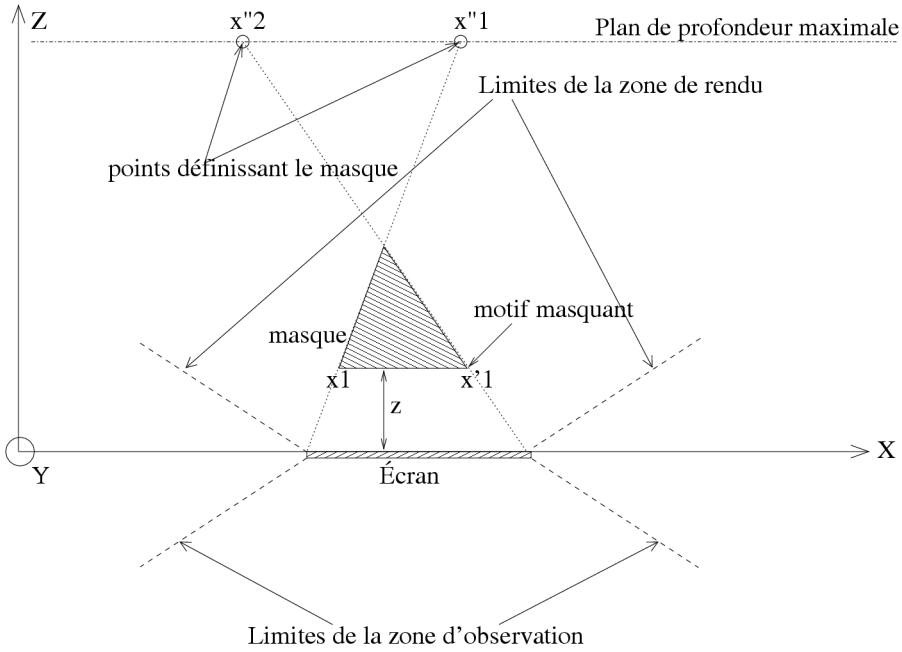


FIG. 2.13 – Autre définition des masques générés par les motifs

Algorithme n° 10 Effacement des voxels cachés

Données : Un espace mémoire contenant les voxels codés avec l'algorithme n°7
Résultat : La zone mémoire résultat ne contient plus que les voxels visibles

```
1: Pour chaque ligne, faire
2:   Pour tous les plans de la ligne, du plus proche au plus lointain, faire
3:     Pour chaque masque, faire
4:       Si l'intersection du plan avec la demi-droite x1x''2 se trouve plus à gauche que intersection
         du plan avec la demi-droite x'1x''1, alors
5:         Efface le masque de la liste des masques
6:       Sinon
7:         Calculer l'intersection entre le masque et le plan
8:         Effacer tous les voxels à l'intérieur de cette intersection
9:       Fin Si
10:    Fin Pour
11:    Calculer les masques de tous les motifs (ou de groupes de voxels opaques contigus) restant
       dans le plan
12:    Pour chaque masque nouvellement créé, faire
13:      Si le masque touche un autre masque, alors
14:        Fusionner les deux masques
15:      Fin Si
16:    Fin Pour
17:  Fin Pour
18: Fin Pour
```

voxels transparents, ceux-ci doivent être considérés comme inexistants car ils ne peuvent

rien masquer. Si un motif contient des voxels opaques séparés par des voxels transparents, chaque groupe de voxels opaques contigus doit être considéré comme un motif indépendant, générant son propre masque. On passe ensuite au plan suivant. On calcule l'intersection de chaque masque avec le plan. Si, pour un des masques, l'intersection du plan avec la demi-droite $x1x''2$ se trouve plus à gauche que l'intersection du plan avec la demi-droite $x'1x''1$, le masque n'existe plus au niveau de ce plan, ni au niveau des suivants. Il peut donc être effacé. Ensuite, on efface tous les voxels de ce plan se trouvant dans un masque déjà défini, c'est-à-dire dont la coordonnée x est comprise entre les intersections des deux demi-droites $x1x''2$ et $x'1x''1$ avec le plan. Chaque motif du plan génère un nouveau masque. Si un des nouveaux masques touche un autre masque, ceux-ci fusionnent pour former un masque unique. Celui-ci est défini par les deux demi-droites (notées $x1x''1$ et $x'2x''2$ sur la figure 2.14) telles que définies ci-dessus, une pour l'extrémité de chaque motif, de sorte que le masque résultant est équivalant au masque généré par la fusion du motif du plan et de l'intersection entre le premier masque et le plan.

La structure de données que nous estimons la plus adaptée à l'implémentation de cet algorithme est un tableau de coordonnées servant à mémoriser les coordonnées $x''1$ et $x''2$ (telles que définies dans les figures 2.12 et 2.13) de chaque masque et les coordonnées des intersections des droites $x1x''1$ et $x'1x''2$ (telles que définies dans les figures 2.12 et 2.13) avec les plans. Sa complexité est en $o(l \times p \times q \times m)$, où :

- l est le nombre de lignes de l'image 3D
- p est le nombre moyen de plans par ligne
- q est le nombre moyen de masques par plan
- m est le nombre moyen de motifs par plan

Elle peut également se définir en $o(m \times r)$, où m est le nombre total de motifs dans les données à traiter et r est le nombre moyen de plans ayant une intersection avec chaque masque.

En résumé, les différentes phases de la chaîne de traitements des données 3D (voir

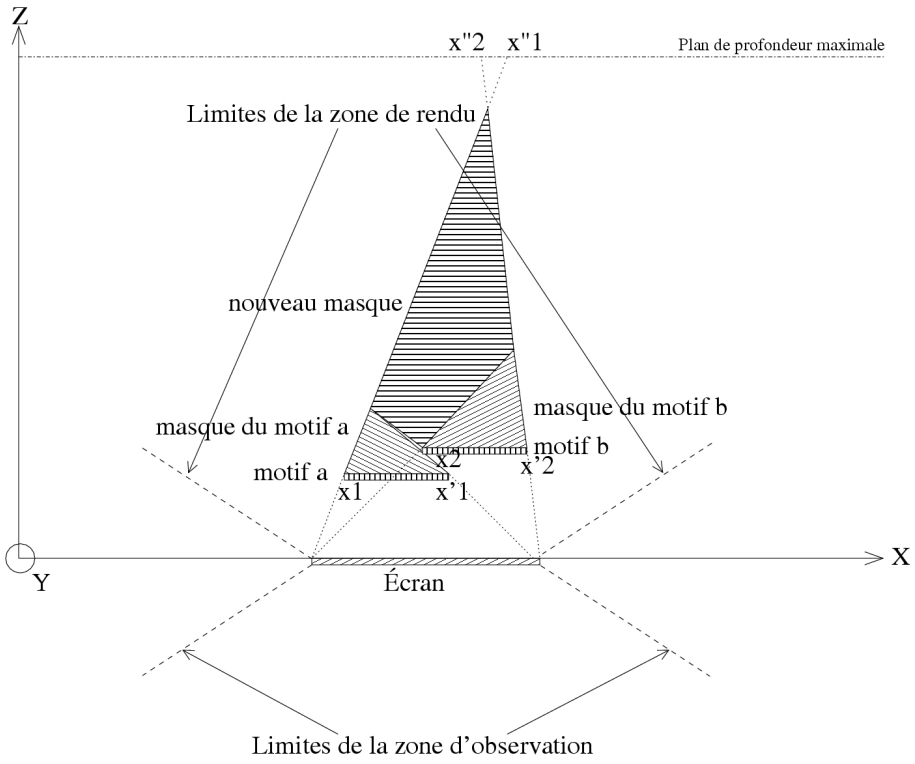


FIG. 2.14 – Exemple de fusion de deux masques

figure 2.9) sont donc constituées des algorithmes suivants :

- Projection des voxels – Il s'agit des algorithmes de projections des objets et de générations des voxels, comme ceux implémentés sur une carte accélératrice 3D.
- Mémorisation des voxels – Il s'agit des algorithmes de codage des voxels, c'est-à-dire les algorithmes n°1 et 4.
- Effacement des voxels cachés – Il s'agit de l'algorithme n°10.
- Mise en forme des données – Il s'agit des algorithmes n°2, 5, 6 et 7.
- Reconstitution des vues – Il s'agit des algorithmes n°3, 8 et 9.

2.4.5 Conclusion de l'algorithme de compression des données 3D

Nous avons proposé plusieurs algorithmes permettant de coder les données 3D d'une scène quelconque, puis de restituer toute une série de vues de cette scène 3D. Ces algorithmes permettent de résoudre les trois problèmes que nous avons posés précédemment.

En effet, l'algorithme n°7 permet de coder toutes les informations relatives à la scène en ne projetant les objets qui la composent qu'une seule fois, ce qui permet de diminuer le temps de calcul des projections et du rendu des objets d'un facteur égale au nombre de vues différentes affichées pour chaque image 3D, c'est-à-dire 200. Le temps nécessaire à ces projections revient donc dans des valeurs acceptables et compatibles avec la contrainte temps réel qui nous imposait la génération des données nécessaires à l'affichage d'une scène 30 fois par seconde.

Ces données n'étant codées qu'une seule fois, nous évitons ainsi les informations redondantes qui sont très nombreuses lorsqu'on mémorise les différentes vues sans compression. De plus, les espaces vides (entre ou dans les objets) ne sont pas mémorisés, ce qui diminue grandement la quantité de données à mémoriser. Ceci permet donc de diminuer l'espace mémoire nécessaire au stockage des données à afficher qui se situe alors à des valeurs inférieures à 20 méga-octets, ce qui est plus raisonnable.

Enfin, la taille des données à envoyer au dispositif d'affichage étant réduite, les débits nécessaires pour transférer ces données reviennent à des valeurs possibles : avec une taille de données inférieure à 20 méga-octets par image 3D, ce débit sera donc inférieur à $20 \times 30 = 600$ méga-octets par seconde, soit 4 800 méga-bits par seconde. Cette valeur reste importante, mais possible avec certaines technologies haut débit, comme le DVI-digital Dual-Link, qui peut atteindre des débits typiques de 9,9 giga-bits par seconde ou en compressant les données : les tests de compressions que nous avons réalisés montrent que les données générées ont été compressées avec l'algorithme de compression sans perte gzip [48] avec un taux de compression (statistiques affichées par la version 1.2.4 du logiciel linux gzip) variant de 69 à 95%, avec une moyenne de 90%, c'est-à-dire que les données compressées avec l'algorithme gzip représentent entre 5 et 30% (10% en moyenne) de la taille des données générées par l'algorithme n°7 (Codage d'une scène contenant des sous-scènes). En utilisant la compression gzip et la décompression gzip entre la chaîne de traitements et le dispositif d'affichage (voir figure 2.15), le débit nécessaire passe donc à une valeur moyenne de 480 méga-bits par seconde, en moyenne, ce qui est largement possible avec une connexion réseau gigabit Ethernet dédiée qui

permet une communication théorique maximale de 1 024 méga-bits par seconde.

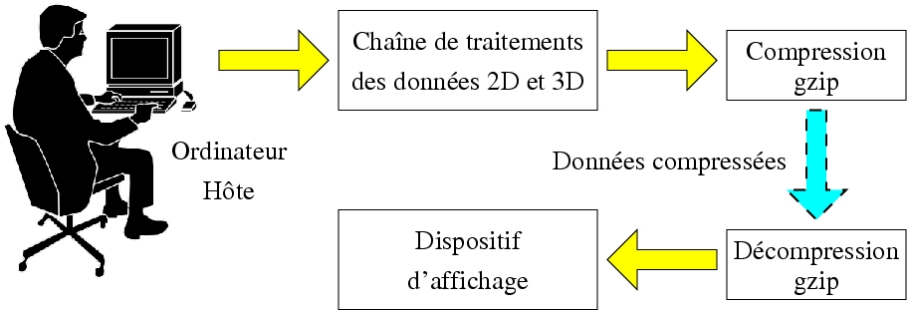


FIG. 2.15 – Utilisation de la compression/décompression gzip pour réduire la débit de données entre la chaîne de traitements et le dispositif d'affichage

2.5 Chaîne de traitements pour la génération d'images 3D compressées

Afin de générer les informations nécessaires à notre dispositif d'affichage, nous avons opté pour l'utilisation d'une chaîne de traitements pouvant traiter à la fois les données 2D et 3D envoyées par l'ordinateur hôte. Ceci permet de concevoir notre chaîne exactement comme une carte graphique du commerce et de la faire piloter par l'ordinateur hôte comme tel. De cette manière, il sera possible d'afficher sur notre dispositif d'affichage tous les programmes affichant quelque chose, qu'ils soient 2D ou 3D. Cette chaîne comporte donc deux pipe-lines parallèles : un pour traiter les données 3D et un pour traiter les données 2D. Les données générées par chacun de ces deux pipe-lines sont ensuite fusionnées et envoyées au dispositif d'affichage.

2.5.1 Génération des vues

Actuellement, il existe de très nombreuses cartes accélératrices 3D dans le commerce. Le problème est que, pour des raisons évidentes, les entreprises ne communiquent que très peu sur l'architecture de leurs produits. On peut cependant faire ressortir une architecture globale pour ces cartes. Cette architecture, représentée par la figure 2.16, est définie comme suit :

– Dans un premier temps, les données sont fournies à la carte par l'ordinateur hôte par l'intermédiaire d'un bus de communication, souvent un bus AGP, voire PCI ou ExpressPCI.
– Ensuite, ces données sont décodées et interprétées par la carte pour déterminer l'action demandée.
– Enfin, les données proprement dites sont envoyées au pipe-line de traitement des données 2D ou 3D selon le cas.

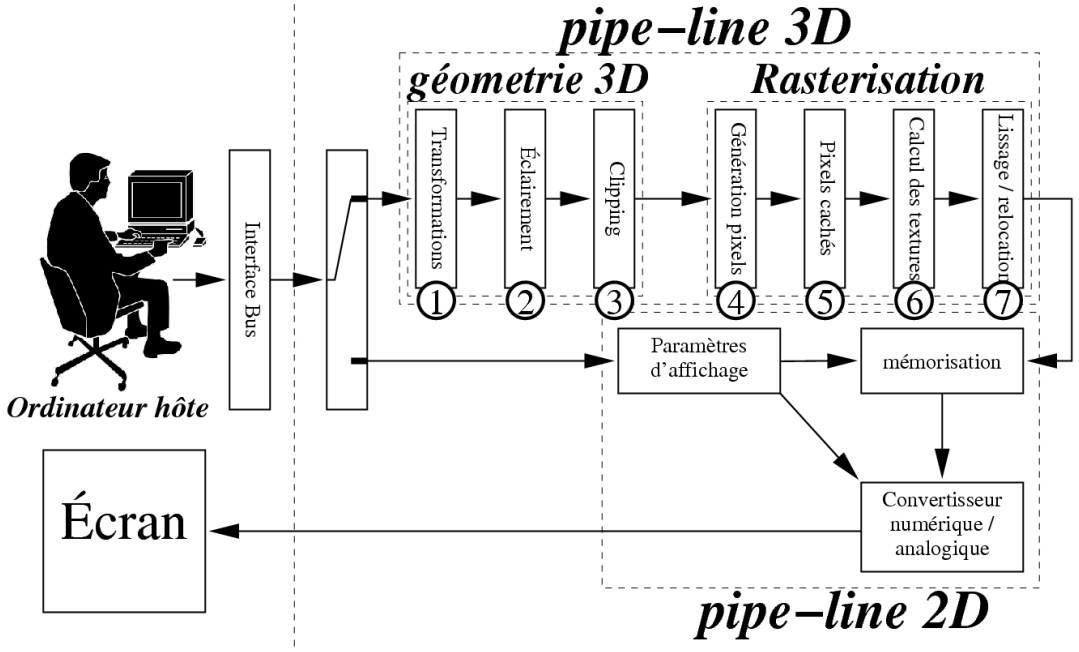


FIG. 2.16 – Principe des cartes 3D du commerce

Le pipe-line 3D se divise en deux parties : une première partie appelée *"géométrie 3D"*, s'occupant de tous les calculs de projection des points des objets sur l'écran, ainsi que les calculs d'éclairement, d'ombres, etc. et une seconde partie, dénommée par l'anglicisme *"rasterisation"*, qui calcule les pixels à afficher sur l'écran pour rendre cet objet. On peut schématiser l'ensemble du pipe-line 3D de la manière suivante :

– La première partie, *Transformations*, s'occupe des calculs de transformations 3D des points des objets à afficher et de leur projection sur l'écran virtuel. Cette partie calcule donc les coordonnées des points à dessiner sur l'écran.

– La seconde partie, *Éclairement*, fait les premiers calculs géométriques qui vont permettre, par la suite, de calculer la luminosité des différentes facettes de l'objet, en particulier, calculer le vecteur normal à la surface de chaque facette et le produit scalaire de ce vecteur normal avec le vecteur normé indiquant la direction de chaque source de lumière. Le produit scalaire ainsi calculé indique la quantité de lumière reçue par la facette de chacune des sources de lumière. Ce calcul ne prend cependant pas en compte les ombres portées par les objets, qui seront calculées dans la partie *rasterisation*.
– La troisième partie s'occupe du *"clipping"*. C'est-à-dire qu'elle détermine si les points projetés sur l'écran sont bien à l'intérieur de l'image qui va être dessinée. Trois cas de figure se présentent :

  1. L'objet dessiné est entièrement dans l'écran. Dans ce cas, on dessine cet objet sans calcul supplémentaire.
  2. L'objet dessiné est entièrement à l'extérieur de l'écran : par exemple, il est placé derrière l'observateur. Dans ce cas, l'objet en question n'est pas visible par l'observateur, il n'a donc pas besoin d'être dessiné et est totalement ignoré par les étages suivants du pipe-line.
  3. L'objet est partiellement visible. Dans ce cas, des calculs supplémentaires sont nécessaires pour déterminer la partie de l'objet qui est visible, ainsi que sa position sur l'écran, en particulier, le point d'intersection entre le dessin des bords des facettes avec le bord de l'écran. Une fois ces informations connues, la zone en question peut être dessinée.

  Ce traitement ne prend pas en compte le cas où l'objet projeté serait caché par un autre objet déjà dessiné. Ce cas sera traité dans la cinquième partie du pipe-line : *Pixels cachés*.
– La quatrième partie du pipe-line 3D, appelée *Génération pixels*, va générer les pixels, sans calculer leur couleur, c'est-à-dire qu'elle va déterminer l'ensemble des pixels à afficher sur l'écran pour dessiner l'objet, mais pas encore les dessiner. Pour chacun de ces pixels, cette partie détermine un certain nombre de paramètres

(comme, par exemple, la coordonnée de la texture correspondant à ce voxel) qui permettront, par la suite, de déterminer la couleur de ce pixel, ainsi que sa profondeur.

- La cinquième partie, *Pixels cachés*, va déterminer quels pixels, parmi ceux générés par l'étage précédent du pipe-line, sont effectivement visibles par l'observateur. Ceci est réalisé en comparant la profondeur du pixel généré avec celle du pixel déjà dessiné à cette position de l'écran, s'il existe. Dans ce cas, si le nouveau pixel est plus lointain, il sera caché par celui déjà affiché. Il ne doit donc pas être affiché et ce point de l'écran devra rester inchangé. Si aucun pixel n'a déjà été affiché à cette position ou si le pixel en mémoire est plus loin de l'observateur que celui que l'on veut afficher, celui-ci sera dessiné à la place du précédent qui sera effacé.
- La sixième partie, *Calcul des textures*, va déterminer, pour chacun des pixels qui vont effectivement être affichés, la couleur et la transparence. Ces informations sont obtenues à partir des informations de texture et de couleur des objets, mais aussi des informations comme les ombres portées, les reflets des lumières et la réflexion des objets.
- La septième partie, appelée *Lissage / relocation*, affine l'affichage des objets en *"lissant"* les objets, c'est-à-dire en ajoutant des pixels sur les bords des objets ou des lignes afin d'éviter les effets d'"escalier", aussi appelés "crénelage" ou "aliasing" en anglais.
- Enfin, ces informations de couleurs des pixels sont ensuite écrites directement dans la mémoire 2D de la carte graphique, en respectant les informations de position et de recouvrement des fenêtres, et en prenant en compte la transparence des pixels dans le cas d'objets transparents ou à cause de l'anti-crénelage. Ces traitements sont opérés par la huitième et dernière étape du pipe-line 3D qui va également se charger de calculer l'adresse à laquelle écrire la valeur du pixel.

Le pipe-line 2D est, quant à lui, beaucoup plus simple. En effet, il ne se compose que de deux parties :

– La première partie, *Paramètres d'affichage*, sert de gestion des paramètres d'affichage : résolution de l'écran, fréquences de rafraîchissement, etc.
– La seconde partie, *mémorisation*, sert à mémoriser les pixels à afficher. Elle contient toutes les informations des données 2D à afficher, mais également les données 3D calculées par le pipe-line 3D qu'il écrit directement dans la mémoire 2D sous la forme de pixels.
– Les données mémorisées sont ensuite lues et converties par le convertisseur numérique/analogique contrôlé par les paramètres d'affichage, qui va générer le signal vidéo envoyé à l'écran.

Toutes les cartes 3D du commerce ne correspondent sûrement pas exactement à cette architecture. Il est même possible qu'aucune ne lui corresponde exactement. Cependant, elle représente une manière d'implémentation matérielle des traitements nécessaires à un affichage 3D en temps réel qui est adaptée aux modifications que nous devons apporter à ce type de chaîne pour implémenter la génération des données 3D codées par l'algorithme n°7. On entend par "affichage 3D temps réel" un affichage d'au moins 30 images par seconde en plein écran pour une résolution d'au moins 1 024 par 768 pixels avec une profondeur d'au moins 24 bits par pixel. Tous les algorithmes utilisés ici sont tous connus et largement implémentés et documentés.

La figure 2.17 montre la chaîne de traitement modifiée pour gérer notre codage des données 3D en indiquant quelles parties ont été conservées et quelles parties ont été modifiées ou ajoutées.

Notre adaptation à cette chaîne se limite au pipe-line 2D et aux étages terminaux du pipe-line 3D. En effet, toute la partie communication avec le bus de l'ordinateur hôte et décodage des commandes n'a aucun besoin d'être modifié car les données 2D et 3D envoyées à notre chaîne de traitements sont les même que celles envoyées à une carte accélératrice 3D "classique".

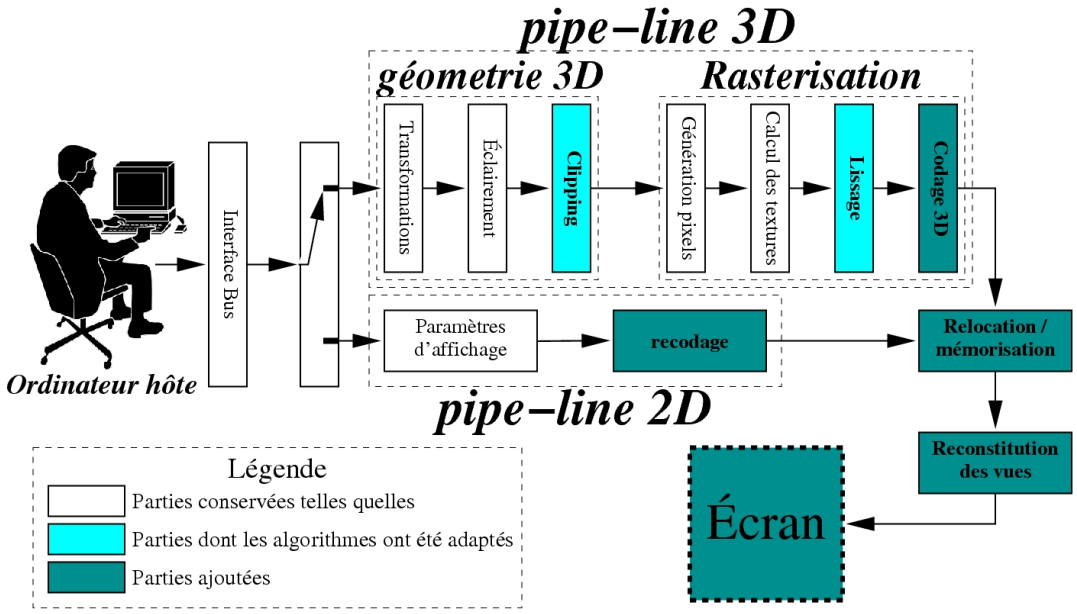


FIG. 2.17 – Modifications apportées à la chaîne de traitements pour y adapter nos traitements

2.5.2 Compression des données 3D

Toute la partie *"géométrie 3D"* permettant de faire les calculs de transformation des coordonnées des points des objets, de luminosité des facettes, du *"clipping"* et de la projection des pixels sur l'écran pourront également être conservées telles quelles ; seul les paramètres du *"clipping"* devront être modifiés afin de prendre en compte la largeur de l'angle de vue pour prendre en compte tous les points de vue possibles pour l'observateur, mais les algorithmes resteront complètement inchangés. De même les deux premiers étages de la partie *"rasterisation"* (*"Génération des pixels"* et *"Calcul des textures"*) permettant de générer les pixels à afficher n'ont besoin d'aucune modification d'algorithme.

La partie *"lissage"* sera, comme pour la partie *"clipping"*, conservée avec le même algorithme. Par contre, à la différence de la partie *"clipping"*, l'algorithme sera adapté dans le sens que, sur les cartes 3D du commerce, les pixels rajoutés sont créés en fonction du pixel déjà mémorisé afin de simuler un effet de transparence, alors qu'ici, nous allons créer des pixels partiellement transparents (en utilisant leur canal *alpha*) et nous allons

les intégrer comme les autres pixels, avec leur couleur et leur profondeur, comme s'il s'agissait des pixels générés par le dessin d'un objet. La complexité de cet algorithme sera donc la même que celle de l'algorithme de lissage choisi.

Tous les voxels générés (par l'étage *"Génération pixel"* et par l'étage *"Clipping"*) seront ensuite simplement codés en utilisant l'algorithme n°7, tâche que devra réaliser le dernier étage du pipe-line 3D : l'étage *Codage 3D*.

2.5.3 Gestion des données 2D

Le même problème se pose avec les informations 2D à afficher : elles doivent être codées dans l'espace 3D du dispositif d'affichage selon le même algorithme n°7. Le second étage du pipe-line 2D qui s'occupait de mémoriser les données à afficher a donc été remplacé par un codage permettant aux données 2D d'être affichées par notre afficheur 3D : l'algorithme n°11 : **l'algorithme de recodage des données 2D**. Le premier étage de ce pipe-line n'a quant à lui aucun besoin d'être modifié.

Algorithme n° 11 algorithme de recodage des données 2D

Données : Largeur de l'écran (nombre de colonnes)
Données : Hauteur de l'écran (nombre de lignes)
Données : Un espace mémoire contenant les pixels à coder (de taille largeur de l'écran × hauteur de l'écran)
Données : Un espace mémoire où écrire le codage 3D des données 2D
Données : valZ : profondeur à laquelle faire apparaître l'image 2D dans l'environnement 3D
Résultat : La zone mémoire résultat contient une scène 3D représentant l'image 2D passée en entrée

1: **Pour chaque** ligne **, faire**
2: La ligne contient un plan de profondeur valZ et contenant un motif de taille largeur de l'écran
3: **Pour chaque** colonne **, faire**
4: La couleur du voxel numéro colonne du motif du plan de la ligne ← la couleur du pixel situé à la ligne *ligne* et à la colonne *colonne* de l'espace mémoire contenant les pixels à coder
5: Le canal alpha de ce voxel = 1 (parfaitement opaque)
6: **Fin Pour**
7: **Fin Pour**

L'algorithme de recodage utilisé dans le second étage du pipe-line 2D consiste à lire les données 2D ligne à ligne et d'en faire autant de motifs dont la taille sera égale à la largeur de l'écran. Chacun de ces motifs sera le seul motif de chaque ligne et sera placé dans un plan de profondeur 0, correspondant à la profondeur de la surface de l'écran, ou à une profondeur arbitraire, définie logiciellement ou matériellement, identique pour

toutes les lignes. Dans le cas d'un affichage uniquement 2D, la scène 3D générée par ce pipe-line contiendra donc un plan parallèle à l'écran, contenant les données 2D sous forme d'une image plane.

La complexité de cet algorithme est en o(largeur × hauteur), où largeur et hauteur sont respectivement le nombre de colonnes et de lignes de l'image 2D à recoder. En d'autres termes, sa complexité est en o(n), où n est le nombre de pixels à recoder.

2.5.4 Fusion des données 2D et 3D

Une fois que les données ont été codées, la partie de mémorisation va les conserver en mémoire jusqu'à ce que tous les objets soient traités. Cette partie va également gérer le problème de "relocation", en utilisant l'algorithme n°12 : **Relocation** pour que les images 3D soient dessinées dans la fenêtre où elles doivent être et que les données 2D et 3D puissent être affichées correctement sans interférer les unes avec les autres. En

Algorithme n° 12 Relocation

Données : Un espace mémoire contenant les données 2D recodées selon l'algorithme de recodage des données 2D n°11 et suffisamment de place libre pour y insérer les données des scènes 3D à insérer et les modifications (5 valeurs par fenêtre × hauteur de la fenêtre)

Données : Un espace mémoire contenant chacune des scènes à insérer dans les fenêtres avec les coordonnées de la fenêtre : coordonnées x et y du coin supérieur gauche + largeur + hauteur de la fenêtre

Résultat : La zone mémoire résultat ne contient plus que les voxels visibles

1: **Pour chaque** ligne **, faire**
2: **Pour chaque** scène secondaire à reloquer **, faire**
3: **Si** la scène secondaire occupe cette ligne ($y \leq ligne < y + hauteur$) **, alors**
4: Couper le(s) motif(s) de la ligne aux coordonnées x et x + largeur de la fenêtre - 1 afin de créer au moins deux nouveaux motifs
5: Recopier la ligne numéro *ligne* - y de la scène secondaire à la fin de l'espace mémoire contenant les données 2D
6: Le motif compris entre x et x + largeur de la fenêtre - 1 est noté comme contenant une scène secondaire et l'adresse de cette scène est l'adresse de la ligne qu'on vient de recopier
7: **Fin Si**
8: **Fin Pour**
9: **Fin Pour**

d'autres termes, elle va modifier la scène fournie par le pipe-line 2D et séparer les voxels pour que l'intérieur des fenêtres affichant des données 3D soit constitué d'un motif contenant une scène secondaire. Cette scène secondaire contiendra, quant à elle, la scène 3D à afficher dans la fenêtre en question.

L'algorithme indiqué comme algorithme n°12 est une version simplifiée de l'algorithme réel, car il ne permet pas à deux fenêtre de se chevaucher. L'algorithme complet est indiqué sous forme de code source écrit en C en annexe.

2.5.5 Décompression et restitution des vues

La restitution des vues se fait simplement par une lecture des données envoyées par l'étage *"Relocation/mémorisation"*. Comme le décodage se résume à quatre boucles imbriquées, il se limite à quatre compteurs, initialisés par les données lues : nombre de plans de chaque ligne, nombre de motifs dans chaque plan et nombre de voxels dans chaque motif. La recopie de la couleur des voxels se fait selon l'équation $C = \alpha \times C_v + (1 - \alpha) \times C$, où C est chaque canal (rouge, vert ou bleu) de la couleur du pixel résultat, C_v est le même canal de la couleur du voxel qui doit être copié sur le pixel résultat et α est le canal alpha (transparence) de ce voxel, sous forme d'une valeur entière comprise dans l'intervalle $]0; 1]$, 1 correspondant à un voxel complètement opaque et 0 un voxel complètement transparent. Dans le cas où le canal alpha est codé sous la forme d'un entier, l'équation devient $C = (\alpha \times C_v + (max_{alpha} - \alpha) \times C) >> N$, où max_{alpha} est la valeur maximale du canal alpha $>> N$ est un décalage binaire de N bits, c'est-à-dire du nombre de bits sur lequel le canal alpha est codé. Par exemple, si alpha est codé sur 8 bits, max_{alpha} est égale à 255 et N est égale à 8.

Comme nous l'avons vu précédemment, le décodage de chaque ligne est indépendant du décodage des autres. De même, le décodage de chaque vue est indépendant du décodage des autres. Par conséquent, il est très facile de paralléliser le décodage des vues, soit en décodant simultanément plusieurs vues et en les conservant dans des mémoires séparées avant de les afficher, soit en décodant simultanément plusieurs lignes de la même vue, afin d'accélérer son décodage. La figure 2.18 montre le principe du parallélisme de la restitution des vues en décodant simultanément plusieurs lignes de la même vue. Ce parallélisme a pour effet d'accélérer la restitution des vues, ce qui permet de diminuer la vitesse de restitution nécessaire.

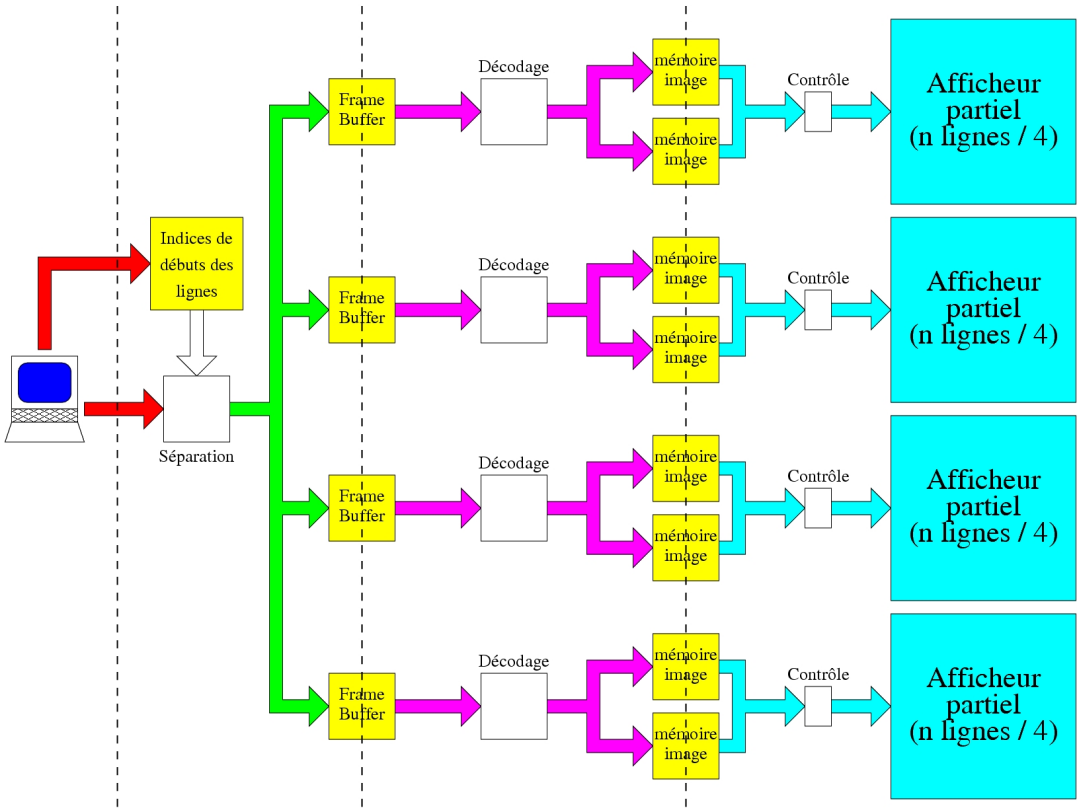


FIG. 2.18 – Parallélisation de la restitution des vues

2.6 Validation et performances des algorithmes

Les algorithmes n°1, 2, 3, 4, 5, 6, 7, 8, 9, 11 et 12 ont été validés par simulation : le codage (algorithmes n°1, 2, 4, 5, 6 et 7) a été réalisé à l'aide d'un logiciel de ray-tracing modifié que nous avons réalisé en utilisant le langage C++. Les algorithmes 11 et 12 ont été implémentés en C, sous la forme d'un logiciel qui code l'environnement 2D/3D en utilisant un fichier image et des fichiers 3D générés par le logiciel de ray-tracing modifié. Le décodage a été réalisé par deux logiciels écrits l'un en JAVA, permettant d'être exécuté au sein d'une page web, afin d'illustrer et de communiquer sur nos travaux, et l'autre en C qui permet d'obtenir des temps de décodage plus courts. Ce dernier est utilisé pour estimer les performances des algorithmes 3, 8 et 9 en mesurant leur temps d'exécution, et qui permet des appels plus simples à Xwindow pour l'affichage et l'utilisation de sémaphores pour une utilisation en parallèle avec une simulation, afin de générer les

données en même temps que de les afficher.

Les tests indiquent que la vue restituée pour la direction correspondant à celle utilisée pour le calcul de la scène est identique à l'image 2D calculée par projection de la scène selon cette même direction. Les autres vues correspondent aux différents points de vue, avec une légère déformation : un étirement horizontal, correspondant à l'inverse de la déformation résultant de l'observation du dispositif d'affichage par le coté. Cette déformation permet donc de corriger automatiquement la déformation due à la parallaxe lors de l'observation du dispositif d'affichage sous des angles présentant une différence de plus de 90°. Utilisée dans un système d'affichage autostéréoscopique réel, la vue perçue (et pas la vue affichée) sera donc identique à la vue 2D calculée par une projection des objets selon cette direction.

La taille des données générées dépend de la scène représentée. Un cas extrême consiste en une image complètement unie qui n'affiche que sa couleur de fond. Dans ce cas, les données générées se limitent à une valeur (la valeur 0, représentant le nombre de plans utilisés) pour chaque ligne.

Une image 2D parallèle à l'écran sera constituée d'un seul plan par ligne, chaque plan étant constitué d'un seul motif ayant une longueur égale à la largeur de l'écran. Les données générées auront la taille des données des pixels auxquels on ajoute le canal alpha de chaque voxel et 5 valeurs numériques par ligne : 1 pour le nombre de plans, 0 ou une profondeur arbitraire pour la profondeur du plan, 1 pour le nombre de motifs de ce plan, la largeur de l'écran pour la taille de ce motif et sa position, l'extrême gauche de l'écran. Une image 2D ayant une résolution de 1 024 colonnes et 768 lignes et 24 bits par pixel prend une place de $1\,024 \times 768 \times 3 = 2\,359\,296$ octets dans une mémoire 2D. Avec nos algorithmes, en codant toutes les valeurs (nombres et profondeurs des plans, nombres, coordonnées et tailles des motifs) sur 32 bits et en utilisant 8 bits pour le canal alpha (la taille d'un voxel est donc 32 bits : 24 bits pour la couleur + 8 bits pour le canal alpha) cette taille passe à $1\,024 \times 768 \times 4 + 768 \times 5 = 3\,149\,568$ octets, soit une augmentation d'$\frac{1}{3}$, due principalement au passage de 24 à 32 bits pour le codage des

pixels.

Le cas extrême inverse consiste en une image contenant un très grand nombre de petits (de l'ordre d'un à une dizaine de voxels) motifs. Ce genre d'image extrême peut être la représentation en 3D d'un nuage de points ou de motifs de moins de 5 voxels ou à une série de plans verticaux perpendiculaires à l'écran. Dans ce cas, la taille de la mémoire nécessaire est maximale et est déterminée par le nombre de profondeurs possibles pour les plans et peut rapidement atteindre plusieurs centaines de méga-octets. Pour limiter la taille de la mémoire utilisée, une solution envisageable consisterait à détecter les zones où le nombre de plans utilisés est le plus grand et à réduire ce nombre en fusionnant les plus éloignés. Cependant, cette réduction ne doit se faire que pour la zone détectée, afin d'éviter une perte de précision visible dans les zones où peu de plans sont utilisés, mais contenant des objets lointains. Il est également important d'éviter de détecter les plans verticaux perpendiculaires à l'écran comme des zones à réduire. La réduction se fait alors en réduisant les petits motifs les plus au centre du nuage, donc les moins visibles, en plans verticaux. La figure 2.19 illustre ce principe d'effacement de voxels trop nombreux.

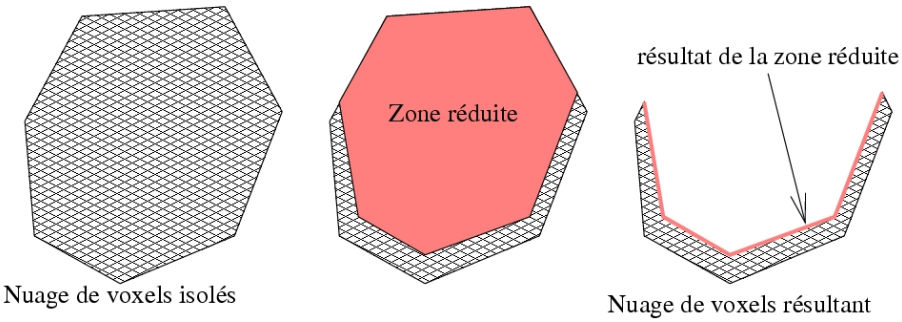
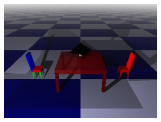
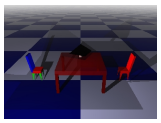
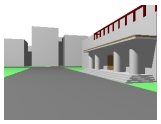
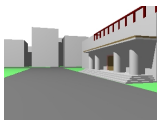
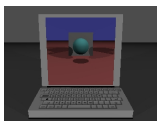
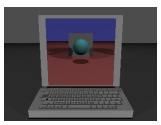


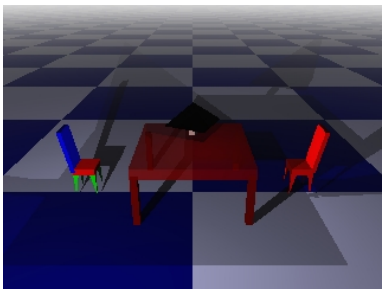
FIG. 2.19 – Principe de réduction de la taille des données en cas de dépassement

Les résultats de taille des fichiers obtenus et la vitesse de codage et décodage pour une série d'images de test sont indiqués dans le tableau 2.2. Ces fichiers ont été générés pour un système d'affichage autostéréoscopique virtuel permettant d'afficher 200 vues différentes. La taille des fichiers est donnée en octets en utilisant 32 bits pour chaque nombre et 8 bits pour chaque composante (R, G, B et A) de couleur des voxels. Les

Image	Résolution	Nombre de voxels	Taille des données compressées	Temps de codage	Temps de décodage	Taille brute de l'image
	320 × 240	389773	1791224 octets	4660 ms	6.87 ms	230400 octets
	160 × 120	97326	449500 octets	480 ms	1.52 ms	57600 octets
	320 × 240	433996	1909276 octets	4250 ms	6.66 ms	230400 octets
	160 × 120	103246	454716 octets	550 ms	1.44 ms	57600 octets
	320 × 240	300443	1339560 octets	2570 ms	5.32 ms	230400 octets
	160 × 120	74700	334396 octets	370 ms	1.07 ms	57600 octets

TAB. 2.2 – Performances des algorithmes

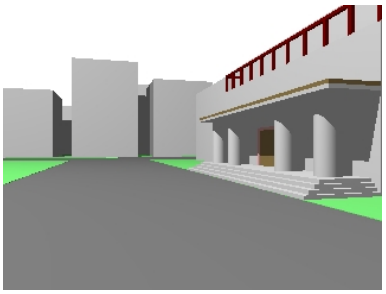
temps indiqués sont en millisecondes et ont été mesurées en utilisant l'appel système Linux *clock*. Étant donné que cette méthode ne fait que mesurer le temps entre le début et la fin de l'algorithme, sans prendre en compte le temps réellement passé par le CPU pour l'exécuter, et est donc sensible à l'exécution des autres programmes et aux accès à la mémoire virtuelle (zone swap), nous avons utilisé des images de faible résolution (160 par 120 et 320 par 240) afin d'éviter que la mémoire nécessaire n'oblige l'ordinateur de test à utiliser la zone d'échanges (zone swap) sur le disque dur, ce qui aurait pour conséquence de fausser nos tests. Les tableaux 2.3, 2.4 et 2.5 donnent les tailles des fichiers correspondant aux même scènes, mais avec un plus grand nombre de résolutions différentes, en particulier plus grandes. On remarque que les tailles des fichiers et le nombre de voxels



Résolution	160 × 120	320 × 240	640 × 480	1024 × 768	1280 × 960
Nombre de voxels	97 326	389 773	1 560 138	3 994 613	6 241 904
Taille des données	449 500	1 791 224	7 168 324	18 361 752	28 668 324
Temps de décodage	1,54	6,85	27,32	69,79	107,85

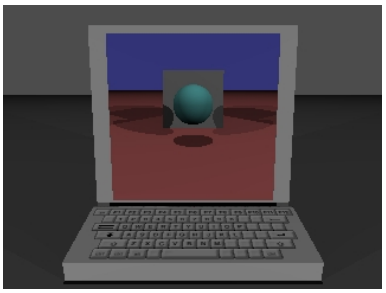
TAB. 2.3 – Performances des algorithmes pour la première scène

générés augmentent prortionnellement à la résolution des vues à générer. Les temps de décodage n'incluent pas le temps de lecture des fichiers, ni le temps d'affichage. Ils ne sont donc pas influencés par les capacités du bus ou de la carte graphique de l'ordinateur, mais dépendent uniquement de la puissance de son CPU et de sa mémoire vive et indiquent les temps de calcul nécessaires au codage et au décodage, indépendamment



Résolution	160 × 120	320 × 240	640 × 480	1024 × 768	1280 × 960
Nombre de voxels	103 246	433 996	1 802 766	4 768 491	7 597 231
Taille des données	454 716	1 909 276	7 889 876	20 778 952	33 060 296
Temps de décodage	1,44	6,66	26,73	69,74	108,95

TAB. 2.4 – Performances des algorithmes pour la seconde scène



Résolution	160 × 120	320 × 240	640 × 480	1024 × 768	1280 × 960
Nombre de voxels	74 700	300 443	1 206 288	3 093 930	4 838 067
Taille des données	334 396	1 339 560	5 380 316	13 852 612	21 628 200
Temps de décodage	1,07	5,32	21,22	55,34	86,97

TAB. 2.5 – Performances des algorithmes pour la troisième scène

du temps nécessaire pour envoyer les données au dispositif d'affichage. Ces tests ont été faits sur un ordinateur équipé d'un processeur AMD Athlon, cadencé à 1,8 GHz, avec

256 méga-octets de mémoire vive et utilisant le système d'exploitation Mandriva Linux 2006. Les images utilisées pour nos tests ont été choisies car elles représentaient des scènes contenant des objets, donc des voxels, répartis différemment : la première représente des objets placés au milieu de la scène à une distance d'environ 5 fois la largeur de l'écran. La seconde montre des objets très lointains et d'autres dont certaines facettes verticales sont presque perpendiculaires à l'écran, ce qui permet de tester le problème de discontinuité des facettes. La troisième contient des objets très proches de l'écran et utilisant peu de plans différents. La taille des données générées correspond approximativement à 10 fois la taille des données brutes d'une des vues, mais elle permet d'en générer beaucoup plus. Le temps de codage et de décodage, même s'il est trop important pour une application temps réel, c'est-à-dire au moins 30 images par seconde, avec 200 vues différentes, permet cependant d'envisager une application temps réel, après une adaptation sur une architecture spécialisée, permettant d'obtenir de meilleurs résultats.

2.7 Conclusion du chapitre

Nous avons décrit, dans ce chapitre, un ensemble d'algorithmes permettant de générer plusieurs vues d'une même scène 3D sous plusieurs angles sans la nécessité de calculer chacune de ces vues indépendamment et de mémoriser les informations permettant de reconstituer ces vues dans un espace mémoire plus petit que la place nécessaire pour mémoriser toutes les vues. Ils représentent donc bien une solution à tous les problèmes qui nous étaient posés par la réalisation de notre dispositif d'affichage 3D.

Les temps d'exécution de ces différents algorithmes permettent d'envisager l'utilisation de ces algorithmes dans des applications temps-réel permettant de générer 30 images 3D par seconde avec 200 vues différentes par images, soit un débit de 6 000 vues générées par seconde. L'utilisation d'une architecture spécialisée utilisant une parallélisation de ces algorithmes est tout à fait envisageable et permettra d'obtenir les débits nécessaires.

Chapitre 3

Faisabilité du système d'affichage 3D

Le dispositif d'affichage que nous avons proposé permettra d'afficher des images en trois dimensions à un nombre d'observateurs quelconque placés à des endroits quelconques dans une zone définie : la zone de projection (la zone où la scène 3D est perceptible en relief par un observateur). Si nous avons vérifié, par des simulations logicielles (simulation des algorithmes de codage et de décodage et simulation optique du dispositif), que le principe est viable et que les algorithmes que nous avons proposés fonctionnent, il nous reste à étudier la faisabilité du système d'affichage 3D. En d'autres termes, le dispositif est-il constructible et peut-il répondre aux exigences du procédé ?

Ces exigences sont simples mais difficiles à obtenir avec la technologie actuelle. En particulier, la contrainte temporelle présente un enjeu important. En effet, la quantité de données représentée par toutes les vues à générer en un temps très court (un trentième de seconde) représente des débits de 13 giga-octets par seconde, ce qui est à la limite des capacités des ordinateurs actuels. Nous allons donc vérifier qu'elles sont atteignables et le moyen de le faire.

Pour cela, nous avons décidé d'implémenter la génération des vues sur une architecture spécialisée. Comme les algorithmes de projection des objets, générations des voxels, mémorisation des coordonnées des voxels et de création des motifs peuvent traiter leurs données en parallèle, nous avons choisi de les implémenter en utilisant l'architecture parallèle du GPU (Graphics Processing Unit). Par contre, les algorithmes de tri et mémorisation des motifs et de mise en forme finale des données sont beaucoup plus

linéaires et plus difficiles à adapter de manière efficace sur une architecture parallèle. Par conséquent, nous avons adapté et implémenté une partie de nos algorithmes (projection des objets, générations des voxels, mémorisation des coordonnées des voxels et création des motifs) sur GPU et une autre (tri et mémorisation des motifs et mise en forme finale des données) sur une architecture spécialisée, que nous avons modélisée et simulée, en utilisant un logiciel de modélisation et simulation fonctionnelle appelé *Visual Elite* et la bibliothèque C++ *SystemC*, permettant de modéliser et de simuler des systèmes composés de plusieurs blocs de traitements échangeant des données. Nous avons également étudié la faisabilité physique du dispositif, en particulier la possibilité d'afficher électroniquement les vues à une vitesse suffisamment importante pour atteindre les 30 images par seconde avec 200 vues par image, c'est-à-dire 6 000 images par seconde, en étudiant les différentes possibilités d'affichage micro-électroniques et leurs caractéristiques techniques.

Ce chapitre présente l'étude et les tests réalisés sur l'adaptation et l'implémentation de notre algorithme de compression de données 3D sur GPU, ainsi que la modélisation d'une architecture spécialisée implémentant la partie de notre chaîne de traitements qui ne peut être implémenté sur ce GPU, ainsi qu'une étude sur la faisabilité d'un dispositif permettant d'afficher les vues suffisamment rapidement (6 000 vues par seconde) pour permettre à un tel dispositif de fonctionner.

3.1 Répartition des traitements entre CPU, GPU et architecture spécialisée

Comme nous l'avons vu précédemment, les traitements appliqués sur les coordonnées des points des objets à afficher peuvent se rassembler en trois parties :

- projection des objets sur l'écran et calcul des voxels à afficher,
- création, mémorisation et tri des motifs (algorithmes n°1 et n°4),
- mise en forme des données mémorisées (algorithmes n°2, n°5, n°6 et n°7).

Comme le GPU était, au départ, prévu pour opérer la projection des objets 3D sur l'écran pour créer l'image de ces objets, les algorithmes de projection des objets et de

génération des voxels y sont codés en dur. Les parties implémentant ces algorithmes ne peuvent donc pas être reprogrammées ni utilisées pour autre chose. Il serait donc pénalisant de ne pas utiliser le GPU pour ces traitements dans notre implémentation. C'est pour cette raison que cette partie sera gérée par le GPU. La création des motifs pourra également être implémentée en utilisant le GPU car celui-ci créera les voxels en parallèle et une majorité d'entre eux seront côte-à-côte. Il sera donc facile de créer des motifs, même s'il est probable que ceux-ci devront être modifiés par la suite car d'autres voxels (placés derrière un autre objet donc non générés en même temps que les autres) pourront s'y insérer ou être supprimés (par l'algorithme n°10 d'effacement des voxels cachés) par la suite.

Par contre, implémenter la mémorisation et le recodage sur le GPU est plus difficile pour les raisons suivantes : premièrement, l'architecture des voxel shaders qui vont écrire les données dans la texture résultat ne permet pas de déplacer les données à l'intérieur d'une texture car la position où le résultat sera écrit dans la texture résultat doit être déterminée avant l'exécution du code du vertex shader. Comme la taille des données codées dépend de la taille et du nombre de motifs et que la taille de ces motifs dépend des objets à afficher, il est impossible de prédire cette position. Deuxièmement, ces données risquent de devoir être déplacées lorsque de nouveaux plans ou motifs doivent être insérés ou que la taille d'un motif doit être modifiée en raison de l'insertion d'un ou de plusieurs voxels ou de la fusion de deux motifs. Dans ce cas, toutes les données mémorisées après la position d'insertion doivent être recopiées plus loin dans la mémoire. Comme le GPU ne peut pas lire et écrire sur la même texture en une seule passe, la totalité des données doit donc être recopiée d'une texture à l'autre à chaque insertion ou fusion de motifs. Par conséquent, ces opérations risqueraient de prendre beaucoup plus de temps à exécuter sur le GPU que sur le CPU car le nombre de passes différentes nécessaires serait alors égal au nombre de plans, motifs ou voxels dans la ligne, le plan ou le motif, respectivement, selon le cas. Il nous a donc paru plus efficace d'exécuter la mémorisation, le tri des voxels et la mise en forme finale des données sur un système dédié. Ce système a été modélisé et est décrit dans la troisième partie de ce chapitre.

Le CPU n'opére ici aucun traitement. Il ne sert qu'à envoyer les données à l'architecture spécialisée qui va les traiter. Le CPU se comporte alors comme lors de l'affichage de données 3D en utilisant une carte accélératrice 3D du commerce.

La solution que nous avons retenue (représentée sur la figure 3.1) est la suivante : les données 3D (coordonnées des points des objets, lumières, textures, etc.) sont envoyées par le CPU vers le GPU, situé sur la carte graphique. Celui-ci se charge de la projection des objets et la création des voxels. Les voxels ainsi créés sont ensuite rassemblés pour former des motifs. La création de ces motifs est également opérée par le GPU. Les motifs ainsi créés sont ensuite envoyés vers une architecture spécialisée qui se charge de les trier, de les fusionner si besoin et de les mémoriser. Une fois tous les voxels de la scène mémorisés, cette partie se charge du recodage des données et les fournit au dispositif d'affichage. On forme ainsi une nouvelle chaîne de traitements comportant un GPU qui

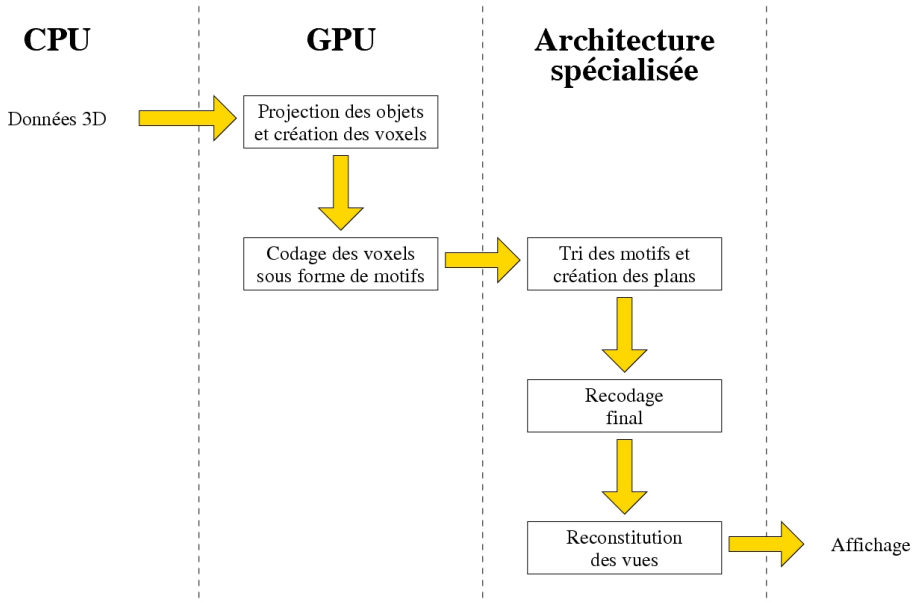


FIG. 3.1 – Adéquation GPU/Architecture spécialisée

traite les données 3D, genère les voxels et les regroupe en motifs. Les motifs sont alors envoyés dans la seconde partie de la chaîne (partie dédiée numérique) qui s'occupe du tri et de la mémorisation des motifs, ainsi que du recodage final, et la reconstitution des vues. Les vues reconstituées sont ensuite envoyées à l'affichage lui-même constitué

d'une partie électronique dédiée pour la gestion du dispositif physique d'affichage. Cette chaîne se comporte alors comme le pipe-line 3D d'une carte accélératrice 3D du point de vue de l'ordinateur hôte qui lui envoie les données 3D.

3.2 Parallèlisation et implémentation sur GPU

Une des manières d'implémenter nos traitements afin d'atteindre des vitesses de traitement permettant son utilisation temps réel est d'utiliser un GPU. Les GPUs, qui équipent les nouvelles cartes accélératrices 3D du commerce, permettent des exécutions d'algorithmes en des temps beaucoup plus courts que s'ils étaient exécutés sur un CPU. Cependant, l'architecture de ces GPUs étant différente de celle des CPUs, les algorithmes doivent être adaptés à cette architecture. En particulier, les GPUs fournissant un traitement parallèle, ils doivent être parallèlisés afin d'obtenir les meilleurs temps d'exécution. Comme notre chaîne de traitements doit calculer les voxels représentant les objets 3D à afficher et que les premiers traitements opérés sur ces voxels (projection des objets, génération des voxels et rassemblement de ces voxels en motifs) peuvent se faire en parallèle sur tous les voxels à la fois, l'exécution de ces traitements en parallèle permet d'obtenir de meilleurs résultats.

3.2.1 La programmation sur GPU

Les GPUs équipent depuis quelques années les cartes accélératrices 3D produites par les sociétés NVidia et ATI. Ce sont des processeurs conçus spécifiquement pour traiter des données 3D de la manière la plus efficace possible. Ils présentent deux types d'unités de calculs programmables placées en parallèles, les *vertex shaders* et les *pixel shaders*, permettant d'exécuter des algorithmes en parallèle. Les vertex shaders sont des unités de calculs en parallèle dont la tâche initale est de calculer la projection des objets 3D sur l'écran et de calculer tous les paramètres permettant le rendu de ces objets, comme la position des points formant les facettes sur l'écran, la position de ces points sur les textures, l'éclairement, etc. Les pixel shaders sont conçus pour calculer la couleur de chaque voxel à afficher : en fonction de la coordonnée de la texture lui correspondant,

de l'éclairement, des ombres, etc. Ces calculs sont de type SIMD (Single Instruction, Multiple Data) ce qui signifie que toutes les unités de calculs d'un des deux types (48 pixels shaders pour la carte ATI Radeon X1900) exécutent les mêmes instructions de calculs sur des données différentes : des vertices (ou des points dans l'espace) pour les vertex shaders et des voxels pour les pixel shaders.

Normalement, ces ressources sont utilisées pour calculer des images 3D en temps réel, comme le font les cartes accélératrices 3D non équipées d'un GPU. Elles peuvent également être utilisées pour des calculs de géométrie dans l'espace et d'affichage de surfaces complexes [87] ou pour des calculs de rendu plus complexes, comme l'éclairage ambiant [27]. Mais il est également possible de les utiliser pour des applications de calculs n'ayant rien à voir avec l'affichage 3D. Par exemple, des simulations de comportement de gaz parfaits en résolvant les équation d'Euler [92] ou de liquides, en utilisant la méthode des masses ponctuelles [61]. Il est également possible de leur faire exécuter des algorithmes de traitement d'images [56] ou de sons [99] et même de comparer des séquences ADN [104].

Poussés par un public de plus en plus exigeant sur la beauté des graphismes dans les jeux vidéo, les constructeurs de GPU ont introduit la possibilité de programmer certaines étapes du traitement 3D pour obtenir des effets visuels plus élaborés. Certaines étapes possèdent des options pour activer ou désactiver certaines fonctions, comme l'algorithme du Z-buffer, utilisé pour ignorer les voxels qui seront cachés par d'autres dans l'image finale. Ces fonctions, cablées directement dans le GPU, sont très rapides : elles sont aussi rapides que la simple écriture dans la mémoire. Par exemple, accéder à une texture en prenant le plus proche voisin est aussi long que d'utiliser une interpolation bilinéaire. On peut donc, en abusant de cette fonction, faire des interpolations linéaires extrêmement rapidement.

La raison pour laquelle de plus en plus de personnes s'intéressent aux GPUs est leur puissance de calcul : les GPU actuels ont plus de 200 GFlops (milliard d'opérations en virgule flottante par seconde) contre une vingtaine pour les CPU grand marché actuels et ce pour le même prix. Cette puissance de calcul permet d'accélérer certains calculs

sur nombres réels, comme les traitements de signaux ou les simulations.

De plus, ces GPUs disposent sur la carte d'une quantité de mémoire non négligeable (entre 256 et 512 méga-octets, voir figure 3.2) avec un débit entre le GPU et la mémoire de la carte (voir figure 3.3) de l'ordre de 25 giga-octets par seconde, ce qui permet des traitements de grandes quantités d'information en un temps très court. Par contre, cette mémoire a un temps de latence élevé, ce qui pousse les constructeurs de GPU à utiliser un cache pour accélérer l'accès et faciliter la tâche du prefetch (pré-chargeur de données).

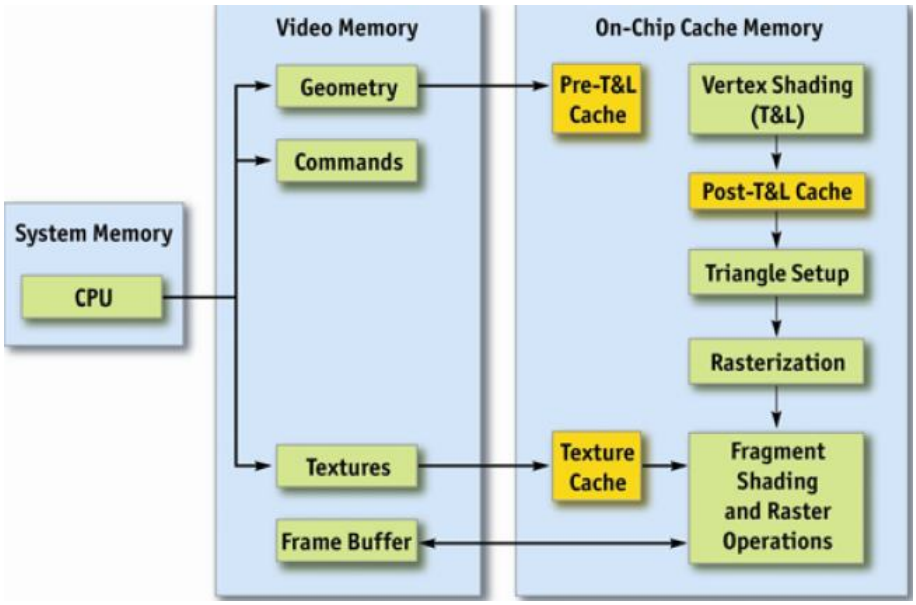


FIG. 3.2 – Mémoires d'un GPU (*Source : NVidia GPU Programming Guide [11]*)

Architecture d'un GPU

Le GPU remplace tout le pipe-line 3D des cartes accélératrices 3D. Il comporte également un certain nombre de blocs de traitements formant un pipe-line, traitant les coordonnées des objets à afficher, ainsi que les informations de texture et d'éclairement pour générer les images résultant de leur projection. La figure 3.4 représente ce pipe-line. La majorité de ces blocs sont identiques à ceux des cartes accélératrices 3D ne possédant pas de GPU et implémentent les mêmes algorithmes (projection des objets, rasterisation,

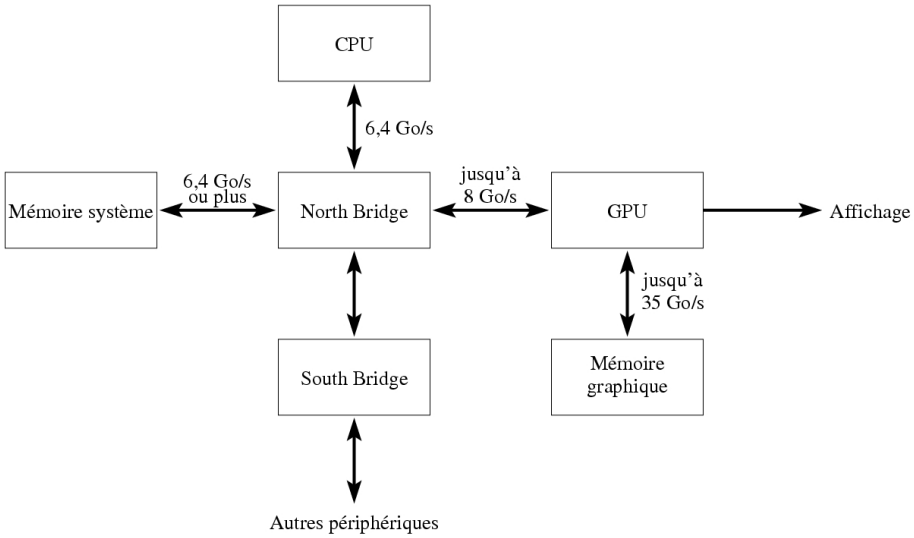


FIG. 3.3 – Débits de données typiques maximaux entre le GPU, le CPU et leurs mémoires (*Source : GPU Gems 2 [43]*)

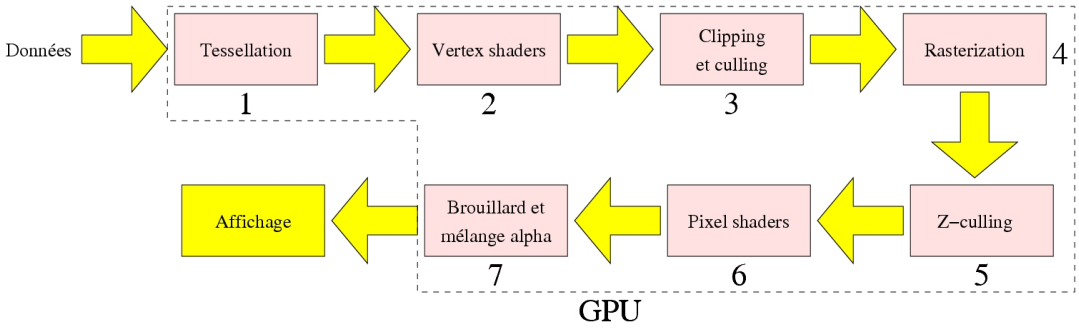


FIG. 3.4 – Schéma de principe d'un GPU

etc.). La différence est que les GPUs comportent deux étages particuliers : l'étage des *vertex shaders* et celui des *pixel shaders*. Ces deux étages sont les étages programmables permettant d'adapter les algorithmes implémentés par la carte et d'utiliser la puissance du GPU pour les exécuter en leur envoyant les données à traiter soit sous la forme d'objets à traiter (pour les vertex shaders), soit sous la forme d'une texture (pour les voxel shaders).

Ce pipe-line se compose donc d'abord d'un étage de tessellation, permettant d'augmenter le nombre de vertices, c'est-à-dire de points et de facettes définissant les objets, par interpolation des coordonnées des vertices passés en paramètre et de leur vecteur

normal. Ceci permet d'augmenter la qualité du rendu, en particulier pour des surfaces courbes. Il existe plusieurs algorithmes de tessellation, qui vont définir où et quand il faut augmenter le nombre de vertices. Une méthode adéquate consiste à faire varier le nombre de vertices afin que les objets proches en possèdent beaucoup et que les objets distants en possèdent peu. Ces traitements ont été initialement présentés par ATI et sont également appelés N-Patches (Normal Patches). NVidia a préféré l'utilisation de primitives appelées les RT-Patches (Real-Time Patches). Cependant, bien que DirectX supporte toujours ces deux types de primitives, ils ont tous les deux plus ou moins abandonné le support de la tessellation, même sur leurs cartes les plus récentes.

Le second étage s'occupe normalement du calcul de projection des points des objets. Mais comme il est programmable, il est possible de lui faire faire des calculs différents plus ou moins complexes. En réalité, il est constitué de plusieurs unités de traitement indépendantes (8 vertex shaders pour la carte ATI Radeon X1900) pouvant ainsi traiter plusieurs vertices en parallèle, ce qui permet une grande vitesse de calculs et une grande flexibilité de traitements. Chaque unité contient un certain nombre de registres (environ 300, mais ce nombre peut varier d'un GPU à l'autre) de différents types : certains contenant les données à traiter, d'autres les données résultats, d'autres les paramètres constants d'un vertex à l'autre qui servent à contenir les paramètres du traitement à opérer et les derniers servent à mémoriser des valeurs temporaires en cours de traitement. Le nombre de ces registres dépendent des GPUs et de la version de vertex shaders qui est implémentée. Les vertex shaders permettent de calculer des transformations et déplacements des points des objets et des normales aux surfaces. En d'autres termes, ils servent à calculer les coordonnées du sommet dans le référentiel de la caméra, ainsi que la couleur dûe à l'illumination de la scène. Ils permettent ainsi de créer des surfaces complexes ou de modifier les surfaces ou leur éclairement.

Le troisième étage du pipe-line constitue le clipping et le culling, c'est-à-dire qu'il supprime ou tronque les surfaces qui sont hors du champ de vision de l'observateur (clipping) ou qui représentent les faces non visibles des objets (culling).

Le quatrième étage transforme les coordonnées des vertices calculées dans les étages

précédents en pixels, en remplissant les projections des facettes sur l'écran en voxels (pixels en 3D). Les coordonnées des pixels, avec les informations d'éclairement des facettes et de coordonnées de textures seront utilisés dans les étages suivants pour calculer la couleur des pixels.

Ces coordonnées sont ensuite envoyés à l'étage permettant de ne prendre que les pixels visibles et d'ignorer ceux qui sont cachés par d'autres, en utilisant l'algorithme du Z-buffer [28]. Ainsi, si un pixel doit être placé à une position où un autre pixel opaque plus proche se trouve déjà, le pixel le plus loin n'a pas besoin d'être calculé car, il n'apparaîtra de toute façon pas sur l'image finale. En réalité, l'algorithme du Z-buffer est utilisé à deux endroits différents dans le pipe-line du GPU. Le premier, situé dans le 5ème étage du pipe-line, est souvent appelé "early-z-culling" ou, plus simplement, "z-cull" ou "early-z" et est utilisé pour retirer un maximum de voxels inutiles avant de les envoyer à l'étage suivant : celui des pixels shaders. Le second se situe dans le septième et dernier étage du GPU, avec les calculs de brouillard et le mélange alpha. Il sert à retirer les derniers voxels qui n'ont pas été retirés par le "early-z" à cause, par exemple, du fait qu'ils sont cachés par des objets dont la transparence n'est pas connue à ce niveau du pipe-line et sera calculée par les voxel shaders. Dans le cas de pixels partiellement transparent, les pixels sont conservés dans une mémoire tampon spéciale, permettant de rendre les pixels du plus lointain au plus proche, afin de rendre la transparence correctement. Cet étage peut être désactivé par le programmeur si les pixels les plus lointains doivent être traités par les pixel shaders (l'étage suivant du pipe-line) même s'ils sont cachés par d'autres.

L'étage suivant constitue la deuxième partie programmable du GPU. Il s'agit des pixel shaders. Ce sont, comme les vertex shaders, plusieurs unités de traitement programmables (48 pixels shaders pour la carte ATI Radeon X1900) pouvant fonctionner en parallèle. Ces blocs traitent les données des pixels calculées par les étages précédents et en calculent la couleur. Contrairement aux vertex shaders dont le résultat était envoyé directement dans l'étage suivant du pipe-line, les pixel shaders écrivent dans une texture. Cette texture peut ensuite être récupérée par le CPU pour un traitement éventuel ou être affichée soit sous la forme d'une image finale, soit sous la forme d'une texture à

plaquer sur un objet.

Le dernier étage s'occupe des derniers traitements comme le brouillard ou la gestion des pixels partiellement transparents. Ces traitements peuvent être paramétrés ou désactivés en fonction des besoins.

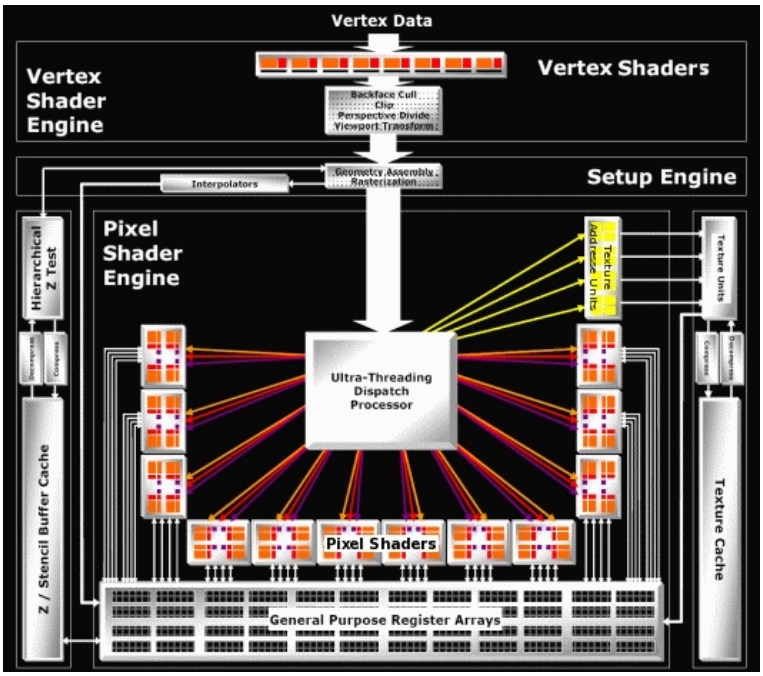


FIG. 3.5 – Schéma fonctionnel du GPU de la carte graphique ATI Radeon X1900 (*Source : ATI*)

La figure 3.5 montre le schéma fonctionnel complet d'un de ces GPUs : la partie haute, notée *"Vertex Shader Engine"*, représente l'étage des vertex shaders et celui du clipping et culling, le bloc du milieu, noté *"Setup Engine"* représente l'étage de rasterisation, c'est-à-dire de génération des voxels, le bloc du bas, noté *"Pixel Shader Engine"* représente les pixel shaders, le bloc en bas à gauche correspond au dernier étage du pipeline, s'occupant du brouillard et mélange alpha, ainsi que le cache d'accès à la mémoire nécessaire à cet étage, et le bloc en bas à droite représente les caches et contrôleurs d'accès aux mémoires de textures, utilisés par les pixel shaders.

De part son architecture, le GPU est très rapide pour faire des traitements pouvant

être traités en parallèle, ce qui lui donne une vitesse de traitement beaucoup plus importante que pour le CPU. Par contre, pour des traitements plus linéaires, même très simples, comme des comptages, le GPU est obligé d'opérer un certain nombre d'itérations pour pouvoir additionner le résultat du calcul précédent, ce qui diminue son efficacité car cela implique que seul un petit nombre de shaders peut travailler et que la plupart ne fait rien, ce qui ne permet pas d'utiliser le potentiel du GPU. Il est donc nécessaire d'adapter certains calculs pour les parallèliser, afin de rendre leur implémentation sur GPU avantageuse.

Spécificités et contraintes de la programmation des GPUs

Le GPU a été conçu uniquement dans le but de traiter des scènes 3D, il possède donc des avantages et des inconvénients par rapport à un processeur généraliste classique :

Les seules structures de données connues de manière native par le GPU sont les différentes formes de texture :

- texture 1D, 2D ou 3D, avec une taille maximale de 2048 ou 4096 pixels de côté. Il faut faire attention à la taille des données allouées : un cube de 512 pixels d'arête utilise l'intégralité des 512 méga-octets de mémoire disponible,
- texture sur plusieurs niveaux de précision, qui offre la possibilité de faire du filtrage trilinéaire "gratuitement" (c'est-à-dire sans surcoût temporel),
- texture cubique, composée de 6 textures 2D plaquées sur les faces d'un cube. Il est possible d'émuler d'autres structures de données, comme des matrices creuses [57] mais une structure dynamique (avec des pointeurs) comme, par exemple, une liste chaînée, n'est pas efficace car, lorsque le GPU demande l'accès à un pixel, le prefetch charge les pixels voisins, opération utile pour tous les calculs de type convolution, mais qui réduit grandement les temps d'accès mémoire dès que l'adresse du prochain élément en mémoire n'est pas prédictible.

La plus grande limitation des GPU est le principe appelé *"Scatter - Gather"* : les vertex shaders prennent en entrée un sommet et ont en sortie un sommet. Aucun de ces sommets ne peut avoir accès aux informations de ses voisins. Par contre, on peut changer

les coordonnées de ce sommet, ou, vu autrement, on peut changer l'adresse mémoire de destination. C'est la partie Scatter (dispersion). Les pixel shaders ont également une entrée et une sortie, mais l'emplacement mémoire de la sortie est fixé. En contrepartie, ils ont la possibilité d'accéder aux informations contenues dans les textures, c'est la partie Gather (rassemblement). Cette architecture impose d'énormes contraintes sur les algorithmes utilisables, à cause de l'impossibilité de lire une donnée et de modifier l'adresse de destination selon la valeur lue.

Le GPU ne peut traiter que des nombres réels à virgule flottante et la précision de ces calculs varie selon le modèle de GPU utilisé. Les modèles actuels réalisent leurs calculs sur 16, 24 ou 32 bits. Il est possible d'émuler des nombre réels double précision sur 64 bits au prix de calculs alourdis. Le fait de ne pas avoir d'entiers empêche de réaliser certains types de calculs, en particulier les opérations bit à bit. De plus, les accès aux éléments d'un tableau peuvent être faussés à cause des arrondis sur les valeurs à virgule flottante.

L'architecture de type SIMD des unités de calcul dans le Pixel shader permet de traiter efficacement en parallèle de grandes quantités de voxels : un seul bloc de contrôle est nécessaire pour le séquencement de toutes les unités. Cela augmente la vitesse et réduit la consommation. De plus, pour augmenter les performances de la prochaine génération de GPU, il suffit d'augmenter le nombre d'unités de calcul.

Le défaut majeur de l'architecture SIMD est que les branchements conditionnels dans le code coûtent cher en temps d'exécution. En effet, il suffit que le test conditionnant un traitement supplémentaire soit vrai pour une seule des unités de calcul pour qu'elles exécutent toutes le code associé, alors que seule celle pour laquelle le code est vrai prendra en compte le résultat de ce traitement. Par exemple, le pseudo-code suivant :

```
Fonction f ( entrée a, entrée b, entrée c) {
  Si ( a == 1 )
    retourne b + c
  Sinon
    retourne b - c
}
```

Le processeur SIMD est obligé de traduire le code en :

```
Fonction f ( entrée a, entrée b, entrée c) {
  Tmp1 = b + c
  Tmp2 = b - c
  retourne  ( (a == 1) * tmp1 + (a != 1) * tmp2 )
}
```

Dans cet exemple, tous les calculs conditionnels sont exécutés et le résultat retourné correspondant à la condition à la fin du traitement, une fois que tous les calculs ont été réalisés. La plupart des GPU du marché réduisent l'impact de cette contrainte en ajoutant l'optimisation suivante : si la condition est une constante, alors le compilateur JIT ("Just In Time", c'est-à-dire le compilateur qui compile le code à la volée juste avant l'exécution) peut couper la branche qui ne sera jamais prise. Si toutes les unités SIMD prennent le même branchement conditionnel, alors seul ce branchement est pris. Comme les données sont traitées par blocs de l'image de sortie, si la condition repose sur la position du point dans l'espace, alors il y a de fortes chances que tous les voxels prennent la même branche conditionnelle.

Les GPUs ne possèdent pas de fonction pour générer de nombre aléatoire. Il existe des moyens pour contourner ce problème. Principalement en générant les nombres sur le CPU puis en les transférant vers le GPU, cela permet de donner des fonctions de bruit pour générer des nuages ou des textures rugueuses aléatoires [78].

Les possibilités de communications entre le CPU et le GPU sont très limitées et coûteuses en temps :

- dans le sens CPU vers GPU, on peut envoyer des informations relatives aux sommets, des textures, des constantes, plus les instructions pour contrôler l'exécution du code sur le GPU.
- dans l'autre sens, les seules informations que l'on peut avoir se font en lisant les textures calculées par le GPU, ainsi que certaines valeurs très spécifiques comme le nombre de points rendus.

Le manque de possibilités de communication entre le CPU et le GPU fait qu'au niveau global, la plupart des fonctions sont exécutées en "aveugle" : on doit considérer une

fonction sur le GPU comme une boîte noire où l'on n'a accès qu'aux données en entrée et en sortie. La conséquence est que les algorithmes du type :

```
faire
  fonction
tant que condition (sortie)
```

exécutés tel quels prennent souvent plus de temps à s'exécuter que de les changer en :

```
faire
  fonction
tant que nombre d'itérations < pire cas
```

si l'on peut connaître le pire cas à l'avance.

Comme les textures sont censées contenir des informations sur les couleurs des points, nous sommes limités à un maximum de quatre canaux par pixel et une profondeur comprise entre 16 et 32 bits en virgule flottante par canal. Les GPUs sont également limités dans le nombre de textures qu'une fonction peut avoir en sortie, la valeur courante étant 4. On a donc un maximum de 16 valeurs en sortie par pixel. Les informations qui nous intéressent pour nos traitements, sont la couleur de chaque voxel (4 valeurs), ses coordonnées (3 valeurs) et la position et la taille de chaque motif (1 valeur) soit un total de 8 valeurs pour chaque voxel. La quantité de valeurs disponibles en sortie est donc suffisante pour nos traitements.

Programmation d'un GPU

Il existe deux méthodes pour programmer un GPU : La première permet d'y accéder de manière totalement abstraite depuis un langage de haut niveau en utilisant une bibliothèque spéciale. Il en existe plusieurs à un stade plus ou moins expérimental comme Brook (pour les langages C et C++) [52], Sh (pour le langage C++) [13] ou Accelerator (pour le langage C#) [33]. Toutes ces bibliothèques sont basées sur le même principe. On crée un tableau à une dimension, on y met toutes les données en entrée, puis on écrit une fonction qui sera appliquée indépendamment à tous les éléments du tableau. La bibliothèque se charge ensuite de traduire le tableau en texture et la fonction en code GPU en passant par une API graphique (OpenGL ou DirectX) pour envoyer les

données au GPU et exécuter l'algorithme. Cette approche permet d'obtenir des résultats rapidement sans avoir à se soucier des considérations graphiques (création d'une fenêtre, allocation des textures, canaux RGBA...) mais possède deux défauts majeurs :

- on n'utilise qu'une partie de la puissance du GPU : le pixel shader. Le vertex shader et les fonctions spéciales précablées sont inutilisées.
- le fait de programmer aussi loin du matériel incite à ignorer les spécificités de son architecture et donc à utiliser des algorithmes inadaptés.

La seconde solution consiste à utiliser directement l'API graphique. Là encore, il existe une multitude de possibilités. Pour l'API, on peut choisir d'utiliser DirectX ou OpenGL, avec chacun des avantages et des inconvénients. DirectX est une API multimédia orientée jeux vidéo. La partie 3D de cet API, Direct3D, a l'avantage d'être un standard dans ce domaine. Un code correctement écrit est censé pouvoir fonctionner sur toutes les cartes graphiques récentes du marché. De plus, les drivers sont plus optimisés pour cette API. Son défaut est de ne fonctionner que sur les ordinateurs utilisant le système d'exploitation Windows.

OpenGL est un standard ouvert écrit par un consortium et dont il existe des implémentations pour la plupart des OS disponibles. Le problème est que, comme le consortium est lent à prendre des décisions, la version 1.5 du standard est obsolète et le support de la version 2.0 est plus ou moins correct selon le constructeur de la carte. Ces mêmes constructeurs, pour permettre aux développeurs d'exploiter pleinement leurs cartes graphiques, ajoutent des extensions au standard OpenGL. On se retrouve donc avec un grand nombre d'extensions plus ou moins compatibles entre elles et avec un autre matériel d'un autre fabricant.

Une fois L'API choisie, il existe plusieurs langages pour programmer le GPU, en plus du code assembleur. On a le choix entre HLSL (développé par Microsoft) [8], GLSL (développé par le consortium OpenGL), Cg (développé par NVidia) [10] et ASHLI (Advanced Shading Language Interface, développé par ATI) [2]. Tous ces langages sont très proches les uns des autres et offrent quasiment les mêmes possibilités.

La plupart des programmes utilisant le GPU pour réaliser des calculs généraux utilisent la méthode suivante :

1. On crée une interface graphique pour le programme.
2. On initialise la carte graphique (on la prépare à recevoir les informations et le programme) et on vérifie si elle est compatible, c'est-à-dire qu'on vérifie si elle contient bien un GPU et que ce GPU est compatible avec la version des vertex et pixel shaders dont nous avons besoin pour l'algorithme que nous voulons y exécuter.
3. On initialise les structures de données, en particulier les textures pour stocker les données, les textures résultat, les textures pour les calculs intermédiaires, les matrices de transformation 3D et les constantes pour le GPU.
4. On met à jour la (ou les) texture(s) entrée(s) avec les données en entrée.
5. On règle le GPU pour que le résultat ne soit pas affiché à l'écran mais redirigé vers une ou plusieurs textures résultat.
6. On demande au GPU de dessiner un carré de la taille de la fenêtre et texturé avec notre texture en entrée.

Dans le GPU :

1. Le vertex shader place les coins du carré de manière à couvrir toute la fenêtre.
2. Le rasterizer remplit la fenêtre de pixels. On a alors une relation 1 pour 1 entre les pixels de la texture en entrée et ceux de la texture en sortie.
3. Le pixel shader, qui connaît les coordonnées x et y du pixel en cours, accède à la donnée correspondante dans la texture entrée. Il peut alors calculer l'algorithme désiré sur cette donnée et écrire le résultat dans la texture sortie.
4. Dans le cas où un passage n'est pas suffisant, on répète cette opération en écrivant les résultats dans une texture intermédiaire et en réutilisant cette texture comme entrée dans une seconde passe. Plusieurs passes sont en effet souvent nécessaires car la quantité de code dans une passe est limitée et parcequ'il est impossible pour un vertex ou un pixel shader de lire et d'écrire dans la même texture lors d'une

même passe. Cette technique qui consiste à utiliser deux textures alternativement en tant qu'entrée puis en tant que sortie est couramment appelée "ping-pong".

5. Une fois le résultat calculé, il suffit de reconvertir la texture résultat en valeurs et à exploiter ces valeurs.

3.2.2 Adaptation de l'algorithme de compression des données 3D

Projection des objets

Pour pouvoir implémenter notre chaîne de traitements sur GPU, il a fallu y faire quelques modifications afin de profiter pleinement du parallèlisme du GPU, pour obtenir des temps d'exécution les plus courts possibles. Dans un premier temps, la projection des objets a dû être adaptée. En effet, notre algorithme doit recevoir les voxels issus de toutes les facettes des objets, et pas uniquement celles visibles depuis le point de vue virtuel utilisé. Ceci permettra de pouvoir regarder les objets selon plusieurs angles.

La première tâche à réaliser est de récupérer l'ensemble des voxels de la scène 3D. Or, à cause du fait que les pixel shaders ne peuvent écrire qu'à un emplacement mémoire dont la position est fixée avant l'exécution et qu'ils doivent tous exécuter le même programme en parallèle, on ne peut récupérer simplement ces voxels. Le seul endroit où on peut noter l'existence d'un voxel, du moins en une seule passe, est sa position dans l'espace dans le référentiel de la caméra. On a alors un autre problème : si on réalise le rendu de la scène 3D dans une première passe, il faut stocker le résultat dans une texture pour pouvoir l'exploiter lors de la seconde passe. Si la texture est bidimensionnelle, on perd toutes les informations relatives aux voxels cachés par les objets au premier plan. Si on utilise une texture 3D, la taille des données explose. La seule solution est donc de faire un rendu en plusieurs fois avec des résultats intermédiaires peu coûteux en mémoire, en calculant, à chaque passe, un certain nombre de voxels puis, lors des passes suivantes, de calculer les voxels situés derrière les voxels calculés au cours des passes précédentes et qui étaient cachés, puis de coder chacun de ces résultats intermédiaires avant de passer

au suivant.

La première idée étudiée a été de désactiver l'algorithme du Z-buffer pour pouvoir récupérer les voxels en arrière-plan en même temps que ceux en premier plan. Le problème qui se posait était que la position du résultat était déterminé par les coordonnées x et y de la projection. Par conséquent, si on projetait plusieurs voxels au même point, le dernier calculé effacerait les autres. Et comme l'algorithme du Z-buffer est désactivé, ce voxel ne serait pas forcément le plus proche.

Ensuite l'idée de rendre chaque facette de tous les objets indépendamment et de récupérer les voxels la composant avant de passer à la suivante a été envisagée. Cette solution permettait de récupérer la totalité des voxels de la scène, y compris les voxels cachés. Par contre, le nombre de facettes d'une scène 3D peut dépasser le million et le rapatriement des voxels nécessite de rapatrier vers le CPU une texture de la taille de l'écran, c'est-à-dire 1024 par 768, avec 24 bits par pixel. Par conséquent, la taille de données à récupérer est très importante : dans le meilleur des cas, c'est-à-dire une texture ayant trois valeurs codées sur 16 bits par pixel, une texture de 1 024 par 768 occupe une place mémoire de $1\,024 \times 768 \times 6 = 4\,718\,592$ octets, soit plus de quatre méga-octets. Par conséquent, si la scène à rendre contient plus de 230 facettes, ce qui est un nombre très réduit pour une scène 3D (dont la plupart dépasse aisément le million de facettes), la taille des données retournées par le GPU dépasse le giga-octet. Une scène contenant un million de facettes représenterait alors quatre millions de méga-octets, c'est-à-dire quatre tera-octets par image. Pour un dispositif permettant d'afficher 30 images par seconde, il faudrait alors traiter 120 tera-octets de données par seconde. Une telle quantité de données est impossible à traiter en si peu de temps avec un seul ordinateur. La solution aurait alors été de détecter la taille de la texture en fonction de la taille réelle de la facette. Le problème est que le GPU n'est pas capable de générer lui-même une nouvelle texture. Celle-ci doit être créée par le CPU puis envoyée vers le GPU pour qu'il la remplisse. Celà nécessite donc un échange de données entre le CPU et le GPU : le GPU

fait le calcul de la facette, puis calcule la taille de la texture nécessaire pour mémoriser les voxels et envoie cette valeur au CPU (sous la forme d'une texture de taille connue contenant 4 valeurs) afin que celui-ci puisse allouer une texure de cette taille. Le CPU envoie ensuite la texture au GPU qui la remplit des données voulues et la retourne au CPU. Même si cette solution permet de réduire la taille des données envoyées, elle n'est cependant pas très efficace à cause du grand nombre de facettes composant la scène 3D.

Ensuite, notre étude a porté sur l'idée de découper la scènes en "tranches" parallèles à l'écran, chacune contenant les objets ou parties d'objets devant être représenté dans un plan, comme schématisé sur la figure 3.6. Il fallait donc projeter uniquement les objets ou parties d'objets situés entre deux distances données et les coder, puis répéter l'opération pour toutes les distances possibles. L'inconvénient de cette méthode est que le nombre de

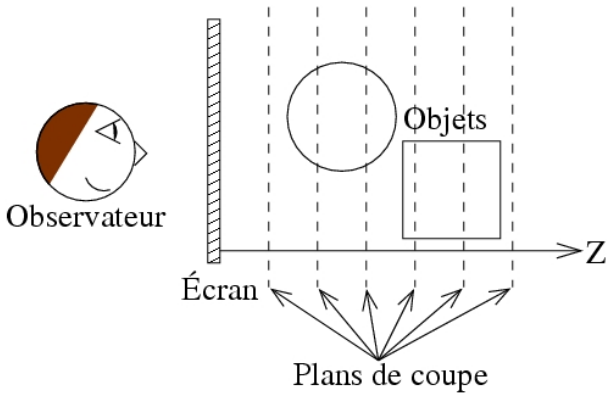


FIG. 3.6 – Principe de récupération des voxels en découpant la scène en plans parallèles à l'écran

plans possibles et donc de tranches à calculer, est très important (supérieur à la résolution horizontale des vues, c'est-à-dire 1 024) et beaucoup d'entre eux ne contiennent aucun voxel, ce qui ralentit inutilement l'exécution de l'algorithme.

Plusieurs autres solutions ont été envisagées pour pouvoir calculer et récupérer les voxels de la scène 3D, comme inverser les coordonnées y et z pour rendre uniquement les voxels situés dans un plan vertical perpendiculaire ou parallèle à l'écran. En rendant

ainsi tous les plans possibles, on arrive à retrouver tous les voxels. Cependant, le nombre de plans possibles est supérieur à la résolution horizontale de l'écran. De plus, la reconstitution des motifs dans ce cas est très difficile et nécessite un grand nombre de copies d'une texture à une autre pour pouvoir reconstituer les motifs dans une texture car les voxels d'un même motif seront répartis chacun dans un plan différent. Il aurait donc fallu calculer plus d'images que si on avait coupé la scène en "tranches" parallèles à l'écran, ce qui conduirait à un temps d'éxécution plus long. Par conséquent, cette méthode n'a pas été retenue.

Nous avons finalement opté pour l'utilisation de l'algorithme du "depth peeling" [39] [42] [75]. Cet algorithme consiste à faire dessiner la scène 3D par le GPU en conservant l'utilisation de l'algorithme du Z-buffer, de récupérer l'ensemble des pixels visibles (en rouge sur la figure 3.7) et leur profondeur (pour en faire des voxels) puis de relancer le rendu pour chaque pixel de l'image à partir de la profondeur au plan situé juste derrière le voxel, le plus près possible. On obtient ainsi certains pixels cachés (en vert

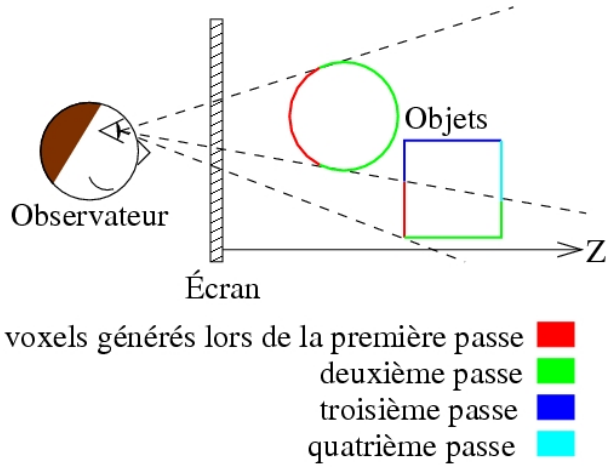


FIG. 3.7 – Principe du *"depth peeling"*

sur la figure 3.7) des objets. En répétant cette technique jusqu'à ce qu'il n'y ait plus de voxel visible (ou que leur nombre soit inférieur à un seuil prédéfini arbitrairement par le programmeur, par exemple 10% du nombre de voxels générés à la première passe), on

récupère tous les voxels (en bleu et bleu clair sur la figure 3.7) constituant la scène 3D à rendre. Cet algorithme est détaillé sous la référence algorithme n°13 : **Depth Peeling**.

Algorithme n° 13 Depth Peeling

Données : G : la scène à rendre

```
1:  i = 0
2:  Répéter
3:      F ← rasterize(G){Calcule les voxels}
4:      Si i == 0 , alors
5:          {1ère passe : on prend les voxels visibles et on mémorise leur profondeur}
6:          Pour chaque fragment ∈ F , faire
7:              booléen test ← performDepthTest(fragment)
8:              Si test , alors
9:                  fragment.depth → Z-buffer
10:                 fragment.color → color buffer
11:             Sinon
12:                 rejeter fragment
13:             Fin Si
14:         Fin Pour
15:     Sinon
16:         Pour chaque fragment ∈ F , faire
17:             {2ème passe : on prend le voxel le plus proche placé derrière les voxels déjà mémorisée}
18:             Si fragment.depth > depthLayerMap(i−1) , alors
19:                 {prendre uniquement les pixels plus loin que les pixels déjà affichés}
20:                 booléen test ← performDepthTest(fragment)
21:                 {prendre le pixel le plus proche parmi ceux qui sont à une distance supérieure à celle du
                    voxel généré par la dernière passe pour la même position}
22:                 Si test , alors
23:                     fragment.depth → Z-buffer
24:                     fragment.color → color buffer
25:                 Sinon
26:                     rejeter fragment
27:                 Fin Si
28:             Fin Si
29:             depthLayerMap(i) ← Z-buffer
30:             colorLayerMap(i) ← color buffer
31:             i++
32:         Fin Pour
33:     Fin Si
34: jusqu'à occlusionQuery() == 0
```

Dans cet algorithme, nous utilisons les notations suivantes :

- la fonction *rasterize* est l'exécution de l'algorithme de rasterisation, programmé "en dur" dans le GPU, qui produit l'ensemble des voxels (visibles ou cachés) de la scène.
- *fragment* représente un voxel, avec ses coordonnées (celles où il doit être dessiné à l'écran de l'image 2D finale si le GPU calculait une image 2D), sa profondeur

(dont l'accès est représenté par *fragment.depth*) et sa couleur (à laquelle on accède par l'intermédiaire de *fragment.color*).

- *Z-buffer* représente une texture servant à conserver la profondeur du dernier voxel traité pour chaque position sur l'écran. L'écriture et la lecture dabs cette texture se fait à la position du voxel *fragment* qu'on est en train de traiter.
- *color buffer* représente une texture servant à conserver la couleur du dernier voxel traité pour chaque position sur l'écran. L'écriture et la lecture dans cette texture se fait à la position du voxel *fragment* qu'on est en train de traiter.
- la fonction *performeDepthTest(fragment)* vérifie que le voxel *fragment* est le plus proche (testé avec la valeur correspondant dans la texture *Z-buffer*).
- depthLayerMap$_{(i)}$ représente une texture, similaire à la texture *Z-buffer*, contenant la profondeur des voxels générés par la passe numéro i.
- la fonction *occlusionQuery()* retourne le nombre de voxels qui ont été générés par la dernière passe du programme.

Cet algorithme se divise en deux passes : dans la première, on projette tous les voxels et on ne garde que ceux visibles, comme on le ferait pour un calcul d'image normal, et on mémoise la profondeur de tous les voxels mémorisés dans un G-buffer [84], c'est-à-dire une texture dans laquelle on écrit les coordonnées des voxels à la place des valeurs des voxels. Comme les textures contiennent jusqu'à quatre valeurs par pixel et que ces valeurs sont des réels en virgule flottante sur 16 ou 32 bits, il est tout à fait possible d'utiliser ces valeurs pour coder les entiers représentant les positions et tailles des motifs, le nombre de motifs dans les plans et le nombre de plans dans les lignes, sans générer d'effets d'arrondis des valeurs. Dans la seconde, on projette tous les voxels, mais on ignore tous ceux qui sont derrière un voxel déjà mémorisé et on ne prend que celui-là. Puis on répète la phase deux jusqu'à ce qu'il n'y ait plus de nouveaux voxels créés. Dans l'algorithme n°13, le nombre de ces voxels est retourné par la fontion *occlusionQuery()*. Cette fonction correspond à une fonction spéciale du GPU qui retourne le nombre de voxels que ce traitement a écrit dans la texture résultat.

Cet algorithme permet un traitement en parallèle de tous les voxels. Il permet de

récupérer les voxels composant toutes les facettes des objets, pas seulement celles visibles. Le nombre d'itérations nécessaires à cet algorithme est, au pire, égale au nombre de plans possibles. À chaque itération, on mémorise les coordonnées de chaque voxel pris en compte.

Construction des motifs

Une fois que ces voxels ont été créés, ils sont écrits dans une texture résultat à la position où ils doivent être dessinés dans la vue utilisée comme référence (la vue centrale). Il est possible, à ce moment de retourner la texture complète au CPU pour lui faire traiter ces voxels, c'est-à-dire les fusionner en motifs et classer ces motifs. Cependant, pour ce traitement, le GPU n'est pas utilisé à son maximum. Nous pouvons donc l'utiliser pour le faire continuer à traiter ces voxels, afin de limiter l'utilisation du CPU. Pour cela, nous allons indexer notre texture pour indiquer au CPU où se trouve ntles voxels qui constituent les motifs et la taille de ces motifs. Ainsi le CPU n'aura plus qu'à recopier les composantes ARGB des voxels dans les motifs, il n'aura plus à détecter leur position, ni leur taille.

L'idée la plus simple pour réaliser cela, est de parcourir la texture résultat, de détecter le début des motifs et de compter le nombre de voxels jusqu'à leur fin. Cependant, cet algorithme trivial ne tire pas avantage du parallèlisme du GPU. Nous avons donc adapté un algorithme beaucoup plus efficace, proposé par W. D. Hillis [49], qui permet de compter la longueur d'une suite d'éléments quelconques, avec une complexité de o(log n), en utilisant le parallèlisme du GPU. Son principe est de prendre un tableau de taille égale au nombre total d'éléments à traiter, de traiter tous les éléments en parallèle, et de mémoriser, dans la case du tableau correspondant à chaque élément, la distance qui le sépare de la fin de la suite d'éléments. Une fois l'algorithme exécuté, la première case du tableau contient la taille de la liste. Cet algorithme peut être adapté pour compter la taille de plusieurs listes à l'intérieur d'un ensemble. Ici, nous allons compter les voxels qui sont dans un même plan, en traitant les cases ne contenant pas de voxel comme des voxels placés à une distance infinie. Ainsi, l'algorithme comptera aussi la longueur des

zones ne contenant pas de voxels, ce qui indiquera la position du motif suivant.

Cet algorithme se décompose en trois parties : la première (dont le pseudo-code est indiqué dans l'algorithme n°14 : **Détection du début des motifs**) initialise la zone mémoire résultat et détecte le début et la fin des motifs. Le résultat est indiqué dans

Algorithme n° 14 Détection du début des motifs

Données : Une texture de taille largeur × hauteur contenant les profondeurs des voxels, notée *pixel*
Données : Une zone mémoire de taille largeur × hauteur où écrire la valeur 0 dans les cases correspondant aux voxels de début des motifs et $+\infty$ dans les autres, notée *résultat*

```
1: Pour chaque ligne de la texture , faire
2:    resultat[ligne, 0] ← 0
3:    Pour chaque colonne de la texture sauf la première et la dernière , faire
4:       Si abs(profondeur du pixel[ligne, colonne] - profondeur du pixel[ligne, colonne + 1]) > distance
         entre les plans , alors
5:          résultat[ligne, colonne] ← 0
6:       Sinon
7:          résultat[ligne, colonne] ← +∞
8:       Fin Si
9:    Fin Pour
10: Fin Pour
```

une zone mémoire dont la taille est égale au nombre de pixels de la texture résultat de l'algorithme précédent. En réalité, la première passe de l'algorithme conduit à un accès aux cases d'une colonne trop loin. Il faut donc prévoir ce dépassement en rajoutant une colonne à la zone mémoire ou en réglant l'accès à cette mémoire par le GPU de telle sorte qu'une tentative d'accès à la (n+m)ème colonne (où m est le nombre de colonne de la texture donnée en entrée) conduise à l'accès réel de la n-ième colonne. Cette zone mémoire contiendra la valeur zéro dans les cases correspondant aux premiers voxels des motifs (y compris des zones où il n'y a pas de voxels et qui seront considérées comme des motifs ayant une profondeur de $+\infty$) et $+\infty$ dans les autres cases.

La seconde partie de cet algorithme (dont le pseudo-code est indiqué dans l'algorithme n°15 : **Comptage de la longueur d'un motif**) compte la longueur de chaque motif. On commence par prendre tous les voxels qui n'ont pas encore été traités (leur distance est toujours à $+\infty$). Pour chacun de ces voxels, on regarde le voxel suivant. Si celui-ci a déjà été traité, la distance du voxel non traité est égale à la distance du voxel suivant incrémenté de 1. Ensuite on recommence en comparant chaque voxel non traité avec celui se trouvant 2 voxels plus loins et, s'il a été lui-même traité, la distance du

Algorithme n° 15 Comptage de la longueur d'un motif

Données : Une texture de taille largeur × hauteur (notée *résultat*) contenant 0 dans les cases correspondant au 1er voxel de chaque motif et $+\infty$ dans les autres (cette texture est fournie par l'algorithme n°14) qui sert également à mémoriser le résultat

```
1: shift ← 1
2: Tant que shift < largeur, faire
3:    Pour chaque ligne, faire
4:       Pour chaque colonne, faire
5:          Si résultat[ligne, colonne] ≠ +∞, alors
6:             Si résultat[ligne, colonne + shift] ≠ +∞, alors
7:                résultat[ligne, colonne] ← résultat[ligne, colonne + shift] + shift
8:             Fin Si
9:          Fin Si
10:      Fin Pour
11:   Fin Pour
12:   shift ← shift * 2
13: Fin Tant que
```

voxel non traité est égale à celle du voxel traité placé à 2 voxels plus loin à laquelle on ajoute 2. Ensuite, on recommence avec 4, 8 ... 2^p, où $p = log(largeur) - 1$. La valeur de la variable *shift* indique quel voxel comparer avec le voxel actuel : si elle est égale à 1, on regarde le voxel suivant, si elle est égale à 2, on regarde le voxel situé deux cases plus loin dans le tableau, etc. Cette variable prend successivement les puissances de deux successives, en commençant à 2^0, c'est-à-dire 1, puis 2, 4, 8, etc. Au bout de log(largeur) itérations, toutes les cases de la zone mémoire résultat contiennent la distance séparant le voxel correspondant de la fin du motif + 1, sauf pour le premier, qui contient 0. La figure 3.8 montre un exemple de ce que peut contenir une ligne du bloc mémoire résultat à la fin de l'algorithme n°15. Celui-ci contient quatre motifs ayant une taille de 6, 8, 4

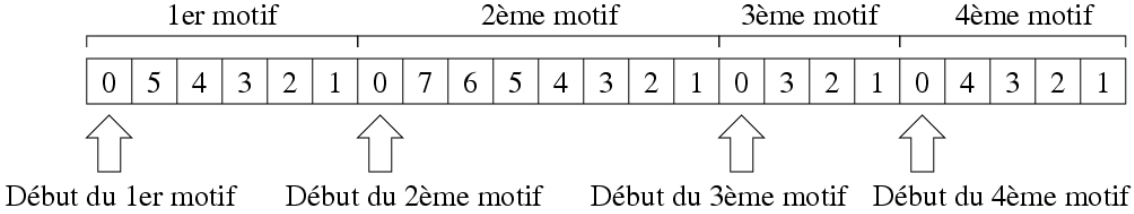


FIG. 3.8 – Exemple de résultat de l'algorithme de comptage de la longueur d'un motif

et 5 voxels, respectivement. La figure 3.9 montre l'exécution de l'algorithme n°15 pour l'exemple de la figure 3.8.

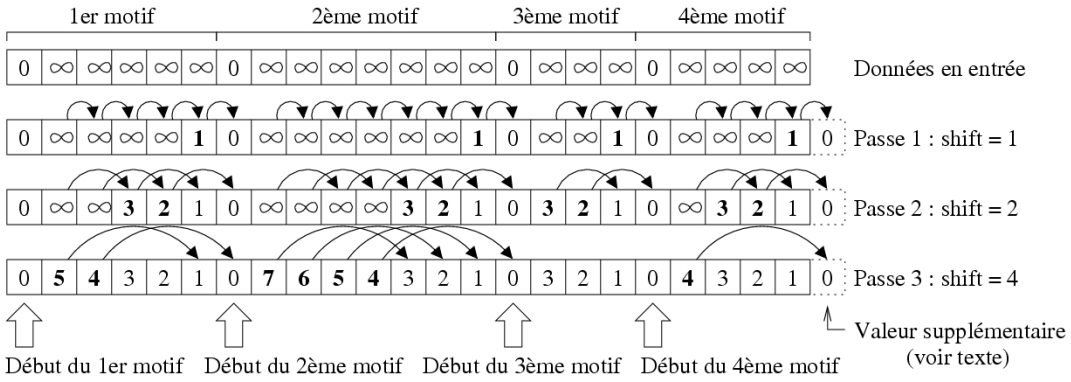


FIG. 3.9 – Exécution de l'algorithme de comptage de la longueur d'un motif

La troisième et dernière partie de cet algorithme consiste à remplacer les zéros, indiquant le début des motifs par la longueur réelle de chaque motif. Cet algorithme est

Algorithme n° 16 Mémorisation de la longueur réelle d'un motif

Données : Une texture de taille largeur × hauteur (notée *résultat*) contenant 0 dans les cases correspondant au 1er voxel de chaque motif et $+\infty$ dans les autres (fournie par l'algorithme n°15) qui sert également à mémoriser le résultat

```
1: Pour chaque ligne, faire
2:    Pour chaque colonne sauf la dernière, faire
3:       Si résultat[ligne, colonne] == 0, alors
4:          résultat[ligne, colonne] ← résultat[ligne, colonne + 1] + 1
5:       Fin Si
6:    Fin Pour
7:    Si résultat[ligne, dernière colonne] == 0, alors
8:       résultat[ligne, dernière colonne] ← 1
9:    Fin Si
10: Fin Pour
```

décrit sous la forme algorithme n°16 : **Mémorisation de la longueur réelle d'un motif** et consiste à parcourir le bloc mémoire à la recherche des zéros et, pour chaque zéro trouvé, on le remplace par la valeur du voxel situé juste à droite + 1 ou 1 si ce voxel est situé sur la dernière colonne. Le résultat obtenu par cet algorithme pour l'exemple indiqué figure 3.8 est représenté figure 3.10. La complexité de cet algorithme est en o(largeur × hauteur).

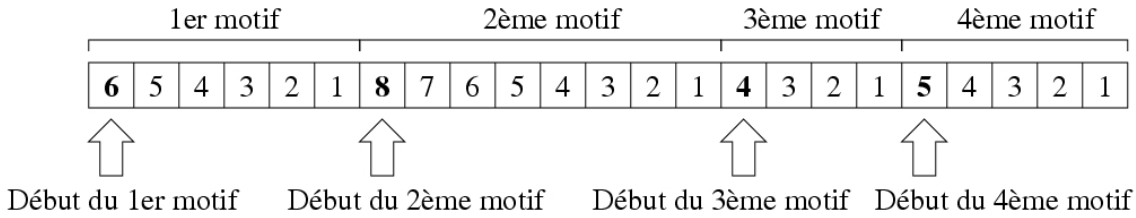


FIG. 3.10 – Exemple de résultat de l'algorithme de mémorisation de la longueur d'un motif

Mémorisation, tri des motifs et mise en forme

Une fois ces traitements effectués, le CPU pourra récupérer ces deux textures : celle contenant la couleur et la profondeur des voxels, et celle générée par l'algorithme 16. Pour récupérer les informations nécessaires, il lui suffit, pour chaque ligne, de lire la profondeur du premier voxel. Si celle-ci est $+\infty$, la première valeur de la même ligne dans la texture contenant les longueurs contient la position d'un premier voxel du premier motif, sinon cette valeur indique la longueur du motif dont fait partie ce voxel. puis on réitère la même opération à partir du premier voxel du motif suivant. Si le premier motif est à une profondeur de $+\infty$, le suivant sera obligatoirement un motif réel à coder, sinon le motif suivant peut soit être un motif réel à coder, soit une zone vide (dont les voxels sont à une distance de $+\infty$). Cet algorithme est détaillé sous l'appellation algorithme n°17 : **Récupération des motifs**. Cet algorithme a une complèxité en o(l), où l est le nombre total de motif dans l'ensemble des lignes de l'image.

3.2.3 Implémentation des algorithmes

Afin de valider l'adaptation de nos traitements sur GPU, nous les avons implémentés sur le GPU d'une carte graphique et nous rapatrions les textures résultats sur le CPU qui extrait les motifs, les trie, crée les plans et recode l'ensemble des données dans un fichier qui sera ensuite lu par nos simulateurs. La figure 3.11 montre la répartition des traitements entre le GPU et le CPU.

Pour ce projet, nous avons choisi d'utiliser HLSL (langage de programmation GPU de DirectX à partir de la version 9) et DirectX car les plateformes de développement

Algorithme n° 17 Récupération des motifs

Données : Une texture contenant les couleurs et profondeurs des voxels calculés, notée *voxel*
Données : Une texture de taille largeur × hauteur fournie par l'algorithme n°16, notée *profondeur*, pour la texture ci-dessus

```
1:  Pour chaque ligne, faire
2:      x ← 0
3:      Tant que x < largeur, faire
4:          Si profondeur[ligne, x] == + ∞, alors
5:              x ← x + profondeur[ligne, x]
6:          Sinon
7:              Crée un motif de taille profondeur[ligne, x]
8:              Pour i allant de 0 à profondeur[ligne, x] - 1, faire
9:                  motif[i] ← voxel[x + i]
10:             Fin Pour
11:             Insérer motif dans ligne, à la position x
12:             x ← x + profondeur[ligne, x]
13:         Fin Si
14:     Fin Tant que
15: Fin Pour
```

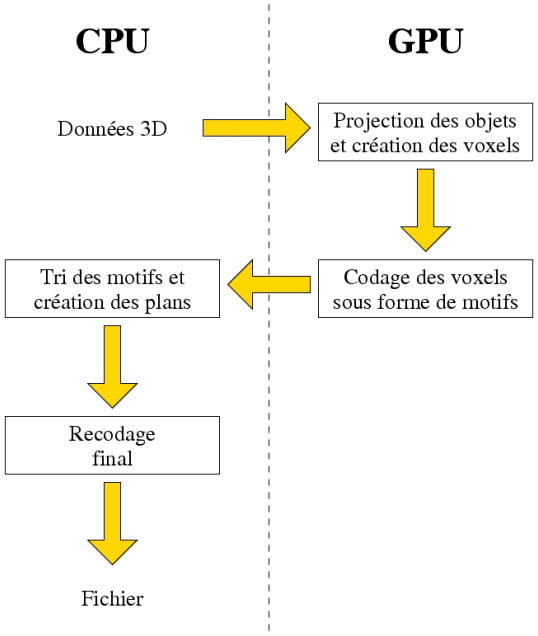


FIG. 3.11 – Adéquation GPU/CPU pour la validation de l'implémentation sur GPU

et de test utilisant des cartes de constructeurs différents, cette solution réduit le risque d'incompatibilité. La solution qui consiste à utiliser un langage de haut niveau comme Brook n'a pas été choisie car :

– le but du programme est d'avoir le maximum de performances possibles, et pour

cela, une optimisation bas niveau écrite à la main est préférable

– il y a énormément de fonctions réutilisables pour notre implémentation dans les bibliothèques graphiques et, bien qu'il soit possible d'utiliser les deux en même temps, il aurait fallu convertir les données pour passer d'un mode à l'autre, entraînant des calculs supplémentaires.
– il est plus facile d'adapter un algorithme décrit dans la littérature.

Un des choix difficiles à effectuer est la précision des calculs : faut-il préférer la vitesse de calcul, l'espace mémoire ou la précision ? Utiliser des textures 16 bits permet de diminuer le temps de calcul, la taille mémoire et, par conséquent, leur vitesse de transfert. Par contre, la précision des nombres réels représentés est alors réduite. Cela peut poser des problèmes pour les voxels placés loin de l'écran car deux voxels ayant des coordonnées Z très grandes et très proches l'une de l'autre risquent de ne plus pouvoir être différenciés car leur coordonnées Z auront été arrondies à la même valeur. Dans ce cas, et si ces deux voxels ont les mêmes coordonnées x et y, il devient impossible de choisir lequel des deux garder. En effet, dans le cas où deux voxels ont les mêmes coordonnées x et y, il faut ne prendre que celui qui est le plus proche, c'est-à-dire qui a la coordonnée z la plus faible. Si ce choix est impossible, il est possible que le mauvais voxel soit pris et certains voxels qui ne devraient pas apparaître deviennent visibles. Au final, nous avons opté pour tout garder en 32 bits car le facteur limitant n'est pas la puissance de calcul et que changer la taille des valeurs ne changera pas la taille ni le nombre des textures résultats.

Ensuite, il fallait limiter au maximum la quantité de données à envoyer au CPU, afin d'éviter un ralentissement des traitements trop importants à cause de la faible vitesse des transferts du GPU vers le CPU. Nous avons six données à envoyer au CPU : quatre pour la couleur et la transparence (ARGB), un pour la profondeur (Z) et un pour la longueur du motif (M). Or, nous avons plusieurs contraintes sur ces textures :

– une texture peut contenir une, deux ou quatre informations du même format, il nous faut donc utiliser au moins deux textures,
– toutes ces textures doivent avoir la même taille en bits par pixel,

- la valeur Z doit être en virgule flottante, de préférence sur 32 bits,
- la valeur M doit être exacte au moins jusqu'à la largeur de la fenêtre, donc au moins sur 16 bits entiers ou flottants.

La solution retenue est la suivante : les composantes ARGB dans une première texture à 4 emplacements entiers sur 8 bits par pixel, soit 32 bits par pixel au total, Z et M dans une seconde texture à 2 emplacements flottants sur 16 bits par pixel, soit 32 bits par pixel au total. On arrive à 64 bits par pixel à transmettre du GPU vers le CPU.

La programmation du GPU et la récupération des données étant des opérations bloquantes, et pour utiliser pleinement les capacités de calcul des deux parties, nous avons créé plusieurs threads dans le code du CPU : une de ces threads s'occupe des appels bloquants vers le GPU et l'autre thread s'occupe des calculs restants sur les données résultats récupérées après traitement par le GPU.

Les dernières passes de l'algorithme n°13 (depth peeling) ne génèrent qu'un nombre réduit de voxels (inférieur à une centaine). Ces voxels étant placés derrière plusieurs couches de voxels, il est probable qu'ils ne soient pas visibles et seront donc effacés par l'algorithme n°10 d'effacement des voxels cachés. Il est donc possible de limiter le nombre de passes nécessaires en arrêtant l'exécution de cet algorithme avant le traitement de tous les voxels, par exemple, en s'arrêtant lorsque le nombre de voxels générés est inférieur au seuil défini.

L'opération spéciale "occlusion query", permettant de savoir combien de pixels ont été écrits, est une commande non bloquante qui prend un temps non négligeable à s'exécuter sur le GPU. Au lieu d'écrire une boucle de type :

```
Demande de l'occlusion query
Tant que le résultat n'est pas disponible
  Lecture de la valeur
Fin tant que
```

ou bien :

```
Demande de l'occlusion query
Lecture de la valeur
Tant que le résultat n'est pas disponible
  Attente
```

```
  Lecture de la valeur
Fin tant que
```

il est préférable de lancer, sur le GPU, des calculs intermédiaires pour lui laisser le temps d'envoyer la valeur, quitte à effectuer des calculs supplémentaires qui se révèleront inutiles.

Le code sur GPU utilise dès que possible la possibilité de quitter prématurément le traitement en cours d'exécution grâce à la commande *clip()* qui permet de ne pas avoir de sortie pour un pixel donné. On économise ainsi une instruction d'écriture en mémoire et, dans le cas où tous les pixel shaders prennent cette branche, on peut passer directement aux pixels suivants.

3.2.4 Résultats et conclusions de l'implémentation sur GPU

Notre implémentation de notre algorithme de génération et de codage des voxels sur GPU ne comportait ni l'effacement des voxels cachés, ni la correction du problème de discontinuité des facettes. Il se contentait de générer les voxels et de les regrouper en motifs. Cependant, l'exécution des algorithmes que nous avons implémentés permet d'atteindre des vitesses de 30 images par seconde avec des images de 640 par 480. Cette résolution a été choisie car elle permettait d'afficher le résultat de la simulation et les données de profiling (vitesse de génération des images, taux d'utilisation du GPU, ...) en même temps sur le même écran.

Pour le traitement simple des données, sans les rapatrier sur le CPU, nos tests, réalisés sur une carte NVidia GeForce 7800 et sur une carte ATI Radeon x800, atteignent 30 images par seconde, avec une utilisation du GPU de l'ordre de 20% seulement sur la Radeon. Cela signifie que le GPU est capable de générer des images plus grande et de traiter un plus grand nombre de facettes et de voxels, mais aussi qu'il est possible de lui faire exécuter d'autres traitements, tout en respectant la contrainte temps-réel. Ce taux de rafraichissement est une limite logicielle fixée par le mode d'utilisation de la carte. Ces deux cartes sont équipées de GPUs implémentant les shaders version 3.0 qui sont nécessaires car ils permettent l'utilisation de l'instruction *vpos*, que nous utilisons

dans notre implémentation. Cette instruction permet de récupérer la position du voxel en cours de traitement sur l'écran, c'est-à-dire les coordonnées x et y que nous utilisons dans les algorithmes n°14 (Détection du début des motifs), 15 (Comptage de la longueur d'un motif) et 16 (Mémorisation de la longueur réelle d'un motif). Le taux d'utilisation du GPU a été déterminé par le logiciel de profiling NVPerfHUD4 [12], développé par NVidia. Ce logiciel permet d'exécuter un programme utilisant la carte graphique (développée par NVidia ou par un concurrent) et d'afficher pendant son exécution différentes informations de profiling, comme le nombre d'images calculées par seconde, l'utilisation du GPU, la quantité de mémoire de la carte utilisée, etc.

Cependant, la vitesse de génération des données sans rapatriement vers le CPU est suffisante pour permettre d'utiliser un GPU dans une architecture spécialisée.

3.3 Modélisation de la chaîne de traitements

Pour valider le principe d'architecture spécialisée permettant d'opérer nos traitements, nous avons modélisé de manière fonctionnelle cette chaîne. Elle est composée d'un GPU, opérant les traitements dont nous venons de parler et d'une partie codage, qui récupère les données générées par le GPU, les trie, les mémorise et les remet en forme (algorithmes n°2 et n°5) avant de les envoyer au dispositif d'affichage.

Le GPU a été modélisé par un programme envoyant les voxels dans la partie codage de la chaîne de traitements. Pour la modélisation de la partie codage, nous avons découpé l'algorithme en plusieurs blocs de traitement, en fonction des informations modifiées, fonctionnant de manière asynchrone car les temps d'exécution des différents blocs ne sont pas identiques, non seulement d'un bloc à l'autre, mais également d'un motif à l'autre. Certains de ces blocs de traitement accèdent à des blocs mémoire servant à mémoriser les différentes informations nécessaires au tri et les données reçues du GPU, comme indiqué figure 3.12. Chaque bloc mémoire (notés A, B et C sur la figure) mémorise les informations utiles au bloc de traitement (notés 3, 4 et 5, respectivement sur la figure) auquel il est rattaché. Tous ces blocs mémoire sont également rattachés au bloc de traitement *"Recodage"* qui lit la totalité des informations contenues dans les différentes

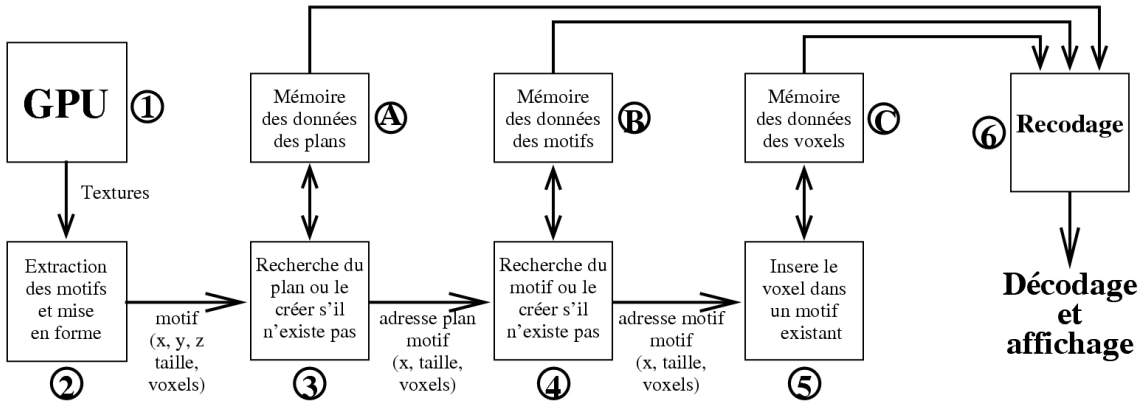


FIG. 3.12 – Modélisation de l'implémentation du codage sur architecture spécialisée

mémoires une fois que tous les motifs ont été mémorisés et remet toutes ces informations en forme pour les envoyer au dispositif d'affichage. Les informations contenues dans les différents blocs mémoire sont schématisées sur la figure 3.13.

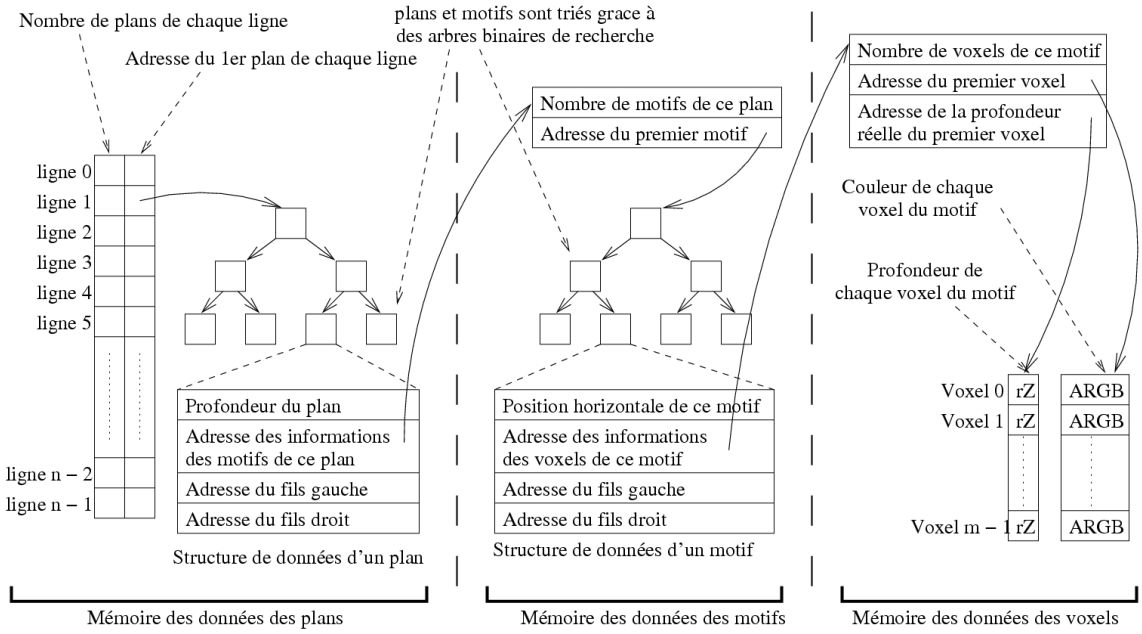


FIG. 3.13 – Structure de données utilisée pour l'implémentation du codage et la répartition des données dans les différentes mémoires

Le premier bloc (noté 1 sur la figure 3.12) représente le GPU, qui génère les textures contenant les motifs dont nous avons parlé précédemment.

Le second bloc extrait les données des textures, en utilisant l'algorithme n°17 (Récupération des motifs) pour envoyer chaque motif indépendamment au bloc suivant du pipe-line.

Le troisième bloc recherche, parmi les plans de la ligne correspondant à la coordonnée y du motif à insérer, celui où doit être inséré le motif. Si ce plan n'existe pas, il le crée. Le bloc mémoire auquel il accède (noté A sur la figure 3.12) contient un tableau (représenté figure 3.13) contenant le nombre de plans de chaque ligne et l'adresse (dans la zone mémoire A) de la racine de l'arbre binaire de recherche servant à mémoriser et à trier les plans de cette ligne (si elle en contient au moins un) et les arbres contenant les plans de toutes les lignes, chaque ligne contenant au plus un arbre. La structure de données des noeuds de ces arbres (également représentée figure 3.13) contient la profondeur du plan auquel il se rapporte, ainsi que l'adresse (dans le bloc mémoire B, cette fois) des informations relatives aux motifs contenus dans ce plan : leur nombre et l'adresse (toujours dans le bloc mémoire B) de la racine de l'arbre binaire servant à les trier et les mémoriser. Une fois le plan trouvé ou créé, le bloc envoie au bloc suivant les informations du bloc, à part sa coordonnée z qui est inutile aux blocs suivants, l'adresse (dans le bloc mémoire B) du plan dans lequel insérer le motif et un signal pour indiquer si cette adresse correspond à un nouveau plan ou pas.

Le quatrième bloc recherche si le motif à insérer doit fusionner à gauche avec un motif déjà présent dans le plan dont l'adresse est passée en entrée, c'est-à-dire un motif dont un des voxels correspond aux coordonnées x, y et z du premier voxel (celui le plus à gauche) du motif à insérer (schéma a de la figure 3.14), ou qui se trouve juste à coté de ce motif (schéma b de la figure 3.14). En d'autres termes, si la coordonnée x du dernier voxel le plus à droite du motif du plan est notée x_1 et la coordonnée x du voxel le plus à gauche du motif à insérer est notée x_2, ce cas se présente quand on a l'égalité $x_1 = x_2 - 1$. Si un tel motif n'est pas trouvé, ce bloc recherche un motif englobé dans le motif à insérer (schéma f de la figure 3.14) ou un motif avec lequel le motif à insérer doit fusionner à droite (schéma b de la figure 3.15). Si aucun de ces motifs n'est trouvé, on crée un nouveau motif dans l'arbre. Sinon, on regarde les motifs suivants pour détecter

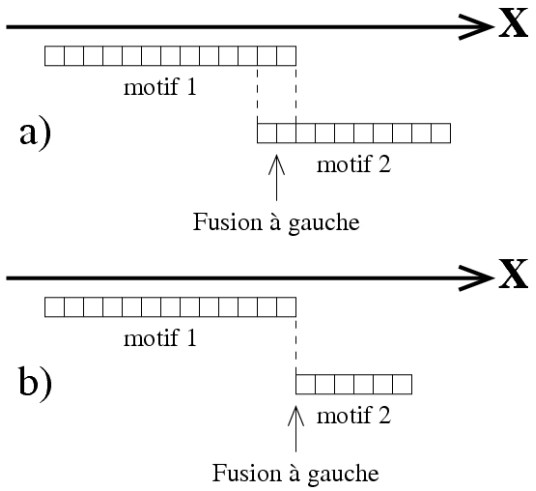


FIG. 3.14 – Principe de la fusion à gauche lors de l'insertion d'un motif

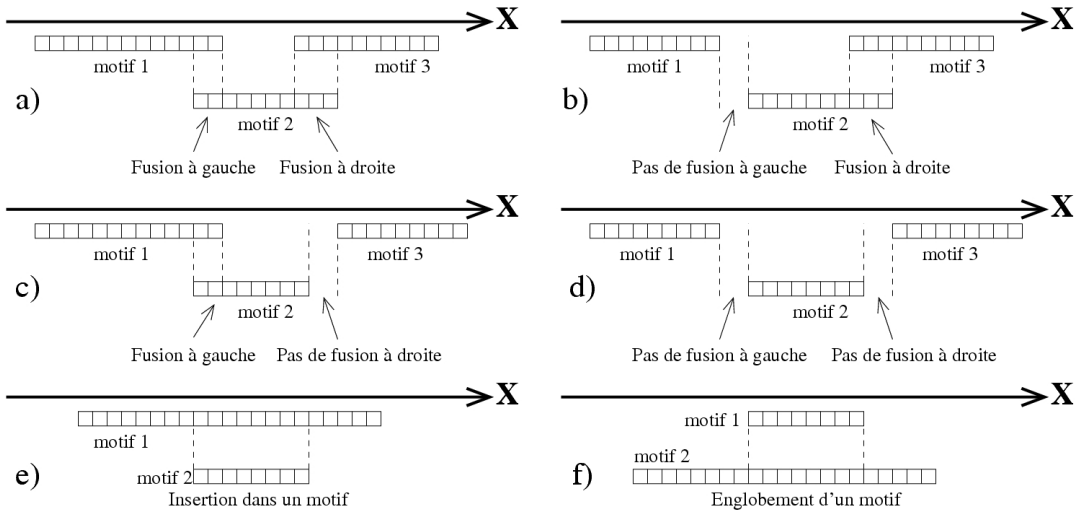


FIG. 3.15 – Différents cas pouvant intervenir au moment de l'insertion d'un motif dans un plan

tous les motifs englobés (schéma f de la figure 3.14) ou fusionnés à droite (schéma b de la figure 3.15). Tous ces motifs détectés sont alors fusionnés en un seul nouveau motif dont l'adresse est passé au bloc suivant (n° 5 sur la figure 3.12), avec l'adresse de tous les autres (motif inséré et tous les motifs englobés ou fusionnés à gauche ou à droite) pour recopier les valeurs des voxels de ces motifs dans le nouveau et les anciens motifs fusionnés sont effacés.

Le dernier bloc (n° 5 sur la figure 3.12) s'occupe alors de déplacer les couleurs et profondeurs des voxels (anciens motifs à effacer et nouveau motif à insérer) en fonction de leur position dans le nouveau motif. Ce bloc s'occupe également, dans le cas où plusieurs motifs se trouvent au même endroit, de déterminer quels voxels conserver et quels voxels effacer, en fonction de leur profondeur réelle, ou les fusionner si le plus proche est transparent.

Une fois que tous les voxels ont été mémorisés, la partie recodage (n° 6 sur la figure 3.12) lit toutes ces informations dans les différentes mémoires (A, B et C sur la figure 3.12) et les restitue au dispositif d'affichage. Comme la mise en forme doit se faire une fois que tous les éléments (plans, blocs) sont classés, aucune information ne peut être fournie avant que tous les voxels n'aient été mémorisés. Il est possible, pour accélérer les traitements et réduire le temps d'attente entre les phases de codage, d'utiliser deux blocs mémoire distincts (comme ceux schématisés figure 3.16) dans chacun des blocs mémoire, afin que, pendant la phase de recodage, le pipe-line puisse recommencer à mémoriser des voxels. Lorsque la mémorisation et le codage sont terminées, on inverse les deux bancs

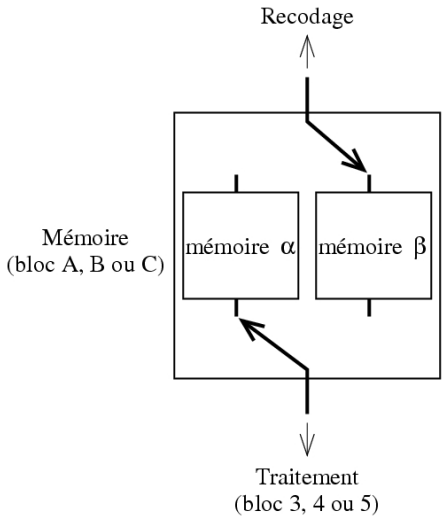


FIG. 3.16 – Principe de la mémoire double

mémoire pour passer à la phase suivante. Ainsi, dans un premier temps, les blocs 3, 4 et 5 écrivent les données dans la mémoire notée α sur la figure 3.16 des mémoires A, B et C. Puis, une fois que tous les voxels ont été mémorisés, le bloc *Recodage* lit ces

données pendant que les blocs 3, 4 et 5 peuvent de nouveau mémoriser des données dans la mémoire β. Une fois que les données de la mémoire α ont été recodées et que tous les voxels de l'image suivante ont été mémorisés dans la mémoire β, le bloc recodage peut recoder les données des blocs β, pendant que les autres blocs du pipe-line mémorisent de nouveux voxels dans les mémoires α. Le pipe-line et le recodage fonctionnent donc en parallèle, ce qui augmente la vitesse de l'architecture : le temps séparant l'envoi au dispositif d'affichage de deux images 3D successives passe donc de $t1 + t2$ à $max(t1, t2)$, avec $t1$ le temps d'exécution de la mémorisation des données et $t2$ le temps d'exécution du recodage.

3.3.1 Simulations et résultats

Cette simulation a été réalisée en *SystemC*, grâce au logiciel de modélisation et de simulation *Visual Elite*, commercialisé par la société *Summit*.

SystemC [15] est, comme *Verilog* et *VHDL*, un langage de description de matériel. C'est le fruit de contributions de plusieurs sociétés. En 1989, *Synopsys* met son outil commercial *Scenic* dans le domaine libre, et crée la version 0.9 de SystemC. Une première contribution de *Frontier Design* donne lieu à la version 1.0, et une autre de *CoWare* aboutit en 2000 à la version 1.1, première version officielle de SystemC. L'OSCI (Open SystemC Initiative) est alors créé, rassemblant une multitude de sociétés et laboratoires de recherche. Cette organisation est en charge de diffuser, promouvoir et rédiger les spécifications de SystemC. Les spécifications de SystemC ont été étendues en 2001 à la modélisation de systèmes abstraits (de très haut niveau, avant partitionnement matériel/logiciel), aboutissant à la version 2.0.

SystemC a pour objectif de modéliser des systèmes numériques matériels et logiciels à l'aide de C++. Il modélise donc non seulement des systèmes matériels, mais aussi des systèmes logiciels, mixtes ou non-partitionnés. Comme en Verilog ou VHDL, un système est une hiérachie d'objets. Généralement ce sont des modules imbriqués et/ou des processus. Les modules communiquent entre eux par des canaux. Le plus haut de la

hiérarchie d'un système complet (module à tester + testbench) est la fonction SystemC sc_main (l'équivalent du main des programmes en C ou C++). Un module en SystemC est composé d'autres modules, de canaux de communication entre ces modules (signaux, ou canaux plus abstraits), et éventuellement de processus. Un module possède un ou plusieurs ports. Les ports sont juste des points d'entrée ou de sortie, qui ne font rien de particulier. Par contre, les ports doivent déclarer les fonctions qui seront utilisées pour communiquer à travers eux. Par exemple, un port en entrée destiné à être relié à un signal déclare qu'il utilise la fonction read des signaux, un port similaire mais bidirectionnel déclare qu'il utilisera les fonctions read et write et un port destiné à être relié à une fifo (un canal de communication abstrait, de haut niveau), déclarera selon le côté de la fifo où il est censé se trouver, qu'il utilisera les fonction read, nb_read (read non bloquant), num_available, write, nb_write, num_free, etc. Les canaux sont les moyens de communication entre les modules. Ils peuvent être basique et concrets (comme des signaux), ou plus évolués (fifo, réseau etc.). Ils peuvent aussi contenir d'autres canaux, voire même des modules si ce sont des canaux de très haut niveau. Les processus en SystemC sont similaires à ceux de Verilog et VHDL. Il décrivent une fonctionnalité, un comportement. Un processus ne doit pas être appelé directement. C'est le moteur de simulation SystemC qui se charge de l'appeler (le déclencher) sur certains évènements particuliers. Les processus peuvent éventuellement communiquer directement avec les canaux.

Voici un exemple de code en *systemC*, définissant une porte combinatoire *AND* à trois entrées :

Exemple de modélisation d'une porte AND à trois entrées en SystemC

```
1  SC_MODULE(and3)
2  {
3    sc_in<bool> e1;
4    sc_in<bool> e2;
5    sc_in<bool> e3;
6    sc_out<bool> s;
7
8    void compute_and()
9    {
10     s = e1 & e2 & e3;
11   };
```

```
12
13    SC_CTOR(and3)
14    {
15      SC_METHOD(compute_and);
16      sensitive(e1);
17      sensitive(e2);
18      sensitive(e3);
19    }
20  };
```

Le bloc SC_MODULE(and3) (de la ligne 1 à la ligne 20) indiqué dans cet exemple crée un module nommé *and3*. Dans ce module, on commence par définir les canaux de communication (lignes 3 à 6) : trois signaux booléens en entrée, nommés *e1*, *e2* et *e3* et un signal booléen en sortie, nommé *s*. La fonction *compute_and* (lignes 8 à 11) est la fonction qui sera appelée par le moteur de simulation SystemC au début de l'exécution du programme, pour initialisation ou lorsqu'un ou plusieurs signaux en entrée change d'état. Cette fonction détermine l'état du signal de sortie *s* en fonction des trois signaux d'entrée *e1*, *e2* et *e3*. Enfin, le bloc *SC_CTOR(and3)* (lignes 13 à 19) détermine quelle fonction doit être appelée (ici, la fonction *compute_and*), indiquée par le mot-clé *SC_METHOD* (ligne 15) et quels évènements (lignes 16 à 18) entraînent l'appel de cette fonction, ici les trois signaux d'entrée *e1*, *e2* et *e3*, indiqués par le mot-clé *sensitive*. Dans cet exemple, dès qu'un des signaux d'entrée est modifié par le moteur de simulation SystemC (signaux provenant de l'extérieur du système modélisé) ou par le changement d'état du signal de sortie d'un autre module, la fonction *compute_and* est appelée et change le signal de sortie. Si ce signal de sortie correspond à un signal en entrée d'un autre module, la fonction associée à ce signal sera à son tour appelée et ainsi de suite.

Visual Elite [18] est un logiciel de conception graphique hiérarchique pour SystemC, de modélisation et de simulation, permettant de modéliser à plusieurs niveaux de précision une architecture, développé par la société Summit Design [14]. Cette architecture est définie sous forme de blocs, reliés entre eux par des signaux de synchronisation

et/ou de données. Ensuite, chaque bloc peut être défini soit par un programme C, C++, systemC, HDL, soit par une machine à états, soit par un bloc diagramme qui peut, à son tour, être affiné. Une fois ces modules décrits, l'ensemble peut être simulé et tous les blocs fonctionneront dans le système, indépendamment du langage utilisé pour leur définition. La définition des blocs peut se faire soit entièrement graphiquement, soit par l'intermédiaire d'un éditeur de textes intégré à l'interface de Visual Elite et permettant d'éditer directement la définition de chaque bloc, sauf les machines à états, qui ne peuvent être éditées que par l'interface graphique.

La modélisation du système se fait en plusieurs étapes. Premièrement, la définition des blocs. L'utilisateur crée les blocs selon le type voulu pour chacun : bloc défini en HDL, en SystemC, machine à état, etc. Cette étape se fait grâce à l'interface graphique du logiciel. L'utilisateur place les blocs dans l'environnement graphique pour former un graphique représentant l'ensemble du système. La position des blocs et leur taille n'influent pas du tout sur la simulation ni la définition du bloc, mais permet un affichage du système sous forme graphique plus claire. Ensuite, chaque bloc doit être édité pour définir les signaux en entrée et en sortie qu'il doit avoir pour pouvoir connecter les différents blocs entre eux. Cette définition se fait grâce à une fenêtre, dépendant du type de bloc. Pour chaque signal en entrée ou en sortie, le type et la taille doivent être définis. Il est également possible de rassembler plusieurs signaux, afin d'améliorer la visibilité du graphique d'ensemble. Puis on relie entre eux les blocs en créant des connexions entre les différens signaux. Pour que l'ensemble puisse fonctionner, il faut, bien entendu, que les types des signaux reliés soient compatibles. Il est également possible de nommer deux signaux de deux blocs différents avec le même nom, ce qui a le même effet que de les relier, mais évite de surcharger le graphisme.

La simulation se fait ensuite grâce à l'interface graphique, en sélectionnant un certain nombre de signaux à surveiller et les signaux entrant à définir. Une fenêtre contenant les états des signaux au cours de la simulation s'affiche et permet à l'utilisateur d'analyser l'état des signaux et le comportement des différents blocs. Il est également possible de lire des données en entrée ou d'écrire des données résultats dans des fichiers.

Dans cette simulation, nous nous sommes limités à une modélisation fonctionnelle de haut niveau en ne définissant que les blocs de traitement à l'aide de programmes exécutant les tâches assignées à chaque bloc. Nous avons simulé le comportement de chaque bloc (y compris les blocs mémoire) indépendamment. Chacun d'eux était modélisé par un programme, écrit en SystemC, et permettant de simuler son comportement. La simulation lit les données en entrée dans un fichier et écrit les données générées devant être envoyées au dispositif d'affichage dans un second fichier qui pourra ensuite être lu par un de nos simulateurs.

L'ensemble de la chaîne de traitements doit répondre à un certain nombre d'exigences. La contrainte la plus forte est celle inhérente à son utilisation temps-réel. Cette exigence nécessite que l'ensemble des voxels d'une image 3D soit généré en un trentième de seconde, au plus. Ceci signifie que la mémorisation de tous ses voxels, c'est-à-dire le traitement par le GPU (bloc 1 sur la figure 3.12) de tous les points de tous les objets de la scène et la génération de tous les voxels correspondant, ainsi que le traitement de ces voxels par les blocs 2 à 5 du pipe-line (figure 3.12), mais aussi le recodage de ces données (bloc 6 de la figure 3.12) doivent se faire en un trentième de seconde au plus. Ces deux traitements étant exécutés en parallèle, le temps global séparant l'envoi de deux images 3D successives au dispositif d'affichage est égal à la valeur maximale des temps d'exécution de la mémorisation de toutes les données et de leur recodage.

Cette simulation a permi de valider le découpage de l'algorithme de codage en blocs (les 6 blocs schématisés figure 3.12), ainsi que le découpage de la mémoire (comme schématisé figure 3.13).

L'ensemble de cette chaîne de traitements, utilisant un GPU devrait permettre de générer les données nécessaires à l'affichage 3D en moins d'un trentième de seconde, ce qui est la condition nécessaire à son utilisation temps-réel.

3.4 Faisabilité physique du dispositif

Le deuxième problème à élucider est la vitesse d'affichage des vues. En effet, si on considère que notre afficheur doit pouvoir afficher au moins 200 vues pour chacune des 30 images 3D à afficher par seconde, cela représente $30 \times 200 = 6\ 000$ vues à afficher, donc à décoder, par seconde. La solution technique que nous avons envisagée étant un multiplexage temporel, ces 6 000 vues doivent être affichées séquentiellement. Nous avons donc besoin d'un dispositif permettant d'afficher ces images suffisamment rapidement. Pour cela, ce dispositif doit être capable

- d'afficher au moins 6 000 images par seconde,
- de passer d'une image à l'autre en $\frac{1}{60000}$ de seconde au plus, ceci afin de permettre l'affichage de chaque image pendant $\frac{9}{60000}$ de seconde et de limiter la zone de "mélange" entre deux images successives à moins de 10% de la zone d'affichage de chacune de ces deux images,
- de conserver l'affichage de ces images stable pendant un temps au moins égal à 90% du temps alloué à l'affichage de chaque image, c'est-à-dire $\frac{1}{6000}$ de seconde, afin d'assurer un maximum de luminosité,
- d'afficher ces images avec une intensité lumineuse suffisante pour que l'œil de l'observateur puisse la percevoir (sa durée d'affichage, qui sera de 30 fois $\frac{9}{60000}$ de seconde par seconde devra émettre autant de lumière qu'un écran LCD, c'est-à-dire environ 500 cd $/m^2$)
- et ceci avec une consommation électrique suffisamment faible pour ne pas risquer de détruire les différents composants du dispositif.

Nous sommes arrivés à la conclusion que, pour atteindre de telles cadences, il était nécessaire d'avoir recours à la micro-électronique. Plusieurs solutions ont été envisagées : par exemple, nous avons pensé utiliser un écran LCD ferro-électrique, plus rapide que les écrans LCD "classiques", des micro-poutres pour réfléchir la lumière avec une intensité dépendant de l'intensité électrique qui lui était fournie, ou un dispositif mobile, muni d'une surface réfléchissante qui se déplacerait sous un cache afin de modifier la surface éclairée et donc l'intensité lumineuse réfléchie. Certaines de ces solutions se sont révélées

trop lentes, d'autres trop gourmandes en énergie en nécessitant une alimentation de plusieurs dizaines d'ampères pour afficher toute l'image, ce qui, dans le domaine de la micro-électronique pose des problèmes de dissipation d'énergie insurmontables actuellement et conduit, à terme, à la destruction pure et simple par la chaleur du dispositif miniature d'affichage.

La solution que nous avons finalement trouvée est une technologie émergente : les PHOLEDs. Le principe des PHOLEDS est montré (en coupe) sur la figure 3.17. PHOLED

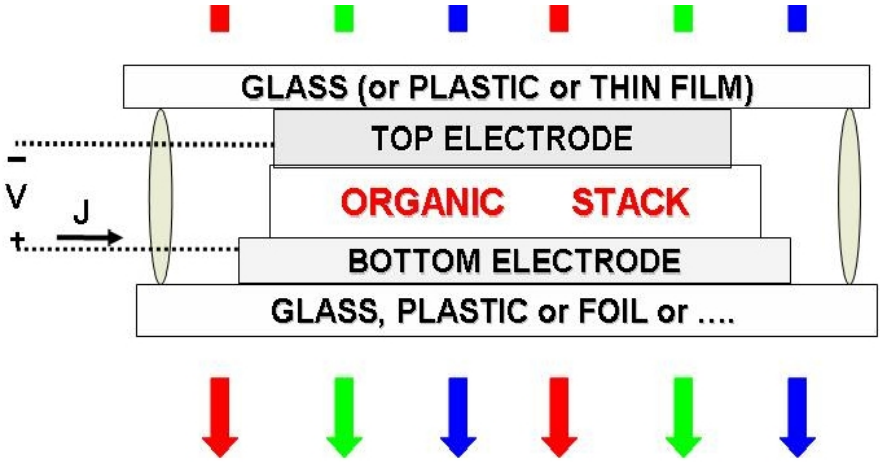


FIG. 3.17 – Principe des PHOLEDS (vue en coupe)

signifie PHosphorescent Organic Light Emitting Diode ou diode électro-luminescente organique phosphorescente. Cette technologie présente deux avantages :

- dans un premier temps, elle permet des temps de commutation (passage de l'état éteint à l'état allumé ou le contraire) inférieurs à la microseconde (10^{-6} seconde – voir figure 3.18), c'est-à-dire une vitesse suffisante pour opérer un balayage complet de la matrice d'affichage en moins de $\frac{1}{60000}$ de seconde.
- Le deuxième avantage est son rendement. En effet, contrairement aux autres types de LEDs (Light Emitting Diodes ou diodes électro-luminescentes) le rendement des PHOLEDs est proche de 100%, c'est-à-dire que près de 100% de l'énergie électrique fournie à la diode est transformée en lumière et pas en chaleur, lorsque l'intensité lumineuse de celle-ci ne dépasse pas 100 Cd/m^2 (voir figure 3.19). Donc

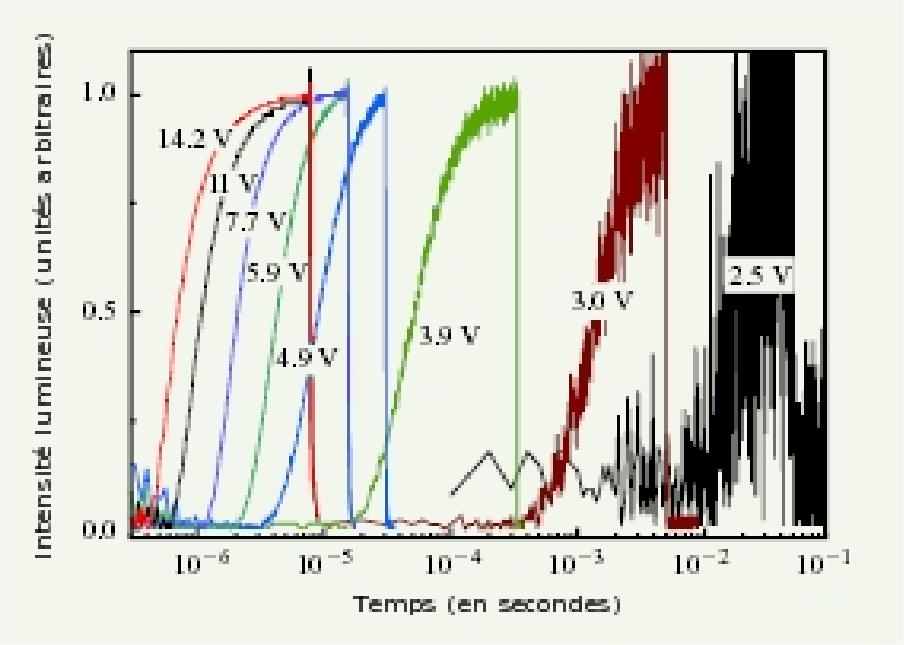


FIG. 3.18 – Variation de l'intensité lumineuse pour des impulsions de différentes amplitudes et longueurs

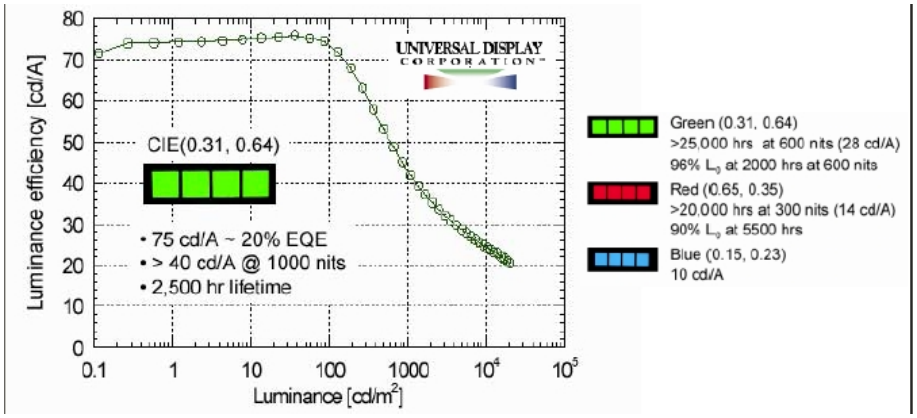


FIG. 3.19 – Variation du gain des PHOLEDs (en Candela/Ampère) en fonction de leur luminance

une déperdition d'énergie faible pour une luminosité plus importante. Ceci réduit les problèmes d'alimentation car l'affichage d'une image complète ne nécessite alors plus que quelques centaines de milli-ampères, ce qui reste acceptable pour ce genre de dispositif.

Ainsi, pour une intensité lumineuse de 500 Cd/m^2, l'intensité électrique nécessaire par une PHOLED est de 10^{-2} A/m^2. La taille approximative d'une PHOLED est

d'environ 20 μm $\times$ 20 μm = 4 $\times 10^{-6} cm^2$, le courant total nécessaire est alors de $4 \times 10^{-6} \times 10^6 = 4$ A sous 15 V, ce qui correspond à 60 Watts.

Reste ensuite le problème du contrôle de ces PHOLEDs. En effet, s'il est facile de contrôler des pixels sur un écran LCD car celui-ci se comporte comme un condensateur, il en va tout autrement des PHOLEDs. Celles-ci ont besoin d'être maintenues sous une certaine tension pour rester allumées. Vu la vitesse de commutation des PHOLEDs et la luminosité nécessaire, celles-ci doivent rester allumées le plus longtemps possible (au moins $\frac{9}{60000}$ de seconde) pour permettre un affichage suffisamment lumineux. Le balayage permettant de changer la luminosité des PHOLEDs doit être suivi d'une phase pendant laquelle elles restent allumées. Si le temps nécessaire au balayage est de $\frac{1}{60000}$ de seconde, le temps où les PHOLEDs doivent rester allumées doit être d'au moins $\frac{9}{60000}$ de seconde. Ainsi l'image perçue n'aura qu'une différence de luminosité de 10% entre le haut et le bas de l'image, ce qui est corrigeable électroniquement (en diminuant graduellement la tension électrique envoyée aux PHOLEDS afin que celles qui sont allumées plus longtemps soit moins lumineuses proportionnellement à leur durée d'allumage) ou logiciellement (en multipliant la valeur d'intensité lumineuse par une valeur inversement proportionnelle à sa durée d'allumage avant de la convertir en tension électrique), en modifiant graduellement la luminosité des pixels affichés pour corriger cette différence. Il est donc nécessaire de maintenir une tension aux bornes de chaque PHOLED et cette tension détermine la luminosité de chacune des PHOLEDS. La solution que nous avons proposée et vérifiée en construisant et testant le dispositif consiste donc à associer chaque PHOLED à un condensateur, permettant de mémoriser la tension de contrôle, associé à un transistor, permettant de fournir à la PHOLED le courant nécessaire à son fonctionnement avec la luminosité voulue, sans vider le condensateur. Nous avons réalisé une série de tests permettant de valider ce type de montage et vérifier qu'il fonctionne avec une LED, aux fréquences ($\frac{1}{60000000}$ de seconde pour le chargement de la mémoire, c'est-à-dire $\frac{1}{1000}$ème du temps nécessaire au balayage de la matrice d'affichage, puisque celle-ci a une résolution de 1 024 par 768, soit environ 1 000 lignes, et entre $\frac{9000}{60000000}$ et $\frac{9999}{60000000}$ de seconde d'affichage constant) dont nous avons besoin et que la luminosité de la LED est

bien conforme à celle voulue et proportionnelle à la tension appliquée en entrée du circuit, c'est-à-dire variant d'une LED noire à une intensité maximale proportionellement à la tension d'entrée entre une tension de 0V à 5V. Ces tests consistaient à mesurer la variation de tension et d'intensité aux bornes de la LED lorsqu'elle est soumise à des changements de luminosité avec différentes fréquences, et de vérifier qu'elles étaient bien constantes. Les PHOLEDs étant des LEDs particulières, mais qui conservent la plupart des caractéristiques des LEDs, et, en particulier, celle qui nous intéresse ici (variation de la luminosié en fonction de la tension et de l'intensité), ces tests sont valables en remplaçant la LED par une PHOLED.

3.5 Conclusion du chapitre

Nous avons décrit, dans ce chapitre, des tests réalisés pour valider notre dispositif, à la fois logiciellement (validation des algorithmes de codage et décodage des données 3D et de leur implémentation sur processeur spécialisé, le GPU, et sur architecture spécialisée) et physiquement, grâce à une étude de faisabilité du système d'affichage en lui-même. Au cours de cette thèse, d'autres études, tests et validations, en particulier mécaniques et optiques, ont été réalisés pour la réalisation de ce dispositif, mais nous ne pouvons les décrire ici, pour des raisons de confidentialité car un brevet est en cours de dépôt sur les parties sur lesquels ces études et tests portaient. Nous avons également décrit une partie des caractéristiques que doit avoir le dispositif d'affichage.

Les différents tests, études et simulations que nous avons décrits dans ce chapitre montrent que la réalisation d'un dispositif d'affichage permettant d'afficher plusieurs milliers d'images par seconde est possible avec la technologie actuelle et que les temps et fréquences ($\frac{1}{60000000}$ de seconde pour la mémorisation de la tension de contrôle, $\frac{1}{60000}$ de seconde pour parcourir la matrice d'affichage et $\frac{9}{60000}$ de seconde pour l'affichage des vues) sont envisageables. Cependant, même si d'autres études sont encore à mener pour valider l'ensemble du dispositif, les résultats obtenus laissent penser que ce dispositif est faisable avec cette technologie.

Bilan et perspectives

Cette thèse s'inscrit dans le cadre de la mise au point d'un dispositif d'affichage 3D innovant. Ce dispositif présente les avantages combinés de plusieurs méthodes d'affichage 3D, sans en avoir les inconvénients et doit donner lieu au dépôt d'un brevet. Parallèlement à l'étude de ce procédé, nous avons également étudié les algorithmes d'une chaîne de traitements destinée à générer les vues des objets affichés en un temps suffisamment court pour permettre son utilisation dans une application temps-réel, c'est-à-dire à une vitesse de 30 images 3D par seconde, ce qui correspond à 6 000 vues par seconde.

Synthèse

Le dispositif que nous avons proposé est fondé sur le principe de l'auto-stéréoscopie, et présente tous les avantages de ce principe :

- la possibilité d'observer les objets 3D opaques ou transparents,
- la possibilité d'afficher des scènes et objets plus grands que la taille réelle du dispositif ou placés en dehors de cette taille,
- la possibilité d'observer ces objets sans lunette, ni interaction entre l'observateur et le dispositif, comme un dispositif de "head-tracking"
- et la possibilité de modifier la position d'observation en se déplaçant simplement devant le dispositif.

De plus, ce principe permet, en plus des avantages apportés par l'utilisation de l'auto-stéréoscopie, de modifier la position d'observation en changeant la distance de l'observateur à l'écran, ce qui n'est pas possible avec les systèmes d'affichage auto-stéréoscopique actuels. En effet, en modifiant sa distance à l'écran, l'observateur va pouvoir modifier la

manière dont il perçoit la scène affichée. En particulier, il va modifier la largeur de son champ de vision. Ceci correspond à la perception réelle des objets telle qu'elle serait si les objets étaient réellement présents derrière l'écran et observés au travers d'un cadre.

Ce dispositif présente donc les avantages de plusieurs principes différents. En particulier, les avantages de l'auto-stéréoscopie dont nous venons de parler, le fait de pouvoir observer les objets depuis n'importe quelle distance, comme pour les systèmes volumiques et les hologrammes. Mais il permet aussi d'éviter plusieurs de leurs inconvénients. Par exemple, il ne nécessite pas que l'observateur porte des lunettes, comme les systèmes stéréoscopiques, il n'oblige pas à afficher des objets transparents, comme les afficheurs volumiques, il permet de changer la vue en fonction de la distance et ne réduit pas la taille des objets proportionnellement à la taille du support, comme les hologrammes. Et enfin, il ne limite pas la zone d'observation à une distance fixe, comme les afficheurs auto-stéréoscopiques.

Pour pouvoir piloter ce dispositif et lui permettre d'afficher correctement les objets voulus, nous avons également proposé un ensemble d'algorithmes permettant de générer les différentes vues sans être obligé de calculer chaque vue indépendamment, ce qui permet ainsi un gain de temps conséquent pour le calcul de ces vues.

Ces algorithmes ont été proposés pour résoudre un certain nombre de problèmes que ce type de dispositif présentait :

- Tout d'abord, le nombre de vues différentes à générer étant de 6 000 par seconde, la vitesse de projection des objets à afficher et de génération des différentes vues nécessaire pour atteindre ces caractéristiques était difficilement atteignable en calculant toutes les vues indépendamment.
- Ensuite, la taille des données nécessaires pour mémoriser les données pour l'affichage des 200 vues constituant une image 3D atteindrait 450 méga-octets, si les informations brutes de toutes les vues étaient mémorisées, ce qui est difficilement envisageable.
- Enfin, toutes les données correspondant aux 6 000 vues à afficher par seconde

doivent être envoyées au dispositif d'affichage en moins d'une seconde pour permettre l'affichage, ce qui représente 13 giga-octets par seconde. De telles débits sont difficilement atteignables.

Ces algorithmes, pouvant être utilisés dans d'autres cas de figure (systèmes d'affichage auto-stéréoscopiques "classiques", génération de sprites 3D, etc.), permettent également de compresser la taille des données générées afin de limiter les besoins de mémoire nécessaire pour les mémoriser, ainsi que le débit nécessaire pour les envoyer au dispositif d'affichage en un temps suffisamment court pour assurer son utilisation dans une application temps-réel, c'est-à-dire la génération et l'affichage d'une image 3D, et donc de toutes ses vues, en un trentième de seconde. De plus, certains de ces algorithmes permettent de coder de manière réaliste des effets tels que l'éclairage spéculaire, la réflexion et la réfraction.

L'adaptation de ces algorithmes sur une architecture spécialisée a également été étudiée. L'utilisation d'une telle architecture est nécessaire pour permettre d'exécuter tous ces algorithmes dans un temps suffisamment court pour l'utilisation de l'ensemble dans le cadre d'une application temps-réel et permettre d'afficher les 6 000 vues par seconde nécessaire au bon fonctionnement du dispositif.

La solution retenue et validée comporte un GPU, opérant les calculs de projection des objets à afficher, la génération des voxels et leur rassemblement en motifs, c'est-à-dire en blocs horizontaux de voxels contigus situés sur la même ligne d'affichage et à la même distance. Les algorithmes définis ont donc dû être adaptés pour tirer parti de l'architecture parallèle du GPU et de ses shaders, et pour s'adapter à ses limitations.

Les données fournies par le GPU sont ensuite envoyées à un pipe-line spécialisé opérant un tri et une mémorisation des motifs générés et des traitements additionnels afin de les coder dans une forme utilisant moins d'espace mémoire que la mémorisation brutes des différentes vues, mais permettant de reconstituer celles-ci sans altération. Une fois que tous les voxels formant une image 3D ont été mémorisés, cette architecture les remet en forme afin de les envoyer au dispositif d'affichage qui se charge de les afficher

correctement. Cette architecture spécialisée permet d'exécuter les algorithmes proposés à une vitesse compatible avec les contraintes temps-réel.

Nous avons également présenté une analyse des solutions techniques utilisables pour la réalisation du dispositif. Ces solutions passent par l'utilisation de diodes électro-luminescentes particulières (les PHOLEDs) pour un affichage des vues suffisamment rapidement pour permettre l'affichage des 6 000 vues par seconde, et par l'utilisation d'une mémoire analogique conçue spécialement pour ce projet et validée par l'expérience, pour le contrôle de ces diodes et permettre d'en obtenir une luminosité constante, afin d'afficher, pour chaque pixel des vues, toutes les nuances de couleur et de luminosité attendues d'un dispositif d'affichage de ce type.

Perspectives

Les tests que nous avons réalisés montrent que le principe d'affichage 3D que nous avons proposé fonctionne. Les tests que nous avons réalisés sur sa faisabilité laissent penser que sa réalisation est possible. Cependant, certains aspects de cette réalisation restent encore à valider et d'autres à définir. Par conséquent, il est nécessaire de réaliser d'autres tests.

Parmi ces points à éclaircir, le contrôle de l'affichage n'a pas été étudié. En effet, si la tension de sortie des mémoires analogiques a bien été validée, la vitesse de réaction et la luminosité des PHOLEDs n'ont pas été vérifiées. De plus, le contrôle des PHOLEDs qui doit convertir les données numériques, correspondant aux couleurs des pixels des vues, en tension de contrôle des PHOLEDs, à envoyer aux mémoires analogiques, n'a pas été étudié. Le dispositif optique permettant de projeter les rayons issus des vues dans la bonne direction (qui fait partie des éléments brevetables de notre dispositif) doit également être étudié plus en détail.

De futurs axes de recherche sur ces travaux peuvent porter sur l'amélioration des algorithmes de codage des données 3D. En particulier, dans l'algorithme de réduction

du nombre de voxels utilisés par des nuages de points, un travail important peut encore être réalisé dans la détection des zones à réduire (celles qui contiennent beaucoup de voxels qui ne sont pas effacés par l'algorithme d'effacement des voxels cachés, mais dont certains ne sont pas réellement visibles) sans influer sur des zones qu'il serait préférable de ne pas modifier. Une solution envisagée est de diviser la scène en blocs dont les arrêtes sont parallèles et perpendiculaires à l'écran et de comparer le nombre de voxels dans chacun de ces blocs. Les blocs ayant un relativement grand nombre de voxels verront leur nombre de voxels réduits et les autres ne seront pas modifiés.

D'autres axes de recherche concernent l'implémentation de nos traitements sur architecture spécialisée et sur GPU. Premièrement, la modélisation que nous avons faite de la chaîne de traitements ne comportait pas tous les traitements que nous avons décrits. En particulier, l'algorithme d'effacement des voxels cachés n'a pas été modélisé. Comme il est possible de traiter indépendamment tous les motifs d'une ligne en parallèle, l'utilisation du GPU semble prometteuse, surtout que celui-ci n'est pas utilisé à 100% par les tests déjà implémentés. La prochaîne étape dans l'élaboration de cette chaîne sera donc d'implémenter ces algorithmes et de les intégrer à la chaîne de traitements. Le choix de l'implémentation de ces traitements sur le GPU ou sur l'architecture spécialisée pourra être discutée et les adaptations des algorithmes et les choix d'implémentation devront être étudiés.

Les nouvelles versions de GPUs pourront aussi apporter des améliorations à l'implémentation de nos algorithmes en permettant de manipuler directement des données entières sur 32 bits au lieu de réels, pour les tailles et positions des motifs et les nombres de motifs et de plans dans les plans et lignes, respectivement, ce qui sera un avantage, puisque ces données sont forcément des entiers, codés comme tels lors du recodage final et n'ont donc aucun besoin d'être codés en réels. Cela permettra d'éviter certaines erreurs d'arrondis qui peuvent se produire lorsqu'on utilise des nombre réels à virgule flottante. De plus, l'introduction des *"geometry shaders"* apportera un nouvel étage programmable au GPU, en plus des pixel et voxel shaders, ce qui permettra de multiplier

les traitements effectués par le GPU. En particulier, nous pourrons y implémenter l'algorithme de récupération des motifs et un début de tri car les limitations sur le nombre et la position en mémoire des données générées, existant sur les pixel shaders n'existeront plus sur les futurs geometry shaders.

À terme, ce projet devrait déboucher sur la mise au point d'une chaîne de traitement des donnés 2D et 3D qui serait reconnue et utilisable par l'ordinateur hôte comme une carte graphique, permettant ainsi un affichage en relief temps-réel de n'importe quelle application 2D ou 3D. Cette carte pourra ensuite être implémentée sur FPGA ou ASIC et être testée en conditions réelles.

Bibliographie

[1] alioscopy web site. http ://www.alioscopy.com/.

[2] Ati developer : Tools - ashli - advanced shading language interface. http ://www.ati.com/developer/ashli.html.

[3] Autocad 2000 dxf reference.
http ://www.autodesk.com/techpubs/autocad/acad2000/dxf/.

[4] Autodesk - autocad. http ://www.autodesk.com/techpubs/autocad/.

[5] Autodesk home page. http ://www.autodesk.com.

[6] Cambridge university department of engineering three dimensional television research. http ://www2.eng.cam.ac.uk/~arlt1/research/research.htm.

[7] Holographic imaging, university of texas southwestern medical center at dallas. http ://innovation.swmed.edu/research/instrumentation/res_inst_dev3d.html.

[8] Microsoft hsls shaders.
http ://msdn.microsoft.com/library/en-us/directx9_c/HLSL_Shaders.asp.

[9] μpol technology web site. http ://www.vrex.com/products/micropol.shtml.

[10] Nvidia cg home page. http ://developer.nvidia.com/page/cg_main.html.

[11] Nvidia gpu programming guide. http ://download.nvidia.com/developer/GPU_Programming_Guide/GPU_Programming_Guide.pdf.

[12] Nvidia nvperfhud 4 home page.
http ://developer.nvidia.com/object/nvperfhud_home.html.

[13] Sh : A high-level metaprogramming language for modern gpus. http ://libsh.org.

[14] Summit design, inc. http ://www.summit-design.com/.

[15] Systemc community. http ://www.systemc.org/.

[16] Vesa releases displayport standard.
http ://www.vesa.org/press/VESA_dportlaunchpr_Final.htm.

[17] Vesa.org. http ://www.vesa.org/.

[18] Visual elite system design.
http ://www.summit-design.com/products/ve_system_design.html.

[19] Vrex stereoscopic 3d web site. http ://www.vrex.com.

[20] R. V. Kollarits A. Puri and B. G. Haskell. Basics of stereoscopic video, new compression results with mpeg-2 and a proposal for mpeg-4. *Signal Proc. Image Comm.*, 10 :201–234, 1997.

[21] J. R. Moore A. R. L. Travis, S. R. Lang and N. A. Dodgson. Time-manipulated 3-d video. Information Display 13(1), 1997. Jan.

[22] H. Aidinoglu and III M. H. Hayes. Compression of multi-view images. In *Proceedings of International Conference onImage Processing (ICIP)*, pages 385–389, 1994.

[23] H. Aidinoglu and III M. H. Hayes. Stereo image coding : A projection approach. In *IEEE transactions on Image Processing*, volume 7, pages 506–516, 1998.

[24] H. Aydinoglu and H. Hayes. Stereo image coding : A projection approach. In *IEEE Trans. Image Proc.*, volume 7, pages 506–516, April 1998.

[25] James F. Blinn and Martin E. Newell. Texture and reflection in computer generated images. In *Communications of the ACM*, volume 19, pages 542–547. ACM Press New York, NY, USA, October 1976.

[26] Sir David Brewster. *The Stereoscope*. 1856.

[27] Michael Bunnel. *GPU Gems 2*, chapter 14 (Dynamic Ambient Occlusion and Indirect Lighting), pages 223–233. Addison-Wesley, 2005.

[28] Edwin E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, Dept. of CS, U. of Utah, December 1974.

[29] Valerio Pascucci Chandrajit L. Bajaj and Guozhong Zhuang. Progressive compressive and transmission of arbitrary triangular meshes. In *Proceedings of the conference on Visualization '99 : celebrating ten years*, pages 307–316, October 1999. San Francisco, California, United States.

[30] G. C. Chang and W. N. Lie. Multi-view image compression and intermediate view synthetis for autostereoscopic applications. In *Proceedings of International Symposium on Circuits and Systems (ISCAS)*, number 2, pages 277–280, 2000. Geneva, Switzerland.

[31] M. Chow. Optimized geometry compression for real-time rendering. In AZ, editor, *IEEE Visualization'97*, pages 347–354, Oct 17-24 1997.

[32] Chromatek inc. web site. http ://www.chromatek.com.

[33] S. Puri D. Tarditi and J. Oglesby. Accelerator : simplified programming of graphics processing units for general-purpose uses via data-parallelism. Technical Report MSR-TR-2005-184, Microsoft Research, December 2005.

[34] Basil A. Omar Nocolas S. Holliman Jonathan Harrold David Ezra, Graham J. Woodgate and Larry S. Shapiro. New autostereoscopic display system. In *IS&T/SPIE Symposium on Electronic Imaging Science and Technology.*

[35] M. Deering. Geometry compression. In *Proc. ACM Siggraph'95*, pages 13–20. Computer Graphics, August 1995.

[36] Yuri N. Denisyuk. Photographic reconstruction of the optical properties of an object in its own scattered radiation field. In *Soviet Physics Doklady, 7*, pages 543–545, 1962.

[37] Yuri N. Denisyuk. On the reproduction of the optical properties of an object by the wave field of its own scattered radiation. In *Optics & Stereoscopy, 18*, pages 365–368, 1963.

[38] M. Denny and C. Sohler. Encoding a triangulation as a permutation of its point set. In *Proceedings of the Ninth Canadian Conference on Computational Geometry*, pages 39–43, Ontario, August 11-14.

[39] P. J. Diefenbach. *Pipeline Rendering : Interaction And Realism Through Hardware-Based Multi-Pass Rendering*. PhD thesis, University of Pennsylvania, 1996.

[40] N. A. Dodgson. Autostereo displays : 3d without glasses. EID '97 (Electronic Information Displays), 1997. 18th-20th Nov 1997, Esher, Surrey [invited paper].

[41] Daniel J. Sandin et. al. Computer-generated barrier-strip autostereography. In *Proc SPIE, Non-Holographic True 3D Display Technologies, 1083*, pages 65–75, Jan. 1989.

[42] C. Everitt. Interactive order-independant transparency. Technical report, NVidia, http ://developer.nvidia.com/object.Interactive_Order_Transparency.html, 2001.

[43] Randima Fernando, editor. *GPU Gems 2*. Addison-Wesley, 2005.

[44] T. Fuji and H. Harashima. Data compression of an autostereoscopic 3-d image. In *Proceedings of SPIE : Stereoscopic Displays and Virtual Reality Systems*, number 2177, pages 108–118, April 994.

[45] W. Horn G. Taubin, A. Gueziec and F. Lazarus. Progressive forest split compression. In *Proc. ACM Siggraph 98*, pages 123–132, July 1998.

[46] D. Gabor. A new microscopic principle. *Nature*, 161 :777–778, 1948.

[47] S. Gumhold and W. Strasser. Real time compression of triangle mesh connectivity. In *Proc. ACM Siggraph 98*, pages 133–140, July 1998.

[48] The gzip home page. http ://www.gzip.org/.

[49] W. D. Hillis and G. L. Steele Jr. Data parallel algorithms. *Communications of the ACM*, 29(12) :1170–1183, December 1986.

[50] Holografika web site. http ://www.holografika.com.

[51] H. Hoppe. Progressive meshes. In *Proceedings ACM SIGGRAPH'96*, pages 99–108, August 1996.

[52] D. Horn J. Sugerman K. Fatahalian M. Houston et al. I. Brook, T. Foley. Brook for gpus : Stream computing on graphics hardware. In *Proceedings of SIGGRAPH 2004*, 2004. Los Angeles, California.

[53] G. Guy I. Dinstein and J. Rabany. On stereo image coding. In *Proceedings of the 9th International Conference on Pattern Recognition*, number 1, pages 357–359, November 1988.

[54] A. Itai and M. Rodeh. Representations of graphs. In *Acta Informatica*, pages 215–219, 1982.

[55] Jeff P M Hultquist Jack Goldfeather and Henry Fuchs. Fast constructive-solid geometry display in the pixel-powers graphics system. In *International Conference on Computer Graphics and Interactive Techniques archive, Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 107 – 116. SIGGRAPH : ACM Special Interest Group on Computer Graphics and Interactive Techniques, ACM Press New York, NY, USA, 1986.

[56] F. Jargstorff. *GPU Gems*, chapter A Framework for Image Processing, pages 445–467. Addison-Wesley, 2004.

[57] Eitan Grinspun Jeffrey Bolz, Ian Farmer and Peter Schröder. Sparse matrix solvers on the gpu : Conjugate gradients and multigrid. In *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2003)*, pages 917–924, 2003.

[58] K. Keeler and J. Westbrook. Short encodings of planar graphs and maps. In *Discrete Applied Mathematics*, number 58, pages 239–252, 995.

[59] Titus Zaharia Khaled Mamou and Françoise Prêteuxt. Compression progressive de maillages 3d par approximation b-spline. In *Actes 10èmes Journées d'études et d'échanges Compression et Représentation des Signaux Audiovisuels (CORESA'2005), Rennes, France*, pages 183–188, November 2005.

[60] D. King and J. Rossignac. Guaranteed 3.67v bit encoding of planar triangle graphs. In *11th Canadian Conference on Computational Geometry, Vancouver, BC*, September 1999.

[61] Andreas Kolb and Nicolas Cuntz. Dynamic particle coupling for gpu-based fluid simulation. In *Proc. 18th Symposium on Simulation Technique*, pages 722–727, Sep. 2005.

[62] Didier Le Gall. Mpeg : a video compression standard for multimedia applications. *Commun. ACM*, 34(4) :46–58, 1991.

[63] E.N. Leith and J. Upatnieks. Photography by laser. *Scientific American*, 212 :24–35, 1965.

[64] J. Li and C.C Kuo. Progressive coding of 3d graphic models. In *IEEE Proc. Multimedia Computing and Systems*, pages 1052–1063, June 1998.

[65] G. Lippmann. La photographie des couleurs. In Gauthier-Villars, editor, *Conférence du 13 décembre 1891*, volume 1, page 12. ill. Mécaniques, 1893.

[66] G. Lippmann. Sur la théorie de la photographie des couleurs simples et composées par la méthode interférentielle. volume 1, page 11, 1894.

[67] G. Lippmann. La photographie integrale, comptes-rendus. volume 146, pages 446–451, 1908.

[68] M. E. Lukacs. Predictive coding of multi-viewpoint image sets. In *Proc. ICASSP*, pages 521–524, October 1986.

[69] N. Occhiogrosso M. Y. Kao and S. H. Teng. Simple and efficient compression schemes for dense and complement graphs. *Journal of Combinatorial Optimization*, 1999.

[70] J. I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *Proceedings of the 38th Annual IEEE Symposium on the Foundations of Computer Science*, pages 118–126, 1997.

[71] S. Muraki. Approximation and rendering of volume data using wavelet transforms. In *Proceedings of Visualization '92*, pages 21–28, October 1992.

[72] S. Muraki. Volume data and wavelet transforms. *IEEE Computer Graphics and Applications*, 13(4) :50–56, 1993.

[73] S. R. Lang G. Martin N. A. Dodgson, J. R. Moore and P. Canepa. A 50" time-multiplexed autostereoscopic display. In *SPIE 3957, SPIE Symposium on Stereoscopic Displays and Applications XI*, 23rd-28th Jan 2000, San Jose, California.

[74] M. Naor. Succinct representation of general unlabeled graphs. In *Discrete Applied Mathematics*, volume 29, pages 303–307, North Holland, 1990.

[75] Marc Nienhaus and Jürgen Döllner. *GPU Gems 2*, chapter 15 (Blueprint rendering and "Sketchy Drawings"), pages 235–252. Addison-Wesley, 2005.

[76] R. Pajarola and J. Rossignac. Compressed progressive meshes. *IEEE Transactions on Visualization and Computer Graphics*, 6(1) :79–93, 2000.

[77] M. G. Perkins. Data compression of stereo pairs. *IEEE Trans. Comm.*, 40 :684–696, April 1992.

[78] K. Perlin. *GPU Gems*, chapter Implementing improved Perlin noise, pages 73–85. Addison-Wesley, 2004.

[79] Pierre Poulin and Alain Fournier. A model for anisotropic reflection. In F. Baskett, editor, *International Conference on Computer Graphics and Interactive Techniques - Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, volume 24, pages 273–282, august 1990.

[80] X. He M. Y. Kao R. C. Chuang, A. Garg and H. I. Lu. Compact encodings of planar graphs via canonical orderings and multiple parentheses. In *Proceedings of the 25th International Colloquium on Automata, Languages, and Programming*, pages 118–129, 1998.

[81] P.N. Tudor (BBC R&D). Mpeg-2 video compression. *IEE Electronics & Communications Engineering Journal*, pages 257–264, December 1995.

[82] J. Rossignac. Edgebreaker : Connectivity compression for triangle meshes. In *IEEE Transactions on Visualization and Computer Graphics*, volume 5, January-March 1999.

[83] J. Rossignac and A. Szymczak. Wrap & zip : Linear decoding of planar triangle graphs. Technical Report Tech Report : GIT-GVU-99-08, GVU Center, Georgia Institute of Technology, http ://www.cc.gatech.edu/ jarek/abstracts.html, January 1999.

[84] T. Saito and T. Takahashi. Comprehensible rendering of 3-d shapes. *Computer Graphics*, 24(4) :187–206, 1990.

[85] Gee-bum Koo Sanghun Park and Insung Ihm. Wavelet-based 3d compression schemes for the visible human data set and their applications. In *The Second Visible Human Project Conference Proceedings*, October 1 & 2.

[86] D. Shah and N. A. Dodgson. Issues in multi-view autostereoscopic image compression. Proc SPIE 4297, "SPIE Symposium on Stereoscopic Displays and Applications XII", 22nd-25th Jan 2001, San Jose, California, pp. 307-316, ISSN 0277-786X, 2001.

[87] Trond Runar Hagen Sverre Briseid, Tor Dokken and Jens Olav Nygaard. Spline surface intersections optimized for gpus. In Peter M.A. Sloot Vassil N. Alexandrov, Geert Dick van Albada and Jack Dongarra, editors, *Computational Science – ICCS 2006*, volume 3994 of *LNCS*, pages 204–211. Springer, 2006.

[88] M. Kaneko T. Naemura and H Harashima. 3-d object-based coding of multiviewimags. In *Proceedings of Picture Coding Symposium of Japan (PCSJ)*, 1995.

[89] D. Taubman and M. Marcellin. *JPEG2000 : Image Compression Fundamentals, Standards and Practice*. Kluwer Academic Publishers, Dordrecht, 2001.

[90] Toutin Th. Qualitative aspects of chromo-stereoscopy for depth perception. In *Photogrammetric Engineering & Remote Sensing*, volume 63, pages 193–203, 1997.

[91] C. Touma and C. Gotsman. Triangle mesh compression. In *Proceedings Graphics Interface 98*, pages 26–34, 1998.

[92] Knut-Andreas Lie Trond Runar Hagen and Jostein R. Natvig. Solving the euler equations on graphics processing units. In Peter M.A. Sloot Vassil N. Alexandrov, Geert Dick van Albada and Jack Dongarra, editors, *Computational Science – ICCS 2006*, volume 3994 of *LNCS*, pages 220–227. Springer, 2006.

[93] G. Turan. Succinct representations of graphs. In *Discrete Applied Math*, number 8, pages 289–294, 1984.

[94] W. Tutte. A census of planar triangulations. *Canadian Journal of Mathematics*, (14) :21–38, 1962.

[95] W. Tutte. The enumerative theory of planar graphs. In J.N. Srinivasan et al., editor, *A Survey of Computational Theory*. North-Holland, 1973.

[96] J. Y. A. Wang and E. H. Adelson. Representing moving images with layers. In *The IEEE Transactions on Image Processing Special Issue : mage Sequence Compression*, number 3, pages 625–638, September 1994.

[97] Texas instrument ltd. web site. http ://www.ti.com.

[98] Texas instrument's dlp^{TM}technology web site. http ://www.dlp.com.

[99] Sean Whalen. Audio and the graphics processing unit. In *IEEE Vis 2004 GPGPU Tutorial*, 2004.

[100] Charles Wheatstone. Contributions to the physiology of vision. part 1. on some remarkable, and hitherto unobserved, phenomena of binocular vision. pages 371–394. Royal Society of London Philosophical Transactions 128, 1838.

[101] Charles Wheatstone. Contributions to the physiology of vision. part 2. on some remarkable, and hitherto unobserved, phenomena of binocular vision (continued). *The London, Edinburgh, and Dublin Philisophical Magazine and Journal of Science, series 4, 3*, pages 504–523, 1838.

[102] M. Y. Kao X. He and H. I. Lu. Linear-time succinct encodings of planar graphs via canonical orderings. *SIAM Journal on Discrete Mathematics*, 1999.

[103] A. Zhirkov D. Timasov A. Konushin L. Levkovich-Maslyuk M. Han Y. Bayakovski, A. Ignatenko and I. K. Park. Depth image-based representations and compression for static and animated 3d objects. In *Proceedings of IEEE International Conference on Image Processing (ICIP)*, number 3, pages 25–28, September 2002.

[104] John Johnson Yang Liu, Wayne Huang and Sheila Vaidya. Gpu accelerated smith-waterman. In Peter M.A. Sloot Vassil N. Alexandrov, Geert Dick van Albada and Jack Dongarra, editors, *Computational Science – ICCS 2006*, volume 3994 of *LNCS*, pages 188–195. Springer, 2006.

Publications

1. Benoit Kaufmann & Mohamed Akil. Video memory compression for multi-view auto-steroscopic displays. *IS&T/SPIE 16th Annual Symposium Electronic Imaging : Science and Technology*
2. Benoit Kaufmann & Mohamed Akil. New 3D display : Algorithms and real time architecture. *17ème Congrès Mondial IMACS Calcul Scientifique, Mathématiques Appliquées et Simulation*
3. Benoit Kaufmann & Mohamed Akil. Real time hardware for a new 3D display. *IS&T/SPIE 16th Annual Symposium Electronic Imaging : Science and Technology*
4. Benoit Kaufmann & Mohamed Akil. 3D images compression for multi-view autostereoscopic displays. *Proc. of Computer Graphics, Imaging and Vision (CGIV06)*
5. Benoit Kaufmann, Mohamed Akil & Jean-Paul Sov. Multiview autostereoscopic display : Real-time implementation of a 3D image compression algorithm on GPU. En cours de préparation pour soumission au journal *IEEE Transactions on Visualization & Computer Graphics*.

Annexes

Annexe A

Programme de reconstitution des vues

Ce code source (écrit en JAVA, plus simple que la version en C, car elle contient moins de tests de dépassement. Ces deux versions implémentent cependant exactement le même algorithme.) correspond aux deux fonctions permettant de décoder les données fournies par nos algorithmes de codage des données 3D, c'est-à-dire les algorithmes n°2 (Mise en forme des données), n°5 (Mise en forme des données avec gestion des sous-scènes), n°6 (Mise en forme des données des sous-scènes) et n°7 (Codage d'une scène contenant des sous-scènes). Elles implémentent l'algorithme n°8 (Reconstitution des vues avec gestion des scènes secondaires). La première, **calcImage**, est appelée avec les coordonnées (par rapport au dispositif d'affichage) du point d'observation pour lequel elle doit calculer la vue. Elle calcule donc l'angle correspondant à ce point d'observation et appelle, pour chaque ligne de la vue à générer, la deuxième fonction **calcLigne** qui va décoder la ligne indiquée et l'écrire à sa place dans la mémoire devant mémoriser la vue résultat. La classe dont ces deux fonctions font partie comporte également, entre autres, les champs suivants :

- **fond_image** est la couleur de fond de la scène
- **pixelsArray** est le tableau devant contenir la vue résultat
- **screenWidth** et **screenHeight** sont respectivement la largeur et la hauteur (en pixels) de la vue à générer
- **frameBuffer** est le tableau contenant les données à décoder pour générer les vues

Sources du programme de reconstitution des vues

```
/**
 * Calcul une vue correspondant à un des yeux de l'observateur
 @param x Position x du point d'observation par rapport à l'écran
 @param y Position y du point d'observation par rapport à l'écran
*/

private void calcImage(int x, int y) {

  int adresse = 0;

  // Initialisation de l'image
  for ( int i = 0; i < screenWidth * screenHeight; i++ )
    pixelsArray[i] = 0xff000000 | fond_image;

    // Vérification de sortie de la zone d'observation
    if ( y <= 0 )  // Observateur derrière l'écran
      return;

    if ( ((double)x/y)*((double)x/y) > 0.76 )
    // 0.76 = (0.87)^2 = ( sin(60 deg.) ) ^ 2
      return; // Observateur hors de la zone de projection

    ////////////////////////////////////////////////
    // Discrétisation des angles de projection
    ////////////////////////////////////////////////
    double scale = (double)x/y;

    ////////////////////////////////////////////////
    // Calcul de chaque ligne de la vue
    ////////////////////////////////////////////////
    for ( int ligne = 0; ligne < loadedLines; ligne++ )
      adresse = calcLigne(adresse, scale, ligne * screenWidth);

}



////////////////////////////////////////////////////////////////////

/**
```

```
43    * Dessine une ligne de l'écran dans la fenêtre d'affichage
44    @param adresse Adresse dans le frame buffer où commencer la lecture
45    @param scale Mise à l'échelle en fonction de l'angle de vision à rendre
46    @param adresse_debut_dessin Adresse dans la mémoire image où écrire la
47    * ligne décodée
48    */
49
50  public int calcLigne(int adresse, double scale, int adresse_debut_dessin) {
51
52    int nPlans = frameBuffer[adresse++];              // Nombre de plans
53
54    for ( int plan = 0; plan < nPlans; plan++ ) {
55
56      double offsetPlan = frameBuffer[adresse++]; // Offset du plan
57
58      // Le décalage est égal à
59      //   D * tan a / ( D + d )   // [ d * ( D - d ) * tan a ] / D
60      // = D * (x/y) / ( D + d )   // [ d * ( D - d ) * (x/y) ] / D
61      // où d = distance point d'observation - écran virtuel
62      //    D = distance point d'observation - voxel
63      //    -> offset du plan = D - d
64      //    a = angle d'observation (ou de projection)
65      //    (x, y) = position de l'observateur
66
67      int decal = -(int)(scale * offsetPlan );
68      int nMotifs = frameBuffer[adresse++];      // Nombre de motifs
69
70      if ( nMotifs > 0 ) {
71
72        for ( int motif = 0; motif < nMotifs; motif++ ) {
73
74          int offsetMotif = frameBuffer[adresse++] + decal; // Offset du motif
75
76          int adresse_debut_motif = adresse_debut_dessin + offsetMotif;
77          // adresse du début du motif dans l'image
78
79          int nPixels = frameBuffer[adresse++]; // Nombre de pixels
80
81          if ( (nPixels & 0x80000000) != 0 ) {
82            // Ce motif contient une sous-scène
83
84            int indice_sub_frame_buf = frameBuffer[adresse++];
85
86            // on ignore les données de décalage pour l'instant
```

```
87            adresse++;
88
89            // calcul de la taille réelle du motif
90            nPixels = nPixels & 0x7fffffff;
91
92            if ( decode_subFB ) {
93
94              int [ ] sub_pixelsArray = new int [screenWidth];
95
96              // on le remplit avec la couleur de fond
97              for ( int pixel = 0; pixel < nPixels; pixel++ )
98                sub_pixelsArray[pixel] = frameBuffer[adresse+pixel];
99
100             // on passe au début de la ligne
101             indice_sub_frame_buf++;
102
103             // On calcule la ligne de la sous-scène
104             int [ ] oldpixelsArray = pixelsArray;
105             pixelsArray = sub_pixelsArray;
106             calcLigne(indice_sub_frame_buf, scale, 0);
107             pixelsArray = oldpixelsArray;
108
109             for ( int pixel = 0; pixel < nPixels; pixel++ ) {
110
111               if ( ((pixel + offsetMotif) >= 0) &&
112                    ((pixel + offsetMotif) < screenWidth)) {
113
114                 // Si le pixel est visible (clipping)
115
116                 pixelsArray[pixel + adresse_debut_motif] =
117                   (255 << 24) | sub_pixelsArray[pixel];
118
119                 adresse++;
120               }
121
122             }
123
124           } else { // if ( ! decode_subFB )
125
126             for ( int pixel = 0; pixel < nPixels; pixel++ )
127               pixelsArray[pixel + adresse_debut_motif] =
128                 frameBuffer[adresse++];
129
130           }
```

```
131
132         } else {
133           // Ce motif ne contient pas une sous-scène
134
135           for ( int pixel = 0; pixel < nPixels; pixel++ )
136
137           if ( ((pixel + offsetMotif) >= 0) &&
138                ((pixel + offsetMotif) < screenWidth)) {
139
140             // Si le pixel est visible (cliping)
141
142             int alpha = frameBuffer[adresse] >> 24;
143             if ( alpha < 0 )
144               alpha += 256;
145
146             if ( alpha == 255 )
147               pixelsArray[pixel + adresse_debut_motif] =
148                 (255 << 24) | frameBuffer[adresse++];
149             else
150               {
151                 int position_pixel = pixel + adresse_debut_motif;
152                 // adresse du pixel dans l'image
153
154                 int red = ( ( ( ( pixelsArray[position_pixel] &
155                                   0xff0000 ) >> 16 ) * (255 - alpha) ) +
156                               ( ( (frameBuffer[adresse] & 0xff0000) >>
157                                   16 ) * alpha) ) >> 8;
158
159                 int green = ( ( ( ( pixelsArray[position_pixel] &
160                                     0xff00 ) >> 8 ) * (255 - alpha) ) +
161                                 ( ( (frameBuffer[adresse] & 0xff00) >>
162                                     8 ) * alpha) ) >> 8;
163
164                 int blue = ( ( ( pixelsArray[position_pixel] & 0xff )
165                                  * (255 - alpha) ) +
166                                ( (frameBuffer[adresse] & 0xff) * alpha)
167                                ) >> 8;
168
169                 pixelsArray[pixel + adresse_debut_motif] =
170                   (255 << 24) | (red << 16) | (green << 8) | blue;
171
172                 adresse++;
173               }
174           }
```

```
175             else
176               adresse++;
177           } // Ce motif ne contient pas une sous-scène
178         }
179       }
180     }
181
182     return adresse;
183
184 }
```

Annexe B

Sources du programme de test de l'algorithme de relocation

Ce programme a été écrit pour tester l'algorithme de relocation, c'est-à-dire le placement des scènes 3D dans des fenêtres d'un environnement 2D. Ce programme prend comme paramètres :

- le nom du fichier dans lequel écrire le résultat,
- un fichier image (au format PPM) correspondant à l'environnement 2D,
- un fichier "cache" indiquant quels pixels font partie des fenêtres (affichant les sous-scènes) et quels pixels correspondent à la partie 2D
- et, pour chaque sous-scène 3D,
  - le fichier contenant les données de la sous-scène
  - les coordonnées x et y du coin supérieur gauche de la fenêtre (sur l'image 2D) devant contenir et afficher cette sous-scène et
  - les dimensions (largeur et hauteur, en pixels) de cette fenêtre

Il comporte trois fonctions :

- **charge_subFB**(char * filename, unsigned char ** pointer, unsigned long * taille, unsigned long * hauteur, unsigned long ** indices_debut_lignes) lit le fichier ayant le nom *filename* et contenant une sous-scène (générée par l'algorithme n°2 (Mise en forme des données) ou par l'algorithme n°5 (Mise en forme des données avec gestion des sous-scènes) précédée d'une en-tête). Alloue une zone mémoire de la taille des données de ce fichier et y écrit les données du fichier, en les décodant

pour pouvoir repérer d'éventuelles erreurs et pour pouvoir construire un tableau contenant l'indice (dans la zone mémoire servant à recevoir les données) du début de chaque ligne. Cette zone mémoire est ensuite fournie à la fonction **main** par l'intermédiaire du pointeur *unsigned char ** pointer*. Le tableau des indices de début de lignes est retourné par l'intermédiaire du pointeur *unsigned long ** indices_debut_lignes*, ainsi que la taille des données (en nombre d'éléments qui sont sur 32 bits donc 4 octets) et les dimensions de l'écran ou la fenêtre pour lequel ce fichier a été généré.

- la fonction **recode** sert à changer le boutisme, ou "endianness" en anglais, c'est-à-dire l'ordre dans lequel les octets sont stockés en mémoire, d'un entier codé sur 32 bits, car le boutisme des machines que nous avons utilisées pour les tests (des PCs sous Linux) ne correspond pas à l'ordre que nous avons choisi pour nos fichiers, c'est-à-dire l'octet de poids fort en premier (MSB, ou Most Signifiant Byte, first, en anglais).
- la fonction **main** s'occupe de vérifier le nombre des paramètres, puis charge les différents fichiers et enfin regénère le fichier final.

Le chargement des fichiers images se fait grace à une petite librairie que nous avons écrite, mais que nous n'avons pas montrée ici.

Un exemple d'utilisation de ce programme est montré après son listing. Il se compose, en figure B.1, de l'image représentant l'environnement 2D, du cache utilisé pour spécifier quels pixels contiennent les sous-scènes et quels pixels contiennent les pixels de la partie 2D, représenté figure B.2 et des deux scènes à insérer, représentées par les figures B.3 et B.4. Le résultat final est représenté figure B.5.

B.1 Listing du programme

Sources du programme de test de l'algorithme de relocation

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/types.h>
4  #include <sys/stat.h>
5  #include <unistd.h>
6  #include "load_image.h"        /* Fonctions de gestion des fichiers image */
7
8
9  /* Fonction de chargement d'une sous-scène */
10 int charge_subFB(char          *  filename,
11                  unsigned char ** pointer,
12                  unsigned long *  taille,
13                  unsigned long *  hauteur,
14                  unsigned long ** indices_debut_lignes);
15
16 /* Fonction permettant de modifier l'endianness d'un entier de 32 bits */
17 unsigned long recode(unsigned long val);
18
19
20 /*****************************************************************/
21
22
23 int main(int argc, char ** argv) {
24
25   FILE          *  out = NULL;
26   image             image2D;
27   image             cache;
28   unsigned int      ligne, pixel;
29   unsigned long *   frame_buffer = NULL;
30   unsigned long     adresse = 0;
31   unsigned char **  subFB = NULL;
32   unsigned long *   taille_subFB = NULL;
33   unsigned long *   hauteur_subFB = NULL;
34   unsigned long     n_subFB = 0;
35   unsigned long **  indices_debut_lignes_subFB = NULL;
36   unsigned long *   x, * y, * l, * h;
37   unsigned long     i, j, next_SFB, tot_SFB, n_mot, last_x, largeur;
38   unsigned long     nMotifs, motif, nVoxels, voxel;
```

```
39    unsigned long    profondeur = 0;
40
41
42    /* Verification du nombre de parametres */
43    if ( argc < 3 ) {
44      printf("usage : %s fichier_sortie fichier_PNG ", argv[0]);
45      printf("[fichier_cache [sub_FB x y largeur hauteur [sub_FB ...]]]\n);
46      return 1;
47    }
48
49
50    /* chargement de l'image 2D */
51    if ( litImage(argv[2], &image2D) == 0 ) {
52      fprintf(stderr, "erreur de lecture du fichier %s.\n, argv[2]);
53      return 1;
54    }
55
56
57    /* Si on a des sous-scènes */
58    if ( (argc - 4) > 0 ) {
59
60      if ( (argc - 4) % 5 != 0 ) {
61        fprintf(stderr, "Nombre de paramètres incorrect!\n);
62        return 1;
63      }
64
65      /* Allocation des tableaux nécessaires */
66      n_subFB = (argc - 4) / 5;
67      subFB = (unsigned char **)malloc(n_subFB * sizeof(char *));
68      if ( subFB == NULL ) {
69        fprintf(stderr, "Erreur d'allocation de mémoire.\n);
70        return 1;
71      }
72
73      /* tailles (en octets) des sous-scènes */
74      taille_subFB = (unsigned long *)malloc(n_subFB *
75                                             sizeof(unsigned long));
76      if ( taille_subFB == NULL ) {
77        fprintf(stderr, "Erreur d'allocation de mémoire.\n);
78        return 1;
79      }
80
81      /* nombre de lignes des sous-scènes */
82      hauteur_subFB = (unsigned long *)malloc(n_subFB *
```

```
83                                          izeof(unsigned long));
84      if ( hauteur_subFB == NULL ) {
85        fprintf(stderr, "Erreur d'allocation de mémoire.\n);
86        return 1;
87      }
88
89      /* indices du début des lignes des sous-scènes */
90      indices_debut_lignes_subFB =
91               (unsigned long **)malloc(n_subFB * sizeof(unsigned long *));
92      if ( indices_debut_lignes_subFB == NULL ) {
93        fprintf(stderr, "Erreur d'allocation de mémoire.\n);
94        return 1;
95      }
96
97      /* dimensions et positions des fenêtres des sous-scènes */
98      x = (unsigned long *)malloc(n_subFB * sizeof(unsigned long));
99      if ( x == NULL ) {
100       fprintf(stderr, "Erreur d'allocation de mémoire.\n);
101       return 1;
102     }
103
104     y = (unsigned long *)malloc(n_subFB * sizeof(unsigned long));
105     if ( y == NULL ) {
106       fprintf(stderr, "Erreur d'allocation de mémoire.\n);
107       return 1;
108     }
109
110     l = (unsigned long *)malloc(n_subFB * sizeof(unsigned long));
111     if ( l == NULL ) {
112       fprintf(stderr, "Erreur d'allocation de mémoire.\n);
113       return 1;
114     }
115
116     h = (unsigned long *)malloc(n_subFB * sizeof(unsigned long));
117     if ( h == NULL ) {
118       fprintf(stderr, "Erreur d'allocation de mémoire.\n);
119       return 1;
120     }
121
122     /* Chargement des sous-scènes */
123     /* Ici, la variable pixel compte les sous-scènes */
124     for ( pixel = 0; pixel < n_subFB; pixel++ ) {
125
126       if ( charge_subFB(argv[pixel * 5 + 4],
```

```
127                         &(subFB[pixel]),
128                         &(taille_subFB[pixel]),
129                         &(hauteur_subFB[pixel]),
130                         &(indices_debut_lignes_subFB[pixel]) ) == 0 ) {
131         fprintf(stderr, "Erreur de chargement du fichier %s\n,
132                 argv[pixel * 5 + 4]);
133         return 1;
134       }
135
136       /* mémorisation des positions et tailles des */
137       /* fenêtres des sous-scènes */
138       x[pixel] = atoi(argv[pixel * 5 + 5]);
139       y[pixel] = atoi(argv[pixel * 5 + 6]);
140       l[pixel] = atoi(argv[pixel * 5 + 7]);
141       h[pixel] = atoi(argv[pixel * 5 + 8]);
142
143     }
144
145   } /* if ( (argc - 3) > 0 ) */
146
147
148   /* chargement du cache */
149   if ( argc > 3 ) {
150     if ( litImage(argv[3], &cache) == 0 ) {
151       fprintf(stderr, "erreur de lecture du fichier %s.\n, argv[3]);
152       return 1;
153     }
154     monochrome(&cache);
155   }
156
157
158   /* Allocation de la memoire pour calculer le résultat */
159   frame_buffer =
160     (unsigned long *)malloc( ( (image2D.largeur + 5 + 6*n_subFB) *
161                                image2D.hauteur ) *
162                              sizeof(unsigned long) );
163   if ( frame_buffer == NULL ) {
164     fprintf(stderr, "Erreur d'allocation de mémoire.\n);
165     return 1;
166   }
167
168
169   /* calcul du résultat */
170
```

```
171   adresse = 0;
172
173   frame_buffer[adresse++] = 0x0000ff; /* couleur de fond */
174
175   if ( image2D.n_plans == 3 ) {  /* image couleur */
176
177     for ( ligne = 0; ligne < image2D.hauteur; ligne++ ) {
178
179       /* on cherche la première fenêtre contenant une sous-scène */
180       tot_SFB = 0;
181       next_SFB = -1;
182       for ( i = 0; i < n_subFB; i++ )
183         if ( y[i] <= ligne && y[i] + h[i] > ligne ) {
184           if ( next_SFB == -1 || x[i] <= x[next_SFB] )
185             next_SFB = i;
186           tot_SFB++;
187         }
188
189
190       if ( tot_SFB == 0 ) { /* pas de sous-scène dans cette ligne */
191
192         frame_buffer[adresse++] = 1;                /* un plan dans la ligne */
193         frame_buffer[adresse++] = profondeur;       /* profondeur du plan */
194         frame_buffer[adresse++] = 1;                /* un motif dans le plan */
195         frame_buffer[adresse++] = 0;                /* position du motif */
196         frame_buffer[adresse++] = image2D.largeur; /* taille du motif */
197
198         for ( pixel = 0; pixel < image2D.largeur; pixel++ )
199           frame_buffer[adresse++] = 0xff000000 +
200             ( (image2D.data[0][ligne*image2D.largeur + pixel] & 0xff)
201                << 16 ) +
202             ( (image2D.data[1][ligne*image2D.largeur + pixel] & 0xff)
203                << 8  ) +
204             (image2D.data[2][ligne*image2D.largeur + pixel] & 0xff);
205
206       } else { /* il y a une sous-scène dans cette ligne */
207
208           frame_buffer[adresse++] = 1;              /* un plan dans la ligne */
209           frame_buffer[adresse++] = profondeur;  /* profondeur du plan    */
210           n_mot = adresse;           /* on mémorise la position du nombre */
211                                  /* de motifs pour pouvoir la modifier ensuite */
212           frame_buffer[adresse++] = 1;              /* un motif dans le plan */
213
214           if ( x[next_SFB] != 0 ||
```

```
215                cache.data[0][image2D.largeur * ligne] >= 128) {
216
217              frame_buffer[adresse++] = 0;              /* position du motif */
218              frame_buffer[adresse++] = x[next_SFB];    /* taille du motif   */
219
220              largeur = frame_buffer[adresse - 1];
221
222              for ( pixel = 0; pixel < largeur; pixel++ ) {
223                frame_buffer[adresse++] = 0xff000000 +
224                  ((image2D.data[0][ligne*image2D.largeur+pixel]&0xff)
225                     << 16 ) +
226                  ((image2D.data[1][ligne*image2D.largeur + pixel]&0xff)
227                     << 8 ) +
228                  (image2D.data[2][ligne*image2D.largeur + pixel] & 0xff);
229              }
230
231              last_x = largeur;
232
233            } else
234              last_x = 0;
235
236
237            while ( last_x < image2D.largeur ) {
238
239              /* codage du motif next_SFB */
240
241              /* il y a un motif de plus */
242              frame_buffer[n_mot]++;
243
244              /* position du motif */
245              frame_buffer[adresse++] = last_x;
246
247              /* taille du motif */
248              frame_buffer[adresse++] = x[next_SFB] + l[next_SFB] - last_x;
249              /* verification du cache */
250              for ( i = x[next_SFB] + l[next_SFB] - 1;
251                    i >= last_x &&
252                      cache.data[0][image2D.largeur * ligne + i] >= 128;
253                    i-- ) ;
254              frame_buffer[adresse - 1] = i - last_x + 1;
255              /* on verifie si une autre sous-scène ne le masque pas */
256              for ( i = next_SFB + 1; i < tot_SFB; i ++ ) {
257                if ( y[i] <= ligne && y[i] + h[i] > ligne &&
258                     x[i] > x[next_SFB] &&
```

```
259                 x[i] < x[next_SFB] + frame_buffer[adresse - 1] ) {
260               frame_buffer[adresse - 1] = x[i] - x[next_SFB];
261             }
262           }
263           largeur = frame_buffer[adresse - 1] + last_x;
264           frame_buffer[adresse - 1] =
265               frame_buffer[adresse - 1] | 0x80000000;
266           frame_buffer[adresse++] = next_SFB;
267           /* adresse du sub frame buf */
268           frame_buffer[adresse++] = 0; /* modifieur de rendu */
269
270           for ( pixel = last_x; pixel < largeur; pixel++ ) {
271             frame_buffer[adresse++] = 0xff000000 +
272               ((image2D.data[0][ligne*image2D.largeur+pixel]&0xff)
273                  << 16 ) +
274               ((image2D.data[1][ligne*image2D.largeur + pixel]&0xff)
275                  << 8 ) +
276               (image2D.data[2][ligne*image2D.largeur + pixel] & 0xff);
277           }
278
279           last_x = largeur;
280
281           if ( last_x < image2D.largeur ) {
282             /* il reste de la place a droite (donc des pixels a coder) */
283
284             /* on verifie le cache */
285             for ( i = last_x + 1;
286                   i < image2D.largeur &&
287                   cache.data[0][ligne*image2D.largeur + i] >= 128;
288                   i++ ) ;
289
290             if ( i == image2D.largeur ) {
291               /* PAS de nouveau sous frame buffer à cause du cache */
292
293               /* il y a un motif de plus */
294               frame_buffer[n_mot]++;
295
296               /* position du motif */
297               frame_buffer[adresse++] = last_x;
298
299               /* taille du motif  */
300               frame_buffer[adresse++] = image2D.largeur - last_x;
301
302               for ( i = last_x; i < image2D.largeur; i++ )
```

```
303                 frame_buffer[adresse++] = 0xff000000 +
304                   ((image2D.data[0][ligne*image2D.largeur+i]&0xff)
305                     <<16) +
306                   ((image2D.data[1][ligne*image2D.largeur + i]&0xff)
307                     <<8)+
308                   (image2D.data[2][ligne*image2D.largeur + i] & 0xff);
309
310               last_x = image2D.largeur;
311
312             } else { /* il y a un autre motif */
313
314               /* on passe au motif suivant */
315               next_SFB = -1;
316               for (i = 0 ; i < n_subFB; i++ )
317                 if ( y[i] <= ligne && y[i] + h[i] > ligne &&
318                      x[i] + l[i] > last_x )
319                   next_SFB = i;
320
321               if ( next_SFB == -1 ) {
322
323                 /* il n'y a plus de sous frame buffer */
324
325                 /* il y a un motif de plus */
326                 frame_buffer[n_mot]++;
327
328                 /* position du motif */
329                 frame_buffer[adresse++] = last_x;
330
331                 /* taille du motif  */
332                 frame_buffer[adresse++] = image2D.largeur - last_x;
333
334                 for ( i = last_x; i < image2D.largeur; i++ )
335                   frame_buffer[adresse++] = 0xff000000 +
336                     ((image2D.data[0][ligne*image2D.largeur+i]&0xff)
337                       <<16) +
338                     ((image2D.data[1][ligne*image2D.largeur + i]&0xff)
339                       <<8)+
340                     (image2D.data[2][ligne*image2D.largeur + i] & 0xff);
341
342                 last_x = image2D.largeur;
343
344               } /* if ( next_SFB == -1 ) */
345
346             } /* il y a un autre motif */
```

```
347
348             } /* if ( last_x < image2D.largeur ) */
349
350           } /* while ( last_x < image2D.largeur ) */
351
352       } /* fin il y a un sous frame buffer dans cette ligne */
353
354     } /* for ( ligne */
355
356   } /* if ( image2D.n_plans == 3 ) */
357
358
359   /* Calcul de l'adresse des sous-scènes */
360   if ( n_subFB > 0 ) {
361     x[0] = adresse;
362     for ( i = 1; i <= n_subFB; i++ )
363       x[i] = x[i - 1] + ( taille_subFB[i - 1] / 4 ) +
364              hauteur_subFB[i - 1] - 1;
365   }
366
367
368   /* Ecriture de adresses des sous-scènes dans le résultat */
369
370   adresse = 1;   /* on saute la couleur de fond */
371
372   for ( ligne = 0; ligne < image2D.hauteur; ligne++ ) {
373
374     adresse+=2; /* on sait qu'on a 1 plan de profondeur 0 */
375
376     nMotifs = frame_buffer[adresse++];
377
378     for ( motif = 0; motif < nMotifs; motif++ ) {
379
380       adresse++; /* offset du motif */
381
382       nVoxels = frame_buffer[adresse++];
383
384       if ( nVoxels & 0x80000000 ) {
385
386          nVoxels = nVoxels & 0x7fffffff;
387
388         frame_buffer[adresse] = x[frame_buffer[adresse]] +
389           indices_debut_lignes_subFB[frame_buffer[adresse]]
390           [ligne - y[frame_buffer[adresse]]] + ligne -
```

```
391             y[frame_buffer[adresse]] - 1;
392
393           adresse += 2; /* on saute le modifieur de rendu */
394
395         }
396
397         for ( voxel = 0; voxel < nVoxels; voxel++ )
398           adresse++;
399
400       } /* for ( motif ... */
401
402     } /* for ( ligne */
403
404
405     /* Ouverture du fichier de sortie en écriture */
406     out = fopen(argv[1], "w");
407     if ( out == NULL ) {
408       fprintf(stderr, "Impossible d'ouvrir le fichier %s en écriture.\n",
409               argv[1]);
410       return 1;
411     }
412
413
414     /* Écriture de l'en-tête */
415     fprintf(out, "BKFBv1.0 %u %u ", image2D.largeur, image2D.hauteur);
416
417
418     /* Écriture du résultat */
419     for ( pixel = 0; pixel < adresse; pixel++ )
420       fprintf(out, "%c%c%c%c",
421               (char)(frame_buffer[pixel] >> 24),
422               (char)((frame_buffer[pixel] >> 16) & 0xff),
423               (char)((frame_buffer[pixel] >> 8) & 0xff),
424               (char)(frame_buffer[pixel] & 0xff));
425
426
427     /* Écriture des sous-scènes */
428     for ( i = 0; i < n_subFB; i++ )
429
430     for ( ligne = 0; ligne < hauteur_subFB[i]; ligne++ ) {
431
432       fprintf(out, "%c%c%c%c",
433               subFB[i][0], subFB[i][1], subFB[i][2], subFB[i][3]);
434
```

```
435       for ( j = indices_debut_lignes_subFB[i][ligne];
436             j < indices_debut_lignes_subFB[i][ligne + 1]; j++ )
437
438         fprintf(out, "%c%c%c%c",
439                 subFB[i][4 * j],
440                 subFB[i][4 * j + 1],
441                 subFB[i][4 * j + 2],
442                 subFB[i][4 * j + 3]);
443
444     } /* for ( ligne ... */
445
446   /* fermeture du fichier de sortie */
447   fclose(out);
448
449   /* libération de la mémoire */
450   if ( n_subFB > 0 ) {
451     for ( pixel = 0; pixel < n_subFB; pixel++ ) {
452       free(subFB[pixel]);
453       free(indices_debut_lignes_subFB[pixel]);
454     }
455     free(subFB);
456     free(indices_debut_lignes_subFB);
457     free(taille_subFB);
458     free(hauteur_subFB);
459     free(x);
460     free(y);
461     free(h);
462     free(l);
463   }
464   free(frame_buffer);
465   effaceImage(&image2D);
466   if ( argc > 3 )
467     effaceImage(&cache);
468
469   return 0;
470
471 }
472
473
474 /*****************************************************************/
475
476
477 /* Fonction de chargement d'une sous-scène */
478
```

```
479 int charge_subFB(char          *  filename,
480                  unsigned char ** pointer,
481                  unsigned long *  taille,
482                  unsigned long *  hauteur,
483                  unsigned long ** indices_debut_lignes) {
484
485   struct stat filestat;
486   off_t i;
487   FILE * fic;
488   int tmp;
489   unsigned long ligne, nPlans, plan, nMotifs, motif, nPixels, pixel;
490   unsigned long * frameBuffer;
491
492
493   /* Lecture de la taille du fichier */
494   if ( filename == NULL || pointer == NULL ) {
495     fprintf(stderr, "Mauvais paramètre\n);
496     return 0;
497   }
498
499   if ( stat(filename, &filestat) == -1 ) {
500     perror("stat");
501     return 0;
502   }
503
504   /* Allocation de la taille nécessaire */
505   *taille = filestat.st_size;
506   *pointer = (unsigned char *)malloc( (*taille) * sizeof(char));
507   if ( *pointer == NULL ) {
508     fprintf(stderr, "Erreur d'allocation de mémoire.\n);
509     return 0;
510   }
511
512   /* Ouverture du fichier */
513   fic = fopen(filename, "r");
514   if ( fic == NULL ) {
515     fprintf(stderr, "Impossible d'ouvrir le fichier %s.\n, filename);
516     return 0;
517   }
518
519   /* Lecture de l'en-tête */
520   if ( fgetc(fic) != 'B' ||
521        fgetc(fic) != 'K' ||
522        fgetc(fic) != 'F' ||
```

```
523         fgetc(fic) != 'B' ||
524         fgetc(fic) != 'v' ||
525         fgetc(fic) != '1' ||
526         fgetc(fic) != '.' ||
527         fgetc(fic) != '0' ||
528         fgetc(fic) != ' ' ) {
529       fprintf(stderr, "Le fichier %s n'est pas un fichier valide.\n,
530               filename);
531       return 0;
532     }
533
534     if ( fscanf(fic, "%d", &tmp)  != 1 ||
535          fscanf(fic, "%u", hauteur) != 1 ||
536          fgetc(fic) != ' ' ) {
537       fprintf(stderr, "Le fichier %s n'est pas un fichier valide.\n,
538               filename);
539       return 0;
540     }
541
542     /* Mémorisation des indices de début de chaque ligne */
543     *indices_debut_lignes =
544               (unsigned long *)malloc( (*hauteur + 1) *
545                                        sizeof(unsigned long) );
546     if ( *indices_debut_lignes == NULL ) {
547       fprintf(stderr, "Erreur d'allocation de mémoire.\n);
548       return 0;
549     }
550
551     *taille -= ftell(fic);
552
553     for ( i = 0; i < *taille; i++ )
554       (*pointer)[i] = (unsigned char)fgetc(fic);
555
556     frameBuffer = (unsigned long *)(*pointer);
557     (*indices_debut_lignes)[0] = 1;
558     i = 1; /* pour le decodage, on saute la couleur du fond */
559
560     for ( ligne = 0; ligne < *hauteur; ligne++ ) {
561       if ( i >= *taille ) {
562         fprintf(stderr,
563                 "Le fichier %s n'est pas un fichier valide.\n,
564                 filename);
565         return 0;
566       }
```

```
567       nPlans = recode(frameBuffer[i++]);
568       for ( plan = 0; plan < nPlans; plan++ ) {
569         if ( i + 1 >= *taille ) {
570           fprintf(stderr,
571                   "Le fichier %s n'est pas un fichier valide.\n,
572                   filename);
573           return 0;
574         }
575         i++; /* offset du plan */
576         nMotifs = recode(frameBuffer[i++]);
577         for ( motif = 0; motif< nMotifs; motif++ ) {
578           if ( i + 1 >= *taille ) {
579             fprintf(stderr,
580                     "Le fichier %s n'est pas un fichier valide.\n,
581                     filename);
582             return 0;
583           }
584           i++; /* offset du motif */
585           nPixels = recode(frameBuffer[i++]);
586           for ( pixel = 0; pixel < nPixels; pixel++ ) {
587             if ( i >= *taille ) {
588               fprintf(stderr,
589                       "Le fichier %s n'est pas un fichier valide.\n,
590                       filename);
591               return 0;
592             }
593             i++;
594           } /* for ( pixel */
595         } /* for ( motif ... */
596       } /* for ( plan ... */
597       (*indices_debut_lignes)[ligne + 1] = i;
598     } /* for ( ligne ... */
599
600     if ( i != ( *taille / 4 ) ) {
601       fprintf(stderr, "Le fichier %s n'est pas un fichier valide.\n,
602               filename);
603       return 0;
604     }
605
606     fclose(fic);
607
608     return 1;
609
610   }
```

```
611
612
613 /*****************************************************************/
614
615
616 /* Fonction permettant de modifier l'endianness d'un entier de 32 bits */
617 /* Cette fonction est utilisée par la fonction charge_subFB */
618
619 unsigned long recode(unsigned long val) {
620
621   return ( ( ( val & 255 ) << 24 ) |
622            ( ( ( val >> 8 ) & 255 ) << 16 ) |
623            ( ( ( val >> 16 ) & 255 ) << 8 ) |
624            ( ( val >> 24 ) & 255 ) );
625
626 }
```

B.2 Exemple d'utilisation du programme et résultats obtenus

Voici un exemple d'utilisation de ce programme. Il se compose, en figure B.1, de l'image représentant l'environnement 2D, du cache utilisé pour spécifier quels pixels contiennent les sous-scènes et quels pixels contiennent les pixels de la partie 2D, représenté figure B.2 et des deux scènes à insérer, représentées par les figures B.3 et B.4. Le résultat final est représenté figure B.5.

FIG. B.1 – Image utilisée pour la partie 2D

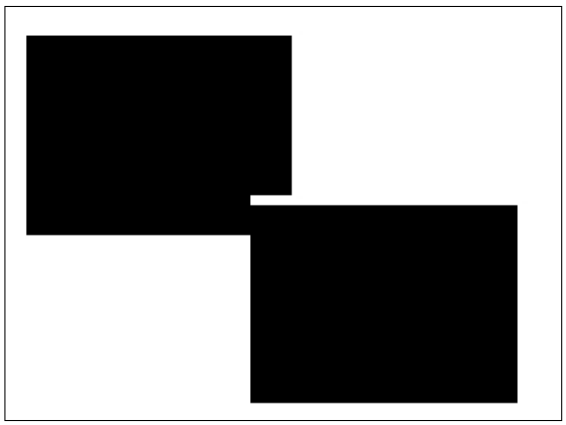


FIG. B.2 – Cache utilisé pour spécifier quels pixels contiennent les sous-scènes et quel pixels contiennent les pixels de la partie 2D

FIG. B.3 – 1ère sous-scène à insérer

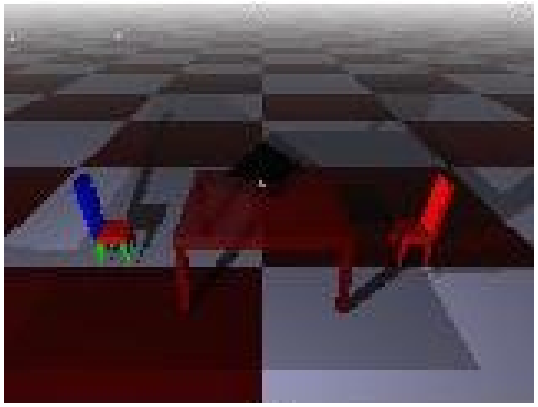


FIG. B.4 – Seconde sous-scène à insérer

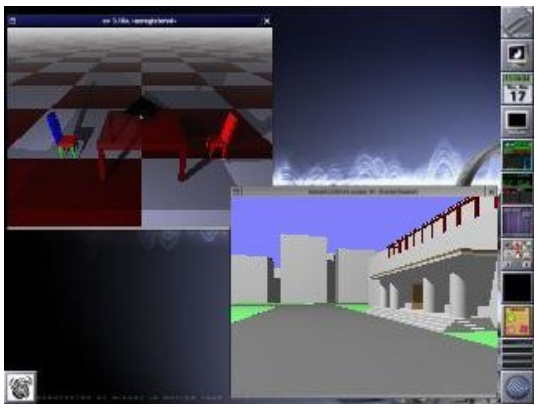


FIG. B.5 – Résultat de l'insertion des sous-scènes dans l'environnement 2D

Annexe C

Code source de l'implémentation de l'algorithme de génération des motifs exécuté par le GPU

Ce code source représente le code exécuté sur le GPU pour implémenter l'algorithme du depth peeling et le calcul des positions et des tailles des motifs. Ces deux algorithmes se déroulent en quatre passes : la première passe exécute, sur les vertex shaders, la projection des objets et mémorise la position à laquelle les objets doivent être projetés. Cet algorithme est implémenté par la fonction *RenderSceneVS*. Pendant cette passe, les pixel shaders implémentent l'algorithme du *depth peeling* (n°13), implémenté par la fonction *RenderScenePS*. La seconde passe, implémentée uniquement sur les pixel shaders, détecte le début de motifs (algorithme n°14). La fonction qui implémente cet algorithme est *detect_starts*. La troisième passe (implémentée par la fonction *detect_length* calcule la longueur des motifs (algorithme n°15). Les paramètres de cette fonction sont *position*, qui contient les coordonnées du pixel sur l'écran et *rshift*, qui correspond aux puissances successives de 2, utilisées pour déterminer à quel voxel comparer le voxel en cours. Enfin, la quatrième et dernière passe est implémentée par la fonction *end_detect*, qui correspond à l'algorithme de mémorisation de la longueur réelle d'un motif (n°16).

Sources de l'implémentation sur GPU

```
1  //--------------------------------------------------------------------------
2  // Variables globales
3  //--------------------------------------------------------------------------
4
5  // Variables utilisées pour le rendu des objets
6  float4 g_MaterialAmbientColor;      // Material's ambient color
7  float4 g_MaterialDiffuseColor;      // Material's diffuse color
8  int g_nNumLights;
9
10 float3 g_LightDir[3];                // Light's direction in world space
11 float4 g_LightDiffuse[3];            // Light's diffuse color
12 float4 g_LightAmbient;               // Light's ambient color
13
14 float    g_fTime;                    // App's time in seconds
15 float4x4 g_mWorld;                   // World matrix for object
16 float4x4 g_mWorldViewProjection;     // World * View * Projection matrix
17
18
19 // Variables utilisées pour nos algorithmes
20 float4 texture_size;  // taille des textures (en entrée et résultat)
21 float epsilon_z;      // distance entre deux plans successifs
22 int g_nRshift;        // décalage de l'algorithme de calcul de la
23                        // taille motifs (appelé shift dans l'algorithme 15)
24
25
26 //--------------------------------------------------------------------------
27 // Texture samplers
28 //--------------------------------------------------------------------------
29 texture g_MeshTexture;              // Color texture for mesh
30 sampler MeshTextureSampler =
31 sampler_state
32 {
33         Texture = <g_MeshTexture>;
34         MipFilter = LINEAR;
35         MinFilter = LINEAR;
36         MagFilter = LINEAR;
37 };
38
39
40 texture GP_buffer;
41 sampler GP_buffer_sampler = sampler_state
42 {
```

```
43          texture = <GP_buffer>;
44          addressu = clamp;
45          addressv = clamp;
46          minfilter = point;
47          magfilter = point;
48          mipfilter = none;
49  };
50
51
52  texture GP_buffer;
53  sampler GP_buffer_sampler = sampler_state
54  {
55          texture = <GP_buffer>;
56          addressu = clamp;
57          addressv = clamp;
58          minfilter = point;
59          magfilter = point;
60          mipfilter = none;
61  };
62
63
64  texture GD_buffer;
65  sampler GD_buffer_sampler = sampler_state
66  {
67          texture = <GD_buffer>;
68          addressu = clamp;
69          addressv = clamp;
70          minfilter = point;
71          magfilter = point;
72          mipfilter = none;
73  };
74
75
76  texture GN_buffer;
77  sampler GN_buffer_sampler = sampler_state
78  {
79          texture = <GD_buffer>;
80          addressu = clamp;
81          addressv = clamp;
82          minfilter = point;
83          magfilter = point;
84          mipfilter = none;
85  };
86
```

```
87
88  texture starts;
89  sampler starts_sampler = sampler_state
90  {
91          texture = <starts>;
92          addressu = wrap;
93          addressv = clamp;
94          minfilter = point;
95          magfilter = point;
96          mipfilter = none;
97  };
98
99
100 struct PS_OUTPUT3
101 {
102         float4 RGBColor2: COLOR0;  // Pixel color
103         float4 position : COLOR1;
104 };
105
106
107 struct PS_OUTPUT2
108 {
109         float4 RGBColor : COLOR0;  // Pixel color
110         float4 position : COLOR1;
111 };
112
113
114 //*************************************************************
115 // Rendu de la scène par les vertex shaders
116 //*************************************************************
117
118 VS_OUTPUT RenderSceneVS( float4 vPos            : POSITION,
119                          float3 vNormal         : NORMAL,
120                          float2 vTexCoord0      : TEXCOORD0,
121                          uniform int nNumLights,
122                          uniform bool bTexture)
123 {
124         VS_OUTPUT Output;
125         float3 vNormalWorldSpace;
126
127         // Transform the position from object space to
128         // homogeneous projection space
129         Output.Position = mul(vPos, g_mWorldViewProjection);
130
```

```
131         // Transform the normal from object space to world space
132         vNormalWorldSpace = normalize(mul(vNormal, (float3x3)g_mWorld));
133         // normal (world space)
134
135         // Compute simple directional lighting equation
136         float3 vTotalLightDiffuse = float3(0,0,0);
137         for(int i=0; i<nNumLights; i++ )
138          vTotalLightDiffuse += g_LightDiffuse[i] *
139                                abs(dot(vNormalWorldSpace, g_LightDir[i]));
140
141         Output.Diffuse.rgb = g_MaterialDiffuseColor * vTotalLightDiffuse +
142                              g_MaterialAmbientColor * g_LightAmbient;
143         Output.Diffuse.a = 1.0f;
144
145         // Just copy the texture coordinate through
146         if( bTexture )
147                 Output.TextureUV = vTexCoord0;
148         else
149                 Output.TextureUV = 0;
150
151         Output.position2 = Output.Position / Output.Position.w;
152
153         return Output;
154 }
155
156
157 //--------------------------------------------------------------------------
158 // Phase 2 du "Depth peeling" et mémorisation des coordonnées des voxels
159 //--------------------------------------------------------------------------
160
161 PS_OUTPUT2 RenderScenePS( PS_INPUT In )
162 {
163         PS_OUTPUT2 Output;
164         float2 uv = (In.uv + 0.5f) / texture_size.xy;
165         float4 old_pos = tex2D(old_G_buffer_sampler, uv);
166
167         if(old_pos.z + epsilon_z > In.position.z)
168                 clip(-1.0f);
169         Output.position = In.position.z;
170
171         // Chaque voxel est vert lorsqu'ils sont vus de l'extérieur de
172         // l'objet et rouge lorsqu'ils sont vus de l'intérieur
173         if(In.face <0)
174                 Output.RGBColor = In.Diffuse * float4(1,0,0,0);
```

```
175         else
176                 Output.RGBColor = In.Diffuse * float4(0,1,0,0);
177
178         return Output;
179 }
180
181
182 //*****************************************************************
183 // Rendu et mémorisation des coordonnées des voxels
184 //*****************************************************************
185
186 technique PeelLayer
187 {
188         pass P0
189         {
190                 VertexShader = compile vs_2_0 RenderSceneVS(1, false);
191                 PixelShader  = compile ps_3_0 RenderScenePS();
192                 ZEnable = true;
193                 ZFunc = Less;
194                 ZWriteEnable = true;
195                 CullMode = none;
196         }
197 }
198
199
200 //*****************************************************************
201 // Détection du début des motifs
202 //*****************************************************************
203
204 float4 detect_starts(float2 position : VPOS) : COLOR0
205 {
206         float4 output;
207         float2 uv = (position+0.5f) / texture_size.xy;
208         float2 texture_step = float2(texture_size.z,0.0f);
209         float2 left = uv - texture_step;
210         float currentz = tex2D(old_G_buffer_sampler, uv).z;
211         float leftz = tex2D(old_G_buffer_sampler, left).z;
212         float dz = abs(currentz - leftz);
213
214         if ( dz > epsilon_z || uv.x < texture_size.z )  //debut de motif
215         {
216                 output = 0.0f;
217         }
218         else
```

```
219         {
220                 output = (-1.0f);
221         }
222
223         return output;
224
225 }
226
227
228 //*****************************************************************
229 // Calcul de la longueur des motifs
230 //*****************************************************************
231
232 float4 detect_length(float2 position : VPOS, uniform int rshift) : COLOR0
233 {
234         float4 output;
235         float2 uv = (position+0.5f) / texture_size.xy;
236         float data = tex2D(starts_sampler,uv).r;
237
238         if( data >= 0.0f)
239         {
240                 output = data;
241         }
242         else
243         {
244                 float2 texture_step = float2(texture_size.z,0.0f);
245                 float2 right = uv + texture_step * rshift;
246                 float data2 = tex2D(starts_sampler,right).r;
247
248                 if(data2 < 0.0f)
249                 {
250                         output = -1.0f;
251                 }
252                 else
253                 {
254                         output = data2.x + rshift;
255                 }
256         }
257
258         return output;
259 }
260
261
262 //*****************************************************************
```

```
263 // Mise à jour de la longeur des motifs (1er voxel de chaque motif) +
264 // mémorisation des coordonnées x, y et z de chaque voxel dans la
265 // texture résultat
266 //*****************************************************************
267
268 half4 end_detect(float2 position : VPOS): COLOR0
269 {
270         half4 output;
271         float2 uv = (position+0.5f) / texture_size.xy;
272         float data = tex2D(starts_sampler,uv).r;
273
274         if( data > 0.0f)
275         {
276                 output.r = data;
277         }
278         else
279         {
280                 float2 texture_step = float2(texture_size.z,0.0f);
281                 float2 right = uv + texture_step;
282                 output.r = (tex2D(starts_sampler,right).r + 1.0f);
283         }
284         output.gba = tex2D(old_G_buffer_sampler,uv).xyz;
285
286         return output;
287 }
288
289
290 //*****************************************************************
291 // Calcul et reconstitution des motifs
292 //*****************************************************************
293
294 technique Detect2
295 {
296         pass p1
297         {
298                 PixelShader = compile ps_3_0 detect_starts();
299                 ColorWriteEnable = RED;
300         }
301         pass p2
302         {
303                 PixelShader = compile ps_3_0 detect_length(g_nRshift);
304                 ColorWriteEnable = RED;
305         }
306         pass p3
```

```
307         {
308                 PixelShader = compile ps_3_0 end_detect();
309                 ColorWriteEnable = RED|GREEN|BLUE|ALPHA;
310         }
311 }
```

Annexe D

Captures d'écran du logiciel Visual Elite

Ces captures d'écran (sauf les n°D.1 et D.2) sont issues de la documentation fournie par la société Summit. Les captures d'écran n° D.1 et D.2 sont des captures de nos premières utilisation du logiciel.

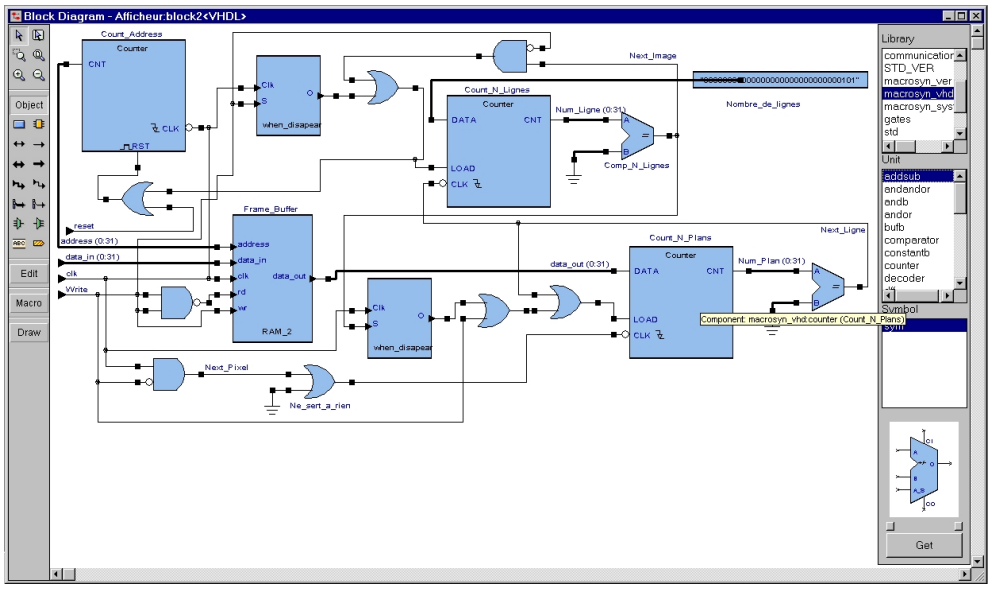


FIG. D.1 – Capture du logiciel Visual Elite montrant un premier essai de modélisation d'une architecture électronique

La figure D.1 montre un premier essai de modélisation d'une architecture électronique permettant de mémoriser des données puis de les décoder. Les données à décoder sont

reçues par le signal Data_in à gauche et écrite dans le bloc noté *Frame_Buffer*. Puis elles sont lues pour initialiser le bloc Count_N_Plans autant de fois qu'indiqué dans le registre Nombre_de_lignes, en haut à gauche. Lorsque ce compteur arrive à zéro, une nouvelle valeur est lue et le compteur de lignes est décrémenté. Cette modélisation ne correspond à rien de concret, elle a été réalisée dans le but de se familiariser avec l'outil logiciel Visual Elite. Elle correspond cependant à un début de modélisation de l'algorithme n°3 (*Reconstitution des vues*).

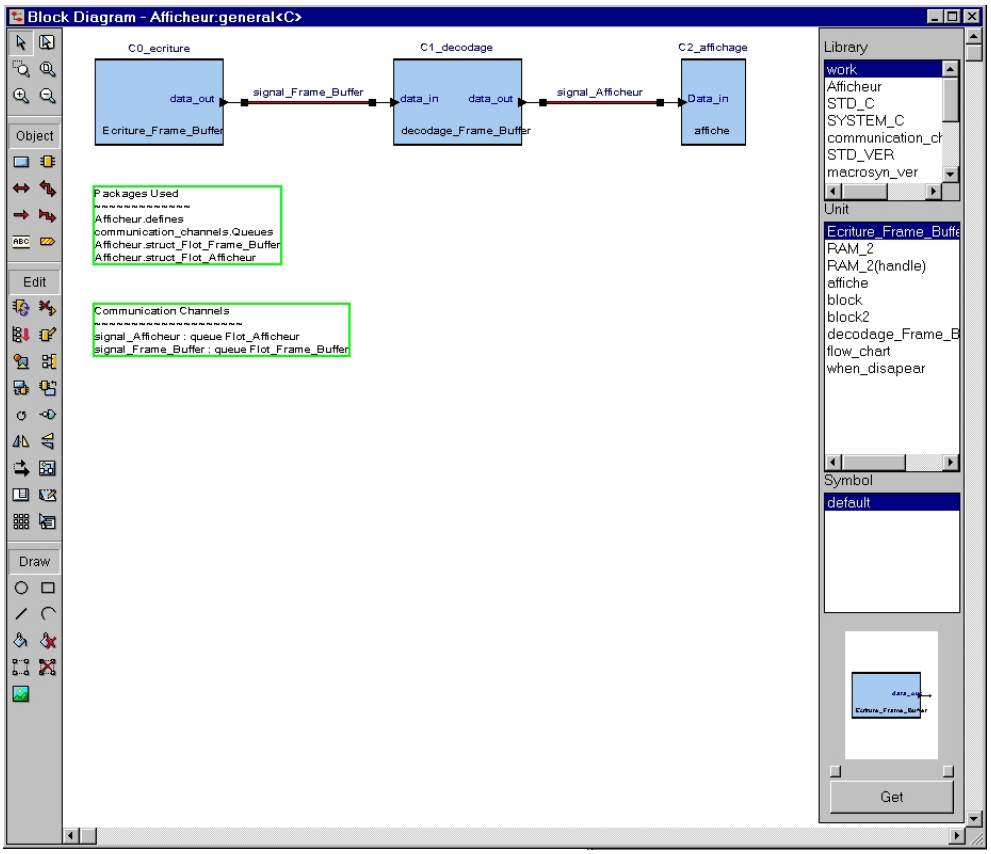


FIG. D.2 – Capture du logiciel Visual Elite montrant une modélisation de très haut niveau de notre chaîne de traitements

La figure D.2 représente une modélisation de haut niveau de notre chaîne de traitements. Le premier bloc, noté *Ecriture_Frame_Buffer*, modélise l'implémentation des algorithmes n°1, 2, 4, 5, 6, 7, 10, 11 et 12, c'est-à-dire tous les algorithmes de codage des

données 3D que nous avons décrits dans cette thèse. Les données générées sont ensuite envoyées dans le second bloc, appelé *C1_decodage*, modélisant l'implémentation des algorithmes n°3, 8 et 9, qui décompose ces données et regénèrent les vues qui sont ensuite envoyées au troisième bloc, nommé *C2_affichage*, qui symbolise le dispositif physique d'affichage.

Les captures d'écran suivantes sont issues de la documentation en ligne du logiciel Visual Elite[18], fournie par la société Summit[14].

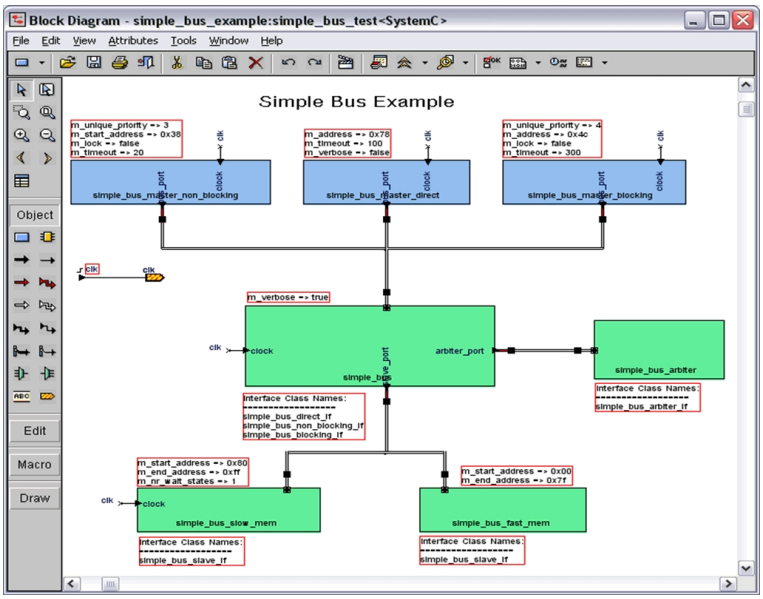


FIG. D.3 – Capture du logiciel Visual Elite

La figure D.3 montre un exemple de modélisation d'une architecture comprenant plusieurs blocs communiquant par l'intermédaire de plusieurs bus de données.

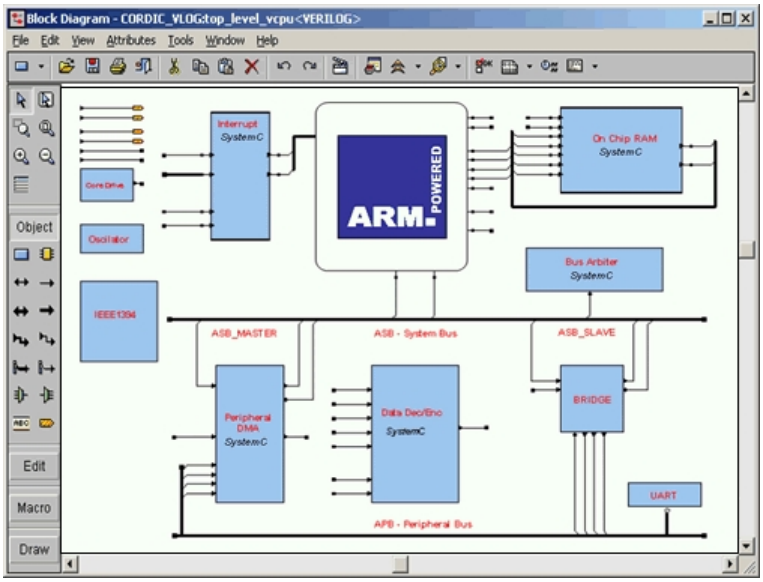


FIG. D.4 – Capture du logiciel Visual Elite

La figure D.4 représente la modélisation d'un système composé d'un micro-contrôleur ARM, de différents périphériques (entre autres, RAM, interface IEEE 1364, contrôleur DMA), dont la plupart ont été décrits en SystemC, et d'un bus de données.

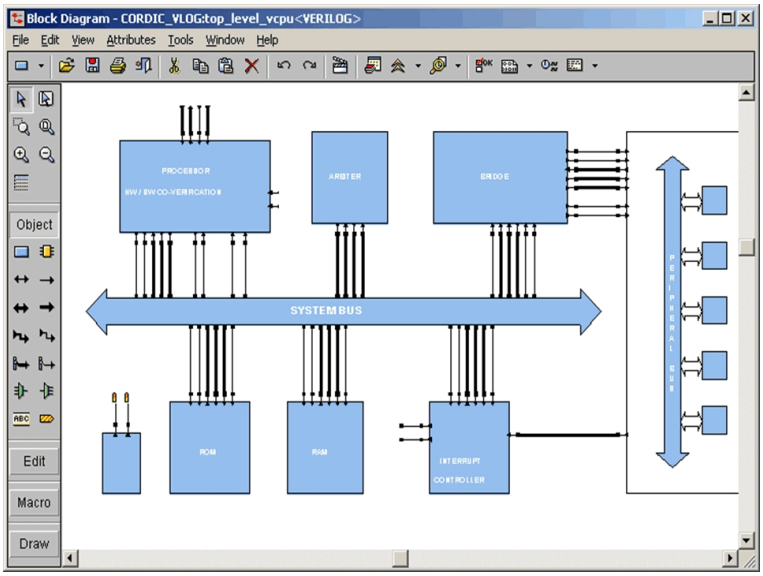


FIG. D.5 – Capture du logiciel Visual Elite

La figure D.5 montre un exemple de modélisation d'une architecture permettant à ses différents blocs de communiquer entre eux par l'intermédiaire d'un bus de données unique.

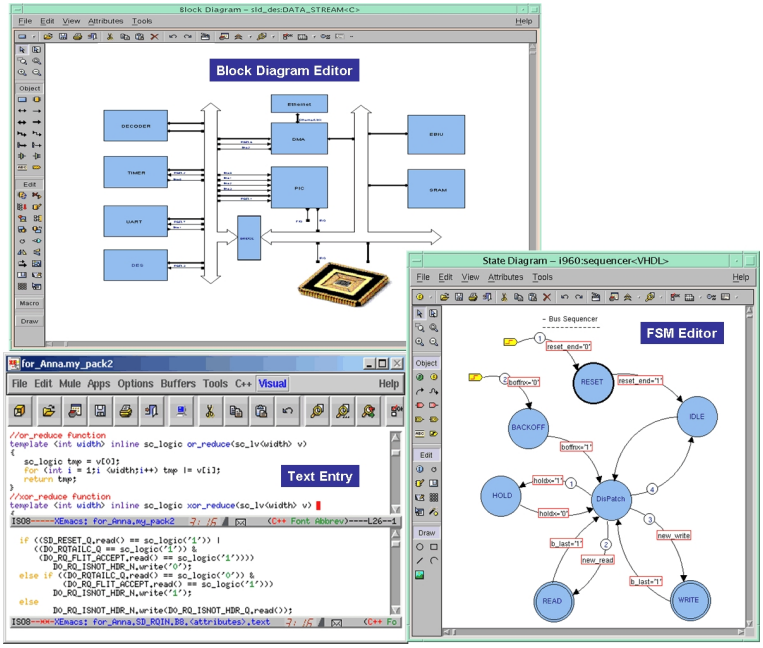


FIG. D.6 – Capture du logiciel Visual Elite

La figure D.6 montre différents moyens de définir des blocs :

- En haut à gauche, vous pouvez voir l'éditeur de diagramme par blocs, permettant de définir des blocs et les signaux de communications entre eux. Cet éditeur permet de définir l'architecture globale à modéliser et d'affiner la modélisation à l'intérieur d'un bloc.
- En bas à gauche, se trouve une fenêtre d'édition de code source, permettant d'écrire des fonctions modélisant le comportement d'un bloc. Ces fonctions sont écrites en C++ et sont ensuite insérées dans un code en SystemC, généré par le programme en fonction des données de l'éditeur de diagramme par blocs.
- À droite, se trouve l'éditeur de machines à états, permettant de modéliser le comportement d'un bloc avec une machine à état. Cette machine à états peut ensuite

être convertie en SystemC pour être simulée, ou être exportée en VHDL pour être utilisée pour programmer un FPGA.

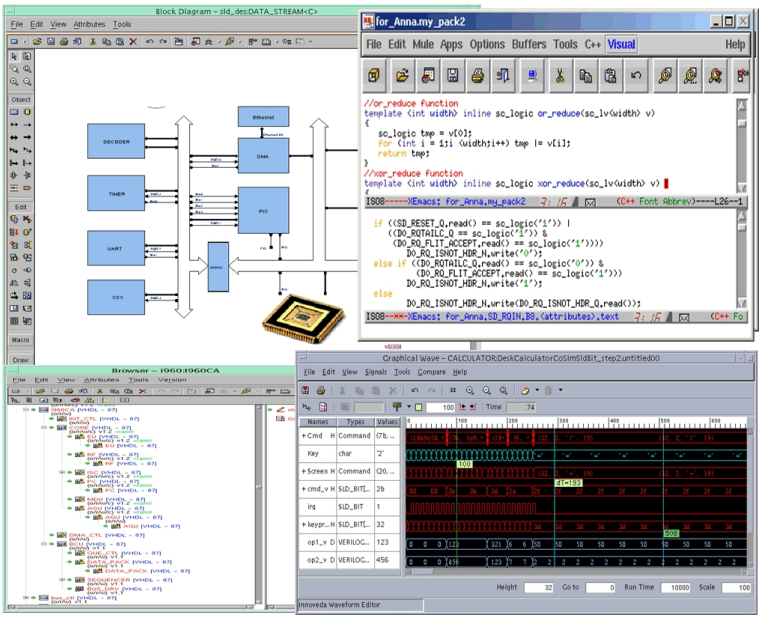


FIG. D.7 – Capture du logiciel Visual Elite

La figure D.7 montre un exemple de simulation (fenêtre en bas à droite) d'une architecture. Les autres fenêtres montrent l'architecture définie sous la forme d'un diagramme de blocs (fenêtre en haut à gauche), la liste des modules (descriptions de blocs créés par l'utilisateur ou fournis par la société Summit avec le logiciel) et la description d'un bloc en C++ (fenêtre en haut à droite).

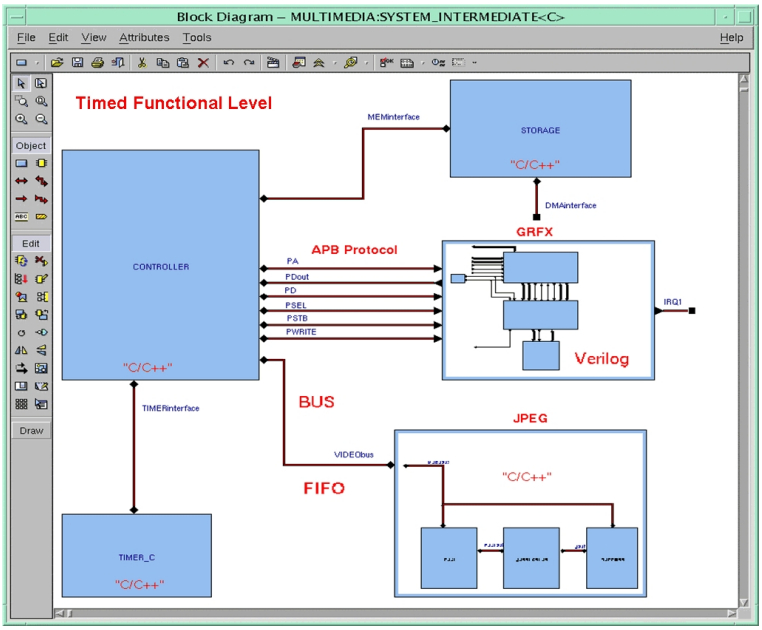


FIG. D.8 – Capture du logiciel Visual Elite

La figure D.8 montre un exemple de modélisation réalisée en utilisant différents langages de description de comportements des blocs. Dans cet exemple, 4 blocs ont été modélisés grâce à des fonctions en C ou C++ et un en utilisant le langage Verilog.

Annexe E

Captures d'écran de l'exécution de l'algorithme du depth peeling

Dans cet exemple, un éclairage ambiant a été utilisé, afin que tous les voxels soient éclairés. Les voxels vus de l'extérieur de l'objet sont colorés en vert, alors que les voxels vus de l'intérieur de l'objet sont colorés en rouge.

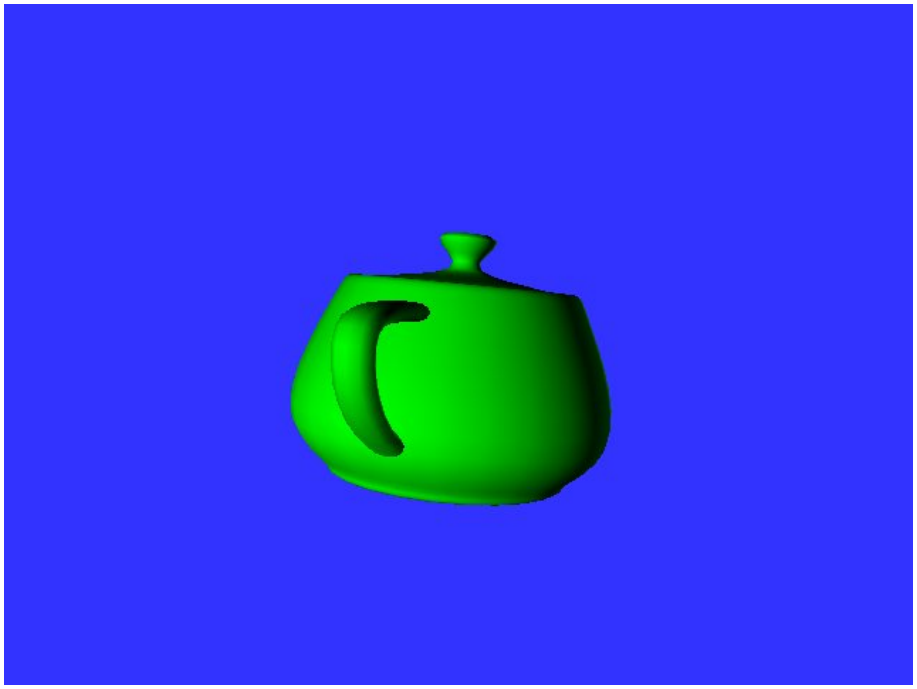


FIG. E.1 – Exemple d'exécution de l'algorithme du depth peeling, passe 1

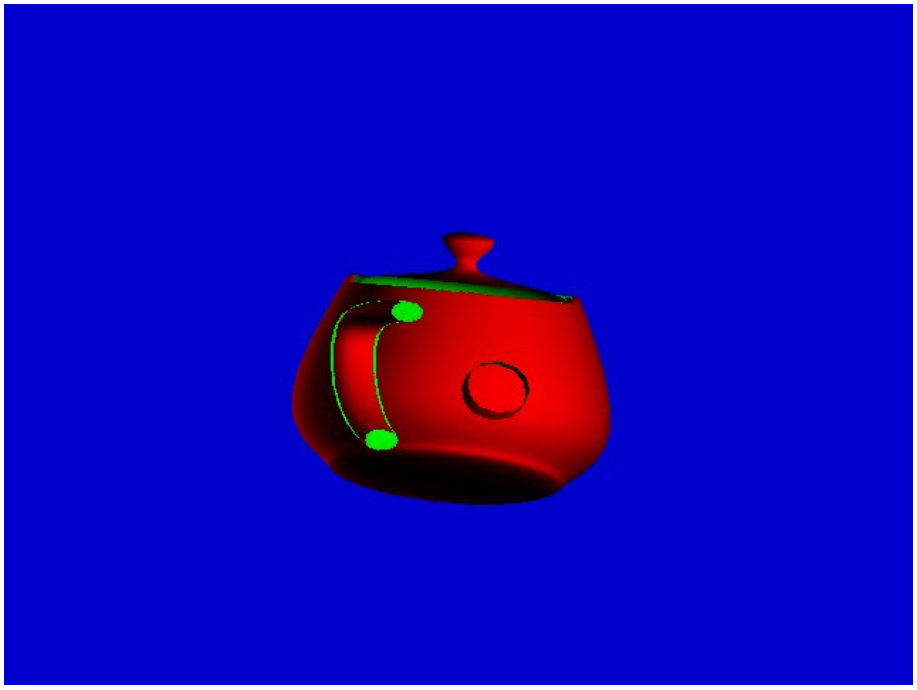


FIG. E.2 – Exemple d'exécution de l'algorithme du depth peeling, passe 2

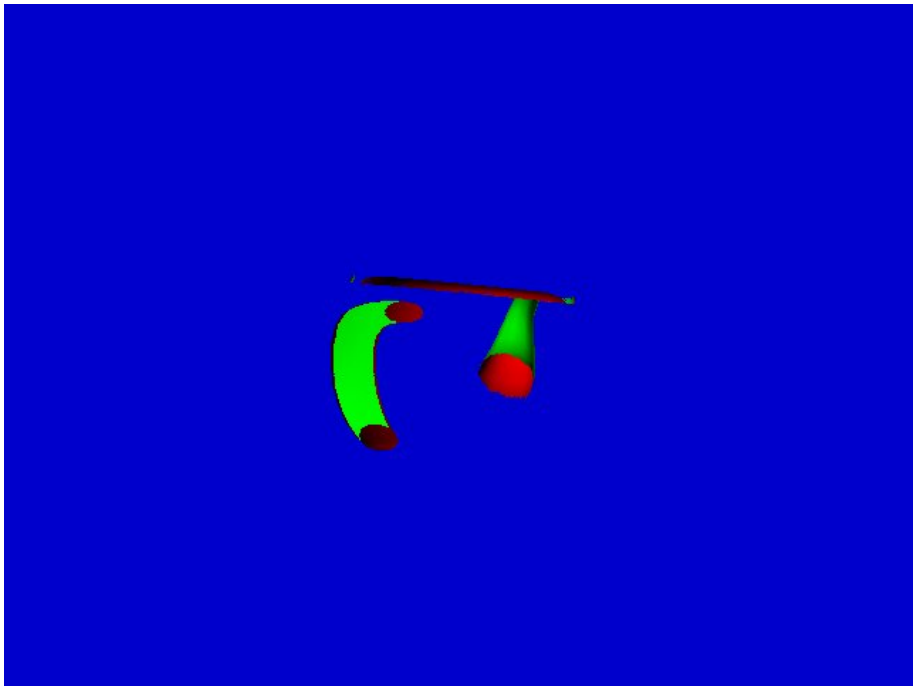


FIG. E.3 – Exemple d'exécution de l'algorithme du depth peeling, passe 3

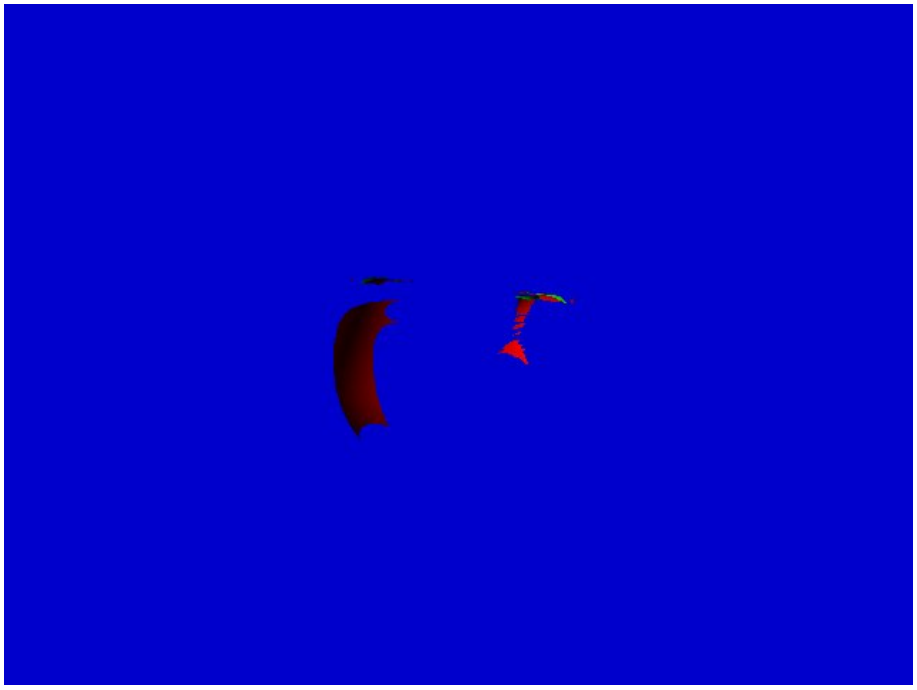


FIG. E.4 – Exemple d'exécution de l'algorithme du depth peeling, passe 4

FIG. E.5 – Exemple d'exécution de l'algorithme du depth peeling, passe 5